Reinforcement Learning Project1

Hui Liu(hliu58@vols.utk.edu)

Yang Xu(yxu71@vols.utk.edu)

1. Introduction

Sam is the sale person at a Unique Ice-cream store. There are N ice-cream flavors at this store, and people will line up in the quene for the flavors they want. However, the current bussiness plan is that Sam is the only sales person at the store and he can only serve one customer each time. Everytime one customer gets the ice-cream and will leave the store immediately, and other customers will show up and wait in the N quenes. Each flavor of ice-cream has its own popularity and the store only can fit M people for each ice-cream line. Sam's goal is to serve as many customers as he can and avoid the loss of customers because of the store capacity. Therefore, we design a reinforcement learning agent to help sam achieve an optimal goal.

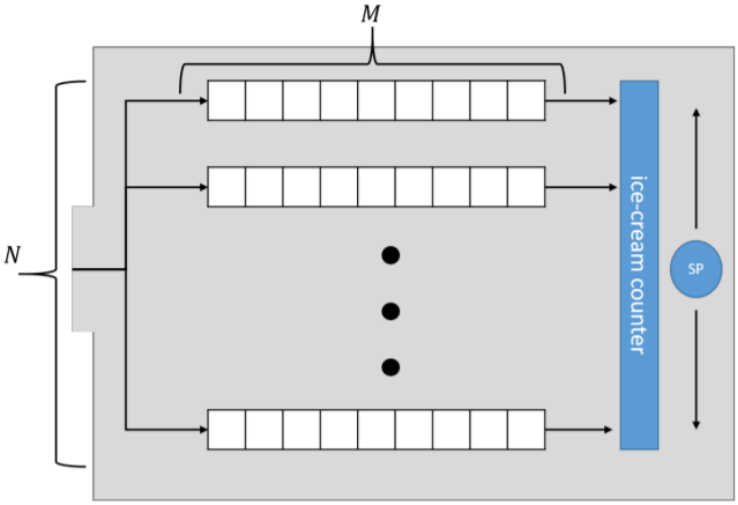


Fig.1 A diagram of the problem

2. Dynamic programming

We approach to this problem by applying dynamic programming to the learning agent. Dynamic programming algorithms are utilized to search for the optmial policies under the perfect model of environment, which we usually assume as a finite Markov Decision Process(MDP). Simply put, a set of probabilities $p(s', r|s, a)$, (taking action A under state S, get Reward R and transit into State S').

The dynamic programming in work for finding optimal policies contains two parts. In the first part, Bellman equations are caculated to iterationally update value functions to the approxiamtions, since the existence and uniqueness of v_π are guaranteed as long as $\gamma < 1$ or eventual termination is certain from all states under the policy π. This is known as policy evaluation. In the second part, new policy will be generated with greedy method according to

the converged value function. As a result, an improved policy is guaranted, which is known as policy improvement.

3. Workframe for problem

Assuming that there are N queues in the stores and the capacity of each line is M. We denote $s \in S^N$ to represent the state. For each state, $s = [k_1, k_2, k_3, \dots k_n]$ for all $0 \leq k_i \leq M$, which means k_i guests are waiting in the i_{th} queue at each time step.

Also, the action is defined as $a \in A$, where $a \in [1, N]$. The action value indicates which queue is supposed to be served. Supposing the cost of the ice cream is c and l is the loss for a customer leaving, the reward of each time step follows $r \in R$, where $r \in \{0, c, l^q, c - l^q\}$ for all $0 \leq q \leq M$. q is the number of leaving customer.

$P_{sa} = p(s', r|s, a)$ denotes the transition probabilities from current state s transmitting to next state s' and obtaining a reward r. Assuming the number of people arriving at each queue as a Bernoulli i.i.d process with specific p_q for the q_{th} line, the transition probabilities is computed as following:

$$P_{sa} = p_a \times \prod_q p_q^{iscoming} (1 - p_q)^{iscoming}.$$

As a result, the problem is designed as an MDP $M = (S, A, P_{sa}, R)$.

4.Data structures and code design

To express state value, we designed it as a $(M+1)^N$ dimension array. At each state, there are N possible actions. To express policy, we created a $(M+1)^N \times N$ dimension array. Transtion from the current state to next state is dependent on N. Thus it turns out 2^N outcomes, and we created a $2^N \times N$ dimension array named after *people* for possible arriving customers. The arriving rate of one of N flavors ice-creams is modeled as a Bernoulli i.i.d process with a specific p. We created the p as a N dimension array, and we can compute transition probality matrix by following.

$$\prod p^{people} (1 - p)^{(1-people)}.$$

In our code, we built 9 functions in total, excluding the main function. The description of the functions are shown if Table.1.

Table.1 Description of functions

Function name	Description
initiate_state_value()	Generate full-zero state values for policy iteration
initiate_random_policy()	Generating a random policy for policy iteration
policy_evalution(state_value,policy)	Given current state value and policy, update value functions by sweeping all states.
policy_improvement(state_value)	Generate better policy according to the current value functions.
policy_stable(new_policy,old_policy)	By comparing the policies to decide if the policy is optimal
policy_iteration()	Perform the policy iteration and return an

	optimal policy and optimal value function
initiate_state_value2()	Generate full-zero state values for value iteration with a different design
initiate_random_policy2()	Generating a policy matrix to store the optimal policy
value_iteration()	Perform the value iteration and return an optimal policy and optimal value function

For policy iteration, the code starts with initiating state values and generating a random policy, and policy evalution is followed by policy improvement. Then, function policy_stable() will determine the stability by comparing the old policy and the new one. If it is unstable, the code loops back to policy evalution followed by policy improvement, until the policy becomes stable. For value iteration, we simply defined a function called value_iteration that loops over every state until convergence of state values and then generates the optimal policy.

5. Results of example

Using the setup of the example, we run our code and record the results, which is shown as following.

5.1 Optimal value functions

Setting theta as 0.01 and after a collection of updating, the value functions converge to a very similar results by using value iteration and policy iteration respectively, which is shown in Fig.2 and Fig.3. The number of epoch for iteration is also recorded. Value iteration goes through 86 epochs and policy iteration goes through 128 epochs. Given the same theta, value iteration converges faster than policy iteration. In other words, value iteration learns faster than policy iteration.

	0	1	2	3	4	5	6	7	8
0	7.46071	8.46071	9.06673	9.43399	9.65653	9.79132	9.87283	9.92177	9.95027
1	8.46071	9.06673	9.43399	9.65653	9.79132	9.87283	9.92177	9.95027	9.96456
2	9.06673	9.43399	9.65653	9.79132	9.87283	9.92177	9.95027	9.96456	9.96557
3	9.43399	9.65653	9.79132	9.87283	9.92177	9.95027	9.96456	9.96557	9.94654
4	9.65653	9.79132	9.87283	9.92177	9.95027	9.96456	9.96557	9.94654	9.88461
5	9.79132	9.87283	9.92177	9.95027	9.96456	9.96557	9.94654	9.88461	9.71772
6	9.87283	9.92177	9.95027	9.96456	9.96557	9.94654	9.88461	9.71772	9.28455
7	9.92177	9.95027	9.96456	9.96557	9.94654	9.88461	9.71772	9.28455	8.16975
8	9.95027	9.96456	9.96557	9.94654	9.88461	9.71772	9.28455	8.16975	5.30652

Fig. 2 State values for optimal policy through value iteration (Epoch=86)

	0	1	2	3	4	5	6	7	8
0	7.46174	8.46174	9.06776	9.43501	9.65754	9.7923	9.87377	9.92264	9.95102
1	8.46174	9.06776	9.43501	9.65754	9.79231	9.87377	9.92265	9.95104	9.96517
2	9.06776	9.43501	9.65754	9.79231	9.87378	9.92266	9.95105	9.9652	9.966
3	9.43501	9.65754	9.79231	9.87379	9.92267	9.95107	9.96521	9.96604	9.94676
4	9.65754	9.79232	9.87379	9.92268	9.95109	9.96524	9.96607	9.94682	9.88462
5	9.79232	9.8738	9.92269	9.9511	9.96527	9.96611	9.94687	9.8847	9.71752
6	9.8738	9.9227	9.95112	9.96529	9.96615	9.94693	9.88476	9.71763	9.28419
7	9.9227	9.95113	9.96532	9.96619	9.94698	9.88484	9.71772	9.28431	8.16933
8	9.95115	9.96534	9.96622	9.94703	9.88491	9.71781	9.28443	8.16946	5.30613

Fig. 3 State values under optimal policy through policy iteration (Epoch=128)

The converge process of sum of state values is shown in Fig.4. The number converges at the 86th epoch with value iteration method.

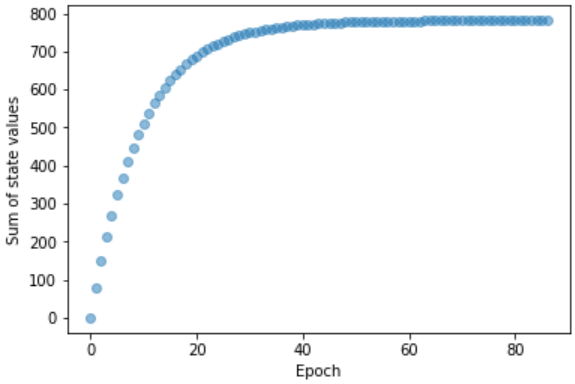


Fig. 4 Converge process of sum of state values with value iteration

5.2 The progression of the policy

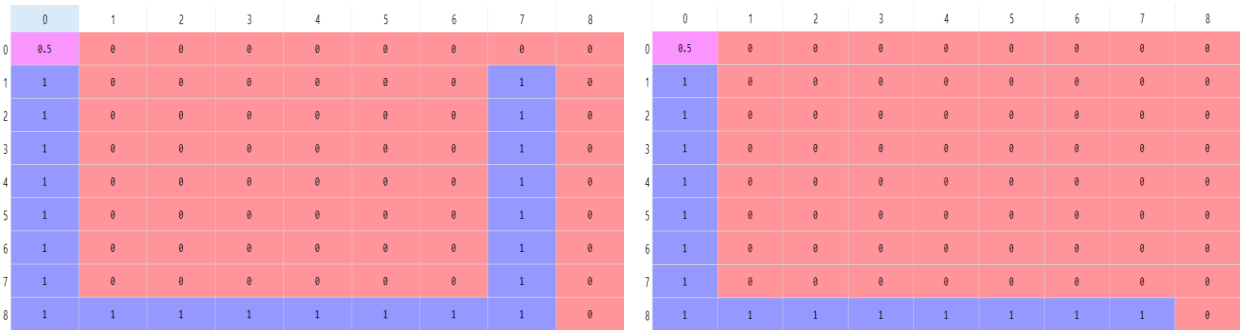
With policy iteration, the random policy starts with half possibility of serveing 1st ice-cream or 2rd ice-cream. Ater 4 iteration, policy was updated to a stable level and also is optimal. For the final policy (optimal policy), 1 indicates the possibility of serving 1st ice-cream. The progression is shown in Fig.5.

	0	1	2	3	4	5	6	7	8
0	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
2	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
3	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
4	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
6	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
7	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
8	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

(1) π_0

	0	1	2	3	4	5	6	7	8
0	0.5	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0
4	1	1	0	0	0	0	0	0	0
5	1	1	0	0	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0
7	1	1	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1	0	0

(2) π_1



(3) π_2 (4) $\pi_{optimal}$

Fig. 5 The progression of the policy with policy iteration

6. Results analysis

Question1: How will the agent learn, if the arriving rate of one flavor is 0.9 and the other is 0.1?

Intuitively, we would expect the agent prefer to serve the ice-cream with higher popularity. However, the optimal policy via policy iteration is in Fig.6.

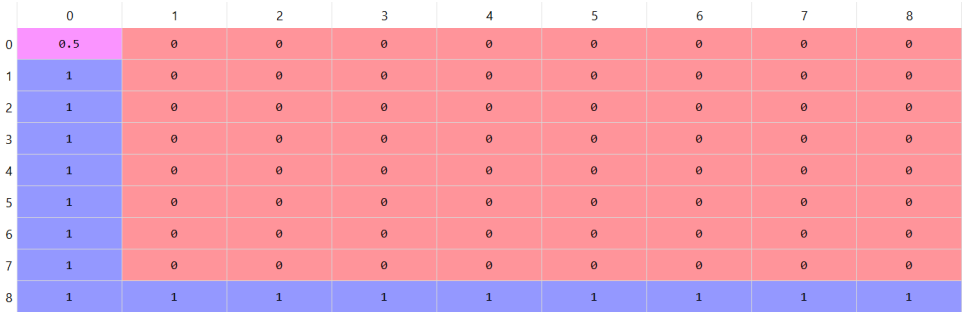
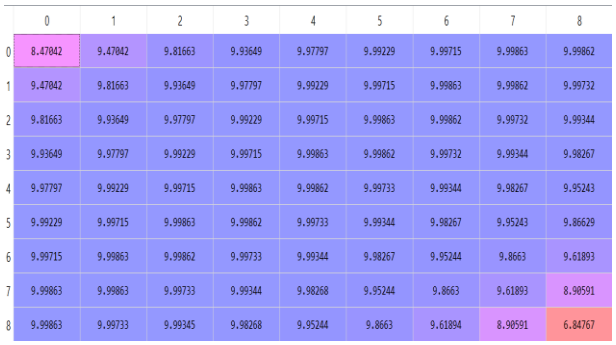


Fig. 6 optimal policy via policy iteration

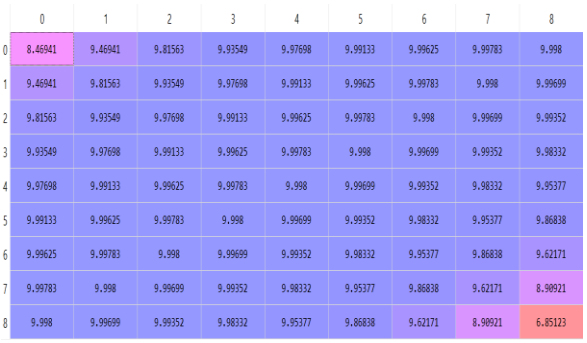
1 indicates the probability of serving the high popular ice-cream. From the result above, our agent learns to treat customers who want the low popularity ice-cream as VIP customer, even though the arriving rate is just 0.1. But we still argue this is a better business plan in reality. 1) Though more people are likely to wait in the popular ice-cream, the agent only serves this ice-cream when the people in line are close to the maximum. Therefore, the agent always gets positive rewards without loss of customers; 2) Agent learns to use the least the effort to get rewards; 3) In real business, companies may give priority to preferred customers. They are not a large group as other normal customers. Since preferred customers are given more priorities, more normal customers will be allured to become a preferred customer. In this case, our agent's sale plan may bring more customers to the low popular ice-cream.

Question 2: For some specific situations, we use same parameters for both value iteration and policy iteration respectively. Though we get very similar optimal value functions, the optimal policy holds some differences. The result is shown in Fig.7.



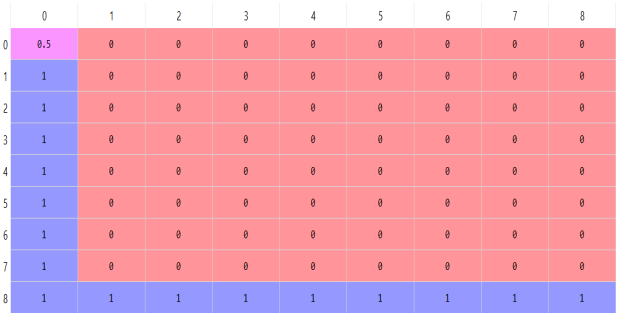
	0	1	2	3	4	5	6	7	8
0	8.47042	9.47042	9.81663	9.93649	9.97797	9.99229	9.99715	9.99863	9.99862
1	9.47042	9.81663	9.93649	9.97797	9.99229	9.99715	9.99863	9.99862	9.99732
2	9.81663	9.93649	9.97797	9.99229	9.99715	9.99863	9.99862	9.99732	9.99344
3	9.93649	9.97797	9.99229	9.99715	9.99863	9.99862	9.99732	9.99344	9.98267
4	9.97797	9.99229	9.99715	9.99863	9.99862	9.99733	9.99344	9.98267	9.95243
5	9.99229	9.99715	9.99863	9.99862	9.99733	9.99344	9.98267	9.95243	9.86629
6	9.99715	9.99863	9.99862	9.99733	9.99344	9.98267	9.95244	9.8663	9.61893
7	9.99863	9.99863	9.99733	9.99344	9.98268	9.95244	9.8663	9.61893	8.90591
8	9.99863	9.99733	9.99345	9.98268	9.95244	9.8663	9.61894	8.90591	6.84767

(1) State values via policy iteration



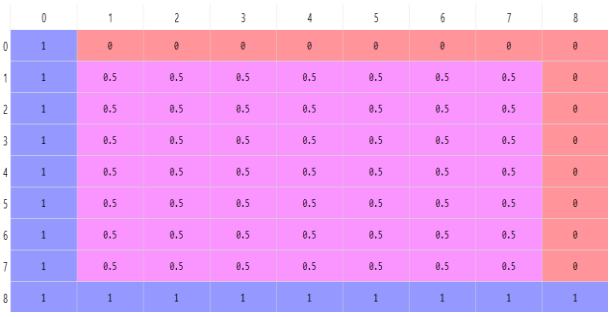
	0	1	2	3	4	5	6	7	8
0	8.46941	9.46941	9.81563	9.93549	9.97698	9.99133	9.99625	9.99783	9.998
1	9.46941	9.81563	9.93549	9.97698	9.99133	9.99625	9.99783	9.998	9.99699
2	9.81563	9.93549	9.97698	9.99133	9.99625	9.99783	9.998	9.99699	9.99352
3	9.93549	9.97698	9.99133	9.99625	9.99783	9.998	9.99699	9.99352	9.98332
4	9.97698	9.99133	9.99625	9.99783	9.998	9.99699	9.99352	9.98332	9.95377
5	9.99133	9.99625	9.99783	9.998	9.99699	9.99352	9.98332	9.95377	9.86838
6	9.99625	9.99783	9.998	9.99699	9.99352	9.98332	9.95377	9.86838	9.62171
7	9.99783	9.998	9.99699	9.99352	9.98332	9.95377	9.86838	9.62171	8.90921
8	9.998	9.99699	9.99352	9.98332	9.95377	9.86838	9.62171	8.90921	6.85123

(2) State values via value iteration



	0	1	2	3	4	5	6	7	8
0	0.5	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0	0
8	1	1	1	1	1	1	1	1	1

(3) $\pi_{policy\ iteration}$



	0	1	2	3	4	5	6	7	8
0	1	0	0	0	0	0	0	0	0
1	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0
2	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0
3	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0
4	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0
5	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0
6	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0
7	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0
8	1	1	1	1	1	1	1	1	1

(4) $\pi_{value\ iteration}$

Fig.7 Difference between value iteration and plicy iteration

As we can see, the value functions are very similar. The optimal policies have some differences in the purple squre, where $\pi_{policy\ iteration}$ will choose to service the second line and $\pi_{value\ iteration}$ will choose to service any one of the lines.

Conclusion

This project gives us a great opportunity to handle a classic DP problem. By framing the problem as an MDP, we learned how to modeling a concrete problem into an abstract representation. Coding the problem with both value iteration and policy iteration is also an interesting part, where we design the data structures and all kinds of functions. That requires us to have a close look to the computation process and the principle of DP. Given the results, we figure out that value iteration and policy iteration usually obtain similar value function and the same optimal policy. However, value iteration has a faster convergence, which is consistent with the contents of the book. All in all, given the perfect model of environment as a MDP, DP is a great method to search for optimal policies.