On Markov Decision Process, Value Iteration and A*

Course assignment: Fundamental knowledge on Artificial Intelligence
University of Piraeus, Demokritos

Vasileios Papadopoulos
vassilispapadop@gmail.com

February 6, 2021

Abstract Reinforcement learning is the science of learning to make decisions. In order to make such decisions, the agent has to learn either a policy, a value function and/or a model. The importance of decision making comes to the affect it has on future expected rewards, agent states and perharps in environment states. In this article, we will discuss the mathematical formulation of agent-enviroment interaction, known as Markov Decision Process (MDP). We will present the Value iteration algorithm which helps us to calculate the value/utility of each possible state the agent can live. In the last part, we will use A* algorithm to find the best possible path in a small 3x4 grid world using value-iteration output as a heuristic function.

1 Environment

Assume a fully observable, non-deterministic 3x4 grid world. In such stochastic environment, each action an agent performs has a certain probability P(a) to succeed(to go as planned) and 1-P(a) to move to a different direction. For example: if the agent wants to go *up*, there is a 0.8 probability to actually move *up* and 0.2 probability to either move *left* or *right*. If the agent wants to move *right* it will succeed with 0.8 probability and with 0.2 it will either move *left* or it will stay and the same state if it hits the wall. Thus, the transition model can be summarized as a probability P(a) to move to the desired direction and 1-P(a) to land to state which is perpendicular of the itentended action. Figure (1) depicts 3x4 grid world and the transition model.

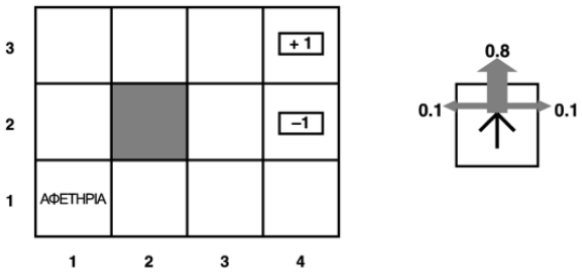

Figure 1: 3x4 grid world and transition model

In case there is a wall in the direction that the agents wishes to move, then it stays put. The agent is in constant feedback loop with the environment meaning it takes an action α in state s and lands in a stochastic manner in state s' and gets a reward r'.

The above desicion making problem can be formulated mathematically using Markov Decision Process which will discuss in next section.

2 Markov Decision Process (MDP)

Markov Decision process can formally describe the interaction between an agent and the environment. MDPs rely on so-called Markov property, which states that the future is independent of the past given the current/present state. Intuitively, markov property tells us that all information gathered from previous states can be consider as irrelevant as the new information of the current state is sufficient. We can formulate this definition as:

A state s has a Markov property if for all states

$\forall s' \in S$ and all rewards $r \in R$

$$p(R_{t+1} = r, S_{t+1} = s' | S_{t=s}) = p(R_{t+1} = r, S_{t+1} = s' | S_1 ... S_{t-1}, S_t) \quad (1)$$

for all possible histories $S_1 ... S_{t-1}, S_t$

In Markov decision process problems the goal is to find an optimal policy π^* that gives the best action for each state. Optimal policy π^* maximizes the expected sum of discounted(or not) rewards. The reward and discount factor together define what it means to be optimal in an MDP framework.

An MDP is defined by the following components:

1. Set of possible states: $S = \{s_0, s_1, ..., s_n\}$
2. Initial state: S_0
3. Set of possible actions: $A = \{a_1, a2, ..., a_m\}$
4. Transition model: $T(s, a, s')$
5. Reward function: $R(s)$
6. $\gamma \in [0, 1]$ is the discount factor

In its general form, the agent operates in a stochastic environment thus we model the non-determistic process with a transition model such as $T(s, a, s')$. However, in real world applications such a model is unknown, the agent is unaware of state transition probabilities or rewards and it could only discover them by taking an action on certain state and receive the reward of the landed state. This approach is called Q-Learning and it a model-free learning. The discount factor γ trades off the later rewards to earlier ones. Expected discounted reward is defined as:

$$\mathbb{E}[R|s, a] = \sum_r r \sum_{s'} p(r, s'|s, a) \quad (2)$$

while the expected utility of each state is:

$$U(s) = \mathbb{E}[\sum_i \gamma^i R(s_i)] \quad (3)$$

and the total utility is:

$$U([s_0, s_1, ...s_i, ...] = \sum_i \gamma^i R(s_i) \quad (4)$$

3 Value Iteration

Value iteration or Bellman update is a recursive dynamic programming algorithm. It is a method of computing the best values for each possible state an agent can be and eventually extracting the optimal policy π^* for an MDP problem. The agent takes an action α from state s and it lands in state s' with a probability as this given by transition model. Consider the grid world we have shown above. The reward of state $R(\langle 3, 4 \rangle) = 1$ while $R(\langle 2, 4 \rangle) = -1$. The Bellman update equation is given below.

$$V(s) \leftarrow \max_a (R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s')) \quad (5)$$

Intuitively, the value of V(s) is the best action that maximizes the expected reward. In order to compute the value state of cell(3,3) we need to calculate all the future rewards for every possible action $A = \{a_1, a2, ..., a_m\}$ and then choose the best action. In this example we set discount factor $\gamma = 0.9$ and living reward $R = 0$. Breaking down equation (5) we write:

1. Agent tries to go right:

$$V(\langle 3, 3 \rangle)_{right} = \sum_{s'} T(\langle 3, 3 \rangle, right, s')[R(\langle 3, 3 \rangle), \gamma V(s')] \quad (6)$$

$$V(\langle 3, 3 \rangle)_{right} = 0.9[0.8 * 1 + 0.1 * 0 + 0.1 * 0] \quad (7)$$

$$V(\langle 3, 3 \rangle)_{right} = 0.9[0.8*1+0.1*0+0.1*0] \quad (8)$$

$$V(\langle 3, 3 \rangle)_{right} = 0.72 \quad (9)$$

2. Agent tries to go left:

$$V(\langle 3, 3 \rangle)_{left} = \sum_{s'} T(\langle 3, 3 \rangle, left, s')[R(\langle 3, 3 \rangle), \gamma V(s')] \quad (10)$$

$$V(\langle 3, 3 \rangle)_{left} = 0.9[0.8 * 0 + 0.1 * 0 + 0.1 * 0] \quad (11)$$

$$V(\langle 3, 3 \rangle)_{left} = 0 \quad (12)$$

3. Agent tries to go down:

$$V(\langle 3,3\rangle)_{down} = \sum_{s'} T(\langle 3,3\rangle, down, s')[R(\langle 3,3\rangle), \gamma V(s')] \tag{13}$$

$$V(\langle 3,3\rangle)_{down} = 0.9[0.8 * 0 + 0.1 * 0 + 0.1 * 1] \tag{14}$$

$$V(\langle 3,3\rangle)_{down} = 0.09 \tag{15}$$

4. Agent tries to go up:

$$V(\langle 3,3\rangle)_{up} = \sum_{s'} T(\langle 3,3\rangle, up, s')[R(\langle 3,3\rangle), \gamma V(s')] \tag{16}$$

$$V(\langle 3,3\rangle)_{up} = 0.9[0.8 * 0 + 0 * 0 + 0.1 * 1] \tag{17}$$

$$V(\langle 3,3\rangle)_{up} = 0.09 \tag{18}$$

Then we take the action that maximizes the value state.

$$V(\langle 3,3\rangle) = \max_a [V(\langle 3,3\rangle)_{right}, V(\langle 3,3\rangle)_{left}, V(\langle 3,3\rangle)_{down}, V(\langle 3,3\rangle)_{up}] \tag{19}$$

$$V(\langle 3,3\rangle) = 0.72 \tag{20}$$

The above process is repeated for all states. After having calculated all values we repeat again until convergence which is guaranteed by Value iteration algorithm.

3.1 Pseudo code

Below we present the pseudo code of Value Iteration algorithm.

Initialize array V with zeros (V(s) = 0)
Repeat:
  $\delta \leftarrow 0$
  For each s $\in S$:
    $u \leftarrow V(s)$
    $V(s) \leftarrow \max(R(s,a) + \gamma \sum_{s'} T(s,a,s')V(s'))$
    $\delta \leftarrow \max \delta, |u - V(s)|$
Until $\delta < \theta$ (small positive number)

3.2 Discount factor

One important parameter for solving MDPs is the discount factor γ. Discount factors are important in MDPs, in a sense they determine how the reward is counted in future states. In this section, we will evaluate the impact discount factor γ has, to optimum value V_s^* of each state s and ultimately to optimal policy π^* after the convergece of value iteration algorithm. Particularly, given 3x4 world we defined in section I, we will explore the results for 3 different γ values, 0.9, 0.6, 0.2 after 100 iterations. Results presented in the table below.

$\gamma = 0.2$			
-0.045	-0.021	0.122	1.000
-0.049	0.000	-0.041	-1.000
-0.050	-0.050	-0.049	-0.050

$\gamma = 0.6$			
0.066	0.215	0.477	1.000
-0.009	0.000	0.137	-1.000
-0.050	-0.035	0.019	-0.085

$\gamma = 0.9$			
0.509	0.650	0.795	1.000
0.399	0.000	0.486	-1.000
0.296	0.254	0.345	0.130

Figure 2 shows the number of iterations required
Due to trivial problem being discussed in this article, there is no substantial differentiation in number of iterations among different γ, though there exists some distinction in behaviour which would be magnified in larger problems.

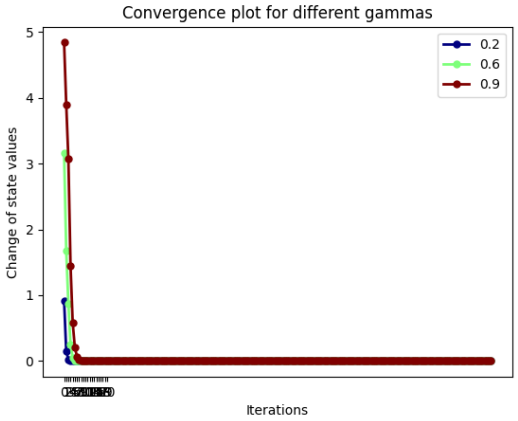

Figure 2: Convergence for $\gamma = 0.2, 0.6, 0.9$

4 A* Algorithm

A* is a graph traversal and path search algorithm. It differs from Dijkstra as it uses the best first search while taking into account the current cost g and an heuristic function h. It gives priority to nodes that are supposed to be better than others according to the value of function f(s) = g(s) + h(s) where h(s) is a heuristic function. For h(s) = 0, A* is similar to Dijkstra.

The heuristic function is a way to guide the algorithm about the direction it needs to take, in order to find the goal in fewer steps. Usually, the heuristic function emerges from solving the given problem with relaxations about the constraints. We can use any heuristic function for a certain problem as long as it **consistent** and **admissible**. Admissible means that the heuristic should never overestimate the actual cost(21), while consistent indicates that the heuristic of predecessor node should be less and/or equal to the sum of the cost and heuristic of successor node (22).

$$\forall n, h(n) \leq g^*(n) \tag{21}$$

$$h(s) \leq g(s,p) + h(p) \tag{22}$$

where:

- h is the consistent heuristic function
- p is any node in the graph
- s is any descendant of p
- g(s,p) is the cost from p to s

Having calculated the optimal values V^* for the 3x4 grid world, we will use them as heuristic to find the sortest path from position (1,1) to terminal state with higher reward (3,4). The output of the algorithm is shown below.

Path Python						
$\gamma = 0.9$	(2,0)	(1,0)	(0,0)	(0,1)	(0,2)	(0,3)

Notice: In Python implementation, grid world indexing starts from top-left. It is flipped horizontally. The stating position (1,1) becomes (0,0) , high reward state (3,4) is (0,3) and negative reward state (2,4) becomes (1,3). Thus the path in our case is:

Path						
$\gamma = 0.9$	(1,1)	(2,1)	(3,1)	(3,2)	(3,3)	(3,4)

In the last section, we present A^* pseudocode.

4.1 Pseudo code

Below we present the pseudo code of A^* .

```
openlist ← 0
closelist ← 0
openlist ← StartNode
While openList not empty:
    m = Node on top of openList with least f
    if m = goalNode:
        return
    openList.pop(m)
    closeList.push(m)
    foreach child in children(m):
        if child in closeList:
            continue
        cost = g(m) + distance(m, child)
        if child in openList and cost < g(child):
            openList.pop(child) //new path is better
        if child in closeList and cost < g(child):
            closeList.pop(child)
```

```
        if child not in (closeList and openList):
            openList.push(child)
            g(child) = cost
            //in our case state values
            h(child) = heuristic-function(child, goal)
            f(child) = g(child) + h(child)
return False
```

5 Technical Details

First, we define a grid with certain *width* and *height*, in our case 3X4. For terminal states we set the desired rewards and store the respected indices. We use the *array* function from *numpy*. The function *non-deterministic-move* returns the direction the agent will actual take based on intended action and *obey-prob* value. We use *random* function from *numpy* package. *obey-prob* can be thought as noise. *calculate-state-values* function initiates value iteration process. After setting all states to zero, we iterate to all non-terminal-states of the grid. For every non-terminal-state and for all possible actions we calculate the pair action, reward and return the action we higher expected utility and set to the considered state-index. We repeat the above process for either certain number of iterations or until the change in all state values becomes smaller than a certain threshold θ. The complete algorithm is repeated for different γ values. We also use *pandas* and *matplotlib* packages to plot the rate of change of the values during the process.

Similarily, for A^* we construct a grid and start the algorithm by passing *grid*, *start, end* nodes and *heuristic* arguments to *astar* method. As presented previously, astar repeats while *openList* contains a node. We find the node with higher f and for all possible moves we keep track of children nodes. For all children nodes, we calculate $f(n) = g(n) + h(n)$ and append them open list. Value iteration and A^* implementations can be found here: Value-Iteration and A-Star.

6 Conclusion

In this article we briefly presented Markov Decision Process problems and the challenge to find the optimal policy for each state an agent could be. We used Value Iteration algorithm to calculate the value of each state and explored the impact of discount factor γ. Lastly, we implemented A* path finding algorithm and used the converged state-values as heuristic. We also provided Python implemantions of the two considered algorithms. valueiteration.py and astar.py.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Fourth Edition 2020.

[2] Casey C.Bennett and Kris Hauser. *Artificial intelligence framework for simulating clinical decision-making: A Markov decision process approach*. Elsevier, January 2013.

[3] van Hasselt, H., Guez, A., and Silver, D. (2016). *Deep Reinforcement Learning with Double Q-Learning*. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1).

[4] FrantišekDuchoň, AndrejBabinec, MartinKajan, PeterBeňo, MartinFlorek, TomášFico, LadislavJurišica (2014). *Path Planning with Modified a Star Algorithm for a Mobile Robot*. Elsevier: Procedia Engineering, Volume 96, 2014, Pages 59-69.