|  |
| --- |
|  |
| Assignment 2 |
| Robots, Agents, and Humans |
|  |
| **Jim Ihrig and Frank Bruno** |
| **3/28/2011** |

# Grid World

The 2d grid program given by the professor was used for the discrete Q-Learning and A\* path planning. The implementation of this world is in java, and therefore the Q-Learning and A\* implementation is java, as well. The 2d grid world program has many of its methods wrapped by our implementation to allow for the stochastic and deterministic aspects of the grid world required for Q-Learning and A\* path planning. The map shown in Figure 1 depicts the typical map used for learning and planning in this assignment.

## Q-Learning

Q-Learning is designed to be a reinforcement learning task. The goal of Q-Learning for this assignment is to learn the best path to the goal from any valid state. This is accomplished by running many episodes of training on the state space in question. A table is used to store values based on the actions taken. Higher values in the table results in the action it represents to be more preferable. During the running of the episode the Q-learning update rule is used to update the value of the action taken given the state that occurs as a result of the action. The episode completes when the goal is found or a maximum number of steps have been taken. When training is completed the optimal path to the goal should be found by following the more preferable actions represented in the table. Additionally, the likelihood of following the action has been reduced to allow for a more realistic environment where errors can occur.

### Design

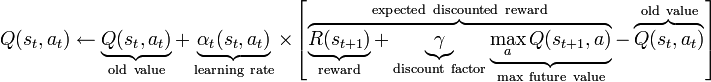
A two dimensional array is used to represent the 2d grid, which is filled with objects representing the state at each of the points in the grid. In addition, these states store the actions associated with this assignment, up, down, left, and right. This design allows for the object oriented properties of java to be leveraged. The reward for finding the goal is 100. There were no other rewards given for any other spaces and simply relies on the update rule to propagate the values to the other states in the search space.

### Exploration Policy

The exploration policy used for this assignment was of our own design. The idea is fifty percent of the time the action is randomly selected. The other fifty percent of the time the best path is taken. In the case where there was no best path then the action was randomly selected. This allowed for a sufficient exploration to take place. We also implemented a max exploration policy for demonstrating the best path after training has taken place.

### Rule Implementation

The typical update rule is used for this assignment. It is shown below.



### Final Q-Table

The learning rate, discount factor, and number of episodes used for this Q-Table are 150, 0.7, and 0.7 respectively. It is important to note that the maximum number of steps allowed per episode is 500 for this example shown in . Given that the number of times the episodes were cut short was 45 times.

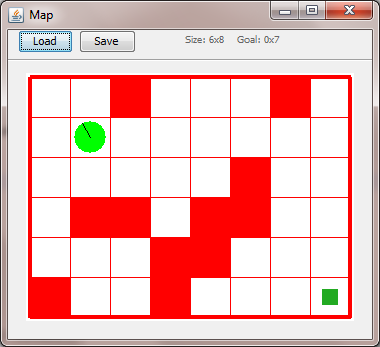


Figure 1 - Example Map used for Q-Learning

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Starting position | (4, 1) | Goal | (0,7) |  |
| Q | up | Down | left | right |
| (5, 0) | Outside Grid | 3.86 | Outside Grid | 3.86 |
| (5, 1) | Outside Grid | 5.52 | 2.7 | Wall |
| (5, 2) | Is Wall | Is Wall | Is Wall | Is Wall |
| (5, 3) | Outside Grid | 11.26 | Wall | 11.26 |
| (5, 4) | Outside Grid | 16.09 | 7.88 | 16.09 |
| (5, 5) | Outside Grid | 22.99 | 11.26 | Wall |
| (5, 6) | Is Wall | Is Wall | Is Wall | Is Wall |
| (5, 7) | Outside Grid | 46.96 | Wall | Outside Grid |
| (4, 0) | 2.7 | 2.7 | Outside Grid | 5.52 |
| (4, 1) | 3.86 | 3.86 | 3.86 | 7.88 |
| (4, 2) | Wall | 5.52 | 5.52 | 11.26 |
| (4, 3) | 7.88 | 7.88 | 7.88 | 16.09 |
| (4, 4) | 11.26 | 11.26 | 11.26 | 22.98 |
| (4, 5) | 16.1 | Wall | 16.09 | 32.87 |
| (4, 6) | Wall | 46.95 | 23 | 46.96 |
| (4, 7) | 32.87 | 67.09 | 32.87 | Outside Grid |
| (3, 0) | 3.86 | 1.89 | Outside Grid | 3.86 |
| (3, 1) | 5.52 | Wall | 2.7 | 5.52 |
| (3, 2) | 7.88 | Wall | 3.86 | 7.88 |
| (3, 3) | 11.26 | 5.52 | 5.52 | 11.26 |
| (3, 4) | 16.09 | Wall | 7.88 | Wall |
| (3, 5) | Is Wall | Is Wall | Is Wall | Is Wall |
| (3, 6) | 32.86 | 67.04 | Wall | 67.1 |
| (3, 7) | 46.96 | 95.86 | 46.96 | Outside Grid |
| (0, 0) | 2.7 | 1.32 | Outside Grid | Wall |
| (2, 1) | Is Wall | Is Wall | Is Wall | Is Wall |
| (2, 2) | Is Wall | Is Wall | Is Wall | Is Wall |
| (2, 3) | 7.88 | Wall | Wall | Wall |
| (2, 4) | Is Wall | Is Wall | Is Wall | Is Wall |
| (2, 5) | Is Wall | Is Wall | Is Wall | Is Wall |
| (2, 6) | 46.9 | 95.83 | Wall | 95.84 |
| (2, 7) | 67.09 | 136.94 | 67.06 | Outside Grid |
| (1, 0) | 1.89 | Wall | Outside Grid | 0.93 |
| (1, 1) | Wall | 0.65 | 1.32 | 0.65 |
| (1, 2) | Wall | 0.45 | 0.93 | Wall |
| (1, 3) | Is Wall | Is Wall | Is Wall | Is Wall |
| (1, 4) | Is Wall | Is Wall | Is Wall | Is Wall |
| (1, 5) | Wall | 95.79 | Wall | 95.85 |
| (1, 6) | 67.04 | 136.89 | 67.02 | 136.94 |
| (1, 7) | 95.85 | 195.67 | 95.83 | Outside Grid |
| (0, 0) | Is Wall | Is Wall | Is Wall | Is Wall |
| (0, 1) | 0.93 | Outside Grid | Wall | 0.45 |
| (0, 2) | 0.65 | Outside Grid | 0.65 | Wall |
| (0, 3) | Is Wall | Is Wall | Is Wall | Is Wall |
| (0, 4) | Wall | Wall | Wall | 95.79 |
| (0, 5) | 67.07 | Outside Grid | 67.03 | 136.9 |
| (0, 6) | 95.81 | Outside Grid | 95.67 | 195.65 |
| (0, 7) | 135.98 | Outside Grid | 136.69 | Outside Grid |

### Tests using different parameters

The tests were done using a variety of parameters. A noticeable pattern was that the smaller number of steps per episode the worse the agent did. This was largely due to the fact that the goal was never found and learning could never take place. Also, smaller values for learning rate and discount factor produced a q-table scaled down largely, and took much longer to learn the best routes. This could be a good thing when there are many different routes of approximately equal distance, since it prevents converging to a local optimum early.

### Action Sequences

There are several example action sequences shown on YouTube. The videos can be found at our Google code page <http://code.google.com/p/robotic-agent-path-planning-project/>.

## A\* Path Planning

### A\* (Pronounced "A Star") is an algorithm for searching for a path in an efficient way. It takes concepts from Dijkstra's shortest path algorithm and adds distance remaining to goal as an added heuristic. If we consider the distance traveled so far (Just as Dijkstra's does) and set that into a value named 'g' and the euclidian distance to the goal assuming no obsticles in the way to a value named 'h', then our heuristic can be defined as 'f' where:

### f = g + h

### or

### "A Star Heuristic" = "Distance traveled so far" + "Distance directly to goal from new position"

### Problems

### A\* requires a discrete environment, any continuous environment must be discretized into a set of waypoints and connections between them. In the continous A\*, because the target may not be at a waypoint, if a waypoint is almost halfway between agent and target, but closer to the target further from the agent, the agent may walk passed the target to the waypoint nearest to the target and then come back to the target. It would be possible to avoid this by checking if there is a clear path to the target, but then it could not be guarnteed that the agent does not brush against the wall enroute to the target.

### Pseudocode

currentPosition = startPosition

while(goalNotFound)

if validPosition(currentPosition.up)

and not visited(currentPosition.up)

and currentPosition.up.f == 0

then

currentPosition.up.previous = s

computeAStar(currentPosition.up)

priorityQueue.push(currentPosition.up)

elif validPosition(currentPosition.down)

and not visited(currentPosition.down)

and currentPosition.down.f == 0

then

currentPosition.down.previous = s

computeAStar(currentPosition.down)

priorityQueue.push(currentPosition.down)

elif validPosition(currentPosition.left)

and not visited(currentPosition.left)

and currentPosition.left.f == 0

then

currentPosition.left.previous = s

computeAStar(currentPosition.left)

priorityQueue.push(currentPosition.left)

elif validPosition(currentPosition.right)

and not visited(currentPosition.right)

and currentPosition.right.f == 0

then

currentPosition.right.previous = s

computeAStar(currentPosition.right)

priorityQueue.push(currentPosition.right)

end if

solution.add(goal)

while (currentPosition.previous)

solution.add(currentPosition.previous)

solution = solution.reverse()

### Values of H, G, and F

Values for (1, 5) -> f = 8.810249675906654, g = 1, h = 7.810249675906654

Values for (1, 3) -> f = 7.708203932499369, g = 1, h = 6.708203932499369

Values for (0, 4) -> f = 9.06225774829855, g = 1, h = 8.06225774829855

Values for (2, 4) -> f = 7.4031242374328485, g = 1, h = 6.4031242374328485

Values for (2, 3) -> f = 7.830951894845301, g = 2, h = 5.830951894845301

Values for (3, 4) -> f = 7.656854249492381, g = 2, h = 5.656854249492381

Values for (3, 5) -> f = 9.403124237432849, g = 3, h = 6.4031242374328485

Values for (3, 3) -> f = 8.0, g = 3, h = 5.0

Values for (4, 4) -> f = 8.0, g = 3, h = 5.0

Values for (0, 3) -> f = 9.615773105863909, g = 2, h = 7.615773105863909

Values for (4, 5) -> f = 9.8309518948453, g = 4, h = 5.830951894845301

Values for (4, 3) -> f = 8.242640687119284, g = 4, h = 4.242640687119285

Values for (5, 4) -> f = 8.47213595499958, g = 4, h = 4.47213595499958

Values for (3, 2) -> f = 8.47213595499958, g = 4, h = 4.47213595499958

Values for (5, 5) -> f = 10.385164807134505, g = 5, h = 5.385164807134504

Values for (6, 4) -> f = 9.123105625617661, g = 5, h = 4.123105625617661

Values for (0, 5) -> f = 10.602325267042627, g = 2, h = 8.602325267042627

Values for (6, 3) -> f = 9.16227766016838, g = 6, h = 3.1622776601683795

Values for (7, 4) -> f = 10.0, g = 6, h = 4.0

Values for (6, 2) -> f = 9.23606797749979, g = 7, h = 2.23606797749979

Values for (7, 3) -> f = 10.0, g = 7, h = 3.0

Values for (6, 1) -> f = 9.414213562373096, g = 8, h = 1.4142135623730951

Values for (7, 2) -> f = 10.0, g = 8, h = 2.0

Values for (6, 0) -> f = 10.0, g = 9, h = 1.0

Values for (5, 1) -> f = 11.23606797749979, g = 9, h = 2.23606797749979

Values for (7, 1) -> f = 10.0, g = 9, h = 1.0

Values for (0, 2) -> f = 10.280109889280517, g = 3, h = 7.280109889280518

Values for (5, 0) -> f = 12.0, g = 10, h = 2.0

Values for (7, 0) -> f = 10.0, g = 10, h = 0.0

# Conclusion

The goal of this assignment was to gain experience with reinforcement learning (Q-learning) and path planning algorithms (A\*) in discrete as well as continuous environments. Q-learning actually learns a space based on an instant reward for finding a goal. A\* can only be used if the space is searchable and the goal is known. For this reason, Q-learning is more advantageous technique for unknown spaces, but for searchable spaces A\* is optimal. Additionally, Q-learning only works for a static environment. A\* can be adapted to work for a moving target.