AI Project 1: Maze Task

Minghao Zhang*
2017011002
School of Software, Class 83

Zhongyu Qiu
2017011050
Department of Electrical Engineering, Class 73

1. INTRODUCTION

A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. For the grid world task, or maze task, it's so natural to consider it in this paradigm. In our project, we used two sufficient optimization algorithms, **Value Iteration and Policy Iteration** to solve our MDP in both deterministic and stochastic setting, and launched various experiment environments (with different reward settings, maps and moving rules) with the popular reinforcement learning environment toolkit it gym[1] to analyze in many views. Our implementation is clear to check and the result can be reproduced easily.

2. ALGORITHMS

Consider this task in an finite MDP framework. This finite MDP can be defined by:

- An finite set of states $s \in S$;
- An finite set of actions $a \in A$;
- A transition function $T(s, a, s')$ which means the probability that a from s leads to s', i.e., $P(s'|s, a)$;
- A reward function $R(s)$;
- A start state and a terminal state;

Here we consider each position as a state, each move step as an action, the moving rule in the maze as transition function, and we can define the reward function by ourselves: for example, we can get 1 if we achieve the goal position; otherwise 0.

Note that the maze task in this homework is a model-given

*two authors contributed equally to this project

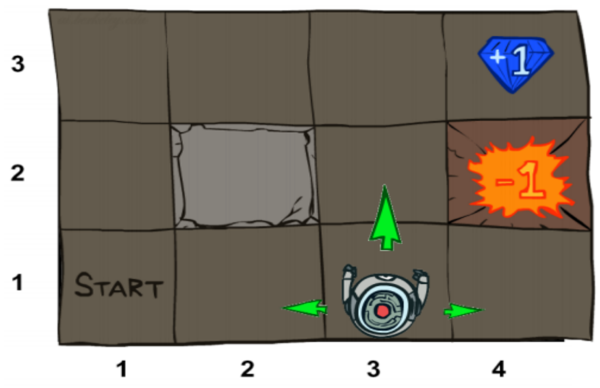


Figure 1: Consider the maze task as a MDP. We can design many reward settings.

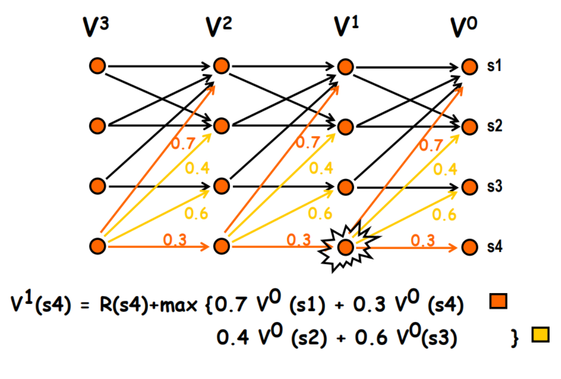
task, i.e., the transition function is already known. So we can develop some planning-based algorithms and some dynamics programmings algorithms to solve it without real experience in environment. In our project, we use two classical tabular reinforcement learning algorithms: value iteration and policy iteration, to find the optimal solution for this MDP. We use some empirical studies to analyze both of them.

2.1 Preliminary

To introduce our two algorithms, we first give some preliminary definitions.
Define a policy π as a stationary mapping from states to actions. Define the value function $V : S \to R$ associates value with each state. $V_\pi(s)$ denotes value of policy π at state s. Normally, we define the value function as a discount sum of rewards in an given length

$$V_\pi = \mathbb{E}\left[\sum_{t=0}^{H} \gamma^t R^t | \pi, s\right]$$

An optimal policy is one that is no worse than any other policy at any state. So the goal for a MDP is to compute or learn an optimal policy. Here we set the horizon H to be a large number to ensure that we can finish the task within the horizon length steps, then every discounted value function can rollout to the episode end. We can develop the following two methods to handle it. Totally, Policy iteration includes policy evaluation + policy improvement, and the two are repeated iteratively until policy converges. Value iteration



V3
V2
V1
V0
s1
s2
s3
s4
0.7
0.4
0.6
0.3
V1(s4) = R(s4)+max {0.7 V0 (s1) + 0.3 V0 (s4)
0.4 V0 (s2) + 0.6 V0(s3) }

Figure 2: Example: Value Iteration

includes finding optimal value function + one policy extraction. There is no repeat of the two because once the value function is optimal, then the policy out of it should also be optimal (i.e. converged).

2.2 Value Iteration

The value iteration algorithm can be summarized as two steps:

- Initialize an estimate for the value function arbitrarily,

$$\hat{V}(s) \leftarrow \ 0, \ \forall \ s \in S$$

- Repeatedly update the estimate according to Bellman optimality equation

$$\hat{V}(s) \leftarrow \ R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a)\hat{V}(s'), \ \forall \ s \in S$$

Intuitively, value iteration always use "argmax" greedy policy to choose the next state with the biggest value currently. According to *value iteration convergence theorem*[2], we know that it must converge to an optimal value $V*$. So value iteration can be considered as the "best" algorithm since it guarantee the solution of the MDP. However, value iteration has some tricky problems:

- Slow. For per iteration, it takes $O(S^2A)$
- The "max" at each state rarely changes
- The policy often converges long before the values

In practical, we found that value iteration always converges to a great result but really takes longer time than policy iteration method.

2.3 Policy Iteration

In this method, there are two problems that we will be interested in solving:

- Policy evaluation: given an MDP, a policy π and a horizon H, compute finite-horizon value function $V_\pi^k(s)$ for any $k \leq H$
- Policy optimization: given an MDP and a horizon H, compute the optimal finite-horizon policy, we will see this is equivalent to computing the optimal value function

The policy iteration algorithm can be summarized as two steps:

- Initialize policy π randomly
- Compute the value V^π of policy π
- Update policy π to be the greedy policy with respect to the V^π

$$\pi(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)V^\pi(s')$$

- If policy is changed in the last iteration, go to step 2

Although seemed very similar to the first one, this algorithm is different from it because its aim is always be the convergence of π. Since each iteration runs in polynomial time in the number of states and actions and there are at most $|A|^n$ policies and policy iteration never repeats a policy (due to the policy improvement theorem[2]), there is no polynomial bound on the number of policy iterations.

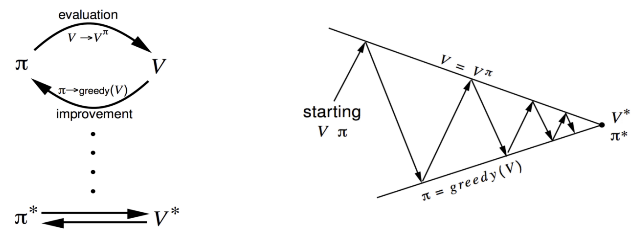


Figure 3: Example: Policy Iteration

2.4 Relation with Q-learning

Q function is a mapping from a tuple of s, a to a reward. $Q : S \times A \to R$. Due to the definition, the optimal Q function satisfies

$$V(s) = \max_{a'} Q(s,a')$$

which gives

$$Q(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a')$$

So consider the TD update of Q-learning

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

if we choose $\alpha = 1$, then it can be reduced to the same equation in value iteration method. So the value iteration method here is equivalent to the simplified Q-learning method. Besides, our value iteration algorithm is an off-policy algorithm, same as Q-learning, and policy iteration is an on-policy algorithm since the update step, or policy learning, always depends on the actions. So the value iteration will be more robust to the randomness of the policy and the transition function.

3. EXPERIMENTS

3.1 Environment

The experiment is performed in an maze consisting of multiple kinds of blocks. Initially, the maze starts with agent on the upper left block. The blocks are arranged in rectangular, each representing a different region.

- Block O: Plain block, agent gets nothing when stepping on this region.
- Block M: Mud bock, agent gets -0.01 reward when stepping on this region. This region is designed mainly for eliminating symmetry.
- Block S: Slippery block, when agent gets to this region, it does not stop and keep moving through it. No reward or punishment attached.
- Block W: Wall block, unreachable. Any attempt to get to this region will bounce agent one block back. For example, if the agent performs an action and go towards blocks "SSSW", this action will end up with agent standing on the third S block.
- Block T: Trap block, agent gets -1 reward and instantly stop the episode when stepping on this region.
- Block E: Target block, agent gets 1 reward and instantly stop the episode when stepping on this region.

We use a 7x7 map as training environment:

O	O	O	O	O	O	O
O	M	S	S	S	O	S
O	T	S	M	S	S	O
M	M	O	O	O	S	O
O	T	O	O	T	S	M
M	M	M	M	M	M	M
O	S	S	T	O	O	E

3.2 Value Iteration

After the maze was all set, we perform the value iteration. With the iteration ongoing:

$$\hat{V}(s) \leftarrow \; R(s) + \gamma \max_{a\in A} \sum_{s'\in S} P(s'|s,a)\hat{V}(s'), \; \forall \; s \in S$$

the value map converges to a fixed point, then we use this convergenced map to search the action choice for every block by greedy algorithm. Finally, we have the converged value map:

0.202	0.236	0.296	0.295	0.314	0.333	0.314
0.185	0.326	0.320	0.374	0.366	0.433	0.367
0.169	**-1.000**	0.255	0.297	0.344	0.486	0.302
-0.132	0.191	0.191	0.217	0.410	0.356	0.404
0.000	**-1.000**	0.173	0.225	**-1.000**	0.546	0.575
-0.024	0.000	-0.045	0.182	0.558	0.739	0.812
0.000	0.000	0.000	**-1.000**	0.673	0.812	**1.000**

And suggested actions:

→	→	→	→	↓	↓	←
↑	→	→	→	→	↓	←
↑	@	↑	→	↓	↓	↑
↑	→	→	↑	→	→	↓
←	@	→	↑	@	↓	↓
↓	↓	↑	→	→	↓	↓
←	↓	↓	@	→	→	@

By now, the value iteration is done. Time consumed is approximately 0.6-0.7s for 7x7 sized MDP.
As easily noticed, some marginal values remain zero after iterations, this phenonmenon is intentionally designed to prove a shortcome of MDP algorithm. Take the lower left zero values as instance. When encountering bounding problems(certain action causes the agent to go outside the map), the program was designed to assume agents stay in the same place, which is natural in actual cases. But when current block is surrounded by traps, "staying" instead of "moving" becomes the best choice. As a result, agent stay still and get no reward during iterations.There are solutions to this issue, which will be covered in later discussion.

3.3 Policy Iteration

Policy iteration is a more efficient way as a result of not using numerical iterations. We randomly choose the initial policy(in form of an action matrix corresponding to the map) and do the policy iteration. We choose some iterations to demonstrate the policy learning process.
Iteration1:

←	↑	↑	→	↑	↑	↑
↑	↑	↑	→	↑	↑	↑
←	@	↑	↑	↑	↑	↑
↑	→	↑	↑	↑	↑	→
↓	@	→	←	@	↑	↑
↓	→	↑	←	→	→	↓
←	←	↓	@	↑	→	@

Iteration 3:

←	←	→	→	↓	↓	↓
↑	→	→	→	→	↓	←
↑	@	↑	→	→	↓	↑
↑	→	→	↑	→	↓	↓
←	@	→	↑	@	↓	↓
↓	↓	↑	→	→	↓	↓
←	↓	↓	@	→	→	@

Convergence at Iteration 7:

→	→	→	→	↓	↓	←
↑	→	→	→	→	↓	←
↑	@	↑	→	↓	↓	↑
↑	→	→	↑	→	→	↓
←	@	→	↑	@	↓	↓
↓	→	↑	→	→	↓	↓
←	↓	↓	@	→	→	@

The optimization is obvious with iterations.
The "staying" problem still exists, because two iterating system shares an evaluating principle.

3.4 Code Implementation

We use Python 3.6 language, openai gym library to build a toy model, then create a whitebox environment with no external dependency.
vi_and_pi.py is the toy model, test.py is whitebox environment launcher. The toy model is simple and based on gym library and its mechanism is same as the whitebox environment, so in the report we mainly focus on the introduction of whitebox model.
We build class **Puzzle** as the representation of the maze. Three maps(2-dimentional list) are stored:

- **state_map**: state information for blocks in string form.
- **reward_map**: value information for blocks in float number form.
- **action_map**: suggested action for blocks in string form.(@ for Terminal states).

And an action set, which transform current state to next state in a **Puzzle** instance. Following are implements in a **Puzzle** instance.

- get_init_reward_map(): The reward map can be randomly initialized so long as terminal states are fixed, we use reward just for code simplicity.
- read_map(): Read map to the instance from file.
- check_valid_cord(): Check if given coordinates are out of map.
- action_reward(): Return reward of a certain action(We assume the reward as a function of (state, next_state) transition instead of (state, action) tuple).
- update_value(): Calulate the value map single time using Bellman equation.
- print_puzzle(): Print the value map.
- build_action_map(): Use argmax to choose an optimal action for each state. Show the action map as final result.
- policy_evaluation(): Calulate the value map under fixed given policy.
- policy_iteration(): Update policy using generated value map.
- print_policy(): Print current action map.

In test.py, value and policy iteration are run once seperately, the time consumption are recorded.

3.5 Results

On this 7x7 map, two different ways of iteration return same output:

→	→	→	→	↓	↓	←
↑	→	→	→	→	↓	←
↑	@	↑	→	↓	↓	↑
↑	→	→	↑	→	→	↓
←	@	→	↑	@	↓	↓
↓	→	↑	→	→	↓	↓
←	↓	↓	@	→	→	@

Time difference exists: value iteration takes 0.06s, policy iteration takes 0.015s.

3.6 Analysis

3.6.1 *Insight & Inspirations*

This simple AI agent is an incarnation of a basic principle: reward maximizing. We tried to control its move by intentionally designing maze map, still, the agent doesn't always work as our intuition.

O	O	O	O	O	O	O
O	M	S	S	S	O	S
O	T	S	M	S	S	O
M	M	O	O	O	S	O
O	T	O	O	T	S	M
M	M	M	M	M	M	M
O	S	S	T	O	O	E

The "memory" mechanism provides a better prediction of reward. We designed the second row of multiple slippery blocks, expecting the agent to take use of them. What we hadn't noticed is the trap block in the third row. As a result, the agent choose to avoid any contact with trap by not going to the second row at all. The action map does not suggest going downward until it is far enough from the trap block.
Another feature is too afraid of risk. When the agent has to pass near trap blocks(must take the chance of falling into them), the choice of agent is to delay. From human perspective, this behaviour is normal, because there is always a chance of map altering. But in MDP, this delaying causes serious problem, which will be covered later.

3.6.2 *Issues remaining*

In former part, the "staying" problem is introduced. Upon further consideration, the problem is not so terrifying as it sounds. When solving actual situations, such situation as surrounded by negatively rewarding states rarely exists. When action set size are far bigger(in project case it's 4), better option usually appears. Besides, There is solution guarantees agent to reach Terminal state from every possible initial state. We can use skills like exploration to build a less intelligent, more flexible agent. Specifically, the agent chooses from following strategies: Either do exploration(random walk) or using information to choose optimal action. The exploration slightly increase the risk of getting negatively rewarded, but solving the ”staying” problem. Another possible way is add punishment on ”staying” behaviour, if the current state is same as next state, the reward of this action will be punished.

This agent is build to solve Markov Decision Problems(MDPs), but MDPs are very small part of searching problem. Agent doesn’t always get to know the information about the whole map; the reward may not only depend on (state, next_state) transitions; it’s hard to directly compute a reward by AI agent itself. Those all are problems that MDP doesn’t cover.

4. CONCLUSION

In this project, we build an AI agent to solve grid formed MDPs. The algorithms we used are value iteration and policy iteration. Our agent solved the problem in both algorithms with proper visulization of iteration process and results. Also, we discusses some connections between our algorithms and Q-learning, providing agent’s possibility of future extending. Finally, Some insights and issues are mentioned, as a conclusion and revision of the whole Project.

5. ACKNOWLEDGE

We would like to give our sincere gratitude to Professor. Li and teaching assistants for offering us the opportunity and thinking of solving MDP ourselves. Many thanks to people offering help during our exploration to this problem.

6. REFERENCES

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.