Dynamic Programming

Dhruv Talwar
Electrical and Computer Engineering
ECE 276B

I. Introduction

Path planning and finding the optimal control policy fr a robot or agent has been a fundamental problem in robotics that has been used in robotics for several decades. Optimal control policies refer to the strategies that a robot uses to make decisions about its actions based on its current state and the desired goal. on the other hand path planning refers to the process of finding a path from a starting location to a goal location while avoiding obstacles in the environment.

These optimal policies aim to minimize some cost function, such as the energy(battery life) consumed by the robot or the time taken to complete the task. Optimal control policies are essential for autonomous robots that need to perform tasks in a resource-efficient manner, such as mobile robots used for environmental monitoring, inspection, or surveillance. Path planning algorithms are essential for autonomous robots that need to navigate through complex environments to perform tasks such as delivering packages, exploring unknown terrains, and inspecting facilities. Path planning algorithms can also be used in industrial automation, where robots need to move efficiently between different workstations in a factory.

This paper tackles the problem described involves designing and implementing a Dynamic Programming algorithm for autonomous navigation in a Door, Key and goal environment. The objective is to get an agent from a starting location to a goal location, while taking into account the possibility of encountering a door that blocks the way. If the door is closed, the agent needs to pick up a key to unlock the door. The agent has five actions to choose from, with each action incurring a positive cost. The goal is to minimize the cost of reaching the goal, and the problem is tackled in two scenarios: ”Known Map” and ”Random Map”.

In the ”Known Map” scenario, different control policies are computed for each of the 7 environments provided in the starter code, while in the ”Random Map” scenario, a single control policy that can be used for any of the 36 random 8x8 environments is computed. The problem has potential applications in the field of robotics, as it addresses the important issue of autonomous navigation in complex environments where obstacles can impede progress towards a goal.

The results of the agent trajectory, construction of the optimal control policy and the formulation of the dynamic programming algorithm map are presented and evaluated in this paper. The effectiveness of the proposed approach in optimal control policy calculation is demonstrated.

II. Problem Formulation

The aim of this project is to implement dynamic programming and compute the optimal trajectory in both cases, part A and part B it is crucial to first formulate and understand the fundamental principles of dynamic programming.

A. Dynamic Programming Formulation

Before formulating a dynamic programming algorithm, certain key components need to be identified, such as the state of the system, the possible actions that can be taken in each state, the transition probabilities, and the cost associated with each action.
The following list shows all the necessary components required needed to solve for the optimal con troll policy. Additionally, the problem should satisfy the principle of optimality, meaning that the optimal policy for a given sub problem is also the optimal policy for the overall problem. Therefore, it is important to break the problem down into sub problems that can be solved independently. Once all of these components are identified, the dynamic programming algorithm can be formulated.

- $t \in 0, \ldots, T$ discrete time
- $x \in X$ discrete/continuous state
- $u \in U$ discrete/continuous control
- $p_0(x)$ prior probability density function defined on X
- $p_f(x_0 \mid x_t, u_t)$ where p_f is a conditional probability density function defined on X for given $x_t \in X$ and $u_t \in U$ (matrices P^u with elements $P^u_{ij} = p_f(j|x_t = i, u_t = u)$ in the finite-dimensional case).
- T is the Time Horizon, can be infinite or finite
- $l(x, u)$ stage cost of choosing control u in state x
- $q(x)$ terminal cost at state x
- $\pi_t(x)$ control policy: function from state x at time t to control u

- $V_t^\pi(x)$ value function: expected cumulative cost of starting at state x at time t and acting according to π
- $\pi_t^*(x)$ optimal control policy
- $V_t^*(x)$ optimal value function

Other than this we also have to formulate a motion model according to the given con straits in out environment. The agent cannot go into a cell occupied by the wall, neither it can go into a cell occupied by the door unless it is open, go occupy the key cell unless the key is picked up. Also to open the door and to pick the key, the agent must be facing the door/key.

We also define the Time horizon for our problem, in our case we use a finite horizon time horizon meaning that the agent has to reach the goal point in a specific defined time. We also have an additional parameter, the discount factor γ , which is used to balance the importance of immediate and future rewards. It is a value between 0 and 1 that determines the extent to which future rewards/cost are considered in the decision-making process. A higher discount factor values long-term rewards/cost more and a lower discount factor values immediate rewards/cost more. The discount factor is typically used to calculate the expected future rewards/cost of a state-action pair and is included in the calculation of the value function and the optimal policy. In our case we assume the problem follows Markov assumptions and thus we formulate a Markov Decision Process as above.

The finite-horizon optimal control problem in an MDP $(X, U, p_0, p_f, T, \lambda, q, \gamma)$ with initial state x at time t is:

$$\begin{aligned}\min_{\pi_{t:T-1}} V_t^\pi(x_{t0}) :=& \mathbb{E}_{x_{t:T}}\left[\gamma^{T-t}q(x_T) + \sum_{\tau=t}^{T-1}\gamma^{\tau-t}l(x_\tau, \pi_\tau(x_\tau))\right]\\ \text{s.t. } & x_t = x_{t0}\\ & x_{\tau+1} \sim p_f(\cdot|x_\tau, \pi_\tau(x_\tau)), \quad \tau = t, \dots, T-1\\ & x_\tau \in X, \quad \pi_\tau(x_\tau) \in U, \quad \tau = t, \dots, T-1\end{aligned}$$

We have to formulate our components of the problem in such a way that when we solve the above equation with them we get the optimal control policy. Now that we have defined all the required terms needed we shall now formulate them as per the python code

B. Problem formulation for part A- known environment

In this part we have 7 known environments of different sizes. The salient feature is that each has only 1 door and 1 key and the location and status of both is known. The goal and start location along with the orientation of the agent is known.

State Space
We know we have the x, y, $orientation$, $keystatus$ and $doorstatus$ therefore I formulated the state space as the possible combinations of [x,y,orientation,key stat, door stat]

- where x can go from 0 to the width of the map -1
- where y can go from 0 to the height of the map -1
- where orientation can be 0 for right, 1 for down, 2 for left and 3 for up
- key stat can be 0 if the agent does not have the key and 1 if it has
- door stat can be 0 if the door is locked and 1 if it is unlocked

So initially the state space had width-1*height-1*4*2*2, but i removed the cells where the x and y coincided with the walls. Thus reducing the state space and optimizing the calculations. To reduce the possible state spaces I removed all those where the x and y coincided with the walls in the map.

Therefore any possible state of the environment can be accurately be captured by this array.

Time Horizon The time horizon I defined was one minus the carnality of the length of the state space. The length of elements present in the state space changed with the map, so depending on the map, wall locations the time horizon has been defined

Motion Model I have defined the following steps in the motion model

if the action is "MF" move forward the first element is the x coordinate, the second element is the y coordinate the 3rd element is the orientation the fourth element is the key status and the 5th element is door status. If the agent is facing right (orientation 0), the first element of the state (which represents the agent's x-coordinate) is incremented by 1, as long as it doesn't exceed the height of the environment minus 1 (i.e., the agent doesn't move off the top edge of the grid).

If the agent is facing down (orientation 1), the second element of the state (which represents the agent's y-coordinate) is incremented by 1, as long as it doesn't exceed the width of the environment minus 1 (i.e., the agent doesn't move off the right edge of the grid).

If the agent is facing left (orientation 2), the first element of the state is decremented by 1, as long as it doesn't go below 0 (i.e., the agent doesn't move off the bottom edge of the grid).

If the agent is facing up (orientation 3), the second element of the state is decremented by 1, as long as it doesn't go below 0 (i.e., the agent doesn't move off the left edge of the grid).

If there is a wall in front of the agent, the "MF" action will have no effect. Similarly, if the door is locked and the cell in front has a door, the agent's position will not change. Similarly, if the next cell is a key and the key has not been picked up, the agent will not move into that cell unless the key is picked up.

If we use "TR" or "TL" we only have to update the orientation of the agent. if the action is "TR"

- agent direction initially 0(right) change to 1(down)

- agent direction initially 1(down) change to 2(left)
- agent direction initially 2(left) change to 3(up)
- agent direction initially 3(up) change to 0(right)

similarly if we use "TL"

- agent direction initially 0(right) change to 3(up)
- agent direction initially 1(down) change to 0(right)
- agent direction initially 2(left) change to 1(down)
- agent direction initially 3(up) change to 2(left)

If the action is "PK" and the next cell in the direction of the agent has the key and the agent does not have the key then the agent will pick up the key and the key status will change from 0 to 1, indicating that the agent has picked up the key.

Similarly if the action is "UD" and the next cell in the direction of the agent is a door and the agent has the key, aka the key status is 1, then only the door will be unlocked and the status of the door will change to 1, else it shall remain 0 only.

Thus this motion model cover all cases in the environment and for each case and scenario I have defined what the state will be post the motion model. The basic psudo algorithm for the motion model is as follows

```
[H] Action $a$, State $s$, Environment $env$ New state $s'$

$a$ is "MF"
$s' \leftarrow$ move forward according to $s$ and $env$;
$a$ is "TR" or $a$ is "TL"
$s' \leftarrow$ update orientation according to $a$ and $s$;
$a$ is "PK"
$s' \leftarrow$ pick up key according to $s$ and $env$;
$a$ is "UD"
$s' \leftarrow$ unlock door according to $s$ and $env$;
$s'$;
```

Pseudo code for the Motion Model

Stage cost The stage cost is defined for the edge cases, favorable and unfavourable. As we want the agent to reach the goal from the starting it must know when to find the key, when to unlock the door and so on. The stage cost is a function of the the state and the action, thus giving what will be the cost if the agent at this state will take this action. The stage cost is defined as follows

If the action is "MF" (move forward), we check the cell in front of the current cell in the direction of the agent. If the new position is obstructed by an obstacle, the cost is set to infinity. If the new position is the goal, the cost is set to 10. If the new position is the door and it is locked, the cost is set to infinity. If the new position is the key and the key has already been picked up, the cost is set to infinity. If none of these conditions are met, the cost is set to 10. Thus till will force the agent to only use MF when there is a path in front

If the action is "TL" (turn left) or "TR" (turn right), the cost is always 10.

If the action is "PK" (pick key), we check the cell in front of the current cell in the direction of the agent. If the new position is the key and the key has not already been picked up and the door is not unlocked, the cost is set to 10. If the new position is the key and the key has not already been picked up and the door is unlocked, the cost is set to infinity. If the new position is the key and the key has already been picked up, the cost is set to infinity. If the new position is not the key, the cost is set to infinity. This will make sure that the agent picks up the key in cases where the door is locked and the agent does not have the key

If the action is "UD" (unlock door), we check the cell in front of the current cell in the direction of the agent. If the new position is the door and the door is locked and the agent has not picked up the key, the cost is set to infinity. If the new position is the door and the door is locked and the agent has picked up the key, the cost is set to infinity. If the new position is the door and the door is unlocked, the cost is set to 10. If the new position is not the door, the cost is set to infinity.

The cost associated with each state-action pair depends on the current state of the environment, the action taken by the agent, and the outcome of that action. The cost is used to guide the search for an optimal path from the starting state to the goal state.

Control Space

The control space is the set of all possible values that the control input can take at any given time. For our given case we have 5 control inputs

- MF - Move forward
- TL - Turn Left
- TR - Turn Right
- PK - Pick Key
- UD - Unlock Door

We can map each of them to a value as well, where 0 corresponds to MF, 1 to TL and so on [0:"MF", 1:"TL", 2:"TR", 3:"PK", 4:"UD"]

Initial State The initial state is the starting state from which the agent begins its interaction with the environment. I have defined this from the environment, as the env gives me parameters

- $agentpos$ - Gives the x,y coordinate of the agent
- $agentdir$ - Gives the initial direction of the agent
- key - Gives if the agent has the key or not
- $isdoor$ - Gives if the door is locked or unlocked

Using the above 5 values, I am able to define the initial state of the agent.

Value function, Q matrix Before we formulate the terminal cost we need to define the Value function and the Q matrix which shall be used to find the optimal policy

The value function V is a function that estimates the total expected cost that an agent will receive from a given state onward, by following a specific policy. In other words, it represents the long-term value of a state. The value of a state is calculated as the sum of the expected immediate cost and the expected value of the next state, discounted by a factor called the discount factor. The value function is used to evaluate the quality of a policy, i.e., how good a policy is at minimizing the expected cumulative cost.

The Q function, on the other hand, is a function that estimates the expected cumulative cost of taking a specific action from a given state and following a specific policy. In other words, it represents the value of taking an action in a state under a particular policy. The Q function is used to select the best action to take in a given state, by selecting the action with the lowest Q value.

During the policy iteration process, the policy and the value function are updated iteratively. The policy is updated by selecting the best action to take in each state based on the Q function. The value function is then updated based on the new policy by applying the Bellman equation. This process continues until the policy and the value function converge to their optimal values. The optimal policy and the optimal value function are those that minimizes the expected cumulative reward.

We initialized the Value function as a matrix of dimension [Time horizon, length of state space] which gives at every time step for each state what is the cost. We initialize this matrix with infinity as all the elements

We initialize the Q matrix by a matrix of dimension length of the state space by the length of the control actions, in our case 5. All the elements are taken as 1 for now. Both these matrix shall play a vital role in the Dynamic programming algorithm.

Terminal Cost Terminal cost is defined as the cost to the agent when it is at a particular location. So ideally in our implementation the terminal cost at the goal must be negative aka it is favorable for the agent to be at the goal position at the T time. I used an implementation where the states iterates over all possible orientations, door statuses, and key statuses. For each combination, it creates a goal state by setting the agent's location to the goal position and the orientation, key status, and door status to the corresponding values. It then finds the index of the goal state in the state space

Once the goal state index is found, I set the value of the V function and the Q function at the goal state index to -1000. This value represents the terminal cost, which is the cost incurred by the agent when it reaches the goal state. By setting the terminal cost to a large negative value, the agent is discouraged from remaining in the goal state and is incentivized to explore other states in search of a better policy.

Policy Matrix The P-matrix is essentially the policy matrix, which specifies the action to be taken at each state in the MDP in order to minimize the expected total cost. At each iteration of the algorithm, the policy and value function are updated based on the Bellman equation until convergence is achieved. The policy matrix is used by the agent to determine the optimal action to take in any given state of the MDP. The elements of P represent the optimal action to take at each state in order to minimize the expected total cost over the remaining time horizon, given the current estimates of the value function. The minimum value in the Q-matrix is taken along each row and stored in the V-matrix. The index of the minimum value in each row of the Q-matrix is stored in the P-matrix.
Convergence is achieved when the value function at time t is equal to the Value function at time t+1
The dimension of the Policy matrix that I defined is Time horizon by length of state space.

After the implementation of the dynamic programming algorithm, once we achieve convergence, we can use that time with the index of any state (the start state is given in our case) and we will get the optimal policy (u0) for state x0 at time T convergence, then similarly we can find the optimal policy for state x1, which is resulted from x0 and u0 at time t+1 and so on.

C. Problem formulation for part B - random environment

In this part we have 36 random environments of same sizes. The salient feature is that each has 2 doors, 3 goal locations and 3 key positions and we do not know where the key or the goal might be. The start location is known and the position for both the doors are known in the environment along with the orientation of the agent is known.

State Space
We know we have the x, y, orientation, door1 status, door2 status, key status, key position and goal position therefore I formulated the state space as the possible combinations of [x, y, orientation, door1, door2, key stat, key position, goal position] where x y and orientation define the pose of the agent.
Door1 and door2 take binary values giving the status of the doors respectively. 0 for closed and 1 for open
Key status tales a binary value, 0 for agent does not have the key and 1 for it has the key
Key position takes 3 values, 0, 1 and 2. 0 means that the key is at (1,1), 1 means the key is at (2,3) and 2 means that the key is at (1,6)
Similarly the goal position takes 3 values, 0, 1 and 2. 0 means that the goal is at (5,1), 1 means the goal is at (6,3) and 2 means that the goal is at (5,6)
With such a choice of the state space, I am able to cover all possible scenarios in the 36 maps

- where x can go from 0 to the width of the map
- where y can go from 0 to the height of the map

- where orientation can be 0 for right, 1 for down, 2 for left and 3 for up
- where door1 can be 0 if door1 is locked, and 1 if door1 is unlocked
- where door2 can be 0 if door2 is locked, and 1 if door2 is unlocked
- key stat can be 0 if the agent does not have the key and 1 if it has
- key position can be 0,1,2 each integer referring to a particular location
- goal position can be 0,1,2 each integer referring to a particular location

So initially the state space had width*height*4*2*2*2*3*3 but i removed the cells where the x and y coincided with the walls. Thus reducing the state space and optimizing the calculations. To reduce the possible state spaces I removed all those where the x and y coincided with the walls in the map.

Therefore any possible state of the environment can be accurately be captured by this array. To give a small example, suppose my state is [3,5,3,0,0,0,1,2] from this we can say the x position is 3, the y is 5 the orientation is looking up, door 1 is locked door 2 is locked, key has not been picked, key is at (1,1) and goal is at (6,3)

Time Horizon The time horizon I defined was one minus the carnality of the length of the state space. The time horizon is: 16704

Motion Model I have defined the motion model similar to that as defined in part A, with the only exception that instead of looking for 1 door I am now catering to both doors following steps in the motion model

if the action is "MF" move forward the first element is the x coordinate, the second element is the y coordinate the 3rd element is the orientation the fourth element is the key status and the 5th element is door status. If the agent is facing right (orientation 0), the first element of the state (which represents the agent's x-coordinate) is incremented by 1, as long as it doesn't exceed the height of the environment minus 1 (i.e., the agent doesn't move off the top edge of the grid).

If the agent is facing down (orientation 1), the second element of the state (which represents the agent's y-coordinate) is incremented by 1, as long as it doesn't exceed the width of the environment minus 1 (i.e., the agent doesn't move off the right edge of the grid).

If the agent is facing left (orientation 2), the first element of the state is decremented by 1, as long as it doesn't go below 0 (i.e., the agent doesn't move off the bottom edge of the grid).

If the agent is facing up (orientation 3), the second element of the state is decremented by 1, as long as it doesn't go below 0 (i.e., the agent doesn't move off the left edge of the grid).

If there is a wall in front of the agent, the "MF" action will have no effect. Similarly, if the door1 or door2 is locked and the cell in front has any door 1 or 2, the agent's position will not change. Similarly, if the next cell is a key and the key has not been picked up, the agent will not move into that cell unless the key is picked up.

If we use "TR" or "TL" we only have to update the orientation of the agent. if the action is "TR"

- agent direction initially 0(right) change to 1(down)
- agent direction initially 1(down) change to 2(left)
- agent direction initially 2(left) change to 3(up)
- agent direction initially 3(up) change to 0(right)

similarly if we use "TL"

- agent direction initially 0(right) change to 3(up)
- agent direction initially 1(down) change to 0(right)
- agent direction initially 2(left) change to 1(down)
- agent direction initially 3(up) change to 2(left)

If the action is "PK" and the next cell in the direction of the agent has the key and the agent does not have the key then the agent will pick up the key and the key status will change from 0 to 1, indicating that the agent has picked up the key.

Similarly if the action is "UD" and the next cell in the direction of the agent is a door(1 or 2) and the agent has the key, aka the key status is 1, And as I know the location of the door, if the coordinates is of door1, we will change the door1 status to 1, similarly if the door ahead is door 2, then upon doing the action of "UD" door 2 will be changed from 0 to 1. The status change will only happen if the door was initially locked

Thus this motion model cover all cases in the environment and for each case and scenario I have defined what the state will be post the motion model.

Stage cost The stage cost is defined similar to that of the part A except we now check both the doors and key locations The stage cost is a function of the the state and the action, thus giving what will be the cost if the agent at this state will take this action. The stage cost is defined as follows

If the action is "MF" (move forward), we check the cell in front of the current cell in the direction of the agent. If the new position is obstructed by an obstacle, the cost is set to infinity. If the new position is the goal(Depending on the state we know where the goal is for that state), the cost is set to 10. If the new position is the door(any door 1 or 2) and it is locked, the cost is set to infinity. If the new position is the key(depending on the state we know where the key is) and the key has already been picked up, the cost is set to infinity. If none of these conditions are met, the cost is set to

10. Thus till will force the agent to only use MF when there is a path in front

If the action is ”TL” (turn left) or ”TR” (turn right), the cost is always 10.

If the action is ”PK” (pick key), we check the cell in front of the current cell in the direction of the agent. If the new position is the key and the key has not already been picked up and the door is not unlocked, the cost is set to 10. If the new position is the key and the key has not already been picked up and the door is unlocked, the cost is set to infinity. If the new position is the key and the key has already been picked up, the cost is set to infinity. If the new position is not the key, the cost is set to infinity. This will make sure that the agent picks up the key in cases where the door is locked and the agent does not have the key

If the action is ”UD” (unlock door), we check the cell in front of the current cell in the direction of the agent. If the new position is the door(any door 1 or 2) and the door is locked and the agent has not picked up the key, the cost is set to infinity. If the new position is the door and the door is locked and the agent has picked up the key, the cost is set to infinity. If the new position is the door and the door is unlocked, the cost is set to 10. If the new position is not the door, the cost is set to infinity.

The cost associated with each state-action pair depends on the current state of the environment, the action taken by the agent, and the outcome of that action. The cost is used to guide the search for an optimal path from the starting state to the goal state.

Control Space

The control space is the set of all possible values that the control input can take at any given time. For our given case we have 5 control inputs

- MF - Move forward
- TL - Turn Left
- TR - Turn Right
- PK - Pick Key
- UD - Unlock Door

We can map each of them to a value as well, where 0 corresponds to MF, 1 to TL and so on [0:”MF”, 1:”TL”, 2:”TR”, 3:”PK”, 4:”UD”]

Initial State The initial state is the starting state from which the agent begins its interaction with the environment. We are given that the The agent is initially spawned at (3, 5) facing up. So the initially the known parts of the state is x = 3,y = 5,orientation = 3, the rest need to be loaded from the environment I have defined this from the environment, as the env gives me parameters

- door 1 - Gives the status of door 1 coordinate of the agent from the environment
- door 2 - Gives the status of door 2 coordinate of the agent from the environment
- key status - Gives if the agent has the key or not from the environment
- key position - Gives the position of the key for that environment and then the value is set as 0,1,2 accordingly
- goal position - Gives the position of the goal for that environment and then the value is set as 0,1,2 accordingly

Using the above 8 values, I am able to define the initial state of the agent.

Value function, Q matrix The Value function and the Q matrix are defined exactly as in part A
The shape of V is: (16704, 16704)
The shape of Q is: (16704, 5)

Terminal Cost Terminal cost is defined as the cost to the agent when it is at a particular location. So ideally in our implementation the terminal cost at the goal must be negative aka it is favorable for the agent to be at the goal position at the T time.
I used an implementation where the states iterates over all goal coordinates with the current orientation, door statuses, key status, and key position. A mask is created to filter the goal state in the state space and the index of the goal state is calculated. The corresponding value in the value function matrix V and the Q function matrix Q are set to a large negative value (-1000) to discourage the agent from reaching the goal state before the end of the time horizon. By setting the terminal cost to a large negative value, the agent is discouraged from remaining in the goal state and is incentivized to explore other states in search of a better policy.

Policy Matrix The policy matrix also is initialized the same way as we did in part A. Except the difference is that this policy matrix caters to all possible door, key and goal combinations and can give the optimal policy for any map from any starting point.
The shape of P is: (16704, 16704)

Thus the formulation for part B has now ended

III. Technical Approach

To solve the problem we have formulated for both part A and B we will be implementing the dynamic programming algorithm. The problem involves finding an optimal sequence of actions for an agent to reach a goal state in a grid-world environment, while considering the state of doors, keys, and their locations.

The implementation first generates all possible states in the environment and initializes their values. Then, it updates the values of each state at each time step by computing the optimal action that leads to the minimum cost. The cost is computed based on the stage cost of the current state and the expected future cost of the next state, which is computed using the value of the next state. The implementation uses the Bellman equation to update the values.

The implementation then returns the optimal values and actions for each state and time step. Finally, the implementation generates a sequence of actions for reaching the goal state using the computed values and actions.

The V matrix contains the optimal value function for each state at each time step. The value function is initialized to infinity for all states except the goal state which is initialized to -1000. The V matrix is then updated iteratively, from the last time step to the first, using the Bellman equation, which calculates the value function for each state as the minimum of the sum of the stage cost and the value function of the next state.

The Q matrix contains the expected total cost of taking a particular action in a given state, plus the expected cost of reaching the goal state from the next state. The Q matrix is also updated iteratively, from the last time step to the first, using the Bellman equation.

The P matrix contains the optimal policy, which is the action that minimizes the total cost at each state and time step. It is populated by storing the index of the action that minimizes the Q value for each state at each time step.

Finally, the V, Q, and P matrices are used to determine the optimal sequence of actions to reach the goal state from the initial state. This is done by initializing the state to the initial state, and repeatedly selecting the action that minimizes the Q value at the current state, until the goal state is reached.

The algorithm is said to converge by comparing the value function at each time step to the value function at the previous time step. If the two value functions are equal, then convergence is assumed to have been achieved. Finally the action for a given state can be extracted from the policy matrix P by looking up the index of the minimum value for that state in the first dimension of P. The index corresponds to the action to take in that state.

The dynamic programming algorithm has the following key points

- Dynamic programming: an algorithm for computing the optimal value function $V_t^*(x)$ and an optimal policy π^*
- Idea: compute the value function and policy backwards in time
- Generality: handles non-linear non-convex problems
- Complexity: polynomial in the number of states $|X|$ and number of actions $|U|$
- Efficiency: much more efficient than a brute-force approach evaluating all possible policies

The algorithm that was used in the technical approach can be seen as follow Dynamic Programming

```
1: **Input:** MDP(X,U,p$_0$, $p_f, T, \lambda, q, \gamma$)
2: $T = |X| - 1$
3: $V_T(\tau) = q(x)\ \forall x \in X$
4: **for** $t = (T-1)$ ,...., $0$ **do**
5:     $Q_t(x,u) = l(x,u) + \gamma E_{x' \sim p_f(\cdot|x,u)}[V_{t+1}(x')]$ $\forall x \in X, u \in U(x)$
6:     $V_t(x) = \min_{u \in U(x)} Q_t(x,u)$ ,$\forall\ x \in X$
7:     $\pi_t(x) = \arg\min_{u \in U(x)} Q_t(x,u)$ ,$\forall\ x \in X$
8:     **if** $V_t(i) = V_{t+1}(i)$ for all $i \in V \setminus \{\tau\}$ **then**
9:         break
10: **end for**
```

using the above technical approach on the formulation for part A and part B, I was able to run the algorithm and the results obtained are discussed further

IV. Results and Discussions

We were provided with two types of environments, the "Known Map" scenario, where different control policies are computed for each of the 7 environments provided in the starter code, while in the "Random Map" scenario, a single control policy matrix that can be used for any of the 36 random 8x8 environments is computed. We ran the dynamic programming algorithm on both parts. We were able to get optimal control policies for both the cases and the agent was successfully able to reach the goal position. The details about the results are shown below

A. Known Maps - Part A

For all the 7 known locations were got the optimal actions and were able to make a gif showing that as well.

5x5 grid - The trajectory path is shown in Figure 1, and we can see the actions taken by the agent to reach the goal position. The optimal policy for this environment came to be [TL,TL,PK,TR,UD,MF,MF,TR,MF]. The agent was able to pick the key, unlock the door and move to the goal position indicating that the formulation and implementation was correct. Thus using these optimal sequence the agent reached the goal position and the gif was made. The gif for this environment can be seen HERE

6x6 direct grid - The trajectory path is shown in Figure 2, and we can see the actions taken by the agent to reach the goal position. The optimal policy for this environment came to be [MF, MF, TR, MF, MF]. The agent did not need to pick the key as the path was a direct one, and truly the agent did not reach for the key and directly made its way to the goal indicating that the formulation and implementation was correct. Thus using these optimal sequence the agent reached the goal position and the gif was made. The gif for this environment can be seen HERE

6x6 Normal grid - The trajectory path is shown in Figure 3, and we can see the actions taken by the agent to reach the goal position. The optimal policy for this environment came to be

[TL, MF, PK, TL, MF, TL, MF, TR, UD, MF, MF, TR, MF]. The agent went to the key picked it up then went to the door unlocked it an then made its way to the goal indicating that the formulation and implementation was correct. Thus using these optimal sequence the agent reached the goal position and the gif was made. The gif for this environment can be seen HERE

6x6 Shortcut grid - The trajectory path is shown in Figure 4, and we can see the actions taken by the agent to reach the goal position. The optimal policy for this environment came to be [PK, TL, TL, UD, MF, MF]. The agent picked the key as it was infront of it, turned to the door and unlocked it and then moved to the goal indicating that the formulation and implementation was correct. Thus using these optimal sequence the agent reached the goal position and the gif was made. The gif for this environment can be seen HERE

8x8 Direct grid - The trajectory path is shown in Figure 5, and we can see the actions taken by the agent to reach the goal position. The optimal policy for this environment came to be [MF, TL, MF, MF, MF, TL, MF]. The agent did not need to pick the key and directly made its way to the goal indicating that the formulation and implementation was correct. Thus using these optimal sequence the agent reached the goal position and the gif was made. The gif for this environment can be seen HERE

8x8 Normal grid - The trajectory path is shown in Figure 6, this was the biggest and most challenging environement for the agent as it had to take 24 steps to reach the goal. The optimal policy for this environment came to be [TR, MF, TL, MF, TR, MF, MF, MF, PK, TL, TL, MF, MF, MF, TR, UD, MF, MF, MF, TR, MF, MF, MF]. The agent had to travel far to pick the key then come back to the door location and unlock it and then move to the goal indicating that the formulation and implementation was correct. Thus using these optimal sequence the agent reached the goal position and the gif was made. The gif for this environment can be seen HERE

8x8 Shortcut grid - The trajectory path is shown in Figure 7, and we can see the actions taken by the agent to reach the goal position. The optimal policy for this environment came to be [TR, MF, TR, PK, TL, UD, MF, MF]. The agent picked the key which was close then unlocked the door and made its way to the goal indicating that the formulation and implementation was correct. Thus using these optimal sequence the agent reached the goal position and the gif was made. The gif for this environment can be seen HERE

B. UnKnown Maps - Part B

In this part the main result we wanted to show was that with one policy matrix can we extract the optimal policies for all 36 maps, This means that running the dynamic programming algorithm only once will yield a policy matrix that can be applied to all 36 maps. This approach greatly reduces the computational resources required to solve the problem, making it more efficient and practical.
Figure 8, 9 10 and 11 show snippets of the control policy for different types of environments, with both doors open, with one door open and with both doors closed and we can see with a same policy matrix we could get the optimal policy for all.
The gif for all 36 environments can be seen HERE

It was also seen that in part B the state space was in order of 16000 and in part A the state space was in order of less than 500, thus even though dynamic programming has a polynomial time complexity, the code took significantly longer time to run in part B than in A, even when the map was only 8x8 which is not very big. Thus as the state space become too large dynamic programming might not be the best algorithm from a practical point of view.

C. Conclusion

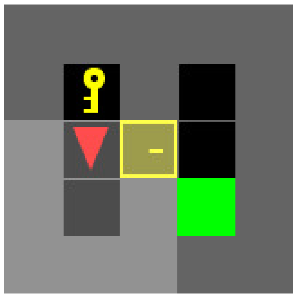
In conclusion, path planning and finding optimal control policies are crucial for autonomous robots that need to navigate complex environments to perform tasks efficiently. This paper presents a dynamic programming algorithm for autonomous navigation in a Door, Key, and Goal environment. The algorithm computes different control policies for known maps and a single control policy for random maps. The proposed approach effectively calculates optimal control policies, and the results demonstrate the effectiveness of the algorithm. By formulating the necessary components, such as the state of the system, possible actions, transition probabilities, and cost associated with each action, the dynamic programming algorithm can be formulated. This problem has potential applications in robotics and addresses the significant issue of autonomous navigation in complex environments where obstacles can impede progress towards a goal.

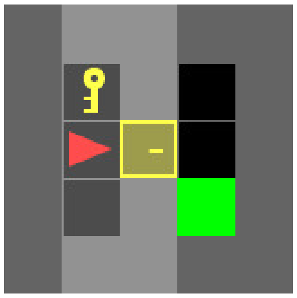
D. Collaboration

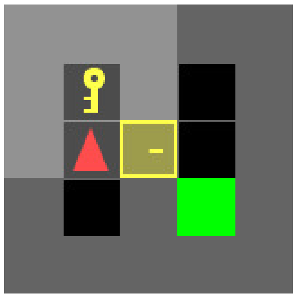
Surya Pilla Pritwiraj Paul

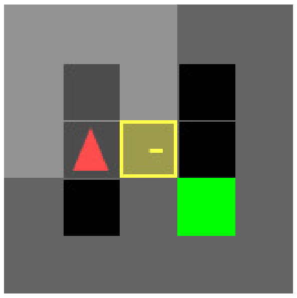
The drive Link to all GIFS

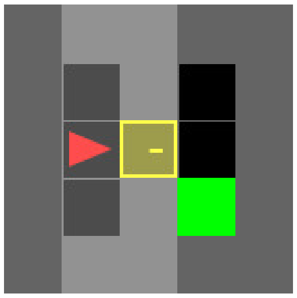
https://drive.google.com/drive/folders/1wkts2Ndg8aYQ89ZaQle0ZyHSS4Y8bA2c?usp=sharing

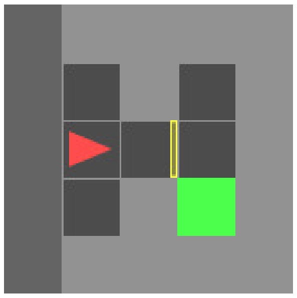

(a) Initial State

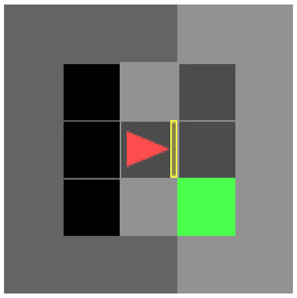

(b) Turn Left

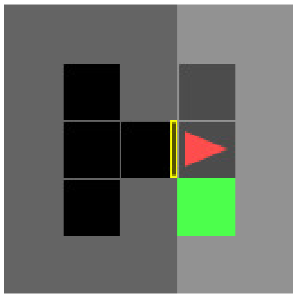

(c) Turn Left

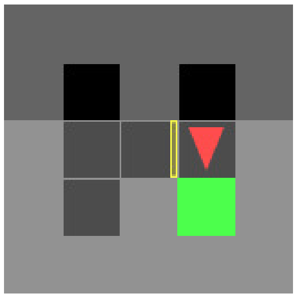

(d) Pick Key


(e) Turn Right


(f) Unlock Door


(g) Move Forward


(h) Move forward


(i) Turn Right

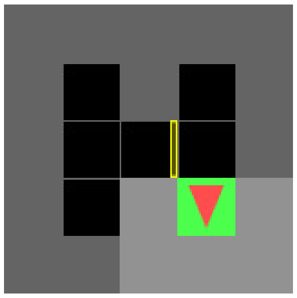
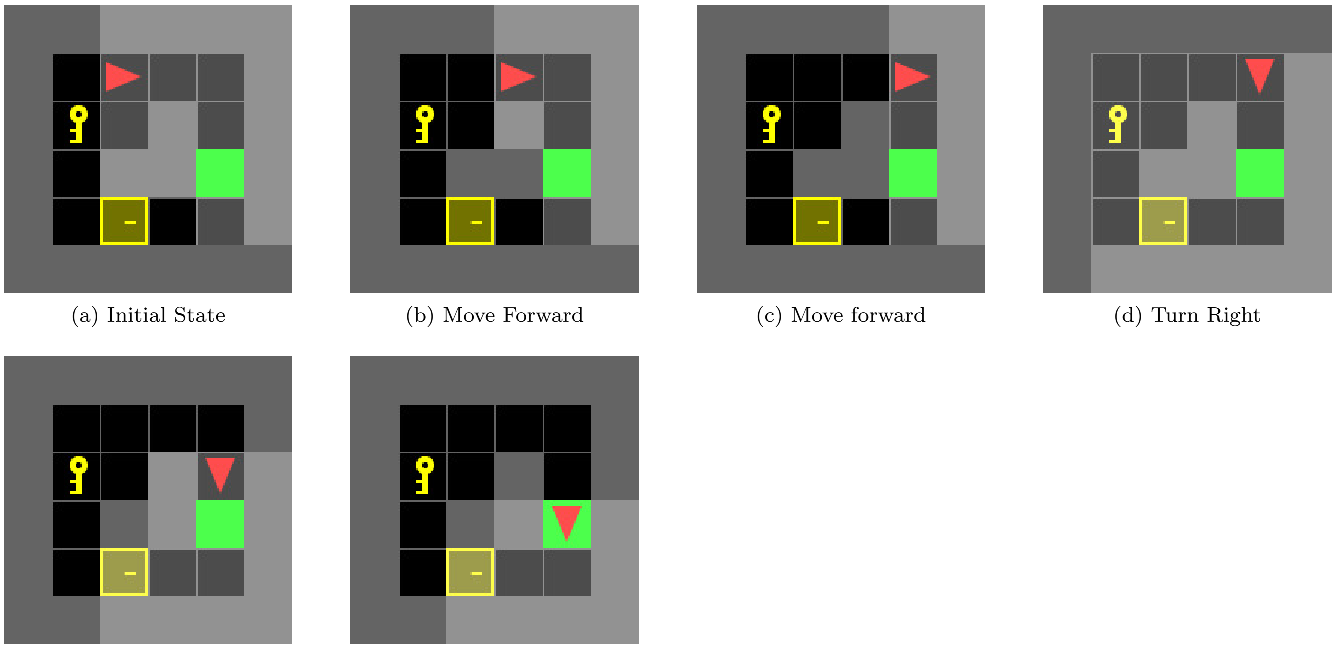

(j) Move forward

Figure 1: Path for 5x5 grid



(a) Initial State

(b) Move Forward

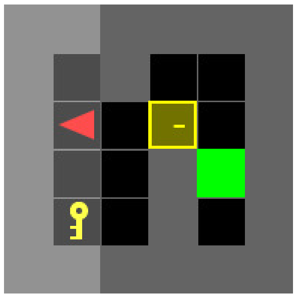
(c) Move forward

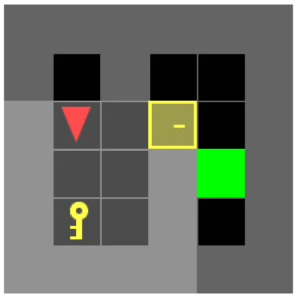
(d) Turn Right

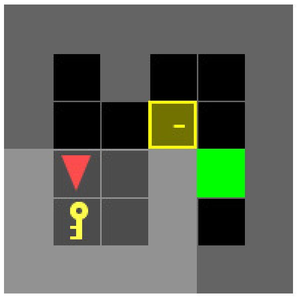
(e) Move forward

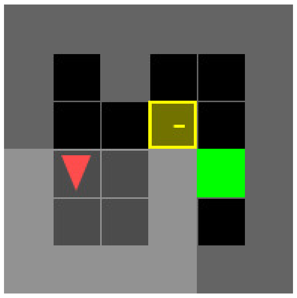
(f) Move forward

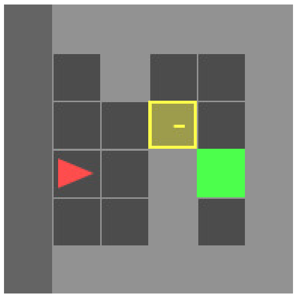
Figure 2: Path for 6x6 Direct Grid

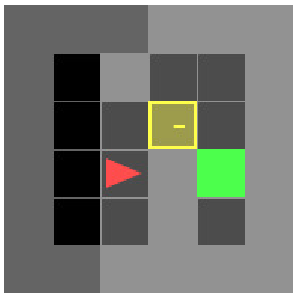

(a) Initial State

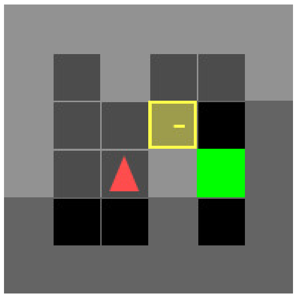

(b) Turn left

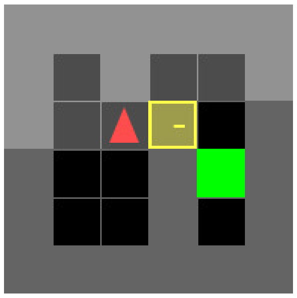

(c) Move forward

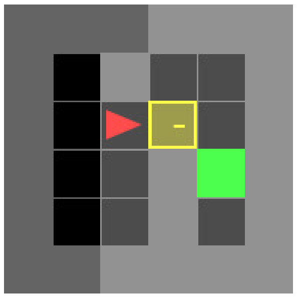

(d) Pick Key

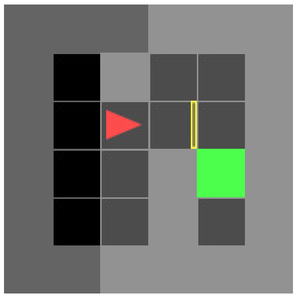

(e) Turn Left

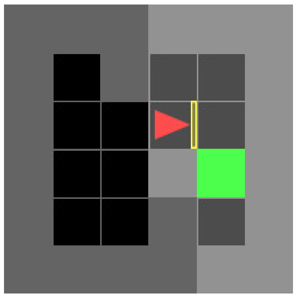

(f) Move forward

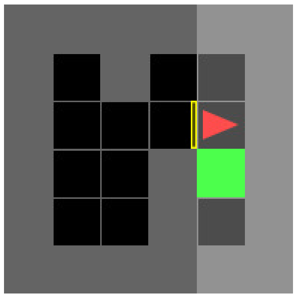

(g) Turn Left

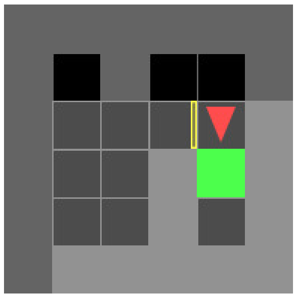

(h) Move forward


(i) Turn Right


(j) Unlock Door


(k) Move forward


(l) Move forward


(m) Turn Right

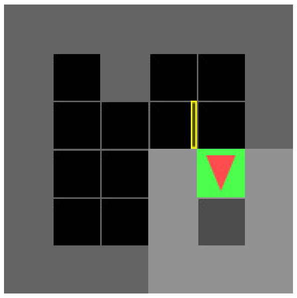
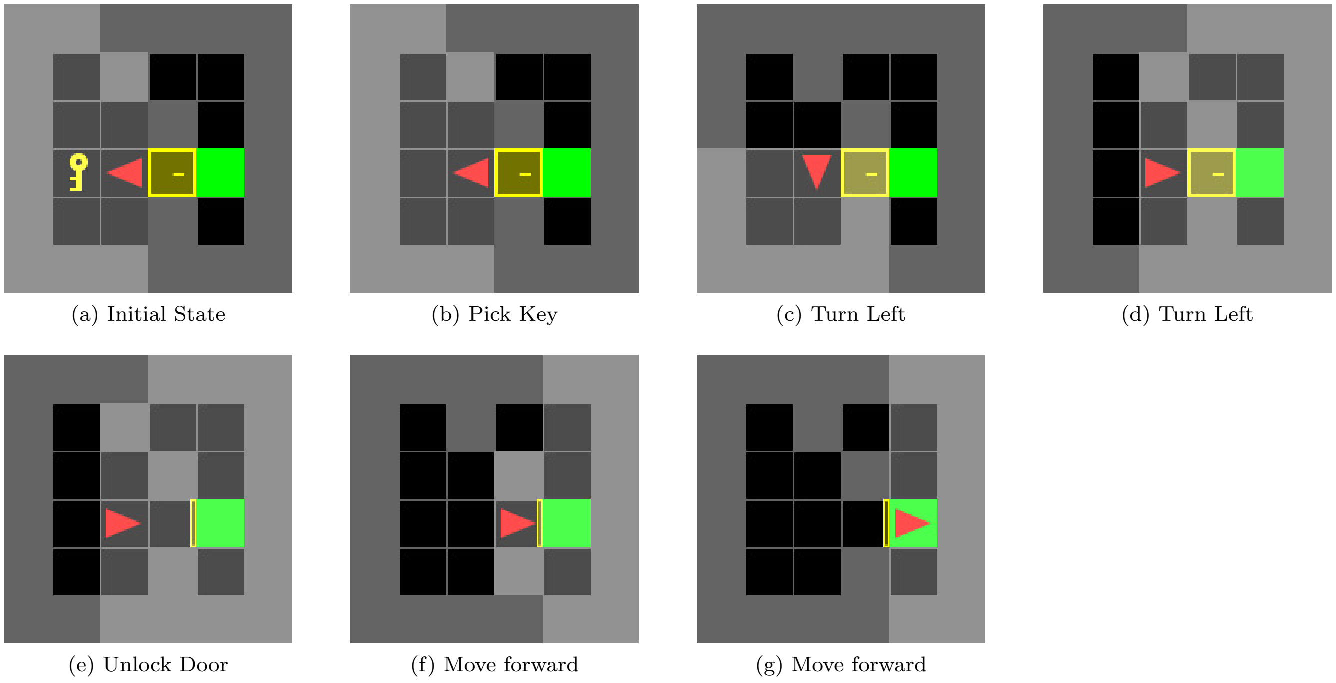
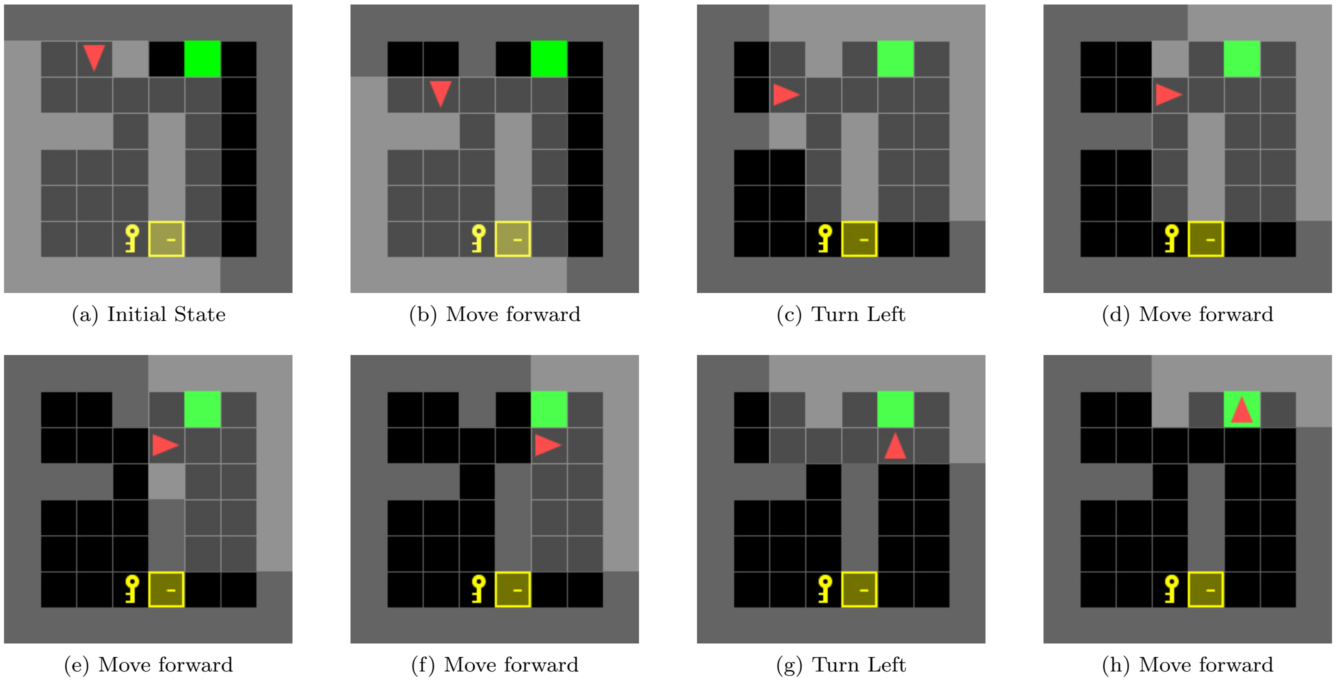

(n) Move forward

Figure 3: Path for 6x6 Normal grid


(a) Initial State (b) Pick Key (c) Turn Left (d) Turn Left

(e) Unlock Door (f) Move forward (g) Move forward

Figure 4: Path for 6x6 Shortcut grid


(a) Initial State (b) Move forward (c) Turn Left (d) Move forward

(e) Move forward (f) Move forward (g) Turn Left (h) Move forward

Figure 5: Path for 8x8 Direct grid

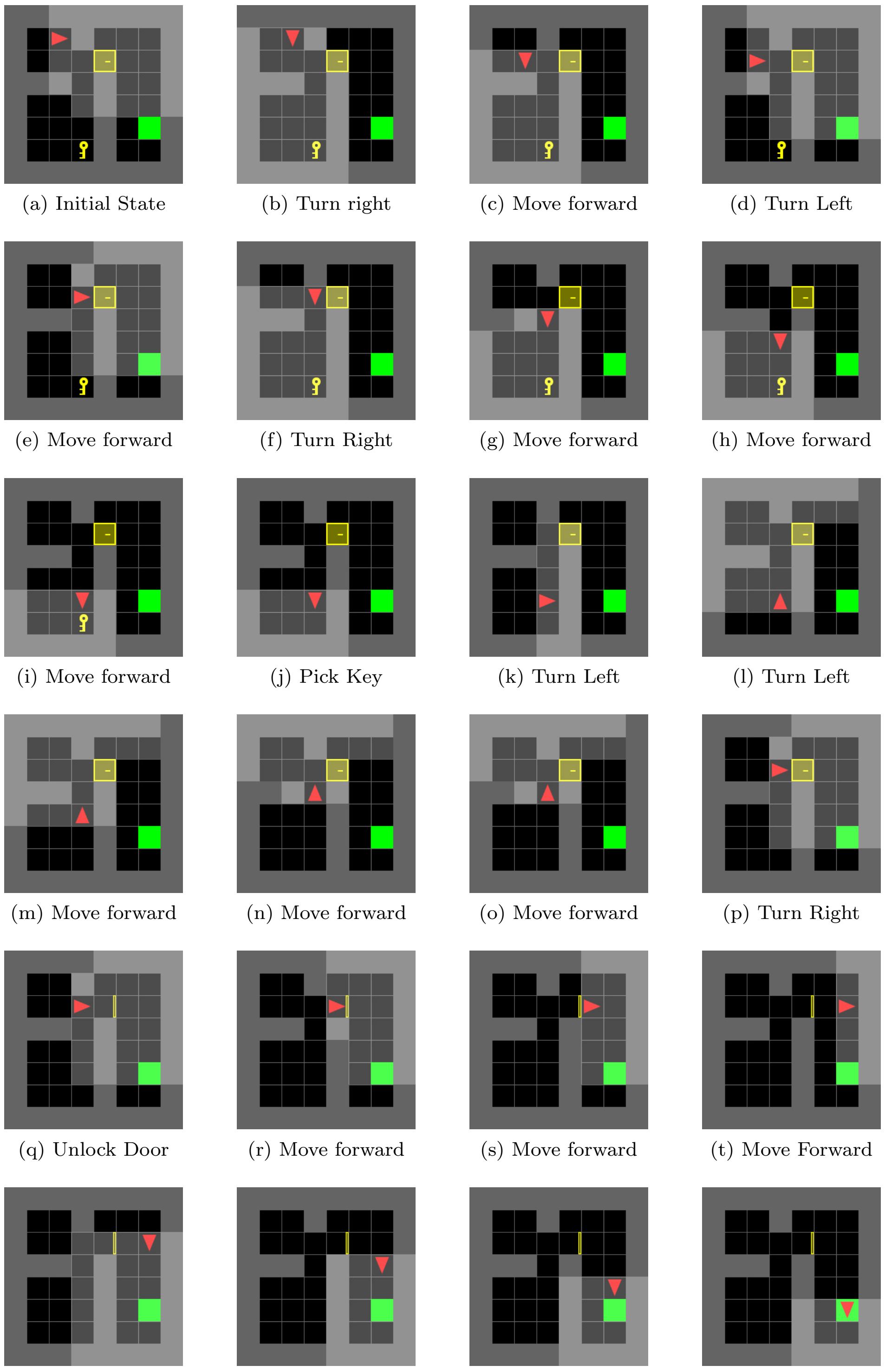
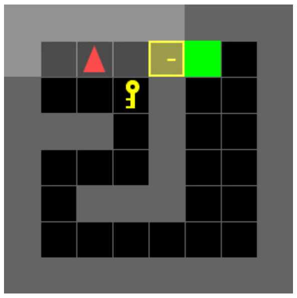
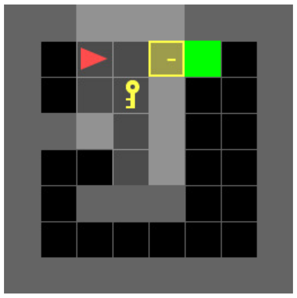
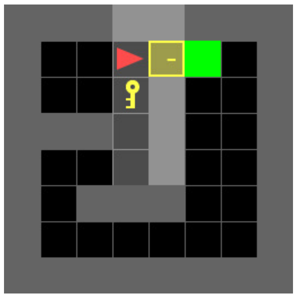
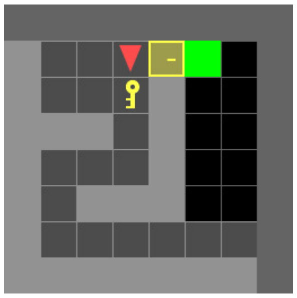

(a) Initial State
(b) Turn right
(c) Move forward
(d) Turn Left
(e) Move forward
(f) Turn Right
(g) Move forward
(h) Move forward
(i) Move forward
(j) Pick Key
(k) Turn Left
(l) Turn Left
(m) Move forward
(n) Move forward
(o) Move forward
(p) Turn Right
(q) Unlock Door
(r) Move forward
(s) Move forward
(t) Move Forward
(u) Turn Right
(v) Move forward
(w) Move forward
(x) Move forward

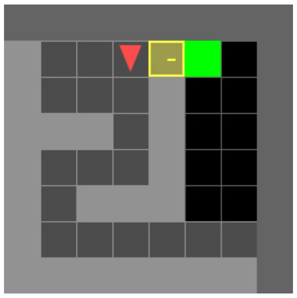
Figure 6: Path for 8x8 Normal

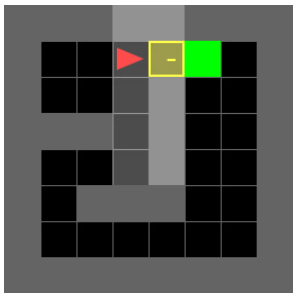

(a) Initial State

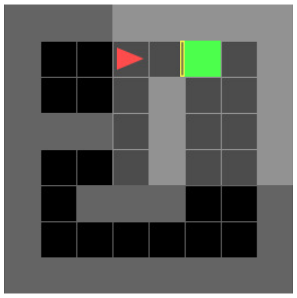

(b) Turn right

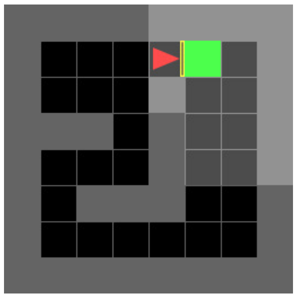

(c) Move forward


(d) Turn Right


(e) Pick Key


(f) Turn Left


(g) Unlock Door


(h) Move forward

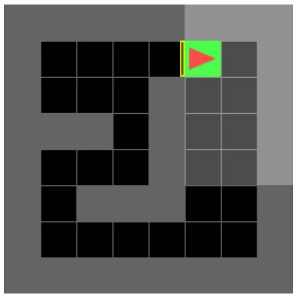

(i) Move forward

Figure 7: Path for 8x8 Shortcut grid

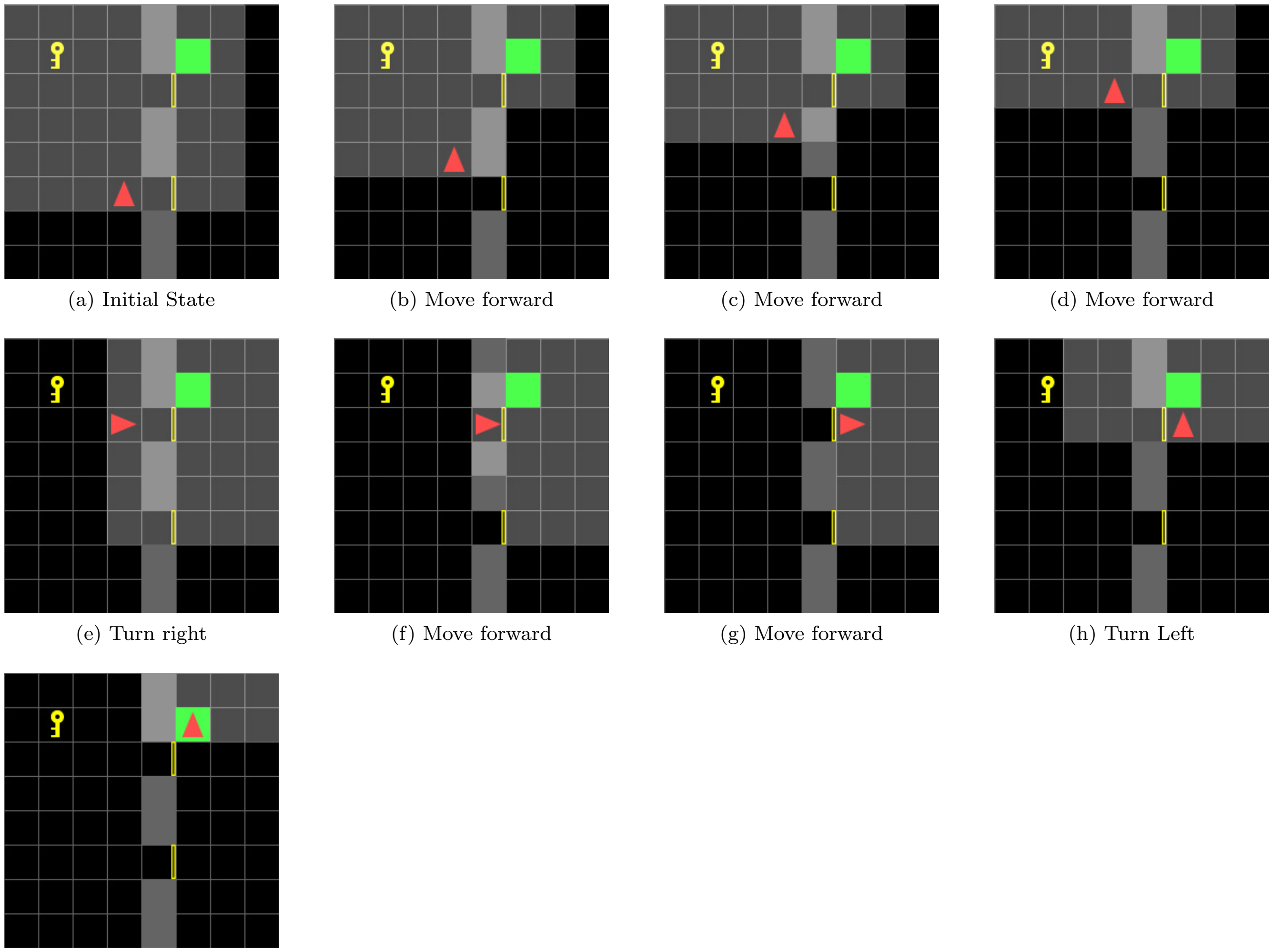
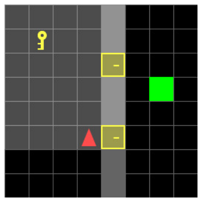
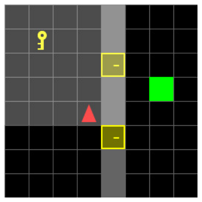
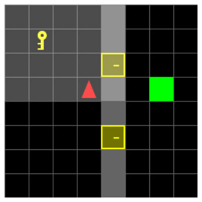
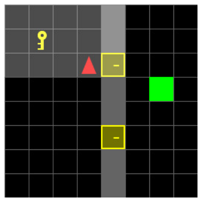

(a) Initial State (b) Move forward (c) Move forward (d) Move forward (e) Turn right (f) Move forward (g) Move forward (h) Turn Left (i) Move forward

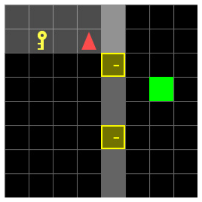
Figure 8: Path for random env with both doors open (rand env 1)

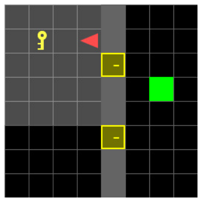

(a) Initial State

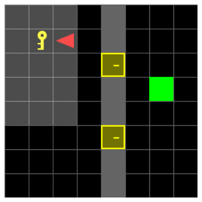

(b) Move forward

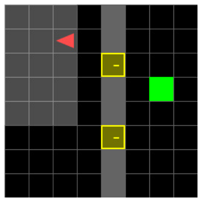

(c) Move forward

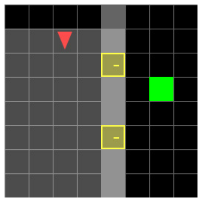

(d) Move forward

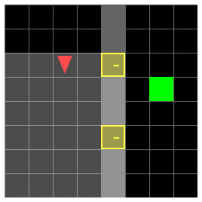

(e) Move forward

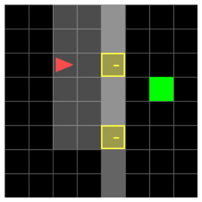

(f) Turn Left

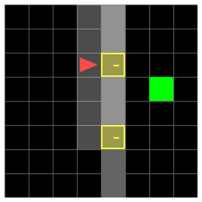

(g) Move forward

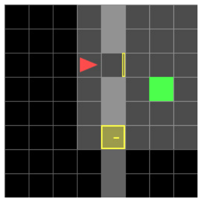

(h) Pick key

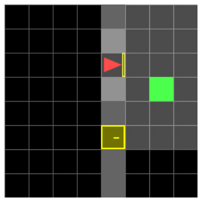

(i) Turn left

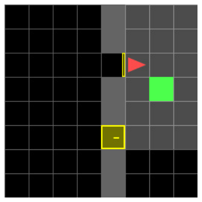

(j) Move forward

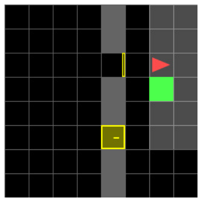

(k) Turn Left

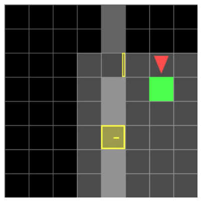

(l) Move Forward


(m) Unlock Door 1


(n) Move forward


(o) Move forward


(p) Move forward


(q) Turn Right

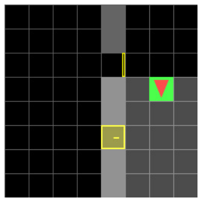
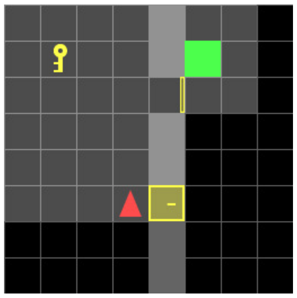
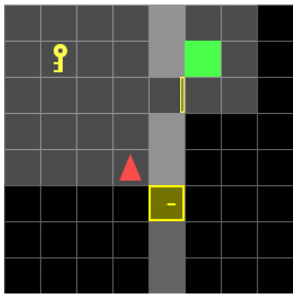
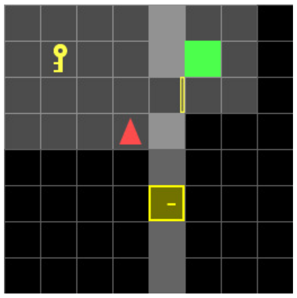
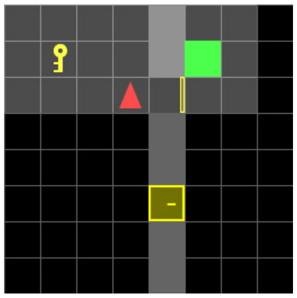

(r) Move forward

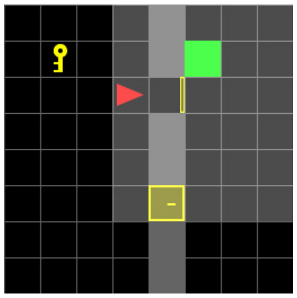
Figure 9: Path for random env booth doors closed (rand env 8)

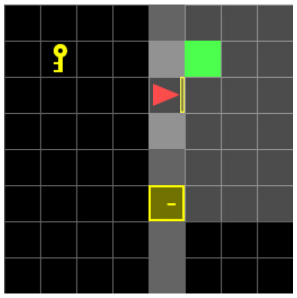

(a) Initial State

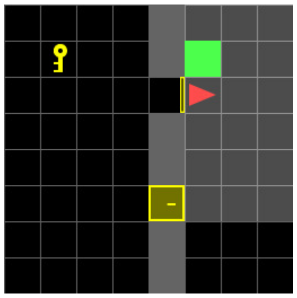

(b) Move forward

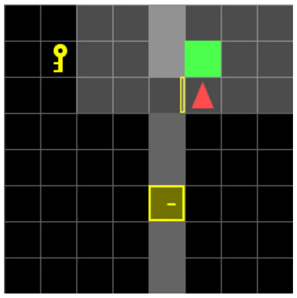

(c) Move forward


(d) Move forward


(e) Turn right


(f) Move forward


(g) Move forward


(h) Turn Left

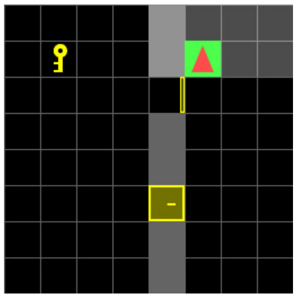
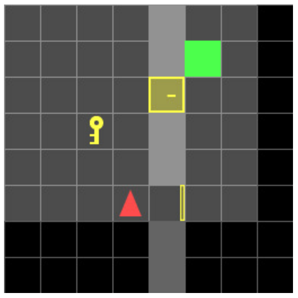
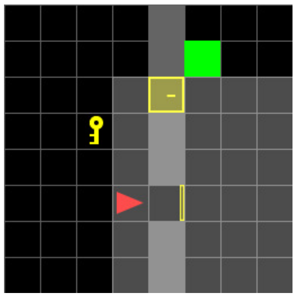
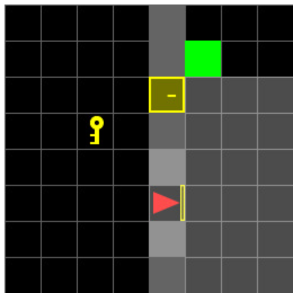
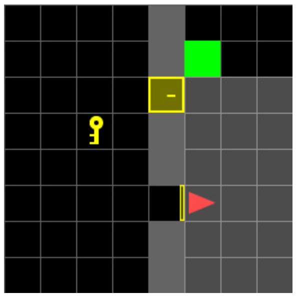

(i) Move forward

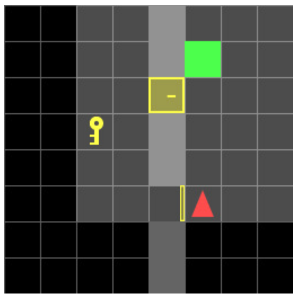
Figure 10: Path for random env with one door open (rand env 2)

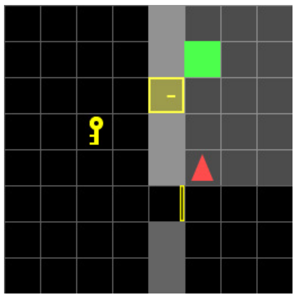

(a) Initial State

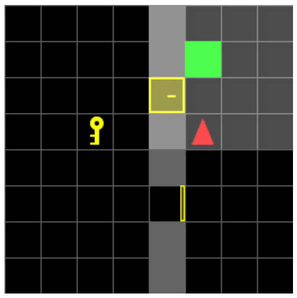

(b) Turn Right

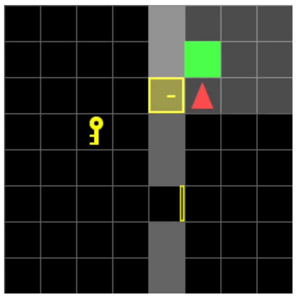

(c) Move forward


(d) Move forward


(e) Turn Left


(f) Move forward


(g) Move forward


(h) Move forward

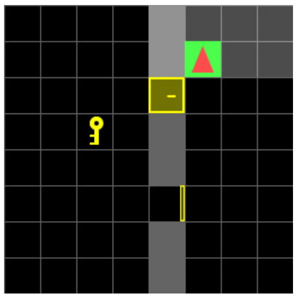

(i) Move forward

Figure 11: Path for random env with other door open (rand env 15)