Project 1: Markov Decision Processes and Dynamic Programming

Aditya Mishra
dept. of Mechanical and Aerospace Engineering
University of California San Diego
San Diego, United States of America
amishra@eng.ucsd.edu

***Abstract*—This document contains my approach to the door key problem which is solved using the label correcting algorithm**
***Index Terms*—label correction, dynamic programming, Markov Decision Process**

I. Introduction

A. Introduction to the Problem

The problem statement consists of an N×N grid where the objective is to find the shortest path for the agent from its initial state to the cell which has the goal. The environment consists of few elements which act as an obstacle for the agent to reach the goal cell. These are wall, key and door. The door can be only unlocked if the key is picked by the agent. The only actions available to the agent are : Move Forward, Turn Left, Turn Right, Pick Key and Unlock the Door.

B. Potential applications

In the field robotics this be applied to driver-less cab-sharing services where the passengers of the cab are supposed to be dropped in a particular sequence. Also, this can be used in supply chain sector. For instance the key can be the storage facility where the delivery package is present, the door can symbolize the quality assurance center where the robot has to carry the package to first and the goal can be the delivery address.

C. My approach to the problem

Using the Label Correcting (lc) Algorithm, I initially calculate the shortest path from the initial state of the agent to the goal, regardless of the position of the door. If the door is not present in the shortest path then this is the optimal path. If the door is present then two alternate paths are calculated. The first path is the door-key path. Here the problem is divided into three parts: initial state to the key, key to the door and door to the goal. From the previous shortest path, states from the door to the end goal are taken which solve the third part of the problem. The first two parts are solved with separate lc algorithms with the costs being reinitialized again. The final policy is the combination of all three sequences of action. The second alternate path is the one which finds the shortest path from the initial state to the goal which does not have the door in the way. This is again done by using lc algorithm. Costs of both of these paths are compared and the least cost path is chosen to be the shortest path.

II. Problem Statement

A. State Space

The state x is a vector consisting the location of the agent in terms of cell coordinate, the direction of the agent, is key picked and is door locked.

$$x = \begin{bmatrix} \alpha \\ \theta \\ \beta \\ \gamma \end{bmatrix} \text{ where, } \alpha \in \mathbb{R}^2 \tag{1}$$

$$\text{and } \theta \in \begin{bmatrix} (1,0) & (0,1) & (-1,0) & (0,-1) \end{bmatrix}$$

$$x \in \chi \ \forall x$$

Here α denotes the cell location which can be traversed, i.e. the cells which are not walls. θ denotes the direction of the agent. The state space χ is the collection of all such x for a particular map. The states

$$\beta, \gamma \in \begin{pmatrix} 0 & 1 \end{pmatrix}$$

If the key is picked by the agent then $\beta = 1$ else $\beta = 0$. Similarly, if the door is unlocked then $\gamma = 1$ else $\gamma = 0$.

B. Control Space

The control space $\mathcal{U}$ is the set of all actions that can be taken by the agent to reach the goal. All such actions are called the controls u.

$$\mathcal{U} = \begin{bmatrix} \text{Move Forward} \\ \text{Turn Right} \\ \text{Turn Right} \\ \text{Pick Key} \\ \text{Unlock Door} \end{bmatrix}$$

$$u \in \mathcal{U} \ \forall u$$

C. Motion Model

The action taken at time t is u_t and the state reached at time t is x_t. The motion model f can be defined as

$$x_{t+1} = f(x_t, u_t)$$

Referring to (1), the motion model can be formulated as the following
If $u_t =$ Move Forward, then

$$\begin{aligned} \alpha_{t+1} &= \alpha_t + \theta_t \\ \theta_{t+1} &= \theta_t \\ \beta_{t+1} &= \beta_t \\ \gamma_{t+1} &= \gamma_t \end{aligned}$$

If $u_t =$ Pick Key, then

$$\begin{aligned} \alpha_{t+1} &= \alpha_t \\ \theta_{t+1} &= \theta_t \\ \beta_{t+1} &= \beta_t + 1 \\ \gamma_{t+1} &= \gamma_t \end{aligned}$$

If $u_t =$ Unlock Door and $\beta_t = 1$, then

$$\begin{aligned} \alpha_{t+1} &= \alpha_t \\ \theta_{t+1} &= \theta_t \\ \beta_{t+1} &= \beta_t - 1 \\ \gamma_{t+1} &= \gamma_t + 1 \end{aligned}$$

Let us define a direction matrix as:

$$\mathcal{D} = \begin{bmatrix} (1,0) & (0,1) & (-1,0) & (0,-1) \end{bmatrix}$$

If $u_t =$ Turn Right and $\beta_t = 1$, then

$$\begin{aligned} \alpha_{t+1} &= \alpha_t \\ \theta_{t+1} &= \mathcal{D}\left[\mathcal{D}.\text{index}(\theta_t) + 1\right] \\ \beta_{t+1} &= \beta_t \\ \gamma_{t+1} &= \gamma_t \end{aligned}$$

If $u_t =$ Turn Left and $\beta_t = 1$, then

$$\begin{aligned} \alpha_{t+1} &= \alpha_t \\ \theta_{t+1} &= \mathcal{D}\left[\mathcal{D}.\text{index}(\theta_t) - 1\right] \\ \beta_{t+1} &= \beta_t \\ \gamma_{t+1} &= \gamma_t \end{aligned}$$

D. Initial State

$$x_0 = \begin{bmatrix} \alpha_0 \\ \theta_0 \\ 0 \\ 0 \end{bmatrix}$$

E. Planning Horizon

The state space formulation allows the goal cell to have 4 equivalent states . Similarly the agent will only enter the door cell in just one state and will not transition into another state which has the same α, i.e $\alpha_k \neq \alpha_M$, $\forall k$ if the agent reaches the door cell at step M. The same formulation holds if the agent reaches the goal at step T. Hence the Planning Horizon T can be formulated as:

$$T = |\chi| - 6$$

where, $|\chi|$ denotes the size of the state space set.

F. Terminal cost and Stage cost

for stage cost l and terminal cost q,

$$\begin{aligned} l(x_t, u_t) &= 1 \\ q(x_t) &= 0 \\ \forall x_t, u_t & \end{aligned}$$

III. Technical Approach

Initially all state costs g are assigned to infinity except the initial state, and a graph is created between the children and the parent nodes. Then ignoring the cells where the key and the door are located, the shortest direct path from the initial agent cell to the goal cell is constructed using the label correcting algorithm. Since the states costs are the same, the transition cost c_{ij} from state i to state j $\forall(i,j)$ are assigned the value 1, i.e $c_{ij} = 1$ $\forall(i,j)$.

Label Correcting Algorithm

$OPEN \leftarrow s$;
if i==s **then**
  $g_i = 0$;
else
  $g_i = \infty$;
  while OPEN is not empty do then **do**
    Remove i from OPEN
    for $j \in$ Children(i) **do**
      if $g_i + c_{ij} < g_j and < g_\tau$ **then**
        $g_\text{j} = g_i + c_{ij}$
        $Parent(j) = i$
      end if
      if $j \neq \tau$ **then**
        OPEN $\cup \{j\}$
      end if
    end for
  end while=0

After the shortest path is discovered, it is checked if the shortest path has a door in the way. The position of the door is recorded in a variable called door_pos. If the door is on the way, the problem finds two alternative paths: the first one is a path which assumes that door is the part of the wall and finds a path which does not have door on the way. The second path uses another label correcting algorithm to reach the states which lead to the key cell. After the key is taken it uses lc algorithm to find the

shortest path to the cell which leads to the door. after the agent reaches the door cell, the final few steps of the agent from the door cell to the goal are taken from the direct shortest path which was evaluated initially.
After these two alternative founds are found out, the cost of each one is formulated and the least cost policy is adopted. The notations for the following algorithm block are as follows:

- The Label Correcting algorithm is represent by lc_algorithm(s, τ). This block finds the shortest path from the s to τ
- The state which leads to the door are represented by token_state_door
- The states which have the door position are represented by door_position
- The states which lead to the key are represented by token_state_key
- first_path is the initial path found using lc algorithm without considering the door position
- key_path is the shortest path from initial state to the key position
- door_path is the shortest path from initial state to the door position
- door_goal_path is the shortest path from door position to the goal position.
- key_door_goal_path is the shortest path from initial to the key to the door to the goal
- no_door_path is the shortest path which does not have door in the way
- s is the initial state
- τ is the final state
- shortest_path is the final shortest path returned by this algorithm
- goal_state contains all the states which have the goal position in it
- cost() function evaluates the total cost of the policies.

Algorithm for the doorkey problem

$first\_path = lc\_algorithm(s, \tau)$;
if $first\_path$ is empty **then**
  $shortest\_path$= null
end if
if $door\_position \in first\_path$ **then**
  door_goal_path=first_path[door_position:goal_state]
  τ=token_state_key;
  key_path=$lc\_algorithm(s, \tau)$;
  g[s]=∞
  $s = \tau$
  g[s]=0
  τ =token_state_door
  door_path=$lc\_algorithm(s, \tau)$
  key_door_goal_path+=key_path;
  key_door_goal_path+=door_path;
  key_door_goal_path+=door_goal_path;
  door=wall;
  reinitialize states;
  no_door_path=lc_algorithm(s,τ);
  if key_door_goal_path is not null **then**
    if no_door_goal_path is not null **then**
      if cost(no_door_path)$\leq$cost(key_door_goal_path) **then**
        shortest_path= no_door_path;
      else
        shortest_path=key_door_goal_path;
      end if
    else
      shortest_path=key_door_goal_path;
    end if
  end if
  if key_door_goal_path is null **then**
    shortest_path= no_door_path;
  end if
else
  shortest_path=first_path;
end if=0

Is it to be noted that the required actions such as unlock door and pick key are taken once the agent reaches the desired token states. The results of this algorithm are discussed in the next section.

IV. Results

For normal case and direct case, there are three value functions since the agent goes to the key , then the door and then the goal. The state values after every end condition is met.

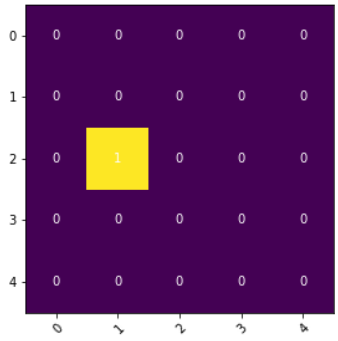

Fig. 1. 5x5-normal value plot for s=initial_state to τ=token_state_key

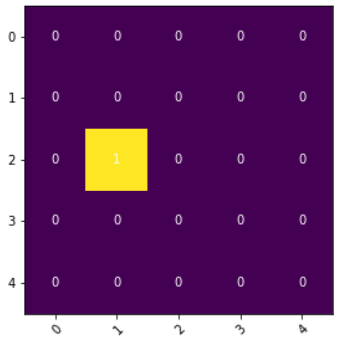

Fig. 2. 5x5-normal value plot for s=token_state_key to τ=token_state_door

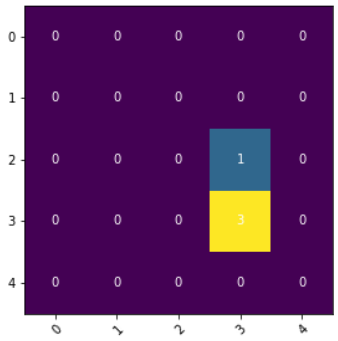

Fig. 3. 5x5-normal value plot for s=door_position to τ=goal_position

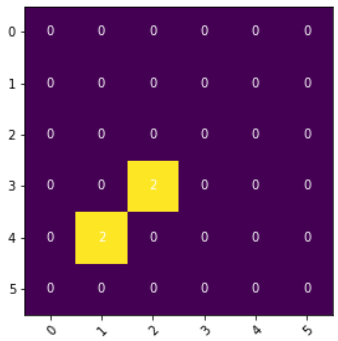

Fig. 4. 6x6-normal value plot for s=initial_state to τ=token_state_key

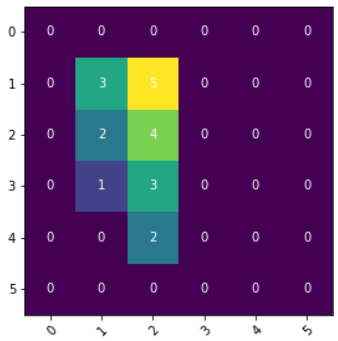

Fig. 5. 6x6-normal value plot for s=token_state_key to τ=token_state_door

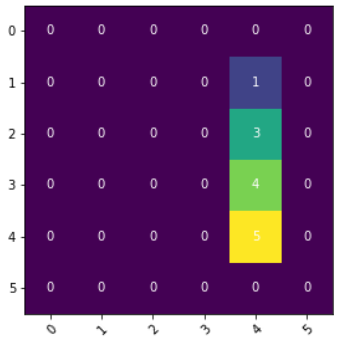

Fig. 6. 6x6-normal value plot for s=door_position to τ=goal_position

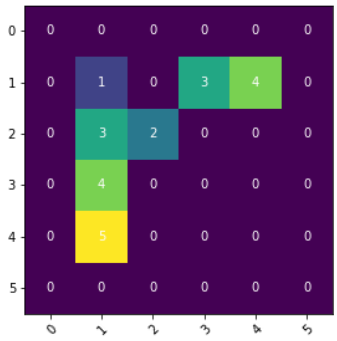

Fig. 7. 6x6-direct value plot for s=initial_state to τ=goal_position

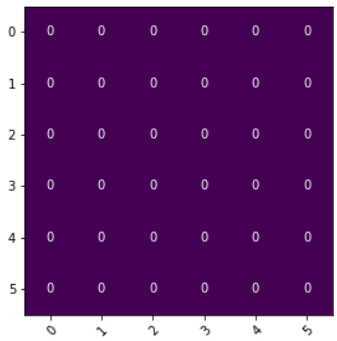

Fig. 8. 6x6-shortcut value plot for s=initial_state to τ=token_state_key

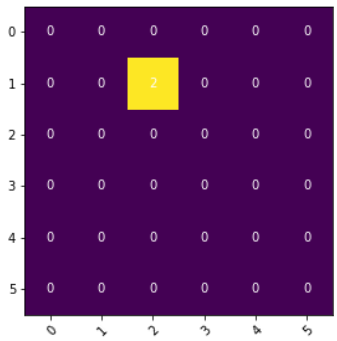

Fig. 9. 6x6-shortcut value plot for s=token_state_key to τ=token_state_door

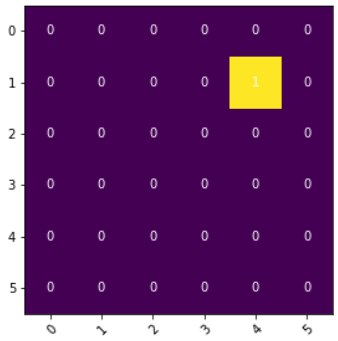

Fig. 10. 6x6-shortcut value plot for s=door_position to τ=goal_position

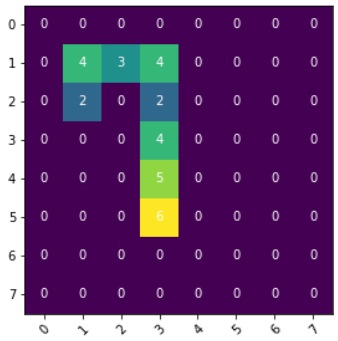

Fig. 11. 8x8-normal value plot for s=initial_state to τ=token_state_key

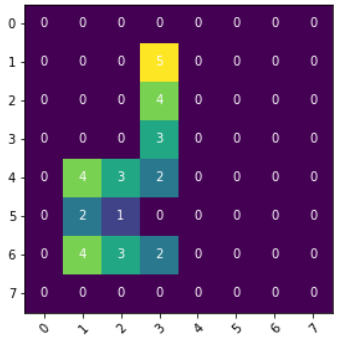

Fig. 12. 8x8-normal value plot for s=token_state_key to τ=token_state_door

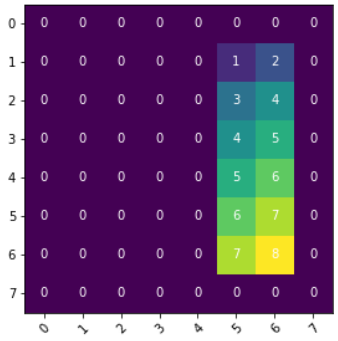

Fig. 13. 8x8-normal value plot for s=door_position to τ=goal_position

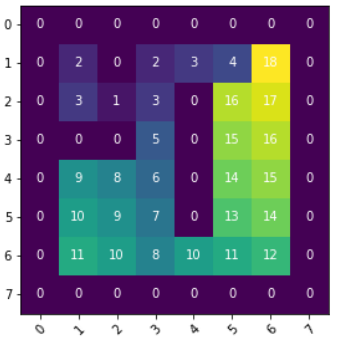

Fig. 14. 8x8-direct value plot for s=initial_state to τ=goal_position

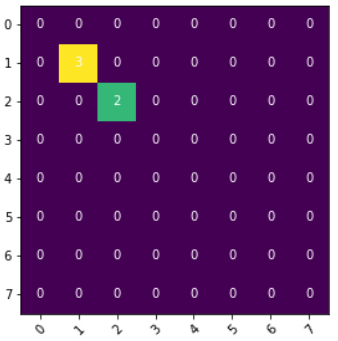

Fig. 15. 8x8-shortcut value plot for s=initial_state to τ=token_state_key

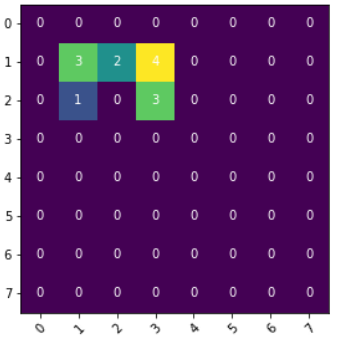

Fig. 16. 8x8-shortcut value plot for s=token_state_key to τ=token_state_door

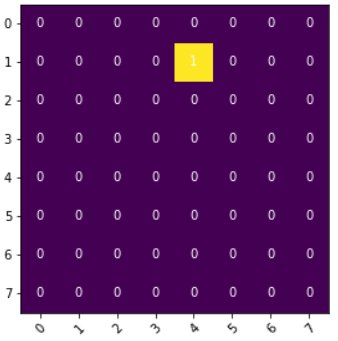

Fig. 17. 8x8-shortcut value plot for s=door_position to τ=goal_position

A. Action Sequence

The action sequence for the different scenarios are given as:
5x5-normal:
TL-PK-TR-UD-MF-MF-TR-MF
6x6-normal:
MF-TR-PK-MF-MF-MF-TR-MF-UD-MF-MF-TR-MF-MF-MF
6x6-direct:
TR-TR-MF-MF
6x6-shortcut:
PK-TR-TR-UD-MF-MF
8x8-normal:
TL-MF-TR-MF-MF-MF-TR-PK-TR-MF-MF-MF-MF-TR-UD-MF-MF-MF-TR-MF-MF-MF-MF-MF
8x8-direct:
TL-MF-MF-MF
8x8-shortcut:
MF-TR-PK-TR-MF-TR-MF-UD-MF-MF

B. Discussion

Initially the motive was to have a common value function for the paths to key, to door and to the goal. This would have been complex to construct. Hence the problem was was broken into three halves, which were getting three shortest paths by using the label correcting algorithm and adding them up. But this alone wouldn't have been enough.
There might be a possibility of another path which is shorter than the path through the door. Hence a new path was found by equating the door to be a wall. Then both the paths are compared and the path with the least cost is chosen. This algorithm is much faster when the direct path is the shortest path.
Another thing which worked well was the transferring of the states from the first_path to the door_goal path. This ensured that it wasn't necessary to evaluate the shortest path from door to goal again. This works because if there is an optimal path between state A and B, and C is a state in transition belonging to this path, then the optimal path from C to B is a part of the path from A to B.
Finally having equal costs helped because ultimately the aim was to reach the goal with least number of tasks. Hence of the tasks had the same value it is easy to minimize the total cost.

1 Code

```
#required imports
import numpy as np
import matplotlib.pyplot as plt
import utils
import gym
import pickle
from utils import *



#function to initliaze the various states,
# children of the states, and the total cost to reach each set
def initialize_states(env,info):
    list_states=[]
    gj=[]
    children={}
    #the direction states set
    dir_states=[[1,0],[0,1],[-1,0],[0,-1]]
    ct=0
    for i in range(0,env.height):
        for j in range(0,env.width):
            cell=env.grid.get(i,j)
            #for every cell which is not a wall add the states of every direction
            if(cell is None or (cell.color!='grey')):
                #Get the states and the immediate children of each state
                for k,direction in enumerate(dir_states):
                    #child is for storing children of the current state
                    child=[]
                    st=(np.array([i,j]),np.array(direction))
                    #add the states whose cell position is not a wall
                    list_states.append(st)
                    #Add the left and right states as children for the current state
                    if(k!=len(dir_states)-1):

                        child.append((st[0],np.array(dir_states[k-1])))
                        child.append((st[0],np.array(dir_states[k+1])))

                    else:
                        child.append((st[0],np.array(dir_states[k-1])))
                        child.append((st[0],np.array(dir_states[0])))
                    #check if the cell in front of the current state cell is not a wall.
                    #If not add it to the children of the current state
                    next_cell=np.sum(st,0)
                    if(next_cell[0]>=0 and next_cell[0]<info['width']):
                        if(next_cell[1]>=0 and next_cell[1]<info['height']):
                            neighbor_cell=env.grid.get(next_cell[0],next_cell[1])
                            if(neighbor_cell is None or neighbor_cell.color!='grey'):
                                child.append((next_cell,st[1]))
                    children.update({ct:child})
                    ct+=1
                    #set the initial cost of every state
                    # to inf except the starting state( set to 0)
                    if((st[0]==env.agent_pos).all() and (st[1]==env.dir_vec).all()):
                            gj.append(0)
                    else:
```

```
                            gj.append(np.inf)
    #return all the states, the initial cost, and the children dictionary
    return list_states,gj,children
#list_states contains a list of all the possible states of the agent
#gj is the list of initial costs of each state
#children is the dictionary with keys as the indices
#of each state and values as a list of states which are children
#to the state pointed by the corresponding key



#function to convert the children dictionary to one where the indices of each state
# is stored instead of the state itself
def get_children_indices(children,list_states):
    children_indices={}
    for i,states in enumerate(list_states):
        index_list=[]
        for child in children[i]:
            for j,st in enumerate(list_states):
                if((st[0]==child[0]).all() and (st[1]==child[1]).all()):
                    index_list.append(j)
        #store index of each children to the index of each parent
        children_indices.update({i:index_list})
    #the keys of this dictionary are the indices of the parents states
    #and the values are lists of the indices of the children
    return children_indices




#function to find the indices of vital states from list_states
def find_token_states(info,token,list_states,env,dir_states):
    token_states=[] #list to store the indices of the vital states
    check_states=[] #temporary list
    #temporary state to store the cell calue which might lead to door or key
    temp_state=np.array([])
    #to find the index of the initial state in list_states
    if token=='agent':
        for i,(a,b) in enumerate(list_states):
            if((a==info['init_agent_pos']).all() and (b==info['init_agent_dir']).all()):
                return i
    #to find the indices of all possible states of agent when it reaches the goal cell
    elif token=='goal':
         for i,(a,b) in enumerate(list_states):
            if((a==info['goal_pos']).all()):
                token_states.append(i)
   #to find the indices of all possible states of agent which leads to the key cell
    elif token=='key':
        for i,(a,b) in enumerate(list_states):
            if((a==info['key_pos']).all()):
                for direction in dir_states:
                    temp_state=a-direction
                    cell=env.grid.get(temp_state[0],temp_state[1])
                    if(cell is None or cell.color not in ['grey','green']):
                        check_states.append((temp_state,direction))
                break

        for i,(a,b) in enumerate(list_states):
                for ch in check_states:
                    if((a==ch[0]).all() and (b==ch[1]).all()):
```

```
                    token_states.append(i)
    #to find the indices of all possible states of agent which leads to the door cell
    elif token=='door':
        for i,(a,b) in enumerate(list_states):
            if((a==info['door_pos']).all()):
                for direction in dir_states:
                    temp_state=a-direction
                    cell=env.grid.get(temp_state[0],temp_state[1])
                    if(cell is None or cell.color not in ['grey','green']):
                        check_states.append((temp_state,direction))
                break

        for i,(a,b) in enumerate(list_states):
                for ch in check_states:
                    if((a==ch[0]).all() and (b==ch[1]).all()):
                        token_states.append(i)
     #to find the indices of all possible states of agent when it reaches the door cell
    elif token=='door_pos':
         for i,(a,b) in enumerate(list_states):
            if((a==info[token]).all()):
                token_states.append(i)
    #to find the indices of all possible states of agent when it reaches the key cell
    elif token=='key_pos':
         for i,(a,b) in enumerate(list_states):
            if((a==info[token]).all()):
                token_states.append(i)


    return token_states




#use lc algorithm to find the shortest path
def lc_algorithm(cij,children_indices,s,tau,gj):
    OPEN=[] #OPEN stack is the same one used in lc algorithm
    Parent={} #this dictionary used to backtrack the path
    index_i=0 #index of the state popped from OPEN
    gtau=[0]*len(tau) #cost of all the possible final states
    dsum=0
    OPEN.append(s)
    #lc algorithm loop
    while (len(OPEN)>0):
        index_i=OPEN.pop()
        for i,t in enumerate(tau):
                gtau[i]=gj[t]
        for k,j in enumerate(children_indices[index_i]):
            dsum=(gj[index_i]+cij[index_i][k])
            if(dsum<gj[j] and dsum<min(gtau)):
                gj[j]=dsum
                Parent.update({j:index_i})
                if(j in tau):
                        gtau[tau.index(j)]=gj[j]
                elif j not in tau:
                    OPEN.append(j)
    check=tau[np.argmin(gtau)]
    #check if the final state exists in Parent
    #if it does Parent and the total cost
    #if the final state doesn't exist return empty dictionary
    if(check in Parent.keys()):
        return Parent,gtau
```

```
    else:
        return {},np.inf



#create a list of the states traversed from the end to the start
def shortest_path(gtau,tau,s,path):
    fin_state=tau[np.argmin(gtau)] #final state of the path tau is a list of possible final state
    states_in_path=[] #list of all the states traversed in the shortest path
    x=fin_state
    states_in_path.append(x)
    #check if x= starting state
    #s= starting state
    while(x!=s):
        x_new=path[x]
        states_in_path.append(x_new)
        x=x_new
    return states_in_path




#use it when the door is not obstructing path ie
#no door in the shortest path or door key both on the way
#function to get the actions of the agent to travel the
#shortest path possible path if the door is not obstructing that
def policy_shortest_path(list_states,states_in_path,door_pos,key_pos,dir_states):
    policy=[] #list of all actions taken by the agent
    cur_dir=0 #storing the direction of the agent of a particular state
    next_dir=0 #storing the direction of the agent in the very next state

    for i in range(0,len(states_in_path)-1):
        # check if the next state is a cell which has the key
        if(states_in_path[i+1] in key_pos):
        # Add pick key to the list of actions
            policy.append(3)
        #check if the next stae has the location of the door
        elif(states_in_path[i+1] in door_pos):
        # add unlock door to the list of actions
            policy.append(4)

        if((list_states[states_in_path[i]][0]==list_states[states_in_path[i+1]][0]).all()):
            for k,direction in enumerate(dir_states):
                #check the direction of the current state
                if((list_states[states_in_path[i]][1]==direction).all()):
                    cur_dir=k
                    #check the direction of the next state
                if((list_states[states_in_path[i+1]][1]==direction).all()):
                    next_dir=k
                #check if the agent is turning left or right
            if(cur_dir==0 and next_dir==k):
                policy.append(1)
            elif(cur_dir==k and next_dir==0):
                policy.append(2)
            elif(next_dir>cur_dir):
                policy.append(2)
            elif(next_dir<cur_dir):
                policy.append(1)
                #append move forward if the agent is moving forward
```

```
        else:
            policy.append(0)

    #return the list of actions
    return policy




# function to create the info dictionary from env (same as the one given in utils.py)
def create_info(env):
    info = {
        'height': env.height,
        'width': env.width,
        'init_agent_pos': env.agent_pos,
        'init_agent_dir': env.dir_vec
        }

    for i in range(env.height):
        for j in range(env.width):
            if isinstance(env.grid.get(j, i),
                          gym_minigrid.minigrid.Key):
                info['key_pos'] = np.array([j, i])
            elif isinstance(env.grid.get(j, i),
                            gym_minigrid.minigrid.Door):
                info['door_pos'] = np.array([j, i])
            elif isinstance(env.grid.get(j, i),
                            gym_minigrid.minigrid.Goal):
                info['goal_pos'] = np.array([j, i])
    return info




#function to split the shortest path into three parts:
#1. Initial state s to the states (key_token)
# leading to the position of the key (key_pos)
#2. From the new state in key_token to the states (door_token)
# leading to the position of the door (door_pos)
#3. From door_pos to the final goal
# (this part is not solved in this function but taken from a previous function)
# The states traversed in part 3 is given by states_door_to_goal
def key_door_goal(list_states,gj,children,states_door_to_goal,\
                  s,tau,key_token,door_token,dir_states,\
                  key_pos,door_pos,is_key_found):
    children_indices=get_children_indices(children,list_states) #get the children indices
    cij={} #transition cost from i to j
    gj_actual=list(gj) #copy of the initial costs
    policy_final=[] #list of final actions to be taken
    #populating the tranisition cost with the value 1
    for i in range(0,len(children_indices.keys())):
        cij.update({i:np.ones(len(children_indices[i]))})
    #1: to get the policies to traverse the shortest path from start to key
    #is_key_found is used to check if the first state is the state leading to the key\
    #if is_key_found=1 then skip the lc algorithm from intial state to the key_token
    if(is_key_found==0):
        #path 1 is a dictionary of state traversal from s to key_token
        #gtau1 is the total cost
        path1,gtau1=lc_algorithm(cij,children_indices,s,key_token,gj_actual)
        #check path exists
        if(bool(path1)==False):
```

```
            print('no_path1')
            return [] #return no path
        #find the list of states traversed
        states_in_path1=shortest_path(gtau1,key_token,s,path1)[::-1] #reverse the list
        #create a list of the actions taken
        policy_final=policy_shortest_path(list_states\
                                          ,states_in_path1,door_pos,key_pos,dir_states)
        #add pick key action to the list
        policy_final.append(PK)
        #2: pick the key
        gj[s]=np.inf #set the cost of the initial state to infinity
        s=states_in_path1[-1] #s_new=key_token
        gj[s]=0 #set the cost of the current state as 0
    #3: from the current state to door
    #path2is a dictionary of state traversal from s_new to door_token
    #gtau2 is the total cost
    path2,gtau2=lc_algorithm(cij,children_indices,s,door_token,gj)
    #check path exists
    if(bool(path2)==False):
        print('no_path2')
        return []  #return no path
    #find the list of states traversed
    states_in_path2=shortest_path(gtau2,door_token,s,path2)[::-1] #reverse the list
    #create a list of the actions taken

    policy2=policy_shortest_path(list_states,states_in_path2,door_pos,key_pos,dir_states)

    for p in policy2:
        #check if the actions is not picking key
        # or unlocking door (ensure that these actions are just useed once)
        if(p!=PK and p!=UD):
            policy_final.append(p) #apppend the final_policy
    #append unlock door action
    policy_final.append(UD)
    #append move forward action
    policy_final.append(MF)
    #4: door to goal
    #we have the states for this traversal
    #compute the actions
    policy3=policy_shortest_path(list_states,states_door_to_goal,door_pos,key_pos,dir_states)
    for p in policy3:
        #check if the actions is not
        #picking key or unlocking door
        # (ensure that these actions are just useed once)
        if(p!=PK and p!=UD):
            policy_final.append(p) #apppend the final_policy

    return policy_final




# This function is exactly similar
#to initialize_states ,
# the only difference is that it is assumed that
# the door cell is a wall cell.
#This function is used to find the
# shortest path from agent
# to goal without using key or door.
def initialize_states_no_door(env,info):
    list_states=[]
```

```
    gj=[]
    children={}
    dir_states=[[1,0],[0,1],[-1,0],[0,-1]]
    ct=0
    for i in range(0,env.height):
        for j in range(0,env.width):
            cell=env.grid.get(i,j)
            if(cell is None or (isinstance(cell,gym_minigrid.minigrid.Wall) is not True)):
                if((isinstance(cell,gym_minigrid.minigrid.Door) is not True)):
                    for k,direction in enumerate(dir_states):
                            child=[]
                            st=(np.array([i,j]),np.array(direction))
                            list_states.append(st)
                            if(k!=len(dir_states)-1):

                                child.append((st[0],np.array(dir_states[k-1])))
                                child.append((st[0],np.array(dir_states[k+1])))

                            else:
                                child.append((st[0],np.array(dir_states[k-1])))
                                child.append((st[0],np.array(dir_states[0])))

                            next_cell=np.sum(st,0)
                            if(next_cell[0]>=0 and next_cell[0]<info['width']):
                                if(next_cell[1]>=0 and next_cell[1]<info['height']):
                                    neighbor_cell=env.grid.get(next_cell[0],next_cell[1])
                                    if(neighbor_cell is None \
                                                        or\
                                                        (isinstance\
                                                        (neighbor_cell\
                                                        ,gym_minigrid.minigrid.Door)\
                                                          is not True)):
                                            child.append((next_cell,st[1]))
                            children.update({ct:child})
                            ct+=1
                            if((st[0]==env.agent_pos).all() and (st[1]==env.dir_vec).all()):
                                    gj.append(0)
                            else:
                                    gj.append(np.inf)
    return list_states,gj,children



#function to calculate the shortest path with no dor cell in the path
def get_states_no_door(env,info,door_pos,dir_states,tokens,list_states,gj,children):
    #list_states,gj,children=initialize_states_no_door(env,info,door_pos)
    children_indices=get_children_indices(children,list_states)
    cij={}
    for i in range(0,len(children_indices.keys())):
        cij.update({i:np.ones(len(children_indices[i]))})
    # define start and end
    s=find_token_states(info,tokens[0],list_states,env,dir_states)
    tau=find_token_states(info,tokens[3],list_states,env,dir_states)
    #use lc algortihm
    path,gtau=lc_algorithm(cij,children_indices,s,tau,gj)
    if(bool(path) is False):
        return []
    #reverse state transition from end to start
    states_in_path=shortest_path(gtau,tau,s,path)[::-1]
    return states_in_path
```

```
def doorkey_problem(env):
    info=create_info(env)
#initialize the list_states, state costs and children etc.
    list_states,gj,children=initialize_states(env,info)
        #list consisting all the directions
    dir_states=[np.array([1,0]),np.array([0,1]),np.array([-1,0]),np.array([0,-1])]
    tokens={0:'agent',1:'key',2:'door',3:'goal'} #tokens for find_tokens
    door_pos=find_token_states(info,'door_pos',list_states,env,dir_states) #find the door positio
    key_pos=find_token_states(info,'key_pos',list_states,env,dir_states) #find the key position

    #compute the shortest path between the initial agent state to the goal
    states_in_path=get_states_no_door(env,\
                                      info,door_pos,dir_states,\
                                      tokens,list_states,gj,children)

    # check if there is a path possible
    if(bool(states_in_path) is False):
        return [] #return empty path

    #check if the shortest path has door in the way
    if(any(item in states_in_path for item in door_pos) is False):
        #compute the policy for the above shortest path and return
        policy=policy_shortest_path(list_states\
                                    ,states_in_path,door_pos,key_pos,dir_states)
        print('no_door_in_shortest_path')
        return policy
    else:
        #if the door is present check if the cell containing
        #the key is on the way to the goal in the shortest path
        if(any(item in states_in_path for item in key_pos) is True):
            #compute the policy for the above shortest path and return
            policy_new=policy_shortest_path\
                       (list_states,states_in_path,door_pos,key_pos,dir_states)
             #initialize states with no door (wall in place of door)
            list_states,gj,children=initialize_states_no_door(env,info)
            #get the state transition for this shortest path which has no door on the way
            states_in_path=get_states_no_door\
                           (env,info,door_pos,dir_states,tokens,list_states,gj,children)
            policy_no_door=[]
            if(bool(states_in_path) is True):
                #find the policy for this shortest path which has no door on the way
                policy_no_door=policy_shortest_path(\
                                                    list_states,states_in_path,\
                                                    door_pos,key_pos,dir_states)
            if(bool(policy_no_door) is True):
                if(len(policy_no_door)>len(policy_new)):
                        policy=policy_new
                else:
                        policy=policy_no_door
            else:
                policy=policy_new
            print('key_and_door_in_shortest_path')
            return policy
        else:
            print('check_key-door-goal_or_no-door_path')
            #find the states leading to the key
            key_token=find_token_states(info,tokens[1],list_states,env,dir_states)
            #find the state right before the agent enters the door cell
            for d in door_pos:
```

```
            if(d in states_in_path):
                door_state_index=states_in_path.index(d)
                break
        door_token=[] #list containing the state leading to the door
        s=find_token_states(info,tokens[0],list_states,env,dir_states)# initial agent state
        policy_key_door_goal=[] #policy for the path from begin to key to door to goal
        is_key_found=0
        #if the key is in front of s
        if(s in key_token):
            policy_key_door_goal.append(3) # append pick key
            is_key_found=1 #set the key has been found
        tau=find_token_states(info,tokens[1],list_states,env,dir_states) #find the goal state
        list_states,gj,children=initialize_states(env,info)  #initialize states
        door_token.append(states_in_path[door_state_index-1])
        states_door_to_goal=states_in_path[door_state_index:]
        #compute the shortest path from key to door to goal
        # and append it to the policy to be  returned
        policy_key_door_goal.extend(\
                                    key_door_goal(list_states,gj,\
                                                  children,states_door_to_goal,\
                                                  s,tau,key_token,door_token,\
                                                  dir_states,key_pos,door_pos,is_key_found)

            #initialize states with no door (wall in place of door)
        list_states,gj,children=initialize_states_no_door(env,info)
        #get the state transition for this shortest path which has no door on the way
        states_in_path=get_states_no_door\
                            (env,info,door_pos,dir_states,tokens,list_states,gj,children)
        if(bool(states_in_path) is True):
            #find the policy for this shortest path which has no door on the way
            policy_no_door=policy_shortest_path\
                            (list_states,states_in_path,door_pos,key_pos,dir_states)
        else:
            #if the path doesn't exist then return the key_door_doal policy
            return policy_key_door_goal
        #compare which policy is better
        if(len(policy_no_door)>len(policy_key_door_goal)):
            policy=policy_key_door_goal
        else:
            policy=policy_no_door

    return policy




def policy_to_words(policy):
    actions={0:"MF",1:"TL",2:"TR",3:"PK",4:'UD'}
    action_list=""
    for p in policy:
        action_list=action_list+actions[p]+"->"
    return action_list[:-2]




def main():
    env_path = './envs/doorkey-5x5-normal.env'
    env, info = load_env(env_path) # load an environment
    seq = doorkey_problem(env) # find the optimal action sequence
    actions=policy_to_words(seq)
```

```
    print(actions)
    draw_gif_from_seq(seq, load_env(env_path)[0]) # draw a GIF & save


if __name__ == '__main__':
    main()
```