TTree: Tree-Based State Generalization with Temporally Abstract Actions

William T. B. Uther and Manuela M. Veloso

Computer Science Department,
Carnegie Mellon University,
Pittsburgh, PA 15213 USA
{uther, veloso}@cs.cmu.edu

Abstract. In this paper we describe the Trajectory Tree or TTree algorithm. TTree takes a set of policies and pieces them together to solve a Semi-Markov Decision Problem (SMDP). The algorithm uses a learned tree based discretization of the state space as an abstract state description and both supplied and generated policies as temporally abstract actions. It uses a generative model of the world to sample the transition function for the abstract SMDP. TTree then finds a policy for the abstract SMDP. In this paper we present the algorithm and some detailed examples of its execution. Furthermore we present empirical comparisons to other SMDP algorithms showing the effectiveness of our algorithm.

1 Introduction

Both Markov Decision Processes (MDPs) and Semi-Markov Decision Processes (SMDPs), summarized in [1], are important formalisms for planning in stochastic domains. SMDPs, like MDPs, suffer from exponential state explosion. The number of states that need to be considered by an algorithm learning a policy for an SMDP is exponential in the number of state dimensions that describe the problem.

A number of techniques have been used to solve large MDPs. These techniques can be broken into two main classes. *State abstraction* refers to the technique of grouping many states together and treating them as one, e.g. [2, 3]. *Temporal abstraction* refers to techniques that group sequences of actions together to form abstract actions which move through large regions of state space, e.g. [4–6]. These techniques usually transform the MDP into a related SMDP.[1]

In this paper we introduce the Trajectory Tree, or TTree, algorithm, that uses both forms of abstraction. TTree uses a new format for defining temporal abstractions that

This research was sponsored by the United States Air Force under Agreement Nos. F30602-00-2-0549 and F30602-98-2-0135. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force, or the US Government.

[1] Using a function approximator for the value function, e.g. [7, 8], can, in theory, subsume both state and temporal abstraction. The authors are unaware of any of these techniques that, in practice, achieve significant temporal abstraction.

does not require the definition of end points, unlike some other formats. Given a set of abstract actions, TTree first generates some additional abstract actions from the base level actions of the domain. TTree then alternates learning a tree based discretization of the state space and learning a policy for an abstract SMDP using the tree as an abstract state representation. In [9] we give a proof that TTree will converge to the optimal policy. In this paper we give a thorough description of the behavior of the algorithm. Moreover we present empirical results showing TTree is an effective anytime algorithm in practice.

2 Styles of Temporal Abstraction

There are two main styles of temporal abstraction described in the literature. The first style, e.g. [5, 6], has a strict, pre-defined hierarchy of actions. Each action can use only the actions in the next level of the hierarchy to achieve its goal. Often this hierarchy includes some state abstraction related to the temporal abstraction. Hengst [10] has recently generated hierarchies of a limited form by finding a fixed ordering of the state variables.

The second main style [4], revolves around defining the end point(s) of a macro action, or *option*. After these end points have been defined, policies can be learnt that achieve them. These policy/end point pairs can then be used as base level actions in an SMDP. Once an option is selected to be executed, its policy is followed until an end point is reached. The combined series of base level actions is treated as one, temporally extended, action by the SMDP. McGovern and Barto [11] have recently managed to automatically find options for the class of endpoints defined by ‘bottlenecks’ in the state space.

In this paper we introduce a style of temporal abstraction similar to options, but we remove the requirement for defining the end points of the option. Our abstract actions are simply policies. It is up to the algorithm using these abstract actions to decide in which regions of the state space each abstract action is followed. In fact, arbitrary controllers can be plugged in to the algorithm, but we do not consider non-markovian controllers in the proof of correctness or experiments.

Consider, for example, a robot that walks through a maze. This robot is legged, and the walking motion is non-trivial. One might imagine a set of four policies, each of which walks the robot in one the four cardinal directions, north, south, east and west. Each of these policies depends only upon the state variables that define the leg position, and not upon the state variables defining the position of the robot in the maze. An algorithm solving the walking robot in a maze problem need only learn which policy to use in which region of the maze.

We have recently published work on decomposing policies in a supervised learning setting [12]. We expect that this style of decomposition will form useful abstract actions of the form expected by TTree, however in this paper all the domain specific macros used in the experiments were generated by hand. We should also note that it is easy to convert an option into a policy by discarding the end points. If no action is specified in a state then a uniform distribution over the actions is used.

3 Definitions

An SMDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, P, R \rangle$.[2] $\mathcal{S}$ is the set of states. $\mathcal{A}$ is the set of actions. $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \Re \to [0, 1]$ is a joint probability distribution over both next-states and times, defined for each state action pair. $R : \mathcal{S} \times \mathcal{A} \to \Re$ defines the expected reward for performing an action in a state.[3]

Our agent starts in a state. It then performs an action. That action takes a length of time to move the agent to a new state, the time and resulting state determined by P. The agent gets reward for the transition defined by R.

Our goal is to learn a policy, $\pi : \mathcal{S} \to \mathcal{A}$, that maps from states to actions. In particular we want the policy, π^*, that maximizes a sum of rewards. To keep this sum of rewards bounded, we will introduce $\gamma \in (0, 1)$. The goal is to find a policy that maximizes $\sum_{t=0}^{\infty} \gamma^t r_t$ where r_t is the reward our agent receives at time t.

We can then define the following standard functions:

$$\begin{aligned} Q(s,a) &= R(s,a) + \\ &\qquad \sum_{s' \in \mathcal{S}} \int_{t=0}^{\infty} P_{s,a}(s',t) \gamma^t V(s') \, \mathrm{d}t && (1) \\ V(s) &= Q\left(s, \pi(s)\right) && (2) \\ \pi^*(s) &= \operatorname*{argmax}_{a \in A} Q^*(s,a) && (3) \end{aligned}$$

In addition, we will define the following T function. This function is defined over a set of states $\mathcal{S}' \subset \mathcal{S}$. It measures the discounted sum of reward for following the given action until the agent leaves $\mathcal{S}'$, then following the optimal policy.

$$\begin{aligned} T_{\mathcal{S}'}(s,a) &= R(s,a) + && (4) \\ &\qquad \sum_{s' \in \mathcal{S}'} \int_{t=0}^{\infty} P_{s,a}(s',t) \gamma^t T_{\mathcal{S}'}(s',a) \, \mathrm{d}t + && (5) \\ &\qquad \sum_{s' \in (\mathcal{S} - \mathcal{S}')} \int_{t=0}^{\infty} P_{s,a}(s',t) \gamma^t V(s') \, \mathrm{d}t && (6) \end{aligned}$$

We will write $T(s,a)$ instead of $T_{\mathcal{S}'}(s,a)$ when $\mathcal{S}'$ is the set of states that embed into the abstract state containing s.

We will assume that instead of being given P and R, our agent is given a *generative model* of the world, e.g. [13]. This is a function, $G : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \Re \times \Re$, that takes a state and an action and returns a next state, a time and a reward for the transition. The returned values are sampled from P and R. This description of the world is more

[2] P is often known as $\mathcal{Q}$ in the SMDP literature, however as we are using Q elsewhere, we are using MDP notation to avoid confusion.

[3] R can also depend upon both next state and time for the transition, but as these in turn depend only upon the state and action, they fall out of the expectation.

limited than P and R in that the agent can only get samples and does not know exact probabilities. However, this model is more powerful than having the agent in the real world as the agent can sample from any state and action it chooses.

4 The TTree Algorithm

TTree uses a tree to divide the world into abstract states. Each leaf in the tree corresponds to an abstract state. We extend previous tree-based discretization algorithms by using abstract actions rather than base-level actions in each abstract state. The abstract actions are policies in the original SMDP. There is a base set of abstract actions automatically generated from the low level actions. Additional abstract actions might be learned or supplied by the user. Together the abstract state and abstract actions form an abstract SMDP.

Each base level action has a corresponding, automatically generated, abstract action which executes that base action in every state. We also generate a random abstract action which picks uniformly among the base level actions in every state. The random abstract action provides additional exploration by executing a random walk through the state space.

TTree starts with the entire state mapped to a single leaf in the tree. It then loops gathering data and improving the accuracy of its abstract SMDP. An abstract state is divided when TTree estimates that the $T(s, a)$ function is not constant across that state. This is detected through the gathering of t values which are a stochastic approximation of T. TTree divides a leaf if it finds that the t sample distribution varies across that leaf for any action.

Each t sample has a corresponding start state. From that state, the generative model is used with each of the abstract actions in turn to form a set of trajectories through the low level state space. The trajectories run until they leave the current abstract state, or they have run long enough that the discounted reward value has converged, or they reach a deterministic transition allowing the trajectory value to be calculated rather than sampled or an absorbing state is reached. The start state, end state, abstract action, discounted reward and total time for each trajectory are recorded. Each sampled trajectory is a sampled transition in the abstract SMDP.

The trajectory can be used to stochastically approximate the T function. The reward for the first step of a trajectory is explicitly in the T function, equation 4. Rather than calculate an expectation over all next states and times, the trajectory samples a next state and time, and then loops instead of recursing, equation 5. The third part of the T function, equation 6, has an expectation that is sampled by the last step of the trajectory. The value function in this third part is approximated by the value of the corresponding abstract state.

The trajectories serve a second purpose besides stochastically approximating T. The abstract transition function is defined using the trajectories as abstract transitions. When our agent enters an abstract state, we assume it is transported to the start of a random trajectory for the abstract action it wishes to follow. The agent then moves to the abstract state that the trajectory ends in, taking time equal to the time taken by the trajectory and gaining reward equal to the discounted reward gained by the trajectory.

Table 1. Procedure $\mathrm{TTree}(\mathcal{S}, \hat{\mathcal{A}}, G(s, a))$

```
 1: $root \leftarrow$ a new leaf containing $\mathcal{S}$
 2: loop
 3:   $\mathcal{S}_a \leftarrow \{s_1, \ldots, s_{N_a}\}$ sampled from $\mathcal{S}$
 4:   for all $s \in \mathcal{S}_a$ do
 5:     $\mathrm{SampleTrajectory}(\hat{\mathcal{A}}, s)$ {see Procedure $\mathrm{SampleTrajectory}(\hat{\mathcal{A}}, s_{start})$ in Table 2}
 6:   end for
 7:   $\mathrm{UpdateAbstractSMDP}()$ {see Procedure $\mathrm{UpdateAbstractSMDP}()$ in Table 3}
 8:   $\mathcal{D} \leftarrow \emptyset$ {Reset split data set}
 9:   for all leaves $l$ and associated points $p$ do
10:     $t \leftarrow \emptyset$ {$t$ is a new array of size $|\hat{\mathcal{A}}|$}
11:     for all trajectories in $p$, $\langle a, s_{stop}, t_{total}, r_{total} \rangle$ do
12:       $l_{stop} \leftarrow \mathrm{LeafContaining}(s_{stop})$
13:       $t[a] \leftarrow t[a] + (r_{total} + \gamma^{t_{total}} V(l_{stop}))/N_t$
14:     end for
15:     $s \leftarrow s_{start}$ in $p$
16:     $\mathcal{D} \leftarrow \mathcal{D} \cup \langle s, t \rangle$ {add $t$-values to data set}
17:     $k \leftarrow \mathrm{argmax}_a\, t[a]$
18:     $\mathcal{D} \leftarrow \mathcal{D} \cup \langle s, k \rangle$ {add best action to data set}
19:   end for
20:   for all new splits in the tree do
21:     $\mathrm{EvaluateSplit}(\mathcal{D})$ {Use the splitting criterion to evaluate this split to see if either $t$, for
        any action, or $k$ varies across a leaf}
22:   end for
23:   if $\mathrm{ShouldSplit}(\mathcal{D})$ then {Evaluate the best split using the stopping criterion}
24:     Introduce best split into tree
25:     Throw out all sample points, $p$, in the leaf that was split
26:   end if
27: end loop
```

The complete algorithm is shown as Procedure $\mathrm{TTree}(\mathcal{S}, \hat{\mathcal{A}}, G(s, a))$ in Table 1, where $\mathcal{S}$ is the set of low level states, $\hat{\mathcal{A}} = \{\pi^0(s), \ldots, \pi^n(s)\}$ is the set of policies for the abstract actions (including the automatically generated ones), and $G(s, a)$ is the generative model. The various constants referred to are defined in Table 4.

4.1 Sampling the transition function

Trajectory sampling is used in many policy gradient ascent methods of solving large (S)MDPs, e.g. [13]. Our sampling is simpler in that we do not require gradients from our samples. We also add extra stopping criteria for the trajectories. In particular, the trajectory ends if it reaches a new leaf.

Ng and Jordan [13] carefully define the entropy source for each trajectory. Our implementation also does this, although we believe it unimportant for in our experiments as we used a deterministic domain. We refer the reader to their paper for details.

Table 2. Procedure $\mathrm{SampleTrajectory}(\hat{\mathcal{A}}, s_{start})$

```
1: Initialize new sample point, $p$, at $s_{start}$
2: $l \leftarrow \mathrm{LeafContaining}(s_{start})$
3: **for all** policies $\pi^i(s) \in \hat{\mathcal{A}}$ **do**
4:     **for** $j = 1$ to $N_t$ **do**
5:         $s \leftarrow s_{start}$
6:         $t_{total} \leftarrow 0$, $r_{total} \leftarrow 0$
7:         **repeat**
8:             $\langle s, t, r \rangle = G(s, \pi^i(s))$
9:             $t_{total} \leftarrow t_{total} + t$
10:            $r_{total} \leftarrow r_{total} + \gamma^{t_{total}} r$
11:        **until** $s \notin l$, **or**
           $t_{total} >$ MAXTIME, **or**
           $\langle s, *, * \rangle = G(s, \pi^i(s))$ is deterministic and $s = s'$, **or**
           $s$ is an absorbing state
12:        **if** the trajectory stopped because of a deterministic self transition **then**
13:            $r_{total} \leftarrow r_{total} + \gamma^{(t_{total}+t)} r/(1-\gamma^t)$
14:        **end if**
15:        **if** the trajectory stopped because the final state was absorbing, **or** because of a deter-
           ministic self transition **then**
16:            $t_{total} \leftarrow \infty$
17:        **end if**
18:        $s_{stop} \leftarrow s$
19:        Add $\langle a, s_{stop}, t_{total}, r_{total} \rangle$ to the trajectory list in $p$
20:    **end for**
21: **end for**
```

4.2 Growing the tree

In Procedure $\mathrm{TTree}(\mathcal{S}, \hat{\mathcal{A}}, G(s,a))$ (Table 1), we mention testing for good divisions using a splitting criterion. A splitting criterion is a statistical test that determines if the t values on either side of the recently introduced split are actually different, and allows the magnitude of the differences to be ranked. In [3] we used a non-parametric test, the Kolmogorov-Smirnov test. The results in this paper are with a Minimum Description Length based test, described in [12].

In the algorithm described we test $t[a], \forall a$, and $k = \mathrm{argmax}_a\, t[a]$ with the splitting criterion. The proof of correctness only requires testing t. We found that biasing the test with k gave a small improvement in empirical results.

We make no claims about which test should be used. The algorithm seems to be fairly robust to different tests. Non-parametric tests seem to perform better for separating t values. The Minimum Description Length test makes it easy to combine the tests on t and k.

The other important part of tree growing is the stopping criterion. Procedure $\mathrm{TTree}(\mathcal{S}, \hat{\mathcal{A}}, G(s,a))$ (Table 1) will not introduce a split if the stopping criterion is fulfilled, but keeps looping gathering more data. The experimental results in this paper are with a Minimum Description Length stopping criterion. We have found that the algorithm tends to get very good results long before the stopping criterion is met. The

Table 3. Procedure $\mathrm{UpdateAbstractSMDP}()$

```
1: for all leaves $l$ with fewer than $N_l$ sample points do
2:     $\mathcal{S}_a \leftarrow \{s_1, \ldots, s_{N_a}\}$ sampled from $l$
3:     for all $s \in \mathcal{S}_a$ do
4:         $\mathrm{SampleTrajectory}(\hat{\mathcal{A}}, s)$ {see Procedure $\mathrm{SampleTrajectory}(\hat{\mathcal{A}}, s_{start})$ in Table 2}
5:     end for
6: end for
7: $\mathcal{P} \leftarrow \emptyset$ {Reset abstract transition count}
8: for all leaves $l$ and associated points $p$ do
9:     for all trajectories, $\langle a, s_{stop}, t_{total}, r_{total}\rangle$, in $p$ do
10:        $l_{stop} \leftarrow \mathrm{LeafContaining}(s_{stop})$
11:        $\mathcal{P} \leftarrow \mathcal{P} + \langle l, a, l_{stop}, t_{total}, r_{total}\rangle$
12:    end for
13: end for
14: Transform $\mathcal{P}$ into transition probabilities
15: Solve the SMDP
```

Table 4. Constants in the TTree algorithm

Constant	Definition
N_a	The number of points sampled from the entire space each iteration
N_l	The minimum number of points sampled in each leaf
N_t	The number of trajectories sampled per start point
MAXTIME	The number of time steps before a trajectory value is assumed to have converged. Often chosen to keep $\gamma^{\mathrm{MAXTIME}} r/(1-\gamma^t) < \epsilon$, where r and t are the largest reward and smallest time step, and ϵ is an acceptable error

outer loop in Table 1 is an infinite loop, although it is possible to modify the algorithm so that it stops when the stopping criterion is fulfilled. We have been using the algorithm as an anytime algorithm.

5 An example TTree execution

This example describes TTree running in the Towers of Hanoi domain. This domain consists of 3 pegs, $\{P_0, P_1, P_2\}$, on which sit N disks, $\{D_A, D_B, \ldots\}$. Each disk is of a different size, D_A being the smallest, and they stack such that smaller disks always sit above larger disks. Some example states from the eight disc problem are shown in Table 5 a). There are six actions, shown in Table 5 b), which move the top disk on one peg to the top of one of the other pegs. An illegal action, trying to move a larger peg on top of a smaller peg, results in no change in the world. The object is to move all the disks to a specified peg; a reward of 50 is received this state. All base level actions take one time step. The decomposed representation we used has a boolean variable for each disk/peg pair. These variables are true if the disk is on the peg.

In our example we'll use the eight disc domain and $\gamma = 0.99$. We will also assume that TTree has been supplied with solutions to the three seven disc problems. These are policies that will move the seven smallest discs, referred to as the G stack, onto a

Table 5. States and actions in the Towers of Hanoi domain. a) A set of sample states. b) The set of base level actions

a)

Example state ID	Disks on peg P_0	P_1	P_2
s_1	D_A		$D_B D_C D_D D_E D_F D_G D_H$
s_2	$D_A D_D D_G$	$D_B D_E$	$D_C D_F D_H$
s_3	$D_A D_D D_G$	$D_B D_E D_H$	$D_C D_F$
s_4	$D_A D_D D_H$	$D_B D_E D_G$	$D_C D_F$

b)

Action	Move Disc From Peg	To Peg
a_0	P_0	P_1
a_1	P_0	P_2
a_2	P_1	P_2
a_3	P_1	P_0
a_4	P_2	P_1
a_5	P_2	P_0

Table 6. The abstract set of actions in the Towers of Hanoi domain

Action	Effect
Generated abstract actions	
A_0	Perform action a_0 in all states
A_1	Perform action a_1 in all states
A_2	Perform action a_2 in all states
A_3	Perform action a_3 in all states
A_4	Perform action a_4 in all states
A_5	Perform action a_5 in all states
A_r	Choose uniformly from $\{a_0, \ldots, a_5\}$ in all states
Supplied abstract actions	
A_{G0}	If stack G is on P_0 then choose uniformly from $\{a_0, \ldots, a_5\}$, otherwise follow the policy that will move stack G to P_0.
A_{G1}	If stack G is on P_1 then choose uniformly from $\{a_0, \ldots, a_5\}$, otherwise follow the policy that will move stack G to P_1.
A_{G2}	If stack G is on P_2 then choose uniformly from $\{a_0, \ldots, a_5\}$, otherwise follow the policy that will move stack G to P_2.

particular peg. These macros choose uniformly among the base level actions when the G stack is already on the appropriate peg. The complete set of abstract actions is shown in Table 6.

5.1 Building the abstract SMDP

Initially the algorithm has the entire state space as a single abstract state. The first thing the algorithm does is sample some trajectories.

Sampling Trajectories Initially it picks some random start points. We'll assume the algorithm chooses the points in Table 5 a).

Trajectories are then taken from each start points with each action. Consider the first sample point. From this location, action a_1 will solve the problem in a single step and receive reward. In our set of abstract actions, A_1 will do the same thing. In addition, A_{G2} will also perform that base level action and solve the problem. A_r has one chance in six of performing the right action in this state. Even if it does not solve the problem with its first move, it will be performing a random walk near the solution and so has a high probability of solving the problem.

None of the other abstract actions will solve the problem from s_1. In detail; A_{G0} and A_{G1} both move the agent away from the goal. Once they have moved the G stack to the appropriate peg, they will perform a random actions, and then fix the stack again if necessary. They will never reach the solution. There are three base level actions that are illegal. The corresponding abstract actions simply stop with deterministic self-transitions after one step. There are two other legal base actions apart from the one that solves the problem. These move the top disc on pegs 0 and 2 respectively to peg 1. These actions do not solve the problem. Furthermore, in the Towers of Hanoi domain performing any action leaves the agent in a state where performing the same action again is illegal, hence the generated abstract actions stop after two steps with a deterministic self-transition.

Having considered the resulting state of these trajectories, let us consider how long they take to run. A_1 and A_{G2} both solve the problem in one step. The solution state is absorbing, so both trajectories stop immediately. The generated deterministic abstract actions that perform 'illegal' actions and so perform deterministic self-transitions likewise generate single-transition trajectories. The two generated deterministic abstract actions that do not immediately perform 'illegal' actions will perform illegal actions for their second action; their trajectories will be two steps long. The 'random' abstract action will take a varying amount of time that stochastically depends upon the distance to the goal. A_{G0} and A_{G1} will each run for MAXTIME time steps.

Now we'll consider the second starting point. This starting point is similar to s_1 except that the problem cannot be solved in a single step from here. A_1 behaves like the other generated abstract actions and only A_{G2} solves the problem efficiently. A_r is the other abstract action that might solve the problem. If it does so, the trajectory is likely to be fairly long.

The third and fourth starting points behave in similar ways. Like the second starting point, there is no base level action that will solve the problem. Again, the random action might solve the problem, but it is unlikely, and we would expect that if a solution is found it is significantly longer than the solution found from start point 2. This is simply due to the fact that the length of the shortest solution from point 2 is shorter than the length of the shortest solution from points 3 and 4.

None of the supplied macros will solve the problem from points 3 or 4. If A_{G2} is selected, then D_H will never be moved onto P_2. If one of the other macros is selected, then D_H might be moved onto P_2 at some point but the G stack will never be moved to P_2.

We can generalize these results to classes of start points. Any start point with D_H on P_2 will reach the goal state using A_{G2}. Any start point with D_H disc on P_0 or P_1 will not be solved by any macro, with the possible exception of the random macro.

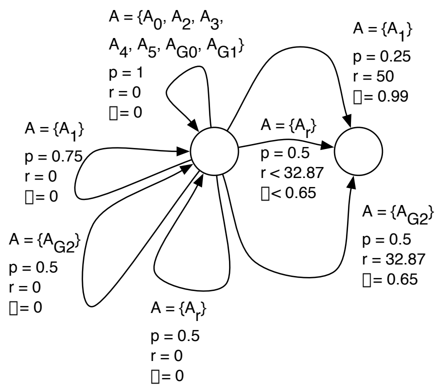


Fig. 1. Example abstract state transition diagram for the Towers of Hanoi domain

The abstract transition function Having sampled these trajectories, TTree uses them to construct an abstract SMDP. All the trajectories either reached the absorbing goal state or ended in the same abstract state they started in (there is only one state). The abstract transition function is as follows: The generated abstract action that solves the problem from start state one will solve the abstract SMDP with low probability, $p = 0.25$ with just the four example start points, and self transition the rest of the time. The random abstract action will solve the problem with decreasing frequency and increasing trajectory length as the start state moves away from the goal state. A_{G2} will solve the problem from s_1 and s_2. More generally, it will solve the problem from any start state where D_H is on P_2. It will self-transition the rest of the time. The other abstract actions will all self-transition. Figure1 shows the abstract state transition diagram. Note that the random transition is marked with the maximum possible discount factor and reward. We would expect that they would both be significantly lower for any particular sample. Also note that it takes A_{G2} 116 steps to solve the problem from state s_2, but only 1 step from state s_1. These lengths give $\gamma \approx 0.31$, and $\gamma = 0.99$, and $r \approx 15.7$ and $\gamma \approx 0.31$. The transition shown from the abstract state to the absorbing state uses weighted average values, $r \approx 32.87$ and $\gamma = 0.65$.

This abstract SMDP can then be solved (with the additional constraint that the absorbing state is assumed to have value 0). The resulting Q values are 0 for all abstract actions excepting A_{G2} and A_r. A_{G2} has a Q value of 32.87. A_r has a Q value smaller than that, the exact value depending upon the random trajectory length. These values cause the algorithm to select A_{G2} as the action to perform in all non-absorbing states. This policy is optimal in approximately $1/3$ of the states.

5.2 Refining the abstract state space

Having formed a policy, the algorithm now attempts to refine the state. Each of the trajectories has a value assigned to the start state, $t_s \leftarrow r + \gamma V(s')$ where r and γ are the recorded values for the transition, and $V(s')$ is the value associated with the resulting abstract state for the transition.

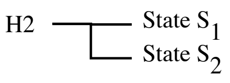


Fig. 2. Example abstract state tree for the Towers of Hanoi domain

In the current example, the value of the trajectory happens to equal the reward for the trajectory because all the transitions either have $\gamma = 0$ or transition into the absorbing state.

If we added more random start states then we would see a pattern similar to the classes of trajectories mentioned in section 5.1 above. The t values of the trajectories for A_r will be high near the goal state, and decrease quite quickly as you move away from the goal state. Any other trajectory starting with D_H on P_0 or P_1 will have $t = 0$. Any trajectory for A_{G2} starting with D_H on P_2 will reach the goal. These trajectories will have t values that increase as the trajectory length decreases. These values will be higher than the values for A_r. Other trajectories will have $t = 0$ except the the one that starts in state s_1 and uses action A_1 which will have $t = 50$.

For a reasonable statistical test, you need more than four start states. With a large enough set, the algorithm finds a distinction between states in which D_H is on P_2 and other states. The resulting abstract state tree is shown in Figure 2.

5.3 Further details

Having divided the state space, we now have two abstract states, S_1, and S_2. Moreover, we don't know when or even if the trajectories previously sampled cross between the states. We discard those trajectories and sample some new ones.

The generated abstract actions don't change their behavior much, so we'll concentrate on the supplied abstract actions.

In abstract state S_2, D_H is on P_2. From any base state in this abstract state, A_{G2} will reach the goal. Neither A_{G0} nor A_{G1} will reach the goal in this abstract state.

In abstract state S_1, the D_H is either on P_0 or P_1. If the D_H is on P_0 then A_{G0} will move other discs on top of D_H. Once the other discs are stacked on P_0, A_{G0} chooses a random action. This action can only move D_A or choose an illegal action.

However, if action A_{G1} is chosen then all the discs except D_H will be moved to P_1. Once in this state, A_{G1} chooses a random base-level action. This has a one in six chance of moving the D_H to P_2. That moves the problem into abstract state S_2, from whence it can be solved. The other two legal base level actions move D_A to another peg. A_{G1} will simply move it straight back and make more random moves until S_2 is reached. If the D_H is on P_1 then symmetric situation arrises - A_{G0} will move to state S_2.

Sampling the transition function and solving the abstract SMDP leads to the following policy: In S_2, A_{G2} is chosen and solves the problem, and in S_1, A_{G0} and A_{G1} are both equally effective - they each lead to state S_2 half the time and self transition the other half of the time.

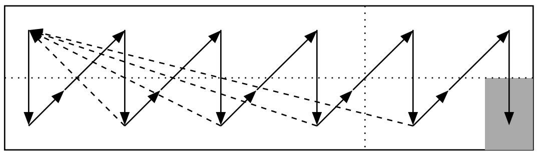


Fig. 3. Example of some trajectories

The t values in state S_1 clearly indicate the regions where A_{G0} and A_{G1} are effective. Dividing S_1 based on the location of D_H will separate those t values; this is the split that TTree introduces next.

At this point, the algorithm has three abstract states, one for each position of D_H. Once the transition function is sampled and the SMDP solved, there is one clearly superior abstract action in each state. Look ahead a little to the empirical results, this state decomposition corresponds to the peak in expected reward at 150 000 samples in figure 4 a). There are only two base level states where the action in non-optimal. These are the two states when D_H is *not* on P_2 and the other discs are stacked such that the supplied abstract action chooses randomly amongst the base level actions.

In addition to these two states not having optimal actions, the actions in the other states have not been *proven* optimal by the algorithm; the t values across the states are not constant. In order to find a provably optimal policy, TTree must continue dividing the state. This division of the state does not necessarily monotonically improve the policy found by TTree.

6 Example II

Consider the example shown in Figure 3. This figure shows a two dimensional state space. The grey region on the right hand side of the state space is absorbing with a reward of 50 for any action that enters that space. No other transitions in the space carry any reward. There are two actions. In the top half of the state space, both actions move the agent down in the same way. In the bottom half of the state space things are more complex. Action A_0, shown with solid arrows, moves the agent up and to the right. It does this in a way that takes two steps to reach the top half of the state space. In the bottom half of the state space action A_1 moves the agent to the top left of the state space.

Also shown as dotted lines are two potential divisions of this state space. One divides the right of the state space from the left of the state space, and the other divides the top from the bottom of the state space. With the state space undivided, then action A_0 has trajectories that reach the absorbing region on the right, whereas action A_1 loops without gaining any reward on the left hand edge of the state space. Action A_0 will be chosen as the optimal action in the abstract SMDP.

If we introduce the first of the two divisions and split the right hand side of the state from the left, then the optimal actions in the abstract SMDP do not change. Action A_0 still moves the agent to the right and to the reward, and action A_1 moves the agent to the left. In fact as long as the state is divided based on the x value of the state then it doesn't matter at which x value the split occurs - action A_0 will still be the optimal action in each of the new abstract states.

If we introduce the second of the two divisions, and divide the top from the bottom of the state space, then we see that the optimal policy in the abstract SMDP is worse after the split than before. To see this, let us construct the transition function for the abstract SMDP. There are two abstract states, the top state, S_t, and the bottom state, S_b. In S_t, both actions have the same transition function that move the agent either into S_b or to the reward in a single step. In S_b the actions have different effects. A_0 moves the agent into S_t in two steps, whereas A_1 takes only one step to reach S_t. Neither action gets any direct reward. Because S_t has positive value, and action A_1 gets the agent to S_t faster than action A_0, action A_1 is selected in S_b. This is far from optimal in the non-abstract problem.

What is happening is that information is lost on abstract state transitions. Often this is good because the information that is discarded is irrelevant. If the state is divided in the wrong way then important information can be lost. This is what happens in the example above. When the state is divided into top and bottom regions, the x location of the agent is lost each time there is a state transition. This makes the action that flips between abstract states faster look better than the one that flips between abstract states more slowly, but which moves the agent to the right.

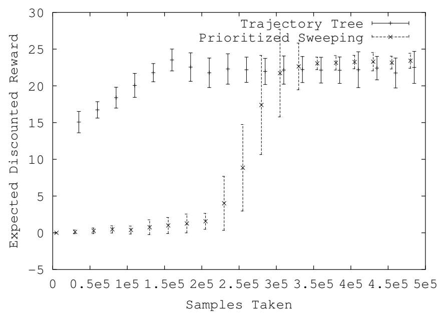
TTree usually makes the correct decision when dividing states. The t values estimate how much discounted reward an agent will receive for performing an action from a particular point. In the example above, the t values for action A_0 would decrease from right to left across the state. This is the largest variation and a split dividing the left of the state from the right of the state would be introduced first.

TTree can make mistakes when there are not enough t values sampled in a region, particularly if the region has high dimensionality. In this case, random variation can make less desirable splits be chosen first. It is important to note that even when this occurs, future splits will be introduced that allow the optimal policy to be found.

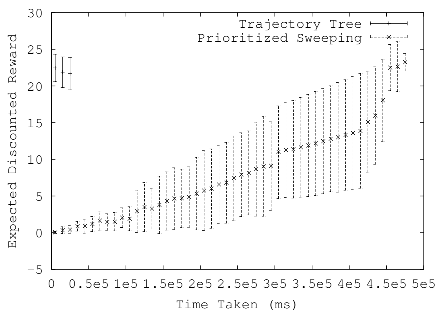
7 Empirical Results

We have evaluated TTree in the Towers of Hanoi domain. This domain is well known in the classical planning literature for the hierarchical structure of the solution; temporal abstraction should work well. We compared TTree against two other algorithms, a well known algorithm without abstraction, the Prioritized Sweeping algorithm [14] and an algorithm similar to TTree that performs only state abstraction, the Continuous U Tree algorithm [3].

Figure 4 shows a comparison of Prioritized Sweeping and TTree in the Towers of Hanoi domain. The data shown are the averages over 15 trials. The error bars show one standard deviation. For each trial the expected discounted reward was measured periodically by running 250 trajectories from random start points. This was recorded along with the number of samples the algorithm had taken from the generative model and a time stamp The y-axis in Figure 4 is the average expected discounted reward. The x-axis of (a) is the number of samples taken from the generative model. The x-



(a) Reward vs. Samples



(b) Reward vs. Time

Fig. 4. (a) A plot of Expected Reward vs. Number of Sample transitions taken from the world. (b) The same data plotted against time instead of the number of samples

axis of (b) is the time stamp In (b) the TTree data finishes significantly earlier than the Prioritized Sweeping data; TTree takes significantly less time per sample. Continuous U Tree results are not shown as that algorithm was unable to solve the problem.

The Towers of Hanoi domain had size $N = 8$. We had a discount factor, $\gamma = 0.99$. TTree was given policies for the three $N = 7$ problems. The TTree constants were, $N_a = 20, N_l = 20, N_t = 1$ and MAXSTEPS $= 400$. Prioritized Sweeping used Boltzmann exploration with carefully tuned parameters (γ was also tuned to help Prioritized Sweeping). The tuning of the parameters for Prioritized Sweeping took significantly longer than for TTree. In fact, the TTree parameters shown are the second set we tested.

We also tested Continuous U Tree and TTree on smaller problems without additional macros. TTree with only the generated abstract actions was able to solve more problems than Continuous U Tree. We attribute this to the fact that the Towers of Hanoi is particularly bad for U Tree style state abstraction. In U Tree the same action is always chosen in a leaf. However, it is never legal to perform the same action twice in a row in Towers of Hanoi. TTree is able to solve these problems because the, automatically generated, random abstract action allows it to gather more useful data than Continuous U Tree.

In addition, the transition function of the abstract SMDP formed by TTree is closer to what the agent will actually see in the real world than the transition function of abstract SMDP formed by Continuous U Tree. TTree samples the transition function assuming it might take a number of steps to leave the abstract state. Continuous U Tree assumes it will leave the abstract state in one step. This makes TTree a better anytime algorithm.

8 Conclusion

We have described the TTree algorithm for combining state and temporal abstraction in Semi-Markov Decision Problems. We have given some detailed examples of the execution of the algorithm on two separate tasks. The example in the Towers of Hanoi domain emphasizes the anytime nature of the algorithm. The second example shows some of the pitfalls of incorrect state abstraction and how TTree avoids those pitfalls. We have also supplied empirical results that show the algorithm is more effective in practice than another state abstraction algorithm, and that when extra macros are supplied, TTree is able to make use of these to further improve its results.

References

1. Puterman, M.L.: Markov Decision Processes : Discrete stochastic dynamic programming. Wiley series in probability and mathematical statistics. Applied probability and statistics section. John Wiley & Sons, New York (1994)
2. Chapman, D., Kaelbling, L.P.: Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia (1991) 726–731
3. Uther, W.T.B., Veloso, M.M.: Tree based discretization for continuous state space reinforcement learning. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), Madison, WI (1998) 769–774
4. Sutton, R.S., Precup, D., Singh, S.: Intra-option learning about temporally abstract actions. In: Machine Learning: Proceedings of the Fifteenth International Conference (ICML98), Madison, WI, Morgan Kaufmann (1998) 556–564
5. Dietterich, T.G.: The MAXQ method for hierarchical reinforcement learning. In: Machine Learning: Proceedings of the Fifteenth International Conference (ICML98), Madison, WI, Morgan Kaufmann (1998) 118–126
6. Parr, R.S., Russell, S.: Reinforcement learning with hierarchies of machines. In: Neural and Information Processing Systems (NIPS-98). Volume 10., MIT Press (1998)
7. Baird, L.C.: Residual algorithms: Reinforcement learning with function approximation. In Prieditis, A., Russell, S., eds.: Machine Learning: Proceedings of the Twelfth International Conference (ICML95), San Mateo, Morgan Kaufmann (1995) 30–37
8. Munos, R., Moore, A.W.: Variable resolution discretization for high-accuracy solutions of optimal control problems. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99). (1999)
9. Uther, W.T.B., Veloso, M.M.: TTree: Combining state and temporal abstraction in semi-markov decision problems. In: Proceedings of the Nineteenth InternationalConference on Machine Learning(ICML-2002). (2002) Under Submission.
10. Hengst, B.: Generating hierarchical structure in reinforcement learning from state variables. In Mizoguchi, R., Slaney, J.K., eds.: 6th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000). Volume 1886 of Lecture Notes in Computer Science., Springer (2000)
11. McGovern, A., Barto, A.G.: Automatic discovery of subgoals in reinforcement learning using diverse density. In: Proceedings of the Eighteenth International Conference on Machine Learning (ICML01). (2001) 361–368
12. Uther, W.T.B., Veloso, M.M.: The Lumberjack algorithm for learning linked decision forests. In Mizoguchi, R., Slaney, J.K., eds.: 6th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000). Volume 1886 of Lecture Notes in Computer Science., Springer (2000)
13. Ng, A.Y., Jordan, M.: PEGASUS: A policy search method for large MDPs and POMDPs. In: Uncertainty in Artificial Intelligence, Proceedings of the Sixteenth Conference. (2000)
14. Moore, A., Atkeson, C.G.: Prioritized sweeping: Reinforcement learning with less data and less real time. Machine Learning **13** (1993)