Feature Reinforcement Learning using Looping Suffix Trees

Mayank Daswani, Peter Sunehag and Marcus Hutter

Research School of Computer Science
Australian National University
{mayank.daswani, peter.sunehag, marcus.hutter}@anu.edu.au

Abstract

There has recently been much interest in using history-based methods using suffix trees to solve POMDPs. However, these suffix tree based methods cannot efficiently deal with 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 dependencies, while for deep memory environments it is competitive with LSTM-based methods.

1. Introduction

“Reinforcement learning (RL) is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment.”-[KLM96]. Traditional RL methods maximise the reward in a given unknown finite Markov Decision Process (MDP). Partially Observable MDPs (POMDPs) generalise the MDP framework. They assume that there is an underlying MDP that cannot be directly observed; the agent receives indirect observations, which are dependent on the state. Solving a POMDP problem is known to be PSPACE-complete even when the underlying MDP is known [PT87].

Feature Reinforcement Learning (ΦMDP) [Hut09] automates the extraction of a good state representation in the form of an MDP from the agent’s observation-reward-action history. This bypasses the POMDP problem and allows the use of existing traditional RL methods. [NSH11] 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) from [HIJ06] within the ΦMDP framework. While they deal with 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.

2. 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 [Mcc95], MC-AIXI-CTW [VNH+11], Active LZ [FMWVR07], CTΦMDP [NSH11] 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, [Mah10] aims at searching the very large space of probabilistic deterministic finite automata (with some restrictions).

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 [HS97]. The LSTM framework was first proposed to predict timeseries data with long-term dependencies. This was first adapted to the RL context by [Bak02] and more recently a new model-free variant based on policy gradients by [WFPS07]. These methods are more often used in the continuous case, but were also tested in the discrete setting.

3. Preliminaries

Agent-Environment Framework. The notation and framework from [Hut04] is used here. An agent acts in an Environment Env. It chooses from actions $a \in A$, and receives observations $o \in O$ and real-valued rewards $r \in R$ where $A, O$ and $R$ are all finite. This observation-reward-action sequence happens in cycles denoted 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 $H := (O \times R \times A)^* \times O \times R$. 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 : $H \rightarrow A$ where Agent($h_t$) = $a_t$. Similarly, the environment can be viewed as a stochastic function of the history, Env : $H \times A \rightarrow O \times R$, where Env($h_{t-1}, a_{t-1}$) = $o_t$ and real-valued rewards $r_t$. The symbol $\sim$ indicates a possibly stochastic transition. A policy is defined as a map $\pi : H \rightarrow A$.

If $Pr(o_t | h_t, a_t) = Pr(o_t | a_{t-1} a_t)$, the environment is said to be a discrete Markov decision process (MDP) [Put94]. In this case the observations form the state space of the MDP. Formally an MDP is a tuple $\langle S, A, T, R \rangle$ where $S$ is the set of states, $A$ is the set of actions $R : S \times A \sim R$ is the (possibly stochastic) reward function which gives the (real-valued) reward gained by the agent after taking action $a$ in state $s$. $T : S \times A \times S \rightarrow [0, 1]$ is the state-transition function. $R$ is the expected instantaneous reward, $R(s, a) = E\{r_{t+1}|s_t = s, a_t = a\}$. The agent’s goal is to maximise its future discounted expected reward, where a geometric discount function with rate $\gamma$ is used. The value of a state according to a stationary policy is given $V^\pi(s) = E_\pi \{R_t|s_t = s\}$ 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 [SB98], in the unknown case the agent must deal with the problem of exploration vs exploitation, which only has approximate efficient solutions.

$\Phi$MDP. [Hut09] is a framework that extracts relevant features from the history for reward prediction. The framework gives us a method to find a map $\phi : H \rightarrow S$ 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 [Ris78], 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) to find the optimal map. The modified cost (used in [NSH11]) adds a parameter $\alpha$ to control the balance between reward coding and state coding,

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

$\text{CL}(s_{1:n} | a_{1:n})$ 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. $\text{CL}(r_{1:n} | s_{1:n}, a_{1:n})$ follows similarly.
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.

[NSH11] search the map space of suffix trees (explained below). Our method extends this to looping suffix trees. The agent is first initialised with some history based on random actions. Then it alternates between finding a “best” $\phi$ using the SimulAnneal algorithm and performs actions based on the optimal policy for that $\phi$ found via the FindPolicy function. For a detailed description of our algorithms please see [DaMH12]. 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 in [Hut09] which is based on [SL08]. This method adds an additional “garden of eden” state $(s_e)$ to the estimated MDP, which is an absorbing state with a very high reward. This encourages systematic exploration of unknown states.

**Definition 1 (Suffix Tree)** Let $O = \{o^1, o^2, o^3, ..., o^d\}$ 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 $O$. Every suffix tree has a corresponding suffix set which is the set of strings $S = \{s^1, s^2, ..., s^n\}$ 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. We distinguish between Markov and non-Markov trees using the terminology from [NSH11]. These may be more commonly known as FSMX and non-FSMX sources [Ris83]. A tree is Markov if for every $s_i \in S$ and $o_j \in O$, there exists a unique suffix for $s_i o_j$. If the suffix remains undetermined (i.e. more than one previous state is required to determine the current state) the tree is non-Markov. Note that any non-Markov tree can be made Markov by adding appropriate splits to the tree. The non-Markov tree will be a sub-tree of this new tree.

**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 $O = \{o^1, o^2, o^3, ..., o^d\}$ is a $d$-ary looping tree in which edges coming from each internal node are labelled by the elements of $O$. 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 we simply follow the edges in the tree until we get to a state. See [DaMH12] for details. 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 an example 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 $s^2$. 

3
HIJ06 considers a deterministic POMDP with actions and observations but no rewards. The problem task considered is to find a predictive model that “can predict the observation following any action given the history preceding the action”. The paper constructs an algorithm that returns a finite looping ST that given a sufficient history correctly predicts the transitions following any resolving history.

4. 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. Unfortunately the loopability criterion in HIJ06 cannot work in the stochastic case since a loop can change not only the possible transitions but also the transition probabilities. HIJ06 restrict their discussion to the deterministic case without rewards.

The cost function of the ΦMDP framework immediately gives us a well-motivated criterion for evaluating looping suffix trees. We now extend this framework to the stochastic case with the Environment including rewards as defined previously. 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 trees is also useful in deterministic environments, where in some cases a smaller stochastic tree 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 same manner.

Algorithm. The algorithm consists of a specification of CL(φ) and the neighbourhood method, that can be easily plugged into the generic ΦMDP algorithm. We call our algorithm LSTΦMDP. A tree with num_nodes can be coded in num_nodes bits [VNH+11, Sec.5] and the starting and ending nodes of all loops can be coded in $2 \log(\text{num_nodes}) \times \text{num_loops}$, so we define the model cost of the map $CL(\phi)$ as

$$CL(\phi) = \text{num_nodes} + 2 \log(\text{num_nodes}) \times \text{num_loops}.$$ 

The getNeighbour() method first selects a state randomly. For a given state it attempts to split or merge with probability 0.5. If a merge is not possible then it splits. Merges are not allowed to transition to the root tree. In addition to splitting or merging, the algorithm also attempts to add or remove a loop to a newly selected random state/looping node with equal probability. The addition of a loop to a looping node increases the length of the loop by one. Similar to the condition for merging, the algorithm is not allowed to add a loop unless there are more states than twice the number of observations. Both these conditions are to ensure that the tree does not get too small early on. The four operations merge, split, addLoop and removeLoop are now explained.

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 $s^i = o^i n'$ where $o^i \in O$ and $n'$ is the 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 $|O|$ new states of the form $o^i s^i$ for all $o^i \in O$ and removing the state $s^i$.

**addLoop** : The addLoop function has two cases. We can add a new length-one loop (i.e. a loop to the direct parent) to a current state thus removing it from the state set and adding it to the loop set. Alternatively, we extend the length of an existing loop by one (i.e. to the parent of the existing loop).

**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 $\Phi$MDP 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 5), 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. Note that if adding a loop of length $n - 1$ makes a tree inconsistent, then so does a loop of length $n$.

The initial state is always set to be the depth one tree (i.e. one split). The root node is not used. This is because the root tree is initially very attractive and results in greater inertia in changing the map. Additionally, the consistency of suffix trees was shown in reference to the Cost function, only by neglecting the root tree [SH10].

The space of looping STs includes the space of ordinary STs. Therefore, results from the non-looping case [NSH11] 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 $\alpha$ for this to be the case. This is further discussed in Section 5.

5. Experiments

In this section we describe our experimental setup and the domains that we tested on. Each domain was used to test a different ability of the algorithm. Every experiment was run 50 times. The agent is given a history produced by 100 random actions to start with. Each run of an experiment was conducted over 500 epochs (except for “Locked door”), with each epoch containing 100 iterations of the agent performing actions according to it’s current policy, based on the current map with a constant $\epsilon$-exploration of 0.1 until the 400th epoch. After every epoch, the agent was given a chance to change its optimal map via a simple 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 $[0.6, 0.7]$.

![Figure 2: TMaze environment showing goal at left](image)

**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.
(a) #optimal runs, varying corridor lengths for LSTF, RPG and RL-LSTM
(b) Deterministic TMaze length 50, with \( \alpha = 0.1 \)
(c) A reward-optimal LST for the TMaze problem

We use the formulation as described in [Bak02]. 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 \( O = \{0, 1, 2, 3\} \), the rewards are \( R = \{-0.1, 4\} \) 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.

We conducted experiments on two variants of TMaze. In the first variant, the observation received at the start determines where the positive reward lies every time. In the second variant (stochastic TMaze) the observation received at the start is correct 80% of the time. In each variant 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.

We compare our LSTFMDP to RL-LSTM [Bak02] and Recurrent Policy Gradients (RPG) [WFPS07] on the deterministic variant as follows. We increase the length of the corridor systematically from 10 to 100. 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 \( \epsilon \)-exploration. Note that 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.

Generating from a looping ST. The environment in the next set of experiments is itself a looping suffix tree used as a source. This is useful in testing the ability of the agent to recover a useful representation from environments with loops. For every run of the experiment we randomly generate a binary looping ST using 50 random neighbour operations. A typical tree can be of depth 10 and include several loops of length up to 5. The tree is then used as a stochastic source i.e. we assign emission probabilities for each observation and reward to every (state,action) pair. The emission probabilities are picked randomly from \([0, 1]\). The generating tree then emits observations and rewards according to the state it is currently in and transitions to a new state. The agent receives a reward of 1 if the action matches the next observation and 0 if it does not. At the end of each experiment we evaluate the generating tree itself in the same way as the agent (without the stochastic search).

Tiger. The Tiger domain is a familiar domain in the reinforcement learning literature. There are two doors and behind one door is a tiger, while the other hides a pot of gold.
The agent must decide which door to open. The agent has a noisy action "Listen" available to it that allows it to tell which door the tiger’s growl is coming from. See for example [NSH11] for more details. The experiment in this case was to compare to the performance of CTΦMDP.

**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 5x4 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 -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.

6. Analysis

In this section we analyse the results from our experiments, and provide explanations for 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 representation, 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 3 for details. The results of the separate experiment comparing the algorithms performance on varying corridor lengths are displayed in Figure 3. In almost all cases the agent reaches the optimal policy, with a few corridor lengths having one run stuck on traversing the length of the corridor. Note that the algorithm does not necessarily reach the optimal tree, but finds a reward-optimal tree that contains it. In comparison, RL-LSTM [Bak02] has increasingly many suboptimal runs as the length of the corridor increases past 50. RPG [WFPS07] has optimal results up to length 90 but has 3 unsuccessful runs at length 100. Additionally, our algorithm uses 50000 iterations (500
epochs) in all cases, while RPG uses around 2 million iterations for corridor length 100. Note that these results are taken from the respective papers. We also tested CTΦMDP but it was not successful for corridor lengths >5. On further testing our algorithm finds the optimal policy until corridor length 180, after which rewards simply occur too rarely and it fails completely. 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 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 expected value of the optimal policy in the stochastic Tmaze case is -0.0344. The agent achieves this value once the ε-exploration has been removed. See Figure 4 for details.

**Generating from a looping ST.** Since the tree in every experiment is different, the maximum achievable reward varies in each experiment. However taking an average over all the experiments still reveals some trends. In Figure 4 we observe that the agent surprisingly appears to perform much better than the generating tree on average. This can be explained by noting that the generating trees may not possess the Markov property. It is possible that the agent is finding Markov completions of the non-Markov source tree, which allow for better prediction.

**Tiger.** The Tiger example is interesting since it shows that the agent can still reproduce results from the regular non-looping suffix tree case without much effort. Only the parameter α needed to be set to a lower value of 0.01 and the agent achieves the optimal reward. Figure 4 shows the comparison between the looping and non-looping versions of ΦMDP as being nearly identical.

**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 collects the key. The LSTΦMDP agent with α = 0.1 succeeds in finding a near-optimal policy in half the runs. CTΦMDP succeeds in learning how to avoid walls but does not improve further in 1000 epochs. See the figure to the right for the graph of the near-optimal runs of LSTΦMDP.

**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. This is necessary due to a combination of the properties of looping STs and the exacerbated exploration-exploitation problem. 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.
7. Conclusion

We introduced looping suffix trees to the feature reinforcement learning framework. The experimental results show that looping suffix trees are particularly useful in representing long-term dependencies by looping over unnecessary observations. The competitiveness with LSTM-based algorithms on discrete domains was demonstrated. The algorithm was also able to replicate results on short-term environments from CTFMDP.

References


