ECE 276B Project 1: Dynamic Programming

1st Pranav Maddireddy
ECE 276B
University of California, San Diego
pmaddire@ucsd.edu

***Abstract*—This paper explores the implementation of Dynamic Programming for the door-key problem.**

I. Introduction

One of the most important concepts in Robotics is motion planning. Motion planning is the method that your robot uses to identify optimal paths and what control sequences will get them there. There are many algorithms that perform motion planning effectively, the method that will be discussed in this paper is known as *Dynamic Programming*. This method uses a backward planning strategy that starts from the desired goal position and iterates to the start position to find the optimal path and input sequence. This method doesn't just find the optimal path from start to goal, but actually finds the optimal path from every state to goal. This means that this algorithm calculates many paths and states that may be unnecessary and is inefficient in environments with many states. Despite these drawbacks, Dynamic Programming is very effective in finding optimal paths and can also be used to find an optimal policy for a general environment, as we will see later on in this paper.

II. Problem Statement

The Door-Key problem can be formulated into a *Markov Decision Process*.

A. Part a

In part a the state space $\mathcal{X}$ consists of all the grid cells when we do or don't have the key, the direction we are pointing, as well as when the door is open or closed:

$$|\mathcal{X}| = numberofcells * 4 * 2 * 2$$

The control space $\mathcal{U}$ is given by the possible control inputs: Move forward, turn left, turn right, pick up key, and unlock door. The initial position x_0 is given in each door-key problem. This is deterministic so there is no distribution required. The time horizon $\mathcal{T}$ is given by $|\mathcal{X}| - 1$ because an optimal policy will only go through each state at most one time so the longest potential path is given by the number of states. The motion model $\{(x, u)$ is deterministic in this problem so it follows exactly with inputs from the control space. If the input is to move forward, the state changes to the state one space forward in the current direction, left and right change the direction we our robot is facing, pick up key makes our robot attempt to pick up a key in the grid directly in front of it and unlock door does the same thing with a door in front of it. The stage cost function l(x,u) is given by:

$$l(x,u) = \begin{cases} 0 & \text{if u=TL or TR when state = Goal} \\ 2 & \text{if u=TL or TR and not at Goal} \\ 2 & \text{if u=MF and the next grid space is a valid empty space} \\ 1 & \text{if u = PK and the next grid has the key} \\ 1 & \text{if u=UD and the next grid has a Door that is closed and we have the key} \\ \infty & \text{otherwise} \end{cases} \tag{1}$$

The terminal cost q(x,u) is 0 when the robot is at the goal and ∞ otherwise. γ is set to 1.

B. Part b

In part b we expand our state space to include all possible key locations, goal locations, and the open/close state of another door. The key and the goal have 3 different positions and there are two doors so:

$$|\mathcal{X}| = numberofcells * 4 * 2 * 2 * 2 * 3 * 3$$

The control space $\mathcal{U}$, and initial position x_0 stay the same. The motion model and the loss function are largely the same as part a but now handle two doors instead of one. The time horizon is equal to the number of states minus 1 like in part a and the terminal cost is 0 at each possible goal and ∞ otherwise. γ is still set to 1.

III. Technical Approach

New that I have defined my Markov Decision problems let's move on to the technical approach using Dynamic Programming.

Input: MDP $(\mathcal{X}, \mathcal{U}, p_0, p_f, T, \ell, q, \gamma)$

Initialize $V_T(\mathbf{x}) = q(\mathbf{x})$ for all $\mathbf{x} \in \mathcal{X}$
for $t = T - 1$ down to 0 **do**
  $Q_t(\mathbf{x}, \mathbf{u}) = \ell(\mathbf{x}, \mathbf{u}) + \gamma \mathbb{E}_{\mathbf{x}' \sim p_f(\cdot|\mathbf{x},\mathbf{u})}[V_{t+1}(\mathbf{x}')]$ for all $\mathbf{x} \in \mathcal{X}, \mathbf{u} \in \mathcal{U}(\mathbf{x})$
  $V_t(\mathbf{x}) = \min_{\mathbf{u} \in \mathcal{U}(\mathbf{x})} Q_t(\mathbf{x}, \mathbf{u})$ for all $\mathbf{x} \in \mathcal{X}$
  $\pi_t(\mathbf{x}) = \arg\min_{\mathbf{u} \in \mathcal{U}(\mathbf{x})} Q_t(\mathbf{x}, \mathbf{u})$
end for

0: **Return** policy $\pi_{0:T-1}$ and value function V_0 =0

This system is deterministic, so we can drop the expectations and solve from there.

A. Part a

In part a I started by defining my time horizon as:

$$\mathcal{T} = (grid.shape[0] * grid.shape[0] * 4 * 2 * 2) - 1$$

Then I get the initial position, initial direction, and goal position from the environment data. Then I initialize my Value function and policy as :

$$Vt = np.ones((grid.shape[0], grid.shape[0], 4, 2, 2, T+1)) * np.inf$$
$$policy = np.full((grid.shape[0], grid.shape[0], 4, 2, 2, T+1), None, dtype = object)$$

This allows me to store values for both at every possible state and time step. Then I set the Value function for the goal state = 0 at the time horizon and set all other to ∞. This ensures that the optimal path will be ones that leads to the goal.

Then I prepared a set of nested for loops iterate in time backwards from one step before the time horizon and and goes through all possible state configurations. The last nested for loop in this chain iterates over the control space to find the optimal input. Then I identify valid states (i.e. states that aren't walls) and check the inputs. I do this by using my next state function (my motion model) that takes the current state and input that I'm checking and outputs what the next state would be. Then I input my state and input into my stage cost function and add it to the Value function evaluated at the next state that I calculated.

$$qcost = costtogo(grid, x, y, dirc, keybool, doorbool, u, Vt[newx, newy, newdirc, newkeyint, newdoorint, t+1])$$

My stage cost function return 2 for turning left and right, 2 for moving forward into a valid cell, 1 for picking up a key if it is in the next cell (if we don't have the key), 1 for unlocking a door if the door is in the next cell, we have the key and its closed, and infinity otherwise.

Then I compare my "cost to go" for each possible input and find the lowest one. The lowest cost to go corresponds with the optimal control policy so save the corresponding control to my policy variable and save the optimal cost to my value function. Then I extract the policy sequence by finding my initial state and using my motion model to iterate through the policy function and find the optimal sequence.

B. Part b

Part b follows a similar process to part a but with some important differences. Part b wants us to find a general policy for a set of random environments that are the same size, have 2 doors that may be open or closed, have 3 possible key locations and 3 different goal locations. To tackle this generalized policy problem, I decided to expand my state space to include the key position, state of the second door, and the goal position.

I started by initializing my new Time horizon:

$$T = (10 * 10 * 4 * 2 * 2 * 2 * 3 * 3) - 1$$

Then I initialized my policy and value functions similarly to part a:

$$Vt = np.ones((10, 10, 4, 2, 2, 2, 3, 3, T+1)) * np.inf$$
$$policy = np.full((10, 10, 4, 2, 2, 2, 3, 3, T+1), None, dtype = object)$$

Then I entered my nested for loops that iterate backwards from one timestep before the time horizon like in part a. This time, however, we have a for loop for the different positions of the key, different positions of the goal, and state of the second door as well as what we had in part a, all the grid cells, if we have the key or not, and the state of the first door.

When iterating through each input for each state I first checked if the states where valid (i.e not a wall) and if the inputs where valid(i.e would not take my robot out of the environment), then I followed the same steps as part a. I used my motion model to evaluate the value function at t+1 and added that to my stage cost for the current input u and state. Then I iterated through each input and found the one with the lowest cost. I saved the corresponding input in my policy function and cost in my Value function. Then I found the initial states for each random environments and used my motion model to iterate through my policy function to find the optimal policy sequence for whichever random environment I wanted. Using this method, I only have to run my Dynamic Programming method once to find the optimal paths for the entire set of random environments.

IV. Results

A. Part a: Known maps

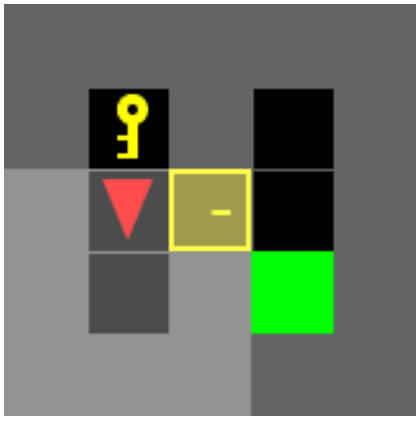

Fig. 1. doorkey-5X5-normal

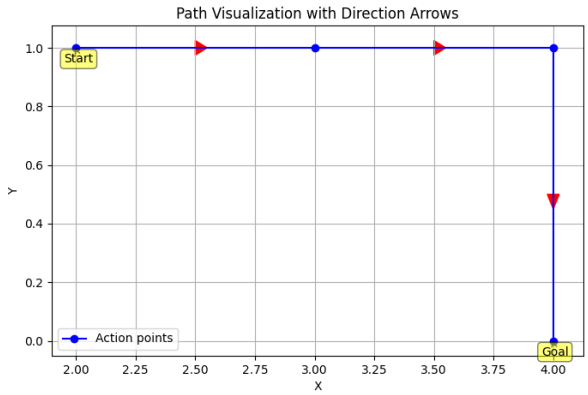

Fig. 2. Optimal path for doorkey-5X5-normal
Optimal path is given by sequence: ['TL', 'TL', 'PK', 'TR', 'UD', 'MF', 'MF', 'TR', 'MF']

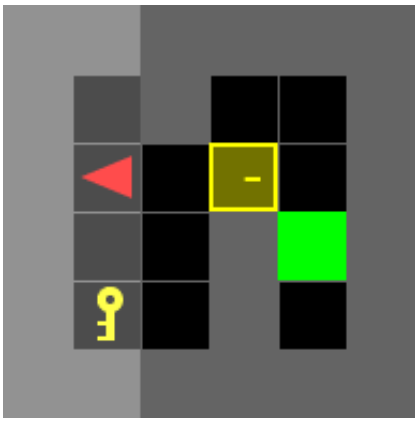

Fig. 3. doorkey-6X6-normal

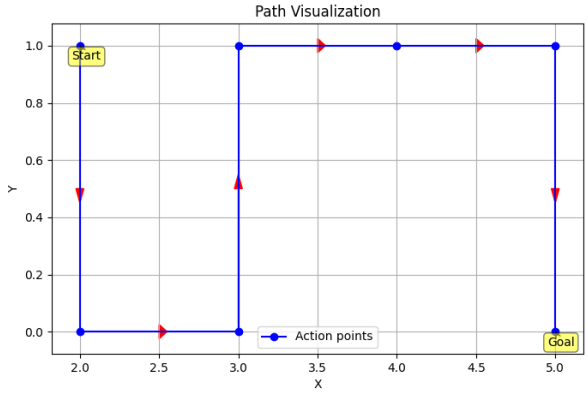

Fig. 4. Optimal path for doorkey-6X6-normal
Optimal path is given by sequence: ['TL', 'MF', 'PK', 'TL', 'MF', 'TL', 'MF', 'TR', 'UD', 'MF', 'MF', 'TR', 'MF']

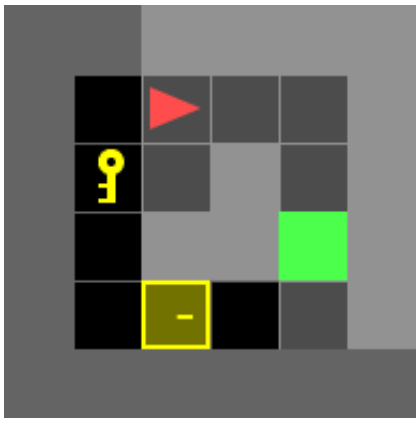

Fig. 5. doorkey-6X6-direct

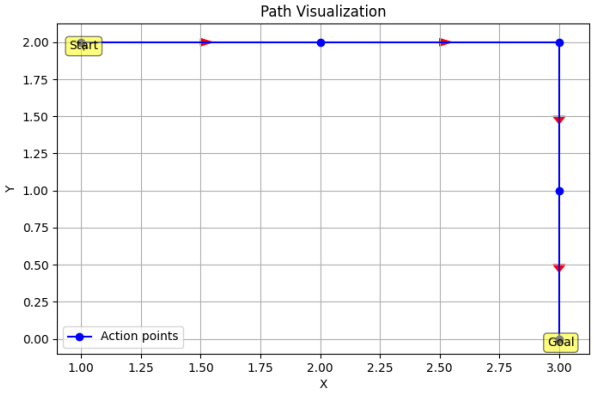

Fig. 6. Optimal path for doorkey-6X6-direct
Optimal path is given by sequence: ['MF', 'MF', 'TR', 'MF', 'MF']

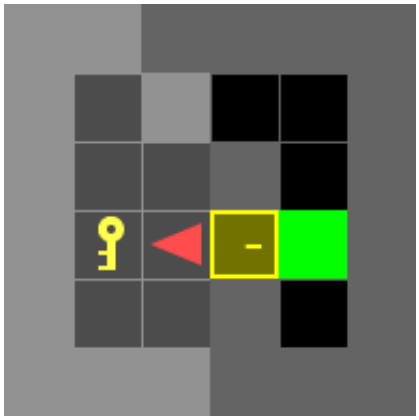

Fig. 7. doorkey-6X6-shortcut

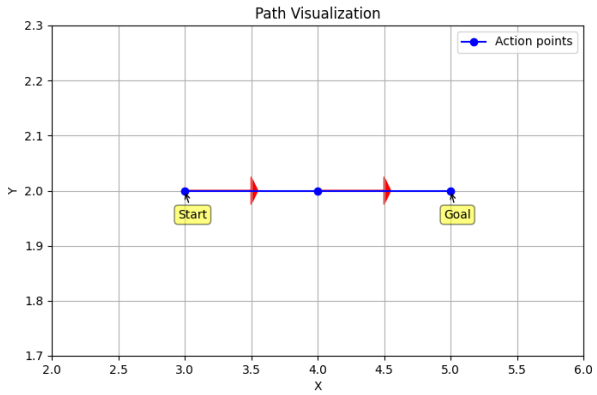

Fig. 8. Optimal path for doorkey-6X6-shortcut
Optimal path is given by sequence: ['PK', 'TL', 'TL', 'UD', 'MF', 'MF']

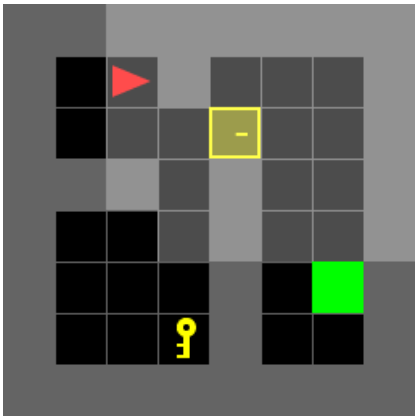

Fig. 9. doorkey-8X8-normal

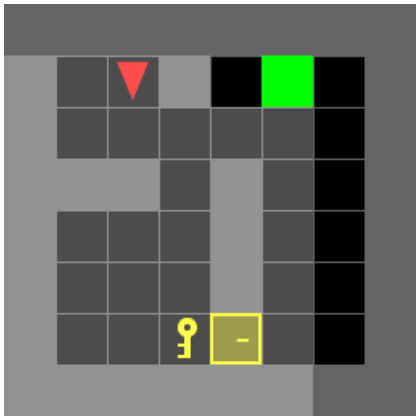

Fig. 11. doorkey-8X8-direct

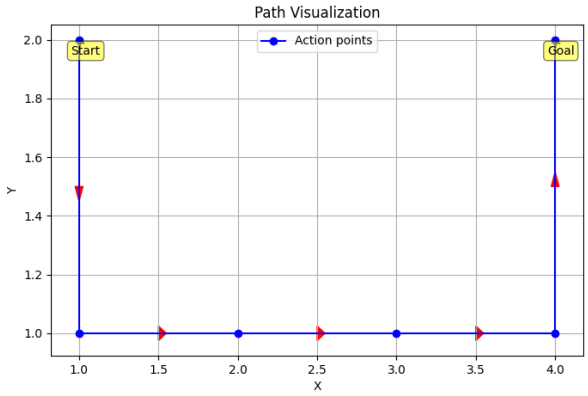

Fig. 12. Optimal path for doorkey-8X8-direct
Optimal path is given by sequence:['MF', 'TL', 'MF', 'MF', 'MF', 'TL', 'MF']

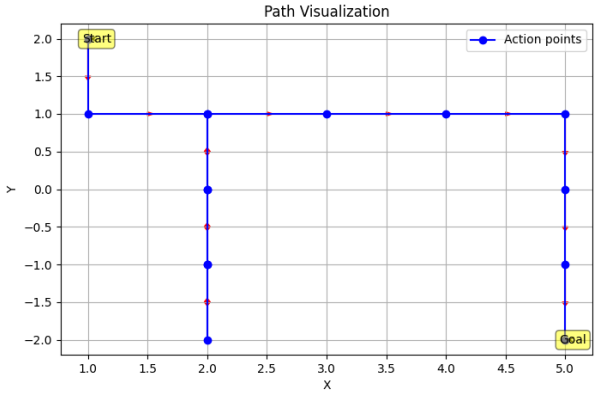

Fig. 10. Optimal path for doorkey-8X8-normal
Optimal path is given by sequence: ['TR', 'MF', 'TL', 'MF', 'TR', 'MF', 'MF', 'MF', 'PK', 'TL', 'TL', 'MF', 'MF', 'MF', 'TR', 'UD', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']

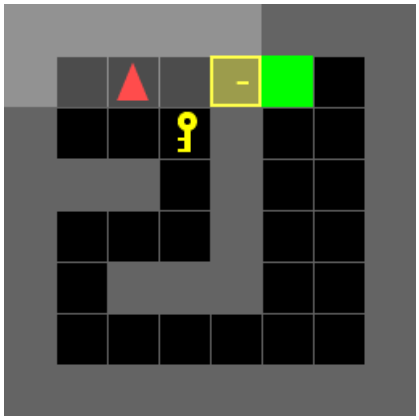

Fig. 13. doorkey-8X8-shortcut

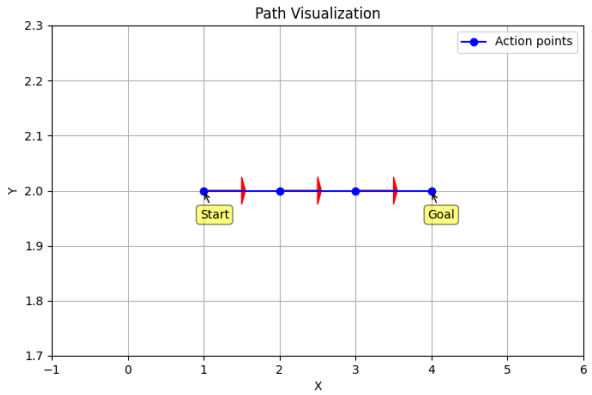

Fig. 14. Optimal path for doorkey-8X8-shortcut
Optimal path is given by sequence: ['TR', 'MF', 'TR', 'PK', 'TL', 'UD', 'MF', 'MF']

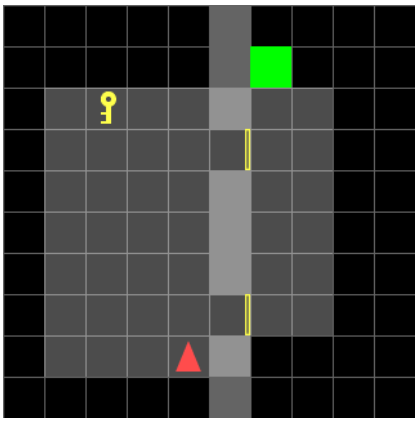

Fig. 15. Doorkey-10X10-1

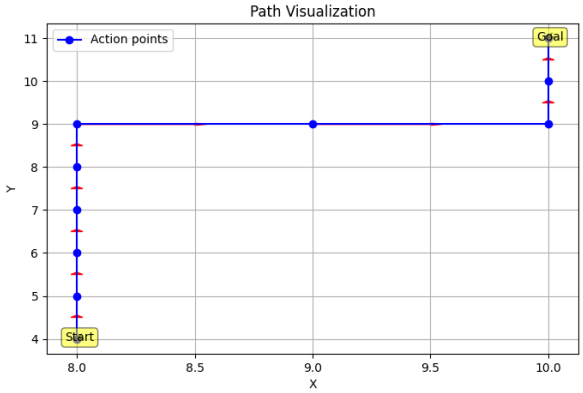

Fig. 16. Optimal path for Doorkey-10X10-1
Optimal path is given by sequence:['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF']

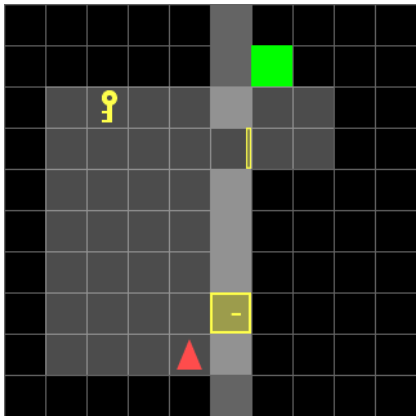

Fig. 17. Doorkey-10X10-2

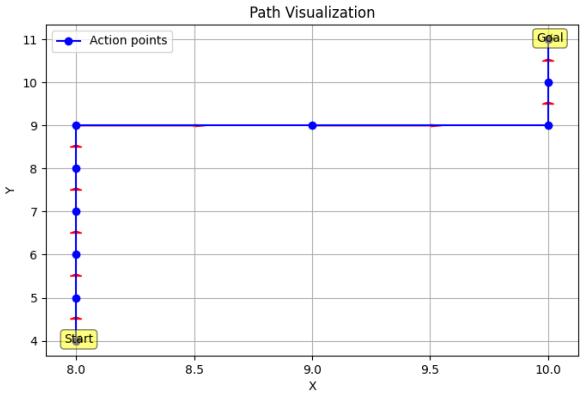

Fig. 18. Optimal path for Doorkey-10X10-2
Optimal path is given by sequence:['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF']

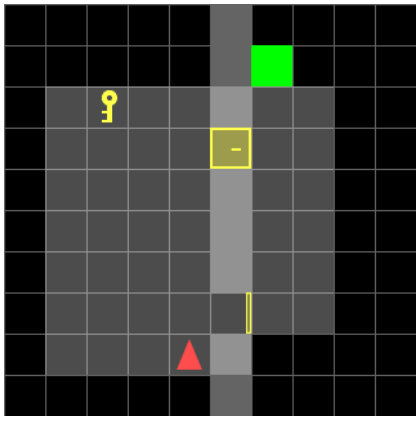

Fig. 19. Doorkey-10X10-3

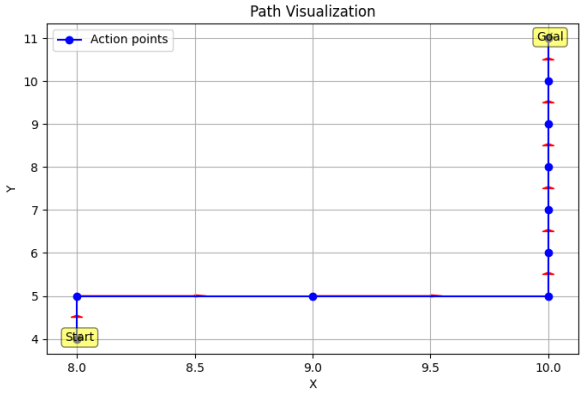

Fig. 20. Optimal path for Doorkey-10X10-3
Optimal path is given by sequence:['MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF', 'MF', 'MF', 'MF']

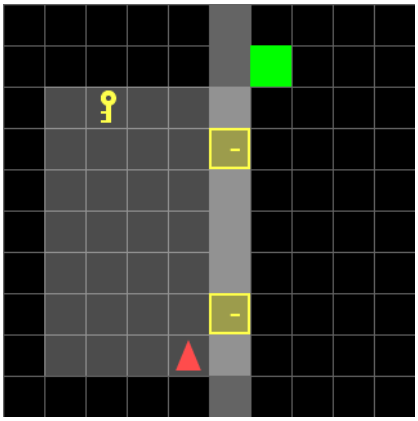

Fig. 21. Doorkey-10X10-4

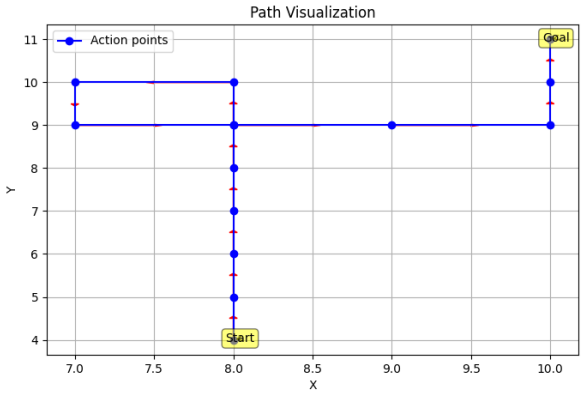

Fig. 22. Optimal path for Doorkey-10X10-4
Optimal path is given by sequence:['MF', 'MF', 'MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', 'TL', 'MF', 'TL', 'MF', 'UD', 'MF', 'MF', 'TL', 'MF', 'MF']

TABLE I
OPTIMAL CONTROL SEQUENCES

Map	Optimal control sequence
Doorkey-10X10-1	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF']
Doorkey-10X10-2	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF']
Doorkey-10X10-3	['MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF', 'MF', 'MF', 'MF']
Doorkey-10X10-4	['MF', 'MF', 'MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', 'TL', 'MF', 'TL', 'MF', 'UD', 'MF', 'MF', 'TL', 'MF', 'MF']
Doorkey-10X10-5	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
Doorkey-10X10-6	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
Doorkey-10X10-7	['MF', 'TR', 'MF', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF', 'MF']
Doorkey-10X10-8	['MF', 'MF', 'MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', 'TL', 'MF', 'TL', 'MF', 'UD', 'MF', 'MF', 'MF']
Doorkey-10X10-9	['MF', 'TR', 'MF', 'MF', 'TL', 'MF']
Doorkey-10X10-10	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
Doorkey-10X10-11	['MF', 'TR', 'MF', 'MF', 'TL', 'MF']
Doorkey-10X10-12	['MF', 'MF', 'MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', 'TL', 'MF', 'TL', 'MF', 'UD', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
Doorkey-10X10-13	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF']
Doorkey-10X10-14	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF']
Doorkey-10X10-15	['MF', 'TR', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF', 'MF', 'MF', 'MF']
Doorkey-10X10-16	['MF', 'MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', ' TL', 'TL', 'MF', 'UD', 'MF', 'MF', 'TL', 'MF', 'MF']
Doorkey-10X10-17	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
Doorkey-10X10-18	['MF', 'MF', 'MF', 'MF', 'MF', 'TR', 'MF', 'MF', 'MF']
Doorkey-10X10-19	['MF', 'TR', 'MF', 'MF', 'MF', 'TL', 'MF', 'MF', 'MF', 'MF']
Doorkey-10X10-20	['MF', 'MF', 'MF', 'MF', 'MF', 'TL', 'MF', 'PK', 'TL', 'TL', 'MF', 'UD', 'MF', 'MF', 'MF']

B. *Part b Random Maps*

Table I summarizes the results.

C. *Discussion*

For part a, my algorithm performed very well and was able to run quite fast. The biggest challenge I had with part a was understanding how to consider the key and the door. Once I figured out that I could include them as parts of the state it became a simple problem. The "known maps" part of this project when pretty smoothly for me and I think my algorithm performs well. For part B I had similar concerns with the different key and goal positions as well as the extra door. Similarly to part a, I was able to get past my issues by including them as part of my state. This however, created a very large state space which made my Time horizon value very large. This made the algorithm take a long time to run, through testing I noticed that I would never actually need that many time steps to compute my optimal policies for all map configurations. Once I brought T down to 30 part b ran fast and didn't loose any accuracy. In both problems I had some issues with my stage loss function missing certain state and input combinations, I was able to solve these issues with minimal difficulty. Overall I think my algorithms in parts a and b do a great job at finding optimal paths with reasonable runtime.