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

With the tool UPPAAL Stratego [1] it is possible to learn safe and (near-)optimal strategies for controllers of any system that can be modelled as a Markov Decision Process (MDP). This has many important applications, one being the government of cyber-physical systems that often have important safety requirements to avoid real-world catastrophees while also needing to perform at a certain level of expectation to satisfy consumers, citizens or other participants in a network.

For these cases of cyber-physical systems – especially within critical infrastructure such as water management, transport or energy distribution — another important feature of a controller is that it is explainable and understandable by a human expert. A well-established technique for representing machine learning strategies in ways interpretable by humans is with decision trees, where the path from a root node to a leaf node directly specifies how a state of the system is evaluated so that a specific action is decided upon.

In UPPAAL Stratego the strategies learned are represented by a collection of decision trees, one for each action. The leaf nodes of each tree then gives an evaluation of the expected cost of taking that action in the state under evaluation. In Q-learning, the Reinforcement Learning technique powering UPPAAL Stratego, this is called the Q-value and as such, we call the trees Q-trees. While this approach allows for an online discretization of a continuous state space and

improved approximation of the optimal strategy, it does not retain the expressive nature of the decision tree representation of its strategy, as each decision is now an argmin function over the unique evaluation of the expected cost of state-action pairs.

In this paper, we therefore suggest a method for converting a set of Q-trees to a single decision tree representing the same deterministic strategy that the argmin over the set of Q-trees would yield. This makes UPPAAL Stratego able to not only give safe and near-optimal strategies but also to provide them in a fashion that has the benefits of decision tree representation.

However, we often see that the number of paths in these trees grows very large, which again makes them exceedingly hard for humans to actually comprehend. Furthermore, since many controllers of cyber-physical systems will run on embedded hardware with limited space and memory capacity, large strategies might be impossible to even run get running in these systems. Therefore, we also provide an algorithm for minimizing the number of leaf nodes the decision tree by inspecting the Euclidean state space of the system under consideration and the partitions that the strategy entails.

Example 1 As a running example, we consider an articifial strategy σ with a Q-table given by Table 1. The state space $\mathcal{S} \in \mathbb{R}^2$ of the underlying environment is given by $x \in [0, 3]$ and $y \in [0, 3]$ and the set of possible actions is $Act = \{a, b, c\}$. Each entry in the Q-table is a tuple (a, b, c) representing the Q-value (cost) of each of the three actions in that state. The dimensionality of the Q-table is $3 \times 3 \times 3$ and it has 27 Q-entries.

Table 1: Q-table for example strategy. Each entry is a tuple of the Q-values (expected cost) of action a, b and c respectively for some configuration of x and y.

	$0 < x \leq 1$	$1 < x \leq 2$	$2 < x \leq 3$
$0 < y \leq 1$	(2, 7, 3)	(4, 3, 1)	(6, 8, 1)
$1 < y \leq 2$	(3, 4, 4)	(3, 2, 4)	(3, 5, 9)
$2 < y \leq 3$	(3, 3, 1)	(4, 3, 1)	(2, 7, 3)

2 Preliminaries

We start by defining some relevant concepts.

Definition 1 (Strategy). *A strategy $\sigma : S \to A$ over the state space $\mathcal{S}$ is a mapping from any state $s \in \mathcal{S}$ to an action $a \in Act$ where Act is a finite set of allowed actions. We say that σ is deterministic if for any $s \in \mathcal{S}$ the probability $P(a \mid \sigma(s)) = 1$ for only a single element $a \in Act$ and $P(a' \mid \sigma(s)) = 0$ for all $a' \in Act$ where $a' \neq a$.*

A strategy can be represented in several different ways. In Q-learning [2], the approach is to estimate a function $Q : S, A \to \mathbb{R}$ that maps a state-action pair (s, a) to a real number, that is an estimation of the expected cost of taking action a in state s.[3] Choosing an action in state s under the deterministic strategy σ is then given by $\sigma(s) = \text{argmin}_{a \in Act}\, Q(s, a)$.

For a continuous state space, the Q-function can either estimated using function approxmiation techniques or by discretization of the state space. In the latter case, the state space is considered as consisting of a set of discrete, bounded subspaces $S \subset \mathbb{R}^K$ and the Q-value is then regarded as equal among all possible configurations within a single discrete state S, ie. $Q(s, a) = Q(s', a)$ for all $s, s' \in S$. This allows for a tabular representation of the Q-function, as in the toy example in Table 1.

Definition 2 (Partitions). *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 $\nu \cap \nu' = \emptyset$. Each region ν can be expressed in terms of two points, $p^{\min}$ and $p^{\max}$, so that for each $p = (p_1, \ldots, p_K) \in \nu$ it holds that $p_i^{\min} < p_i \leq p_i^{\max}$ for $i = 1, \ldots, K$.*

Evidently, any discretization of a state space $\mathcal{S} \in \mathbb{R}^K$ is effectively a partitioning. For example, the Q-table in Table 1 corresponds to the partitioning $\mathcal{A} = \{((0,0),(1,1)), ((1,0),(2,1)), \ldots, ((1,2),(2,3)), ((2,2),(3,3))\}$

Definition 3 (Decision tree). *A binary decision tree over the domain $\mathcal{S} \in \mathbb{R}^K$ is a tuple $\mathcal{T} = (T, \rho, \ell)$ where T is a full binary tree, ρ assigns to each branch node a predicate function of the form $\rho(s) = s_i \leq c$ where $s \in \mathcal{S}$ and $c \in \mathcal{S}_i$ for each $i = 1, \ldots, K$ and ℓ assigns to each leaf node a label (categorical or numerical).*

For a decision tree $\mathcal{T}$, we can obtain a decision $\delta = \mathcal{T}(s)$ from any state $s \in \mathcal{S}$ by following the *path* from the root node to a leaf by evaluating $\rho(s)$ at every branch node and following the left path if the predicate is true and the right path otherwise. For each node η in the tree (branching or leaf) the path to this node defines a bounded subspace S of $\mathcal{S}$ and we denote this as $S = \lambda(\eta)$.

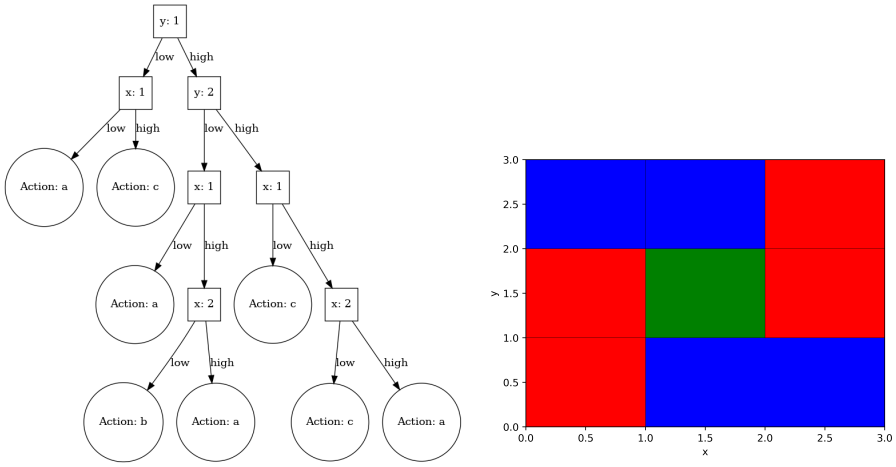
Further, for a euclidean domain we also allow evaluating a region of $\mathcal{S}$. Given a region $\nu = (p^{\min}, p^{\max})$, $[\delta]_\nu = \mathcal{T}(\nu)$ is the set of all decisions that can be obtained evaluating configurations of ν, ie. $\mathcal{T}(\nu) = \{\mathcal{T}(s) \mid s \in \nu\}$.

3 MaxPartitions algorithm

The idea of the `MaxPartitions` algorithm is to maximize the size of the state space partitions, so that the overall number of partitions used to represent the strategy is minimized. As previously described, the partitioning that happens during training in UPPAAL Stratego creates a lot of redundant splits in the state space and this is amplified when we convert the Q-trees to a unified decision tree.

[3] The Q-value can also be the expected reward.

As an example, take a look at the example strategy in Figure 1. On the left, is the decision tree representation and on the right is a 2D visualization of the partitioning of the state space. It is easy to see, that this perfectly valid decision tree entails a state space partitioning with several redundant partitions, where areas of the same color (meaning they suggests the same optimal action) are split in two. Since each confined area, ie. partition, is represented in our decision tree as a leaf node, the fewer partitions we have the smaller a tree we need to represent it.



(a) (b)

Fig. 1: Two representations of the toy strategy introduced in Example 1. In (a) the deterministic version of 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).

The `MaxPartitions` algorithms achieves this by maximizing the size of each partition so the strategy can be represented with a minimum number of partitions. The basic concept of the algorithm is to try and construct as large a partition as possible starting from a given point (typically the point representing the minimum of each dimension) while retaining the invariant that only one action is considered optimal in this partition. When this invariant is violated, the new partition is stored and the operation continues from the 'corner' points of the new partition. The pseudocode is given in Algorithm 1.

3.1 Description of the algorithm

The algorithm takes 4 inputs as arguments. $\mathcal{T}$ is a decision tree representing a strategy σ and a complete partitioning of the state space $\mathcal{S} \in \mathbb{R}^K$; $\mathcal{C}$ is the set of

constraints entailed by $\mathcal{T}$ so that each element of $\mathcal{C}$ is a tuple (V_i, c) representing the predicate $\rho(x) = x_i \leq c$ of a branch node in $\mathcal{T}$; $\mathcal{V}$ is the set of variables (dimensions); and p_{start} is the initial point the algorithm should start from.

In line 2 and 3 we define two lists. *regions* is where we will store the regions we identify during the algorith, and $\mathcal{P}$ is used to keep track of points from where we will search the state space. We initialize $\mathcal{P}$ with the single point p_{start}. In line 4, we start our main loop which terminates when $\mathcal{P}$ is empty.

Line 5-11 sets up the search we are about to do from the next point in $\mathcal{P}$. First, we initialize an empty list *exhausted* to keep track of the dimensions that we are done exploring. Then we sort $\mathcal{P}$ so that we process the points in a consistent order. We then pop the first element of $\mathcal{P}$ and uses this as $p^{\min}$ for the region we are about to create. However, if we have already covered $p^{\min}$ in a previous iteration (by constructing a region ν in which $p^{\min}$ is contained), we skip this iteration and continue at line 4. Otherwise, we define $p^{\max} = p^{\min}$ so we now have an infinitesimally small region $\nu = (p^{\min}, p^{\max})$, and finally we sort $\mathcal{C}$ in ascending order with respect to the distance to $p^{\min}$. That is, we order $\mathcal{C}$ so that for any two consecutive tuples $(V_i^m, c^m), (V_j^{m+1}, c^{m+1}) \in \mathcal{C}$ it holds that $abs(c^m - p_i^{\min}) \leq abs(c^{m+1} - p_j^{\min})$.

This last part about sorting $\mathcal{C}$ currently isn't in the pseudocode, as it got cut during refactoring — but it needs to go back in!

In line 12 we begin our search through $\mathcal{C}$ for possible candidate bounds to grow our new region with. As mentioned, each bound is a tuple giving a dimension V_i and a constraint value c. We only look at the subset of $\mathcal{C}$ where V_i is not exhausted and $c > p_i^{\min}$, and we process these bounds in sorted order according to their distance to $p^{\min}$ (currently not reflected in the pseudocode). When we find a candidate bound, we store the current value of $p_i^{\max}$ in a variable $oldValue$ (line 14) before updating it with the new bound (line 15).

In line 17 we do 3 different checks to see if our current region $\nu = (p^{\min}, p^{\max})$ is now violating any requirements. First, we check if evaluating ν in $\mathcal{T}$ only returns 1 action. Secondly, we check if we have moved $p^{\max}$ into an area already explored earlier (akin to the check in line 8). And lastly, we check if the latest update to $p_i^{\max}$ moved it to the limit of the values V_i can attain (denoted $V_i^{\max}$). If neither of these are true, we continue our attempt at updating $p^{\max}$ from line 12.

If any of the conditions are true, we proceed to check if and how we should store the region spanned by $p^{\min}$ and $p^{\max}$. First, in line 18, we add V_i to the list of exhausted dimensions. This is because we know, that in one way or another, expanding $p^{\max}$ in the direction of V_i has violated on of the requirements in line 17 and as such, V_i should be ignored going forward. Then in line 19, we check if $p_i^{\max}$ is still smaller than $V_i^{\max}$. If it is, that means that we have to roll back the last update, which we do in line 20.

Then in line 21-22, we check if there are still unexhausted dimensions, and if so, we continue the search from line 12. Otherwise, we have have found $p^{\max}$ that spans the largest area from $p^{\min}$ that only prescribes one action in $\mathcal{T}$ and neither exceeds any dimensional upper limits of the state space or expands into other found regions. We can therefore, in line 23, add the region $\nu = (p^{\min}, p^{\max})$ to the list of regions.

Algorithm 1 MaxPartitions

```
1: function MaxPartitions(𝒯, 𝒞, 𝒱, p_start)
2:     regions ← {}
3:     𝒫 ← {p_start}
4:     while 𝒫 is not empty do
5:         exhausted ← {}
6:         p^min ← min 𝒫
7:         𝒫 ← 𝒫 \ {p^min}
8:         if p^min is explored then
9:             continue
10:        p^max ← p^min
11:                ▷ The list also need to be sorted according to distance to p^min
12:        for V_i, c in {(V_i, c) | (V_i, c) ∈ 𝒞, c > p_i^min, V_i ∉ exhausted} do
13:
14:            oldVal ← p_i^max
15:            p_i^max ← c
16:
17:            if |𝒯((p^min, p^max))| > 1 or p^max is explored or p_i^max = V_i^max then
18:                exhausted ← exhausted ∪ V_i
19:                if p_i^max < V_i^max then
20:                    p_i^max ← oldVal
21:                if |exhausted| < |𝒱| then
22:                    continue from 12
23:                regions ← regions ∪ {(p^min, p^max)}
24:
25:                for V_i in 𝒱 do
26:                    if p_i^min ≠ p_i^max and p_i^max < V_i^max then
27:                        𝒫 ← 𝒫 ∪ {(p_1^min, ..., p_i^max, ..., p_k^min)
28:                continue line 4
29:    return regions
```

The last thing needed is to add new starting points to $\mathcal{P}$. This is done in line 25-27. For every V_i in $\mathcal{V}$ we create a new point $p' = (p_1^{\min}, \ldots, p_i^{\max}, \ldots, p_k^{\min})$ and add it to $\mathcal{P}$ as long as long as some basic properties hold. Then we break (line 28) and continue processing the next point in $\mathcal{P}$.

3.2 Analyzing the algorithm

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

For the running time, we first turn our attention to the outer while loop iterating through $\mathcal{P}$. Since $\mathcal{P}$ is dynamically grown, we must consider what the accumulated size during the entire run of the algorithm can amount to. For this, we look at the case where new points are added to $\mathcal{P}$, namely in line 27-31 where we have found a new region. We add a point for each of the K variables, barring

some edge cases. This means, that an upper bound to the number of points we must process in total is K times the number of regions we find.

The worst case number of regions the algorithm can find is on the other hand bounded by the number of partitions (or leaf nodes) in the original tree $\mathcal{T}$. This because either the partitioning $\mathcal{A}$ entailed by $\mathcal{T}$ is already optimal, in which case, the algorithm will just find one region for every leaf node in $\mathcal{T}$, or $\mathcal{A}$ is not optimal, meaning the algorithm will find fewer (but larger) regions. Therefore, the outer while loop is bounded by $O(KN)$ with N being the number of leaf nodes in $\mathcal{T}$.

Inside the while loop, we have two sorting operations and a for-loop. In line 6, we sort $\mathcal{P}$, but we ignore this in our analysis, as $\mathcal{P}$ is expected to at any time only contain a small subset of all the points entering and exiting $\mathcal{P}$ during the algorithm. Secondly, a smart data structure (eg. a heap) could sort the points at insertion and thus remove the need to sort every time.

In line 11, however, we sort $\mathcal{C}$ according to the current $p^{\min}$. As presented here, this sorting is unavoidable as we cannot expect to know the correct sorting with respect to $p^{\min}$.[4] If we assume that the sorting operation is $O(C \log C)$ with C being the size of $\mathcal{C}$, then question becomes one of estimating C. Also here we have that C is proportional to N and K, as $\mathcal{C}$ is constructed from all the predicates of the branch nodes in $\mathcal{T}$. This is at most N , or rather $N - 1$. Something something $\approx O(KN^2 \log N)$.

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 disjunction) 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 $\nu \in \mathbf{R}$ with a_ν being the action associated with ν. We iteratively create a branch node that splits $\mathbf{R}$ into two, $\mathbf{R}_{low}$ and $\mathbf{R}_{high}$, based on a predicate function $\rho(x_i, c) = x_i \leq c$ with $c \in \mathbb{R}$ so that $\mathbf{R}_{low} = \{\nu \in \mathbf{R} \mid \nu_{i,\min} \leq c\}$ and $\mathbf{R}_{\mathbf{high}} = \{\nu \in \mathbf{R} \mid \nu_{i,\max} > c\}$. 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 $\rho(x_i, c)$, more specifically which variable x_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,

[4] One could imagine, that K lists with the bounds in $\mathcal{C}$ sorted according their respective V_i were stored and consulted in the following for-loop, thus alleviating the need to sort $\mathcal{C}$ in every iteration. This approach is not pursued here.

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 disjunction 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 ν^j, ν^{j+1} it holds that $\nu^j_{i,\max} \leq \nu^{j+1}_{i,\max}$. If we then let $\rho(x_i, c) = x_i \leq c$ with $c = \nu^j_{i,\max}$ 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, $\rho(x_i, c) = x_i \leq c$ will be true for $x_i = \nu^{j+m}_{i,\min}$ and $c = \nu^j_{i,\max}$) and in $\mathbf{R}_{high}$ (because our sorting ensures that for all ν^j, ν^{j+m} it holds that $\nu^j_{i,\max} \leq \nu^{j+m}_{i,\max}$).

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

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

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

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.

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 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 $\pi(s) = \text{argmin}_{a \in Act}\, \mathcal{T}_A(s)$.

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

4.2 Converting to decision tree

With Definition **??** we can now consider how to construct a single decision tree $\mathcal{T}$ so that $\pi(s) = \text{argmin}_{a \in Act}\, \mathcal{T}_A(s) = \mathcal{T}(s)$ 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 $A = \text{argmin}_{a \in Act}\, T_A(\lambda(l))$ 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 $A \in Act$ 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.

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

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.

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 2a.
- 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 2b.
- 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 2.

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.

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 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

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

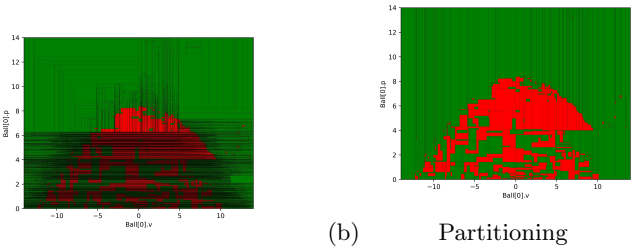
Third, the decreasement in the number of paths after empirically pruning the tree indicates that many areas of the state space are never visited in practice. It is also remarkable, that by utilizing this sample data we can further reduce the already minimized tree by a factor of 4 without sacrificing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

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



(a) Original paritioning

(b) Partitioning after `MaxPartitions`.

Fig. 2: 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'.

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.

The results when applying the generated strategies to the model in UPPAAL are given in Table 3 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

Table 3: 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

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. A. David, P. G. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist, "Uppaal stratego," in *TACAS*, 2015.
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

```
1: procedure MakeNewNode(var, bound, action, isHigh, prevNode)
2:     node ← Node(var, bound)
3:     if isHigh then
4:         node.high ← Leaf(action, ∞, State(·))
5:     else
6:         node.low ← Leaf(action, ∞, State(·))
7:     if prevNode is not None then
8:         if prevNode.low is None then
9:             prevNode.low ← node
10:        else
11:            prevNode.high ← node
12:    return node

1: function MakeRoot(Leaf(a_0, q_0, S_0))
2:     rootNode ← None
3:     prevNode ← None
4:     for (l_{0,i}, u_{0,i}) in S_0 do
5:         if u_{0,i} < ∞ then
6:             prevNode ← MakeNewNode(V_i, u_{0,i}, a_0, True, prevNode)
7:             if rootNode is None then
8:                 rootNode ← prevNode
9:         if l_{0,i} > −∞ then
10:            prevNode ← MakeNewNode(V_i, l_{0,i}, a_0, False, prevNode)
11:            if rootNode is None then
12:                rootNode ← prevNode
13:    if prevNode.low is None then
14:        prevNode.low ← Leaf(a_0, q_0, S_0)
15:    else
16:        prevNode.high ← Leaf(a_0, q_0, S_0)
17:    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

```
1: function PUT(root, Leaf(a, q, S) )
2:     if root is Node then
3:         return PUTATBRANCHNODE(root, Leaf(a, q, S))
4:     else                                              ▷ root is a Leaf
5:         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

```
1: function PUTATBRANCHNODE(Node(V_k, b, low, high), Leaf(a_i, q_i, S_i) )
2:     if l_{i,k} < b then
3:         S'_i ← SETUPPER(S_i, V_k, b)
4:         low ← PUT(low, Leaf(a_i, q_i, S'_i))
5:     if u_{i,k} > b then
6:         S'_i ← SETLOWER(S_i, V_k, b)
7:         high ← PUT(high, Leaf(a_i, q_i, S'_i))
8:     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

```
1: procedure Split(action, q, var, bound, state)
2:     highState ← SetLower(state, var, bound)
3:     lowState ← SetUpper(state, var, bound)
4:     high ← Leaf(action, q, highState)
5:     low ← Leaf(action, q, lowState)
6:     return Node(var, bound, low, high)

1: function PutAtLeafNode(Leaf(a_t, q_t, S_t), Leaf(a_i, q_i, S_i))
2:     if q_t ≤ q_i then
3:         return Leaf(a_t, q_t, S_t)

4:     for (l_{t,j}, u_{t,j}) in S_t do
5:         if l_{t,j} < l_{i,j} then
6:             newNode ← Split(a_t, q_t, V_j, l_{i,j})
7:             return Put(newNode, Leaf(a_i, q_i, S_i))

8:         else if u_{t,j} > u_{i,j} then
9:             newNode ← Split(a_t, q_t, V_j, u_{i,j})
10:            return Put(newNode, Leaf(a_i, q_i, S_i))

11:    return Leaf(a_i, q_i, S_i)
```