

Reinforcement Learning and Shape Grammars

Technical report

Author	Manuela Ruiz Montiel
Date	April 15, 2011
Version	1.0

Contents

0. Introduction	3
1. Tabular approach.....	4
1.1 Tabular Q-learning	4
1.1.1 Problems considered	4
1.1.2 Algorithm	4
1.1.3 Results and conclusions.....	5
2. Generalization approach.....	5
2.1 Problem considered.....	6
2.2 Linear Q-learning	6
2.2.1 Algorithm	6
2.2.2 Results.....	7
2.3 Linear Sarsa(λ) and Linear Q(λ)	7
2.3.1 Algorithms	7
2.3.2 Results.....	7
3. Conclusions	10
References	10

0. Introduction

Shape grammars [1] are a powerful tool to generate different designs from a fixed set of rules. In addition, we may want the synthesized solutions to satisfy some criteria or constraints. A particular way to achieve it is by using reinforcement learning [2] techniques in order to choose which rule and transformation are to be applied in each step of the grammar application.

According to the common vocabulary in reinforcement learning, we define the following elements:

- Action: A pair: rule, transformation. The transformation determines the sub-shape of the current design that will undergo the rule application.
- Step: Application of an action.
- Episode: Successive application of a number of steps.
- Environment: In our case it is completely deterministic. The environment is just the method that applies a rule to the current design and returns a new one.
- State: In each step of the process, the state is the current design.
- Reward: Number associated with a state. The higher it is, the better the state is in the short term. The way of calculating it depends on the problem that is being considered. In our case, the intermediate states have always a reward of zero, and in the final ones we make the calculation.
- Policy: A function $\pi(\text{state})$ that returns the action to take next.
- Value: Number associated with a pair (state, action), that stands for the expected return obtained when following the given action from the given state. The return is defined as a function of the obtained rewards along the episode. The value determines the policy, since at each step of an episode, the action which yields a better value is chosen.

In the next sections we describe different approaches that have been explored, as well as the conclusions that we have reached.

1. Tabular approach

Our first approach is based on tabular learning, that is, the values of each pair (state, action) are stored into a table and they update them in successive episodes.

1.1 Tabular Q-learning

1.1.1 Problems considered

We have considered six different problems, summarized in table 1.

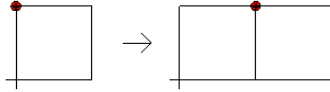
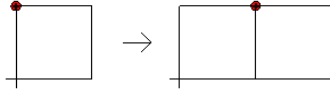
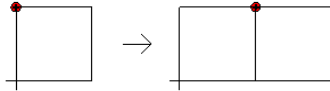
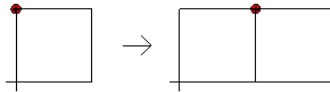
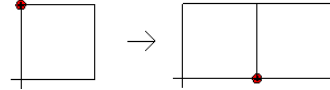
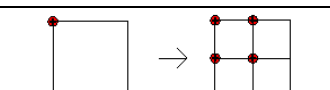
Problem ID	Grammar	Goal
PTab1		Maximize the area inside the convex hull
PTab2		Minimize the area inside the convex hull
PTab3		Produce horizontally symmetric figures
PTab4		Produce vertically symmetric figures
PTab5		Produce vertically and horizontally symmetric figures
PTab6		Produce vertically and horizontally symmetric figures

Table 1: Considered problems in tabular Q-learning

1.1.2 Algorithm

Q-learning is a temporal difference, off-policy algorithm [3]. In figure 1 we can see its pseudo code, where $Q(s,a)$ is the expected value of an (state, action) pair.

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Repeat (for each step of episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $a$ , observe  $r, s'$ 
     $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'$ ;
  until  $s$  is terminal

```

Figure 1: Pseudocode of Q-learning

1.1.3 Results and conclusions

In this section we summarize some tests done using the problems described above.

In all the executions, the setting of the algorithm was:

$$\alpha = 0.1, \gamma = 1, \varepsilon = 0.3$$

Tests run inside SketchUp, with Ruby 1.8.2, in a machine with configuration:

- Intel Core i7
- 8GB RAM
- Windows 7 64 bits

Problem	Steps	Episodes	Success?	Time
PTab1	3	500	Yes	~40s
PTab2	3	500	Yes	~40s
PTab3	3	500	Yes	~40s
PTab4	3	500	Yes	~40s
PTab5	5	500	Yes	~60s
PTab6	5	600	No	~300s

For toy examples like the five first problems everything is ok, but for a grammar a little bit complex like the one in PTab6, the learner was not able to reach the optimal solution. Moreover, the elapsed time grows too much.

In PTab6, there is only one possible solution that satisfies the goal within 5 steps. Using a tabular algorithm, it is unlikely that one particular state (concretely, the solution) is visited, since the set of all possible states that can be generated is huge.

We can extract the conclusion that generalization of states is of vital importance. The application of a shape grammar produces a large number of states, and this number grows exponentially as the episode size (that is, the number of steps) rises. We need mechanisms to represent similar states in the same manner, and thus be able to update the value of many different steps at once. Even if a desirable state is not visited, having visited one that is similar to it can be enough to learn its value in a proper way.

Another issue is that the produced policies are deterministic. In the case that more than one solution exists, the policy will always generate the same shape, ignoring the rest that would also be satisfactory.

2. Generalization approach

We have decided to use linear generalization. This method consists of using a linear function to obtain the values of the pairs (state, action), instead of using a table. The updating is done over the coefficients of the linear function.

Given a (state, action) pair we must be able to obtain its value using the linear function. For doing this, we represent each pair as a set of *features* that aim to characterize the pairs in the sense that similar pairs will have similar features.

The value function will look like this:

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \dots + w_n f_n(s, a)$$

Where w_i are the coefficients of the linear function and f_i are the features.

2.1 Problem considered

We have considered the problem summarized in table 2.

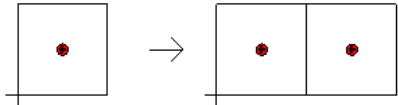
Problem ID	Grammar	Goal
PGen1		Produce compact shapes, using the compactness index as reward: (area/perimeter ²)

Table 2: Considered problem in the generalization approach

The features for characterizing the states in this problem are four numbers between 0 and 4, standing for the number of squares with 1, 2, 3 and 4 neighbours in the produced shape by the pair (state, action):

[#squares with 1 neighbour,
#squares with 2 neighbours,
#squares with 3 neighbours,
#squares with 4 neighbours]

The linear function will thus have four coefficients, one for each feature.

2.2 Linear Q-learning

2.2.1 Algorithm

Linear Q-learning is the same algorithm that the one shown in Figure 1, except for the updating step. In figure 2 we can see how the approximate updating is done.

$transition = (s, a, r, s')$	
$difference = [r + \gamma \max_{a'} Q(s', a')] - Q(s, a)$	
$Q(s, a) \leftarrow Q(s, a) + \alpha [difference]$	Exact Q's
$w_i \leftarrow w_i + \alpha [difference] f_i(s, a)$	Approximate Q's

Figure 2: Updating the coefficients

2.2.2 Results

Unfortunately, in every test we made with this algorithm there was a problem of convergence. In [4], a detailed discussion about using function approximation with Q-learning can be found. The problem is that function approximation produces unstable processes when combined with *bootstrapping*¹ methods, even more with off-policy methods like Q-learning.

2.3 Linear Sarsa(λ) and Linear Q(λ)

These methods allow controlling the level of bootstrapping. Sarsa(λ) is an on-policy method, while Q(λ) is an off-policy one.

2.3.1 Algorithms

Algorithms for binary features and with ϵ -greedy policies can be found in chapter 8 of [2]. We have adapted these algorithms in order to deal with our continuous features.

These methods associate an eligibility trace e to every feature. When a (state, action) pair appears inside an episode in order to be updated, then the eligibility traces of the features that characterize that pair (that is, those that are 1-valued) increase by one, while the ones that do not appear (the 0-valued) decrease (the traces accumulate).

Then, the updating is done:

$$w_i \leftarrow w_i + \alpha [difference] e_i$$

In order to use these algorithms with our continuous features, we add the *value* of the feature to the eligibility trace. It is equivalent to the binary case, because then, the added value was 1 when the feature was 1, and 0 when the feature was 0. We have normalized our features in the interval [0, 1] in order to guarantee the convergence properties of the algorithms.

2.3.2 Results

¹ By *bootstrapping* we mean updating a value from other value estimations.

In this section we summarize some tests done using the problem and the algorithms described above.

In all the executions, the setting of the algorithm was:

$\alpha = 0.1$, $\gamma = 0.8$, $\varepsilon = 0.3$, $\lambda = 0.5$, #steps = 15 (16 squares)

Tests run in a machine with configuration:

- Intel Core i7
- 8GB RAM
- Windows 7 64 bits

Plataform	Episodes	Q(λ)		
SketchUp (Ruby 1.8.6)	3000	c[1] = 0.0027 c[2] = -0.0017 c[3] = 0.0239 c[4] = 0.0631 t: 4389.12s 1	c[1] = 0.0059 c[2] = 0.0023 c[3] = 0.0332 c[4] = 0.0637 t: 4388.74s	c[1] = 0.0057 c[2] = 0.0013 c[3] = 0.0243 c[4] = 0.0655 t: 3655.39s
Netbeans (Ruby 1.8.6)	5000	c[1] = 0.0037 c[2] = 0.0044 c[3] = 0.0338 c[4] = 0.0624 t: 4728.42s 2	c[1] = 0.0063 c[2] = 0.0023 c[3] = 0.0263 c[4] = 0.0658 t: 4612.59s	c[1] = 0.0035 c[2] = 0.001 c[3] = 0.0266 c[4] = 0.0592 t: 4349.51s
SciTe (Ruby 1.9.2)	10000	c[1] = 0.0027 c[2] = 0.0018 c[3] = 0.0305 c[4] = 0.0631 t: 3173.86s	c[1] = 0.0067 c[2] = 0.0042 c[3] = 0.0315 c[4] = 0.0644 t: 3146.98s	c[1] = 0.0052 c[2] = 0.0034 c[3] = 0.027 c[4] = 0.065 t: 3157.07s

Table 3: Results of Q(λ)

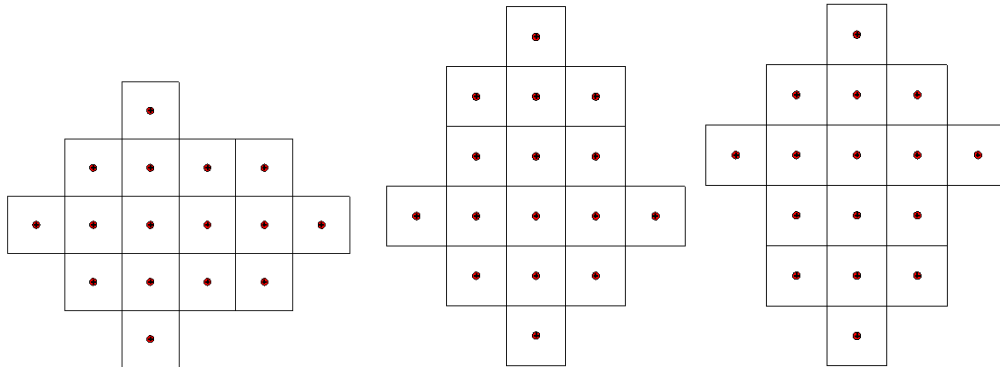
Plataform	Episodes	Sarsa(λ)		
SketchUp (Ruby 1.8.6)	3000	c[1] = -0.0016 c[2] = 0.0009 c[3] = 0.0137 c[4] = 0.0499 t: 2932.69s	c[1] = 3.7173e-005 c[2] = -0.0017 c[3] = 0.0204 c[4] = 0.043 t: 2875.99s	c[1] = 0.0006 c[2] = -0.0028 c[3] = 0.0201 c[4] = 0.054 t: 2584.5s
Netbeans (Ruby 1.8.6)	5000	c[1] = 0.0018 c[2] = -0.0066 c[3] = 0.014 c[4] = 0.0523 t: 2798.54s	c[1] = -0.0018 c[2] = -0.0015 c[3] = 0.0181 c[4] = 0.0462 t: 2697.27s	c[1] = 0.0001 c[2] = -0.0008 c[3] = 0.0246 c[4] = 0.0466 t: 2704.97s
SciTE (Ruby 1.9.2)	10000	c[1] = -0.0028 c[2] = -0.0037 c[3] = 0.0175 c[4] = 0.052 t: 1897.09s	c[1] = 0.0012 c[2] = -0.0045 c[3] = 0.019 c[4] = 0.0476 t: 1863.81s	c[1] = -0.0024 c[2] = 0.0027 c[3] = 0.0268 c[4] = 0.0553 t: 1868.65s

Table 4: Results of Sarsa(λ)

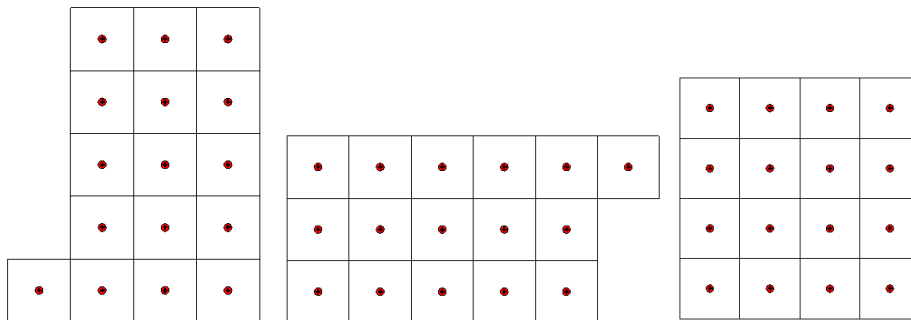
Where $c[i]$ is the coefficient for the feature that counts the squares with i neighbours.

All policies (coefficients) give more importance to the fourth and third features, in that order. Depending on the order of the first and second features, we have two kinds of policies: those which give priority to the second one and those which give priority to the first. Policies labelled with numbers 1 and 2 are proxies for these two classes. We have applied them several times:

Applications of policy 1:



Applications of policy 2:



As we can see, now the policies are not deterministic any more, since the value for (state, action) pairs that have the same features now are exactly the same. In case of tie, the policy chooses randomly.

3. Conclusions

Reinforcement learning is useful in combination with shape grammars, but only when generalization of states is used. Otherwise, the process is equivalent to an exploration of the state space. Moreover, using generalization, we can produce non-deterministic policies that produce different shapes, all of them satisfying the desired goals.

References

- [1] Stiny, G. (1980): Introduction to shape and shape grammars. Environment and Planning B, Vol. 7, No. 3. (1980), pp. 343-351
- [2] Sutton, R.S., Barto, A. G. (1998): Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning). MIT Press
- [3] Watkins, C. J. C. H. (1989). Learning from Delayed Rewards. Ph.D. thesis, Cambridge University
- [4] Thrun, S., Schwartz, A (1993). Issues in Using Function Approximation for Reinforcement Learning Proceedings of the 1993 Connectionist Models Summer School. Number: 255.