Technical Papers

A Logarithmic-time Updating Algorithm for TD(λ) Learning†

Susumu Katayama* Shigenobu Kobayashi*

* Graduate School of Interdisciplinary Science and Engineering, Tokyo Institute of Technology, Yokohama 226-8502, Japan.

Received October 16, 1998.

Keywords: eligibility traces, computational complexity, temporal-difference learning, reinforcement learning, replacing eligibility traces

Summary

Temporal-difference (TD) method is an incremental learning method for long term prediction problem. Most reinforcement learning methods are based on it. So as to cope with partial observability, we have to combine it with the idea of eligibility traces, which causes the matter of time complexity. There are some conventional ways to reduce it, which are unavailable in environments where there may be long delay between observations and their consequent rewards.

In this paper we propose an algorithm which accurately computes TD(λ) updating in logarithmic time. It can safely be used for all kinds of environments, because it is proved to give the accurate TD prediction.

We also apply our algorithm to Sarsa(λ), which is a reinforcement learning method using eligibility traces. We can also apply it to Q(λ)-learnings. The accumulating Sarsa(λ) usually takes time linear in the number of the actions for action selection. There exists two definitions of replacing Sarsa(λ), the more common and better one of which can be computed in time logarithmic in the number of the observations and that of the actions, owing to a device.

1. Introduction

1 · 1 Background

Temporal-difference (TD) method [Sutton 88] is a prediction method based on the Bellman equation. TD method can obtain the value for each state (or the expectation of the discounted sum of the rewards obtained after the state) from time sequences of states and rewards, without knowledge about the state transition and the reward distribution after each state. Most popular reinforcement learning algorithms including Q-learning [Watkins 92], Sarsa [Dorigo 94], [Sutton 96], and actor-critic [Barto 83] are based on TD.

TD(0), which is the simplest form of TD method, assumes that the environment forms a Markov decision process. In order to apply TD method to non-Markovian environments one needs to let $\lambda > 0$, or to make use of eligibility traces. Also, even for a Markovian environment with a lot of delay in rewards, one can expect quicker convergence by letting $\lambda > 0$. Thus, TD(λ) that uses eligibility traces has a broader range of applications.

One problem on the usage of eligibility traces is about time complexity. TD(0) updating can be computed in time logarithmic in the number of states, while TD(λ) with eligibility traces requires

† This is an unofficial English translation of a published paper in Japanese. This English version has never been reviewed, although it should be exact translation from the Japanese version, which has passed the reviewing process of the Journal of JSAI.

linear time. Thus, on the occasion of solving large real problems the improvement of the time complexity of TD(λ) updating is necessary.

For the purpose of improving the time complexity of TD(λ) updating, truncated temporal differences (TTD) procedure [Cichosz 95] was proposed. Also, [Sutton 98] presents a way to reduce the time complexity. However, those methods are not effective for the cases where there is a long delay between each reward and its eligible states. In this paper we propose a logarithmic-time implementation algorithm that is assured to compute the exact TD prediction with eligibility traces for any environment.

1 · 2 Outline

In **Chapter 2** we define TD(λ), and explain its problem on time complexity and its conventional remedy. In **Chapter 3** we explain how TD(λ) updating can be computed in logarithmic time. In **Chapter 4** we explain how to apply the method described in **Chapter 3** to a reinforcement learning method Sarsa(λ). In **Chapter 5** we discuss a good effect of our algorithm, and conclude the paper. In **Chapter A** we prove that our algorithm exactly computes TD(λ).

2. The Complexities of TD(λ)

The time and space complexities of an algorithm are the numbers of the required reduction steps and memory units to complete the computation respectively. The order of these values are known to be invariable to the difference in the computation models. We should measure the computation cost of reinforcement learning algorithms with such complexities because they are not applied to many real world problems.

In this chapter we define TD(λ) updating and discuss its complexities in **Sec. 2·1**. Then, in **Sec. 2·2** we review conventional methods for efficient implementation of TD(λ).

2 · 1 The definition of TD(λ)

[1] TD(0)

TD method is a learning method for long term prediction in Markovian environment, and it is the most popular method for value estimation in reinforcement learning. Firstly we explain TD(0) that is the most simple form of TD. TD(0) asymptotically estimates the value function $V^*(y)$ for state y via V_t in

$$V_t(y) = \begin{cases} V_{t-1}(y) + \alpha\delta_t, & \text{if } y = y_{t-1}; \\ V_{t-1}(y), & \text{o.w.} \end{cases} \quad (1)$$

$$\delta_t = r_t + \gamma V_{t-1}(y_t) - V_{t-1}(y_{t-1}) \quad (2)$$

Here y_t and r_t denotes the state and reward at time t respectively. [*1] δ is called TD error.

TD(0) asymptotically obtains

$$V^*(y) = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{i+1} \,\middle|\, y_1 = y\right]$$

as $V_t(y)$ for finite ergodic Markov chain.

[2] TD(λ)

TD(λ) is an extension of TD(0) to partially observable environments in general. TD(λ) is defined as follows:

$$V_t(y) = V_{t-1}(y) + \alpha D_t(y)\delta_t \quad (3)$$

$$\delta_t = r_t + \gamma V_{t-1}(y_t) - V_{t-1}(y_{t-1}) \quad (4)$$

where D denotes eligibility traces that assign more TD error to more recently visited observations. As eligibility traces either of the following two is usually used:

i. accumulating traces

Accumulating trace D^A is defined as follows: [*2]

$$D_1^A(y) = 0 \quad (5)$$

$$D_t^A(y) = \begin{cases} \lambda D_{t-1}^A(y) + 1, & \text{if } y = y_{t-1} \\ \lambda D_{t-1}^A(y), & \text{o.w.} \end{cases} \quad (6)$$

The more frequently a state is visited, the larger its accumulating trace is. Thus in accumulating traces the information of frequency and that of recency are merged.

---

*1 We use the same t both as the suffix of state and reward and as that of the value function, which does not mean that we need to share the same time but that we can do so.

*2 The usage of $\lambda\gamma$ instead of λ is more common, in which case the same discussion is true by replacing all λ with $\lambda\gamma$.

ii. replacing traces

[Singh 96] proposed replacing traces which only preserves the information of recency.

Replacing traces D^R are defined as follows:

$$D_1^R(y) = 0 \tag{7}$$

$$D_t^R(y) = \begin{cases} 1, & \text{if } y = y_{t-1} \\ \lambda D_{t-1}^R(y), & \text{o.w.} \end{cases} \tag{8}$$

[Singh 96] empirically shows that replacing traces provide better convergence than accumulating traces.

[3] Complexities of TD(λ)

Let $|Y|$ denote the number of states. The time complexity of TD(0) updating is $O(\log |Y|)$, and its space complexity is $O(|Y|)$.

The time complexity of the naive implementation of (3) with $\lambda > 0$ is $O(|Y|)$ and its space complexity is $O(|Y|)$.

Note that the time complexity of one updating is linear in the number of states. It is not small enough because this computation is repeated until convergence.

2 · 2 Conventional approaches

[1] TTD(λ, m)

Naively implemented updating rules using eligibility traces such as TD(λ) and Sarsa(λ) described later require time complexity linear in the number of states or state-action pairs. TTD procedure [Cichosz 95] computes something like TD(λ) and its implementation is simpler.

TD(λ) with accumulating traces estimates the TD return z_t^λ:

$$z_t^\lambda = \sum_{k=0}^{\infty} \lambda^k [r_{t+k} + (\gamma - \lambda) V_{t+k-1}(y_{t+k})] \tag{9}$$

while TTD(λ, m) estimates m-step truncated TD return:

$$z_t^{\lambda,m} = \sum_{k=0}^{m-1} \lambda^k [r_{t+k} + (\gamma - \lambda) V_{t+k-1}(y_{t+k})] + \lambda^m V_{t+m-2}(y_{t+m-1}) \tag{10}$$

instead of estimating the TD return defined with (9).

[Cichosz 95] proposes the following efficient implementation method for (10). (10) can be decomposed into

$$S_t^{\lambda,m} = \sum_{k=0}^{m-1} \lambda^k [r_{t+k} + (\gamma - \lambda) V_{t+k-1}(y_{t+k})] \tag{11}$$

and

$$T_t^{\lambda,m} = \lambda^m V_{t+m-2}(y_{t+m-1}), \tag{12}$$

and thus $S_t^{\lambda,m}$ can be computed with

$$S_t^{\lambda,m} = \frac{1}{\lambda}[S_{t-1}^{\lambda,m} - (r_t + (\gamma - \lambda) V_{t-2}(y_{t-1})) + \lambda^m (r_{t+m} + (\gamma - \lambda) V_{t+m-2}(y_{t+m-1}))] \tag{13}$$

TTD procedure can be regarded as TD(λ) whose reward is a weighted average of $1 \ldots m$ step return which is obtained via recording m-step sequence. m must be decided in advance, which has the following problems:

- Because the environment is assumed to be m-Markovian rather than hidden Markovian, small m cannot cope with such environment as depicted in Fig. 1.
- Because it learns m steps later after state observation, too large m deteriorates the convergence rate.
- (13) may accumulate errors for $\lambda^m \ll 1$. *3 In addition, without using (13) the time complexity is linear in m.

[The complexities of TTD procedure] The time complexity of TTD procedure is $O(\log m |Y|)$ with (13) and $O(m + \log |Y|)$ without (13). The space complexity is $O(m \log |Y| + |Y|)$.

[2] The solution described in [Sutton 98]

[Sutton 98] suggests that the actual number of updatings required can be made small, taking advantage of the sparseness of the traces in many cases.

If the algorithm checked whether the trace for each state is 0 or not, the sparseness would be almost useless for efficiency. More efficient implementation would preserve at each time step t the set of states

*3 (13) does not use r_{t+m} to compute S after adding $\lambda^{m-1}(r_{t+m} + (\gamma - \lambda) V_{t+m-2}(y_{t+m-1}))$ to $S_t^{\lambda,m}$ at time $t+m$ until m-step later $S_{t+m}^{\lambda,m}$ is obtained from this value. Moreover, $\lambda^m \ll 1$ may cause $\lambda^{m-1}(r_{t+m} + (\gamma - \lambda) V_{t+m-2}(y_{t+m-1})) \ll S_t^{\lambda,m}$. Thus, $S_{t+m}^{\lambda,m}$ obtained later is not assured to be exact.

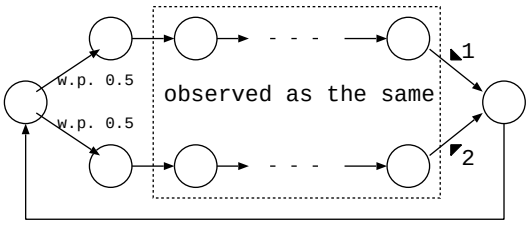

Fig. 1 An example case where TTD does not work well Each circle describes each states, and each number with filled triangle describes the reward obtained during the transition. All the states except those in the dotted box are observed differently.
TTD does not work well unless the environment is m-Markovian, while TD(λ) with replacing traces with large λ can discriminate between the two observations with different values.

$$D_t(y) \max_{r \in R} |r| \ll |V_t(y)| \tag{14}$$

whose traces are too small to change the values when updating, where R denotes the set of possible reward values.

In this case the time complexity of one updating is $O(\min\{|Y|, m\} \log |Y|)$, where m denotes the small real value that satisfies $\lambda^m \ll 1$. Thus, the implementation is not fast with large λ.

3. Novel Approach: the logarithmic-time implementation algorithm for TD(λ) updating

In this chapter we propose a logarithmic-time and linear-space algorithm which computes TD(λ) accurately.

3 · 1 Data structure

Assume a tree that has the same number of leaf nodes (or leaves) as the number of $|Y|$, and assign each state to some leaf exclusively*4. By regarding each state y in the same light as its corresponding leaf L we can define $D_t(L)$ and $V_t(L)$. The algorithm lets each node preserve some information, and it updates the tree (by updating the information) in time linear in the depth of the tree (Fig. 2). If $|Y|$ is finite by adopting a minimal tree

*4 Just like other divide-and-conquer algorithms, this algorithm enforces the tree structure to the set of states — the tree need not reflect the actual state structure.

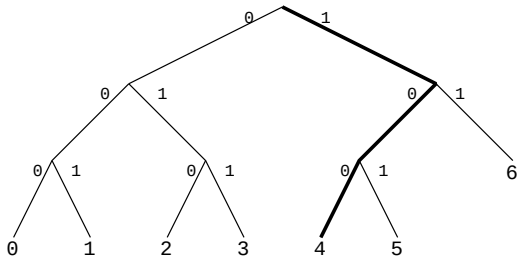

Fig. 2 The correspondence between the tree and each state. For example, the computation about the state 4 costs 3 (= the depth of the tree) times some constant.

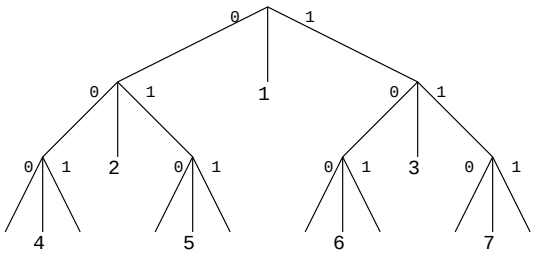

Fig. 3 In the case where the number of states is infinite.

the depth is $O(\log |Y|)$, and we can update the tree in $O(\log |Y|)$ time and $O(|Y|)$ space. If $|Y|$ is infinite countable by adopting a ternary tree (Fig. 3) which has leaves at each node of an infinite binary tree we can update the tree in $O(l_t)$ time and $O(\exp \max_{0 \le i \le t} l_i)$ space where l_t is the observation length.

We can regard the sequence describing "which child node to select" at each node in the same light as its corresponding state, because the route from the root node to each leaf is unique. For example state 4 in Fig. 2 can be represented as $[1, 0, 0]$.

For time step t we call each node in the route to the current state y_t at time t *visited node at time* t and other nodes *non-visited nodes at time* t. $S_t(N)$ means that N is a visited node at time t.

$f(N)$ denotes the visited child node of a visited node N. Also, $f_y(N)$ denotes $f(N)$ assuming the state y is visited.

$\tau_t(N)$ denotes *the last time to visit the node* N *before time* t. defined as follows:

$$\tau_1(N) = -\infty \tag{15}$$

$$\tau_t(N) = \begin{cases} t-1, & \text{if } y = S_{t-1}(N) \\ \tau_{t-1}(N), & \text{o.w.} \end{cases} \tag{16}$$

$\top$ denotes the root node and $h(N)$ denotes the parent node of N.

3 · 2 *Lazy updating*

The definition of TD(λ) requests that as long as $\lambda > 0$ $V(y)$ for all states y must be updated at every time step. The naive implementation of the overall updating costs at least $O(|Y|)$ time where Y is the set of states.

Actually, however, the instant values for all states are not needed at every time step. Considering that even if we always have all the instant values for all the states the theoretical access time to each value is at least $O(\log|Y|)$, we say an algorithm consuming only

- $O(\log|Y|)$ time for updating the data the program maintains and
- $O(\log|Y|)$ time for accessing each $V(y)$

is a logarithmic-time computation algorithm for every step. Now we call it *lazy updating* to update least at every time step and not to update $V(y)$ itself until y is visited

3 · 3 *Logarithmic-time implementation algorithm*

In this section we describe our logarithmic-time implementation algorithm with the idea behind it. Since this section is devoted for intuitive comprehension of the proof in **Chapter A**, for its complete understandings you should refer to **Chapter A**.

Because the information at each leaf is updated when it is visited, the value stored at each leaf L at time t is not $V_t(L)$ but $V_{\tau_{t+1}(L)}(L)$ that is the value at the last visit to L, where $\tau_t(N)$ denotes *the last time to visit the node N before time t*. We let $p_t(L)$ denote $V_{\tau_{t+1}(L)}(L)$. In the same manner, $z_t(L)$ denotes the eligibility trace of L's corresponding state at the last visit to L.

The number of the visited nodes is equal to the depth of the tree. In addition, each visited node has a non-visited child node whose descendants are all non-visited. Therefore a minimal binary tree has the same number of "non-visited subtrees" (subtrees consisting of non-visited nodes) as its depth. It follows that if we put some pieces of information at the root nodes of the non-visited subtrees about what operations to apply to all the nodes in the non-visited subtrees, and can update the pieces of information in constant time per piece, we can update the whole binary tree in time linear in its depth (or in $O(\log|Y|)$ time) (Fig. 4).

One piece of information to be put at the root node of the non-visited subtree is the addend to p's in each non-visited subtree, or the discounted sum of the addends while it has been non-visited. Since it is cleared to 0 when the subtree is visited, the sum at time t is

$$\alpha \sum_{i=\tau_t+1}^{t} \lambda^{i-1-\tau_t}\delta_i \tag{17}$$

. We call the sum *put-off update* and let $\tilde{s}_t$ denote it.

$\tilde{s}_t$ is a summation until t. In actual computations the information put at the node is not updated until its parent node will be updated. Therefore at node N we do not have the instant $\tilde{s}_t$ but its difference from that of the parent node. We let s_t denote the difference.

The put-off update of a node needs to be added to the p's of all of its descendants after being discounted depending on the time passed since the last visit to the node. The rate for discounting can be computed by preserving

$$\tilde{d} = \lambda^{t-\tau_t}$$

at each node. Note that at each leaf $\tilde{d}$ corresponds to its eligibility trace.

Even as for $\tilde{d}$ we have to preserve only some differential thing, which is called *recency gap* d, that is $\tilde{d}$ divided by $\tilde{d}$ at its parent node.

From above discussion, the logarithmic-time implementation algorithm for TD(λ) with replacing trace is described in Fig. 5. Also, for z such that

$$z_0(L) = 1$$

$$z_t(L) = \begin{cases} \lambda D_t^R(L) z_{t-1}(L) + 1, & \text{if } S_t(L); \\ z_{t-1}(L), & \text{o.w.} \end{cases}$$

we obtain

$$D_t^A(L) = z_t(L) D_t^R(L) \tag{18}$$

Thus, we can also compute TD(λ) with accumulating traces.

Note that the value for the current state is ob-

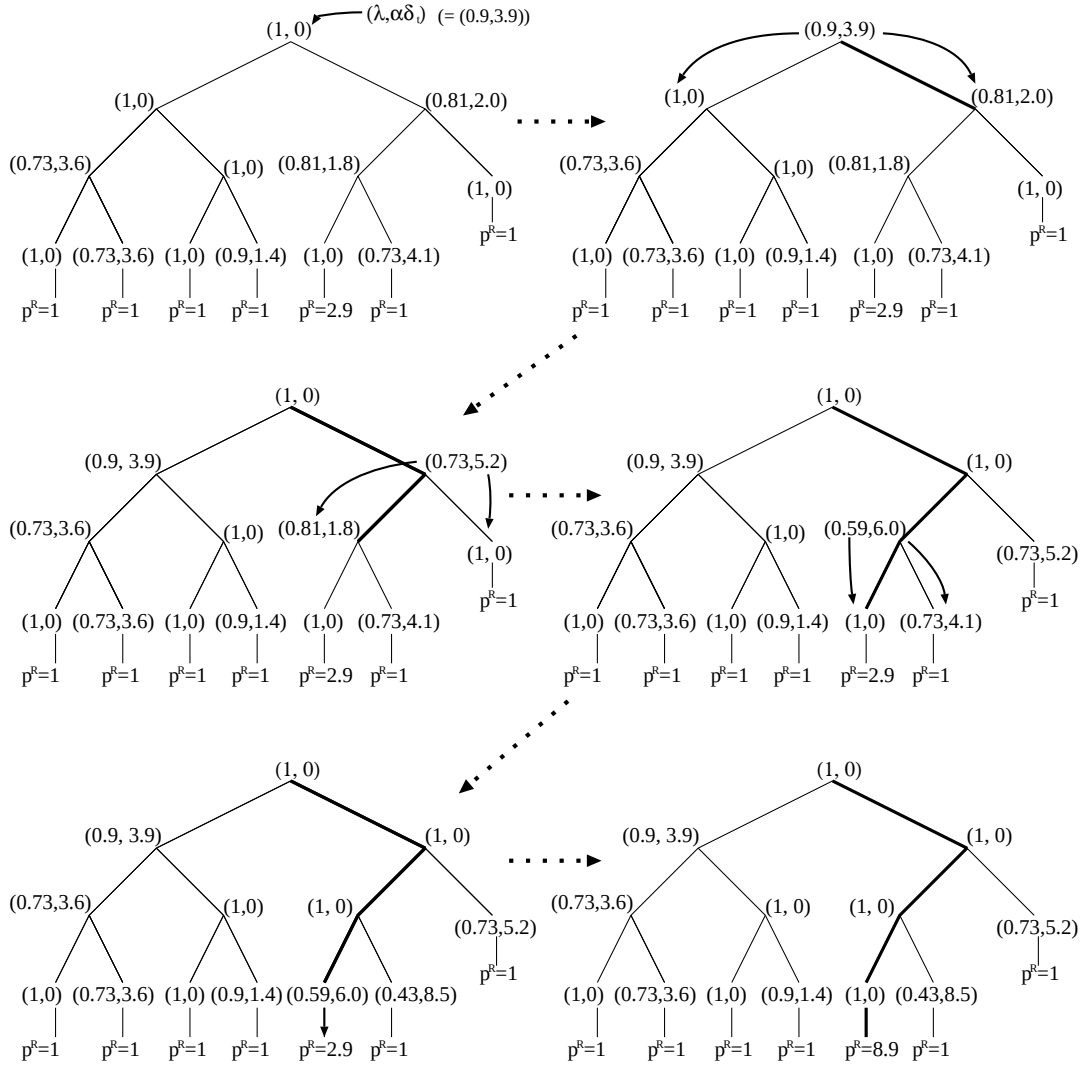

Fig. 4 Computation flow in the proposed algorithm.
Each tuple (d, s) means that the recency gap is d and the put-off update is s. When a state is observed, the algorithm collects the information of its value decentralized to each selected nodes, updates relevant nodes, and obtains its new value as new p. The values of the other states can be obtained through putting $(1, 0)$ into the tree after updating and following down the route to the states without updating the tree.

tained via $p^R(L)$ or $p^A(L)$, where L denotes the leaf corresponding to the current state.

The naive implementation of Fig. 5 requires linear time for initialization, and is unable to cope with environments with infinite number of states. However, lazy initialization, that is, postponing the initialization of each node until it is first visited, resolves this problem.

We can prove that the algorithm described above satisfies the specification of TD(λ) defined with (8). The proof is described in **Chapter A**.

3 · 4 The complexities of the proposed method

Here we discuss the complexities of the proposed method with a minimal binary tree.

Since there is no need to preserve pointers at each node in this case, the algorithm uses $6|Y|x$ space,[*5]

*5 Actually it can be compressed to $4|Y|x$ because $d = 1$ and $s = 0$ in half of the nodes.

Each node N preserves its recency gap $d(N)$ and its put-off return $s(N)$. In addition, each leaf L preserves either $p^R(L)$ for replacing trace version, or $p^A(L)$ and $z(L)$ for accumulating trace version.

[Tree Initialization]
For each node N
  $d(N) \leftarrow 0$, $s(N) \leftarrow 0$
and for each leaf N
  $p^R(N) \leftarrow v_0(N)$, $p^A(N) \leftarrow v_0(N)$, $z(N) \leftarrow 1$
where $v_0(N)$ denotes the initial value for the state corresponding to N.

[Tree Updating Rule]
Obtain δ_t from (4)
$N \leftarrow \top$, $d(\top) \leftarrow \lambda$, $s(\top) \leftarrow \alpha\delta_t$
Repeat
  for each child node n of N
    $s(n) \leftarrow s(N)d(n) + s(n)$, $d(n) \leftarrow d(N)d(n)$
  $d(N) \leftarrow 1$, $s(N) \leftarrow 0$
  $N \leftarrow f(N)$
Until N is a leaf
$p^R(N) \leftarrow p^R(N) + s(N)$
$p^A(N) \leftarrow p^A(N) + z(N)s(N)$
$z(N) \leftarrow 1 + d(N)z(N)$
$d(N) \leftarrow 1$, $s(N) \leftarrow 0$
(Now $V_t^R(y_t) = p_t^R(N)$ in replacing version, and
$V_t^A(y_t) = p_t^A(N)$ in accumulating version.)

[Computation of $V_t(y)$]
$V_t(y)$ can be obtained between the tree updating with δ_t and that with δ_{t+1}.
$N \leftarrow \top$,
$\tilde{s} \leftarrow 0$
Repeat
$\tilde{s} \leftarrow \tilde{s}d(f_y(N)) + s(f_y(N))$
$N \leftarrow f_y(N)$
Until N is a leaf.
$V_t^R(y) = p^R(N) + \tilde{s}$ in replacing version, and
$V_t^A(y) = p^A(N) + z(N)\tilde{s}$ in accumulating version.

Fig. 5 Our logarithmic-time implementation algorithm for TD(λ)
For simplicity we wrote the replacing trace version and the accumulating trace version at the same time. In actual implementation you cannot compute both at the same time with the same instance of the tree because δ_t differs between the replacing version and the accumulating version. Also, there is no need to compute $z(N)$, $p^A(N)$, and $V^A(y)$ for the computation of the replacing trace version, and there is no need to compute $p^R(N)$ and $V^R(y)$ for the computation of the accumulating trace version.

where x is the size of the real number used.

The computation of $V(y)$ at each time step requires $\log_2|Y|$ additions, $\log_2|Y|$ multiplications, and $2\log_2|Y|$ memory readings. The updating of the tree including the computation of $V_{t-1}(y_t)$ which is required in the computation of δ_t requires $5 + 3\log_2|Y|$ additions, $4 + 5\log_2|Y|$ multiplications, $4 + 4\log_2|Y|$ memory readings, and $4 + 4\log_2|Y|$ memory writings.

4. Application of the proposed method to Sarsa(λ)

In this chapter we describe how to apply the algorithm described in **Chapter 3** to a reinforcement learning algorithm Sarsa(λ).

4 · 1 Fast implementation of Sarsa(λ)

Sarsa(λ) is a TD(λ) for control regarding state-action pairs as states. The name Sarsa is from [Sutton 96], and it is taken from the initials of the variables in the updating rule, state, action, reward, next state, and next action. Thus, the updating rule is

$$Q_t(y,a) = Q_{t-1}(y,a) + \alpha D_t(y,a)\delta_t \tag{19}$$

$$\delta_t = r_t + \gamma Q_{t-1}(y_t,a_t) - Q_{t-1}(y_{t-1},a_{t-1})$$

$$D_t^R(y,a) = \begin{cases} 1, & \text{if } y = y_{t-1} \text{ and } a = a_{t-1}; \\ \lambda D_{t-1}^R(y,a), & \text{o.w.} \end{cases} \tag{20}$$

$$D_t^A(y,a) = \begin{cases} \lambda D_{t-1}^A(y,a) + 1, & \\ \qquad \text{if } y = y_{t-1} \text{ and } a = a_{t-1}; & \\ \lambda D_{t-1}^A(y,a), & \text{o.w.} \end{cases} \tag{21}$$

$$D \in \{D^A, D^R\} \tag{22}$$

By regarding each state-action pair as each state, the algorithm described in the previous chapter can be applied to Sarsa(λ). If the action selector uses the values for all actions available, however, $O(|A|\log|Y|)$ time is required because there is no algorithm found that can make action selection in $O(\log|Y||A|)$ time in the Sarsa defined with (19), ..., (22).*6 The space complexity is $O(|Y||A|)$.

Another option of TD(λ) control is $Q(\lambda)$ learning. $Q(\lambda)$ learning is for obtaining the best policy that is independent of the adopted exploration strategy.

*6 The time complexity is the same as that of the naive implementation of one-step Q-learning and the application of TTD to Q-learning described in [Cichosz 95] where m is small enough to be ignored. Thus, this may not be so critical.

There are three versions of $Q(\lambda)$, namely, Watkins's $Q(\lambda)$ [Sutton 98], [Peng 94]'s $Q(\lambda)$, and naive $Q(\lambda)$ ([Sutton 98], [Cichosz 95]). The difference in implementation is small between these $Q(\lambda)$'s and Sarsa(λ), and thus $Q(\lambda)$ learning can also be implemented in $O(|A|\log|Y|)$ time and $O(|Y||A|)$ space.

4 · 2 Fast implementation of Sarsa(λ) in [Singh 96]

[Singh 96] suggests that the following replacing traces make Sarsa(λ) quicker to converge than accumulating traces and straightforward replacing traces (20).

$$D_t(y,a) = \begin{cases} 1, & \text{if } y = y_{t-1} \text{ and } a = a_{t-1}; \\ 0, & \text{if } y = y_{t-1} \text{ and } a \neq a_{t-1}; \\ \lambda D_{t-1}(y,a), & \text{o.w.} \end{cases} \quad (23)$$

The intuitive idea under the traces above is that for each state only the trace for the most recent action is left and no credits are given for other actions.

(23) updates only the estimated value for one action for each state. Taking advantage of this we propose the algorithm to compute everything including action selection in time linear in the number of states and actions.

[1] Greedy selection in logarithmic time

Greedy selection selects the action with the largest current Q-value estimation for the current state.

$$a_t = \arg\max_a Q_{t-1}(y_t, a) \quad (24)$$

For the implementation of the equation above in time logarithmic in the number of actions you could form for each state a tree with actions at leaves, and make each node in the tree preserve the maximum of the Q-values for the actions corresponding to the leaves under the node. In other words, the following equation should be made to hold:

$$m(N) = \max_{n \in F(N)} m(n) \quad (25)$$

where $F(N)$ denotes the set of child nodes of N, $m(N)$ denotes what is preserved at the node N.

In this tree only the visited nodes are needed to be updated. Thus, logarithmic-time implementation is possible. In addition the space complexity is linear.

[2] Stochastic selections in logarithmic-time

Most of common stochastic selectors such as Boltzmann selector and random selector fit the form of

$$\Pr(a_t = a) = \frac{g(Q_{t-1}(y_t, a))}{\sum_{b \in A} g(Q_{t-1}(y_t, b))} \quad (26)$$

where g is a function that is dependent on the action selector.

The following is a logarithmic-time implementation of the action selection:

- Form for each state a tree with actions at leaves,
- At each node N preserve the sum $u(N)$ of Q values corresponding to the leaves under N, that is, for all the nodes N that are not leaves let
  $$u(N) = \sum_{n \in F(N)} u(n) \quad (27)$$
  and for all the leaves L let
  $$u(L) = g(Q(L)) \quad (28)$$
  where $Q(L)$ denotes the Q-value of the action corresponding to the leaf L.
- From the root node select a child node in turn stochastically following the equation below, and the action is selected as the resulting leaf:
  $$\Pr\{S(n)|S(N), n \in F(N)\} = \frac{u(n)}{u(N)} \quad (29)$$
  where $S(N)$ means N is visited.

Also in this case logarithmic-time implementation is possible because only visited nodes are updated. The space complexity is linear.

5. Discussion and Conclusion

5 · 1 Discussion

The algorithms described in **Chapter 3** and **Chapter 4** implement TD(λ) updating precisely and efficiently for any λ and are indispensable in applying TD(λ) to large problems with partial observability and long delay in rewards.

Logarithmic-time computation in the number of states enables, for example, even TD(λ) updating with millions of states to be implemented in tens of reduction steps. Considering the memory resource of recent computers, it directly follows that

there is no problem about time complexity of tabular TD(λ) updating any longer.

Applications of our algorithm to real time control would be able to compute everything required in time even when given the observation of raw sensor data.

5 · 2 Conclusion

A logarithmic-time implementation algorithm for TD(λ) updating is presented. This algorithm is applied to Sarsa(λ), which is a TD(λ) based reinforcement learning algorithm. Although straightforward application of our method to Sarsa(λ) requires linear time in the number of actions, Sarsa(λ) with replacing traces described in [Singh 96] can be implemented in logarithmic time.

Apart from conventional methods, our method can implement TD(λ) precisely and fast even with $\lambda = 1$ or $\lambda \approx 1$. Thus, our algorithm can apply TD(λ) to problems with long delay of reward and many states.

In tabular TD(λ) dealt in this paper the convergence rate becomes slower as the number of states increases. For this reason we have to apply our algorithm to TD(λ) with generalization using adequate bases. This remains future work.

References

[Barto 83] Barto, A. G., Sutton, R. S. and Anderson, C. W.: Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems, *IEEE Transactions on Systems, Man, and Cybernetics*, 13, pp. 835-846 (1983).

[Cichosz 95] Cichosz, P. and Mulawka, J. J.: Fast and Efficient Reinforcement Learning with Truncated Temporal Differences, *Proceedings of the 14th International Conference on Machine Learning*, pp. 99-107 (1995).

[Dorigo 94] Dorigo, M. and Bersini, M.: A Comparison of Q-learning and Classifier Systems, *From Animals to Animats 3: Proceedings of the 3rd International Conference on Simulation of Adaptive Behavior*, pp. 248-255 (1994).

[Peng 94] Peng, J. and Williams, R. J.: Incremental Multi-Step Q-Learning, *Proceedings of the 11th International Conference on Machine Learning*, pp. 226-232 (1994).

[Singh 96] Singh, S. P. and Sutton, R.S.: Reinforcement Learning with Replacing Eligibility Traces, *Machine Learning 22*, pp. 123-158 (1996).

[Sutton 88] Sutton, R. S.: Learning to Predict by the Methods of Temporal Differences, *Machine Learning 3*, pp. 9-44 (1988).

[Sutton 96] Sutton, R. S.: Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding, *Advances in Neural Information Processing Systems 8*, pp. 1038-1044 (1996).

[Sutton 98] Sutton, R. S. and Barto, A.: Reinforcement Learning: An Introduction, *A Bradford Book*, The MIT Press (1998).

[Watkins 92] Watkins, C. J. C. H. and Dayan, P.: Technical Note: Q-learning, *Machine Learning 8*, pp. 279-292 (1992).

Appendix

A. A proof for the correctness of the algorithm in Fig. 5

A · 1 Theorem

Let $d_t(N)$, $s_t(N)$, $z_t(L)$, $p_t^R(L)$, $p_t^A(L)$, $V_t^R(L)$, and $V_t^A(L)$ be $d(N)$, $s(N)$, $z(L)$, $p^R(L)$, $p^A(L)$, $V^R(L)$, and $V^A(L)$ in Fig. 5 at time t. Then, those values satisfy the following equations.

(the initial values)

$$d_0(N) = s_0(N) = 0 \tag{A.1}$$

$$p_0^R(L) = v_0(L) \tag{A.2}$$

$$z_0(L) = 1 \tag{A.3}$$

$$p_0^A(L) = v_0(L) \tag{A.4}$$

(the updating rules)

$$d_t(N) = \begin{cases} 1, \text{ if } S_t(N); \\ \tilde{d}_t(h(N))d_{t-1}(N), \\ \quad \text{if } S_t(h(N)) \text{ and not } S_t(N); \\ d_{t-1}(N), \quad \text{unless } S_t(h(N)) \end{cases} \tag{A.5}$$

$$s_t(N) = \begin{cases} 0, \text{ if } S_t(N); \\ \tilde{s}_t(h(N))d_{t-1}(N) + s_{t-1}(N), \\ \quad \text{if } S_t(h(N)) \text{ and not } S_t(N); \\ s_{t-1}(N), \quad \text{unless } S_t(h(N)) \end{cases} \tag{A.6}$$

$$\tilde{d}_t(N) = \begin{cases} \lambda, \text{ if } N = \top; \\ \tilde{d}_t(h(N))d_{t-1}(N), \text{ o.w.} \end{cases} \tag{A.7}$$

$$\tilde{s}_t(N) = \begin{cases} \alpha\delta_t, \text{ if } N = \top; \\ \tilde{s}_t(h(N))d_{t-1}(N) + s_{t-1}(N), \\ \quad \text{o.w.} \end{cases} \tag{A.8}$$

$$p_t^R(L) = \begin{cases} p_{t-1}^R(L) + \bar{s}_t(L), & \text{if } S_t(L); \\ p_{t-1}^R(L), & \text{o.w.} \end{cases} \tag{A.9}$$

$$V_t^R(L) = p_{t-1}^R(L) + \bar{s}_t(L) \tag{A.10}$$

$$p_t^A(L) = \begin{cases} p_{t-1}^A(L) + z_{t-1}(L)\tilde{s}_t(L), \\ \quad \text{if } S_t(L); \\ p_{t-1}^A(L), \text{ o.w.} \end{cases} \tag{A.11}$$

$$z_t(L) = \begin{cases} \tilde{d}_t(L)z_{t-1}(L) + 1, & \text{if } S_t(L); \\ z_{t-1}(L), & \text{o.w.} \end{cases} \tag{A.12}$$

$$V_t^A(L) = p_{t-1}^A(L) + z_{t-1}(L)\tilde{s}_t(L) \tag{A.13}$$

The equations above satisfy the following theorem.

[Theorem 1] Under (A.1)...(A.13) and D^R and D^A such that

$$D_1^R(L) = 0 \tag{A.14}$$

$$D_t^R(L) = \begin{cases} 1, & \text{if } S_{t-1}(L) \\ \lambda D_{t-1}^R(L), & \text{o.w.} \end{cases} \tag{A.15}$$

$$D_1^A(L) = 0 \tag{A.16}$$

$$D_t^A(L) = \begin{cases} \lambda D_{t-1}^A(L)+1, & \text{if } S_{t-1}(L) \\ \lambda D_{t-1}^A(L), & \text{o.w.} \end{cases} \tag{A.17}$$

for $t \geq 1$

$$V_t^R(L) = V_{t-1}^R(L) + \alpha D_t^R(L)\delta_t \tag{A.18}$$

$$V_t^A(L) = V_{t-1}^A(L) + \alpha D_t^A(L)\delta_t \tag{A.19}$$

The following describes the outline of its proof.

(1) Lemma 1 for d and $\tilde{d}$,
(2) Lemma 2 for s and $\tilde{s}$,
(3) Lemma 3 for D^R,
(4) Lemma 4 for D^A, and
(5) Theorem 1 for V

B. Lemmas and their proofs

[Lemma 1] For each time step $t \geq 0$,

(1) for each node N, if $t \geq 1$ then

$$\tilde{d}_t(N) = \begin{cases} 0, & \text{if } \tau_t(N) = -\infty; \\ \lambda^{t-\tau_t(N)}, & \text{o.w.} \end{cases} \tag{B.20}$$

and

(2) for each node $N \neq \top$,

$$d_t(N) = \begin{cases} 0, & \text{if } \tau_{t+1}(N) = -\infty; \\ \lambda^{\tau_{t+1}(h(N))-\tau_{t+1}(N)}, & \text{o.w.} \end{cases} \tag{B.21}$$

≪Proof≫ (1) Case $t = 0$:

From the definition of τ, for all N $\tau_1(N) = -\infty$. From (A.1) $d_t(N) = 0$.

(2) Let us assume that the theorem is true for $t = k-1$ where $k > 0$. Then for all node N,

$$d_{k-1}(N) = \begin{cases} 0, & \text{if } \tau_k(N) = -\infty; \\ \lambda^{\tau_k(h(N))-\tau_k(N)}, & \text{o.w.} \end{cases} \tag{B.22}$$

Under the above assumption, we prove

(a) (B.20) is true for $t = k$, and then
(b) (B.21) is true for $t = k$.

(a) i. Case $N = \top$:

From the definition,

$$\tilde{d}_k(\top) = \lambda. \tag{B.23}$$

On the other hand,

$$\lambda^{k-\tau_k(\top)} = \lambda^{k-(k-1)} = \lambda \tag{B.24}$$

because $\top$ is always a non-visited node. Therefore, (B.20) is true for $t = k$.

ii. Let us assume (B.20) is true for $t = k$ and $N = h(n)$. Then,

$$\tilde{d}_k(h(n)) = \lambda^{k-\tau_k(h(n))}. \tag{B.25}$$

From this assumption and (A.7),

$$\tilde{d}_k(n) = \tilde{d}_k(h(n))d_{k-1}(n) \tag{B.26}$$

$$= \lambda^{k-\tau_k(h(n))}d_{k-1}(n). \tag{B.27}$$

By applying (B.22) we obtain

$$\tilde{d}_k(n) = \begin{cases} 0, & \text{if } \tau_k(n) = -\infty; \\ \lambda^{k-\tau_k(n)}, & \text{o.w.} \end{cases}$$

Therefore, (B.20) is true for $t = k$ and $N = n$.

From i and ii, we obtain for all node N

$$\tilde{d}_k(N) = \begin{cases} 0, & \text{if } \tau_k(N) = -\infty; \\ \lambda^{k-\tau_k(N)}, & \text{o.w.} \end{cases} \tag{B.28}$$

(b) Each node N is either

i. a visited node at time k, namely, N s.t. $S_k(N)$;
ii. a non-visited node with its visited parent node at time k, namely, N s.t. $S_k(h(N)) \wedge \neg S_k(N)$; or
iii. a non-visited node with its non-visited parent node at time k, namely, N s.t. $\neg S_k(h(N))$.

i. For N s.t. $S_k(N)$:

$$d_k(N) = 1. \tag{B.29}$$

In addition,

$$\lambda^{\tau_{k+1}(h(N))-\tau_{k+1}(N)} = \lambda^{k-k} = 1.$$

Therefore,

$$d_k(N) = \lambda^{\tau_{k+1}(h(N))-\tau_{k+1}(N)} \tag{B.30}$$

Thus, (B.21) is true for $t = k$ and N s.t. $S_k(N)$.

ii. For N s.t. $S_k(h(N))$ and not $S_k(h(N))$:

from (A.5)

$$d_k(N) = \tilde{d}_k(h(N))d_{k-1}(N) \tag{B.31}$$

A. Case $\tau_k(N) = -\infty$:

From (B.22) and (B.31),

$$d_k(N) = 0. \tag{B.32}$$

B. Case $\tau_k(N) \neq -\infty$:

$\tau_k(h(N)) \neq -\infty$. Thus, from (B.28), (B.22), and (B.31)

$$d_k(N) = \lambda^{k-\tau_k(h(N))}\lambda^{\tau_k(h(N))-\tau_k(N)} = \lambda^{k-\tau_k(N)} \tag{B.33}$$

On the other hand, since $S_k(h(N))$,

$$\tau_{k+1}(h(N)) = k \tag{B.34}$$

In addition, since $\neg S_k(N)$,

$$\tau_{k+1}(N) = \tau_k(N) \tag{B.35}$$

By applying (B.34) and (B.35) to (B.33), we obtain

$$d_k(N) = \lambda^{\tau_{k+1}(h(N))-\tau_{k+1}(N)} \tag{B.36}$$

For A and B, (B.21) is true for $t = k$ and the node N.

iii. For N s.t. $\neg S_k(h(N))$:

from the definition,

$$d_k(N) = d_{k-1}(N) \tag{B.37}$$

$$\tau_{k+1}(h(N)) = \tau_k(h(N)) \tag{B.38}$$

$$\tau_{k+1}(N) = \tau_k(N) \tag{B.39}$$

By applying above equations to (B.22), we obtain

$$d_k(N) = \begin{cases} 0, & \text{if } \tau_{k+1}(N) = -\infty; \\ \lambda^{\tau_{k+1}(h(N))-\tau_{k+1}(N)}, & \text{o.w.} \end{cases}$$

From i, ii, and iii, (B.21) is true for $t = k$ and any node N.

From (a) and (b), Lemma 1 is true for $t = k$.

From (1) and (2), Lemma 1 is true for any time t. □

[Lemma 2] For each time step $t \geq 0$,

(1) for each node N, if $t \geq 1$ then

$$\tilde{s}_t(N) = \begin{cases} 0, & \text{if } \tau_t(N) = -\infty; \\ \sum_{i=\tau_t(N)+1}^{t} \lambda^{i-1-\tau_t(N)}\alpha\delta_i, & \text{o.w.} \end{cases} \tag{B.40}$$

and

(2) for each node $N \neq \top$,

$$s_t(N) = \begin{cases} 0, & \text{if } \tau_{t+1}(N) = -\infty; \\ \sum_{i=\tau_{t+1}(N)+1}^{\tau_{t+1}(h(N))} \lambda^{i-1-\tau_{t+1}(N)}\alpha\delta_i, & \text{o.w.} \end{cases} \tag{B.41}$$

≪Proof≫ (1) Case $t = 0$:

From (A.1) $s_t(N) = 0$. In addition from the definition of τ,

$$\tau_{t+1}(N) = -\infty \quad \text{(B.42)}$$

Thus, (B.41) is true for any N.

(2) Let us assume that (B.41) is true for $t = k-1$ where $k > 0$. Then for all node N,

$$s_{k-1}(N) = \begin{cases} 0, & \text{if } \tau_k(N) = -\infty; \\ \sum_{i=\tau_k(N)+1}^{\tau_k(h(N))} \lambda^{i-1-\tau_k(N)} \alpha\delta_i, & \text{o.w.} \end{cases} \quad \text{(B.43)}$$

Under the above assumption, we prove

(a) (B.40) is true for $t = k$, and then

(b) (B.41) is true for $t = k$.

(a) i. Case $N = \top$:

From the definition,

$$\tilde{s}_k(\top) = \alpha\delta_k \quad \text{(B.44)}$$

On the other hand,

$$\sum_{i=\tau_k(N)+1}^{k} \lambda^{i-1-\tau_k(N)} \alpha\delta_i = \alpha\delta_k \quad \text{(B.45)}$$

because $\top$ is always a non-visited node. Therefore, (B.40) is true for $t = k$.

ii. Let us assume (B.40) is true for $t = k$ and $N = h(n)$.

A. Case $\tau_k(n) = -\infty$:

from Lemma 1, $d_{k-1}(n) = 0$. In addition, $s_{k-1}(n) = 0$ from (B.43). Therefore, from (A.8),

$$\tilde{s}_k(n) = 0 \quad \text{(B.46)}$$

B. Case $\tau_k(n) \neq -\infty$:

from Lemma 1 we obtain

$$d_{k-1}(n) = \lambda^{\tau_k(h(n))-\tau_k(n)} \quad \text{(B.47)}$$

In addition, from (B.43),

$$s_{k-1}(n) = \sum_{i=1+\tau_k(n)}^{\tau_k(h(n))} \lambda^{i-1-\tau_k(n)} \alpha\delta_i \quad \text{(B.48)}$$

By applying (B.47) and (B.48) to (A.8), and from the assumption that (B.40) is true for $t = k$ and $N = h(n)$; we obtain

$$\tilde{s}_k(n) = \sum_{i=1+\tau_k(n)}^{k} \lambda^{i-1-\tau_k(n)} \alpha\delta_i$$

From A and B, (B.40) is true for $t = k$ and $N = n$.

From i and ii (B.40) is true for $t = k$ for any node N.

(b) i. For N s.t. $S_k(N)$:

$$s_k(N) = 0. \quad \text{(B.49)}$$

In addition, since $\tau_{k+1}(h(N)) = \tau_{k+1}(N) = k$,

$$\sum_{i=1+\tau_{k+1}(N)}^{\tau_{k+1}(h(N))} \lambda^{i-1-\tau_{k+1}(N)} \alpha\delta_i = 0$$

Therefore, (B.41) is true for $t = k$ and N s.t. $S_k(N)$.

ii. For N s.t. $S_k(h(N))$ and not $S_k(h(N))$: from (A.6)

$$s_k(N) = \tilde{s}_k(h(N)) d_{k-1}(N) + s_{k-1}(N) \quad \text{(B.50)}$$

By applying the result of (a) to $\tilde{s}_k(h(N))$, we obtain

$$s_k(N) = \sum_{i=1+\tau_k(N)}^{k} \lambda^{i-1-\tau_k(N)} \alpha\delta_i$$

where $s_k(N) = 0$ for N s.t. $\tau_k(N) = -\infty$, in the same way as the Case ii in (a).

Since $S_k(h(N))$, $\tau_{k+1}(h(N)) = k$. In addition, since $\neg S_k(N)$, $\tau_{k+1}(N) = \tau_k(N)$. Therefore

$$s_k(N) = \sum_{i=1+\tau_{k+1}(N)}^{\tau_{k+1}(h(N))} \lambda^{i-1-\tau_{k+1}(N)} \alpha\delta_i$$

where $s_k(N) = 0$ for N s.t. $\tau_{k+1}(N) = -\infty$.

iii. For N s.t. $\neg S_k(h(N))$:

from the definition,

$$s_k(N) = s_{k-1}(N) \quad \text{(B.51)}$$

$$\tau_{k+1}(h(N)) = \tau_k(h(N)) \quad \text{(B.52)}$$

$$\tau_{k+1}(N) = \tau_k(N) \quad \text{(B.53)}$$

From (B.43), (B.41) is true for $t = k$ and N s.t. $\neg S_k(h(N))$.

From i, ii, and iii, (B.41) is true for $t = k$ and any node N.

From (a) and (b), Lemma 2 is true for $t = k$.

From (1) and (2), Lemma 2 is true for any time t. □

[Lemma 3] For any time $t > 0$ and for any leaf node L

$$D_t^R(L) = \begin{cases} 0, & \text{if } \tau_t(L) = -\infty; \\ \lambda^{t-1-\tau_t(L)}, & \text{o.w.} \end{cases} \quad \text{(B.54)}$$

≪**Proof**≫ (1) Case $t = 1$:

from the definition of τ, $\tau_1(L) = -\infty$.

From the definition of D^R, $D_1^R(L) = 0$.

Therefore (B.54) is true.

(2) Let us assume the lemma is true for $t = k-1$ where $k > 1$, namely,

$$D_{k-1}^R(L) = \begin{cases} 0, & \text{if } \tau_{k-1}(L) = -\infty; \\ \lambda^{k-2-\tau_{k-1}(L)}, & \text{o.w.} \end{cases} \quad \text{(B.55)}$$

(a) Case $S_{k-1}(L)$ is true:

from the definition of D^R

$$D_k^R(L) = 1 \quad \text{(B.56)}$$

In addition, since $\tau_k(L) = k-1$ from the definition of τ,

$$\lambda^{k-1-\tau_k(L)} = 1 \quad \text{(B.57)}$$

(b) Case $S_{k-1}(L)$ is false:

from the definition of D^R,

$$D_k^R(L) = \lambda D_{k-1}^R(L) \quad \text{(B.58)}$$

By applying (B.55)

$$D_k^R(L) = \begin{cases} 0, & \text{if } \tau_{k-1}(L) = -\infty; \\ \lambda^{k-1-\tau_{k-1}(L)}, & \text{o.w.} \end{cases} \quad \text{(B.59)}$$

In addition, from the definition of τ, unless $S_{k-1}(L)$, $\tau_k(L) = \tau_{k-1}(L)$, and thus

$$D_k^R(L) = \begin{cases} 0, & \text{if } \tau_k(L) = -\infty; \\ \lambda^{k-1-\tau_k(L)}, & \text{o.w.} \end{cases} \quad \text{(B.60)}$$

From (a) and (b), (B.54) is true for $t = k$.

From (1) and (2), (B.54) is true for any time t in general. □

[Lemma 4] For any time $t > 0$ and any leaf node L

$$D_t^A(L) = \begin{cases} 0, & \text{if } \tau_t(L) = -\infty; \\ z_{t-1}(L)\lambda^{t-1-\tau_t(L)}, & \text{o.w.} \end{cases} \quad \text{(B.61)}$$

≪**Proof**≫ (1) Case $t = 1$:

from the definition of τ, $\tau_1(L) = -\infty$.
From the definition of D^A, $D_1^A(L) = 0$.
Thus (B.61) is true.

(2) Let us assume the lemma is true for $t = k-1$ where $k > 1$, namely,

$$D_{k-1}^A(L) = \begin{cases} 0, & \text{if } \tau_{k-1}(L) = -\infty; \\ z_{k-2}(L)\lambda^{k-2-\tau_{k-1}(L)}, & \text{o.w.} \end{cases} \quad \text{(B.62)}$$

(a) Case $S_{k-1}(L)$ is true:
from the definition of τ, $\tau_k(L) = k-1$.
From the definition of D^A,

$$D_k^A(L) = \lambda D_{k-1}^A(L) + 1 = \begin{cases} 1, & \text{if } \tau_{k-1}(L) = -\infty; \\ z_{k-2}(L)\lambda^{k-1-\tau_{k-1}(L)} + 1, & \text{o.w.} \end{cases}$$

From (A.12)

$$z_{k-1}(L) = \tilde{d}_{k-1}(L) z_{k-2}(L) + 1 \quad \text{(B.63)}$$

From Lemma 1

$$z_{k-1}(L) = \begin{cases} 1, & \text{if } \tau_{k-1}(L) = -\infty; \\ z_{k-2}(L)\lambda^{k-1-\tau_{k-1}(L)} + 1, & \text{o.w.} \end{cases}$$

Thus

$$D_k^A(L) = z_{k-1}(L) \quad \text{(B.64)}$$

$$= z_{k-1}(L)\lambda^{k-1-\tau_k(L)} \quad \text{(B.65)}$$

(b) Case $S_{k-1}(L)$ is false:
from the definition of D^A

$$D_k^A(L) = \lambda D_{k-1}^A(L) \quad \text{(B.66)}$$

By applying (B.62)

$$D_k^A(L) = \begin{cases} 0, & \text{if } \tau_{k-1}(L) = -\infty; \\ z_{k-2}(L)\lambda^{k-1-\tau_{k-1}(L)}, & \text{o.w.} \end{cases}$$

From the definition of τ, $\tau_k(L) = \tau_{k-1}(L)$; from (A.12), $z_{k-1}(L) = z_{k-2}(L)$; and thus

$$D_k^A(L) = \begin{cases} 0, & \text{if } \tau_k(L) = -\infty; \\ z_{k-1}(L)\lambda^{k-1-\tau_k(L)}, & \text{o.w.} \end{cases}$$

From (a) and (b), (B.61) is true for $t = k$.

From (1) and (2), (B.61) is true for any time t in general. □

B · 1 The proof for the theorem

≪**Proof**≫ (1) Proof for (A.18):
from the definition of τ for i s.t. $\tau_k(L) < i \leq k$ $\tau_i(L) = \tau_k(L)$. Thus from Lemma 2 and Lemma 3,

$$\tilde{s}_t(L) = \begin{cases} 0, & \text{if } \tau_t(L) = -\infty; \\ \sum_{i=\tau_t(L)+1}^{t} \lambda^{i-1-\tau_i(L)}\alpha\delta_i, & \text{o.w.} \end{cases} = \begin{cases} 0, & \text{if } \tau_t(L) = -\infty; \\ \sum_{i=\tau_t(L)+1}^{t} D_i^R(L)\alpha\delta_i, & \text{o.w.} \end{cases} \quad \text{(B.67)}$$

By the way, from (A.9)

$$p_{t-1}^R(L) = p_{\tau_t(L)}^R(L) \quad \text{(B.68)}$$

On the other hand, if $S_t(L)$ then $p_t^R(L) = V_t^R(L)$ because of (A.9) and (A.10), and thus

$$p_{\tau_t(L)}^R(L) = V_{\tau_t(L)}^R(L) \quad \text{(B.69)}$$

Therefore, from (A.10) and (B.67)

$$V_t^R(L) = \begin{cases} v_0(L), & \text{if } \tau_t(L) = -\infty; \\ V_{\tau_t(L)}^R(L) + \sum_{i=\tau_t(L)+1}^{t} D_i^R(L)\alpha\delta_i, & \text{o.w.} \end{cases}$$

Hence

$$V_t^R(L) - V_{t-1}^R(L) = D_t^R(L)\alpha\delta_t \quad \text{(B.70)}$$

(2) Proof for (A.19):
since $z_i(L) = z_{\tau_k(L)}(L)$ for i s.t. $\tau_k(L) < i \leq k$, from Lemma 2 and Lemma 4,

$$z_{t-1}(L)\tilde{s}_t(L) = \begin{cases} 0, & \text{if } \tau_t(L) = -\infty; \\ \sum_{i=\tau_t(L)+1}^{t} z_{\tau_i(L)}\lambda^{i-1-\tau_i(L)}\alpha\delta_i, & \text{o.w.} \end{cases} = \begin{cases} 0, & \text{if } \tau_t(L) = -\infty; \\ \sum_{i=\tau_t(L)+1}^{t} D_i^A(L)\alpha\delta_i, & \text{o.w.} \end{cases} \quad \text{(B.71)}$$

By doing the same thing as the proof for (A.18); replacing (B.67), (A.9), and (A.10) with (B.71), (A.11), and (A.13) respectively; we obtain

$$V_t^A(L) - V_{t-1}^A(L) = D_t^A(L)\alpha\delta_t \quad \text{(B.72)}$$

From (1) and (2), the theorem is proved. □