Assignment 1

Yuxiang Lu 518021911194

1 Introduction

Dynamic programming is an effective method for solving MDP problem. In this assignment, three dynamic programming methods, policy evaluation, policy iteration and value iteration are implemented to solve the small Gridworld, a classic MDP problem.

2 Small Gridworld

Small Gridworld is a classic example for MDP problem. As shown in Figure 1, the Gridworld used in this assignment is 6×6, each grid represents a certain state.

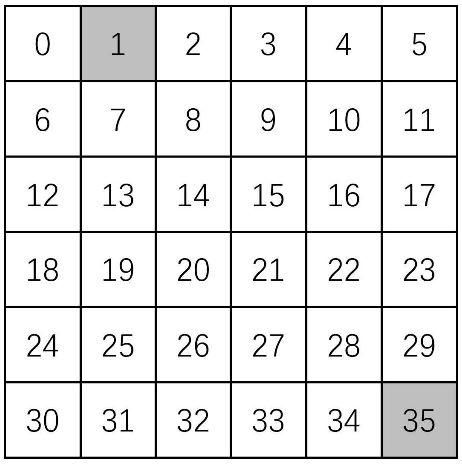


Figure 1: Gridworld

For a MDP problem, we should first define the tuple $\langle S, A, P, R, \gamma \rangle$:

- S is the state space, which can be denoted as $S = \{s_t | t \in \{0, \dots, 35\}\}$, while s_t stands for the state in grid t. Each state can also be described by its location (i, j), for $i, j \in \{0, \dots, 5\}$. In particular, s_1 and s_{35} are terminal states, which means the process will terminate once entering these grids.

- A is the action space, each state except the terminal states, has four actions: North, South, East and West, which can be denoted as $A = \{n, s, e, w\}$. The actions leading out of the Gridworld leave the state unchanged, mainly about the grids on the edges.

- P is the state transition probability matrix, since every tranition in the Gridworld is definite, for any valid tuple $\langle s, a, s' \rangle$, there is

$$P^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a] = 1$$

- R is the reward function, which is denoted as

  $$R_s^a = E[R_{t+1}|S_t = s, A_t = a]$$

  Each action will get a reward -1 except that s_t is the terminal state.

- γ is the discount value, usually Gridworld is an undiscounted episodic MDP, which means $\gamma = 1$. Different discount values are also discussed in Section 6.

3 Policy Evaluation

Iterative Policy Evaluation is used to evaluate a given policy π by calculate the value function v_π. It uses dynamic programming to iteratively update the Bellman expectation equation for state value:

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s')) \tag{1}$$

Policy evaluation uses synchronous backups, which means that at each iteration $k+1$, for all state $s \in S$, $v_{k+1}(s)$ is updated from $v_k(s')$ where s' is a successor state of s.

In policy evaluation, all four actions are selected randomly, so the probability to take each action $\pi(a|s) = 0.25, a \in A$. Note that $P_{ss'}^a = 1$, the Bellman expectation function can be simplified as

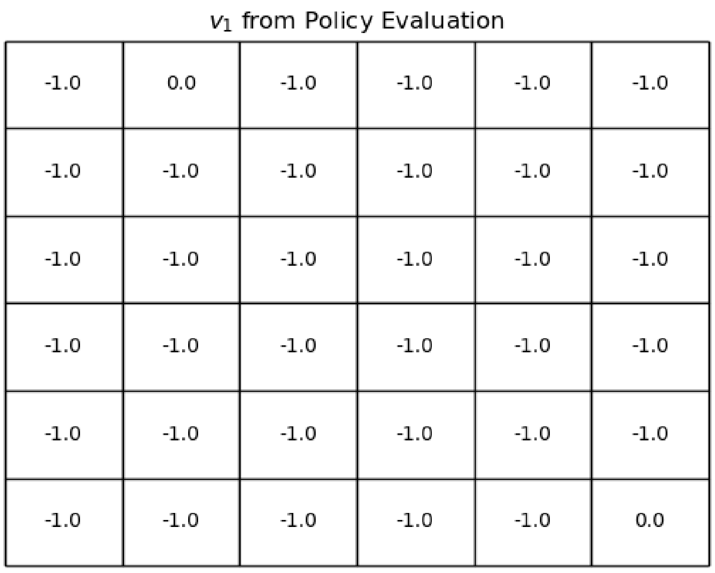
$$v_{k+1}(s) = \sum_{a \in A} 0.25(R_s^a + \gamma v_k(s')) \tag{2}$$

The complete algorithm is shown in Algorithm 1, the parameter $\theta > 0$ is a small threshold determining the accuracy of estimation. θ is set as $1e-4$ in this assignment.

Algorithm 1: Iterative Policy Evaluation

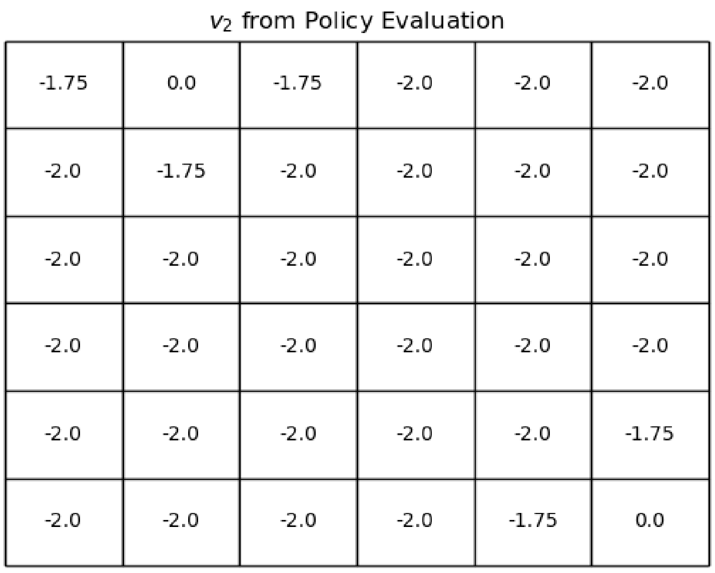
```
  Input: $\pi$
  Output: $v(s)$
1 repeat
2     $\Delta \leftarrow 0$;
3     for each $s \in S$ do
4         $v \leftarrow v(s)$;
5         $v_(s) \leftarrow \sum_{a \in A} 0.25(R_s^a + \gamma v(s'))$;
6         $\Delta \leftarrow \max(\Delta, |v - v(s)|)$
7     end
8 until $\Delta < \theta$;
```

When $k = 0$, all state values v_0 are initialized to 0. After one iteration, the state values in v_1 are shown in Figure 2. v_k after 2, 10, 100 iterations are shown in Figure 3∼5. We can find that the state values of non-terminal states are gradually getting smaller in each iteration.


v_1 from Policy Evaluation

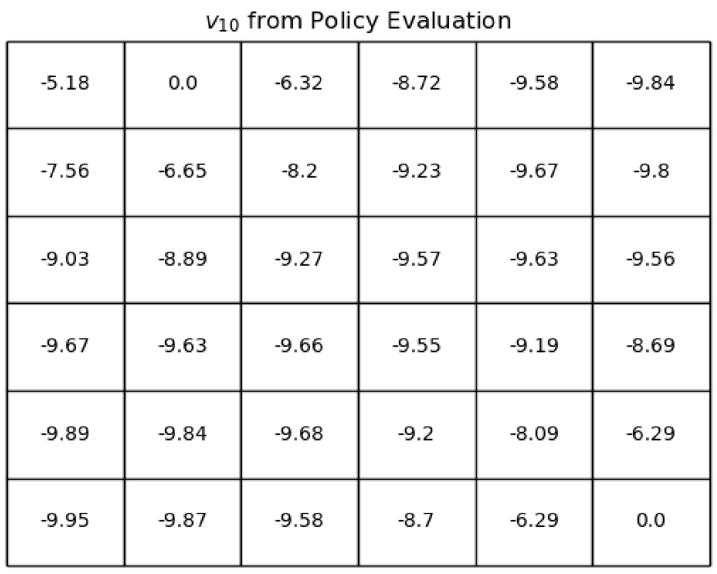
-1.0	0.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0	-1.0	0.0

Figure 2: v_1 state value


v_2 from Policy Evaluation

-1.75	0.0	-1.75	-2.0	-2.0	-2.0
-2.0	-1.75	-2.0	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-2.0	-2.0	-1.75
-2.0	-2.0	-2.0	-2.0	-1.75	0.0

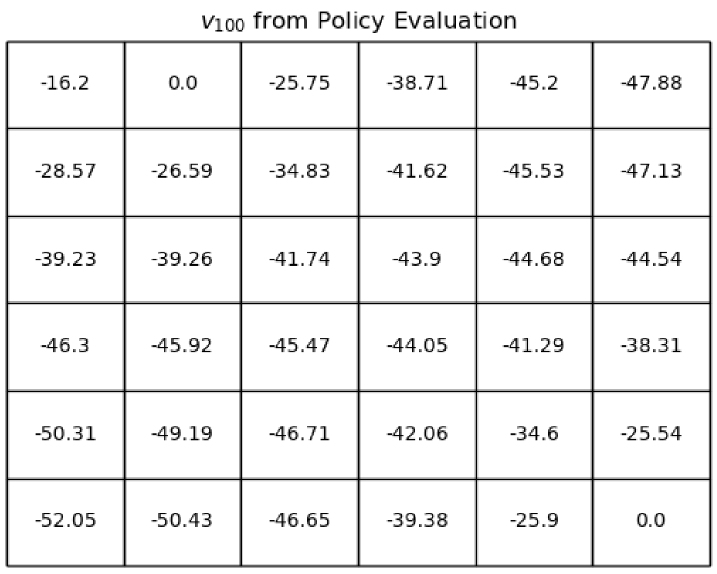
Figure 3: v_2 state value



v_{10} from Policy Evaluation

-5.18	0.0	-6.32	-8.72	-9.58	-9.84
-7.56	-6.65	-8.2	-9.23	-9.67	-9.8
-9.03	-8.89	-9.27	-9.57	-9.63	-9.56
-9.67	-9.63	-9.66	-9.55	-9.19	-8.69
-9.89	-9.84	-9.68	-9.2	-8.09	-6.29
-9.95	-9.87	-9.58	-8.7	-6.29	0.0

Figure 4: v_{10} state value

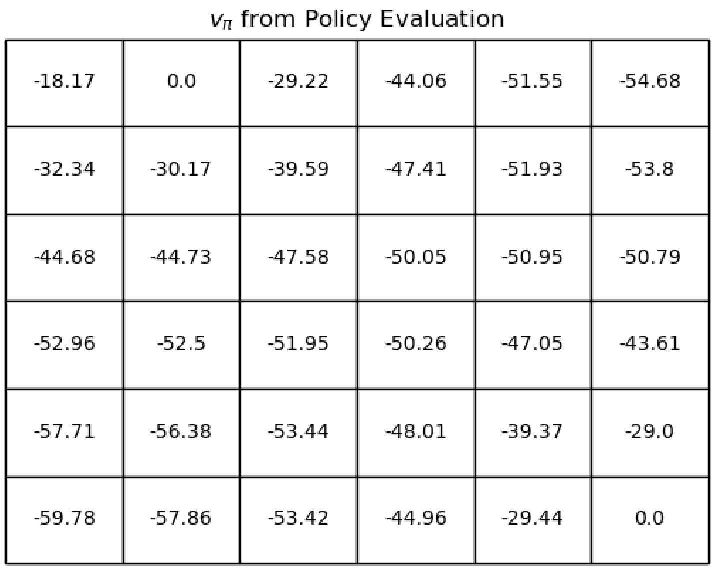


v_{100} from Policy Evaluation

-16.2	0.0	-25.75	-38.71	-45.2	-47.88
-28.57	-26.59	-34.83	-41.62	-45.53	-47.13
-39.23	-39.26	-41.74	-43.9	-44.68	-44.54
-46.3	-45.92	-45.47	-44.05	-41.29	-38.31
-50.31	-49.19	-46.71	-42.06	-34.6	-25.54
-52.05	-50.43	-46.65	-39.38	-25.9	0.0

Figure 5: v_{100} state value

After 455 iterations, the iterative process stops as the maximum change in a loop is smaller than the threshold. The output state value v_π from policy evaluation is shown in Figure 6.



v_π from Policy Evaluation

-18.17	0.0	-29.22	-44.06	-51.55	-54.68
-32.34	-30.17	-39.59	-47.41	-51.93	-53.8
-44.68	-44.73	-47.58	-50.05	-50.95	-50.79
-52.96	-52.5	-51.95	-50.26	-47.05	-43.61
-57.71	-56.38	-53.44	-48.01	-39.37	-29.0
-59.78	-57.86	-53.42	-44.96	-29.44	0.0

Figure 6: v_π state value

4 Policy Iteration

Policy Iteration is used to improve a deterministic policy $a = \pi(s)$ by acting greedily

$$\pi'(s) = arg \max_{a \in A} q_\pi(s, a)$$

This can improve the state value because for any state s, there is

$$v_{\pi'} = q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

For each iteration, the first step is to evaluate the policy π by policy evaluation to get $v_\pi(s)$. Then in the second step, the policy is improved by acting greedily according to v_π to find a better action for each state. When the policy cannot be improved any more, we should have the optimal policy π_* and $v_\pi = v_*$, which satisfies the Bellman Optimality Equation:

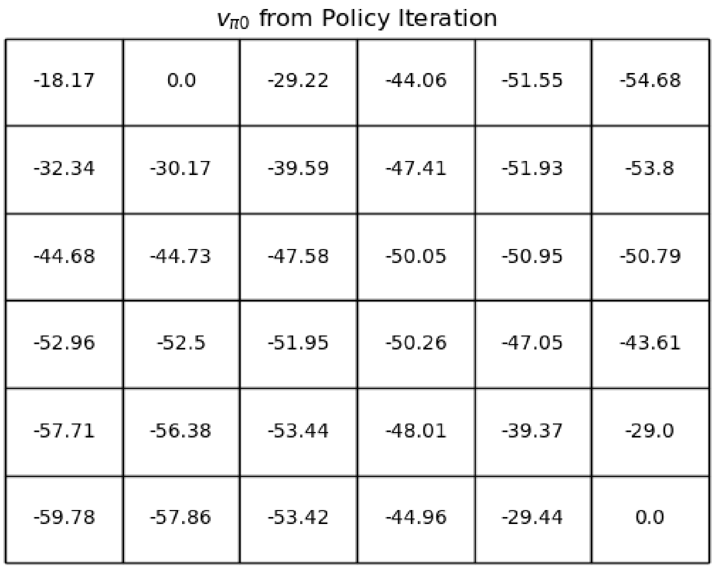
$$v_*(s) = \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \tag{3}$$

The complete algorithm of policy iteration is shown in Algorithm 2, which is mainly a combination of policy evaluation and policy improvement. Moreover, the policy $\pi(s)$ may include more than one choice, in this case the state values of the successors s' are the same, so the policy can select anyone of them.

```
Algorithm 2: Policy Iteration
  Input: $\pi_0, v_0$
  Output: $\pi_*$
  repeat
      repeat
          $\Delta \leftarrow 0$;
          for each $s \in S$ do
              $v \leftarrow v(s)$;
              $v_(s) \leftarrow R + \gamma v(s')$;
              $\Delta \leftarrow \max(\Delta, |v - v(s)|)$;
          end
      until $\Delta < \theta$;

      $policy - stable \leftarrow true$;
      for each $s \in S$ do
          $old - action \leftarrow \pi(s)$;
          $\pi(s) \leftarrow arg \max_a R + \gamma v(s')$;
          if $old - action \neq \pi(s)$ then
              $policy - stable \leftarrow false$;
          end
      end
  until $policy - stable$;
  output $\pi(s)$
```

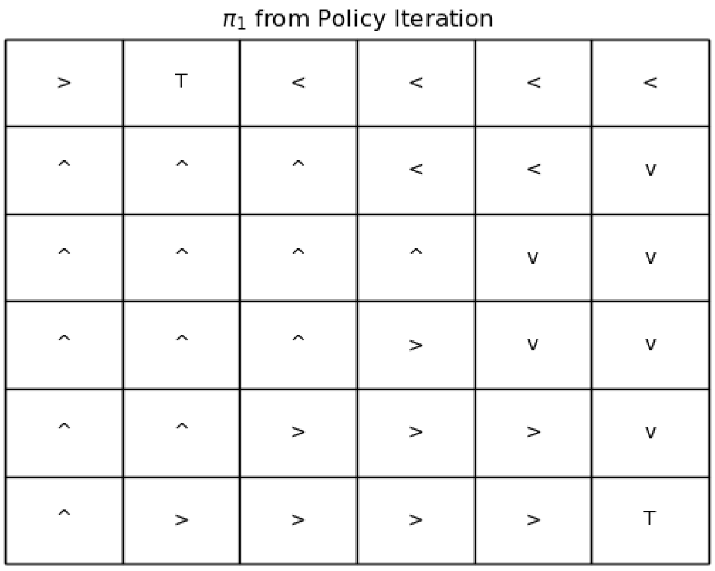
At the beginning, state values in v_0 are initialized to 0, and the policy π_0 is random selection. The first value evaluation gets the state value $v_{\pi 0}$, shown in Figure 7. According to $v_{\pi 0}$, policy is updated to π_1, shown in Figure 8.



$v_{\pi 0}$ from Policy Iteration

-18.17	0.0	-29.22	-44.06	-51.55	-54.68
-32.34	-30.17	-39.59	-47.41	-51.93	-53.8
-44.68	-44.73	-47.58	-50.05	-50.95	-50.79
-52.96	-52.5	-51.95	-50.26	-47.05	-43.61
-57.71	-56.38	-53.44	-48.01	-39.37	-29.0
-59.78	-57.86	-53.42	-44.96	-29.44	0.0

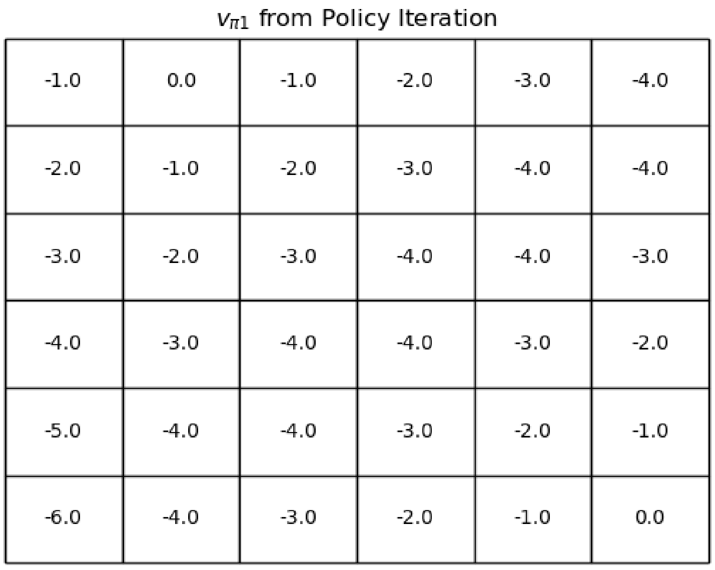
Figure 7: $v_{\pi 0}$ state value


π_1 from Policy Iteration

>	T	<	<	<	<
^	^	^	<	<	v
^	^	^	^	v	v
^	^	^	>	v	v
^	^	>	>	>	v
^	>	>	>	>	T

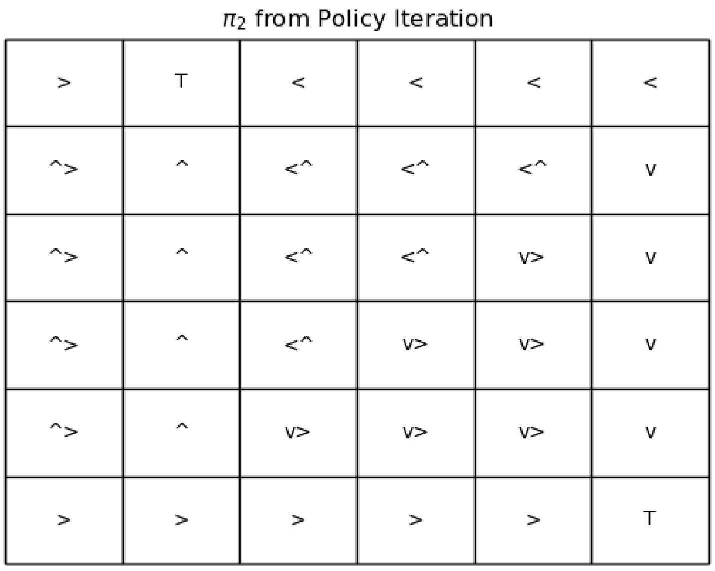
Figure 8: π_1 policy

Then policy evaluation and policy improvement are repeated to get state value $v_{\pi 1}$ and policy π_2, shown in Figure 9 and 10.


$v_{\pi 1}$ from Policy Iteration

-1.0	0.0	-1.0	-2.0	-3.0	-4.0
-2.0	-1.0	-2.0	-3.0	-4.0	-4.0
-3.0	-2.0	-3.0	-4.0	-4.0	-3.0
-4.0	-3.0	-4.0	-4.0	-3.0	-2.0
-5.0	-4.0	-4.0	-3.0	-2.0	-1.0
-6.0	-4.0	-3.0	-2.0	-1.0	0.0

Figure 9: $v_{\pi 1}$ state value

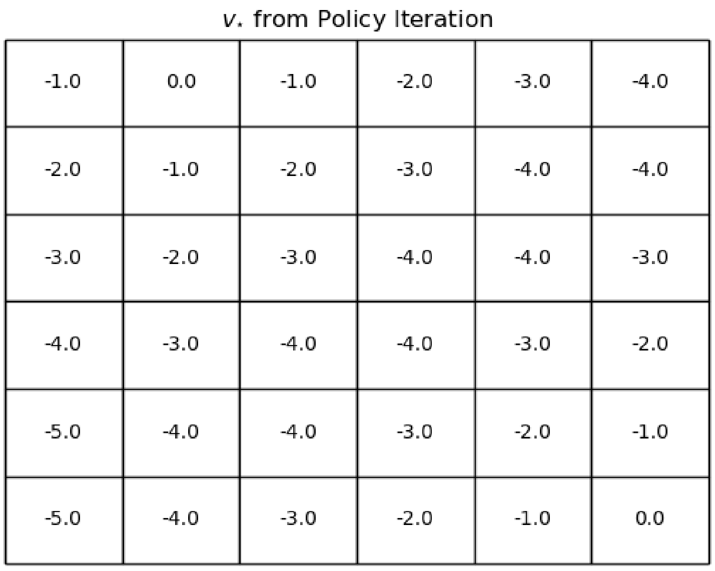


π_2 from Policy Iteration

>	T	<	<	<	<
^>	^	<^	<^	<^	v
^>	^	<^	<^	v>	v
^>	^	<^	v>	v>	v
^>	^	v>	v>	v>	v
>	>	>	>	>	T

Figure 10: π_2 policy

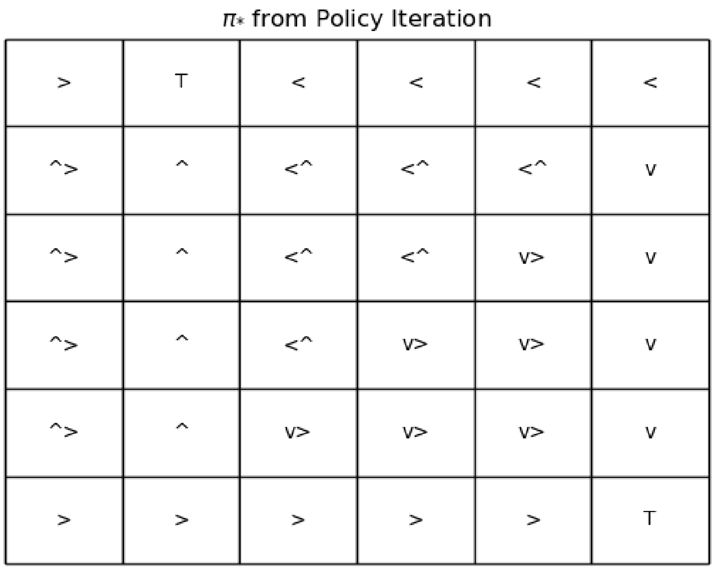
Then $v_{\pi2}$ is evaluated in Figure 11, and get policy π_3 in Figure 12. We could find that π_3 is the same as π_2, so the iteration stops and π_3 is the optimal policy while the $v_{\pi2}$ is the v_* that satisfies the Bellman optimality equation.



v_* from Policy Iteration

-1.0	0.0	-1.0	-2.0	-3.0	-4.0
-2.0	-1.0	-2.0	-3.0	-4.0	-4.0
-3.0	-2.0	-3.0	-4.0	-4.0	-3.0
-4.0	-3.0	-4.0	-4.0	-3.0	-2.0
-5.0	-4.0	-4.0	-3.0	-2.0	-1.0
-5.0	-4.0	-3.0	-2.0	-1.0	0.0

Figure 11: $v_{\pi2} = v_*$ state value


π* from Policy Iteration

>	T	<	<	<	<
^>	^	<^	<^	<^	v
^>	^	<^	<^	v>	v
^>	^	<^	v>	v>	v
^>	^	v>	v>	v>	v
>	>	>	>	>	T

Figure 12: $\pi_3 = \pi_*$ policy

We could also find that state value $v_{\pi 0}$ in Figure 7 is the same as the result from policy evaluation in Figure 6. Since both the initial policy π_0 and policy in Section 3 are random selection, the evaluation results of the same policy are sure to be identical.

5 Value Iteration

Value Iteration is also used to find the optimal policy. The difference between value iteration and policy iteration is that value iteration does policy improvement and policy evaluation simultaneously. During each iteration in policy iteration, state value is updated by Bellman optimality equation:

$$v_{k+1}(s) = \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \tag{4}$$

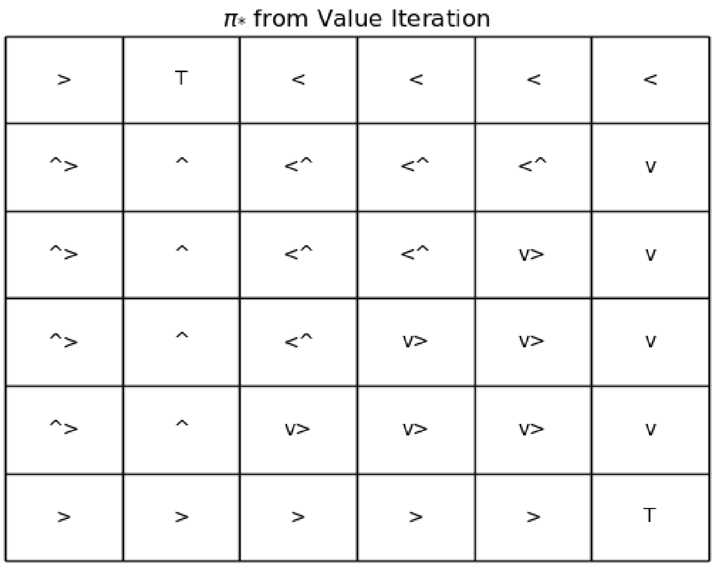
When the iteration converges, we will get the optimal state value v_*, and can conduct the optimal policy π_* by:

$$\pi_*(s) = arg \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \tag{5}$$

The complete algorithm of value iteration is shown in Algorithm 3. Generally, value iteration is a better version of policy evaluation and a simplified version of policy iteration.

```
Algorithm 3: Value Evaluation
  Input: $v_0$
  Output: $\pi_*$
repeat
    $\Delta \leftarrow 0$;
    for each $s \in S$ do
        $v \leftarrow v(s)$;
        $v_(s) \leftarrow \max_a[R_s^a + \gamma v(s')]$;
        $\Delta \leftarrow \max(\Delta, |v - v(s)|)$
    end
until $\Delta < \theta$;
output $\pi_*(s) = arg \max_a[r + \gamma v(s')]$
```

The optimal policy π_* resulting from value iteration is shown in Figure 13, which is the same as the result from policy iteration in Figure 11. However, this process only takes 6 loops to converge to the optimality, which is much faster than policy evaluation.


π_* from Value Iteration

>	T	<	<	<	<
^>	^	<^	<^	<^	v
^>	^	<^	<^	v>	v
^>	^	<^	v>	v>	v
^>	^	v>	v>	v>	v
>	>	>	>	>	T

Figure 13: π_* policy

6 Discussion

6.1 Early Stop in Policy Iteration

With respect to the loop cycles in policy evaluation part in policy iteration, evaluation of π_0 takes 455 loops, while evaluations of π_1 and π_2 take 6 loops and 1 loop respectively. With the policy being improved, the policy will converges faster. So a question rises, whether policy evaluation process need to converge, or it can stop earlier?

Looking at Figure 4, we could find that after 10 iterations, the information from state value is enough to improve the policy to some extent. Therefore the idea is to simply stop after k iterations. Different k are tested for policy iteration, and the results are shown in Table 1.

When k equals 2 or 3, it takes four policy iterations (outer loop), while every policy evaluation part is restricted to k iterations (inner loop). If k is increased to greater or equal to 4, only three outer loops are needed before obtaining the optimal policy. Remember that if early stop is not applied, the first policy evaluation would take 455 loops, which means the early stop strategy does work in policy iteration.

Considering the total loops taken, the parameter k can value from 2 to 5, to make the algorithm more efficient. When k decreases to 1, we will get the value iteration algorithm. As mentioned in Section 5, value iteration only takes 6 loops totally, which outperforms all these trials.

Table 1: Iteration times for different k

k	v_{π_0} eval	v_{π_1} eval	v_{π_2} eval	v_{π_3} eval	outer loops	total loops
2	2	2	2	2	4	12
3	3	3	3	1	4	14
4	4	4	2	/	3	13
5	5	5	1	/	3	14
6	6	6	1	/	3	16
7	7	6	1	/	3	17

6.2 Discount matters

In the problems above, the discount value γ is set as 1 in the MDP. But whether the discount value influence on the performance of the algorithm? Different discount values are tested, ranging from 0 to 1, on policy evaluation described in Section 3. The loop cycles before the process converges are counted and the result is shown in Figure 14.

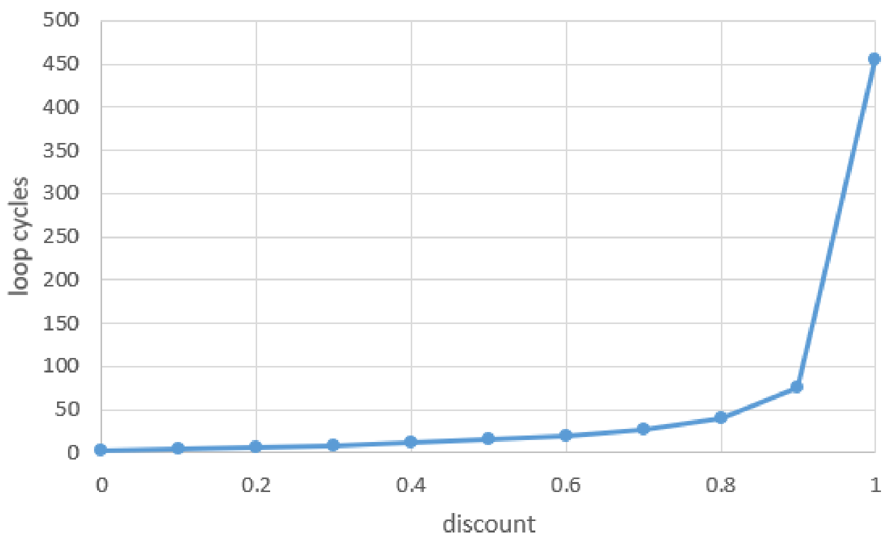


Figure 14: discount influence on loop cycles

We could find that just by decreasing the discount to 0.9, the loop cycle drops incredibly to 76. If the discount is reduced further to less or equal to 0.6, it would take less than 20 loops to converge. So we can conclude that the smaller the discount is, the more quickly the algorithm converges.

When the discount goes down, the evaluation would consider less on the future reward and mainly pay attention on the immediate reward. With fewer factors, the process tends to converge quickly.