Nikhil Nagraj Rao
PID: n3621940

Date: 30-Mar-15
Subject: CAP 6671

CAP 6671- Intelligent Systems - Robots Agents and Humans
Robotics Assignment

Background: A* Search and Q-Learning are popular methods for path planning. The aim of this assignment was to implement the A* Search and Q-learning Algorithm to the task of path planning in an agent in a discrete world. A deterministic and a stochastic environment were defined and the implemented planners were tested on two different arenas of varying complexity. This report presents the results and observations of the implementation and testing. The entire assignment was implemented on Matlab.

Arenas: Two 10x10 grid world arenas were defined and they are as shown below:

```
1 1 1 1 1 1 1 1 1 1
1 0 1 0 0 0 0 0 0 1
1 0 1 0 0 0 1 1 2 1
1 0 1 0 0 0 0 1 0 1
1 0 1 0 0 0 0 1 0 1
1 0 1 0 0 0 0 1 0 1
1 0 1 0 0 0 0 1 0 1
1 0 1 0 0 0 0 0 0 1
1 3 0 0 1 0 1 0 0 1
1 1 1 1 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 1 1
1 0 0 1 0 1 0 1 0 1
1 1 0 0 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1
1 1 0 1 0 1 0 0 0 1
1 0 0 1 0 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1
1 0 0 1 0 1 0 1 0 1
1 3 0 1 0 0 0 1 2 1
1 1 1 1 1 1 1 1 1 1
```

Figure 1: Left: Arena 1; Right: Arena 2

The arena is input to the program in the form of a text file. In the arena, '1's denote walls, '0's denote free space, '2' denotes the source, '3' denotes the destination.

Environments: A stochastic environment and a deterministic environment were developed. The robot had moves in the four directions 'Up', 'Down', 'Left' and 'Right'. No diagonal moves were allowed.
In the deterministic environment, the robot movement is error free and always moves to the intended position. This environment was tested on the A* Search as required by the assignment.

The stochastic environment was developed in view of the fact that the movement of the robot is not always as intended but is prone to errors in practice. In the stochastic environment, the following rules apply.

Occurrence Rate	Behavior
60% of the time in each run	The agent moves to its intended position
30% of the time in each run	The agent moves into possible positions other than the intended position. All the other possible moves are equally probable and the robot never moves onto a wall.
10% of the time in each run	The agent does not move and stays in the same position

Figure 2: Rules of the stochastic environment

A* Search: The A* search was implemented on the deterministic version of the environment and tested on both the arenas. The admissible heuristic h(x) was taken to be the Euclidean distance between the current position and the destination.

Below is a pseudo-code of the implementation:

```
Import Arena

// Initializations
initialize visited list (matrix) and parent matrix to zeroes
initialize the costs f, g, and h for each position to infinity
initialize the start position to the source grid

// A* Algorithm
while (current position is not the destination)
{
        1. initialize flags;
        2. increment the move count;
        3. update the current grid to closed in the visit matrix;
        4. point the parent list of the current grid's neighbors to the current grid;
        5. if already visited, evaluate the neighbor's cost matrix to see if moves from the current grid
           lead to smaller costs;
        6. traceback if no plausible neighbors to move were found;
        7. set the costs using the function setCostMatrix() (Pseudo code for this function is available
           below);
        8. evaluate move options to remove moves leading to an obstacle ;
        9. pick the move option which corresponds to the smallest cost as the next move;
        10. update the current grid to the next move;
}

// Visualization
{
        visualize the arena;
        visualize the path tracing back from the moves which yielded the lowest cost;
        visualize the f costs
}
```

```
function setcostMatrix()
{
        if (this is the source node)
        {
                set g value to zero;
                update the h value of the current grid using the
        Euclidean metric;
                calculate the f cost using f = g + h
        }
        else
        {
                update the g value of the current grid as the g
        value of its parent grid plus one
                update the h value of the current grid using the
        Euclidean metric;
                calculate the f cost using f = g + h
        }
        return the f matrix
}
```

Results – A*: The following figures show the g, h and f cost matrices for Arena 1 run of the A*Search

	A	B	C	D	E	F	G	H	I	J
1	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
2	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	1	Inf
3	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf
4	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	1	Inf
5	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	2	Inf
6	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	3	Inf
7	Inf	Inf	Inf	11	10	9	8	Inf	4	Inf
8	Inf	Inf	Inf	10	9	8	7	6	5	Inf
9	Inf	13	12	11	Inf	9	Inf	7	6	Inf
10	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
11										
12				G Costs – Arena-1						
13		Source								
14		Destination								
15		Wall								
16		Free Space								
17		Path								

Figure 3: G Costs for Arena 1

	A	B	C	D	E	F	G	H	I	J
1	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
2	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	9.899494937	Inf
3	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	9.219544457	Inf
4	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	8.602325267	Inf
5	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	8.062257748	Inf
6	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	7.615773106	Inf
7	Inf	Inf	Inf	2.828427125	3.605551275	4.472135955	5.385164807	Inf	7.280109889	Inf
8	Inf	Inf	Inf	2.236067977	3.16227766	4.123105626	5.099019514	6.08276253	7.071067812	Inf
9	Inf	0	1	2	Inf	4	Inf	6	7	Inf
10	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
11										
12				H Costs – Arena-1						
13		Source								
14		Destination								
15		Wall								
16		Free Space								
17		Path								

Figure 4: H costs for Arena 1

	A	B	C	D	E	F	G	H	I	J
1	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
2	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	10.89949494	Inf
3	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	9.219544457	Inf
4	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	9.602325267	Inf
5	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	10.06225775	Inf
6	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	10.61577311	Inf
7	Inf	Inf	Inf	13.82842712	13.60555128	13.47213595	13.38516481	Inf	11.28010989	Inf
8	Inf	Inf	Inf	12.23606798	12.16227766	12.12310563	12.09901951	12.08276253	12.07106781	Inf
9	Inf	13	13	13	Inf	13	Inf	13	13	Inf
10	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
11										
12				F Costs – Arena-1						
13		Source								
14		Destination								
15		Wall								
16		Free Space								
17		Path								

Figure 5: F Costs for Arena 1

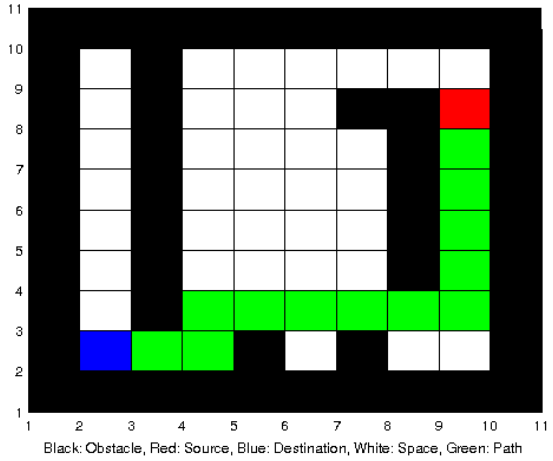

Figure 6: Path followed by the robot using A*

Q- Learning: The Q-Learning algorithm was implemented and tested for the stochastic version of the environment and tested with two different source nodes in Arena-1 described above.

Transition in Q-Learning: The transition function for the Q-Learning was as dictated by the stochastic environment discussed in the previous sections. The agent was able to move in the four directions- Up, Down, Left and Right. No diagonal moves were allowed. Each of the transitions put the agent into another state with a reward as summarized in the following section.

Naming of States: The states were assigned numbers as shown in the following example for a 4x4 grid.

S1	S5	S9	S13
S2	S6	S10	S14
S3	S7	S11	S15
S4	S8	S12	S16

Figure 7: State Nomenclature for an example grid

Reward values - Q-Learning: The rewards set for each of the actions taken by the agent are summarized in the table below:

Transition	Reward
Current Node into itself when current node is not the destination.	-1
Current Node to one of its 4 neighbors – Up, Down, Right, Left; when the neighbor nodes are neither wall positions nor the destination.	0
Current Node to itself when current node is destination.	100
Current Node to a destination node (destination node must be among the 4 neighbors of the current node).	100
Current Node to a node which is a wall (the wall node must be among the 4 neighbors of the current node)	-1

Figure 8: Rewards in Q-learning

Exploration Policy - Q-Learning: A greedy exploration policy was used in the implementation. Thus the agent always chose the action associated with the highest Q-Value. Thus the implementation was a purely based on exploitation.

Parameter Settings - Q-Learning: The following table summarizes the values used for the various parameters in the learning. Two sets of parameter values were used and the behavior of the algorithm with these changes was studied.

Parameter	Value
α	0.9
γ	0.5
No. of episodes	100
Maximum Steps per episode	150

Figure 9: Parameter Values Trial 1

Parameter	Value
α	0.001
γ	0.1
No. of episodes	180
Maximum Steps per episode	150

Figure 10: Parameter Values Trial 2

Q – Learning algorithm:

Function: Main

1. Import the arena
2. Generate the rewards matrix which is an nxn matrix, where 'n' is the total number of grids available in the arena. (For example: A 10x10 arena will have 100 grids. Therefore the reward matrix will be 100x100). The Rows in the reward matrix indicate the current state and the columns indicate the next state. The values defined by a row-column pair is the reward associated for an action taken when the agent is in a state defined by that particular row, thereby putting the agent into the state defined by that particular column. Rewards are generated as defined by Figure-8.
3. Q-Table is also an nxn matrix with similar representations by the rows and columns. The value associated by a row-column pair is the Q value associated for with the current state and next state. Learn the Q Table using the function learnQMatrix(). Save the learnt Q Table.
4. Load the saved Q-Table.
5. Get the source state
6. Repeat this step until the Goal State is reached. From the current source state, find the next state which is associated with the highest Q-value. Visualize the path using green rectangles.

Function: learnQMatrix()

1. Repeat the below steps as many times as the number of episodes
2. Generate a random initial state. Make sure the state does not land on the extreme corners of the arena where there are no plausible moves due to obstacles.
3. Repeat this step until the goal state has not been reached and the number of steps in this episode has not exceeded the maximum value.
    a. Increment Step Count.
    b. Find plausible next states by finding the rewards which are greater than or equal to zero. (This is to make sure that the agent does not stay in the current state)
    c. Pick a random state of the plausible next states.
    d. Evaluate the Q- Matrix of the next state's plausible neighbors and get the value of the maximum q value among them. Let the maximum Q value be 'qAdd'.

e. Update the Q Value using the equation:
Q(currState,nextState) = (1-α)*Q(currState,nextState) + α*(Reward(currState, nextState) + γ*qAdd).
f. Update the next state as dictated by the stochastic environment.

4. Return the learnt Q-Table.

Q-Learning Results:

Example Sequences generated by the Q-Table from 2 different sources

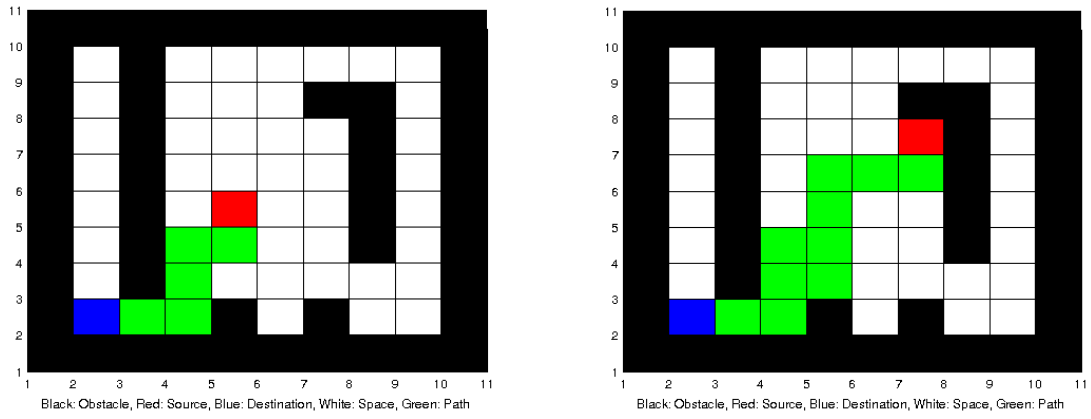

Figure 11: Example runs for Arena with src = (5, 5) - Left; src = (7, 7) – Right

Heat Map for an example arena of size 5x5

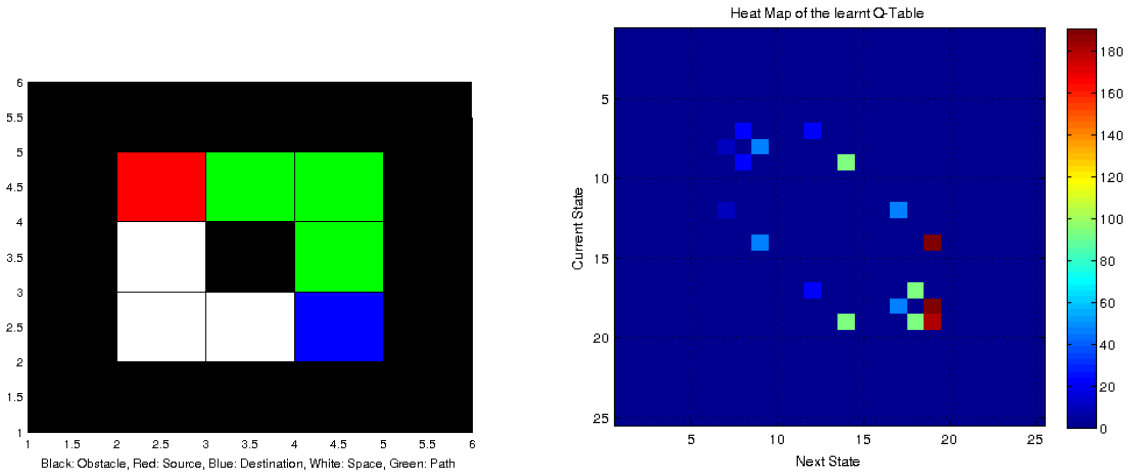

Figure 12: Heat Map of the learnt Q-Table for a simpler example arena. See section 'Naming of States' for state details.

Q Table Heat Map for Arena-1

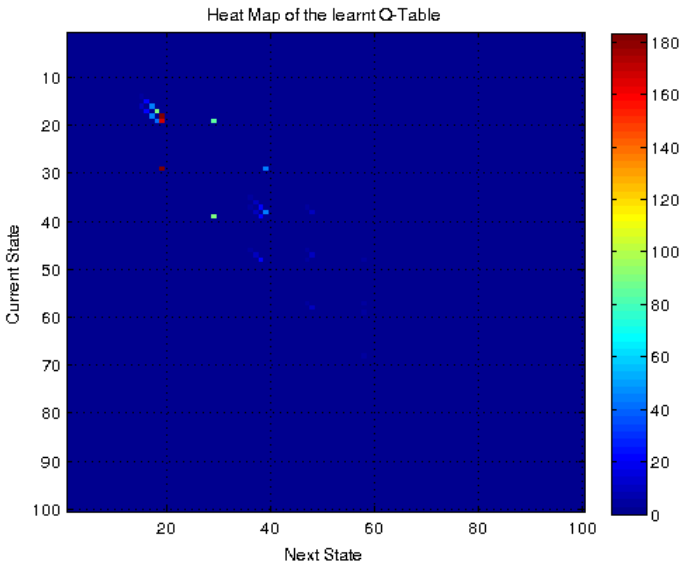

Figure 13: Heat Map for Arena 1

Discussion and Conclusion:

In the case of the Q-Learning algorithm, the parameters were varied according to Figure 8 and Figure 9. The value of the number of training episodes was set based on when the algorithm was able to find an optimal path in 2/3rd of the cases. As seen from the figures 8 and 9, lower values of α required more number of episodes for a satisfactory performance. The γ factor also had an effect in deciding the number of steps and was found that higher values of γ give better generalization of the algorithm. However, too high a value compromised on the algorithm's ability to find optimal paths during the training itself. The stochastic world visualization is seen in Figure 11 on the right file. We see an unintended movement of the agent forming a box like path pattern. However this is not seen on the left one.

Both the A* and Q-Learning Algorithms were able to find the optimal paths, with the Q-Learning requiring a little bit of tuning for performance. It will be of interest to study the tuning requirements of the Q-Learning in case of more complicated arena's involving more sophisticated actions such as picking up an object and delivering it to the destination. In this case, the working of both these algorithms need to be tweaked to find the optimal path to the pickup location first and then to the destination. A* Search might be more advantageous in this case due as extensive tuning would not be required.

Video Demonstration:

A video demonstration of the working of both of these algorithms is available on:

www.youtube.com/watch?v=mK2sR05ILKw