CS489 HW1 Report

Sheng Cen

March 2021

1 Introduction

In known Markov reward/decision process, it's necessary for us to evaluate certain policy and try to improve it, so that we could achieve optimal value given each state. Usually, we need to solve Bellman Expectation Equation and Bellman Optimality Equation, which cannot be tackled in acceptable time periods when the number of states is large. Dynamic programming is designed to use iterative methods to find desired value/policy for each state, which can usually reach convergence quickly. This report experiments on a simple grid world game, using dynamic programming to evaluate the random policy, and try to improve it using both policy iteration and value iteration. Comparison of these two methods is analyzed in the final part, and we conclude the report by assessing dynamic programming used in this setting.

2 Setting of Grid World

In the 6*6 grid world we have set up (see Figure 1), each grid represents a state. Grid 1 and 35 are two terminated states, in which we expect to reach as quickly as possible. In each state, four actions (towards east, west, south and north) could be taken, and in order to reach terminated states quickly, we penalize and assign a reward of -1 for each action. Also, actions are bounded in the 6*6 grid world, meaning that agent could not get out of the world. The initial policy we use is random policy, i.e., each direction we take in any non-terminated state is assigned to a probability of 1/4.

3 Dynamic Programming in Markov Reward and Decision Process

3.1 Policy Evaluation

For a known Markov reward/decision process, it's necessary to evaluate a given policy an agent plans to take. For each state, a value would be assigned to reflect its expected return, under the given policy. The value functions for all

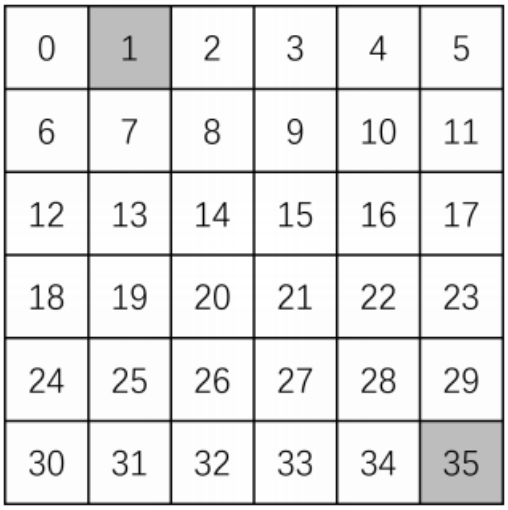

Figure 1: The Grid World

states could be obtained through dynamic programming, when convergence is reached. The pseudocode for the policy evaluation is shown in Figure 2.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Figure 2: Policy Evaluation

3.2 Policy Iteration

It's meaningless to only evaluate a given policy, so improvement should be executed after we evaluate the policy. Policy would be updated only if there exists a state that would gain a higher value when taken another action. Policy im-

provement and evaluation need to be done repetitively until we reach an optimal policy. The pseudocode for the policy iteration is shown in Figure 3.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 3: Policy Iteration

3.3 Value Iteration

We could also improve the policy by finding the optimum value function for each state first in iterative manner, then choose the optimum policy given these value functions. Therefore, we do not need to iteratively evaluate intermediate policy, saving lots of time. The pseudocode for the value iteration is shown in Figure 4.

4 Experiment

We use Python 3 to simulate the grid world, and use dynamic programming to effectively evaluate and improve the random policy.

4.1 Policy Evaluation

We set the discounting factor γ to be 1, and get the resulting value functions for each state. The accuracy threshold θ is set to be 0.001. Value function for each state is shown in Figure 5, and we need to iterate 216 times to reach convergence.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

Figure 4: Value Iteration

```
[-18.162202424101675, 0, -29.20743476830735, -44.04464994687094, -51.5365615623675, -54.65649015869137]
[-32.325456661830785, -30.15493698056146, -39.579412139829685, -47.392192740646934, -51.91102747756714, -53.778981490624766]
[-44.660922503533584, -44.716187838179394, -47.56454134276321, -50.03523421170065, -50.93796072727035, -50.77100662349406]
[-52.943410392049884, -52.485910850999545, -51.92888273201376, -50.247750933202184, -47.0360119848812, -43.597456112667665]
[-57.686050022981235, -56.35685659431247, -53.41889477270166, -47.992265003934726, -39.36206191670355, -28.986373080166103]
[-59.760727384852956, -57.83831374389077, -53.39909989212457, -44.94156771613735, -29.434406504886503, 0]
Total Iteration: 216
```

Figure 5: Result for Policy Evaluation

4.2 Policy Iteration

We set the discounting factor γ to be 0.9, and the accuracy threshold θ is set to be 0.001. Probabilistic policy matrix for each state is shown in Figure 6, while value function is shown in Figure 7. Policy gets stable after three rounds of improvement.

4.3 Value Iteration

We set the discounting factor γ to be 0.9, and the accuracy threshold θ is set to be 0.001. Value function for each state is shown in Figure 8, while probabilistic policy matrix is shown in Figure 9. Optimum value functions are obtained after six rounds of improvement.

5 Analysis for Experiment

For the policy evaluation part, we notice that the final stable value function is intuitive. Expected return is small in the upper-right and bottom-left part of

```
Up probabilistic matrix:
[0, 0, 0, 0, 0, 0]
[0.5, 1.0, 0.5, 0.5, 0.5, 0]
[0.5, 1.0, 0.5, 0.5, 0, 0]
[0.5, 1.0, 0.5, 0, 0, 0]
[0.5, 1.0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
Down probabilistic matrix:
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1.0]
[0, 0, 0, 0, 0.5, 1.0]
[0, 0, 0, 0.5, 0.5, 1.0]
[0, 0, 0.5, 0.5, 0.5, 1.0]
[0, 0, 0, 0, 0, 0]
Left probabilistic matrix:
[0, 0, 1.0, 1.0, 1.0, 1.0]
[0, 0, 0.5, 0.5, 0.5, 0]
[0, 0, 0.5, 0.5, 0, 0]
[0, 0, 0.5, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
Right probabilistic matrix:
[1.0, 0, 0, 0, 0, 0]
[0.5, 0, 0, 0, 0, 0]
[0.5, 0, 0, 0, 0.5, 0]
[0.5, 0, 0, 0.5, 0.5, 0]
[0.5, 0, 0.5, 0.5, 0.5, 0]
[1.0, 1.0, 1.0, 1.0, 1.0, 0]
```

Figure 6: Probabilistic policy matrix for Policy Iteration

```
Value matrix:
[-1.0, 0, -1.0, -1.9, -2.71, -3.439]
[-1.9, -1.0, -1.9, -2.71, -3.439, -3.439]
[-2.71, -1.9, -2.71, -3.439, -3.439, -2.71]
[-3.439, -2.71, -3.439, -3.439, -2.71, -1.9]
[-4.0951, -3.439, -3.439, -2.71, -1.9, -1.0]
[-4.0951, -3.439, -2.71, -1.9, -1.0, 0]
Improvement Count:
3
```

Figure 7: Value Function for Policy Iteration

```
Value matrix:
[-1.0, 0, -1.0, -1.9, -2.71, -3.439]
[-1.9, -1.0, -1.9, -2.71, -3.439, -3.439]
[-2.71, -1.9, -2.71, -3.439, -3.439, -2.71]
[-3.439, -2.71, -3.439, -3.439, -2.71, -1.9]
[-4.0951, -3.439, -3.439, -2.71, -1.9, -1.0]
[-4.0951, -3.439, -2.71, -1.9, -1.0, 0]
```

Figure 8: Value Function for Value Iteration

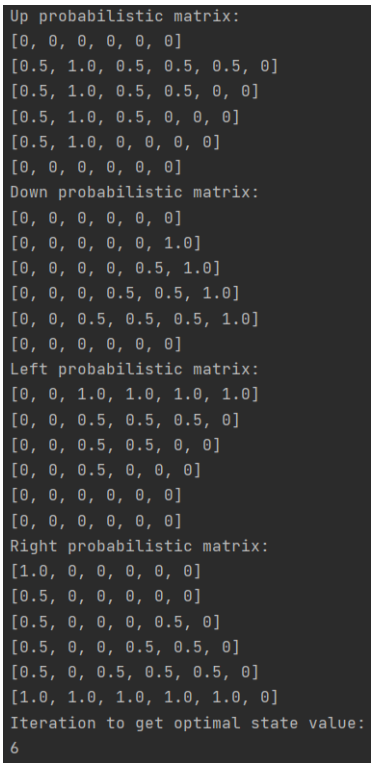

Figure 9: Probabilistic policy matrix for Value Iteration

the grid world, since these states are far away from the terminated states. And the expected return is large when close to grid 1 and 35.

For the last two experiments meant to improve the random policy and get optimum value function and policy, we find that both methods (policy iteration and value iteration) get the optimum policy and value function. Policy iteration needs fewer rounds of improvement, but needs to evaluate the intermediate policies after each improvement; while value iteration needs more rounds of improvement, but only needs to derive the optimum policy. More time needs to be consumed when using policy iteration.

The final optimum policy is shown in Figure 10.

6 Conclusion

Dynamic programming avoids the problem of solving Bellman Equation directly, and in the simple grid world setting, it can achieve optimum policy in few rounds of improvement.

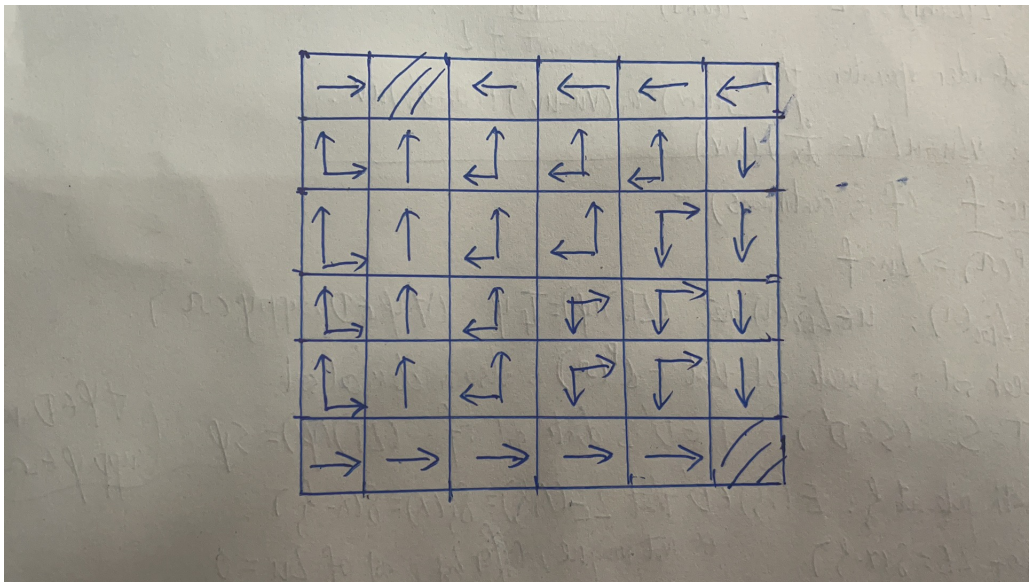

Figure 10: Optimum Policy Derived from the Experiments

References

[1] Prof. Junni Zou. *Reinforcement Learning, Lecture 3.*