Jake Traut

11/11/16

CSCI 3202 Assignment 7 Report

Purpose:

This assignment gave us a chance to work with MDPs (Markov Decision Processes) and understand the algorithms involved; specifically value iteration and policy iteration. Our task was to modify the code such that it works with the MDP problem of navigating a horse through a maze to reach an apple. This allowed us to learn more about the inner-workings of the algorithm, such that we can get it to work with our specific matrix and transition properties, as well as a modification in the possible actions that can be taken. We also had to find an effective way of interpreting the solutions given from the algorithms into readable states to actions and pulling out the optimal solution or best path to the apple.

Procedure:

To accomplish this assignment we had to modify the code to accept our specific MDP scenario and run a number of experiments on the program to evaluate the stability and flexibility of the algorithm and its solutions. The first changes I made to the code were small loops with print statements for better readability and comprehension of the output policy solutions. Printing the variable (storing the policy) results in a much too clustered and unorganized display in the terminal to effectively read, so I looped through the policy in order of x,y to produce a much more clear display of "State (x,y): Action (i,j)" starting with state (0,0), (0,1), (0,2), and so on… I then made a copy of the MDP such that I could experiment with the living-reward and gamma values of one of them and compare the solutions. Realizing I hadn't checked if these solutions were even valid, or were reaching the terminal state, I wrote another loop that prints out the "desired path" from the policy; starting from state (0, 0) and its action and following until it hitting a wall or reaching the terminal state. I could now check if solutions were indeed reaching the terminal state, and at this point they were not. With some debugging I found that the provided matrix doesn't exactly match the matrix in the assignment description so I fixed that, and then had to find a way to stop the horse (or policy telling it) from going into walls and stopping there. To do so I made a simple modification to the transition function T that checks if the next state (using vector_add) is in the MDP states or not, and if it is not then there is no chance of moving and it will stay in the same state, otherwise carry on as usual. Now with everything working properly, I could determine if a solution reaches a terminal state or not and further experiment with different values.

In experimenting with the living-reward, I tested a range of values down to four decimal places in negative and positive directions to find precisely at what values changes in the solutions occur. I started checking within the range of [-.5, .5] and tuned this range down to my four point precision findings of the range of values that produce the same optimal solution. In searching for living-reward values that don't have solutions that reach the terminal state, I started with a greater range exceeding

+/-1. I narrowed these values down to only 2nd decimal point precision as the magnitude has to be greater than 1 to cause these problems. Living reward certainly has a close tie to policy.

For the changes in gamma values, I noticed just how sensitive the algorithm is to this parameter. I started with setting gamma to .5 and then .99, but realized these changes were both much too great and began to drastically narrow down the range to numbers significantly closer to .9.

Modifying the code to support the ability of the horse jumping over a state was a fairly quick fix. The first step involved changing the orientations list, which is what the actlist is set to, to include jumping. The resulting orientations list: [(1, 0), (0, 1), (-1, 0), (0, -1), (2, 0), (0, 2), (-2, 0), (0, -2)]. Now with this updated list of possible actions I made another simple change to the transition function T that checks if an action is in a newly defined list called jumplist (including only the new actions) and if so the 50-50 probability of success is returned, and if the action is not in jumplist proceed to the original transition matrix.

Data:

The crucial data in this program is the matrix describing the environment for our problem that we transform to an MDP using the classes. In this matrix, the 0 values are seen as neutral living state with no gain or loss, while the -.5 values signify snakes in that state that will bite you for a loss, the -1 represents a treacherous mountain that must be crossed with a loss in utility, and the 50 is the reward for reaching the apple or goal state. All **None** values are walls in the environment that you cannot enter, but can go around or jump (if implementing the jumping action).

Another important piece of data in this program is the actions and how they are defined. There is an actlist that stores these possible actions, which is really just the orientations from the utils.py. These include (1, 0), (0, 1), (-1, 0) and (0, -1), signifying to move a unit right, up, left, and down, respectively. For the jumping modification, this list is changed to also include (2, 0), (0, 2), (-2, 0) and (0, -2).

Results:

1. Living Reward (Data without jumping)
    a. The path remains the same for values in the range of (-.0097, .3162).
    As the living reward gets more negative the path tends to head immediately up instead of to the right.
    When the living reward exceeds .3162 the path begins to change as it starts heading up after moving right across the bottom row.
    b. If the living reward is set to -1.29 or less then there is no solution found as the first move attempts to go left out of bounds.
    The highest the living reward can be and still reach the terminal state is 3.20, after exceeding this value the path enters a cycle as the policy shifts to gain utility from this reward and the barns rather than trying to reach the apple, such that it never attempts to enter a terminal state or invalid state.

2. Gamma (Data without jumping)

   The algorithm is undeniably sensitive to changes in gamma. With the default value being .9, the lowest value I could find that produces the same solution is .899. The highest gamma can be and still produce the optimal solution is .915.

   Other minimal changes in gamma will produce different paths to the goal state, and greater changes stop the program from finding a valid path to the goal, by either going off course or creating a policy that doesn't include the goal state.

3. Actions

   With the inclusion of the jumping action the policy becomes even better, or more efficient in reaching the goal state. The original desired path was (0, 0) -> (1, 0) -> (2, 0) -> (3, 0) -> (4, 0) -> (5, 0) -> (6, 0) -> (6, 1) -> (6, 2) -> (7, 2) -> (7, 3) -> (7, 4) -> (7, 5) -> (8, 5) -> (9, 5) -> (9, 6) -> Terminal State (9, 7).

   When the horse is able to jump, the desired path is simplified to:

   (0, 0) -> (2, 0) -> (4, 0) -> (6, 0) -> (7, 0) -> (7, 2) -> (7, 4) -> (9, 4) -> (9, 5) -> Terminal State (9, 7).

4. Transition probabilities for jumping

   Changing the transition probabilities does result in producing different policies, even for very minor changes like .51 and .49, but these changes in the overall policy still don't contribute to a change in the desired path to the apple. Although, a more significant change in the probabilities, like .75 and .25 for jumping and staying stationary, respectively, produce a new policy and well as path. Likewise, in the opposite direction, with a probability of .45 for a successful jump and .55 for no jump, then we already start seeing changes in both policy and path. It isn't very surprising that altering the probabilities in this direction has a more direct effect on the best path as the policy is now becoming more concerned about unsuccessful jumps.