

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 chapter we describe the Trajectory Tree, or TTree, algorithm. TTree uses a small set of supplied policies to help 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 user supplied and auto-generated policies as temporally abstract actions. It uses a generative model of the world to sample the transition function for the abstract SMDP defined by those state and temporal abstractions, and then finds a policy for that abstract SMDP. This policy for the abstract SMDP can then be mapped back to a policy for the base SMDP, solving the supplied problem. In this chapter we present the TTree algorithm and give empirical comparisons to other SMDP algorithms showing its effectiveness.

1 Introduction

Both Markov Decision Processes (MDPs) and Semi-Markov Decision Processes (SMDPs), presented in [1], are important formalisms for agent control. They are used for describing the state dynamics and reward structure in stochastic domains and can be processed to find a policy; a function from the world state to the action that should be performed in that state. In particular, it is useful to have the policy that maximizes the sum of rewards over time. Unfortunately, the number of states that need to be considered when finding a policy is exponential in the number of dimensions that describe the state space. This exponential state explosion is a well known difficulty when finding policies for large (S)MDPs.

A number of techniques have been used to help overcome exponential state explosion and solve large (S)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 abstract state, *e.g.* [2–4]. *Temporal abstraction*

This research was sponsored by the United States Air Force under Agreement Nos. F30602-00-2-0549 and F30602-98-2-0135. The content of this chapter does not necessarily reflect the position of the funding agencies and no official endorsement should be inferred.

refers to techniques that group sequences of actions together and treat them as one abstract action, *e.g.* [5–9]. Using a function approximator for the value function, *e.g.* [10], can, in theory, subsume both state and temporal abstraction, but the authors are unaware of any of these techniques that, in practice, achieve significant temporal abstraction.

In this chapter we introduce the Trajectory Tree, or TTree, algorithm with two advantages over previous algorithms. It can both learn an abstract state representation and use temporal abstraction to improve problem solving speed. It also uses a new format for defining temporal abstractions that relaxes a major requirement of previous formats – it does not require a termination criterion as part of the abstract action.

Starting with a set of user supplied abstract actions, TTree first generates some additional abstract actions from the base level actions of the domain. TTree then alternates between 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 this chapter we give a description of the behavior of the algorithm. Moreover we present empirical results showing TTree is an effective anytime algorithm.

2 TTree

The goal of the TTree algorithm is to take an SMDP and a small collection of supplied policies, and discover which supplied policy should be used in each state to solve the SMDP. We wish to do this in a way that is more efficient than finding the optimal policy directly.

The TTree algorithm is an extension of the Continuous U Tree algorithm [3]. In addition to adding the ability to use temporal abstraction, we also improve the Continuous U Tree algorithm by removing some approximations in the semantics of the algorithm.

TTree uses policies as temporally abstract actions. They are solutions to sub-tasks that we expect the agent to encounter. We refer to these supplied policies as abstract actions to distinguish them from the solution – the policy we are trying to find. This definition of “abstract actions” is different from previous definitions. Other definitions of abstract actions in reinforcement learning, *e.g.* [5, 6], have termination criteria that our definition does not. Definitions of abstract actions in planning, *e.g.* [11], where an abstract action is a normal action with some pre-conditions removed, are even further removed from our definition. This ‘planning’ definition of abstract actions is closer to the concept of state abstraction than temporal abstraction.

Each of the supplied abstract actions is defined over the same set of base-level actions as the SMDP being solved. As a result, using the abstract actions gives us no more representational power than representing the policy through some other means, *e.g.* a table. Additionally, we ensure that there is at least one abstract action that uses each base-level action in each state, so that we have

no less representational power than representing the policy through some other means.

We noticed that a policy over the abstract actions has identical representational power to a normal policy over the states of an SMDP. However, if we have a policy mapping abstract states to abstract actions, then we have increased the representation power over a policy mapping abstract states to normal actions. This increase in power allows our abstract states to be larger while still representing the same policy.

3 Definitions

An SMDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, P, R \rangle$. \mathcal{S} is the set of world states. We will use s to refer to particular states, e.g. $\{s, s'\} \in \mathcal{S}$. We also assume that the states embed into an n -dimensional space: $\mathcal{S} \equiv \mathcal{S}^1 \times \mathcal{S}^2 \times \mathcal{S}^3 \times \dots \times \mathcal{S}^n$. In this chapter we assume that each dimension, \mathcal{S}^i , is discrete. \mathcal{A} is the set of actions. We will use a to refer to particular actions, e.g. $\{a_0, a_1\} \in \mathcal{A}$. Defined for each state action pair, $P_{s,a}(s', t) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R} \rightarrow [0, 1]$ is a joint probability distribution over both next-states and time taken. It is this distribution over the time taken for a transition that separates an SMDP from an MDP. $R(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ defines the expected reward for performing an action in a state.¹

The agent interacts with the world as follows. The agent knows the current state: the world is Markovian and fully observable. 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 determined by R . As the world is fully observable, the agent can detect the new world state and act again, etc.

Our goal is to learn a policy, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, 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 a multiplicative discount factor, $\gamma \in (0, 1)$. The goal is to find a policy that maximizes $\sum_{i=0}^{\infty} \gamma^{\tau_i} r_i$ where τ_i is the time that the agent starts its i^{th} action, and r_i is the reward our agent receives for its i^{th} action. Note that sometimes it will be useful to refer to a stochastic policy. This is a function from states to probability distributions over the actions.

We can then define the following standard functions:

$$Q(s, a) = R(s, a) + \sum_{s' \in \mathcal{S}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t V(s') dt \quad (1)$$

$$V(s) = Q(s, \pi(s)) \quad (2)$$

$$\pi^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q^*(s, a) \quad (3)$$

¹ 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.

We now introduce a related function, the T^π 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 policy π .

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

We assume that instead of sampling P and R directly from the world, our agent instead samples from a *generative model* of the world, *e.g.* [12]. This is a function, $G : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S} \times \mathbb{R} \times \mathbb{R}$, that takes a state and an action and returns a next state, a time and a reward for the transition. Our algorithm uses G to sample trajectories through the state space starting from randomly selected states.

4 The TTree Algorithm

TTree works by building an abstract SMDP that is smaller than the original, or base, SMDP. The solution to this abstract SMDP is an approximation to the solution to the base SMDP. The abstract SMDP is formed as follows: The states in the abstract SMDP, the *abstract states*, are formed by partitioning the states in the base SMDP; each abstract state corresponds to the set of base level states in one element of the partition. Each base level state falls into exactly one abstract state. Each action in the abstract SMDP, an *abstract action*, corresponds to a policy, or stochastic policy, in the base SMDP. The abstract transition and reward functions are found by sampling trajectories from the base SMDP.

We introduce some notation to help explain the algorithm. We use a bar over a symbol to distinguish the abstract SMDP from the base SMDP, *e.g.* \bar{s} vs. s , or $\bar{\mathcal{A}}$ vs. \mathcal{A} . This allows us a shorthand notation: when we have a base state, s , we use \bar{s} to refer specifically to the abstract state containing s . Also, when we have an abstract action \bar{a} we use $\pi_{\bar{a}}$ to refer to the base policy corresponding to \bar{a} and hence $\pi_{\bar{a}}(s)$ is the corresponding base action at state s . Additionally, we sometimes overload \bar{s} to refer to the set of base states that it corresponds to, *e.g.* $s \in \bar{s}$. Finally, it is useful, particularly in the proofs, to define functions that describe the base states within an abstract state, \bar{s} , but only refer to abstract states outside of \bar{s} . We mark these functions with a tilde. For example, we can define a function related to $T_{\mathcal{S}'}(s, a)$ in equation 4 above, $\tilde{T}_{\bar{s}}(s, a)$.

$$\begin{aligned}
\tilde{T}_{\bar{s}}(s, a) &= R(s, a) \\
&+ \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \tilde{T}_{\bar{s}}(s', a) dt \\
&+ \sum_{s' \in \bar{s}', \bar{s}' \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \bar{V}(\bar{s}') dt
\end{aligned} \tag{5}$$

Note that the $\tilde{T}_{\bar{s}}$ function is labelled with a tilde, and hence within the abstract state \bar{s} we refer to base level states, outside of \bar{s} we refer to the abstract value function over abstract states.

We describe the TTree algorithm from a number of different viewpoints. First we describe how TTree builds up the abstract SMDP, $\langle \bar{\mathcal{S}}, \bar{\mathcal{A}}, \bar{P}, \bar{R} \rangle$. Then we follow through the algorithm in detail, and finally we give a high level overview of the algorithm comparing it with previous algorithms.

4.1 Defining the Abstract SMDP

TTree uses a tree to partition the base level state space into abstract states. Each node in the tree corresponds to a region of the state space with the root node corresponding to the entire space. As our current implementation assumes state dimensions are discrete, internal nodes divide their region of state space along one dimension with one child for each discrete value along that dimension. It is a small extension to handle continuous and ordered discrete attributes in the same manner that Continuous U Tree [3] does. Leaf nodes correspond to abstract states; all the base level states that fall into that region of space are part of the abstract state.

TTree uses a set of abstract actions for the abstract SMDP. Each abstract action corresponds to a base level policy. There are two ways in which these abstract actions can be obtained; they can be supplied by the user, or they can be generated by TTree. In particular, TTree generates one abstract action for each base level action, and one additional ‘random’ abstract action. The ‘random’ abstract action is a base level stochastic policy that performs a random base level action in each base level state. The other generated abstract actions are degenerate base level policies: they perform the same base level action in every base level state: $\forall s; \pi_{\bar{a}_1}(s) = a_1, \pi_{\bar{a}_2}(s) = a_2, \dots, \pi_{\bar{a}_k}(s) = a_k$. These generated abstract actions are all that is required by the proof of correctness. Any abstract actions supplied by the user are hints to speed up the algorithm and are not required for correctness.

Informally, the abstract transition and reward functions are the expected result of starting in a random base state in the current abstract state and following a trajectory through the base states until we reach a new abstract state. To formalize this we define two functions. $\bar{R}_{\bar{s}}(s, \bar{a})$ is the expected discounted reward of starting in state s and following a trajectory through the base states using $\pi_{\bar{a}}$ until a new abstract state is reached. If no new abstract state is ever reached,

then \tilde{R} is the expected discounted reward of the infinite trajectory. $\tilde{P}_{s,\bar{a}}(\bar{s}', t)$ is the expected probability, over the same set of trajectories as $\tilde{R}_{\bar{s}}(s, \bar{a})$, of reaching the abstract state \bar{s}' in time t . If \bar{s}' is \bar{s} then we change the definition; when $t = \infty$, $\tilde{P}_{s,\bar{a}}(\bar{s}', t)$ is the probability that the trajectory never leaves state \bar{s} , and $\tilde{P}_{s,\bar{a}}(\bar{s}', t)$ is 0 otherwise.

We note that assigning a probability mass to $t = \infty$ is a mathematically suspect thing to do as it assigns a probability mass, rather than a density, to a single ‘point’ and, furthermore, that ‘point’ is ∞ . We justify the use of $\tilde{P}_{s,\bar{a}}(\bar{s}, \infty)$ as a notational convenience for “the probability we never leave the current state” as follows. We note that each time P is referenced with $\bar{s}' = \bar{s}$, it is then multiplied by γ^t , and hence for $t = \infty$ the product is zero. This is the correct value for an infinitely discounted reward. In the algorithm, as opposed to the proof, $t = \infty$ is approximated by $t \in (\text{MAXTIME}, \infty)$. MAXTIME is a constant in the algorithm, chosen so that γ^{MAXTIME} multiplied by the largest reward in the SMDP is approximately zero. The exponential discounting involved means that MAXTIME is usually not very large.

The definitions of \tilde{P} and \tilde{R} are expressed in the following equations:

$$\tilde{R}_{\bar{s}}(s, \bar{a}) = R(s, \pi_{\bar{a}}(s)) + \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s, \pi_{\bar{a}}(s)}(s', t) \gamma^t \tilde{R}_{\bar{s}}(s', \bar{a}) dt \quad (6)$$

$$\tilde{P}_{s,\bar{a}}(\bar{s}', t) = \begin{cases} \sum_{s'' \in \bar{s}'} P_{s, \pi_{\bar{a}}(s)}(s'', t) & : \bar{s}' \neq \bar{s} \\ + \sum_{s'' \in \bar{s}} \int_{t'=0}^t P_{s, \pi_{\bar{a}}(s)}(s'', t') \tilde{P}_{s'', \bar{a}}(\bar{s}', t - t') dt' & : \bar{s}' \neq \bar{s} \\ 0 & : \bar{s}' = \bar{s}, t \neq \infty \\ 1 - \sum_{s'' \neq \bar{s}} \int_{t=0}^{\infty} \tilde{P}_{s, \bar{a}}(s'', t) dt & : \bar{s}' = \bar{s}, t = \infty \end{cases} \quad (7)$$

Here \tilde{R} is recursively defined as the expected reward of the first step plus the expected reward of the rest of the trajectory. \tilde{P} also has a recursive formula. The first summation is the probability of moving from s to \bar{s}' in one transition. The second summation is the probability of transitioning from s to another state $s'' \in \bar{s}$ in one transition, and then continuing from s'' on to \bar{s}' in a trajectory using the remaining time. Note, the recursion in the definition of \tilde{P} is going to be bounded as we disallow zero time cycles in the SMDP.

We can now define the abstract transition and reward functions, \bar{P} and \bar{R} , as the expected values over all base states in the current abstract state of \tilde{P} and \tilde{R} :

$$\bar{P}_{\bar{s}, \bar{a}}(\bar{s}', t) = \mathbb{E}_{s \in \bar{s}} \tilde{P}_{s, \bar{a}}(\bar{s}', t) \quad (8)$$

$$\bar{R}(\bar{s}, \bar{a}) = \mathbb{E}_{s \in \bar{s}} \tilde{R}_{\bar{s}}(s, \bar{a}) \quad (9)$$

In English, \bar{P} and \bar{R} are the expected transition and reward functions if we start in a random base level state within the current abstract state and follow the supplied abstract action until we reach a new abstract state.

4.2 An Overview of the TTree Algorithm

In the algorithm \bar{P} and \bar{R} are not calculated directly from the above formulae. Rather, they are sampled by following trajectories through the base level state space as follows. A set of base level states is sampled from each abstract state. From each of these start states, for each abstract action, the algorithm uses the generative model to sample a series of trajectories through the base level states that make up the abstract state. In detail for one trajectory: let the abstract state we are considering be the state \bar{s} . The algorithm first samples a set of base level start states, $\{s_0, s_1, \dots, s_k\} \in \bar{s}$. It then gathers the set of base level policies for the abstract actions, $\{\pi_{\bar{a}_1}, \pi_{\bar{a}_2}, \dots, \pi_{\bar{a}_l}\}$. For each start state, s_i , and policy, $\pi_{\bar{a}_j}$, in turn, the agent samples a series of base level states from the generative model forming a trajectory through the low level state space. As the trajectory progresses, the algorithm tracks the sum of discounted reward for the trajectory, and the total time taken by the trajectory. The algorithm does not keep track of the intermediate base level states.

These trajectories have a number of termination criteria. The most important is that the trajectory stops if it reaches a new abstract state. The trajectory also stops if the system detects a deterministic self-transition in the base level state, if an absorbing state is reached, or if the trajectory exceeds a predefined length of time, MAXTIME. The result for each trajectory is a tuple, $\langle s_{start}, \bar{a}_j, s_{stop}, t, r \rangle$, of the start base level state, abstract action, end base level state, total time and total discounted reward.

We turn the trajectory into a sample transition in the abstract SMDP, *i.e.* a tuple $\langle \bar{s}_{start}, \bar{a}_j, \bar{s}_{stop}, t, r \rangle$. The sample transitions are combined to estimate the abstract transition and reward functions, \bar{P} and \bar{R} .

The algorithm now has a complete abstract SMDP. It can solve it using traditional techniques, *e.g.* [13], to find a policy for the abstract SMDP: a function from abstract states to the abstract action that should be performed in that abstract state. However, the abstract actions are base level policies, and the abstract states are sets of base level states, so we also have a function from base level states to base level actions; we have a policy for the base SMDP – an approximate solution to the supplied problem.

Having found this policy, TTree then looks to improve the accuracy of its approximation by increasing the resolution of the state abstraction. It does this by dividing abstract states – growing the tree. In order to grow the tree, we need to know which leaves should be divided and where they should be divided. A leaf should be divided when the utility of performing an abstract action is not constant across the leaf, or if the best action changes across a leaf.

We can use the trajectories sampled earlier to get point estimates of the T function defined in equation 4, itself an approximation of the utility of performing an abstract action in a given state. First, we assume that the abstract value

Table 1. Constants in the TTree algorithm

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

function, \bar{V} , is an approximation of the base value function, V . Making this substitution gives us the \tilde{T} function defined in equation 5. The sampled trajectories with the current abstract value function allow us to estimate \tilde{T} . We refer to these estimates as \hat{T} . For a single trajectory $\langle s_i, \bar{a}_j, s_{stop}, r, t \rangle$ we can find \bar{s}_{stop} and then get the estimate²:

$$\hat{T}_{\bar{s}}(s_i, \bar{a}_j) = r + \gamma^t \bar{V}(\bar{s}_{stop}) \quad (10)$$

From these $\hat{T}(s, \bar{a})$ estimates we obtain three different values used to divide the abstract state. Firstly, we divide the abstract state if $\max_{\bar{a}} \hat{T}(s, \bar{a})$ varies across the abstract state. Secondly, we divide the abstract state if the best action, $\arg\max_{\bar{a}} \hat{T}(s, \bar{a})$, varies across the abstract state. Finally, we divide the abstract state if $\hat{T}(s, \bar{a})$ varies across the state for any abstract action. It is interesting to note that while the last of these criteria contains a superset of the information in the first two, and leads to a higher resolution discretization of the state space once all splitting is done, it leads to the splits being introduced in a different order. If used as the sole splitting criterion, $\hat{T}(s, \bar{a})$ is not as effective as $\max_{\bar{a}} \hat{T}(s, \bar{a})$ for intermediate trees.

Once a division has been introduced, all trajectories sampled within the leaf that was divided are discarded, a new set of trajectories is sampled in each of the new leaves, and the algorithm iterates.

4.3 The TTree Algorithm in Detail

The TTree algorithm is shown in Procedure 1. The various constants referred to are defined in Table 1.

The core of the TTree algorithm is the trajectory. As described above, these are paths through the base-level states within a single abstract state. They are used in two different ways in the algorithm; to discover the abstract transition

² It has been suggested that it might be possible to use a single trajectory to gain \hat{T} estimates at many locations. We are wary of this suggestion as those estimates would be highly correlated; samples taken from the generative model near the end of a trajectory would affect the calculation of many point estimates.

Procedure 1 Procedure TTree($\mathcal{S}, \bar{\mathcal{A}}, G, \gamma$)

```
1:  $tree \leftarrow$  a new tree with a single leaf corresponding to  $\mathcal{S}$ 
2: loop
3:    $\mathcal{S}_a \leftarrow \{s_1, \dots, s_{N_a}\}$  sampled from  $\mathcal{S}$ 
4:   for all  $s \in \mathcal{S}_a$  do
5:     SampleTrajectories( $s, tree, \bar{\mathcal{A}}, G, \gamma$ ) {see Procedure 2}
6:   end for
7:   UpdateAbstractSMDP( $tree, \bar{\mathcal{A}}, G, \gamma$ ) {see Procedure 3}
8:   GrowTTree( $tree, \bar{\mathcal{A}}, \gamma$ ) {see Procedure 4}
9: end loop
```

function and to gather data about where to grow the tree and increase the resolution of the state abstraction. We first discuss how trajectories are sampled, then discuss how they are used.

Trajectories are sampled in sets, each set starting at a single base level state. The function to sample one of these sets of trajectories is shown in Procedure 2. The set of trajectories contains N_t trajectories for each abstract action. Once sampled, each trajectory is recorded as a tuple of start state, abstract action, resulting state, time taken and total discounted reward, $\langle s_{start}, \bar{a}, s_{stop}, t_{total}, r_{total} \rangle$, with s_{start} being the same for each tuple in the set. The tuples in the trajectory set are stored along with s_{start} as a sample point, and added to the leaf containing s_{start} .

The individual trajectories are sampled with the randomness being controlled [12, 14]. Initially the algorithm stores a set of N_t random numbers that are used as seeds to reset the random number generator. Before the j^{th} trajectory is sampled, the random number generator used in both the generative model and any stochastic abstract actions is reset to the j^{th} random seed. This removes some of the randomness in the comparison of the different abstract actions within this set of trajectories.

There are four stopping criteria for a sampled trajectory. Reaching another abstract state and reaching an absorbing state are stopping criteria that have already been discussed. Stopping when MAXTIME time steps have passed is an approximation. It allows us to get approximate values for trajectories that never leave the current state. Because future values decay exponentially, MAXTIME does not have to be very large to accurately approximate the trajectory value [12]. The final stopping criterion, stopping when a deterministic self-transition occurs, is an optimization, but it is not always possible to detect deterministic self-transitions. The algorithm works without this, but samples longer trajectories waiting for MAXTIME to expire, and hence is less efficient.

The TTree algorithm samples trajectory sets in two places. In the main procedure, TTree randomly samples start points from the entire base level state space and then samples trajectory sets from these start points. This serves to increase the number of trajectories sampled by the algorithm over time regardless of resolution. Procedure 3 also samples trajectories to ensure that there sampled trajectories in every abstract state to build the abstract transition function.

Procedure 2 Procedure SampleTrajectories($s_{start}, tree, \bar{\mathcal{A}}, G, \gamma$)

```
1: Initialize new trajectory sample point,  $p$ , at  $s_{start}$  { $p$  will store  $N_t$  trajectories for
   each of the  $|\bar{\mathcal{A}}|$  actions}
2: Let  $\{seed_1, seed_2, \dots, seed_{N_t}\}$  be a collection of random seeds
3:  $l \leftarrow \text{LeafContaining}(tree, s_{start})$ 
4: for all abstract actions  $\bar{a} \in \bar{\mathcal{A}}$  do
5:   let  $\pi_{\bar{a}}$  be the base policy associated with  $\bar{a}$ 
6:   for  $j = 1$  to  $N_t$  do
7:     Reset the random number generator to  $seed_j$ 
8:      $s \leftarrow s_{start}$ 
9:      $t_{total} \leftarrow 0, r_{total} \leftarrow 0$ 
10:    repeat
11:       $\langle s, t, r \rangle \leftarrow G(s, \pi_{\bar{a}}(s))$ 
12:       $t_{total} \leftarrow t_{total} + t$ 
13:       $r_{total} \leftarrow r_{total} + \gamma^{t_{total}} r$ 
14:    until  $s \notin l$ , or  $t_{total} > \text{MAXTIME}$ , or
       $\langle s', *, * \rangle = G(s, \pi_{\bar{a}}(s))$  is deterministic and  $s = s'$ , or  $s$  is an absorbing state
15:    if the trajectory stopped because of a deterministic self transition then
16:       $r_{total} \leftarrow r_{total} + \gamma^{(t_{total}+t)} r / (1 - \gamma^t)$ 
17:       $t_{total} \leftarrow \infty$ 
18:    else if the trajectory stopped because the final state was absorbing then
19:       $t_{total} \leftarrow \infty$ 
20:    end if
21:     $s_{stop} \leftarrow s$ 
22:    Add  $\langle s_{start}, \bar{a}, s_{stop}, t_{total}, r_{total} \rangle$  to the trajectory list in  $p$ 
23:  end for
24: end for
25: Add  $p$  to  $l$ 
```

Procedure 3 Procedure UpdateAbstractSMDP($tree, \bar{\mathcal{A}}, G, \gamma$)

```
1: for all leaves  $l$  with fewer than  $N_l$  sample points do
2:    $\mathcal{S}_a \leftarrow \{s_1, \dots, s_{N_a}\}$  sampled from  $l$ 
3:   for all  $s \in \mathcal{S}_a$  do
4:     SampleTrajectories( $s, tree, \bar{\mathcal{A}}, G, \gamma$ ) {see Procedure 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,  $\langle s_{start}, \bar{a}, s_{stop}, t_{total}, r_{total} \rangle$ , in  $p$  do
10:     $l_{stop} \leftarrow \text{LeafContaining}(tree, s_{stop})$ 
11:     $\mathcal{P} \leftarrow \mathcal{P} \cup \{ \langle l, \bar{a}, l_{stop}, t_{total}, r_{total} \rangle \}$ 
12:   end for
13: end for
14: Transform  $\mathcal{P}$  into transition probabilities
15: Solve the abstract SMDP
```

As well as using trajectories to find the abstract transition function, TTree also uses them to generate data to grow the tree. Here trajectories are used to generate three values. The first is an estimate of the T function, \hat{T} , the second is an estimate of the optimal abstract action, $\hat{\pi}(s) = \operatorname{argmax}_{\bar{a}} \hat{T}(s, \bar{a})$, and the third is the value of that action, $\max_{\bar{a}} \hat{T}(s, \bar{a})$. As noted above, trajectories are sampled in sets. The entire set is used by TTree to estimate the \hat{T} values and hence reduce the variance of the estimates.

As noted above (equation 10 – reprinted here), for a single trajectory, stored as the tuple $\langle s_{start}, \bar{a}_j, s_{stop}, r, t \rangle$, we can find \bar{s}_{stop} and can calculate \hat{T} :

$$\hat{T}_{\bar{s}_{start}}(s_{start}, \bar{a}) = r + \gamma^t \bar{V}(\bar{s}_{stop}) \quad (11)$$

For a set of trajectories all starting at the same base level state with the same abstract action we find a better estimate:

$$\hat{T}_{\bar{s}_{start}}(s_{start}, \bar{a}) = \frac{1}{N_t} \sum_{i=0}^{N_t} [r_i + \gamma^{t_i} \bar{V}(\bar{s}_{stop_i})] \quad (12)$$

This is the estimated expected discounted reward for following the abstract action \bar{a} starting at the base level state s_{start} , until a new abstract state is reached, and then following the policy defined by the abstract SMDP. If there is a statistically significant change in the \hat{T} value across a state for any action then we should divide the state in two.

Additionally, we can find which abstract action has the highest³ \hat{T} estimate, $\hat{\pi}$, and the value of that estimate, \hat{V} :

$$\hat{V}(s_{start}) = \max_{\bar{a}} \hat{T}(s_{start}, \bar{a}) \quad (13)$$

$$\hat{\pi}(s_{start}) = \operatorname{argmax}_{\bar{a}} \hat{T}(s_{start}, \bar{a}) \quad (14)$$

If the \hat{V} or $\hat{\pi}$ value changes across an abstract state, then we should divide that abstract state in two. Note that it is impossible for $\hat{\pi}(s)$ or $\hat{V}(s)$ to change without $\hat{T}(s, \bar{a})$ changing and so these extra criteria do not cause us to introduce any extra splits. However, they do change the order in which splits are introduced. Splits that would allow a change in policy, or a change in value function, are preferred over those that just improve our estimate of the Q function.

The division that maximizes the statistical difference between the two sides is chosen. Our implementation of TTree uses a Minimum Description Length test that is fully described in [8] to decide when to divide a leaf.

As well as knowing *how* to grow a tree, we also need to decide *if* we should grow the tree. This is decided by a stopping criterion. Procedure 4 does not introduce a split if the stopping criterion is fulfilled, but neither does it halt the algorithm. TTree keeps looping gathering more data. In the experimental results we use a Minimum Description Length stopping criterion. We have found that

³ Ties are broken in favor of the abstract action selected in the current abstract state.

Procedure 4 Procedure GrowTTree($tree, \bar{\mathcal{A}}, \gamma$)

```
1:  $\mathcal{D}^T \leftarrow \emptyset$  {Reset split data set.  $\mathcal{D}^T$  is a set of states with associated  $\hat{T}$  estimates.}
2:  $\mathcal{D}^\pi \leftarrow \emptyset$ 
3:  $\mathcal{D}^V \leftarrow \emptyset$ 
4: for all leaves  $l$  and associated points  $p$  do {a point contains a set of trajectories
   starting in the same state}
5:    $\hat{T}(s_{start}, \cdot) \leftarrow \emptyset$  { $\hat{T}(s_{start}, \cdot)$  is a new array of size  $|\bar{\mathcal{A}}|$ }
6:   for all trajectories in  $p$ ,  $\langle s_{start}, \bar{a}, s_{stop}, t, r \rangle$  do { $N_t$  trajectories for each of  $|\bar{\mathcal{A}}|$ 
   actions}
7:      $l_{stop} \leftarrow \text{LeafContaining}(tree, s_{stop})$ 
8:      $\hat{T}(s_{start}, \bar{a}) \leftarrow \hat{T}(s_{start}, \bar{a}) + (r + \gamma^t V(l_{stop}))/N_t$ 
9:   end for
10:   $\mathcal{D}^T \leftarrow \mathcal{D}^T \cup \{\langle s_{start}, \hat{T} \rangle\}$  {add  $\hat{T}$  estimates to data set}
11:   $\hat{V} \leftarrow \max_{\bar{a}} \hat{T}(s_{start}, \bar{a})$ 
12:   $\hat{\pi} \leftarrow \text{argmax}_{\bar{a}} \hat{T}(s_{start}, \bar{a})$ 
13:   $\mathcal{D}^V \leftarrow \mathcal{D}^V \cup \{\langle s, \hat{V} \rangle\}$  {add best value to data set}
14:   $\mathcal{D}^\pi \leftarrow \mathcal{D}^\pi \cup \{\langle s, \hat{\pi} \rangle\}$  {add best action to data set}
15: end for
16: for all new splits in the tree do
17:   EvaluateSplit( $\mathcal{D}^V \cup \mathcal{D}^\pi \cup \mathcal{D}^T$ ) {Use the splitting criterion to evaluate this split }
18: end for
19: if ShouldSplit( $\mathcal{D}^V \cup \mathcal{D}^\pi \cup \mathcal{D}^T$ ) then {Evaluate the best split using the stopping
   criterion}
20:   Introduce best split into tree
21:   Throw out all sample points,  $p$ , in the leaf that was split
22: end if
```

the algorithm tends to get very good results long before the stopping criterion is met, and we did not usually run the algorithm for that long. The outer loop in Procedure 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.

4.4 Discussion of TTree

Now that we have described the technical details of the algorithm, we look at the motivation and effects of these details. TTree was developed to fix some of the limitations of previous algorithms such as Continuous U Tree [3]. In particular we wanted to reduce the splitting from the edges of abstract states and we wanted to allow the measurement of the usefulness of abstract actions. Finally, we wanted to improve the match between the way the abstract policy is used and the way the abstract SMDP is modelled to increase the quality of the policy when the tree is not fully grown.

Introducing trajectories instead of transitions solves these problems. The \hat{T} values, unlike the q values in Continuous U Tree, vary all across an abstract state, solving the edge slicing issue. The use of trajectories allows us to measure

the effectiveness of abstract actions along a whole trajectory rather than only for a single step. Finally, the use of trajectories allows us to build a more accurate abstract transition function.

Edge slicing was an issue in Continuous U Tree where all abstract self-transitions with the same reward had the same q values, regardless of the dynamics of the self-transition. This means that often only the transitions out of an abstract state have different q values, and hence that the algorithm tends to slice from the edges of abstract states into the middle. TTree does not suffer from this problem as the trajectories include a measure of how much time the agent spends following the trajectory before leaving the abstract state. If the state-dynamics change across a state, then that is apparent in the \hat{T} values.

Trajectories allow us to select abstract actions for a state because they provide a way to differentiate abstract actions from base level actions. In one step there is no way to differentiate an abstract action from a base level action. Over multiple steps, this becomes possible.

Finally, trajectories allow a more accurate transition function because they more accurately model the execution of the abstract policy. When the abstract SMDP is solved, an abstract action is selected for each abstract state. During execution that action is executed repeatedly until the agent leaves the abstract state. This repeated execution until the abstract state is exited is modelled by a trajectory. This is different from how Continuous U Tree forms its abstract MDP where each step is modelled individually. TTree only applies the Markov assumption at the start of a trajectory, whereas Continuous U Tree applies it at each step. When the tree is not fully grown, and the Markov assumption inaccurate, fewer applications of the assumption lead to a more accurate model.

However, the use of trajectories also brings its own issues. If the same action is always selected until a new abstract state is reached, then we have lost the ability to change direction halfway across an abstract state. Our first answer to this is to sample trajectories from random starting points throughout the state, as described above. This allows us to measure the effect of changing direction in a state by starting a new trajectory in that state. To achieve this we require a generative model of the world. With this sampling, if the optimal policy changes halfway across a state, then the \hat{T} values should change. But we only get \hat{T} values where we start trajectories.

It is not immediately obvious that we can find the optimal policy in this constrained model. In fact, with a fixed size tree we usually can not find the optimal policy, and hence we need to grow the tree. With a large enough tree the abstract states and base level states are equivalent, so we know that expanding the tree can lead to optimality. However, it is still not obvious that the \hat{T} values contain the information we need to decide if we should keep expanding the tree; *i.e.* it is not obvious that there are no local maxima, with all the \hat{T} values equal within all leaves, but with a non-optimal policy. We prove that no such local maxima exist in Section 5 below.

The fact that we split first on $\hat{V} = \max_{\bar{a}} \hat{T}(\cdot, \bar{a})$ and $\hat{\pi} = \operatorname{argmax}_{\bar{a}} \hat{T}(\cdot, \bar{a})$ values before looking at all the \hat{T} values deserves some explanation. If you split

on \hat{T} values then you sometimes split based on the data for non-optimal abstract actions. While this is required for the proof in Section 5 (see the example in Section 5.2), it also tends to cause problems empirically [8]. Our solution is to only split on non-optimal actions when no splits would otherwise be introduced.

Finally, we make some comments about the random abstract action. The random abstract action has \hat{T} values that are a smoothed version of the reward function. If there is only a single point reward there can be a problem finding an initial split. The point reward may not be sampled often enough to find a statistically significant difference between it and surrounding states. The random abstract action improves the chance of finding the point reward and introducing the initial split. In some of the empirical results we generalize this to the notion of an abstract action for exploration.

5 Proof of Convergence

Some previous state abstraction algorithms [2, 3] have generated data in a manner similar to TTree, but using single transitions rather than trajectories. In that case, the data can be interpreted as a sample from a stochastic form of the Q -function (TTree exhibits this behavior as a special case when MAXTIME = 0). When trajectories are introduced, the sample values no longer have this interpretation and it is no longer clear that splitting on the sample values leads to an abstract SMDP with any formal relation to the original SMDP. Other state abstraction algorithms, *e.g.* [4], generate data in a manner similar to TTree but are known not to converge to optimality in all cases.

In this section, we analyze the trajectory values. We introduce a theorem that shows that splitting such that the \hat{T} values are equal for all actions across a leaf leads to the optimal policy for the abstract SMDP, $\bar{\pi}^*$, also being an optimal policy for the original SMDP. The complete proofs are available at [8]. We also give a counter-example for a simplified version of TTree showing that having constant trajectory values for only the highest valued action is not enough to achieve optimality.

5.1 Assumptions

In order to separate the effectiveness of the splitting and stopping criteria from the convergence of the SMDP solving, we assume optimal splitting and stopping criteria, and that the sample sizes, N_l and N_t , are sufficient. That is, the splitting and stopping criteria introduce a split in a leaf if, and only if, there exist two regions, one on each side of the split, and the distribution of the value being tested is different in those regions.

Of course, real world splitting criteria are not optimal, even with infinite sample sizes. For example, most splitting criteria have trouble introducing splits if the data follows an XOR or checkerboard pattern. Our assumption is still useful as it allows us to verify the correctness of the SMDP solving part of the algorithm independently of the splitting and stopping criteria.

This proof only refers to base level actions. We assume that the only abstract actions are the automatically generated degenerate abstract actions, and hence $\forall \bar{a}, \forall s, \pi_{\bar{a}}(s) = a$ and we do not have to distinguish between a and \bar{a} . Adding extra abstract actions does not affect the proof, and so we ignore them for convenience of notation.

Theorem 1. *If the \hat{T} samples are statistically constant across all states for all actions, then an optimal abstract policy is an optimal base level policy. Formally,*

$$\forall \bar{a} \in \bar{\mathcal{A}}, \forall \bar{s} \in \bar{\mathcal{S}}, \forall s_1 \in \bar{s}, \forall s_2 \in \bar{s}, \tilde{T}(s_1, \bar{a}) = \tilde{T}(s_2, \bar{a}) \Rightarrow \bar{\pi}^*(s_1) = \pi^*(s_1) \quad (15)$$

We first review the definition of \tilde{T} introduced in equation 5:

$$\begin{aligned} \tilde{T}_{\bar{s}}(s, a) &= R(s, a) \\ &+ \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \tilde{T}_{\bar{s}}(s', a) dt \\ &+ \sum_{s' \in \bar{s}', \bar{s}' \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \bar{V}(\bar{s}') dt \end{aligned} \quad (16)$$

This function describes the expected value of the \hat{T} samples used in the algorithm, assuming a large sample size. It is also closely related to the T function defined in equation 4; the two are identical except for the value used when the region defined by \mathcal{S}' or \bar{s} is exited. The T function used the value of a base level value function, V , whereas the \tilde{T} function uses the value of the abstract level value function, \bar{V} .

We also define functions $\tilde{V}_{\bar{s}}^*(s)$ and $\tilde{Q}_{\bar{s}}^*(s, a)$ to be similar to the normal V^* and Q^* functions within the set of states corresponding to \bar{s} , but once the agent leaves \bar{s} it gets a one-time reward equal to the value of the abstract state it enters, \bar{V} .

$$\begin{aligned} \tilde{Q}_{\bar{s}}^*(s, a) &= R(s, a) \\ &+ \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \tilde{V}_{\bar{s}}^*(s') dt \\ &+ \sum_{s' \in \bar{s}', \bar{s}' \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \bar{V}(\bar{s}') dt \end{aligned} \quad (17)$$

$$\tilde{V}_{\bar{s}}^*(s) = \max_a \tilde{Q}_{\bar{s}}^*(s, a) \quad (18)$$

Intuitively these functions give the value of acting optimally within \bar{s} , assuming that the values of the base level states outside \bar{s} are fixed.

We now have a spectrum of functions. At one end of the spectrum is the base Q^* function from which it is possible to extract the set of optimal policies

for the original SMDP. Next in line is the \tilde{Q}^* function which is optimal within an abstract state given the values of the abstract states around it. Then we have the \tilde{T} function which can have different values across an abstract state, but assumes a constant action until a new abstract state is reached. Finally we have the abstract \bar{Q}^* function which does not vary across the abstract state and gives us the optimal policy for the abstract SMDP.

The outline of the proof of optimality when splitting is complete is as follows. First, we introduce in Lemma 1 that \tilde{T} really is the same as our estimates, \hat{T} , for large enough sample sizes. We then show that, when splitting has stopped, the maximum over the actions of each of the functions in the spectrum mentioned in the previous paragraph is equal and is reached by the same set of actions. We also show that $\bar{Q}^* \leq Q^*$. This implies that an optimal policy in the abstract SMDP is also an optimal policy in the base SMDP. The proofs of the lemmas 1 and 3 are available at [8].

Lemma 1. *The \hat{T} samples are an unbiased estimate of \tilde{T} . Formally,*

$$E_{\substack{\text{trajectories starting at } s \\ \langle s, \bar{a}, s', r, t \rangle}} \hat{T}_{\bar{s}}(s, \bar{a}) = \tilde{T}_{\bar{s}}(s, a) \quad (19)$$

Lemma 2. $\forall s \in \bar{s}, \forall a, \tilde{Q}_{\bar{s}}^*(s, a) \geq \tilde{T}_{\bar{s}}(s, a)$

This is true by inspection. Equations 5 and 17 are reprinted here for reference:

$$\begin{aligned} \tilde{T}_{\bar{s}}(s, a) &= R(s, a) \\ &+ \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \tilde{T}_{\bar{s}}(s', a) dt \\ &+ \sum_{s' \in \bar{s}', s' \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \bar{V}(\bar{s}') dt \end{aligned} \quad (20)$$

$$\begin{aligned} \tilde{Q}_{\bar{s}}^*(s, a) &= R(s, a) \\ &+ \sum_{s' \in \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \tilde{V}_{\bar{s}}^*(s') dt \\ &+ \sum_{s' \in \bar{s}', s' \neq \bar{s}} \int_{t=0}^{\infty} P_{s,a}(s', t) \gamma^t \bar{V}(\bar{s}') dt \end{aligned} \quad (21)$$

Substituting $\tilde{V}_{\bar{s}}^*(s) = \max_a \tilde{Q}_{\bar{s}}^*(s, a)$ into equation 21 makes the two functions differ only in that \bar{Q} has a max where \tilde{T} does not. Hence $\bar{Q} \geq \tilde{T}$. *q.e.d.*

Lemma 3. *If $\tilde{T}_{\bar{s}}$ is constant across \bar{s} for all actions, then $\max_a \tilde{T}_{\bar{s}}(\cdot, a) = \max_a \tilde{Q}_{\bar{s}}^*(\cdot, a)$ and $\arg\max_a \tilde{T}_{\bar{s}}(\cdot, a) = \arg\max_a \tilde{Q}_{\bar{s}}^*(\cdot, a)$.*

Lemma 4. *If $\tilde{T}_{\bar{s}}$ is constant across the abstract state \bar{s} for all actions then $\bar{Q}(\bar{s}, a) = \tilde{T}_{\bar{s}}(s, a)$ for all actions.*

During the proof of lemma 1 we show,

$$\tilde{T}_{\bar{s}}(s, a) = \tilde{R}_{\bar{s}}(s, a) + \sum_{\bar{s}' \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \tilde{P}_{s,a}(\bar{s}', t) \gamma^t \bar{V}(\bar{s}') dt$$

Now,

$$\bar{Q}(\bar{s}, a) = \bar{R}(\bar{s}, a) + \sum_{\bar{s}' \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \bar{P}_{\bar{s},a}(\bar{s}', t) \gamma^t \bar{V}^{\pi}(\bar{s}') dt \quad (22)$$

Substituting equations 8 and 9,

$$= \mathbb{E}_{s \in \bar{s}} [\tilde{R}_{\bar{s}}(s, a)] + \sum_{\bar{s}' \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \mathbb{E}_{s \in \bar{s}} [\tilde{P}_{s,a}(\bar{s}', t)] \gamma^t \bar{V}^{\pi}(\bar{s}') dt \quad (23)$$

$$= \frac{1}{|\bar{s}|} \left[\sum_{s \in \bar{s}} \tilde{R}_{\bar{s}}(s, a) \right] + \sum_{\bar{s}' \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \frac{1}{|\bar{s}|} \left[\sum_{s \in \bar{s}} \tilde{P}_{s,a}(\bar{s}', t) \right] \gamma^t \bar{V}^{\pi}(\bar{s}') dt \quad (24)$$

$$= \frac{1}{|\bar{s}|} \sum_{s \in \bar{s}} \tilde{R}_{\bar{s}}(s, a) + \frac{1}{|\bar{s}|} \sum_{s \in \bar{s}} \sum_{\bar{s}' \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \tilde{P}_{s,a}(\bar{s}', t) \gamma^t \bar{V}^{\pi}(\bar{s}') dt \quad (25)$$

$$= \frac{1}{|\bar{s}|} \sum_{s \in \bar{s}} \left[\tilde{R}_{\bar{s}}(s, a) + \sum_{\bar{s}' \in \bar{\mathcal{S}}} \int_{t=0}^{\infty} \tilde{P}_{s,a}(\bar{s}', t) \gamma^t \bar{V}^{\pi}(\bar{s}') dt \right] \quad (26)$$

$$= \mathbb{E}_{s \in \bar{s}} \tilde{T}_{\bar{s}}(s, a) \quad (27)$$

Given that $\tilde{T}_{\bar{s}}(s, a)$ is constant across $s \in \bar{s}$, then $\forall s' \in \bar{s}, \mathbb{E}_{s \in \bar{s}} \tilde{T}_{\bar{s}}(s, a) = \tilde{T}_{\bar{s}}(s', a)$. *q.e.d.*

Lemma 5. *If $\tilde{T}_{\bar{s}}$ is constant across the abstract state \bar{s} for all actions, and $\bar{V}(\bar{s}') = V^*(s')$ for all base states in all abstract states \bar{s}' , $\bar{s}' \neq \bar{s}$, then $\bar{V}(\bar{s}) = V^*(s)$ in \bar{s} .*

Substituting $\bar{V}(\bar{s}') = V^*(s')$ for other states into equation 17, we see that $\tilde{Q}^* = Q^*$ for the current state and so $\tilde{V}^* = V^*$ for the current state. Also, $\text{argmax}_a \tilde{Q}^*(s, a) = \text{argmax}_a Q^*(s, a)$ and so the policies implicit in these functions are also equal.

Moreover, because $\tilde{T}_{\bar{s}}$ is constant across the current abstract state, we know, by Lemma 4, that $\bar{Q}(\bar{s}, a) = \tilde{T}_{\bar{s}}(s, a)$. For the same reason we also know by Lemma 3 that $\max_a \tilde{T}_{\bar{s}}(s, a) = \tilde{V}_{\bar{s}}^*(s)$.

$$\bar{Q}(\bar{s}, a) = \tilde{T}_{\bar{s}}(s, a) \quad (28)$$

$$\text{therefore } \bar{V}(\bar{s}) = \max_a \tilde{T}_{\bar{s}}(s, a) \quad (29)$$

$$= \tilde{V}_{\bar{s}}^*(s) \quad (30)$$

$$= V^*(s) \quad (31)$$

q.e.d.

Lemma 6. *If $\tilde{T}_{\bar{s}}$ is constant across each abstract state for each action, then setting $V^* = \bar{V}$ is a consistent solution to the Bellman equations of the base level SMDP.*

This is most easily seen by contradiction. Assume we have a tabular representation of the base level value function. We will initialize this table with the values from \bar{V} . We will further assume that $\tilde{T}_{\bar{s}}$ is constant across each abstract state for each action, but that our table is not optimal, and show that this leads to a contradiction.

As in lemma 5, because $\tilde{T}_{\bar{s}}$ is constant across the current abstract state, we know, by Lemma 4, that $\bar{Q}(\bar{s}, a) = \tilde{T}_{\bar{s}}(s, a)$. For the same reason we also know by Lemma 3 that $\max_a \tilde{T}_{\bar{s}}(s, a) = \tilde{V}_{\bar{s}}^*(s)$.

This means that our table contains $\tilde{V}_{\bar{s}}^*$ for each abstract state. Hence, there is no single base level state that can have its value increased by a single bellman update. Hence the table must be optimal.

This optimal value function is achieved with the same actions in both the base and abstract SMDPs. Hence any optimal policy in one is an optimal policy in the other. *q.e.d.*

5.2 Splitting on Non-Optimal Actions

We did not show above that the \tilde{T} and Q^* functions are equal for non-optimal actions. One might propose a simpler algorithm that only divides a state when \tilde{T} is not uniform for the action with the highest value, rather than checking for uniformity all the actions. Here is a counter-example showing this simplified algorithm does not converge.

Consider an MDP with three states, s_1 , s_2 and s_3 . s_3 is an absorbing state with zero value. States s_1 and s_2 are both part of a single abstract state, s_3 is in a separate abstract state. There are two deterministic actions. a_1 takes us from either state into s_3 with a reward of 10. a_2 takes us from s_1 to s_2 with a reward of 100, and from s_2 to s_3 with a reward of -1000 . Table 2 shows the \tilde{T} and Q^* values for each state when $\gamma = 0.9$. Note that even though the $T(s, a_1)$ values are constant and higher than the $T(s, a_2)$ values, the optimal policy does not choose action a_1 in both states.

Table 2. T , Q and V for sample MDP

Function	s_1	s_2
$Q(s, a_1)$	9	9
$T(s, a_1)$	9	9
$Q(s, a_2)$	108.1	-900
$T(s, a_2)$	-710	-900
$V(s)$	108.1	9

6 Empirical Results

We evaluated TTree in a number of domains. For each domain the experimental setup was similar. We compared mainly against the Prioritized Sweeping algorithm [13]. The reason for this is that, in the domains tested, Continuous U Tree was ineffective as the domains do not have much scope for normal state abstraction. It is important to note that Prioritized Sweeping is a *certainty equivalence* algorithm. This means that it builds an internal model of the state space from its experience in the world, and then solves that model to find its policy. The model is built without any state or temporal abstraction and so tends to be large, but, aside from the lack of abstraction, it makes very efficient use of the transitions sampled from the environment.

The experimental procedure was as follows. There were 15 learning trials. During each trial, each algorithm was tested in a series of epochs. At the start of their trials, Prioritized Sweeping had its value function initialized optimistically at 500, and TTree was reset to a single leaf. At each time step Prioritized Sweeping performed 5 value function updates. At the start of each epoch the world was set to a random state. The algorithm being tested was then given control of the agent. The epoch ended after 1000 steps were taken, or if an absorbing state was reached. At that point the algorithm was informed that the epoch was over. TTree then used its generative model to sample trajectories, introduce one split, sample more trajectories to build the abstract transition function, and update its abstract value function and find a policy. Prioritized Sweeping used its certainty equivalence model to update its value function and find a policy. Having updated its policy, the algorithm being tested was then started at 20 randomly selected start points and the discounted reward summed for 1000 steps from each of those start points. This was used to estimate the expected discounted reward for each agent’s current policy. These trajectories were not used for learning by either algorithm. An entry was then recorded in the log with the number of milliseconds spent by the agent so far this trial (not including the 20 test trajectories), the total number of samples taken by the agent so far this trial (both in the world and from the generative model), the size of the agent’s model, and the expected discounted reward measured at the end of the epoch. For Prioritized Sweeping, the size of the model was the number of visited state/action pairs divided by the number of actions. For TTree the size

of the model was the number of leaves in the tree. The trial lasted until each agent had sampled a fixed number of transitions (which varied by domain).

The data was graphed as follows. We have two plots in each domain. The first has the number of transitions sampled from the world on the x -axis and the expected reward on the y -axis. The second has time taken by the algorithm on the x -axis and expected reward on the y -axis. Some domains have a third graph showing the number of transitions sampled on the x -axis and the size of the model on the y -axis.

For each of the 15 trials there was a log file with an entry recorded at the end of each epoch. However, the number of samples taken in an epoch varies, making it impossible to simply average the 15 trials. Our solution was to connect each consecutive sample point within each trial to form a piecewise-linear curve for that trial. We then selected an evenly spaced set of sample points, and took the mean and standard deviation of the 15 piecewise-linear curves at each sample point. We stopped sampling when any of the log files was finished (when sampling with time on the x -axis, the log files are different lengths).

6.1 Towers of Hanoi

The Towers of Hanoi domain is well known in the classical planning literature for the hierarchical structure of the solution; temporal abstraction should work well. This domain consists of 3 pegs, on which sit N disks. Each disk is of a different size and they stack such that smaller disks always sit above larger disks. There are six actions 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 100 is received in this state. All base level actions take one time step, with $\gamma = 0.99$. 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.

The Towers of Hanoi domain had size $N = 8$. We used a discount factor, $\gamma = 0.99$. TTree was given policies for the three $N = 7$ problems, the complete set of abstract actions is shown in Table 3. 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.

Figure 1 shows a comparison of Prioritized Sweeping and TTree. In Figure 1b 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 problem has 24 state dimensions and Continuous U Tree was unable to find an initial split.

We also tested Continuous U Tree and TTree on smaller Towers of Hanoi 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 domain is particularly bad for U Tree

Table 3. Actions in the Towers of Hanoi domain

Base Level Actions		
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

A Set of Abstract Actions	
Action	Effect
Generated abstract actions	
\bar{a}_0	Perform action a_0 in all states
\bar{a}_1	Perform action a_1 in all states
\bar{a}_2	Perform action a_2 in all states
\bar{a}_3	Perform action a_3 in all states
\bar{a}_4	Perform action a_4 in all states
\bar{a}_5	Perform action a_5 in all states
\bar{a}_r	Choose uniformly from $\{a_0, \dots, a_5\}$ in all states
Supplied abstract actions	
\bar{a}_{7P_0}	If the 7 disc stack is on P_0 then choose uniformly from $\{a_0, \dots, a_5\}$, otherwise follow the policy that moves the 7 disc stack to P_0 .
\bar{a}_{7P_1}	If the 7 disc stack is on P_1 then choose uniformly from $\{a_0, \dots, a_5\}$, otherwise follow the policy that moves the 7 disc stack to P_1 .
\bar{a}_{7P_2}	If the 7 disc stack is on P_2 then choose uniformly from $\{a_0, \dots, a_5\}$, otherwise follow the policy that moves the 7 disc stack to P_2 .

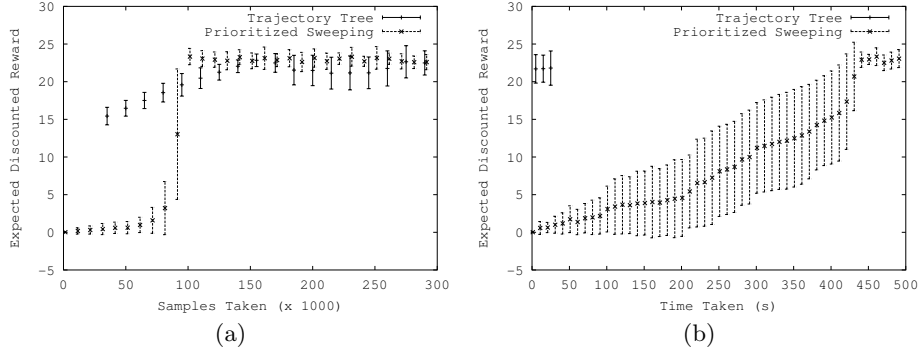


Fig. 1. Results from the Towers of Hanoi domain. (a) A plot of Expected Reward vs. Number of Sample transitions taken from the world. (b) Data from the same log plotted against time instead of the number of samples

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 actually sees 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 that it leaves the abstract state in one step. This makes TTree a better anytime algorithm.

6.2 The Rooms Domains

This domain simulates a two legged robot walking through a maze. The two legs are designated left and right. With a few restrictions, each of these legs can be raised and lowered one unit, and the raised foot can be moved one unit in each of the four compass directions: north, south, east and west. The legs are restricted in movement so that they are not both in the air at the same time. They are also restricted to not be diagonally separated, *e.g.* the right leg can be either east or north of the left leg, but it cannot be both east and north of the left leg.

More formally, we represent the position of the robot using the two dimensional coordinates of the right foot, $\langle x, y \rangle$. We then represent the pose of the robot's legs by storing the three dimensional position of the left foot relative to the right foot, $\langle \Delta x, \Delta y, \Delta z \rangle$. We represent East on the $+x$ axis, North on the $+y$ axis and up on the $+z$ axis. The formal restrictions on movement are that Δx and Δy cannot both be non-zero at the same time and that each of Δx , Δy and Δz are in the set $\{-1, 0, 1\}$. A subset of the state space is shown diagrammatically in Figures 2 and 3. These figures do not show the entire global state space and also ignore the existence of walls.

The robot walks through a grid with a simple maze imposed on it. The mazes have the effect of blocking some of the available actions: any action that would result in the robot having its feet on either side of a maze wall fails. Any action that would result in an illegal leg configuration fails and gives the robot reward of -1 . Upon reaching the grey square in the maze the robot receives a reward of 100.

In our current implementation of TTree we do not handle ordered discrete attributes such as the global maze coordinates, x and y . In these cases we transform each of the ordered discrete attributes into a set of binary attributes. There is one binary attribute for each ordered discrete attribute/value pair describing if the attribute is less than the value. For example, we replace the x attribute with a series of binary attributes of the form: $\{x < 1, x < 2, \dots, x < 9\}$. The y attribute is transformed similarly.

In addition to the mazes above, we use the 'gridworlds' shown in Figure 4 for experiments. It should be remembered that the agent has to walk through

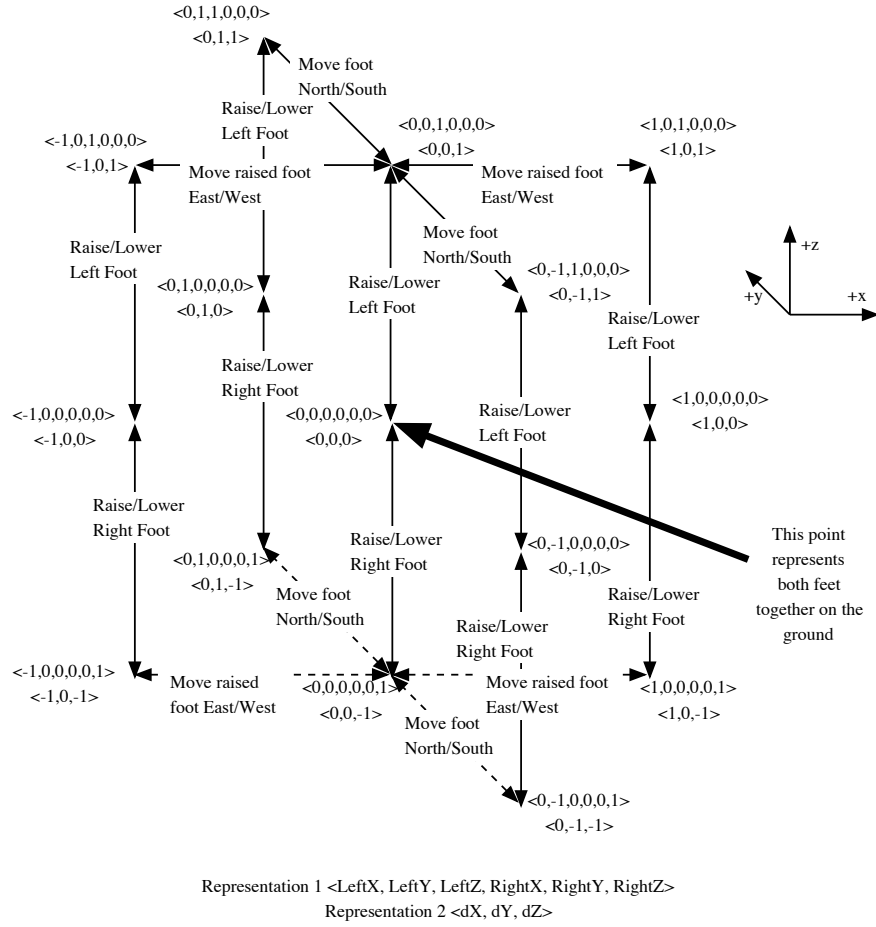


Fig. 2. The local transition diagram for the walking robot domain without walls. This shows the positions of the feet relative to each other. Solid arrows represent transitions possible without a change in global location. Dashed arrows represent transitions possible with a change in global location. The different states are shown in two different coordinate systems. The top coordinate system shows the positions of each foot relative to the ground at the global position of the robot. The bottom coordinate system shows the position of the left foot relative to the right foot

these grids. Unless stated otherwise in the experiments we have a reward of 100 in the bottom right square of the gridworld.

When solving the smaller of the two worlds, shown in Figure 4 (a), TTree was given abstract actions that walk in the four cardinal directions: north, south, east and west. These are the same actions described in the introduction, *e.g.* Tables 4. The various constants were $\gamma = 0.99$, $N_a = 40$, $N_l = 40$, $N_t = 2$ and MAXSTEPS = 150. Additionally, the random abstract action was not useful in this domain,

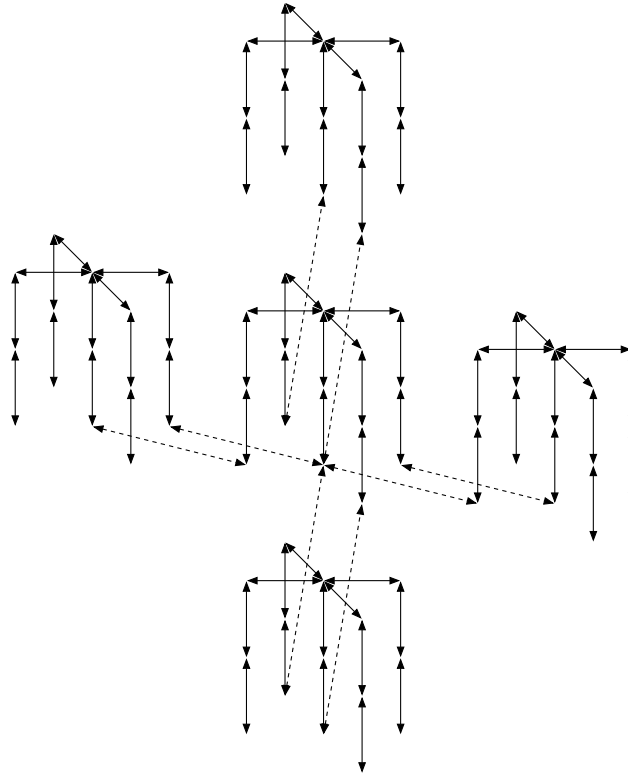
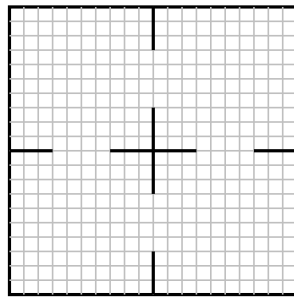
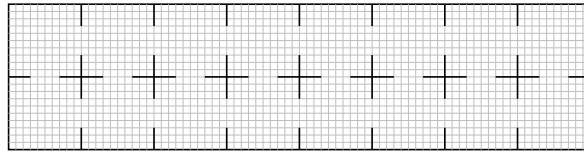


Fig. 3. A subset of the global transition diagram for the walking robot domain. Each of the sets of solid lines is a copy of the local transition diagram shown in Figure 2. As in that figure, solid arrows represent transitions that do not change global location and dashed arrows represent transitions that do change global location



(a)



(b)

Fig. 4. (a) A set of four 10×10 rooms for our robot to walk through; (b) A set of sixteen 10×10 rooms for our robot to walk through

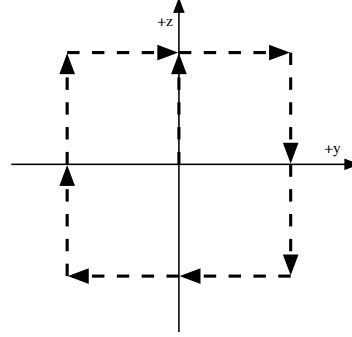
Table 4. The policy for walking north when starting with both feet together. (a) Shows the policy in tree form, (b) shows the policy in diagram form. Note: only the Δz - Δy plane of the policy is shown as that is all that is required when starting to walk with your feet together

```

if  $\Delta z = 0$  then {both feet on the ground}
  if  $\Delta y > 0$  then {left foot north of right foot}
    raise the right foot
  else
    raise the left foot
  end if
else if  $\Delta z = 1$  then {the left foot is in the air}
  if  $\Delta y > 0$  then {left foot north of right foot}
    lower the left foot
  else
    move the raised foot north one unit
  end if
else {the right foot is in the air}
  if  $\Delta y < 0$  then {right foot north of left foot}
    lower the right foot
  else
    move the raised foot north one unit
  end if
end if

```

(a)



(b)

so it was removed. The other generated abstract actions, one for each base level action, remained. The results for the small rooms domain are shown in Figure 5.

When solving the larger world, shown in Figure 4 (b), we gave the agent three additional abstract actions above what was used when solving the smaller world. The first of these was a ‘stagger’ abstract action, shown in Table 5. This abstract action is related to both the random abstract action and the walking actions: it takes full steps, but each step is in a random direction. This improves the exploration of the domain. The other two abstract actions move the agent through all the rooms. One moves the agent clockwise through the world and the other counter-clockwise. The policy for the clockwise abstract action is shown in Figure 6. The counter-clockwise abstract action is similar, but follows a path in the other direction around the central walls.

The results for this larger domain are shown in Figure 7. The various constants were $\gamma = 0.99$, $N_a = 40$, $N_l = 40$, $N_t = 1$ and $\text{MAXSTEPS} = 250$. Additionally the coefficient on the policy code length in the MDL coding was modified to be 10 instead of 20.

6.3 Discussion

There are a number of points to note about the TTree algorithm. Firstly, it generally takes TTree significantly more data than Prioritized Sweeping to converge,

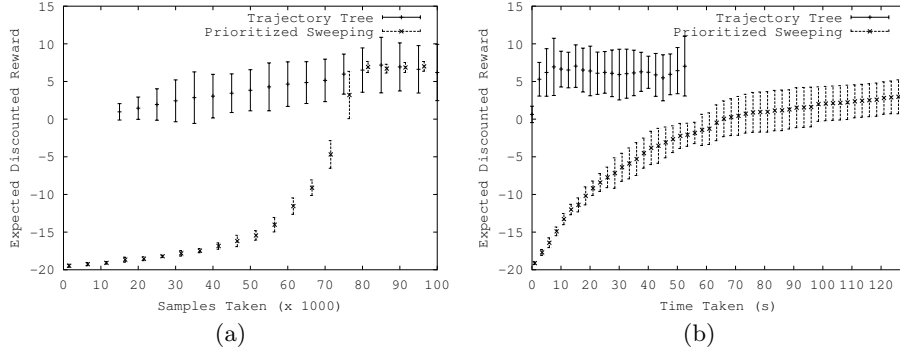


Fig. 5. Results from the walking robot domain with the four room world. (a) A plot of expected reward vs. number of transitions sampled. (b) Data from the same log plotted against time instead of the number of samples

Table 5. The ‘stagger’ policy for taking full steps in random directions

```

if  $\Delta z < 0$  then {the right foot is in the air}
  if  $\Delta x < 0$  then {left foot west of right foot}
    move the raised foot one unit west
  else if  $\Delta x = 0$  then {right foot is same distance east/west as left foot}
    if  $\Delta y < 0$  then {left foot south of right foot}
      move the raised foot one unit south
    else if  $\Delta y = 0$  then {left foot is same distance north/south as right foot}
      lower the right foot
    else {left foot north of right foot}
      move the raised foot one unit north
    end if
  else {left foot east of right foot}
    move the raised foot one unit east
  end if
else if  $\Delta z = 0$  then {both feet are on the ground}
  if  $\Delta x = 0$  and  $\Delta y = 0$  then {the feet are together}
    raise the left foot
  else
    raise the right foot
  end if
else {the left foot is in the air}
  if  $\Delta x = 0$  and  $\Delta y = 0$  then {the left foot is directly above the right foot}
    Move the raised foot north, south, east or west with equal probability
  else
    lower the left foot
  end if
end if

```

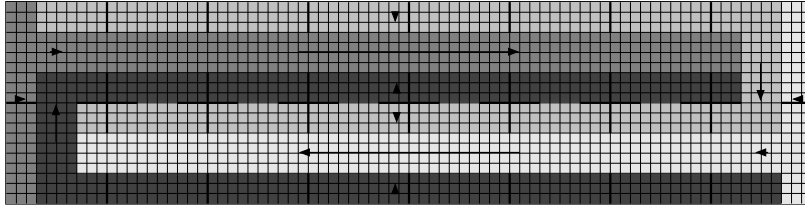


Fig. 6. The clockwise tour abstract action. This is a policy over the rooms shown in Figure 4 (b)

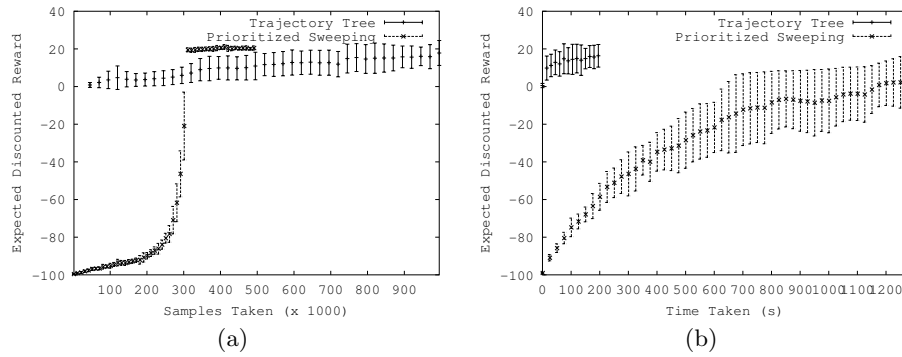


Fig. 7. Results from the walking robot domain with the sixteen room world. (a) A plot of Expected Reward vs. Number of Sample transitions taken from the world. (b) Data from the same log plotted against time instead of the number of samples

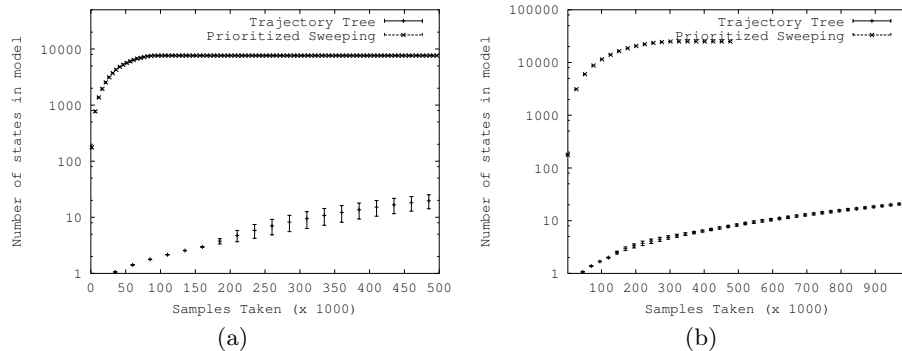


Fig. 8. Plots of the number of states seen by Prioritized Sweeping and the number of abstract states in the TTree model vs. number of samples gathered from the world. The domains tested were (a) the Towers of Hanoi domain, and (b) the walking robot domain with the sixteen room world. Note that the y -axis is logarithmic

although TTree performs well long before convergence. This is unsurprising. Prioritized Sweeping is remembering all it sees, whereas TTree is throwing out all trajectories in a leaf when that leaf is split. For example, all data gathered before the first split is discarded after the first split.

However, TTree is significantly faster than Prioritized Sweeping in real time in large domains (see Figures 1b, 5b and 7b). It performs significantly less processing on each data point as it is gathered and this speeds up the algorithm. It also generalizes across large regions of the state space. Figure 8 shows the sizes of the data structures stored by the two algorithms. Note that the y -axis is logarithmic. TTree does not do so well in small domains like the taxi domain [6].

Given this generalization, it is important to note why we did not compare to other state abstraction algorithms. The reason is because other state abstraction algorithms do not have a temporal abstraction component and so cannot generalize across those large regions. *e.g.* Continuous U Tree performs very poorly on these problems.

The next point we would like to make is that the abstract actions help TTree avoid negative rewards even when it has not found the positive reward yet. In the walking robot domain, the agent is given a small negative reward for attempting to move its legs in an illegal manner. TTree notices that all the trajectories using the generated abstract actions receive these negative rewards, but that the supplied abstract actions do not. It chooses to use the supplied abstract actions and hence avoid these negative rewards. This is evident in Figure 5 where TTree’s expected reward is never below zero.

The large walking domain shows a capability of TTree that we have not emphasized yet. TTree was designed with abstract actions like the walking actions in mind where the algorithm has to choose the regions in which to use each abstract action, and it uses the whole abstract action. However TTree can also

Table 6. Part of the policy tree during the learning of a solution for the large rooms domain in Figure 4 (b)

```

if  $x < 78$  then
  if  $x < 68$  then
    if  $y < 10$  then
      perform the loop counter-clockwise abstract action
    else
      perform the loop clockwise abstract action
    end if
  else
    {Rest of tree removed for space}
  end if
else
  {Rest of tree removed for space}
end if

```

choose to use only part of an abstract action. In the large walking domain, we supplied two additional abstract actions which walk in a large loop through all the rooms. One of these abstract actions is shown in Figure 6. The other is similar, but loops through the rooms in the other direction.

To see how TTree uses these ‘loop’ abstract actions, Table 6 shows a small part of a tree seen while running experiments in the large walking domain. In the particular experiment that created this tree, there was a small, -0.1 , penalty for walking into walls. This induces TTree to use the abstract actions to walk around walls, at the expense of more complexity breaking out of the loop to reach the goal. The policy represented by this tree is interesting as it shows that the algorithm is using part of each of the abstract actions rather than the whole of either abstract action. The abstract actions are only used in those regions where they are useful, even if that is only part of the abstract action.

This tree fragment also shows that TTree has introduced some non-optimal splits. If the values 78 and 68 were replaced by 79 and 70 respectively then the final tree would be smaller.⁴ As TTree chooses its splits based on sampling, it sometimes makes less than optimal splits early in tree growth. The introduction of splits causes TTree to increase its sample density in the region just divided. This allows TTree to introduce further splits to achieve the desired division of the state space.

The note above about adding a small penalty for running into walls in order to induce TTree to use the supplied abstract actions deserves further comment. The Taxi domain [15] has a penalty of -10 for misusing the pick up and put down actions. It has a reward of 20 for successfully delivering the passenger. We found TTree had some difficulty with this setup. The macros we supplied chose randomly between the pick up and put down actions when the taxi was at the appropriate taxi stand. While this gives a net positive reward for the final move

⁴ The value 79 comes from the need to separate the last column to separate the reward. The value 70 lines up with the edge of the end rooms.

(with an expected reward of 10), it gives a negative expected reward when going to pick up the passenger. This makes the abstract action a bad choice on average. Raising the final reward makes the utility of the abstract actions positive and helps solve the problem.

When running our preliminary experiments in the larger walking domain, we noticed that sometimes TTree was unable to find the reward. This did not happen in the other domains we tested. In the other domains there were either abstract actions that moved the agent directly to the reward, or the random abstract action was discovering the reward. In the walking domain the random abstract action is largely ineffective. The walking motion is too complex for the random action to effectively explore the space. The abstract actions that walk in each of the four compass directions will only discover the reward if they are directly in line with that reward without an intervening wall. Unless the number of sample points made very large, this is unlikely. Our solution was to supply extra abstract actions whose goal was not to be used in the final policy, but rather to explore the space. In contrast to the description of McGovern [16], where macros are used to move the agent through bottlenecks and hence move the agent to another tightly connected component of the state space, we use these exploration abstract actions to make sure we have fully explored the current connected component. We use these ‘exploration’ abstract actions to explore within a room rather than to move between rooms.

An example of this type of exploratory abstract action is the ‘stagger’ abstract action shown in Table 5. We also implemented another abstract action that walked the agent through a looping search pattern in each room. This search pattern covered every space in the room, and was replicated for each room. The stagger policy turned out to be enough to find the reward in the large walking domain and it was significantly less domain specific than the full search, so it was used to generate the results above.

7 Conclusion

We have introduced the TTree algorithm for finding policies for Semi-Markov Decision Problems. This algorithm uses both state and temporal abstraction to help solve the supplied SMDP. Unlike previous temporal abstraction algorithms, TTree does not require termination criteria on its abstract actions. This allows it to piece together solutions to previous problems to solve new problems. We have supplied both a proof of correctness and empirical evidence of the effectiveness of the TTree algorithm.

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. 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)
5. 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
6. 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
7. Parr, R.S., Russell, S.: Reinforcement learning with hierarchies of machines. In: Neural and Information Processing Systems (NIPS-98). Volume 10., MIT Press (1998)
8. Uther, W.T.B.: Tree Based Hierarchical Reinforcement Learning. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (2002)
9. Hengst, B.: Discovering hierarchy in reinforcement learning with HEXQ. In: International Conference on Machine Learning (ICML02). (2002)
10. Baird, L.C.: Residual algorithms: Reinforcement learning with function approximation. In Frieditis, A., Russell, S., eds.: Machine Learning: Proceedings of the Twelfth International Conference (ICML95), San Mateo, Morgan Kaufmann (1995) 30–37
11. Knoblock, C.A.: Automatically Generating Abstractions for Problem Solving. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA (1991)
12. 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)
13. Moore, A.W., Atkeson, C.G.: Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning* **13** (1993)
14. Strens, M., Moore, A.: Direct policy search using paired statistical tests. In: International Conference on Machine Learning (ICML 2001). (2001)
15. Dietterich, T.G.: Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research* **13** (2000) 227–303
16. McGovern, A.: Autonomous Discovery Of Temporal Abstractions From Interaction With An Environment. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts (2002)