Solving an MDP with Value Iteration

***Abstract*— MDPs or Markov Decision Processes are a common occurrence in artificial intelligence. It is used in a multitude of ways such as reinforcement learning, unsupervised learning, dynamic programming, and much more. MDPs have the interesting ability to transverse the environment based on the actions of the current state that in return affects all subsequent states. It can be used in video games, solve real-world problems, or in our case solve a simple 4 by 3 grid to reach its terminal states. In this paper, I will cover what an MDP is, what value iteration is, and how I coded these to accomplish reaching the goal and my results.**

Keywords— Markov Decision Process (MDP), Value Iteration, Rewards, States, Discount, Policy, Utilities

I. Introduction

In this project, we were tasked to solve an MDP or a Markov Decision Process using value iteration. The goal was to learn about MDPs and Value iteration and produce a practical code solution producing the utilities and policy output. A big difference between using MDPs with value iteration and the previous projects is that this can not be solved using linear algebra. So, this posed a new challenge however with value iteration this is solved easily. In an attempt to reduce the mistakes of my previous projects I am focusing on lowering variables used, using efficient data structures, and utilizing methods more to create sleeker methods.

II. Overview of MDPs and the Project

In this project, we are solving an MDP or Markov Decision Process which comes from Ronald Howard's 1960 book "Dynamic Programming and Markov Processes." MDPs are used in a large variety of fields such as artificial intelligence, robotics, manufacturing, and even economics. In the world of MDPs, they are used for sequential decision problems in a fully stochastic environment. Within the MDP they use a Markovian transition model and rewards for the model to allow the agent to transverse the environment. Within our project, we have eleven states with the initial state being represented as s0 and terminal states represented as +1 and -1. Along with states we have, actions such as up, down, left, and right, a transition model P(s | s, a), and a reward for going to a specific state. The expected output of a Markov Decision Process is a policy. A policy is denoted by π and represents the actions recommended by the policy for a specific state. You can measure the quality of your policy by the expected utility. The goal of the MDP is to find a sequence of actions that maximizes the likelihood of reaching the goal within the time constraint. Below is an overview of the environment:

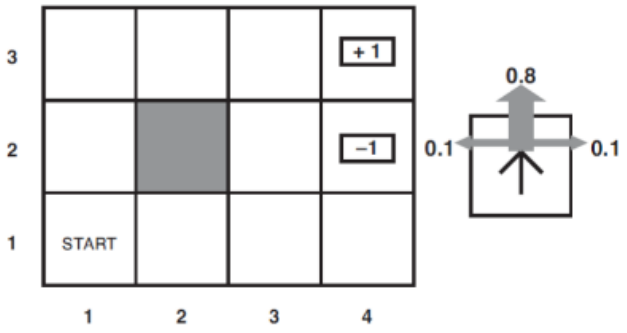

Figure 1: Example of project grid

The goal of the project is to solve the above pathing which starts at 1,1 (tile with the START) and find the optimal path to the terminal states given the discounts and rewards. The gray block is representing an obstacle that the agent will have to move around. The terminal states or goal states are +1 and -1 in the top right. The box on the right is a breakdown of how the agent will potentially act in the environment. Given the transition model, the agent has a 0.1 percent chance to right or left and a 0.8 percent chance to go up. Using value iteration we will be able to take in the discount, rewards, and transition model and find the optimal utility from the start to the terminals.

III. Overview of Value Iteration

Within this project, we are using value iteration to solve our MDP. Value iteration is different from anything else we have done so far. Until this point, our project used linear algebra to solve our environments, however, with MDP we can not do this. So, this is where value iteration comes in to save the day. Below is the basic pseudocode for value iteration:

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

Figure 2: Pseudocode of Value Iteration

As it can be seen above the inputs are the MDP itself with the states. As stated previously there are eleven states in total with two of those being the terminal states. Within the value iteration the reason this can not be solved with linear algebra is because of the max inside the Bellman equation which is below:

$$U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$$

Figure 3: Pseudocode of Bellman Equation

The importance of the Bellman equation is that it assigns a value to a decision at this state. It is stating that the U prime is the reward at the current state plus the discount for the sum of each action at the current state. It calculates the utility for each action at each state so it calculates the utility for up, left, down, and right and returns the action with the highest utility back.

Once the Bellman equation is figured out we will figure out if the delta is lower than the threshold which I calculated by taking 0.0001 times (1 - discount) divided by the discount. The delta is the absolute value of u_prime at this state minus U at this state. If the delta is lower than the threshold return the utilities up until this point. Value iteration is a great way to solve an MDP because of the inability to use linear algebra. It gives us the ability to understand which optimal action for each state to take for each state.

IV. Code Overview

In this section, I will break down the methods I used within this project, how they work together, and their purpose. My getExpectedUtility() method and the value iteration() method are where a large amount of the work is getting done and the subsequent methods call on these methods.

A. State Class Initialization and Basic Set up

Unlike the previous project there was not a large amount of initialization needed in the beginning of this project. There is the State class, the state list, the action list, and the transition model. The initialization of the class State takes in three parameters: self, x, and y. This is needed so we can get the x and y position of the state in the grid. This makes it easier to check if the agent is running into boundaries. The next basic setup needed for this project was to create the list of states and the actions. This was simply done by hard coding the states as State objects with their x and y coordinates which correspond to figure 1. An example of this is [State(1,1), State(1,2), …] till all eleven states are filled out. An important note is that the gray obstacle is excluded from the list. The action list is simply a list of the four actions allowed, [“U”, “R”, “D”, “L”]. This will allow us to cycle through all the available actions for each state to check which option is best based on the discount and reward inputs. Lastly, the transition model is the last thing needed which was given to us. This allows us to calculate the probability of where the agent will be moving in the environment.

B. valueIteration Method

This method is responsible for helping produce the optimal utilities for the MDP. It takes in three parameters: P which is the transition model, passed_in_rewards, and passed_in_discount. The overview of the logic of what value iteration does was presented above so I will go over more specifically how I coded then the logic of value iteration. This method starts off by setting delta to zero, creating our threshold value, and creating the numpy array for U and U_prime.

Now that we have all the items we need initalizaized we can begin the while loop. Within this loop I utilized a basic nested for loop. It starts off with i for the range of 11 to iterate through each state. We then check if we are at a terminal state; if so we break and move to the next state. If not, we enter the second for loop which is represented by J, that will loop four times. This loop is what allows us to select each action and figure out what the utilities for each. From here we enter the last for loop that is represented by K, which has a range of 11 to once again calculate each state. Within this last loop we can calculate the sum count for each state and action. This is so we can later ask for the max action.

Once this is done we can move to the bellman equation. Here we are figuring out what U_prime is by adding the reward, the discount and times it by the max action we just calculated. After this we check if our delta is lower than our threshold. If it fails the check will continue, if it passes it will break, print out the utilities to the console, and return the utilities. An example of the output of this method is below:

```
Utilities: [ 0.64768803  0.58799284  0.55945429  0.33740097  0.71620328  0.64132693
 -1.          0.7761173   0.84393372  0.90509583  1.        ]
```

Figure 4: Output of Value Iteration Method

C. getExpectedUtility Method

This method is responsible for assisting the program select the best action to take based on the utilities from value iteration, the transition model, the objects, and the location. This assists in doing so by looping through each of the four actions and for each state it will figure out the probability of moving to the next state. If this probability is better than we make that the chosen action. This continues until all actions for all nine states (we skipped the terminal states) are chosen. An example of the below output is below:

Figure 5: Output of Expected Utility Method

D. getValidStates Method

This method is responsible for returning a list of valid states the agent can use. It takes in two parameters: objects which is the list of states, and state which is the current state we are at. The method will check each of the four directions in the order of up, right, down, and left. It will check up by adding one to the y value. Right is checked by adding one to the x value. Down is checked by subtracting one from the value. Lastly left is checked by subtracting one from the x value. This method calls on the isNextStateValid() method to return if the above adjustments are valid. If they are valid

they are added to the list. Once the method has gone through all four directions, the list is returned.

E. isNextStateValid Method

The purpose of this method was to reduce redundant code for the getValidStates() method. It takes in three parameters: states, x_coord, and y_coord. This method has a very simple purpose which is to tell the getValidStates() method if the surrounding states are valid or not. Since getValidStates is simply checking each direction it is passing in an x and y value for each direction. If it is valid it returns the index of the state and if it is not valid it will return a -1. Since getValidStates immediately checks the output of this method it determines if it adds it to the list.

F. printPolicy method

This method is responsible for printing the policy to the console. This method takes in two parameters: utilities, and the transition model (P). This method will loop through each of the states. If we reach one of the terminal states we will continue and go to the next state. If we are not a terminal state it will call on the getValidStates() method to produce the valid surrounding states to our current state. This method returns a list of valid states that gets passed into the getExpectedUtility method. The getExpectedUtility method returns what action to take and then that action gets added to the policy list to make printing out easier. Once we have iterated through each state in the objects list, I inserted a '0' at position five to place the obstacle in the environment. After this, each line is printed starting with sections 8-12, 4-8, then from 0-4 to get the correct representation of the grid. An example of the output is below:

```
['R', 'R', 'R', 'T']
['U', '0', 'U', 'T']
['D', 'L', 'U', 'D']
```

Figure 6: Example output of printPolicy method

G. runMDP method

This is a simplistic method that I created because I disliked how messy the main method became and was redundant to copy and paste this code multiple times. This method takes in the transition model (P), the desired reward, and discount. This method will then print out the discount and reward value, call the value iteration method to get the utilities, then pass these utilities into the printPolicy() method that in returns prints the grid with the updated actions per state.

H. Main

This is where I call the runMDP method and input the three examples for the project. It requires three simple calls, assigns the discount and reward to your desired numbers, and calls runMPD() and passes in the transition model, the rewards, and the discount. The program will print out figure 7 to the console.

V. Results

Below is a screenshot of my results based on the three given examples:

```
Discount = 0.99  Reward = -0.04
Utilities: [ 0.64768803  0.58799284  0.55945429  0.33740097  0.71620328  0.64132693
 -1.          0.7761173   0.84393372  0.90509583  1.        ]
['R', 'R', 'R', 'T']
['U', '0', 'U', 'T']
['U', 'L', 'U', 'L']

################################################################################

Discount = 0.5  Reward = -0.04
Utilities: [-0.06651939 -0.05392804 -0.01984808 -0.07346227 -0.04256962  0.066275
 -1.          0.00821725  0.1254793   0.38243475  1.        ]
['R', 'R', 'R', 'T']
['U', '0', 'U', 'T']
['D', 'L', 'U', 'D']

################################################################################

Discount = 0.5  Reward = -0.25
Utilities: [-0.46875    -0.45185313 -0.39692991 -0.46962628 -0.44524    -0.25499591
 -1.         -0.37698025 -0.21367398  0.14447165  1.        ]
['L', 'R', 'R', 'T']
['D', '0', 'U', 'T']
['D', 'L', 'D', 'D']

################################################################################
```

Figure 7: Console output of program

An unexpected outcome was that the first grid is correct but the subsequent discount and reward combinations got progressively more incorrect. My utilities are very similar to the given examples that indicate that my value iteration is working correctly. So this points me in the direction of my issue is in either my expected utility or my policy printing method. It's an interesting issue because the first is correct. I believe my issue is within the checking for valid states because it is clearing going out of bound as we can see in the last two. Both representing down as a viable option to go in.

A result I wish to highlight is that my projects all semester have had run time issues. With each project getting smarter about my data structures and keeping in mind the run time of my program. This runs very quickly as it can be seen below.

```
[Done] exited with code=0 in 0.217 seconds
```

Figure 8: Execution speed of the program

VI. Conclusion

MDP was a very interesting project and came with a unique set of challenges compared to the other projects. It is was a different type of math involved in this project as stated earlier, our previous projects used linear algebra to solve and this project used value iteration. I can see why MDPs are a trademark of artificial intelligence and how difficult handling events that are affected by your current state choices can be.