Compact State Representation for Tree-Structured RL

Yang Gao Y.GAO11@IMPERIAL.AC.UK
Imperial College London, Queen's Gate, SW7 2AZ, London

Francesca Toni F.TONI@IMPERIAL.AC.UK
Imperial College London, Queen's Gate, SW7 2AZ, London

Abstract

This paper describes our Reinforcement Learning (RL) based strategy for the Invasive Plant domain in the RL Competition 2013. We propose a novel state representation which improves the learning speed and robustness of the RL-based strategy. Also, to improve convergence speed and prevent poor performance (exceeding the budget), we propose heuristics and use reward shaping techniques to integrate these heuristics.

1. Introduction

Applying Reinforcement Learning (RL) to the Invasive Plant problem faces several problems, including how to represent the tree-structured environment and how to prevent punishment caused by exceeding the budget. Because the size and structure of the tree can change, the default representation of the environment, i.e. a list of integers representing the local state of all habitats, is vulnerable to even slight changes of the environment, which results in difficulties in reusing existing learning experiences in new environments. So in order to successfully apply RL to the Invasive Plant problem, we propose a more robust and effective way to represent the environment.

Another important task in this problem is to avoid exceeding the budget limit in each step of decision making, as this will result in huge punishment. We employ a supervised linear function learning algorithm to learn the relationship between actions and their costs, and propose heuristics to avoid actions that will exceed the budget. We integrate these heuristics into our RL-based strategy by means of look-ahead reward shaping (Wiewiora et al., 2003).

Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28. Copyright 2013 by the author(s).

The remainder of this paper is organised as follows: Section 2 gives an overview of our strategy, Section 3 describes our state representation, Section 4 describes how we predict the costs of actions, Section 5 discusses how we choose the most suitable RL algorithm for this problem, and Section 6 presents the shaping rewards we used in this problem. In Section 7 we will give the parameters we used and briefly discuss the performance of our strategy. Throughout this paper, we let E and H represent the number of reaches and habitats in each reach, respectively.

2. Our Strategy

Our strategy is based on a variant of the SARSA algorithm (Sutton & Barto, 1998) incorporating look-ahead reward shaping (Wiewiora et al., 2003). The target of our strategy is to choose the best *Action Vector*. An Action Vector av consists of E actions, where the ith element of av is the action for the ith reach. Algorithm 1 shows the main structure of our strategy. In Algorithm 1, two functions are invoked: function **getQValue**, as described in Algorithm 2, is used to evaluate the value of each Action Vector, and function **update**ρ, as presented in Algorithm 3, updates the learning experiences. The reward shaping is performed within function **update**ρ and the shaping rewards are discussed in Section 6. The definition of ρ, Integer List and percentage representation will be given in Section 3, and the definition of Weight Vector W and Input Vector I will be given in Section 4.

3. State Representation

The default state representation is an *Integer List* of length $E \times H$. Using the Integer List as the state representation of a RL-based strategy would cause the following problems:

1. The structure of the tree is not represented by the Integer List.

Algorithm 1 SARSA algorithm used in our strategy

```
1 Initialise $\rho(r, a) = 0$ for all reaches $r$ and actions $a$
2 Read reach number $E$ and habitat number $H$
3 Repeat (for each episode):
4   Get the *Integer List* $L_t$ of the current state
5   Get *Action Vector* set $A_t$ for the current state
6   For each $av \in A_t$, invoke **getQValue**($L_t$, $av$)
     and then choose $av_t \in A_t$ using $\varepsilon$-greedy policy
7   Repeat (until episode terminates)
8     Perform actions in $av_t$, observe reward $r_t$ and
       get *Integer List* $L_{t+1}$ for the new state
9     Learn the *Weight Vector* $W$
10    Get *Action Vector* set $A_{t+1}$ for $L_{t+1}$
11    For each $av \in A_{t+1}$, invoke
       **getQValue**($L_{t+1}$, $av$) and then choose
       $av_{t+1} \in A_{t+1}$ using $\varepsilon$-greedy policy
12    invoke **update**$\rho$($L_t$, $L_{t+1}$, $av_t$, $av_{t+1}$, $W$)
13    $av_t := av_{t+1}$, $L_t := L_{t+1}$
```

Algorithm 2 Function **getQValue**(*Integer List L*, *Action Vector av*)

```
1 Initialise $i := 1$, $sum := 0$
2 While $i \leq E$:
3   Extract the percentage representation $r^i$ of $i$th
     reach from *Integer List L*
4   Get the $i$th action $a^i$ in $av$
5   $sum := sum + \rho(r^i, a^i)$
6   $i := i + 1$
7 Return $sum$
```

Algorithm 3 Function **update**ρ(*Integer List* L_t, *Integer List* L_{t+1}, *Action Vector* av_t, *Action Vector* av_{t+1}, *Weight Vector* W)

```
1 Initialise $i := 1$
2 While $i \leq E$:
3   Extract the percentage representation $r^i_t$ of $i$th
     reach from *Integer List* $L_t$
4   Extract the percentage representation $r^i_{t+1}$ of $i$th
     reach from *Integer List* $L_{t+1}$
5   Get the $i$th action $a^i_t$ in $av_t$
6   Get the $i$th action $a^i_{t+1}$ in $av_{t+1}$
7   Extract *Input Vector* $I^i$ for the $i$th reach from
     $L_t$ and compute the reward $p^i = I^i \cdot W$ for the
     $i$th reach
8   $\rho(s^i_t, a^i_t) := \rho(s^i_t, a^i_t) + \alpha[p^i + \gamma\phi(s^i_{t+1}, a^i_{t+1}) -$
     $\phi(s^i_t, a^i_t) + \gamma\rho(s^i_{t+1}, a^i_{t+1}) - \rho(s^i_t, a^i_t)]$
9   $i := i + 1$
```

2. The order of habitats in each reach is encoded in the Integer List. However, since all actions are performed over reaches and the upstream/downstream spread is also over reaches, the effect of an action performed on a reach will not be influenced by the order of habitats in this reach.

3. The Integer List state representation is prone to any changes of the tree, which results in difficulties in reusing learning experiences.

Because actions are performed over reaches, selecting the best action for each independent reach is more direct and much easier than selecting the best Action Vector. Hence, we assume that in the best Action Vector, each element is the best action for the corresponding reach[1]. Then we just need to select the best action for each single reach. The cost of a reach is determined by the number of each type of plant in this reach. So an intuitive idea is to use the number of each type of plant in a reach to represent this reach. However, when the number of habitats in a reach (i.e. H) changes, the learning experiences obtained in the old environment cannot be directly used in the new environment. So a more generic state representation could keep record of the percentage of each kind of plant in a reach. By doing this, no matter how E and H change, the state representation will keep the same length and the unified meaning in all different settings.

When budget cannot afford all invaded reaches to be eradicated and restored, we have to choose the more "important" reaches to perform these actions. Because plants may generate and spread "propagules" to upstream and downstream reaches, and downstream spread is much more likely than upstream spread, reaches with more downstream reaches are more "important" in the sense that they have more significant impact on other reaches. In the remainder of this paper, we use U^n (D^n) to denote the set of upstream (downstream) reaches of reach r^n, defined as:

- $r^i \in U^n$ ($r^i \in D^n$) if r^i is directly connected to r^n and farther (closer) to the root than r^n
- $r^j \in U^n$ ($r^j \in D^n$) if r^j is directly connected to $r^i \in U^n$ ($r^i \in D^n$) and farther (closer) to the root than r^i

where $i, j, n \in \{1, 2, \cdots, E\}$ and r^i stands for the ith reach in the environment. To summarise, our state representation for a reach r^n uses a vector consisting of 4 elements:

1. the percentage of invasive plants in r^n, i.e. $Num^n(invasive)/H$

[1]When the budget cannot afford all reaches, this assumption fails. In Section 6 we use reward shaping to tackle this problem.

2. the percentage of native plants in r^n, i.e. Num^n*(native)*$/H$
3. the percentage of upstream reaches, i.e. $|U^n|/E$
4. the percentage of downstream reaches, i.e. $|D^n|/E$

where $Num^n(K)$ stands for the number of plant K in reach r^n. Because all these elements are percentages and are in $[0, 1]$, we use the term *percentage representation* to refer to our state representation. The structure of the tree is compactly represented in the percentage representation because the percentages of upstream and downstream reaches are included. In our strategy, we use data structure $\rho(r^i, a^i)$ to record the long-term expected accumulated reward of performing action a^i in reach r^i. The most significant advantages of percentage representation are:

1. Robust to changes of the environment. The percentage representation keeps the same length and unified meaning in environments with any E and H values.

2. Improve the learning speed. Since all reaches share the same ρ, they share their learning experiences and, therefore, the algorithm converges faster.

However, in percentage representation, we have to use function approximation techniques, e.g. tile coding (Sutton & Barto, 1998), to discretise the state space, because all elements are real numbers in $[0, 1]$.

4. Cost Prediction

In this section, we describe how we predict the cost of actions so as to prevent extremely poor performance. The overall cost C can be calculated by obtaining the inner product of *Input Vector* I and *Weight Vector* W, i.e. $C = I \cdot W$. The elements of I as well as the default values of W are described in Table 1. Note that every element in the Input Vector can be extracted from the Integer List, and cost C is the immediate reward received from the environment. In other words, we know I and C and want to learn W so as to use W to evaluate the cost of actions.

We use the Stochastic Gradient Descent (Gardner, 1984) (SGD) algorithm to learn the values of W. We use the default values of W to initialise the weights in the SGD learner. Furthermore, to ensure that the learnt weight values are consistent with their semantics, we propose the following requirements for the learnt weights[2]:

1. $W_i \leq 0$, $i = 1, 2, \cdots, 8$

2. $|W_2| \geq |W_3|$: the cost of each native plant should not be less than the cost of each empty slot

[2]All these inequalities are inferred from the semantics of each parameter. The default parameters satisfy all these inequalities.

3. $|W_8| \geq |W_7|$: the variable restoration cost for each invaded slot should not be less than each empty slot

where W_i means the ith element in the weight vector. If the estimated value $P = W \cdot I$ and the received reward (cost) C are very close ($|C - P| < 0.1$) and all the inequalities above are satisfied, we use the learnt weights; otherwise, we use the default weights. Note that we use an external file to store the values of W. Whenever W is updated, we will update the external file and use the new W as the default.

5. Choosing an RL Algorithm

In this section, we justify our choice of using SARSA as our RL algorithm. We evaluate RL algorithms in terms of these two perspectives:

1. model-based or model-free
2. on-policy or off-policy

When the number of experiments is small, model-based RL can learn the model of the environment from the limited data and, therefore, use data more efficiently. But it needs to keep record of all existing trajectories. In this problem, the number of experiments is large. So we choose a model-free RL algorithm.

In off-policy RL algorithms, the policy being used is not the policy being updated. The advantage of off-policy algorithms is their stability during each episode: the same policy is used throughout an episode. But on-policy RL algorithms use the latest policy immediately and, therefore, typically converge faster than off-policy algorithms. In this problem, the speed of convergence is important. Also, the stability of on-policy algorithms can be improved by using techniques such as reward shaping (Ng et al., 1999).

6. Shaping Rewards

Reward shaping is used in Algorithm 3 to improve the learning speed and avoid "bad" actions. There are two methods to integrate shaping rewards into SARSA: look-ahead and look-back advise. Our empirical results show that with the same shaping rewards, the long-term performance of these two techniques are very similar, but the initial performance of look-ahead advise is much better than look-back because the policy is manipulated by the potential values in look-ahead (Wiewiora et al., 2003). The potential values for each reach-action pair is described in Table 2.

7. Implementation and Results

The parameters we used are as follows: α (line 8 of Algorithm 3) is initialised as 0.1 and linearly decreased to 0.01 within 20,000 iterations. After 20,000 iterations, α is fixed to 0.01. $\varepsilon = 0.1$. Default values of the

Index	Type	Semantics	Default weight
1	Boolean	whether this reach is invaded or not	-10
2	Integer	number of native plants in this reach	-0.1
3	Integer	number of empty slots in this reach	-0.05
4	Boolean	whether the action on this reach is eradicate	-0.5
5	Integer $\times$ Boolean	number of eradicated invasive plants	-0.4
6	Boolean	whether the action on this reach is restore	-0.9
7	Integer $\times$ Boolean	number of restored empty slots	-0.4
8	Integer $\times$ Boolean	number of restored invasive plants	-0.8

Table 1. Index, type and semantics of elements in the Input Vector and the default values of the Weight Vector. The length of both Input Vector and Weight Vector is 8, regardless of the value of E and H. As for an integer variable i and a boolean variable b, $i \times b$ equals i and 0 when b equals true and false, respectively.

Shaping Reward	Condition	Description
-20	The cost of this action vector exceeds 90% of the budget.	Prevent an agent from performing actions whose cost is likely to exceed budget.
$+15 \times \|D\|/E$	State: The reach has invasive plants. Action: **eradicate+restore**.	$\|D\|/E$ indicates the importance of this reach. This rule encourages action **eradicate+restore** in invaded reaches.
$+5 \times \|D\|/E$	State: the reach has invasive plants. Action: **eradicate**.	This rule encourages action **eradicate** in invaded reaches. Because the result of **eradicate** is not so wanted as action **eradicate+restore**, its shaping reward is smaller.
$+10 \times \|D\|/E$	State: the reach has no invasive plants but has more than 30% empty slots. Action: **restore**.	This rule encourages performing **restore** in un-invaded reaches which have many empty slots.
$+10$	State: the reach has no invasive plants and has less than 30% empty slots. Action: **null**.	This rule encourages doing nothing in un-invaded reaches which have few empty slots.

Table 2. The potential function $\phi(r, a)$ for some reach-action pairs. The potential values of reach-action pairs that are not covered by these rules are 0.

Weight Vector W are showed in Table 1. γ (line 8 of Algorithm 3) value is read from the environment. As for tile coding, we use 40 tilings with 10 tiles in each tiling. The length of each tile is 0.1.

Under the default setting, i.e. $H = 4$, $E = 7$ and budget $= 100$, the cost of our strategy is around 50% less than that of the sample SARSA-based strategy. When budget $= 5$, our strategy is able to avoid exceeding the budget in most occasions, whereas the sample strategy cannot avoid the punishment, even after 100,000 episodes of learning.

A phenomenon worth noticing is that when budget $=$ 5, our strategy will not exceed the budget in the initial stage. However, as the learning proceeds, the strategy may exceed the budget occasionally and perform worse than the initial performance. This phenomenon is consistent with the results reported by (Wiewiora et al., 2003): when the reward is negative and look-ahead reward shaping is used, the performance will fluctuate because the recommended actions have lower Q-values after few hundred steps of learning than other unexplored actions. However, the average performance of look-ahead reward shaping is still better than look-back, because, when using look-back, the policy is not directly manipulated by the shaping rewards. So the shaping rewards cannot avoid punishments until the "bad" actions have performed.

References

Gardner, W.A. Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique. *Signal Processing*, 6:113–133, 1984.

Ng, A., Harada, D., and Russell, S. Policy invariance under reward transformations: theory and application to reward shaping. In *Proc. ICML*, 1999.

Sutton, R. and Barto, A. *Reinforcement Learning*. MIT Press, 1998.

Wiewiora, E., Cottrell, G., and Elkan, C. Principled methods for advising reinforcement learning agents. In *Proc. ICML*, 2003.