ECE276B-P1

Zhenyu Wu | PID: A69030822

April 24, 2025

1 Introduction

In this project we are given a discretized grid map representation with key, door, and goal position. Our agent has position (x, y) and orientation with angular resolution of 90 degrees. Our goal is to compute the optimal control input sequence for our agent to interact with environment and reach the goal. In this report, I applied the Forward DP algorithm on DSP where I first convert the orginal DOC problem to DSP problem and compute forward DP. This problem instance is structurally similar to classic path planning, where we treat the robot as a point agent with discrete states and actions. The proposed approach can be extended to real-world applications such as mobile robot navigation in structured environments. However, in real-world robot navigation problems, more advanced designs are needed, for example, taking into account the C-space under different orientations.

2 Problem Formulation

In this Optimal Control problem, we have no disturbances with finite states and finite horizons. We have a DOC problem, and we are given following instances

1. Discretized grid map $M = H \times W$
2. Key Position $p^k = (x, y)$ for key position
3. Door Positions $p_i^d = (x_i, y_i)$ for i-th door
4. Goal Position $p^g = (x, y)$
5. Door State $d_i \in \{0, 1\}$ for i-th door, 0 stands for close and 1 stands for open
6. Agent Position $p = (x, y)$
7. Agent Direction $\theta \in \{UP, DOWN, LEFT, RIGHT\}$
8. Wall Positions $p^w \in R^{n_w \times 2}$ where each wall $p_i^w = (x_i, y_i)$

We need to solve the Known Map problem and the Unknown Map problem separately. In my formulation, both problems share the same action space, transition model, and cost structure. The only difference lies in the definition of the state in the Markov Decision Process (MDP).

2.1 State Space

For the **Known Map problem**, I formulate the state as $s = (p, \theta, k, d)$ where p is agent position, θ is agent direction, $k \in \{0, 1\}$ is a binary indicates if agent carries the key or not, d is a list of door state where each door $d_i \in \{0, 1\}$ can be open or close. Under this formulation, we have state space of size $|S| \in O(H \times W \times 4 \times 2 \times n_d \times 2) = O(H \times W \times n_d)$ where H and W is the size of map, n_d is number of doors. The factor of 4 accounts for 4 directions. The factor of 2 accounts for binary variables such as whether the agent has the key or whether each door is open. Note that this is an upper bound, as during construction I

filter out infeasible positions, such as those occupied by walls, which effectively reduces the size of the state space.

For the **Unknown Map problem**, I formulate the state as $s = (p, \theta, k, d, p^k, p^g)$ where p is agent position, θ is agent direction, $k \in \{0, 1\}$ is a binary indicates if agent carries the key or not, d is a list of door state where each door $d_i \in \{0, 1\}$ can be open or close, p^k is the key position, and p^g is the goal position. Since the environment layout is initially unknown, we must compute a policy that is general across all possible configurations. This requires the policy to consider every feasible combination of key and goal positions, as well as all potential door open/close patterns. Under this formulation, we have state space of size $|S| \in O(H \times W \times 4 \times 2 \times n \times 2 \times 3 \times 3) = O(H \times W \times n_d)$ where H and W is the size of map, n_d is number of doors. In the implementation, we are given $H = W = 10$, $n_d = 2$ and we have $n_w = 8$ walls. Under this condition, we have the state space $|S| = 92 \times 4 \times 2 \times 2 \times 2 \times 3 \times 3 = 26496$. For unknown map problem, we have agent initial position $p_0 = [4, 8]$ and direction $\theta_0 = UP$.

2.2 Control Space

For both problem, we have allowed controls as $\mathcal{U} \in \{MF, TL, TR, PK, UD\}$ where MF is move forward, TL is turn left, TR is turn right, PK is pick key, UD is unlock door.

2.3 Transition Model

2.3.1 Move forward

For the MF, we have the transition model as $f(s, MF) = s' = (p', \cdot)$ where $p' = p + \theta$ and all other elements in the input state s will stay the same. Such transition is **Invalid** if following conditions are met

1. $p' \in p^w$ front position is not wall
2. $p' \in p^k \wedge k = 0$ front position is key position and agent hasn't picked the key yet.
3. $p' = p_i^d \wedge d_i = 0$ front position is one of the door position p_i^d and door i hasn't open yet.

2.3.2 Turn left

For the TL, we have the transition model as $f(s, TL) = s' = (\theta', \cdot)$ where θ' follows below transitions and all other elements in the input state s will stay the same.

1. $\theta = RIGHT \rightarrow \theta' = UP$
2. $\theta = UP \rightarrow \theta' = LEFT$
3. $\theta = LEFT \rightarrow \theta' = DOWN$
4. $\theta = DOWN \rightarrow \theta' = RIGHT$

2.3.3 Turn right

For the TL, we have the transition model as $f(s, TL) = s' = (\theta', \cdot)$ where θ' follows below transitions and all other elements in the input state s will stay the same.

1. $\theta = RIGHT \rightarrow \theta' = DOWN$
2. $\theta = DOWN \rightarrow \theta' = LEFT$
3. $\theta = LEFT \rightarrow \theta' = UP$
4. $\theta = UP \rightarrow \theta' = RIGHT$

2.3.4 Pick key

For the PK, we have the transition model as $f(s, PK) = s' = (k', \cdot)$ where all other elements in the input state s will stay the same. For the k', we have rules that $k = 0 \wedge (p + \theta) = p^k \rightarrow k' = 1$, which means agent can only pick the key if the front position is key position and agent hasn't carried the key yet.

2.3.5 Unlock door

For the UD, we have the transition model as $f(s, UD) = s' = (d', \cdot)$ where $d' = \{d'_1, d'_2, \dots d'_{n_d}\}$ and all other elements in the input state s will stay the same. Since agent can only open 1 door at each time, we have following transition rules: $(p + \theta) = p_i^d \wedge d_i = 0 \wedge k = 1 \rightarrow d'_i = 1$, which means agent can only open the door i if the front position is p_i^d, agent must carry the key, and door i hasn't been opened yet.

2.4 Cost

For the states transition cost, in my formulation and implementation, $l(s, MF) = l(s, TL) = l(s, TR) = l(s, PK) = l(s, UD) = c \in R$ where $c > 0$ and I set cost for all transitions as the same. For the termination cost $q(\tau) = c' \in R$ where $c' < 0$ and terminal state τ is defined as any state with $p = p^g$.

2.5 Planning horizon

For the planning horizon T, according to the DP on DSP problem, we have $T = |V|$ where V is the set of vertices. In this problem, I set $T = |S|$ where $|S|$ is number of states.

2.6 Objectives

In this DOC problem, we have objectives

$$\min_{u_0:T-1} q(\tau) + \sum_{t=0}^{T-1} l(s_t, u_t)$$

such that $s_{t+1} = f(s_t, u_t)$ and $s_t \in S, u_t \in \mathcal{U}$.

3 Technical Approach

3.1 Transfer to DSP

As we discussed in class, the finite-state DSP problem is equivalent to a finite-horizon finite-state DOC problem. Therefore, I choose to first convert DOC problem to DSP problem by building graph $G = (V, E)$, then run forward DP to solve it. For the vertices V, I combined the original state with timestep t to form each vertex $v = (t, s_t)$. For the edges E, any two vertices a, b ($a = (t, s_t)$, $b = (t + 1, s_{t+1})$) have a edge $(a, b) \in E$ with edge weight c if $b = f(a, u_t)$ where $u_t \in \mathcal{U}$. If we have invalid transition from a to b, I put a edge weight of ∞. In the implementation, since I applied the FDP, I only constructed the V and use the transition to determine edge and edge weights.

3.2 Forward Dynamic Programming

For both problem, they applies the same forward dynamic programming algorithm which is shown below.

Algorithm Deterministic Shortest Path via Forward Dynamic Programming

Input: vertices $\mathcal{V}$, start $s \in \mathcal{V}$, goal $\tau \in \mathcal{V}$, and costs c_{ij} for $i, j \in \mathcal{V}$
$T = |\mathcal{V}| - 1$
$V_0^F(s) = V_1^F(s) = \ldots V_T^F(s) = 0$
$V_0^F(j) = \infty, \quad \forall j \in \mathcal{V} \setminus \{s\}$
$V_1^F(j) = c_{s,j}, \quad \forall j \in \mathcal{V} \setminus \{s\}$
for $t = 2, \ldots, T$ **do**
    $V_t^F(j) = \min_{i \in \mathcal{V}} \left(c_{i,j} + V_{t-1}^F(i) \right), \quad \forall j \in \mathcal{V} \setminus \{s\}$
    if $V_t^F(i) = V_{t-1}^F(i), \forall i \in \mathcal{V} \setminus \{s\}$ **then**
        break

Figure 1: FDP

My implementation strictly follows above algorithm. At the end of the algorithm, it would return **two dictionaries**: *Value* and *Policy*. The keys of both dictionary are the vertices $v = (t, s_t)$. One modification I made is that *Policy* dictionary stores the optimal u_t for vertex (t, s_t) instead of parent nodes $(t-1, s_{t-1})$.

3.3 Query Action Sequence

After computing *Value* and *Policy* dictionaries, I built the optimal action sequence in backward manner. Below is the pseudo code of this process.

Algorithm 1 Query Action Sequence

function QUERYACTION(env, s_0)
    Initialize $best\_cost \leftarrow \infty$, $best\_traj \leftarrow$ None
    for all $(node, value)$ in Value Dict **do**
        if $v = \infty$ **then** ▷ Invalid transition
            continue
        end if

        Check If node satisfy Goal Condition (potential goal node)

        $path \leftarrow [\,]$, $cur \leftarrow node$, $ok \leftarrow$ True
        while $cur[time] > 0$ **do**
            $enc \leftarrow$ ENCODENODE(cur)
            if $cur \notin$ Policy Dict **then**
                $ok \leftarrow$ False; **break**
            end if
            $u \leftarrow$ policy[cur]
            Append u to $path$
            $cur \leftarrow$ REVERSETRANSITION(cur, a)
        end while
        if ok and $cur = s_0$ and $value < best\_cost$ **then**
            $best\_cost \leftarrow value$
            $best\_traj \leftarrow$ REVERSE($path$)
        end if
    end for
    if $best\_traj$ is None **then**
        return [ ]
    end if
    return $best\_traj$
end function

3.3.1 Find Potential goal vertex

The first part of the code include find a potential goal vertex $v_g = (t, p_{v_g}, \theta_{v_g}, \dots)$ from all state stored in $Value$ dict. For the **Known Map** problem, it check if the goal vertex has the same position as goal position $p_{v_g} = p^g$. For the **Unknown Map**, we also need to check if v_g has the same key position as the key position in initial environment. In addition, if two doors in initial environment is closed $d_{initial} = [0, 0]$, then during search it only consider goal vertex with $d_{v_g} \neq [0, 0]$ since we need at least 1 door open to reach goal.

3.3.2 Construct Path

Here I write the reverse transition to construct the path from v_g backward, where reverse transition is $s_t = f_{reverse}(s_{t+1}, u_t)$ which based on the transition model described earlier.

3.3.3 Check Valid Path

Lastly, it check if the end of the path reach the initial state of the environment. If not we could not accept this path. It also keep track of the best path and best value we found so far in order to find the optimal action sequence.

3.4 Workflow

For the **Known Map** problem, the workflow is

1. Load env, use env to initialize initial state s_0
2. Compute FDP (use env enforce transition rules)
3. Query action sequence

In this workflow, we directly use the env to initialize initial state and enforce transition rules (such as check if MF hit the wall). This workflow will generate 7 different $Policy$ dict and action sequence for 7 different envs.
For the **Unknown Map** problem, the workflow is

1. Check if there exist precomputed $Policy$ and $Value$ dict (in the /**output** folder)
2. If not, Compute FDP to get $Policy$ and $Value$ dict
    a. Initialize state space S including all states (all different key position p^k, door conditions d, goal position p^g), initial state s_0 also needs to consider all possible configurations.
    b. This time, since we don't load env, we need to write functions to explictly enforce transition rules
3. Load env, use env to initialize initial state s_0
4. Query action sequence based on $Policy$, $Value$ dict and s_0

In this workflow, the FDP would compute a single policies that works for all possible unknown maps. After FDP, I stored the $Policy$ and $Value$ dict, and load them along with env to directly query action sequence.

4 Reulsts

In this section, I will present results include

1. Initial Environment
2. Plotted trajectory
3. Action sequence

For **Known Map** problem, I included results for all map. However for the **Unknown Map** problem, there are 36 of them, I randomly loaded 8 of them and present the results of them (**Based on my testing, my implementation could successfully solve any of them**). The trajectory plots I presented in below can be found at /**traj** folder of project. I also generated gif for them, under /**gif** folder.

4.1 Known Map

4.1.1 5x5 normal

Action Sequence: $[1, 1, 3, 2, 4, 0, 0, 2, 0]$

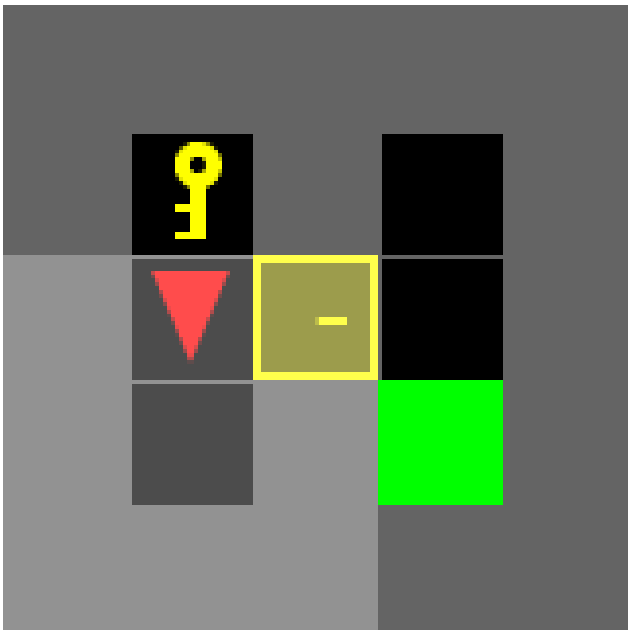

Figure 2: Initial Env

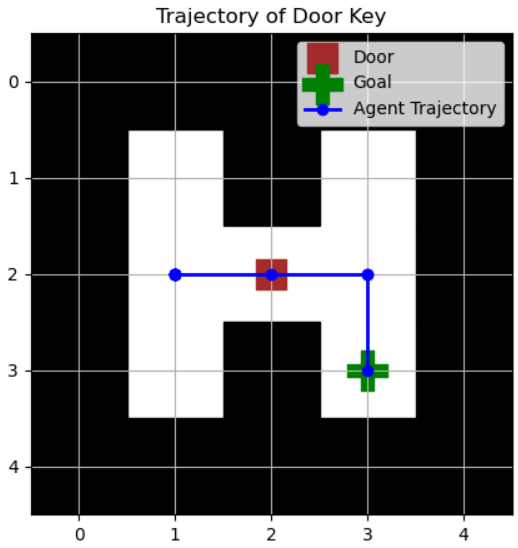

Figure 3: Trajectory

4.1.2 6x6 normal

Action Sequence: $[1, 0, 3, 1, 1, 0, 2, 0, 4, 0, 0, 2, 0]$

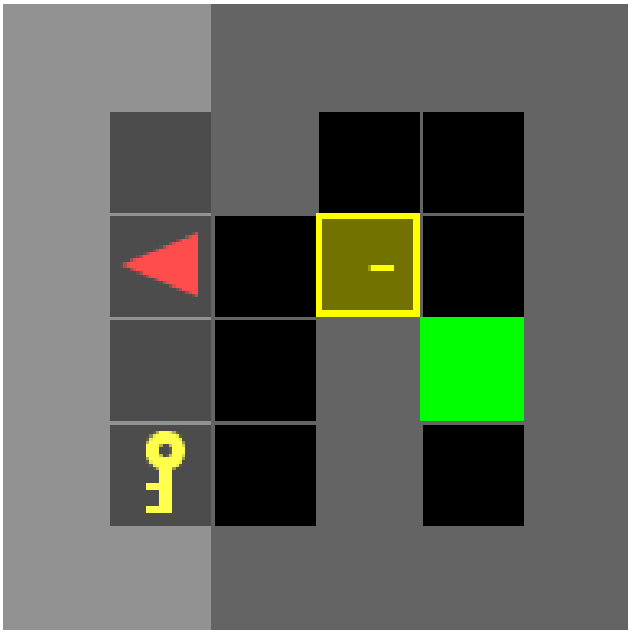

Figure 4: Initial Env

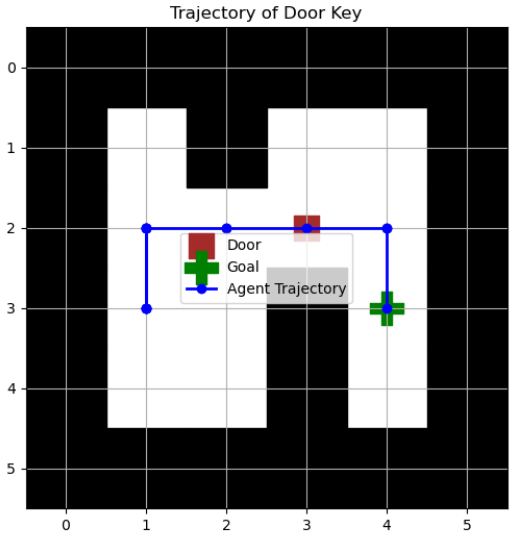

Figure 5: Trajectory

4.1.3 6x6 direct

Action Sequence: $[0, 0, 2, 0, 0]$

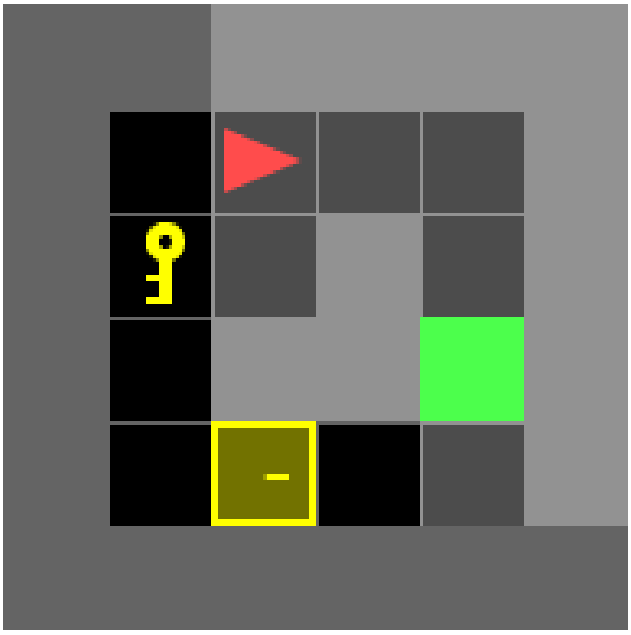

Figure 6: Initial Env

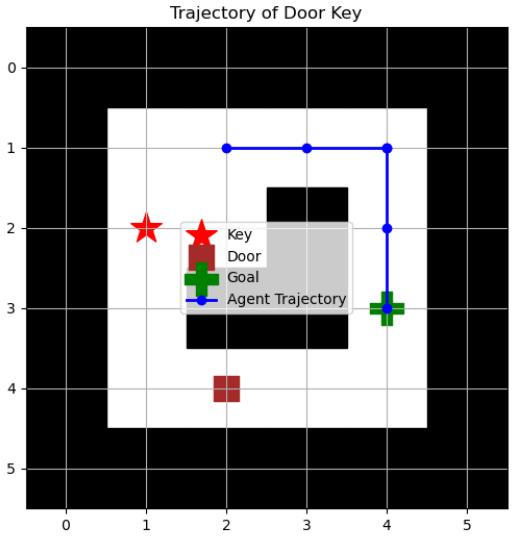

Figure 7: Trajectory

4.1.4 6x6 shortcut

Action Sequence: $[3, 1, 1, 4, 0, 0]$

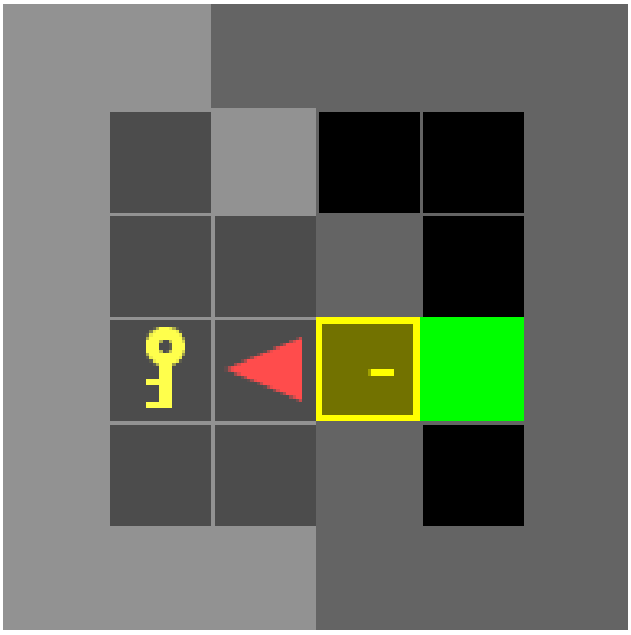

Figure 8: Initial Env

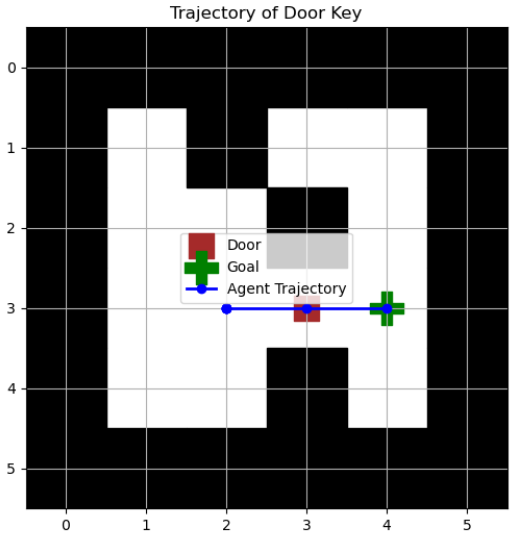

Figure 9: Trajectory

4.1.5 8x8 normal

Action Sequence: $[2, 0, 1, 0, 2, 0, 0, 0, 3, 1, 1, 0, 0, 0, 2, 4, 0, 0, 0, 2, 0, 0, 0]$

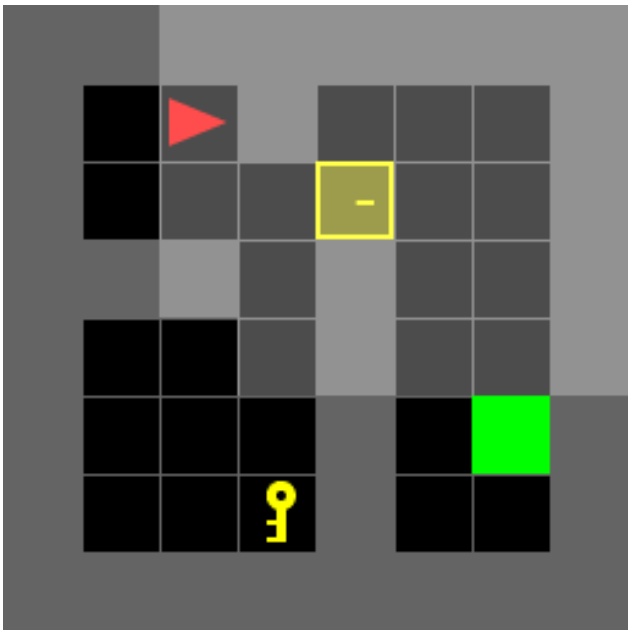

Figure 10: Initial Env

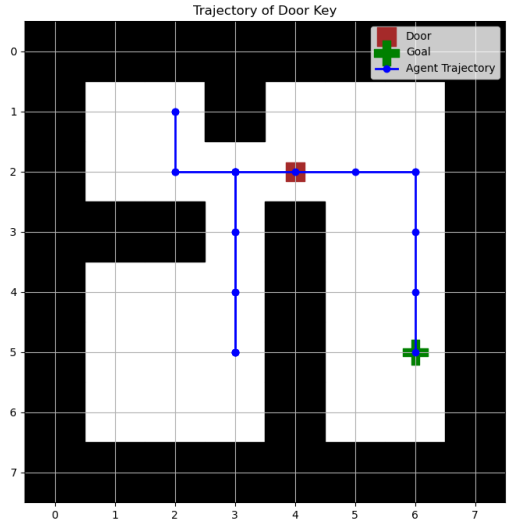

Figure 11: Trajectory

4.1.6 8x8 direct

Action Sequence: $[0, 1, 0, 0, 0, 1, 0]$

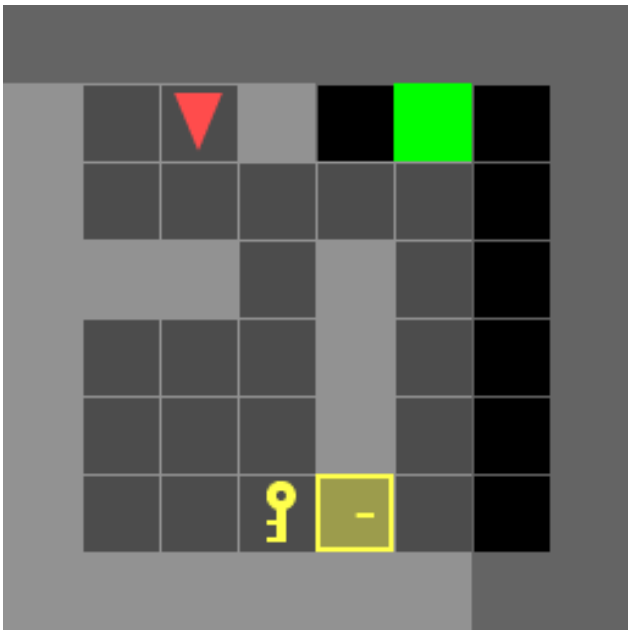

Figure 12: Initial Env

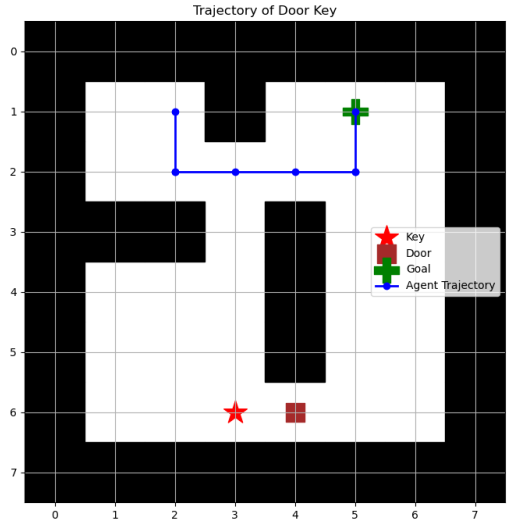

Figure 13: Trajectory

4.1.7 8x8 shortcut

Action Sequence: $[2, 0, 2, 3, 1, 4, 0, 0]$

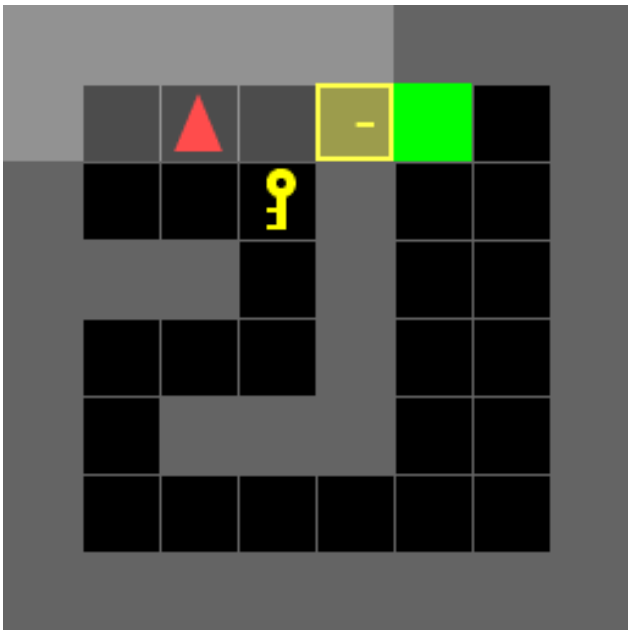

Figure 14: Initial Env

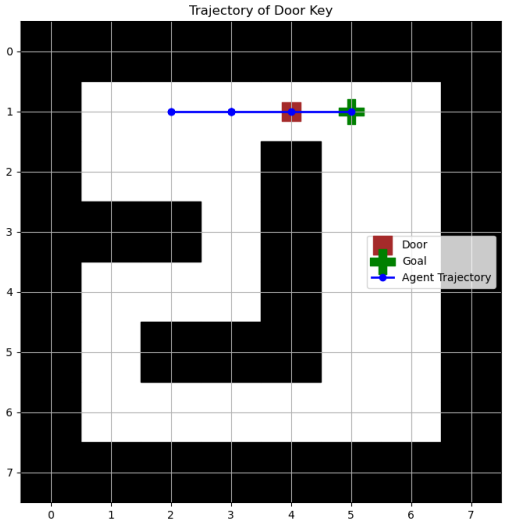

Figure 15: Trajectory

4.2 Unknown Map

4.2.1 Random 1

Action Sequence: $[0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0]$

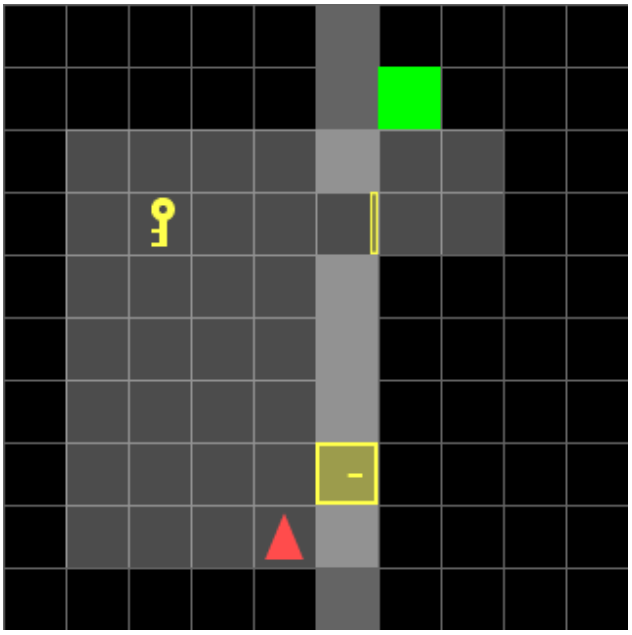

Figure 16: Initial Env

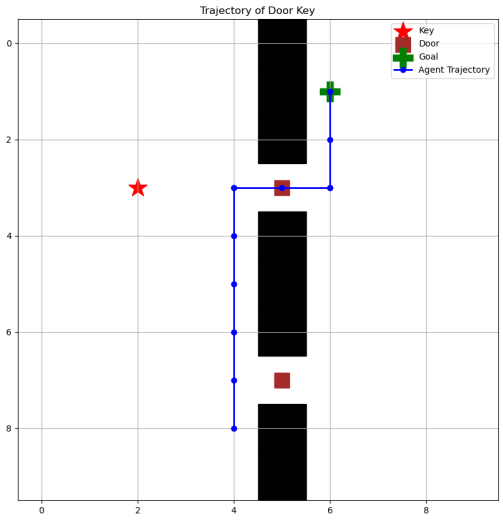

Figure 17: Trajectory

4.2.2 Random 2

Action Sequence: $[0, 0, 0, 0, 0, 2, 0, 0, 1, 0, 0]$

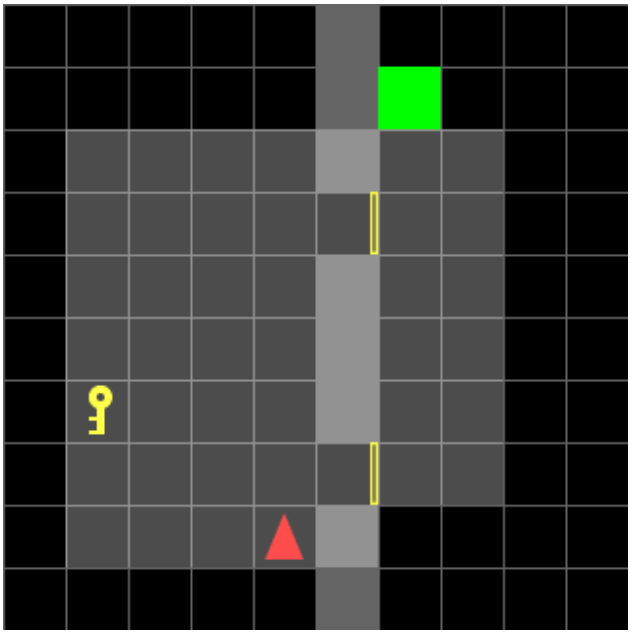

Figure 18: Initial Env

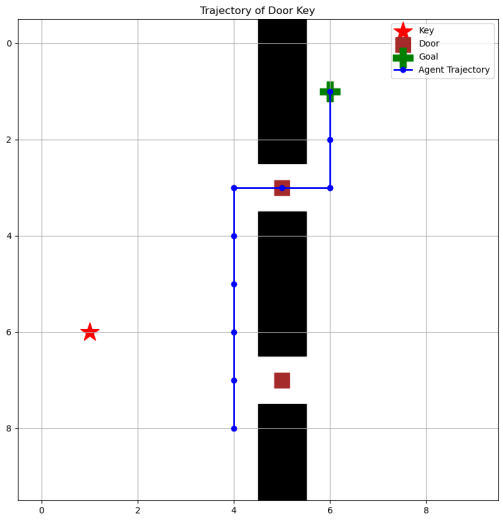

Figure 19: Trajectory

4.2.3 Random 3

Action Sequence: $[0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 0]$

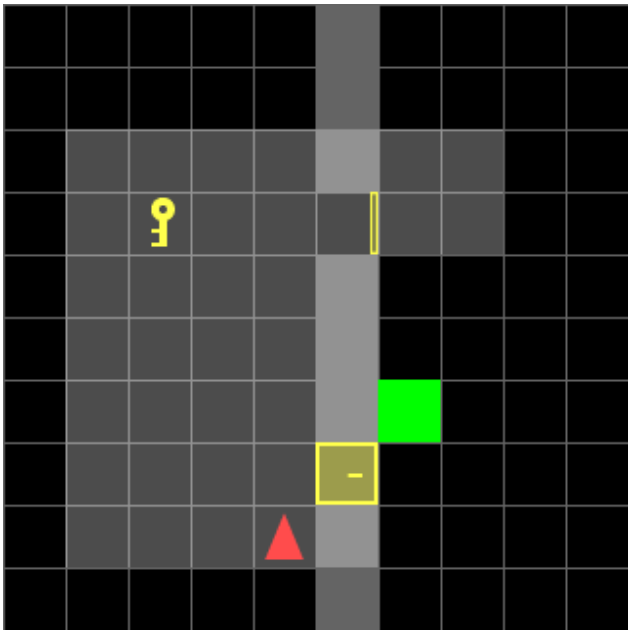

Figure 20: Initial Env

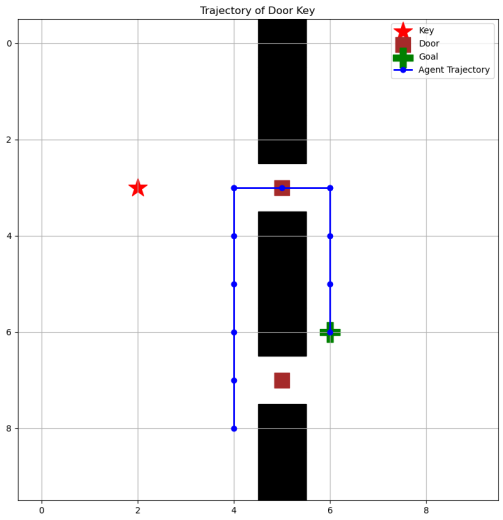

Figure 21: Trajectory

4.2.4 Random 4

Action Sequence: $[0, 0, 0, 0, 0, 1, 0, 3, 1, 1, 0, 4, 0, 0, 2, 0, 0, 0]$

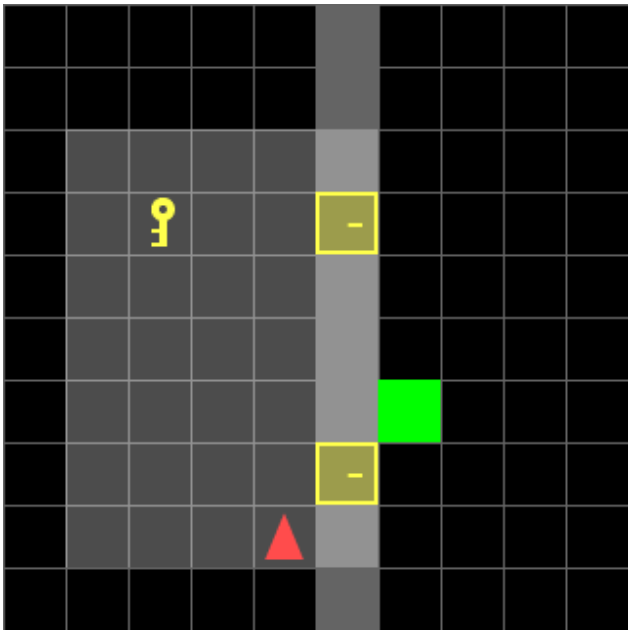

Figure 22: Initial Env

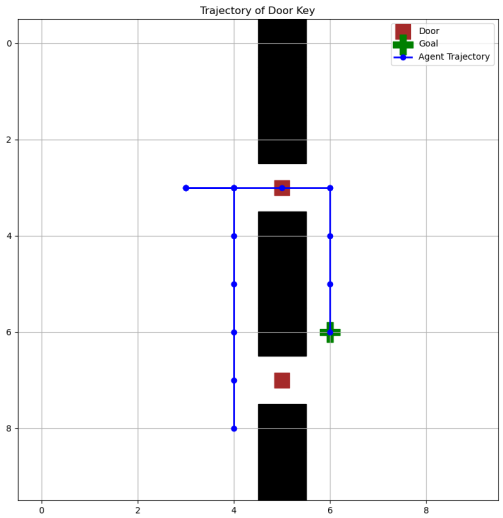

Figure 23: Trajectory

4.2.5 Random 5

Action Sequence: $[0, 2, 0, 0, 1, 0]$

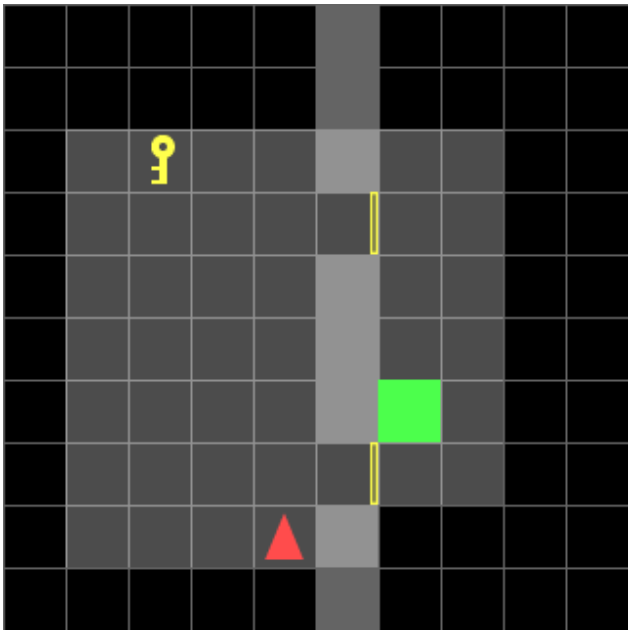

Figure 24: Initial Env

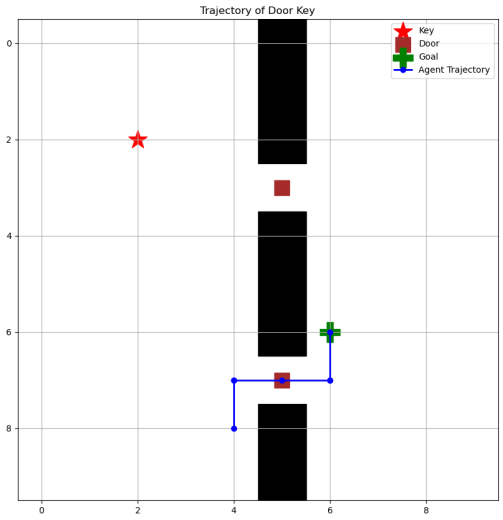

Figure 25: Trajectory

4.2.6 Random 6

Action Sequence: $[0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0]$

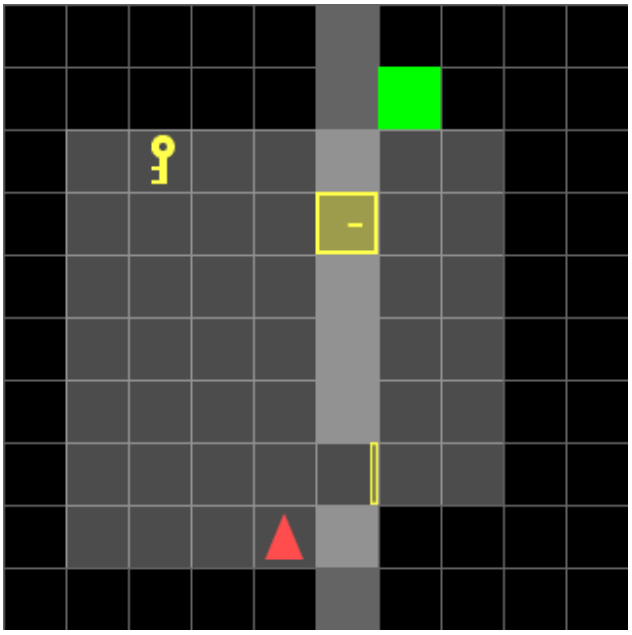

Figure 26: Initial Env

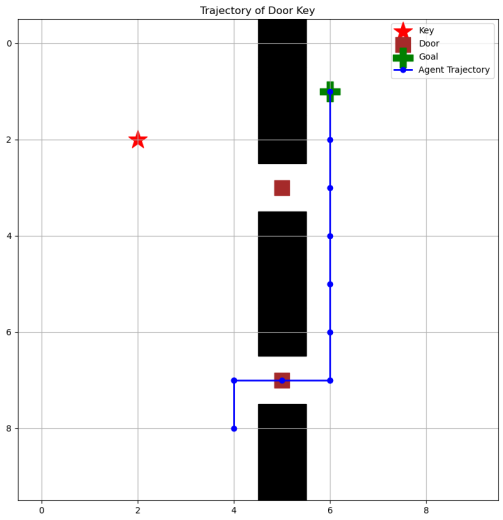

Figure 27: Trajectory

4.2.7 Random 7

Action Sequence: $[0, 0, 0, 0, 0, 1, 0, 0, 2, 3, 2, 0, 0, 4, 0, 0, 2, 0, 0, 0]$

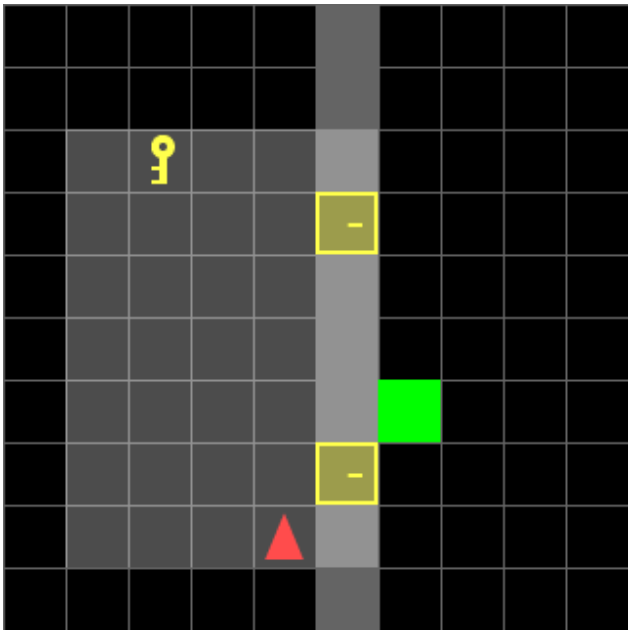

Figure 28: Initial Env

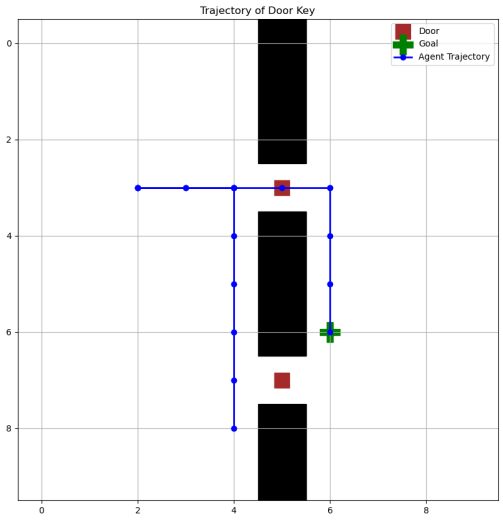

Figure 29: Trajectory

4.2.8 Random 8

Action Sequence: $[0, 0, 1, 0, 0, 3, 2, 0, 0, 0, 2, 0, 0, 4, 0, 0, 0]$

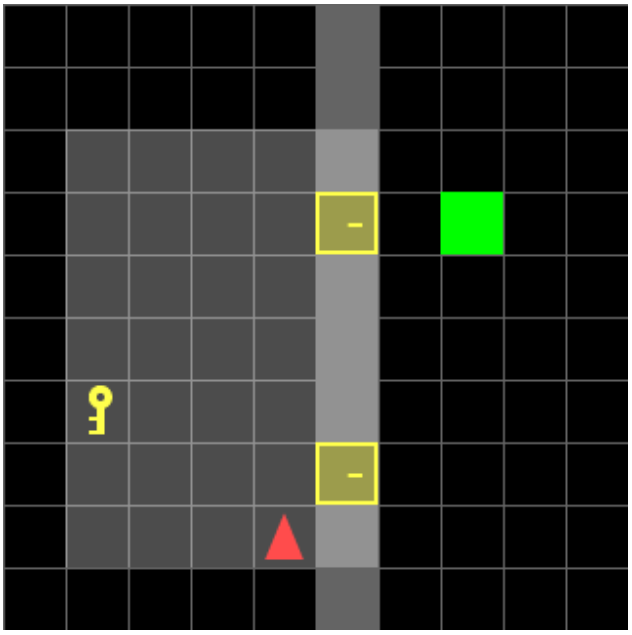

Figure 30: Initial Env

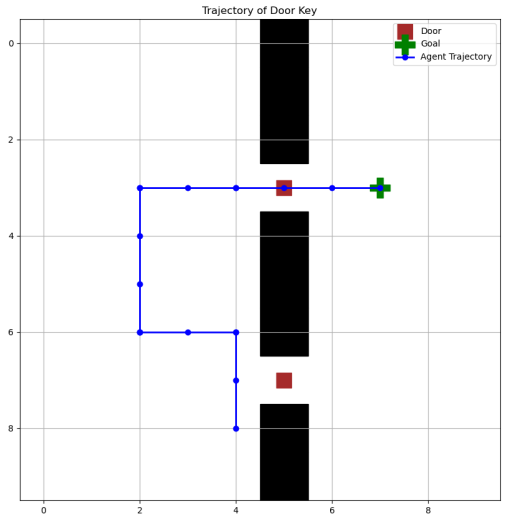

Figure 31: Trajectory

4.3 Performance

The FDP on DSP algorithm I applied is a useful tools to find optimal control sequence in the DOC problem. From the Known map problem part, the DP algorithm could successfully generate different optimal policy under different environment. From the Unknown map problem, we can use DP to compute a single "universal" policy that could work on all random maps. Furthermore, we could store the universal policy and query it at runtime to determine the optimal action at each state, without the need for online re-computation.

However, this method is not an efficient method as it needs to iterate through all states and control

inputs per iteration. For known map problem, since we don't need to include possible key position and goal position in state, and we have moderate state space size, which comes with a relatively fast computation. However, for the unknown map problem, as I mentioned the size of the state space is around $|S| \approx 26496$ plus we have $T = |S|$. This large state space and time horizon would make DP algorithm inefficient since we have $O(|S|T)$ for DP. In implementation, I only use $T = 300$ for unknown map. In addition, even after I store the *Policy* dict for direct query, the query process would also take some times due to large state space.

In addition, this method only works on DOC problems where we have finite T and finite states. If we have infinite states or T, we can no longer explicitly enumerate all possibilities to update the value function and policy. Moreover, if the environment involves stochastic transitions rather than deterministic ones, the transition probabilities must also be incorporated into the value update process. This requires modifying the algorithm to account for expected values over possible next states.