Minimizing State Space Partitions Using Decision Trees

Andreas Holck Høeg-Petersen
Aalborg University
Denmark

Kim Guldstrand Larsen
Aalborg University
Denmark

Peter Gjøl Jensen
Aalborg University
Denmark

Andrzej Wąsowski
IT University of Copenhagen
Denmark

***Abstract*— Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.**

I. 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 [1] 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 [2]. 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 [2]. 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 [2]. 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 [3], [4]. 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 policy 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.

II. Preliminaries

Definition II.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* $\nu \cap \nu' = \emptyset$.

do we assume that the state space $\mathcal{S}$ is rectangular?

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 $s = (s_1, \dots, s_K) \in \nu$ it holds that $s_i^{\min} < s_i \leq s_i^{\max}$ for $i = 1, \dots, K$. 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 $i = 1, \dots, K$.

Definition II.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. $\mathcal{T}(\nu) = \{\mathcal{T}(s) \mid s \in \nu\}$.

We denote the path $\lambda(\ell)$ and define it as an ordered list of tuples of the form (η_j, b) where η_j is the jth node on the path (η_0 will always be the root node) and b is a binary value indicating wether the path continues with the left child ($b = 1$) or right child ($b = 0$).[inline]Not sure if it makes sense or is necessary to state which value of b means what. The reasoning for my choice is that $b = 1$ indicates 'true' and it is when $\rho_{\eta_j}(s)$ is true, that we choose the left path. The path can then be said to define a region where the corner points $s^{\min}$ and $s^{\max}$ are given by compiling the bounds on each dimension $i = 1, \dots, K$ given by the predicate function $\rho_{\eta_j} = i_{\eta_j} \leq c_{\eta_j}$ for each $\eta_j \in \lambda(\ell)$ into points. The coordinates for each point is given by

$$s_i^{\min} = \max(\{\, c_\eta \mid (\eta, b) \in \lambda(\ell),\ i_\eta = i,\ b = 1 \,\})$$
$$s_i^{\max} = \min(\{\, c_\eta \mid (\eta, b) \in \lambda(\ell),\ i_\eta = i,\ b = 0 \,\})$$

[inline]I struggled a lot with coming up with a good way of writing this definition. Do let me know if it works (and if it is even necessary to describe how a region is constructed from a leaf node).

for all $i = 1, \dots, K$. We write ν_ℓ to denote the region associated with the leaf node ℓ.

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 DefinitionII.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 $\mathcal{T}$.

III. MaxPartitions algorithm

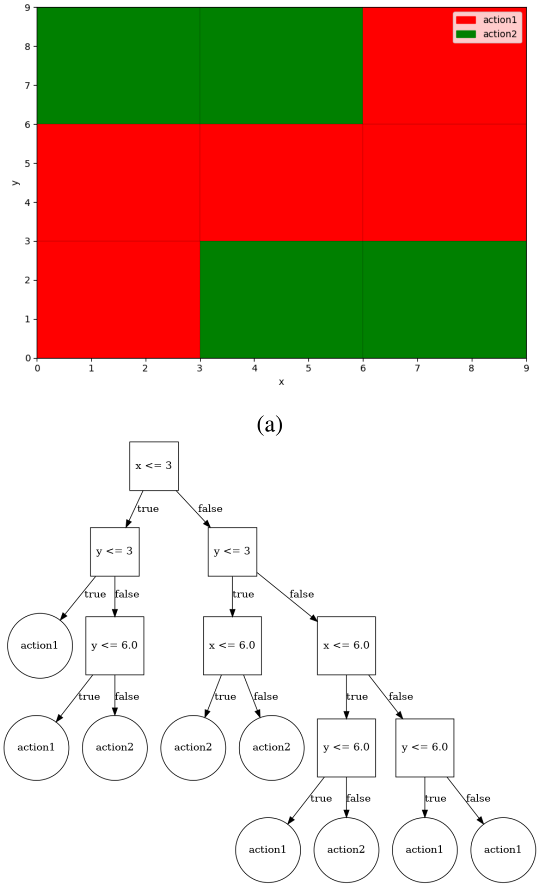
Since state space discretization for Reinforcement Learning is usually done *before* any learning takes place, it tends to be conservative. For this reason, discretization is likely to create adjacent discrete states that are mapped to the same optimal action. The question we would then like to answer 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.

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 [5], 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 post-processes 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.



(a)

(b)

Fig. 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.

A. 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, 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 III.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' does not have singular mapping in $\mathcal{T}$*
2) *ν' intersects with one or more regions already in $\mathcal{T}_{track}$*
3) *ν' intersects with a region ν_o in the original partitioning, such that the disjunction $\nu' \cap \nu_o$ 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 …[inline]Make an example to show.

How do we determine this expansion? Let $(p^{\min}, p^{\max})$ define region in the original partitioning (or potentially the remains of a region previously cut in two). 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 III.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 $\nu = (s^{p^{\min}}_{\mathcal{T}}, s^{p^{\max}}_{\mathcal{T}})$ does not violate any of the expansion rules in Definition III.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 $\nu = (s^{p^{\min}}_{\mathcal{T}}, s^{p^{\max}}_{\mathcal{T}})$. 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 III.2.

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 region bounds of unexplored region
    $\Delta_p \leftarrow p^{\max} - p^{\min}$

    while not all dimensions have been exhausted do
        $d \leftarrow$ randomly select unexhausted dimension
        $\Delta_{p_d} \leftarrow \Delta_{p_d} + 1$
        $\nu' \leftarrow (s_{\mathcal{T}}^{p^{\min}}, s_{\mathcal{T}}^{p^{\min}+\Delta_p})$

        if $\nu'$ violates Rule 1 or 2 (Definition III.1) then
            $\Delta_{p_d} \leftarrow \Delta_{p_d} - 1$
            mark $d$ as exhausted
        else if $\nu'$ violates Rule 3 (Definition III.1) then
            attempt $healing$ in dimension $d$

    $\nu' \leftarrow (s_{\mathcal{T}}^{p^{\min}}, s_{\mathcal{T}}^{p^{\min}+\Delta_p})$
    PUT($\mathcal{T}_{track}$, $\nu'$)
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{\nu'\}$
return $\mathcal{R}$
```

B. Analyzing the algorithm

In the following we provide an upper bound of the running time of MAXPARTITIONS and a proof of correctness.

1) Running time:

2) Proof of correctness:

C. 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.

Therefore, we suggest a brute-force algorithm that tries to separate the regions as cleanly as possible. Let $\mathbf{R}$ be a list of regions and let a_ν be the action associated with the region $\nu = (s^{\min}, s^{\min})$. In the following, we 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 $\nu_{\max,i}$.

We iteratively create a branch node that splits $\mathbf{R}$ into two, $\mathbf{R}_{low}$ and $\mathbf{R}_{high}$, based on a predicate function $\rho(x) = x_i \leq c$ with $c \in \mathbb{R}$ so that $\mathbf{R}_{low} = \{\nu \in \mathbf{R} \mid \rho(\nu_{\min}) \text{ is True}\}$ and $\mathbf{R}_{high} = \{\nu \in \mathbf{R} \mid \rho(\nu_{\max}) \text{ is False}\}$. When the list only contains a single element ν, we create a leaf node with action a_ν and return.

The question is how to determine $\rho(x)$, more specifically which dimension i to predicate on and at which value c. Ideally, we want to split $\mathbf{R}$ in two equally sized subsets and in a way that no single region would have to occur in both, ie. we would like $\mathbf{R}_{low} \cap \mathbf{R}_{high} = \emptyset$. For this we define an impurity measure $I(\mathbf{R}_{low}, \mathbf{R}_{high})$ that penalises the difference in size between $\mathbf{R}_{low}$ and $\mathbf{R}_{high}$ and the size of the intersection between the two. Let $abs(a)$ be the absolute value of a and let $|b|$ denote the size of a set b, then

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

Our brute-force way of finding the predicate that minimizes I is to iterate over the dimensions in $\mathcal{S}$ and for each dimension i we sort the regions according to their upper bound. Let $\mathbf{R}_i = \{\nu^1, \nu^2, \ldots, \nu^n\}$ be the list sorted according to the i th dimension so that for all $\nu^j, \nu^{j+1} \in \mathbf{R}_i$ it holds 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 size of $\mathbf{R}_{low} \cap \mathbf{R}_{high}$ 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, since these regions will appear both in $\mathbf{R}_{low}$ (because then, by definition, $\rho(x) = x_i \leq c$ will be true for $x_i = \nu^{j+m}_{\min,i}$ and $c = \nu^j_{\max,i}$) and in $\mathbf{R}_{high}$ (because our sorting ensures that for all ν^j, ν^{j+m} it holds that $\nu^j_{\max,i} \leq \nu^{j+m}_{\max,i}$).

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

$$I(\mathbf{R}_{low}, \mathbf{R}_{high}) = abs(j-(n-j)) + \sum_{m=1}^{n} \mathbb{1}(\rho(\nu^{j+m}_{\min})), \quad \text{for all } \nu^j \in \mathbf{R}_i$$

where $\mathbb{1}$ is the indicator function, $\mathbf{R}_i$ is the list of regions sorted according to upper bounds in dimension i and $\mathbf{R}_{low}$ and $\mathbf{R}_{high}$ are the subsets resulting from splitting on the predicate function $\rho(x) = x_i \leq c$ with $c = \nu^j_{\max,i}$ so that $\mathbf{R}_{low} \subsetneq \mathbf{R}$, $\mathbf{R}_{high} \subsetneq \mathbf{R}$ and $\mathbf{R} \supseteq \mathbf{R}_{low} \cup \mathbf{R}_{high}$.

Finding the best split, ie. the one that minimizes the impurity, is a $O(Kn^2)$ operation, as it requires a nested loop through all the regions for each of the Kdimensions (the nested loop being the final summation term over $m =$

$1, 2, \ldots, n - j$ for all $j = 1, 2, \ldots, n - 1$). In this work, we have not attempted to find a faster implementation as we found that the size of $\mathbf{R}$ obtained by using our MAXPARTITIONS algorithm did not cause performance issues.

IV. EXPERIMENTS

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

REFERENCES

[1] A. David, P. G. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist, "Uppaal stratego," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings* (C. Baier and C. Tinelli, eds.), vol. 9035 of *Lecture Notes in Computer Science*, pp. 206–211, Springer, 2015.

[2] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada* (S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 2499–2509, 2018.

[3] P. Ashok, M. Jackermeier, P. Jagtap, J. Křetínský, M. Weininger, and M. 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), Association for Computing Machinery, 2020.

[4] P. Ashok, M. Jackermeier, J. Kretínský, C. Weinhuber, M. Weininger, and M. Yadav, "dtcontrol 2.0: Explainable strategy representation via decision tree learning steered by experts," in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II* (J. F. Groote and K. G. Larsen, eds.), vol. 12652 of *Lecture Notes in Computer Science*, pp. 326–345, Springer, 2021.

[5] M. Jaeger, P. G. Jensen, K. Guldstrand Larsen, A. Legay, S. Sedwards, and J. H. Taankvist, "Teaching stratego to play ball: Optimal synthesis for continuous space MDPs," in *Automated Technology for Verification and Analysis* (Y.-F. Chen, C.-H. Cheng, and J. Esparza, eds.), (Cham), pp. 81–97, Springer International Publishing, 2019.

APPENDIX

The goal is to take all the leafs of a Q-forest and from these build a single binary decision tree, that through evaluation of a state arrives at a leaf node indicating the best action to take. The general idea of the algorithm is to repeatedly take the next leaf L_i in $\mathcal{L}$, which we (because of our sorting) know to have the best Q-value, and insert its action into the decision tree we are building so that it respects S_i. This means, that the internal nodes of the tree will still be checks on variables in $\mathcal{V}$.

A. Creating the root

We initialise the tree from the lowest valued leaf in $\mathcal{L}$ which we will denote L_0. The operation of inserting L_0 into the at this point empty tree requires special attention, since we have to create split nodes for all the variable bounds $(l_{0,i}, u_{0,i})$ in S_0. We denote this operation `MakeRoot` and its pseudocode is given in Algorithm 2.

The operation iterates through all tuples $(u_{0,i}, l_{0,i})$ in S_0 and whenever it encounters a bound that is different from the limit (which we here assume to be infinity) it has to insert a new node into the tree. This is done via the `MakeNewNode` procedure, which we define as a helper function to avoid repetitions in the algorithm. This function takes the variable V_i that the encountered bound relates to, the value of the bound and the action of the leaf, we are inserting. It then creates a new internal node, that represents the partition according to the inequality $V_i \leq b$ where $b \in (u_{0,i}, l_{0,i})$.

Now, `MakeNewNode` also takes two additional arguments, one boolean to indicate if the bound is an upper bound or not and the current parent node (which might be `None`). If

TABLE I: Comparing MaxPartitions and VIPER for pure shields and shielded oracles.

Model	Pure shield			Shielded oracle		
	Input	MaxPartitions	VIPER	Input	MaxPartitions	VIPER
Bouncing ball	xxx	xxx	xxx	xxx	xxx	xxx
Random walk	xxx	xxx	xxx	xxx	xxx	xxx
Cartpole	xxx	xxx	xxx	xxx	xxx	xxx
Cruise	xxx	xxx	xxx	xxx	xxx	xxx
DCDC boost control	xxx	xxx	xxx	xxx	xxx	xxx

Algorithm 2 MakeRoot

```
procedure MakeNewNode($var, bound, action, isHigh, prevNode$)
    $node \leftarrow$ Node($var, bound$)
    if isHigh then
        $node.high \leftarrow$ Leaf($action, \infty$, State($\cdot$))
    else
        $node.low \leftarrow$ Leaf($action, \infty$, State($\cdot$))
    if $prevNode$ is not None then
        if $prevNode.low$ is None then
            $prevNode.low \leftarrow node$
        else
            $prevNode.high \leftarrow node$
    return $node$

function MakeRoot(Leaf($a_0, q_0, S_0$))
    $rootNode \leftarrow$ None
    $prevNode \leftarrow$ None
    for $(l_{0,i}, u_{0,i})$ in $S_0$ do
        if $u_{0,i} < \infty$ then
            $prevNode \leftarrow$ MakeNewNode($V_i, u_{0,i}, a_0$, True, $prevNode$)
            if $rootNode$ is None then
                $rootNode \leftarrow prevNode$
        if $l_{0,i} > -\infty$ then
            $prevNode \leftarrow$ MakeNewNode($V_i, l_{0,i}, a_0$, False, $prevNode$)
            if $rootNode$ is None then
                $rootNode \leftarrow prevNode$
    if $prevNode.low$ is None then
        $prevNode.low \leftarrow$ Leaf($a_0, q_0, S_0$)
    else
        $prevNode.high \leftarrow$ Leaf($a_0, q_0, S_0$)
    return $rootNode$
```

the bound is an upper bound, then we know that $Q(a_0) = q_0$ only holds if $V_i \leq u_{0,i}$. This means, that L_0 should be inserted somewhere in the left ('low') subtree of the newly created node, while the right ('high') subtree should for now just be set to some random leaf with a very poor Q-score (we will find better values for these subtrees when we process the remaining leaf nodes of $\mathcal{L}$).

Lastly, if the current parent node is not None, then we know it has one subtree that is set (like above) and one that itself is None as it is reserved for our newly created node. We therefore set the undefined child of the parent node to our new node and returns this new node, which is then marked as being the new parent node. If this our first new, we also mark it as our root node.

When we have processed all the bounds in S_0, we insert L_0 at the 'free' spot in the parent node and return the root node. With this approach, we end up with an initial tree that is very shallow and basically only has one interesting leaf at the maximum depth of the tree.

B. Inserting the leaf nodes

To insert the remaining leaf nodes we need a couple of extra helper functions. The task is to identify or construct all the paths in the tree under construction that leads to leaf nodes where the action of the leaf that are being inserted is to be preferred.

When we insert $L_i = (a_i, q_i, S_i)$, we will always either encounter an internal branch node that defines a split on a variable and has two subtrees, or we encounter a leaf node storing an action, a Q-value and a state partition. We therefore define a general Put function (Algorithm 3), that takes a root node and a leaf triplet to be inserted and decides what to do based on the type of the root node (either a branch node or a leaf node).

Algorithm 3 Build decision tree from leaf nodes of Q-tree
.

```
function Put($root$, Leaf($a, q, S$) )
    if $root$ is Node then
        return PutAtBranchNode($root$, Leaf($a, q, S$))
    else                                   ▷ $root$ is a Leaf
        return PutAtLeafNode($root$, Leaf($a, q, S$))
```

We will first deal with situation where we encounter an internal node, N_j, which splits on variable V_k at bound b. We now need to check on what side of this split S_i falls (it might be both). So we test on both $u_{i,k} > b$ and $l_{i,k} < b$. If the first check is true, then we know that S_i defines an area for V_k that can be larger than b, why we have to visit the right ('high') subtree of N_j. Likewise for the latter test, only then we have to continue our insertion in the left ('low') subtree. Note that both tests can be true.

We do, however, need to keep track of the implicit limitations we put on S_i as we go along. When continuing our insertion of L_i in the subtree of N_j defined by $V_k > b$, then we should reflect in S_i that now V_k has a lower bound b, that is, we should set $l_{i,k} = b$. We do this in the algorithm through an implicit helper function `SetLower`$(state, var, bound)$ (and likewise `SetUpper` for updating the upper bound). The pseudocode for this is given in Algorithm 4.

Algorithm 4 PUT Q-leaf into an internal node

```
function PutAtBranchNode(Node($V_k, b, low, high$), Leaf($a_i, q_i, S_i$) )
    if $l_{i,k} < b$ then
        $S_i' \leftarrow$ SetUpper($S_i, V_k, b$)
        $low \leftarrow$ Put($low$, Leaf($a_i, q_i, S_i'$))
    if $u_{i,k} > b$ then
        $S_i' \leftarrow$ SetLower($S_i, V_k, b$)
        $high \leftarrow$ Put($high$, Leaf($a_i, q_i, S_i'$))
    return Node($V_k, b, low, high$)
```

The second case is when we encounter a leaf node during the insertion operation. We denote this node as L_t to indicate that it is a leaf already present in the tree under construction. First, we check if the Q-value of L_t is better than that of L_i, in which case we do nothing and abort the insert operation. If q_i on the other hand is the better option, then we need to insert L_i but in a way that respects S_i.

It is guaranteed at this stage, that S_t contains S_i, that is $u_{t,j} \geq u_{i,j}$ and $l_{t,j} \leq l_{i,j}$ for all $j = 1, 2, \ldots, k$. But this also means that in the cases where the bounds on S_t are strictly larger or smaller than those of S_i then we need to insert a new internal node to ensure this partition before we can insert L_i. In other words, if $u_{t,j} > u_{i,j}$, then we need to create a branch node that splits on V_j at bound $b = u_{i,j}$ and whose right ('high') subtree is the original leaf L_t but with an updated state S_t where $l_{t,j} = u_{i,j}$. The left ('low') subtree should also, for a start, be set to L_t but then we continue the insert operation on this side, either creating more branch nodes or eventually inserting a_i and q_i in place of a_t and q_t.

The pseudocode for the function is given in Algorithm 5.

Algorithm 5 Put Q-leaf into a leaf node

```
procedure Split($action, q, var, bound, state$)
    $highState \leftarrow$ SetLower($state, var, bound$)
    $lowState \leftarrow$ SetUpper($state, var, bound$)
    $high \leftarrow$ Leaf($action, q, highState$)
    $low \leftarrow$ Leaf($action, q, lowState$)
    return Node($var, bound, low, high$)

function PutAtLeafNode(Leaf($a_t, q_t, S_t$), Leaf($a_i, q_i, S_i$))
    if $q_t \leq q_i$ then
        return Leaf($a_t, q_t, S_t$)

    for $(l_{t,j}, u_{t,j})$ in $S_t$ do
        if $l_{t,j} < l_{i,j}$ then
            $newNode \leftarrow$ Split($a_t, q_t, V_j, l_{i,j}$)
            return Put($newNode$, Leaf($a_i, q_i, S_i$))

        else if $u_{t,j} > u_{i,j}$ then
            $newNode \leftarrow$ Split($a_t, q_t, V_j, u_{i,j}$)
            return Put($newNode$, Leaf($a_i, q_i, S_i$))

    return Leaf($a_i, q_i, S_i$)
```