ECE276B PR1 Report

1st Weixiao Zhan
weixiao-zhan[at]ucsd[dot]edu

I. Introduction

The Door & Key problem is a type of deterministic Markov process, where an agent navigates to through a grid of cells to reach certain goal position. Along the way, the agent needs to avoid obstacles (walls in our environment), retrieve a key and use it to open doors. The scenario is applicable in robotics, particularly in environments like warehouses where robots perform multi-step tasks to retrieve items and avoid obstacles and other robots.

The deterministic optimal control (DOC) policy of Door & Key problem can be found via dynamic programming (DP). The DP program evaluates and optimizes control decisions at all states, achieving efficient computation of DOC policy and their corresponding cost.

II. Problem Formulation

Markov Decision Process can be described as the collection of state space $\mathcal{X}$, control space $\mathcal{U}$, motion model f, time horizon T, stage cost l, and terminal costs q.

Denote the DOC policy and value function as:

$$\pi_t(x) = u \Big| x \in \mathcal{X}$$
$$V_t(x) \in \mathbb{R} \Big| u \in \mathcal{U}$$

A. state space

The environment of Door & Key problem is a grid (GD), which has certain width and height. Each cell in the grid can either be free space (FS) or occupied by wall (WL). Formally:

$$GD = \left\{ \begin{bmatrix} x \\ y \end{bmatrix} \Big| \begin{matrix} \forall x \in [0, \text{width}) \\ \forall y \in [0, \text{height}) \end{matrix} \right\}$$
$$WL \subset GD$$
$$FS = GD \setminus FS$$

State $x \in \mathcal{X}$ is defined as the collection of agent position (ap), agent direction (ad), goal position (gp), key position (kp), key carrying status (ks), doors position (dp), and doors open status (ds).

$$x = \begin{pmatrix} ap & ad & gp \\ kp & ks & \\ dp & ds & \end{pmatrix}$$

$$\mathcal{X} = \left\{ x \;\middle|\; \begin{matrix} ap, gp, kp \in FS \\ ad \in \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right\} \\ ks \in \{0, 1\} \\ dp \in FS^n \\ ds \in \{0, 1\}^n \end{matrix} \right\}$$

in which, n is the number of doors and inherent in state; $ks = 1$ means agent is carrying the key; and $ds[i] = 1$ means i-th door is open.

B. control space

There are 5 actions the agent can take:

$$\mathcal{U} = \begin{cases} MF : \text{move forward} \\ TL : \text{turn left} \\ TR : \text{turn right} \\ PK : \text{pick up key} \\ UD : \text{unlock the door} \\ ST : \text{stay at goal location} \end{cases}$$

C. motion model

Consider state $x \in \mathcal{X}$ taking action $u \in \mathcal{U}$. The motion model is:

1) $u = MF$: The agent moves forward 1 cell in agent direction. This action is only valid when the front cell is within boundaries and not wall nor closed doors (ensured on Algorithm 1 line 4).
2) $u = TL$: The agent turn 90 degree in counter-clock direction.
3) $u = TR$: The agent turn 90 degree in clock direction.
4) $u = PK$: The agent pick up key in front cell. This action is only valid when the front cell is the key position and the agent hasn't already picked the key. (ensured on Algorithm 1 line 13)
5) $u = UD$: The agent unlock the door in front cell. This action is only valid when the agent is carrying the key and front position is a closed door. (ensured on Algorithm 1 line 16)
6) $u = ST$: The agent should be able to stay at the goal cell. (ensured on Algorithm 1 line 20).

D. stage cost

Stage cost is defined on state-control combinations, whose motion models are valid.

$$l(x, u) = \begin{cases} 2 & u = MF \\ 1 & u = TL \\ 1 & u = TR \\ 1 & u = PK \\ 1 & u = UD \\ 0 & u = ST \end{cases}$$

E. terminal cost

$$q(x) = \begin{cases} 0 & x.ap = x.gp \\ \infty & o/w \end{cases}$$

F. time horizon

Markov assumption bounds the length of DOC sequence no greater than the number of states.

$$T < |\mathcal{X}|$$

G. backward DP

In Door & Key DP, the initial values are:

$$V_T(x) = q(x), \forall x \in \mathcal{X}$$

State transition equations is:

$$\left.\begin{cases} Q_t(x,u) = l(x,u) + V_{t+1}(f(x,u)) \\ V_t(x) = \min_u Q_t(x,u) \\ \pi_t(x) = \arg\min_u Q_t(x,u) \end{cases}\right\}, \forall t \in [0,T)$$

The backward DP program starts at $t = T$ and works it way to $t = 0$. Meanwhile, backward DP can terminate early if the value function of current time step is the same as the one of next time step. This ensures the DP has find shortest DOC policy of all states that can reach the goal state.

III. Technical Approach

The `solver.py` implements a dynamic programming-based solution to the Door & Key problem. The core of the implementation lies within two main classes: `State` and `DoorKeySolver`.

A. hash-able `State` class

An helper class, `XY`, is defined to abstract 2D position and direction coordinates. It provides initialize, add (handy in computing forward position), multiply (handy in turning) and hash methods.

The `State` class is the abstraction of state x, and encapsulates: the position and direction of the agent, the position of the goal, the position of the key, whether the key is being carried, and the status (open/closed) of each door. Each state is uniquely identifiable and comparable based on the hash of its properties, allowing efficient state management and lookup.

```
Class XY
    x : int
    y : int
Class State
    agent_pos : XY
    agent_dir : XY
    goal_pos : XY
    key_pos : XY
    key_carrying : bool
    door_num : int
    door_pos : tuple[XY]
    door_is_open : tuple[bool]
```

B. abstract `DoorKeySolver` classes

`DoorKeySolver` class is an abstract solver class for the Door & Key problem. It implements three functions that are intrinsic of the problem and four functions to implement the backward DP algorithm over a finite time horizon.

```
Class DoorKeySolver:
    # intrinsic
    is_wall(pos: XY) -> bool
    motion_model(state: State, control)
        -> tuple[State, int]
    terminate_cost(state:State) -> int
    # DP
    iter_state_space() -> Generator[State]
    iter_control_space() -> Generator[int]
    solve(time_horizon)
    query(state: State)
```

Algorithm 1 implements the motion model and stage cost, described in section II-C and II-D. The new state and the stage cost are return together if the action is valid.

Algorithm 1 motion model and state cost

procedure (x, u)
  $fp \leftarrow x.ps + x.dir$ ▷ forward position
  $x' \leftarrow copy(x)$
  if $u = MF \wedge fp \in FS \wedge (!\exists i$ s.t. $fp = x.dp[i] \wedge x.ds[i] = 0)$ **then**
    $x'.ap \leftarrow x.ap + x.ad$
    return $x', 2$
  else if $u = TL$ **then**
    $x'.ad \leftarrow \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} x.ad$
    return $x', 1$
  else if $u = TR$ **then**
    $x'.ad \leftarrow \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} x.ad$
    return $x', 1$
  else if $u = PK \wedge fp = x.kp \wedge x.ks = 0$ **then**
    $x'.ks \leftarrow 1$
    return $x', 1$
  else if $u = UD \wedge x.ks = 1 \wedge (\exists i$ s.t. $fp = x.dp[i] \wedge x.ds[i] = 0)$ **then**
    $x'.ks = 0$
    $x'.ds[i] = 1$
    return $x', 1$
  else if $u = ST \wedge x.ap = x.gp$ **then**
    return $x', 0$
  end if
  return $\emptyset, \infty$ ▷ non-valid state-control combination
end procedure

Algorithm 2 implements `solve`, the main backward DP logic.

Algorithm 3 implements `query`, which build control sequence from certain initial state.

Algorithm 2 backward DP

procedure SOLVE(T)
  $V \leftarrow$ dictionary()
  $\pi \leftarrow$ dictionary()
  ▷ set init values
  $V[T] \leftarrow$ dictionary()
  for all x in iter_state_space() **do**
    $V[T][x] \leftarrow$ terminate_cost(x)
  end for
  ▷ backward DP
  for $t \leftarrow [T-1, -1, -1]$ **do**
    $V[t] \leftarrow$ dictionary()
    $\pi[t] \leftarrow$ dictionary()
    for all x in iter_state_space() **do**
      $best\_cost \leftarrow \infty$
      $best\_control \leftarrow$ None
      for all u in iter_control_space() **do**
        $x', l \leftarrow$ motion_model(x, u)
        if $x' \neq$ None **then**
          $q \leftarrow l + V[t+1][x']$
          if $q < best\_cost$ **then**
            $best\_cost \leftarrow q$
            $best\_control \leftarrow u$
          end if
        end if
      end for
      if $best\_control \neq$ None **then**
        $V[t][x] \leftarrow best\_cost$
        $\pi[t][x] \leftarrow best\_control$
      end if
    end for
    if $V[t] = V[t+1]$ **then**
      break
    end if
  end for
  self.$V \leftarrow V$
  self.$\pi \leftarrow \pi$
end procedure

Algorithm 3 query

procedure QUERY(x)
  controls $\leftarrow$ empty list
  for $t = 0$ to T **do**
    $u \leftarrow \pi[t][x]$
    if best_control = ST **then** ▷ skip stay action
      break
    end if
    controls.append(u)
    $x \leftarrow f(x, u)$
  end for
  return controls
end procedure

C. specific implementations

Two implementations of `DoorKeySolver` are provided, They over-write `is_wall` and `iter_state_space` functions to tailor different grid environments and problem setups, demonstrating the solver's adaptability to various configurations.

`DoorKeySolver_1` class is adapted for one door and known environment setup (part A).

Specifically, `is_wall` function directly query the cell type from `pygym.env`. And `iter_state_space` would set the goal position, key position, door position and status based on `pygym.env` info.

`DoorKeySolver_2` class handles a more complex scenario (part B). The size of the grid is 8×8 and the perimeter is surrounded by walls. There is a vertical wall at column 4 with two doors at $(4, 2)$ and $(4, 5)$. Each door can either be open or locked (requires a key to open). The key is randomly located in one of three positions $(1, 1), (2, 3), (1, 6)$. The goal is randomly located in one of three positions $(5, 1), (6, 3), (5, 6)$.

specifically, `is_wall` determines if a cell is free space by checking its position coordinates. `iter_state_space` iterates through all 36 possible environments and all possible agent position, direction, and key carrying states.

IV. Results

Figure 1 and 2 shows the optimal control sequence of known environments. Figure 3 shows the optimal control sequence of one realization in the random environment. Full results are present in `envs/known_envs/*.gif` and `envs/random_env/*.gif`.

Figure 4 shows the number of states being considered at each time steps. As the environment gets bigger, and less knowledge of the environment setup, the number of states increase rapidly. The graph also shows after some steps, the DP has computed a policy for every state that can reach the goal state, and terminate early.

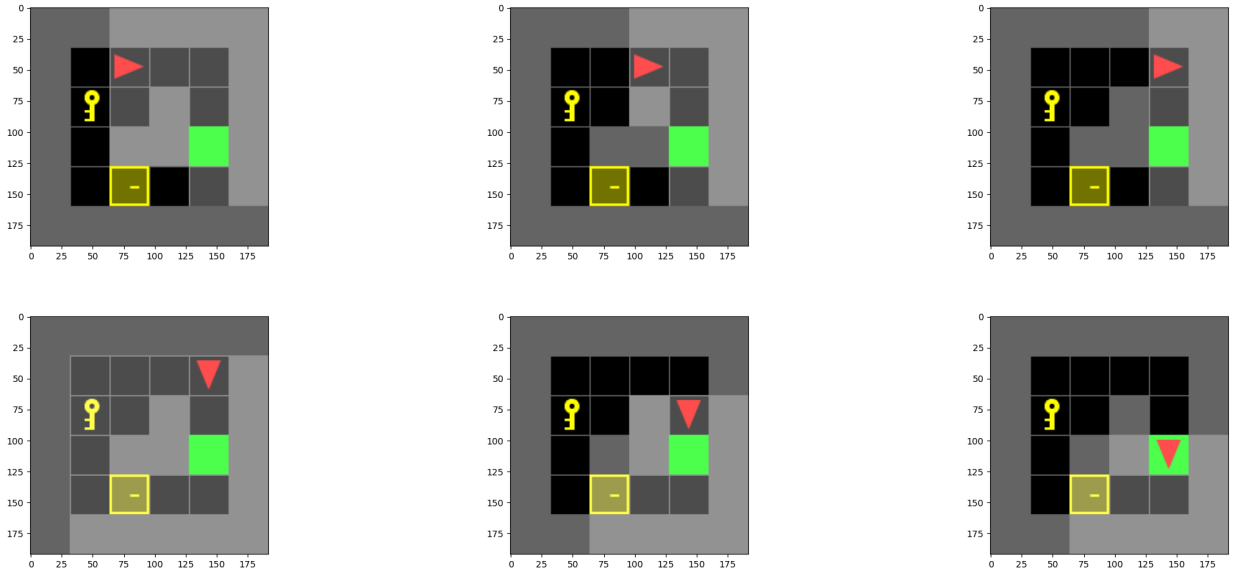


Fig. 1: optimal control sequence of 6x6 direct environments

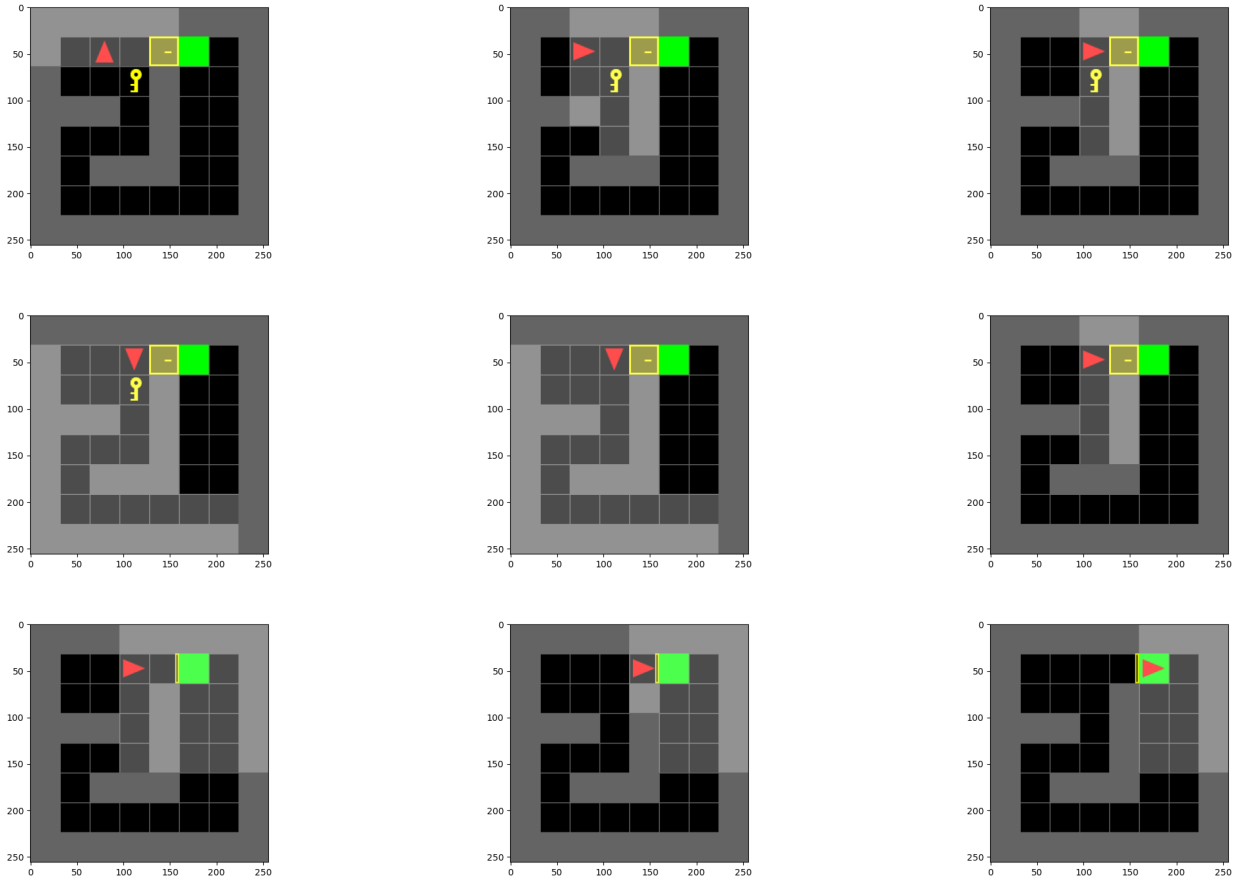


Fig. 2: optimal control sequence of 8x8 shortcut environments

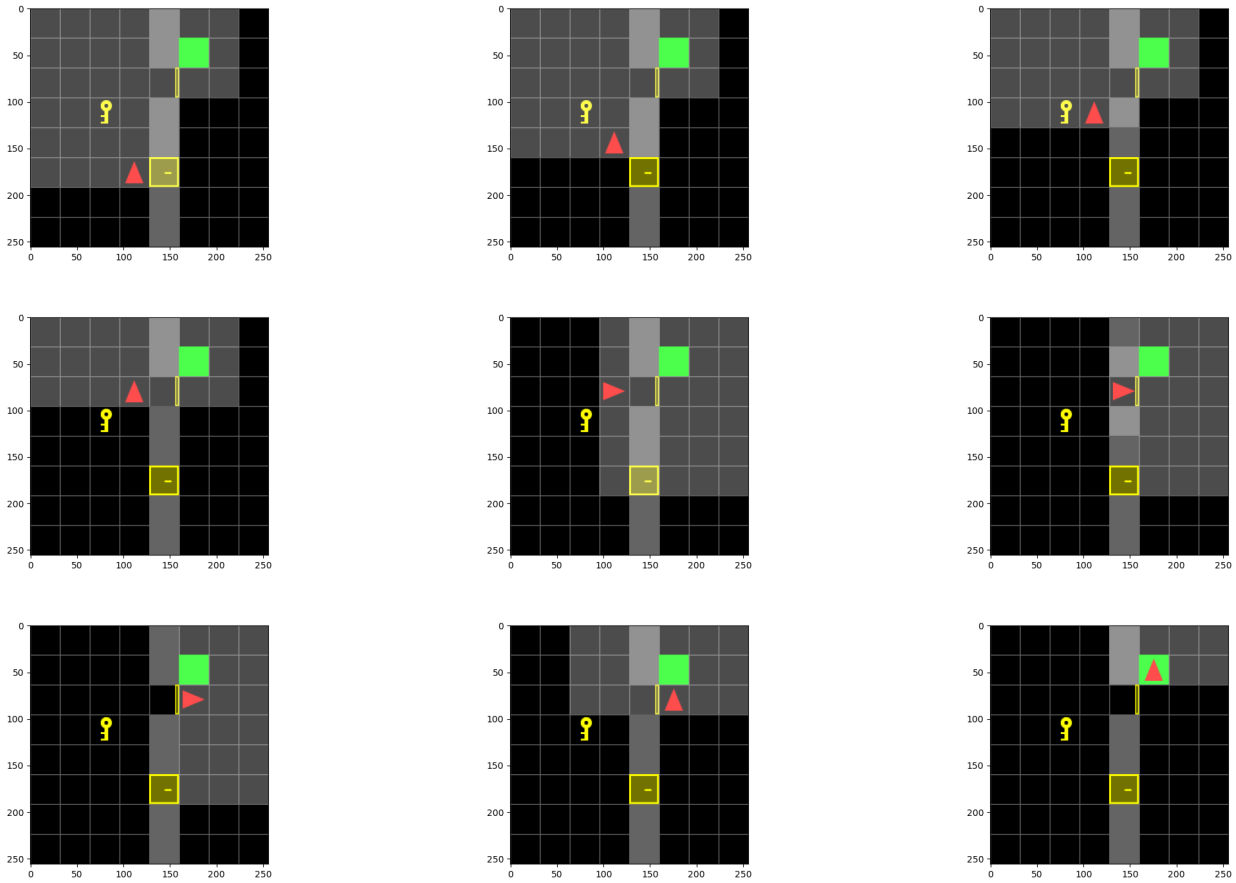
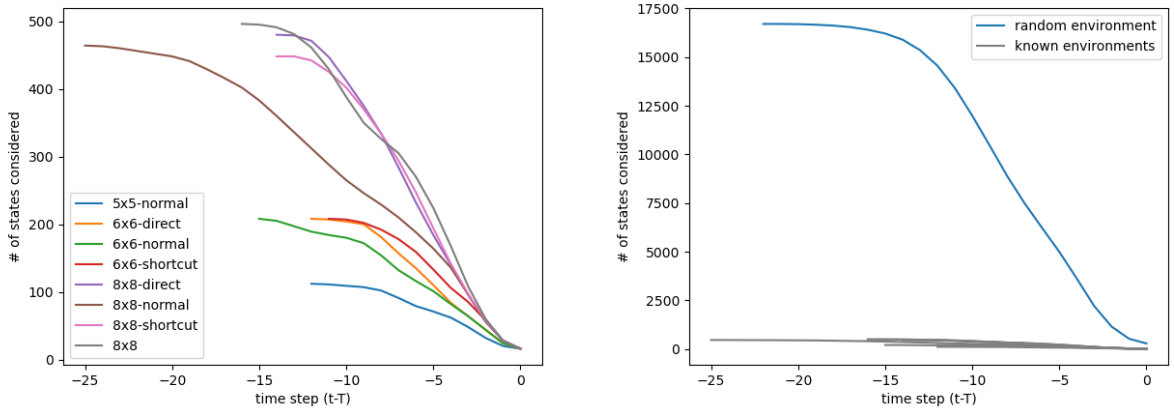


Fig. 3: optimal control sequence of one random environments



(a) known environments

(b) random environment

Fig. 4: number of state been considered at each time steps