Minimizing decision tree representation of controller strategy

Andreas Holck Høeg-Petersen[1], Kim Guldstrand Larsen[1], Peter Gjøl Jensen[1], Andrzej Wasowsk[2]

[1] Aalborg University, Denmark
[2] 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.

1 Introduction

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.

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.

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.

2 Preliminaries

We start by defining some relevant concepts.

Definition 1 (Partitionings). *A partitioning $\mathcal{A}$ of the state space $\mathcal{S} \in \mathbb{R}^K$ is a set of regions ν that divides $\mathcal{S}$ such that $\bigcup_{\nu \in \mathcal{A}} \nu = \mathcal{S}$ and for any two regions $\nu, \nu' \in \mathcal{A}$ where $\nu \neq \nu'$ it holds that $\nu \cap \nu' = \emptyset$.*

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 (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 ($b = 0$). 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 $i = 1, \ldots, K$ 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

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.

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

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

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).

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 Definition1. 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 $\mathcal{T}$.

3 MaxPartitions algorithm

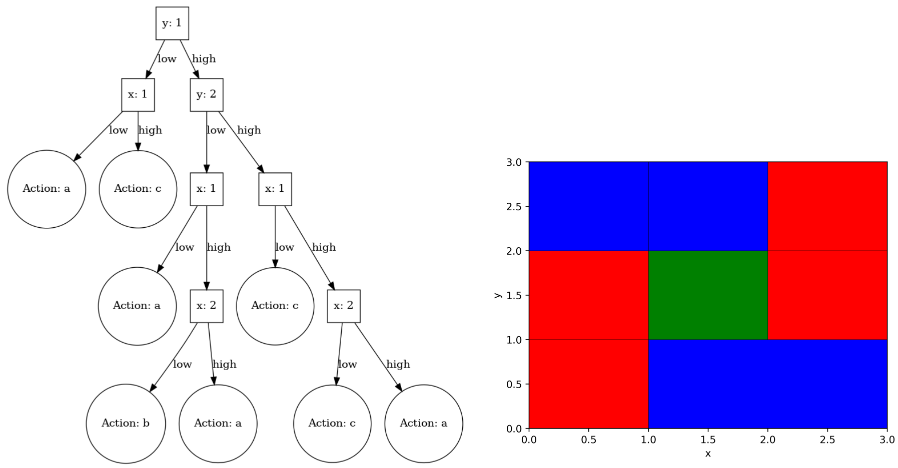
Since state space discretization for Reinforcement Learning is usually done *before* any learning takes place, it tends to be conservative. For this reason, discretization is likely to create adjacent discrete states that are mapped to the same optimal action. The question we would then like to answer 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, we can consider the toy strategy from Example **??**. In Figure 1 the strategy is represented as a decision tree (1a) by omitting the specification of cost values of each action and only preserving the optimal action for each discrete state. On the right (1b) is a 2D visualization of the induced partitioning of the state space. The partitioning has several redundant splits where areas of the same color (meaning they suggests the same optimal action) are split in two. For instance, the region $((0, 0), (1, 1))$ and the region $((0, 1), (1, 2))$ both specify a as the optimal action, and we could replace these two regions with a single one given by $((0, 0), (1, 2))$. Since each region is represented in our decision tree as a leaf node, the fewer regions we have the smaller a tree we need to represent it.

The problem of finding maximum sized regions is a local optimization probem: Given a point $s^{\min}$, find $s^{\max}$ such that $\nu = (s^{\min}, s^{\max})$ has singular mapping in $\mathcal{T}$ while no other region $\nu' = (s^{\min}, s')$ where $s'_j = s_j^{\max}$ for $j = 1, \ldots, i-1, i+1, \ldots, K$ and $s'_i > s_i^{\max}$ has this property.

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



(a) (b)

Fig. 1: Two representations of the toy strategy introduced in Example **??**. In (a) the Q-table is represented as a decision tree. In (b) a 2D visualization of the state space partitioning is showed, where the colors indicate what the optimal action is in that area of the state space (red for a, green for b and blue for c).

time by collecting and sorting the bounds on all branch nodes in $\mathcal{T}$ and allows accessing $\mathcal{T}_{i,j}$ in constant time.

Exploiting this notation, if p is a K-dimensional vector of index pointers to bounds in $\mathcal{T}$, such that $p_i \in p$ is a pointer to $\mathcal{T}_{i,p_i}$, then we can define a point at an intersection of bounds in all K dimensions as $s^p_{\mathcal{T}} = (\mathcal{T}_{1,p_1}, \mathcal{T}_{2,p_2}, \ldots, \mathcal{T}_{K,p_K})$. We will omit the subscript $\mathcal{T}$ on $s^p_{\mathcal{T}}$ when it is clear from the context. 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 $\exists p_i \in p, p_i = |\mathcal{T}_i| + 1$.

The algorithm works by maintaining two vectors of index pointers, $p^{\min}$ and $p^{\max}$, and iteratively increasing $p^{\max}$ until a region $\nu = (s^{p^{\min}}, s^{p^{\max}})$ cannot be expanded further. The regions are stored in a list $\mathcal{R}$ and a list $\mathcal{P}$ is keeping track of the $p^{\min}$ vectors to use as starting points for the region search. Initially, $\mathcal{R}$ is empty and $\mathcal{P}$ contains the lowest bound for each dimension (so it equals the first column of the matrix $\mathcal{T}$). The pseudo-code is given in Algorithm 1.

Let $p^{\min}$ be the result of popping the lexicographically smallest element of $\mathcal{P}$ (line 4). We can then define $s^{\min} = s^{p^{\min}}$ as the 'lower left' corner in (or the origin of) a region $\nu = (s^{\min}, s^{\max})$ where $s^{\max}$ is the point we want to determine, so that ν is maximized and has singular mapping in $\mathcal{T}$. By definition, $s^{\max} = s^{p^{\max}}$ satifies the singular mapping requirement for $p^{\max} = (p^{\min}_1 + 1, \ldots, p^{\min}_K + 1)$, since no branch node in $\mathcal{T}$ splits on a predicate c where $\mathcal{T}_{i,p^{\min}_i} < c < \mathcal{T}_{i,p^{\min}_i+1}$ for any $i = 1, \ldots, K$.

Finding a $p^{\max}$ that maximizes the region comes down to finding a vector $\Delta_p \in \mathbb{Z}^K$ so that $p^{\max} = p^{\min} + \Delta_p$. The definition of Δ_p is given below.

Definition 3. *Given* $p^{\min} \in \mathbb{Z}^K$*, a decision tree* $\mathcal{T}$ *over a* K*-dimensional space and a list* $\mathcal{R}$ *of already found regions,* $\Delta_p \in \mathbb{Z}^K$ *is a vector such that for* $p^{\max} = p^{\min} + \Delta_p$ *the region* $\nu = (s_{\mathcal{T}}^{p^{\min}}, s_{\mathcal{T}}^{p^{\max}})$ *has singular mapping in* $\mathcal{T}$*, does not overlap with any other region in* $\mathcal{R}$ *and no other* $\Delta_{p'} > \Delta_p$ *has this property.*

A greedy approach to finding Δ_p starts with $\Delta_p = \mathbf{1}^K$, where $\mathbf{1}^K$ is the K-dimensional vector of ones. 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 and no overlap with regions in $\mathcal{R}$. 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. When all dimensions have been exhausted, Δ_p adheres to Defintion 3.1.

Having found a region $\nu = (s^{p^{\min}}, s^{p^{\max}})$, we add it to $\mathcal{R}$. Further, we add to $\mathcal{P}$ the points from which the search should continue at future iteration. These are the K points that agree with $p^{\min}$ in all dimensions except i, where they agree with $p^{\max}$, ie. we add to $\mathcal{P}$ points $(p_1^{\min}, \ldots, p_i^{\max}, \ldots, p_K^{\min})$ for $i = 1, \ldots, K$ (where $p_i^{\max} < |\mathcal{T}_i|$). Then if $\mathcal{P}$ is not empty, we repeat the entire process. Otherwise the algorithm terminates and returns $\mathcal{R}$ which now represents a new partitioning that respects $\mathcal{T}$.

Algorithm 1 MaxPartitions

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

$\mathcal{R} \leftarrow \{\}$

$\mathcal{P} \leftarrow \{\mathbf{1}^K\}$ ▷ $\mathbf{1}^K$ is a K-dimensional vector of ones

while $\mathcal{P}$ is not empty **do**

  $p^{\min} \leftarrow$ Remove lexicographically smallest element of $\mathcal{P}$

  if $s_{\mathcal{T}}^{p^{\min}}$ is not covered by any region in $\mathcal{R}$ **then**

    $p^{\max} \leftarrow p^{\min} + \Delta_p$ ▷ See Definition 3.1

    $\mathcal{R} \leftarrow \mathcal{R} \cup \{ (s_{\mathcal{T}}^{p^{\min}}, s_{\mathcal{T}}^{p^{\max}}) \}$

    $\mathcal{P} \leftarrow \mathcal{P} \cup \{ (p_1^{\min}, \ldots, p_i^{\max}, \ldots, p_K^{\min}) \mid i \in 1, \ldots, K, \quad p_i^{\max} < |\mathcal{T}_i| \}$

return $\mathcal{R}$

3.2 Analyzing the algorithm

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

Running time The first thing to notice is the outer while loop over $\mathcal{P}$. Points are dynamically added to $\mathcal{P}$ every time a new region is constructed, and in the worst case, K new points (one for each dimension) are added for each region. The number of regions that can be found and constructed is bounded by the size of the original parition $\mathcal{A}_{\mathcal{T}}$, as the worst case is when $\mathcal{A}_{\mathcal{T}}$ is already a minimal partitioning that respects $\mathcal{T}$. In this case, the algorithm will produce $\mathcal{R} = \mathcal{A}_{\mathcal{T}}$ and the number of regions will necessarily be the same. Let $N = |\mathcal{A}_{\mathcal{T}}|$. Then we can state that the outer while loop is bounded by $O(KN)$.

How about finding Δ_p? The procedure is to increment by 1 in any one unexhausted dimension and then check for the validity of that increment. This check has two components: (a) check if the new region still has singular mapping in $\mathcal{T}$ and (b) check if the new region overlaps with any region already in $\mathcal{R}$. Let $\nu = (s^{\min}, s^{\max})$ be the candidate region for some $\Delta_p = \mathbf{1}^K + \hat{\mathbf{e}}_i$ (ie. so $s^{\max} = s^{p^{\min} + \Delta_p}$).

For (a), we have to query $\mathcal{T}(\nu)$ which visits all leaves in $l \in \mathcal{T}$ for which $\lambda(l) \cap \nu \neq \emptyset$. Assuming $\mathcal{T}$ is balanced, then the path from the root to a leaf is $O(\log T)$ where $T-1$ is the size of the tree (and $T = 2N-1$). The worst case for retrieving a set of size L, is that all L leaves share the least amount of path. Since all paths share the root node, the worst case for $L = 2$ is when the root is the *only* shared node, in which case the operation would require $(2 * \log T) - 1$ visits (with the last term representing 1 shared node on the paths). For $L = 3$ and $L = 4$, all paths must at least share the root node as well as one of its two children. Thus, for $L = 3$ we have $(3 * \log T) - 3$ and for $L = 4$ we have $(4 * \log T) - 5$ (since now, adding the 3rd and 4th leaf would require $\log T$ operations minus the checks on the nodes on the path shared by the 1st and 2nd leaf).

In general, this becomes

$$L \log T - \sum_{i=1}^{L} \lceil \log i \rceil \tag{1}$$

Using Stirlings approximation [1] we can get rid of the summation further reduce:

$$\begin{aligned} L \log T - \sum_{i=1}^{L} \lceil \log i \rceil &\approx L \log T - (L \log L - L + 1) \\ &= L \left(\log T - \left(\log L - 1 + \frac{1}{L} \right) \right) \\ &= L \left(\log T - \log \left(\frac{L}{2} \right) - \frac{1}{L} \right) \\ &= L \log \left(\frac{2T}{L} \right) - 1 \end{aligned} \tag{2}$$

In Big-O notation, this is $O(L\left(\log \frac{T}{L}\right))$, ie. the complexity of the query $\mathcal{T}(\nu)$ is linear in the size of the output set times the logarithm of the ratio between the size of the tree and the size of the output set.

By keeping L small, we can therefore obtain a complexity that is close to $O \log T$. We can do this by noting, that we do not need to query the entire candidate region at each increment. Say we start from some region $\nu_1 = (s^{p^{\min}}, s^{p^{\max}})$ with $p^{\max} = p^{\min} + \mathbf{1}^K$. We then have $\mathcal{T}(\nu_1) = \{\alpha\}$ with $\alpha \in \mathit{Act}$ (as ν_1 by design cannot span more than one region in the original paritioning). Then, for our next candidate region $\nu_2 = (s^{p^{\min}}, s^{p^{\max}+\hat{\mathbf{e}}_i})$ we only need to check that the region given by $\nu_2 \backslash \nu_1 = (s^{p'}, s^{p^{\max}+\hat{\mathbf{e}}_i})$ with $p' = (p_1^{\min}, \ldots, p_i^{\max}, \ldots, p_K^{\min})$ being an intermediate minimum point defining the lower bounds of the new part of the candidate region. With this technique, we avoid querying the same region again and again until the search terminates and we keep the expected complexity of each query operation minimal.

How often is this operation performed? We increase Δ_p (starting from $\Delta_p = \mathbf{1}^K$) by $\hat{\mathbf{e}}_i$ until the region $\nu = (s^{p^{\min}}, s^{p^{\min}+\Delta_p})$ no longer has singular mapping and so for each such increment, we have to query $\mathcal{T}(\nu)$. In the worst case, a single search for Δ_p thus searches through each bound on each dimension. However, this would then result in the outer loop only running once since then a region spanning the entire state space would have been found. On the other hand, it is very difficult to say excactly how many increments of Δ_p can be expected.

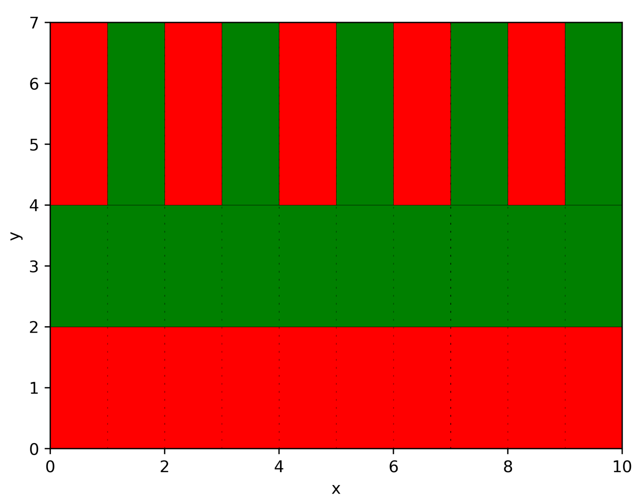


Fig. 2: Example of a bad situation in terms of checking for Δ_p. The dashed lines extends the bounds of each region to show how the algorithm processes the bounds.

Figure 2 gives an example of a partioning with a structure that requires checking a lot of bounds unrelated to the current region under construction. The algorithm would start by considering the region $\nu = (s^{\min}, s^{\max})$ with $s^{\min} = (0,0)$ and $s^{\max} = (1,2)$. But then, $s^{\max}$ would be increased to $(2,2)$ then to $(3,2)$, then $(4,2)$ and so forth. The same would be the case for the region starting in $s^{\min} = (0,2)$. In total, this partioning would make the algorithm attempt $11 + 11 + 2 * 10 = 42$ increments even thoug, the number of (upper) bounds is only 13.

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.

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 K dimensions (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.

4 From Q-trees to Decision Tree

4.1 Defining Q-trees

In Reinforcement Learning [2] an agent is trying to estimate the expected value (cost or reward) of taking and action A in a state S. This is called the Q-value. Let Act be a finite set of actions and let $\mathcal{S} \in \mathbb{R}^K$ be the state space (a bounded K-dimensional euclidean space) then the goal is to learn the function $Q(s,a) : S, A \mapsto \mathbb{R}$ that for any $s \in \mathcal{S}$ and $a \in Act$ maps to the Q-value of the state-action pair.

When $\mathcal{S}$ is continuous, the Q-function either has to be approximated or the state space needs to be discretized. In the latter case, $\mathcal{S}$ can be redefined in terms of well-defined bounded subspaces where each $S \in 2^{\mathbb{R}^K}$ now defines a smaller area of the original state space $\mathcal{S}$ and we by $S_{i,lower}$ and $S_{i,upper}$ respectively denote the lower and upper bound of dimension i in S. Further, we require that $\bigcup_S S = \mathcal{S}$.

For evaluating a particular state s, we say that $S = s$ iff $S_{i_{lower}} \leq s_i < S_{i_{upper}}$ for all $i = 1, \ldots, K$. This allows for a tabular representation of $Q(s,a)$, where the function is essentially just at lookup-table with $|\mathcal{S}| \times |Act|$ entries. The disadvantage of this approach is that the Q-table quickly grows very large and that many of the discrete states are irrelevant (in the sense that they are never actually visited). This can be remedied if close care is taken to designing the discretization, but this would in itself impose bias onto the learning.

UPPAAL Stratego approaches the task of discretizing the state space in a different way. Instead of schematically discretizing $\mathcal{S}$ *a priori* to the training, discretization is part of the Q-value estimation. What happens is ...

The introduction of a partitioning $\mathcal{A}$ and regions ν which I describe in Section 3 should probably come here instead.

The result is a strategy represented by a set of binary decision trees, each pertaining to a specific action in $a \in Act$, and whose leaf nodes carries the Q-value of taking action a in the state s defined by the constraints in the branch nodes on the path from the root to the leaf. We call these trees *Q-trees* and denote by $\mathcal{T}_A$ the Q-tree for action $A \in Act$ and we define $\mathcal{T}_A(s) = Q(s, a)$ when $A = a$. Given the complete set of Q-trees the matter of choosing the optimal action in a state s can — for a greedy policy π and with the Q-values representing expected cost — be defined as $\pi(s) = \operatorname{argmin}_{a \in Act} \mathcal{T}_A(s)$.

4.2 Converting to decision tree

With Definition **??** we can now consider how to construct a single decision tree $\mathcal{T}$ so that $\pi(s) = \operatorname{argmin}_{a \in Act} \mathcal{T}_A(s) = \mathcal{T}(s)$ for all $s \in \mathcal{S}$. That is, instead of a Q-tree we will construct a decision tree where the leaf nodes carries the action A that satisfies $A = \operatorname{argmin}_{a \in Act} T_A(\lambda(l))$ for a given leaf l. In the following, we will present the procedure for doing so in general terms while the full specification of the algorithm is available in Appendix A.

First, let $\mathcal{L}$ be the set of every leaf in the set of Q-trees and let each leaf $l \in \mathcal{L}$ be defined as $l = (S^l, a_l, q_l)$ where $S^l = \lambda(l)$ in T_A, a_l is the action of the Q-tree l originally belonged to and q_l is the Q-value of taking action a_l in state S^l (we use superscripts in S^l to avoid notational clutter when we later need to index variables and bounds in S^{l_i} and S^{l_j} at the same time). We sort $\mathcal{L}$ in ascending order according to q_l (meaning l_0 has the best Q-value of any leaf) and use the first leaf, l_0, to build the first path in the tree. This path requires $2 \times K$ branch nodes, one for each lower and upper bound of each dimension in S^l.

The decision about the order in which to predicate the branch nodes on each variable bound can and will greatly affect the size of the tree. However, as determining the optimal ordering of predicates is computationally infeasible [3], we will simply resort to a randomized picking order. For the root node v_0, we thus pick a variable i and a bound j at random and set $\rho(v_0) = x_i \leq c$ where $c = S^{l_0}_{i,j}$. If j is a lower bound, then we set the left child node to a dummy leaf (we will complete this subtree later) and construct a new branch node for the right child from the remaining pairs of i, j in S^{l_0} and vice-versa if j is an upper bound. We continue this procedure until $S^{l_0} = \lambda(l_0)$ holds true in the tree under construction.

For inserting the another leaf, l_j, we now need to check at each branch node v_m whether we should insert in the left subtree, in the right subtree or in both. In other words, we do two checks: if $\rho(v_m)$ is $true$ for $x_i = S^{l_j}_{i,lower}$ we continue the insertion procedure in the left subtree. If $\rho(v_m)$ is $false$ for $x_i = S^{l_j}_{i,upper}$ we *also* insert l_j into the right subtree. If both cases evaluates to $false$, we *only* do the insertion in the right subtree. If we encounter a dummy leaf, we either construct a new branch node as we did for the initial path, ie. by randomly picking a still unchecked variable and bound to use for the predicate function, or — in the case that $S^{l_j} = \lambda(l_j)$ already holds true for the tree under construction — simply insert l_j instead of the dummy.

If we encounter a non-dummy leaf we can exploit the fact that the leaf nodes are inserted in a sorted order according to their Q-values. This ensures that if we encounter l_i during insertion of l_j then we know that $q_i \leq q_j$ and we can therefore safely stop the insertion of l_j (in this particular subtree) as we know that for all $s \in S^{l_i} \cap S^{l_j}$ it must hold true that $\pi(s) = a_i$.

When all leaf nodes from the set of Q-trees have been processed the resulting tree $\mathcal{T}$ represents the exact same strategy but now without any notion of Q-values. In the Python library built for this paper, it is possible to export a decision tree representation to a Q-tree representation that can then be imported into `UPPAAL Stratego` in order to test the performance of the strategy. This is done by for each $A \in \mathit{Act}$ creating $\mathcal{T}_A$ as a copy of $\mathcal{T}$ and then for every leaf l setting $q_l = 0$ if $a_l = A$ and $q_l = 999$ if $a_l \neq A$.

5 Minimization techniques

As we can give no guarantees to the minimality of the decision tree created from a set of Q-trees, $\mathcal{T}$ can grow very large and even contain more paths than all the Q-trees combined. This is unwanted, and we therefore present several minimization techniques that can drastically decrease the size of $\mathcal{T}$.

5.1 Simple pruning

This subsection and the next (Section 5.1 and 5.2) I am not at all certain about how to actually approach yet. The simple pruning is so simple, that it is almost redundant to describe and the analytical pruning is only partially implemented and not very systematic yet.

The algorithm described in Section 4.2 naively inserts leaf nodes without any consideration of optimality (except what little is given from the fact that leaf nodes are inserted in order of Q-value). This yields some obvious cases where branch nodes can be pruned away.

Say we have a path $p = \{v_0, v_1, \ldots, v_n\}$ where both children of v_n are leaf nodes, l_i and l_j. If $a_i = a_j$ then the predicate at v_n bears no significance and we can replace that node with either l_i or l_j. Now the child of v_{n-1} that used to be v_n is a leaf, which might again result in a situation where v_{n-1} has two leaf children with the same action. Thus, we iteratively check for this condition all the way up through the tree, pruning any such cases.

Something something $\lambda(v_n) = \lambda(l_i) \cup \lambda(l_j) \wedge a_i = a_j \implies \pi(\lambda(v_n)) = a_i = a_j$.

5.2 Analytical pruning

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.

5.3 Empirical pruning

During training, the partitioning scheme used by UPPAAL Stratego to discretize the state space is explorative in the sense that it is non-deterministic when it creates new partitions. As it goes along, the algorithm discovers areas of interest where the choice of action seem to have greater impact on the overall cost and it then refines its partitioning in these areas.

This leads to somewhat abundance of partitions early on in the training, where more or less random splits turns out to not influence the decision making. These splits are carried on into the final strategy and they are also imported into our converted decision tree where the merging of the different Q-trees actually amplify this abundance.

This has two consequences. One is that the state space will be partitioned in such a way that neighboring partitions actually prescribe the same action, but do not necessarily appear as neighboring leafs in the strategy tree (ie. they do not have the same parent). We will deal with this problem in the Section 3. The other consequence is that we end up with a lot of leaf nodes that in practice will never be visited, as the system that is modeled either never end up in such a state or because the strategy has the controller behave in such a way, that such a state is always avoided.

This entire intro should possible be moved to the beginning of this section (Section 5). Also, I would like to be able to describe the 'issue' stemming from UPPAAL more precisely but for that, I probably need help from Peter.

To deal with this, we employ a technique we call *empirical pruning*. In contrast to our other effort, this technique is not based on any analysis of the tree structure or state space but instead employs sample data to prune the tree of any leaf nodes that are either never or rarely visited. This have the drawback, that we loose our ability to give guarantees about the strategy, as our sampling *might* just miss an important edge case that our empirically pruned strategy then does not know how to handle. On the other hand, given a large a enough sample, this risk is negligible since such a case would most likely have been just as rare during training, meaning the strategy would not even be ready to deal with it properly had it not been pruned from the tree.

The way it works is by gathering a sample of data points $D = \{s_0, s_1, \ldots, s_T\}$ representing the state of the system at each time step during a run (or preferably several runs) where the controller acts according to a well trained strategy represented by the tree $\mathcal{T}$. For each state $s_t \in D$ we increase a counter at the leaf node at the end of the path that came from evaluating $\mathcal{T}(s_t)$.

When this process is done with a sufficiently large D, we can prune all the leaf nodes that were never visited or rarely visited. If we prune the never visited nodes, we call it *zero-pruning*. Pruning nodes visited once is called *one-pruning* and so forth. The pruning is simply done by removing every branch node with a leaf child that is never visited and instead 'promoting' the subtree that is its other child.

For example, if we have a path $p = \{\ldots, v_{i-1}, v_i, v_{i+1}, \ldots\}$ and v_i has a leaf node l for its other child and we see from sampling that l is never visited, then the constraint that v_i represent has no relevance for v_{i+1}. Therefore, we remove v_i and set v_{i+1} as a child of v_{i-1} instead. Doing this iteratively from the 'left-most' leaf and all the way through the tree can lead to substantial reductions, as we will show in Section 6, and provided that the sample size is large enough the performance of the pruned strategy stays on par with the original.

6 Experiments

In this section, we will apply the techniques presented in the previous sections to the Bouncing Ball example introduced in [4]. In that example, a ball is given by its position and velocity as it bounces up and down from the floor. A controller is given the choice between two actions — hit or do nothing — and is tasked with keeping the ball bouncing for as long as possible with as few hits as possible.

We perform the following steps:

- Synthesize a strategy in UPPAAL Stratego. This gives us two Q-trees (one for $a = 1$ (hit) and one for $a = 0$ (no hit)) with a combined number of paths 13,405.
- Convert the Q-trees to a single decision tree following the procedure described in Section 4. The conversion is followed by simple pruning (Section 5.1). The resulting tree has 84,336 paths and the entailed partitioning can be seen in Figure 3a.
- Run the `MaxPartitions` algorithm on the converted tree. This reduces the number of partitions to only 703.
- Construct a decision tree to represent the new partitioning as described in Section 3.3. This gave a tree with 892 paths. The resulting partitioning is seen in Figure 3b.
- Generate samples from 1000 runs with a maximum of 300 time steps each, logging the state at every 0.5 time step. This gives 600,000 sample data points which we use for empirical pruning, resulting in a tree of just 189 paths.
- *Now maybe something about my analytical pruning that got the tree down to just 164 paths*
- Export all versions of the strategy to UPPAAL Stratego format and compare performance on the system. Results are shown in Table 1.

A couple of things are of notable interest here.

First, converting the strategy from a set of Q-trees to a decision tree vastly increases the total number of paths. Without any minimization effort, it seems that the decision tree of size 84,336 does not yield any particular advantages in terms of explainability and interpretability compared to the two Q-trees with a combined size of 13,405 paths. For cases where there are more variables or possible actions, this would most likely pose an even greater issue.

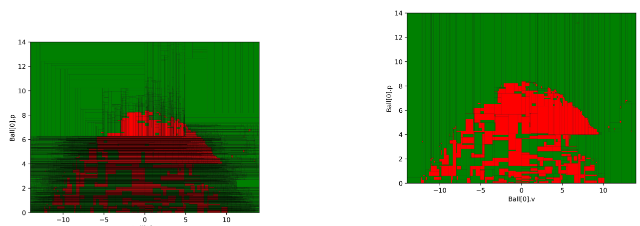
Second, the effect of `MaxPartitions` is a drastic reduction in the number of partitions. Even though the list of regions cannot be perfectly represented by a decision tree and the number of paths in the constructed tree thus increases a

Table 1: Comparing the performance of controllers for the bouncing ball example over 1000 runs for 120 time steps each before and after various attempts at minimizing the size through either empirical pruning or the `MaxPartitions` algorithm.

Version	Paths	Expectation (hits)	Deviation
Q-trees	13,405	38.401	0.177
Original DT	84,336	38.431	0.178
MaxPartitions	895	38.435	0.177
MaxPartitions then Prune	189	38.347	0.173

bit, the tree of size 892 is a decisive improvement from both the original set of Q-trees and the especially the converted tree.

Third, the decreasement in the number of paths after empirically pruning the tree indicates that many areas of the state space are never visited in practice. It is also remarkable, that by utilizing this sample data we can further reduce the already minimized tree by a factor of 4 without sacrificing performance.



(a) Original paritioning

(b) Partitioning after `MaxPartitions`.

Fig. 3: A 2D visualization of the partitioning (dotted lines) of the state space before and after `MaxPartitions`. The x-axis is velocity and the y-axis is the balls position. Green areas represents the action 'no hit', red areas represents 'hit'.

6.1 dtControl

This is very experimental

The tool `dtControl` [5] has the ability to take a synthesized strategy represented as a look-up table and convert the representation to a decision tree that respects the safety requirements but compresses the size immensely. This is naturally of great interest for our case, but both the Q-tree representation and our own decision tree conversion alters the setup somewhat.

However, even though the tool actually supports directly working with the output format of UPPAAL Stratego, this is only the case for strategies learned using the `control[]` directive, which controls for certain defined safety parameters to always be respected. In our case of the bouncing ball example, the controller is trained with the `minE[]` directive, which minimizes a given parameter (in this case, the number of times the controller hits the ball).

Instead, we had to create our own output files to use as input for `dtControl`. According to the documentation, a controller strategy can be specified in a simple CSV format where each line contains an allowed state/action pair, that is, N values representing a state where the following M values constitutes an allowed action. For example, in the case of the bouncing ball, we have two state variables (position and velocity) and one action variable (hit or not hit), meaning each line would have three values.

We have attempted two experiments with different ways of specifying our original controller.

In the first experiment, we used the trained controller to generate 30,000 samples of state/action pairs. That is, the UPPAAL model was run for 300 time steps with the trained controller deciding what action to take in each encountered state and then the state/action pairs were logged at each 0.01 time step.

In the second experiment, we converted the strategy a set of Q-trees (the initial UPPAAL format) to a DT as described in Section 4. This DT had a partition size (number of leaves/paths) of 91,054. We used these partitions as the input data to `dtControl` by taking the maximum value of each variable in each individual partition together with the optimal action of that state. That is, we effectively specified the discretization of the state space by defining the bounds of each state paired with the allowed/optimal action.

Table 2: Comparing the performance of controllers for the bouncing ball example over 1000 runs for 120 time steps each before and after various attempts at minimizing the size through `dtControl`.

Version	Paths	Construction time	Expectation (hits)
Original DT	91,054	—	38.431
Samples	27,234	8:14	318.411
State bounds	521	0:41	315.769

The results when applying the generated strategies to the model in UPPAAL are given in Table 2 together with the baseline original decision tree directly converted from the Q-tree set. As is seen, when we used samples, we still got a somewhat large DT that took more than 8 minutes to generate. And the performance (expected number of hits) is substantially worse than the baseline version. For the version based on state bounds, we got a much smaller tree with only 521 paths, but the performance was still very far from the original.

References

1. M.T.L.B., "An Introduction to Probability Theory and its Applications, vol. I. By William Feller. [Pp. xii + 419. New York: John Wiley and Sons Inc.; London: Chapman and Hall Ltd. 1950. Price 48s.]," *Journal of the Staple Inn Actuarial Society*, vol. 10, pp. 316–318, Nov. 1951.
2. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* The MIT Press, second ed., 2018.
3. L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is NP-complete," *Information Processing Letters*, vol. 5, no. 1, pp. 15–17, 1976.
4. 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.
5. P. Ashok, M. Jackermeier, J. Křetínský, 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* (J. F. Groote and K. G. Larsen, eds.), (Cham), pp. 326–345, Springer International Publishing, 2021.

Appendices

A Converting Q-trees to Decision Tree — Algorithm

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.1 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 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.

A.2 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

Algorithm 2 MakeRoot

```
procedure MakeNewNode(var, bound, action, isHigh, prevNode)
    node ← Node(var, bound)
    if isHigh then
        node.high ← Leaf(action, ∞, State(·))
    else
        node.low ← Leaf(action, ∞, State(·))
    if prevNode is not None then
        if prevNode.low is None then
            prevNode.low ← node
        else
            prevNode.high ← node
    return node

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

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).

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

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))

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 ← SETLOWER(state, var, bound)
    lowState ← SETUPPER(state, var, bound)
    high ← Leaf(action, q, highState)
    low ← 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 ≤ 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 ← 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 ← 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)
```