JMLR: Workshop and Conference Proceedings 24:11–22, 2012 10th European Workshop on Reinforcement Learning

Feature Reinforcement Learning using Looping Suffix Trees

Mayank Daswani MAYANK.DASWANI@ANU.EDU.AU
Peter Sunehag PETER.SUNEHAG@ANU.EDU.AU
Marcus Hutter MARCUS.HUTTER@ANU.EDU.AU
Research School of Computer Science,
Australian National University,
Canberra, ACT, 0200, Australia.

Editor: Marc Peter Deisenroth, Csaba Szepesvári, Jan Peters

Abstract

There has recently been much interest in history-based methods using suffix trees to solve POMDPs. However, these suffix trees cannot efficiently represent environments that have long-term dependencies. We extend the recently introduced CTΦMDP algorithm to the space of looping suffix trees which have previously only been used in solving deterministic POMDPs. The resulting algorithm replicates results from CTΦMDP for environments with short term dependencies, while it outperforms LSTM-based methods on TMaze, a deep memory environment.

1. Introduction

In reinforcement learning (RL) an agent must learn a policy (behaviour) that performs well in a given (dynamic) environment through interactions with the environment itself [Kaelbling et al., 1996]. Traditional RL methods maximise the reward in a given unknown finite Markov Decision Process (MDP). In the general RL problem there are no states, the agent instead receives observations that are not necessarily Markov. Feature Reinforcement Learning (ΦMDP) Hutter [2009] automates the extraction of a good state representation in the form of an MDP from the agent's observation-reward-action history. More clearly, this means that given a class of maps Φ we consider a class of environments that at least approximately reduce to an MDP through a map $\phi \in \Phi$. Nguyen et al. [2011] recently showed that this is viable in practice using a map class of history suffix trees.

The problem with using suffix trees as a map class is that they cannot efficiently remember events over a long period of time. The depth of the suffix tree is proportional to the length of history that it has to remember. In order to deal with long-term dependencies we make use of the class of looping suffix trees (looping STs) within the ΦMDP framework. Holmes and Jr. [2006] first proposed looping STs in the deterministic case, we also consider stochastic models which can be crucial even in deterministic environments. We show that looping suffix trees in conjunction with the ΦMDP framework can be used to successfully find compact representations of environments that require long-term memory in order to perform optimally.

1.1. Related Work

Our looping suffix tree method learns a finite state automaton that is well suited to long-term memory tasks. While tree-based methods such as USM [McCallum, 1995], MC-AIXI-CTW [Veness et al., 2011], Active LZ [Farias et al., 2010], CTΦMDP [Nguyen et al., 2011] and many others can in principle handle long-term memory tasks, they require excessively large trees to represent such environments. These large trees can result in large state spaces, which then promote the exacerbated exploration-exploitation problem. More related to our work, Mahmud [2010] aims at searching the very large space of probabilistic deterministic finite automata (with some restrictions). In a similar vein, but restricted to deterministic observations [Haghighi et al., 2007] also construct finite automata that aim at being the minimal predicting machine.

A popular alternative to finite state automaton learning is a class of algorithms based on recurrent neural networks (RNNs) particularly those based on the Long Short-Term Memory (LSTM) framework by Hochreiter and Schmidhuber [1997]. The LSTM framework was first proposed to predict time-series data with long-term dependencies. This was introduced to the RL context by Bakker [2002] and more recently a new model-free variant based on policy gradients by Wierstra et al. [2007]. These methods are more often used in the continuous case, but were also tested in the discrete setting. Recently, Echo State Networks [Szita et al., 2006] which are also RNN-based have also been tested on long-term memory tasks.

2. Preliminaries

Agent-Environment Framework. We use the notation and framework from [Hutter, 2005]. An agent acts in an Environment Env. It chooses actions $a \in \mathcal{A}$, and receives observations $o \in \mathcal{O}$ and real-valued rewards $r \in \mathcal{R}$ where $\mathcal{A}, \mathcal{O}$ and $\mathcal{R}$ are all finite. This observation-reward-action sequence happens in cycles indexed by $t = 1, 2, 3, \ldots$. We use $x_{1:n}$ throughout to represent the sequence $x_1 \ldots x_n$. We define the space of histories as $\mathcal{H} := (\mathcal{O} \times \mathcal{R} \times \mathcal{A})^* \times \mathcal{O} \times \mathcal{R}$. The history at time t is given by $h_t = o_1 r_1 a_1 \ldots o_{t-1} r_{t-1} a_{t-1} o_t r_t$. Using this definition of history we formally define the agent to be a function $Agent : \mathcal{H} \rightsquigarrow \mathcal{A}$ where $Agent(h_t) := a_t$. Similarly, the environment can be viewed as a stochastic function of the history, $Env : \mathcal{H} \times \mathcal{A} \rightsquigarrow \mathcal{O} \times \mathcal{R}$, where $Env(h_{t-1}, a_{t-1}) := o_t r_t$. The symbol $\rightsquigarrow$ indicates a possibly stochastic function. A policy is defined as a map $\pi : \mathcal{H} \rightsquigarrow A$.

If $Pr(o_t r_t | h_t, a_t) = Pr(o_t r_t | o_{t-1} a_t)$, the environment is said to be a discrete **Markov Decision Process (MDP)** [Puterman, 1994]. In this case, the observations form the state space of the MDP. Formally an MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, R \rangle$ where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions and $R : \mathcal{S} \times \mathcal{A} \rightsquigarrow \mathcal{R}$ is the (possibly stochastic) reward function which gives the (real-valued) reward gained by the agent after taking action a in state s. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the *state-transition function*. The agent's goal is to maximise its future discounted expected reward, where a geometric discount function with rate γ is used. The value of a state according to a stationary policy is given by $V^\pi(s) = E_\pi\{R_t | s_t = s\}$ where $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ is the *return*. We want to find the optimal value function V^* such that $V^*(s) = \max_\pi V^\pi(s)$. If an MDP is known then this can be done by value iteration [Sutton and Barto, 1998]; in the unknown case the agent must deal with the problem of exploration vs exploitation, which only has efficient approximate solutions.

ΦMDP. Hutter [2009] proposes a framework that extracts relevant features from the history for reward prediction. This framework gives us a method to find a map $\phi : \mathcal{H} \to \mathcal{S}$ such that the state at any time step $s_t = \phi(h_t)$ is approximately a sufficient statistic of the history. It uses a global cost function that is inspired by the minimum description length principle [Rissanen, 1978]. The cost is the sum of the code lengths of state and reward sequences given actions. This cost is combined with a global stochastic search technique (such as simulated annealing [Liu, 2008]) to find the optimal map. The standard cost is defined as follows

$$Cost(\phi|h_n) := CL(s_{1:n}|a_{1:n}) + CL(r_{1:n}|s_{1:n}, a_{1:n}) + CL(\phi)$$

$CL(s_{1:n}|a_{1:n})$ is the code length of the state sequence given the action sequence. The subsequence of states reached from a given state s via action a is i.i.d as it is sampled from an MDP. We form a frequency estimate of the model of this MDP. The code length is then the length of the arithmetic code with respect to the model plus a penalty for coding parameters. The coding is optimal by construction. $CL(r_{1:n}|s_{1:n}, a_{1:n})$ follows similarly.

The consistency of this cost criterion was proven by Sunehag and Hutter [2010]. The modified cost by Nguyen et al. [2011] adds a parameter α to control the balance between reward coding and state coding,

$$Cost_\alpha(\phi|h_n) := \alpha CL(s_{1:n}|a_{1:n}) + (1-\alpha)CL(r_{1:n}|s_{1:n}, a_{1:n}) + CL(\phi).$$

We primarily care about reward prediction. However, since rewards depend on states we also need to code the states. The *Cost* is well-motivated since it balances between coding states and coding rewards. A state space that is too large results in poor learning and a long state coding, while a state space that is too small can obscure structure in the reward sequence resulting in a long code for the rewards.

Nguyen et al. [2011] search the map space of suffix trees (explained below). Our method extends this to looping suffix trees. The generic ΦMDP algorithm is given in Algorithm 1 in the Appendix. The agent is first initialised with some history based on random actions. Then it alternates between finding a "best" ϕ using the *simulated annealing* algorithm and performing actions based on the optimal policy for that ϕ found via the *FindPolicy()* function. The *FindPolicy()* function can be any standard reinforcement learning algorithm that finds the optimal policy in an unknown MDP, and should perform some amount of exploration, generally via an optimistic initialisation. In this paper, we use the model-based method as specified by Hutter [2009] which is based on Szita and Lörincz [2008]. This method adds an additional "garden of eden" state (s_e) to the estimated MDP, which is an absorbing state with a high reward. The agent is told that it has been to s_e once from every other state, however the agent cannot actually transition to this state. Then we simply perform value iteration on this augmented MDP. Initially the agent will explore in a systematic manner to try and visit s_e, but as it accumulates more transitions from a particular state, the estimated transition probability to s_e decreases, and the agent eventually settles on the optimal policy.

Definition 1 (Suffix Tree) *Let $\mathcal{O} = \{o^1, o^2, o^3, ..., o^d\}$ be a d-ary alphabet. A suffix tree is a d-ary tree in which the outgoing edges from each internal node are labelled by the elements of $\mathcal{O}$. Every suffix tree has a corresponding suffix set which is the set of strings*

$\mathcal{S} = \{s^1, s^2, ..., s^n\}$ *generated by listing the labels on the path from each leaf node to the root of the tree.*

The suffix set has the property that no string is a suffix of any other string and any sufficiently long string must have a suffix in the set. Each string in the suffix set is called a state, and hence this is also called a suffix state set. The *l-th* level of the tree corresponds to the *l-th* last observation in the history. By the above properties, any history of sufficient length must be mapped to one and only one state based on its suffix.

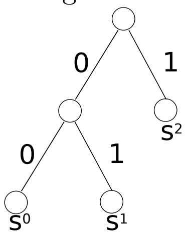
Definition 2 (Looping Tree) *A looping tree is a tree which may have loops from any leaf node to an ancestor.*

Definition 3 (Looping suffix Tree) *A looping suffix tree based on a d-ary alphabet* $\mathcal{O} = \{o^1, o^2, o^3, ..., o^d\}$ *is a d-ary looping tree in which edges coming from each internal node are labelled by the elements of* $\mathcal{O}$*. The loops in the tree are unlabelled. The non-looping leaf nodes in the looping ST form the state set along with an additional state* s^{empty} *known as the empty state.*

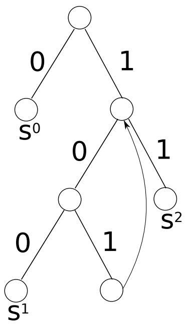
In order to map a history sequence to a state in a looping ST we simply follow the edges in the tree until we get to a state (Algorithm 2, Appendix). If we reach the beginning of the history sequence without reaching a state, we map the sequence to the empty state.

Looping suffix trees have the effect of giving Kleene-star like representational ability to the standard suffix set. For example, Figure 1 shows a looping suffix tree which has the suffix set $\{0, 00(10)^*1, 1(10)^*1\}$. Let $h = [0, 0, 1, 1, 0, 1]$. We can map this history sequence to the state sequence $stateSeq = [s^0, s^0, s^1, s^2, s^0, s^2]$. The last state is mapped by following 1,0,1 down the tree, then following the loop back up the tree to finally take another 1 to end in s^2.

Figure 1:



(a) Suffix Tree



(b) Looping ST

3. Looping Suffix Trees in ΦMDP

The benefit of using looping STs comes from the ability to remember relevant past events by 'forgetting' or looping over irrelevant details. Holmes and Jr. [2006] restrict their discussion to the deterministic case without rewards. Unfortunately their loopability criterion cannot work in the stochastic case since a loop can change not only the possible transitions but also the transition probabilities.

The cost function of the ΦMDP framework immediately gives us a well-motivated criterion for evaluating looping suffix trees. Using looping suffix trees as the map class in this framework allows us to extend them to stochastic environments. While we have not yet shown theoretical guarantees, experimental results show that ΦMDP works well in the space of looping suffix trees. The extension to stochastic tree sources is also useful in deterministic environments, where in some cases a smaller stochastic tree source can sufficiently capture a deterministic environment.

Definition 4 *A history is said to be* consistent *with respect to a particular looping suffix tree if it can be mapped to a state sequence that does not include the empty state. The definition of inconsistent follows in the obvious manner.*

Algorithm. The algorithm consists of a specification of $CL(\phi)$ and the neighbourhood method which is needed for the simulated annealing algorithm in the generic ΦMDP algorithm (Algorithm 1, Appendix). We call our algorithm LSTΦMDP. A tree with num_nodes can be coded in num_nodes bits [Veness et al., 2011, Sec.5] and the starting and ending nodes of all loops can be coded in $2\log(num\_nodes) num\_loops$, so we define the model cost of the map $CL(\phi)$ as

$$CL(\phi) = num\_nodes + 2\log(num\_nodes) num\_loops.$$

The getNeighbour() method (Algorithm 3, Appendix) first selects a state randomly and then with equal probability subject to certain conditions, it selects between one of 4 operations. Note that the simulated annealing procedure that we use is a very simple generic method. However this can be extended to more sophisticated annealing schemes such as parallel tempering as done by Nguyen et al. [2011].

merge : In order to merge a state, all sibling nodes must also be states. From the definition of a suffix set, we know that every state corresponds to a unique suffix. The merge is simply the shortening of a context for those states. If s^i is the state being merged and $s^i = o^j n'$ where $o^j \in \mathcal{O}$ and n' is the remainder of the suffix corresponding to that state, then the siblings of s^i are $o^k n'$ where $k \neq i$. If these siblings are also states then the merge operator removes $o^i n'$ for all i from the suffix set and adds a new state n'.

split : Analogously, we can split any state s^i by adding a depth one context to the state i.e. by constructing $|\mathcal{O}|$ new states of the form $o^j s^i$ for all $o^j \in \mathcal{O}$ and removing the state s^i.

addLoop : The addLoop function has two cases. Either we add a loop from an existing state to it's parent (thereby removing it from the state set and adding it to the loop set) or we extend an existing loop to the parent of the existing node looped to.

removeLoop : The removeLoop function is simply a reverse of the addLoop function allowing us to decrease the length of a loop, or if it is a length one loop create a new state from the node.

Loops introduce a few problems to the ΦMDP procedure. A looped tree can be inconsistent with the current history. This can be problematic if, for instance, the optimal tree is inconsistent with the current history. One solution is to always provide a reasonable pre-history that the optimal tree should be consistent with. For example in the TMaze case (see Section 4), we ensure that the first observation is in fact the start of an episode, which is a reasonable assumption. Then any trees that are inconsistent can be discarded. In fact to make the search quicker, we can mark nodes where loops make the tree inconsistent and no longer add those loops. The initial map is always set to be the depth one tree (i.e. one split). Non-looping suffix trees were shown to be statistically consistent with reference to the **Cost** function (under the restrictions of a fixed policy and ergodicity) only by neglecting the root tree by Sunehag and Hutter [2010].

The space of looping STs includes the space of ordinary STs. Therefore, results from the non-looping case [Nguyen et al., 2011] should be reproducible, as long as the simulated annealing procedure is not adversely affected by the enlargement of the search space. Experimental results show that some care must be taken in choosing α for this to be the case. This is further discussed in Section 4.

4. Experiments

In this section we describe our experimental setup and the domains that we used to evaluate our algorithm. Each domain was used to test a different ability of the algorithm. Every experiment was run 50 times. The agent is given an initial history produced by taking random actions. Each run of an experiment was conducted over some number of epochs with each epoch containing 100 iterations of the agent performing actions according to its current policy, based on the current map with a constant ϵ-exploration of 0.1 until a point where it stops exploration. After every epoch, the agent was given a chance to change its optimal map via a simulated annealing procedure. The annealing procedure used an exponential cooling function with constants chosen so that the first few maps had an initial acceptance probability in the range $[0.6, 0.7]$. Plots show every 10th point with 2 standard error on either side. The exact constants used for all the experiments can be found in Table 1 in the Appendix.

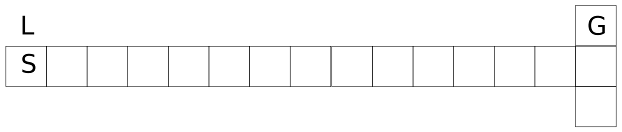


Figure 2: TMaze environment showing goal at left

TMaze. The TMaze problem is a classic non-Markovian problem in RL. It nicely demonstrates the need for long-term memory as well as the exploration vs exploitation problem. We use the formulation as described by Bakker [2002]. The environment is a T-shaped maze (see Figure 2) with the length of the neck of the T (the corridor) being adjustable. The observation space is $\mathcal{O} = \{0, 1, 2, 3\}$, the rewards are $\mathcal{R} = \{-0.1, 4\}$ and there are four actions denoted by up, right, left, down. The agent needs to remember the observation it receives at the start of the maze, which tells it whether to turn left or right at the end.

The agent receives an observation (either 1 or 2) at the start of the maze that it must remember until it reaches the decision node (observation 3), at which point it must turn left(1) or right(2) according to the initial observation in order to receive a reward of 4. If it chooses any other action it gets reset into the decision state and gets another observation of 0 and a reward of -0.1.

We conducted experiments on three variants of TMaze. In the first variant, the observation it receives at the start determines where the goal lies every time. In the second variant, the agent receives two different observations in the corridor with equal probability. This means that the looping ST needs to loop over both observations in any possible order. The third variant adds uncertainty to the accuracy of the starting observation along with the stochasticity in the corridor, it predicts the position of the reward with 0.8 probability. In each variant

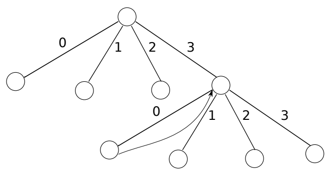


Figure 3: A reward optimal LST for the TMaze problem

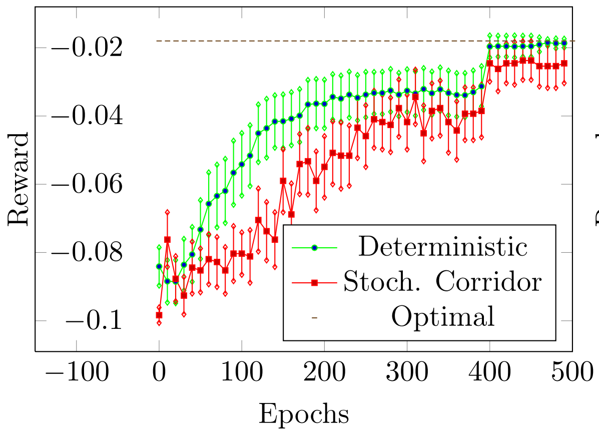


Figure 4: $LST\Phi MDP$ on TMaze length 50

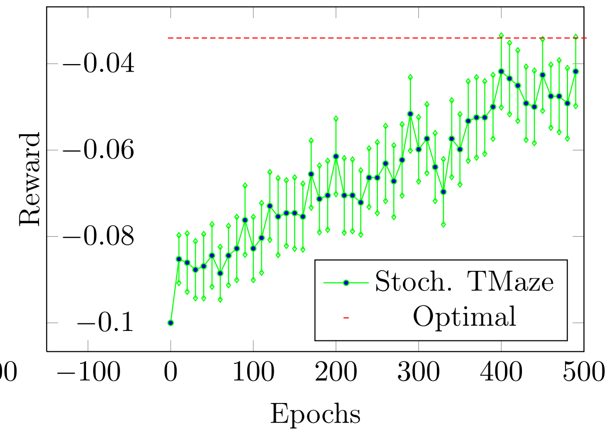


Figure 5: Stochastic TMaze length 50

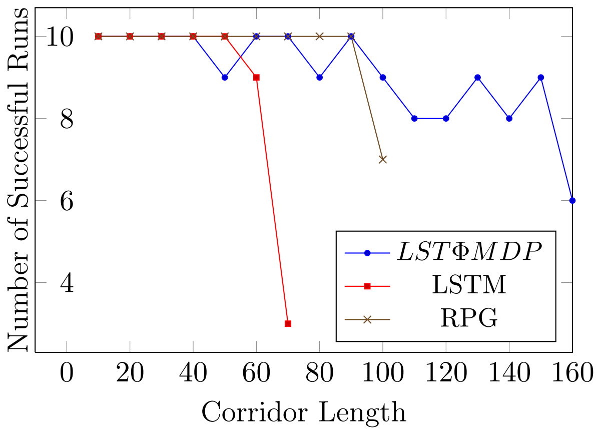


Figure 6: #optimal runs, varying corridor lengths for LSTΦ, RPG and RL-LSTM on Det. TMaze

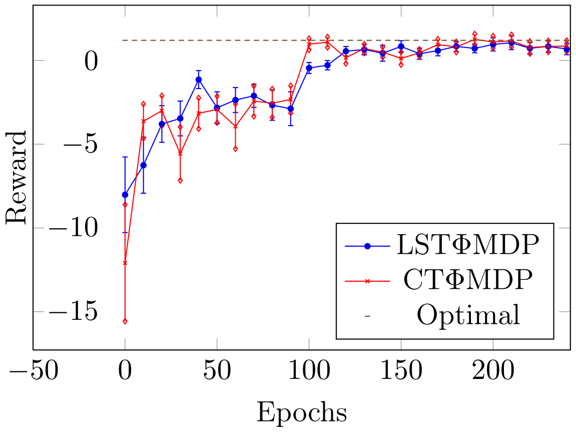


Figure 7: Comparison on Tiger with prob(listen)=0.85 for LSTΦMDP and CTΦMDP

we can adjust the length of the corridor. Note that the first variant is deterministic within a given episode, however the history itself is not deterministic since the observation received at the start of the episode is selected randomly, which is enough to prohibit deterministic approaches.

We compare our LSTΦMDP to RL-LSTM [Bakker, 2002] and Recurrent Policy Gradients (RPG) [Wierstra et al., 2007] on the deterministic TMaze. Note that we use the results from the corresponding papers; we did not implement the methods ourselves. Following the experiments in those papers, we increase the length of the corridor systematically from 10 to 100, in increments of 10. In this case each experiment was run 10 times, and we measured the number of successful runs per length. A run is said to be successful if the agent achieves the optimal policy and hence the optimal reward in at least the last 10 epochs. We used this metric to compare with other methods. All the successful runs had optimal policies from 400 epochs onward i.e. once there was no longer any ϵ-exploration. We continued to increase the corridor length until the performance of our algorithm was worse than the performance of the RPG method at length 100, which happened at corridor length 160 (6 successful runs).

Locked door. In order to show that our algorithm was useful in solving other long-term dependency problems we tested on a new domain we call the "locked door". The agent is in a room (represented by a grid). The room has a locked door and in order to leave, the agent must collect a key from a particular location. In our experiment we use a 7x7 grid with the door in the top-left corner, the key in the top-right corner and the agent starting in the location one square below the door. The agent has actions up, down, left and right and receives observations that are a binary coding of the adjacent walls. This means that states with the same wall configuration have the same observation. Bumping into a wall, collecting the key, and visiting the door have their own unique observations. The agent gets a reward -5 for bumping into a wall, +10 for visiting the door after obtaining the key and and -1 for every other timestep. The agent was given a history of 1000 random actions at the start and every run of the experiment was 1000 epochs long with each epoch being a 100 iterations as usual.

5. Analysis

In this section we analyse the results from our experiments, and explain characteristic behaviours and parameter settings. The neighbourhood function was chosen to traverse the state space slowly through the looping trees linked to a particular suffix tree, after a few experiments with larger jumps failed. Loops make smaller representations of large environments possible. The difference in cost between two adjacent trees can be quite large, since a loop can suddenly explain a very large amount of data by ignoring irrelevant sequences.

Deterministic TMaze. In the case of corridor length 50, the optimal policy has a value of -0.018. The agent reaches the optimal policy in every run once the ϵ-exploration has been turned off at 400 epochs. See Figure 4 for details. The results of the separate experiment comparing the algorithms performance on varying corridor lengths are displayed in Figure 6. Up to length 100 the agent reaches the optimal policy, with a few corridor lengths having one run stuck on traversing the corridor without every having seen the goal. Note that the algorithm does not necessarily reach the optimal tree, but finds a reward-optimal tree that contains it. In comparison, RL-LSTM [Bakker, 2002] has increasingly many suboptimal runs as the length of the corridor increases past 50. RPG [Wierstra et al., 2007] has optimal results up to length 90 but has 3 unsuccessful runs at length 100. We continue increasing corridor length until we have more than 3 unsuccessful runs at length 160. Additionally, our algorithm uses 50000 iterations (500 epochs) in all cases, while RPG uses around 2 million iterations for corridor length 100. We also tested CTΦMDP but it was not successful for corridor lengths>5. We would need a depth n suffix tree to represent a TMaze with length n. However, a looping suffix tree with optimal reward prediction is much easier to find, as shown in Figure 3 and also much smaller, leading to greater data efficiency. We did not test Echo State Networks, however from [Szita et al., 2006] we note that the method was not successful on corridor lengths greater than 25. In this environment, the optimal looping suffix tree (Figure 3) is the same regardless of the length of the corridor, since the tree simply loops over the corridor observations. Of course the exploration-exploitation problem gets harder as the corridor length increases. Despite this the systematic exploration of the agent appears to work well. We also note that in comparison to Recurrent Neural Networks (i.e. the LSTM based methods) it is relatively much simpler to interpret a looping suffix tree.

Stochastic TMaze (corridor length 50). The optimal policy in the stochastic corridor TMaze case has a value of -0.018 the same as the deterministic case. However, the agent has to loop over a new observation, and hence needs a larger tree. The task is made hard by the stochastic nature of the corridor observations. Failures occur mainly due to exploration issues (agent not finding a reward often enough) rather than problems with simulated annealing. This means that the average reward is a little lower than the optimal, however in most cases the agent did reach the optimal. In the case where the accuracy of the initial observation is 0.8, the expected reward is -0.03404. The results have more variability at each point as seen in the higher error bars, but overall the agent reaches nearly optimal reward in every run with the average of the final point being -0.04178.

Tiger. The Tiger example is interesting since it shows that the agent can still reproduce results from the regular non-looping suffix tree case. The agent achieves the optimal reward when the parameter α is set to a lower value of 0.01. Figure 7 shows that LSTΦMDP and CTΦMDP perform nearly identically on this problem.

Locked door. When the agent visits the door location there are two contexts, it either has the key or it doesn't. Remembering that it has a key is much easier with loops, since it can simply loop over observations once it has collected the key. The LSTΦMDP agent with $\alpha = 0.1$ succeeds in finding a near-optimal policy in about half the runs. CTΦMDP succeeds in learning how to avoid walls but never improves further in 1000 epochs. See Figure 8 for the graph of the near-optimal runs of LSTΦMDP.

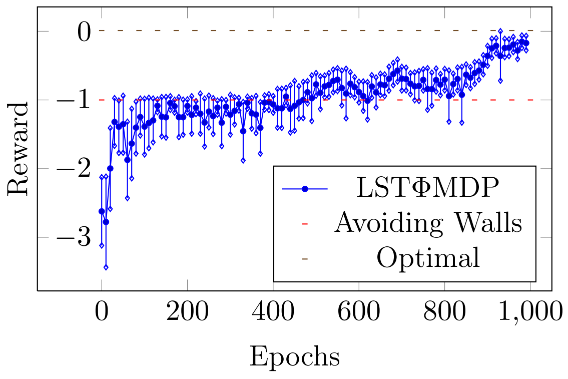


Figure 8: Locked door

General Problems. The cost function needed mild tuning of the parameter α for the experiments, generally relying on low values (especially in Tiger). This emphasises reward prediction over state prediction. Looping STs can reduce the cost of coding state sequences dramatically by looping over several observations and substantially reducing the number of states. Obviously this can lead to a bad reward coding, which should eventually cause the tree to be rejected. However, if the agent has not seen enough of the various available rewards then the reward cost may not be particularly high. This can be self-reinforcing. Bad models of the environment can result in policies that only rarely experience critical events, for example opening the door in the Tiger or Locked door problem. This means that the reward cost changes very slowly and may not ever dominate the total cost. Note that this often means that if the agent does not find the reward early on in the run, then it has not much chance of finding it later. Inspecting the failed runs for the deterministic TMaze, we see that the agent never experiences the reward or only sees it once or twice. Particularly, as the length of the maze increases both the optimism and the ϵ-exploration are insufficient to fully explore the maze.

Computational Complexity. The most time consuming part of ΦMDP is the calculation of the **Cost** of a new map. The calculation of **Cost** from the statistics is $O(|S|^2|A|) + O(|S||A||R|)$. However, since the state space changes the statistics must be recomputed. In

CTΦMDP this can be done using a pass over the history (of length n) with backtracking limited to the depth d of the tree, making the worst-case complexity $O(dn)$. However, loops can require backtracking to the start of the history making the worst-case complexity $O(n^2)$. Note that if $n < |S|^2|A|$ there are some transitions that have not been seen and can thus be ignored when calculating **Cost**, so the complexity is dominated by the $O(dn)$ or $O(n^2)$ term. In practice, the execution times are competitive to (non-looping) suffix tree based methods on environments that do not require loops. For example on Tiger, the average execution time for LSTΦMDP is 11.49s and for CTΦMDP it is 11.27s. In environments where loops matter, LSTΦMDP is much slower, for example on TMaze (length 50) an average run for LSTΦMDP is 216.93s while for CTΦMDP it is 38s. The large speed difference is because CTΦMDP remains on the (sub-optimal) minimal tree of 4 states, which results in less time spent in the annealing procedure.

6. Conclusion

We introduced looping suffix trees to the feature reinforcement learning framework [Hutter, 2009] to create an algorithm called LSTΦMDP. The experimental results show that looping suffix trees are particularly useful in representing long-term dependencies by looping over unnecessary observations. Loops allow for smaller representations leading to greater data efficiency. We outperform LSTM-based algorithms [Bakker, 2002; Wierstra et al., 2007] on TMaze. LSTΦMDP was also able to perform well on stochastic environments, which is a handicap of previous methods using looping suffix trees [Holmes and Jr., 2006; Haghighi et al., 2007]. We also replicated results of CTΦMDP [Nguyen et al., 2011] on short-term environments.

Scaling to larger domains is the future direction of our research. Nguyen et al. [2012] deal with large observation spaces by adding an unseen context state to the suffix tree, which can be problematic with regard to data efficiency. A potentially better way is function approximation techniques. An interesting idea is to use loops in instead of an unseen symbol, e.g. requiring all things unseen to be looped over. In order to deal with the added time complexity of loops, we aim to develop a neighbourhood function over *Markov* looping suffix trees only, which would reduce the complexity of finding **Cost** of a new tree to $O(n)$ by allowing us to incrementally update the state based on the new observation.

Acknowledgements. The research was partly supported by the Australian Research Council Discovery Project DP120100950.

References

B. Bakker. Reinforcement Learning with long short-term Memory. *Advances in Neural Information Processing Systems*, 2(14):1475–1482, 2002.

V. F. Farias, C. C. Moallemi, T. Weissman, and B. Van Roy. Universal Reinforcement Learning. *IEEE Transactions on Information Theory*, 56(5):2441–2454, 2010.

D. K. Haghighi, Supervised D. Precup, J. Pineau, and P. Panangaden. Learning Algorithms for Automata with Observations. Technical report, School of Computer Science, McGill University, Canada, 2007.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

M.P. Holmes and C. L. Isbell , Jr. Looping Suffix Tree-Based Inference of Partially Observable Hidden State. In *Proceedings of ICML. 2006*, pages 409–416. Apple Deve1oper Press, 2006.

M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability.* Springer, Berlin, 2005.

M. Hutter. Feature Reinforcement Learning: Part I: Unstructured MDPs. *Journal of Artificial General Intelligence*, 1:3–24, 2009.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

J. S. Liu. *Monte Carlo Strategies in Scientific Computing.* Springer, 2nd edition, 2008.

M. M. H. Mahmud. Constructing States for Reinforcement Learning. In J. Fürnkranz and T. Joachims, editors, *International Conference on Machine Learning*, pages 727–734. Omnipress, 2010.

R. A. McCallum. Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 387–395. Morgan Kaufmann, 1995.

P. Nguyen, P. Sunehag, and M. Hutter. Feature Reinforcement Learning in Practice. In *Proc. 9th European Workshop on Reinforcement Learning (EWRL-9)*, volume 7188 of *LNAI*, pages 66–77. Springer, September 2011.

P. Nguyen, P. Sunehag, and M. Hutter. Context Tree Maximizing Reinforcement Learning. In Jörg Hoffmann and Bart Selman, editors, *Proc. of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 1075–1082. AAAI Press, 2012.

M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

J. J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14(5):465–471, 1978.

P. Sunehag and M. Hutter. Consistency of Feature Markov Processes. In *Proc. 21st International Conf. on Algorithmic Learning Theory (ALT'10)*, volume 6331 of *LNAI*, pages 360–374, Canberra, 2010. Springer, Berlin.

R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA, USA, 1998.

I. Szita and A. Lörincz. The Many Faces of Optimism: A Unifying Approach. In *Proc. 12th International Conference (ICML'08)*, volume 307, Helsinki, Finland, 2008.

I. Szita, V. Gyenes, and A. Lőrincz. Reinforcement Learning with Echo State Networks. In *Proceedings of the 16th international conference on Artificial Neural Networks - Volume Part I*, ICANN'06, pages 830–839, Berlin, Heidelberg, 2006. Springer-Verlag.

J. Veness, K. S. Ng, M. Hutter, W. Uther, and D. Silver. A Monte Carlo AIXI Approximation. *Journal of Artificial Intelligence Research*, 40:95–142, 2011.

D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber. Solving deep memory POMDPs with recurrent policy gradients. In *Proceedings of the 17th international conference on Artificial neural networks (ICANN'07)*, pages 697–706, Berlin, Heidelberg, 2007. Springer-Verlag.

Appendix A. Table of constants

Experiment	α	*epochs*	*init-history*	*stop-explore*	*max-reward*	*anneal-temp*
Det Tmaze	0.1	500	20	100	400	1
Stoch Tmaze	0.1	500	100	100	400	1
Tiger	$1 \cdot 10^{-2}$	500	100	100	400	5
Locked Door	$1 \cdot 10^{-2}$	1,000	100	1,000	900	10

Table 1: The table lists the various constants used for each experiment. Common to all experiments were the maximum number of steps for a single annealing run capped at 50, the value of k in the exponential cooling scheme at 0.005, $\epsilon = 0.1$ and $\gamma = 0.99$. α is a parameter of the **Cost** that controls the balance between state and reward code-lengths, *anneal-temp* refers to the temperature T in the cooling schedule, *init-history* is the number of initial random actions performed by the agent, *stop-explore* is the epoch beyond which the agent no longer uses ϵ-exploration and *max-reward* is the value of the reward given to the garden-of-eden state in the extended MDP for all actions.

Appendix B. Algorithms

Algorithm 1: A high-level view of the generic ΦMDP algorithm.

```
Initialise $\phi$ ;
**Input** : Environment $Env()$;
Initialise history with observations and rewards from $t = init\_history$ random actions;
Initialise $M$ to be the number of timesteps per epoch;
**while** $true$ **do**
    $\phi = SimulAnneal(\phi, h_t)$;
    $s_{1:t} = \phi(h_t)$;
    $\pi = FindPolicy(s_{1:t}, r_{1:t}, a_{1:t-1})$ ;
    **for** $i = 1, 2, 3, ...M$ **do**
        $a_t \leftarrow \pi(s_t)$;
        $o_{t+1}, r_{t+1} \leftarrow Env(h_t, a_t)$;
        $h_{t+1} \leftarrow h_t a_t o_{t+1} r_{t+1}$;
        $t \leftarrow t + 1$;
    **end**
**end**
**return** $[\phi, \pi]$
```

Algorithm 2: Get current state given an observation sequence and a looping ST

```
getCurrentState(Observation sequence $o_{1:t}$);
currentNode = root;
$i = t$;
while currentNode is not a state do
    if currentNode has a loop then
        currentNode = node at the end of the loop;
    else
        currentNode = the $o_i$-th child of currentNode;
        $i = i - 1$;
    end
    if $i \leq 0$ then
        return $s^{empty}$;
    end
end
return currentNode
```

Algorithm 3: getNeighbour() method for looping ST

```
Input: num_ obs : number of observations, statelist : list of states in current tree,
looplist : list of loops in current tree
state = random state from current statelist;
Let $c$ be a random number in {1,2,3,4};
if $c == 1$ and (num_ states > num_ obs) and
every sibling of the current state is also a state then
    merge(state);
else if $c == 2$ and (num_ states > 2 × num_ obs) then
    if $uniform(0, 1) > 0.5$ and looplist ≠ {} then
        state = random state from looplist;
    end
    addLoop(state);
else if $c == 3$ and looplist ≠ {} then
    state = random state from looplist;
    removeLoop(state)
else
    split(state);
end
```