Shortest Path Dynamic Programming

1st Marcus Schaller
Department of Electrical and Computer Engineering (ECE 276B)
University of California San Diego
La Jolla, U.S.A
mschalle@ucsd.edu

***Abstract*—The Markov Decision Process is a very important tool used by a variety of AI and robotics. The Markov Decision Process is used to help a robot plan paths. Used in conjunction with dynamic programming it is even more powerful as it can generate a many number of paths from a given point. This paper will discuss how these techniques can be used to help an agent navigate through a simple maze therefore demonstrating how powerful these tools can be.**

I. Introduction

Dynamic programming is very common technique used in path planning. By using dynamic programming a robot can determine the optimal path from any location to its goal. It is very important to be able to generate paths from many different locations because it allows for robots to function successfully in a Stochastic environment. Perhaps the most common example of dynamic programming used in every day technology is our GPS or Google maps. Often a driver might miss their exit and by using dynamic programming the GPS is quickly able to determine that they missed there exit and recalculate based on the policy generated from dynamic programming. A much simpler example of this was done in this project and will be demonstrated in this paper

II. Problem Formulation

In order to run the dynamic programming algorithm to generate a policy for a given environment one must first understand the environment that was used in the project as well as formulate the Markov Decision Process to be used.

A. Problem Environment

The environment used in this project is a simple maze environment. The maze is broken into a grid with each square housing either a key, door, barrier, open space, or the agent. Figure 1 shows an example of what the environment can look like. By calling the info function the environment returns the starting position of the agent, the key location, the goal location, the door location, and whether or not the door is open or closed. In order to open the door the agent must navigate to a position adjacent to the key and face the key to pick it up, then navigate to the door to unlock it. The possible movements are move forward, turn left, turn right, pickup key, and unlock door. Each of these actions cost 1 and the goal is to navigate to the goal position in as few actions as possible.

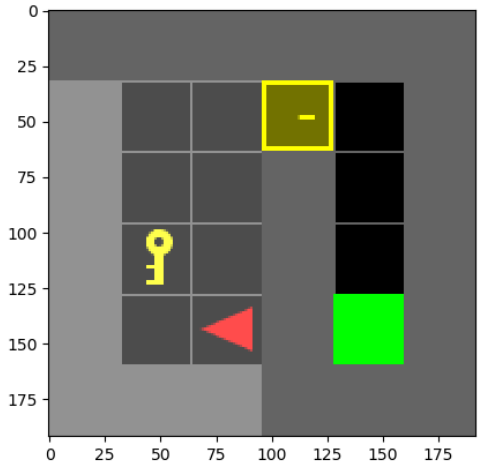

Fig. 1. Grid-Maze Environment

B. Markov Decision Process

In order to implement the dynamic programming algorithm the problem must be formulated as a Markov Decision Process (MDP). The MDP is defined by a tuple $(\mathcal{X}, x_0, \mathcal{U}, \ell, p_f, T, \mathrm{q})$. Each of these can be defined as the following; $\mathcal{X}$ is the state space or each open square in the maze. x_0 is the initial starting location of the agent. $\mathcal{U}$ is the control space or possible movements ℓ is a function specifying cost of applying control $\mathbf{u} \in \mathcal{U}$ in state $\mathbf{x} \in \mathcal{X}$. In the case of this problem +1 for each movement. q is the terminal cost which is infinite when not in the goal space and 0 when it is. p_f is the motion model or the movements required to navigate to state x. T is the time horizon which in the case of this problem will be equal to the sum of the open squares. Once this MDP is formulated dynamic programming can be used to compute the optimal control policy π^*. [1]

III. Technical Approach

This project is divided into 2 parts. Part A involves using different known mazes that required different control policies in order to solve. Part B involves using one map with changing, door, key, and goal states. The technical approach section will discuss the method used to solve each of these problems.

A. Known Map

Part A includes 7 different environments each of different sizes and layouts. Some include a door that is open therefore resulting in a quicker path if the agent does not grab the key. In order to calculate the ideal policy a form of the Dijkstra was used. However, rather than stopping at the goal state this modified algorithm continues to calculate all a policy for all T states. This was in order to help solve part B of the project which will be discussed later.

1) Shortest Path Algorithm: This algorithm began by initializing the initial agent square x_0 as cost 0, each open square state $x_i \in X$ as cost 1000, and each closed door or barrier as cost ∞. An empty matrix was initialized to contain an empty list for each open square x_i which would house the optimal control policy. Finally a matrix was initialized to hold the agent direction as a vector in each square and a Boolean matrix to signify whether or not a square had been visited. The starting square of each matrices would be assigned a value based on the agent's initial conditions. From there the algorithm loops through T times. First the cost to reach the position of the immediate neighbors of the agent are calculated and assigned to the cost matrix. If the square is directly in front, it would cost +1 as the agent only has to move forward. If it was to the side of the agent it would cost +2 as it had to turn then move forward. Behind the agent would cost +3 as it had to turn twice. The neighbor that cost the least would become the next position. The direction and movement required to reach that square would be added to the direction and policy matrix respectively. During the next iteration, the new direction (if the agent had to turn) and position was used to calculate the cost of the new neighbors. The new neighbors and the previous non-visited neighbors are compared to determine the next position. This would loop as it updated the policy, cost, visited, and direction matrix until either it reach T iterations or an error was reached signifying there were no more neighbors it could reach. This error occurred when there was a locked door blocking the path to the rest of the open squares signifying that the agent can not reach the goal state without using the key to unlock the door.

2) Key Path Vs. Direct Path: In many of the environments the agent could not reach the goal state without using the key to unlock the door. Since the algorithm updates the policy that describes the movements required to reach any position from a given starting point after the first run of the algorithm the path to reach the key square was returned. Using this array of movements the last move forward command is replaced with pickup key and the square that the agent is on before picking up the key is calculated using the direction vectors. Simply adding the direction vector to the position would return the previous neighbor position. This position was inputted into the shortest path algorithm and each of the matrices were reset. The algorithm would then find the optimal path to the door and unlock. The square in front of the door is updated as the starting position once again and finally the algorithm finds the optimal path to the goal position. After running the algorithm 3 times and returning the desired paths (starting position to key, key to door, door to goal position) these paths were concatenated in order to create the full path. If it was found that the goal position could be reached without unlocking the door, the cost of the key path and the cost of the direct path were compared the shortest path was returned as the optimal policy. This optimal policy was indeed optimal for nearly all of the known environments. The edge case that were not optimal will be discussed in the results section.

B. Random Map

In order to solve this part of the project the policy could not be recalculated between maps. In order to do this 6 policy matrices were calculated and stored to be used depending on the state of the map. The key states that were needed in order to determine which policy matrices were to be used are the door states (one open, booth open, or neither open) and the key locations. Depending on these states a policy would be selected and returned. For example, if both doors were closed as in figure 2 the door states and key location would be recognized and a policy would be returned to navigate the agent to the key and a policy to navigate the agent from the key to the goal was returned. The pickup key and unlock door was encoded into the policy to make sure the agent would complete those actions when required.

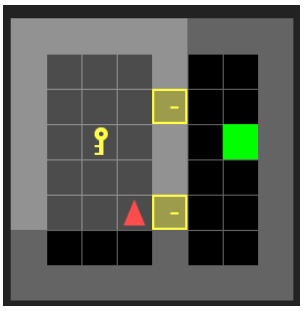

Fig. 2. Both doors locked

If both or one of the doors were open as in figure 3 the path cost directly to the goal as well as using the key then unlocking were compared and the path with the lowest cost was deemed to be optimal. However it was found that if the door was open it was always more efficient to navigate directly to the door.

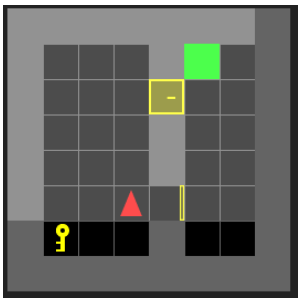

Fig. 3. Door Open

IV. Results

Each of the gifs for the paths were submitted with the code and the path cost is appended at the end of the gif name. For most of the maps the returned path was ideal.

A. Part A results

The chart of paths can be found below. The control inputs are represented as 0 = move forward, 1 = turn left, 2 = turn right, 3 = pickup key, 4 = unlock door

TABLE I

Map Name	Path	Cost	Optimal
5x5 normal	1, 3, 2, 4, 0, 0, 2, 0	8	Yes
6x6 direct	2, 2, 0, 0	4	Yes
6x6 normal	0, 2, 3, 0, 0, 0, 2, 0, 4, 0, 0, 2, 0, 0, 0	15	No
6x6 shortcut	3, 2, 2, 4, 0, 0	6	Yes
8x8 direct	1, 0, 0, 0	4	Yes
8x8 shortcut	0, 2, 3, 2, 0, 2, 0, 4, 0, 0	10	Yes
8x8 normal	1, 0, 2, 0, 0, 0, 2, 3, 2, 0, 0, 0, 0, 2, 4, 0, 0, 0, 2, 0, 0, 0, 0, 0	24	Yes

The more difficult paths to picture are illustrated below. Further illustration can be found in the gifs

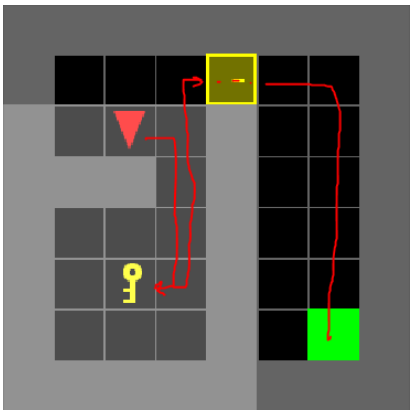

Fig. 4. 8x8 Normal

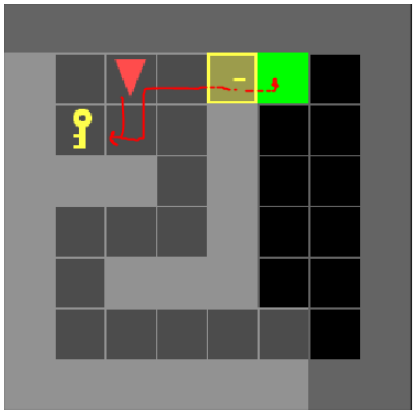

Fig. 5. 8x8 Shortcut

1) Known path improvements: The only path that was unable to create an optimal policy was the 6x6 normal. This is because by nature of the algorithm used the first square will be the square directly in front of the agent as it is the lowest cost square. A possible solution to solve this would be a way to determine if the key could be picked up on the way to the door and include that in the algorithm. The optimal path is included in the figure.

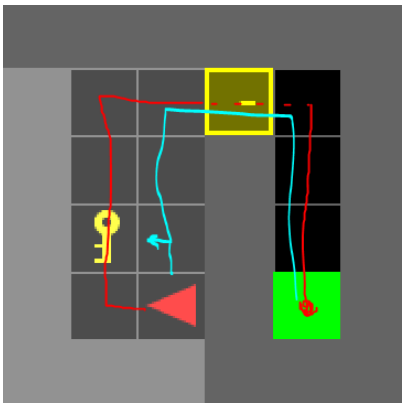

Fig. 6. 6x6 Normal

B. Unknown Map

It appears that all of the paths in the unknown section are indeed optimal. A couple of the paths are displayed below. All 36 of the paths can be found in the gifs folder along with the cost for the calculated path

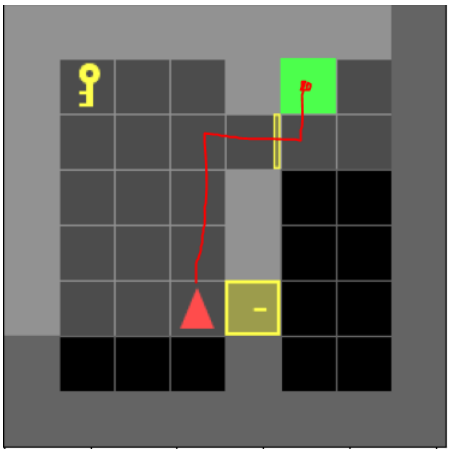

Fig. 7. Random Map, Cost 8

C. Algorithm performance

Overall the algorithm computed the correct path for the majority of the environments except for the one mentioned previously. The algorithm was very fast as it only had to loop through a maximum of 26 times for the larger random map. As mentioned before adding a heuristic to reward the policy that picks up the key while on the way to the door would be a useful improvement to the overall algorithm.

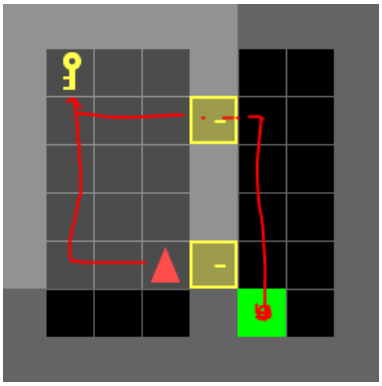

Fig. 8. Random Map, Cost 19

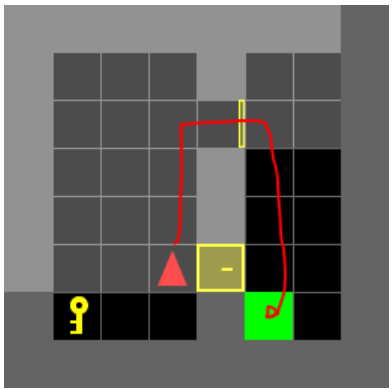

Fig. 9. Random Map, Cost 11

ACKNOWLEDGMENT

Thank you Professor Atanasov as well as Thai. I appreciate all the effort you put into making this project. I learned alot and had a great time solving this problem.

REFERENCES

[1] N. Atanasov, "Planning amp; Learning in Robotics Lecture 3: Markov Decision Processes," in ECE276B, 06-Apr-2021.

[2] N. Atanasov, "Lecture 4: The Dynamic Programming Algorithm," in ECE276B: Planning amp; Learning in Robotics, 08-Apr-2021.