Artificial Intelligence/Intelligent Agents 2022/2023

Homework 3 Reinforcement learning

Name : Cindy Aprilia
Matricole number : 286702

The Cooking Chef Problem Consider the case where the agent is your personal Chef. In particular, the agent (the smiley on the map) wants to cook the eggs recipe according to your indication (scrambled or pudding). In order to cook the desired recipe, the agent must first collect the needed tools (the egg beater on the map). Then he must reach the stove (the frying pan or the oven on the map). Finally, he can cook. Not that there are two special interlinked cells (marked with the G) that allow the agent to go from one side of the map to the other. But to do so, the agent needs to express his will to go on the other side. Cells in (4, 1) and (6, 1) are the special gate ones. They allow the agent to go from one side of the map to another. Those two special cells are interlinked, but the agent needs to express his will to go on the other side (Left or Right). Since you have a lot hungry, it is fundamental that the agent cooks the eggs accordingly to your taste (scrambled/pudding) as fast as he can without letting you wait for more than necessary.

In order to apply optimal control techniques such as value iteration, you need to model the aforementioned scenario as an MDP. Recall that an MDP is defined as tuple (S, A, P, R, γ), where: S: The (finite) set of all possible states. A: The (finite) set of all possible actions.

P : The transition function P : S x S x A [0, 1], which maps (sJ, s, a) to P (sJ|s, a), i.e., the probability of transitioning to state sJ ∈ S when taking action a ∈ A in state s ∈ S. Note that $\Sigma s' \in S\ P(S'|s,a) = 1$ for all s ∈ S, a ∈ A.

Figure 1: A particular instance of the cooking Chef problem. The goal is for the agent currently located in state (4, 3) to have a policy that always leads to cooking the eggs in location (4, 4) or (8, 4). Cells in (4, 1) and (6, 1) are the special gate ones.

R: The reward function R : S × A × S → R, which maps (s, a, sJ) to R(s, a, sJ), i.e., the reward obtained when taking action a ∈ A in state s ∈ S and arriving at state sJ ∈ S. γ: The discount factor, which controls how important are rewards in the future.

To answer the questions, consider the instance shown in Figure 1: • In the figure, the agent is at (4; 3) (but it can start at any of the grid cells). • The agent needed cooking tools as the egg beater is in position (1; 3) and (8; 3). • There are two different final goals, displayed as the frying pan is in position (1; 4) and the oven in position (8; 4). • Cells in (4; 1) and (6; 1) are the special gate ones. They allow the agent to go from one side of the map to another. Those two special cells are interlinked, but the agent needs to express his will to go on the other side. • The agent is not able to move diagonally. • Walls are represented by thick black lines. • The agent cannot move through walls. • An episode will end when the agent successfully cooks the scrambled eggs (see the above description).

Part a
Modeling the MDP as an infinite horizon MDP: the agent, once he starts to cook successfully, never ends, and it remains in an absorbing state.

Using the above problem description, answer the following questions:

In the answers, I change the tupple position as (x,y), which the x-axis is the horizontal row, and the y-axis is the vertical column.

a) Provide a concise description of the states of the MDP. How many states are in this MDP? (i.e.what is |S|).

The amount of possible combinations of many factors would determine how many states there would be in the MDP. The combination factors usually consist of the environment (the map, the tools, the stove, and the recipe), and the stating point of the agent).
The states of the MDP in this example are represented by the grid cells of the kitchen. Each state in the question is a ***tuple of the form (i, j) where i and j are the row and column number respectively****. These represent the coordinates of the agent's current position on the grid. Additionally, each state can be described by its attributes, such as whether the agent has collected the cooking tools, the position of the egg beater, the position of the stove, whether the agent is on one side or the other of the special gates and the recipe that the agent is trying to cook.*

There are ***36 total states*** *in this MDP (9 columns x 4 rows). There are 4 unaccessible cells in column 5, however this not decrease number of the state, as column 5 cells will be put between 2 walls with special door function.*

b) Provide a concise description of the actions of the MDP. How many actions are in this MDP? (i.e.what is |A|).

There are ***4 actions*** *in this MDP. Each action corresponds to a movement of the agent on the grid, but there are some additional action. It is important to note that the agent can only move in four cardinal direction and can not move diagonally.*
The actions of the MDP in this example include:

- *moving up*
- *moving down*
- *moving left*
- *moving right*

There are additional 'special' actions of the MDP in this example too, which includes:

- *using the special gates - right (i.e. expressing the will to go to the other side of the kitchen)*
- *using the special gates - left (i.e. expressing the will to go to the other side of the kitchen)*
- *Picking up tools*

c) What is the dimensionality of the transition function P?

The dimensionality of the transition function P is |S| x |A| x |S|. The number of possible states |S| is 36. The number of possible actions |A| is 4 which includes only common actions. So, the ***dimensionality of P will be 36 x 4 x 36.***
It assigns a probability to each possible transition given a current state and action.

However, the number may be a little lower than the real computation, due to the 'special' action and walls blockage.

d) Report the transition function P for any state s and action a in a tabular format.

It is not possible for me to report the transition function P for every state S and action A in a tabular format because it would require a huge amount of data, as it would be a 36 x 4 x 36 table. Therefore, I will give some examples of how the transition function P would work for a specific state and action.

For example, consider the state s = (3, 4) and the action a = "move right".
If the agent attempts to move right from state (3, 4), and there is no wall or obstacle in the way, then the agent will transition to state (3, 5).
If there is a wall in the way, the agent will stay in the same state (3, 4).
If the agent is attempting to move diagonally, the agent will stay in the same state (3, 4).

It s important to notice that the transition function P is stochastic, as the agent may have a probability of failure when attempting to use the special gates or move through walls.

PS. Here is some examples of the transition functions. The rest of the transition functions can be checked on attached spreedsheets

State	***Up***	***New Position After Up***	***Down***	***New Position After Down***	***Left***	***New Position After Left***	***Right***	***New Position After Right***	***Special Door Right***	***New Position After Special Door Right***	***Special Door Left***	***New Position After Special Door Left***
(1,1)	*0 (Try to go up but there are wall, so stay in same position)*	*(1,1)*	*0 (Try to go down but there are wall, so stay in same position)*	*(1,1)*	*0 (Try to go left but there are wall, so stay in same position)*	*(1,1)*	*1 (Success go right)*	*(2,1)*	*0 (No door)*	*(1,1)*	*0 (No door)*	*(1,1)*
(2,1)	*0*	*(2,1)*	*0*	*(2,1)*	*0.5*	*(1,1)*	*0.5*	*(3,1)*	*0*	*(2,1)*	*0*	*(2,1)*
(3,1)	*0.33*	*(3,2)*	*0*	*(3,1)*	*0.33*	*(2,1)*	*0.33*	*(4,1)*	*0*	*(3,1)*	*0*	*(3,1)*
(4,1)	*0.33*	*(4,2)*	*0*	*(4,1)*	*0.33*	*(3,1)*	*0*	*(4,1)*	*0.33*	*(6,1)*	*0*	*(4,1)*

e) Describe a reward function R : S × A × S and a value of γ that will lead to an optimal policy

The ***reward function assigns positive rewards for achieving the final goal of cooking the eggs in the desired recipe****, negative rewards for failing to use the special gates, negative rewards for each step taken before reaching the final goal and negative rewards for crashing into walls or attempting to move diagonally.*

A reward function R : S × A × S that will lead to an optimal policy could be defined as follows:
R(s, a, s') = -10 for crashing into a wall
R(s, a, s') = -1 for backtracking.
R(s, a, s') = 100 for successfully picking up a egg beater (only for the first time in each round, so the agent would not go back and forth in front of egg beater to accumulate reward).
R(s, a, s') = 100 for successfully cooking the eggs (only after picking up an egg beater, so the agent would not rush to cooking before using the egg beater).
R(s, a, s') = -1 for each step taken before the agent reaches the final goal.

As for the discount factor, a value of γ that will lead to an optimal policy could be 0.9. A value of γ close to 1 will give more importance to future rewards while a value close to 0 will give more importance to immediate rewards. In this case, a value of 0.9 strikes a good balance between the two. ***γ = 0.9*** *gives more weight to future rewards which means that the agent will try to reach the goal in fewer steps, but still take into account the immediate rewards, and it will not ignore them.*

f) Does γ ∈ (0, 1) affect the optimal policy in this case? Explain why.

The discount factor (γ), determines the importance of future rewards. A value of γ close to 1 will place a high importance on future rewards, while a value close to 0 will place a low importance on future rewards.
In the case of the Cooking Chef problem, ***the optimal policy may be affected by the value of γ chosen****. For example, if γ is set to a high value, the agent may prioritize reaching the goal state (the frying pan or the oven) as quickly as possible, even if it means not collecting the cooking tools or going through the special gate cells. On the other hand, if γ is set to a lower value, the agent may prioritize collecting the cooking tools and going through the special gate cells, even if it takes longer to reach the goal state.*

So, ***γ = 0.9*** *will strikes a good balance between the two. γ = 0.9 gives more weight to future rewards which means that the agent will try to reach the goal in fewer steps, but still take into account the immediate rewards, and it will not ignore them.*

g) How many possible policies are there? (All policies, not just optimal policies.)

The number of possible policies in this case would depend on the number of states and actions available to the agent. Since the agent can start in any of the grid cells, and can move in four directions (up, down, left, right), there are potentially a large number of possible policies. Additionally, if there are multiple tools and stoves in different locations, the number of possible policies increases.
The rough calculation will be 4^36 possible policies, as there are 36 states and 4 'basic' actions in this MDP. However the exact number may be different due to special gates and walls blokages

h) Now, considering the problem as a model-free scenario, provide a program (written in python) able to compute the optimal policy for this world considering the pudding eggs scenario solely. Draw the computed policy in the grid by putting in each cell the optimal action. If multiple actions are possible, include the probability of each arrow. There may be multiple optimal policies, pick one to show it. Note that the model is not available for computation but must be encoded to be used as the "real-world" environment.

The code will use tupple position as (x,y), which the x-axis is the horizontal row, and the y-axis is the vertical column.

In [99]:
```
import numpy as np
from collections import deque
```

In [114…
```
# Define the grid size
grid_size = (5, 10) #Because 0,0 still exist in array, but will be considered as unaccessible

# Define the starting position of the agent
starting_state = (3, 4)

# Define the positions of the cooking tools (egg beater), frying pan and oven
egg_beater_pos = [(3, 1), (3, 8)]
frying_pan_pos = (4, 1)
oven_pos = (4, 8)

# Define the positions of the special gates
gate_pos = [(1, 4), (1, 6)]

# Define the actions the agent can take (up, down, left, right)
possible_actions = ['up', 'down', 'left', 'right']

# Define the learning rate and discount factor
alpha = 0.1
gamma = 0.9

#Randomize of the way taken heuristic moves
epsilon = 1
isRandom = True

#How many testing
num_episodes = 3
```

In [115…
```
# Define the reward
def get_reward(current_state, next_state, num_steps, prev_state, fulfiled_prev):
    reward = 0
    if check_walls(current_state, next_state):
        reward = -10
    if ((current_state == prev_state) and (prev_state != (0,0))): #Not new, but backtracking
        reward = -1
    if ((next_state == prev_state) and (prev_state != (0,0))): #Not new, but backtracking
        reward = -1
    if current_state == next_state: #Not new, but backtracking
        reward = -1
    if ((fulfiled_prev == False) and (next_state in egg_beater_pos)):
        reward = 100 - num_steps
    if ((fulfiled_prev) and (next_state == frying_pan_pos or next_state == oven_pos)):
        reward = 100 - num_steps
    return reward
```

In [116…
```
#Check Walls :
def check_walls(current_state, next_state):
    #From/To (1,1) to/from (2,1)
    if ((current_state == (1,1)) and (next_state == (2,1))):
```

```
        return True
    elif ((current_state == (2,1)) and (next_state == (1,1))):
        return True

    #From/To (1,2) to/from (2,2)
    elif ((current_state == (1,2)) and (next_state == (2,2))):
        return True
    elif ((current_state == (2,2)) and (next_state == (1,2))):
        return True

    #From/To (2,2) to/from (3,2)
    elif ((current_state == (2,2)) and (next_state == (3,2))):
        return True
    elif ((current_state == (3,2)) and (next_state == (2,2))):
        return True

    #From/To (2,3) to/from (3,3)
    elif ((current_state == (2,3)) and (next_state == (3,3))):
        return True
    elif ((current_state == (3,3)) and (next_state == (2,3))):
        return True

    #From/To (3,1) to/from (4,1)
    elif ((current_state == (3,1)) and (next_state == (4,1))):
        return True
    elif ((current_state == (4,1)) and (next_state == (3,1))):
        return True

    #From/To (3,1) to/from (3,2)
    elif ((current_state == (3,1)) and (next_state == (3,2))):
        return True
    elif ((current_state == (3,2)) and (next_state == (3,1))):
        return True

    #From/To (1,8) to/from (2,8)
    elif ((current_state == (1,8)) and (next_state == (2,8))):
        return True
    elif ((current_state == (2,8)) and (next_state == (1,8))):
        return True

    #From/To (3,8) to/from (4,8)
    elif ((current_state == (3,8)) and (next_state == (4,8))):
        return True
    elif ((current_state == (4,8)) and (next_state == (3,8))):
        return True

    #From/To (3,9) to/from (4,9)
    elif ((current_state == (3,9)) and (next_state == (4,9))):
        return True
    elif ((current_state == (4,9)) and (next_state == (3,9))):
        return True

    #From/To (2,7) to/from (2,8)
    elif ((current_state == (2,7)) and (next_state == (2,8))):
        return True
    elif ((current_state == (2,8)) and (next_state == (2,7))):
        return True

    #From/To (3,7) to/from (3,8)
    elif ((current_state == (3,7)) and (next_state == (3,8))):
        return True
    elif ((current_state == (3,8)) and (next_state == (3,7))):
        return True

    #From/To (1,4) to/from (1,5)
    elif ((current_state == (1,4)) and (next_state == (1,5))):
        return True
    elif ((current_state == (1,5)) and (next_state == (1,4))):
        return True

    #From/To (1,5) to/from (1,6)
    elif ((current_state == (1,5)) and (next_state == (1,6))):
        return True
    elif ((current_state == (1,6)) and (next_state == (1,5))):
        return True

    #From/To (2,4) to/from (2,5)
    elif ((current_state == (2,4)) and (next_state == (2,5))):
        return True
    elif ((current_state == (2,5)) and (next_state == (2,4))):
        return True

    #From/To (2,5) to/from (2,6)
    elif ((current_state == (2,5)) and (next_state == (2,6))):
        return True
    elif ((current_state == (2,6)) and (next_state == (2,5))):
        return True
```

```
    #From/To (3,4) to/from (3,5)
    elif ((current_state == (3,4)) and (next_state == (3,5))):
        return True
    elif ((current_state == (3,5)) and (next_state == (3,4))):
        return True

    #From/To (3,5) to/from (3,6)
    elif ((current_state == (3,5)) and (next_state == (3,6))):
        return True
    elif ((current_state == (3,6)) and (next_state == (3,5))):
        return

    #From/To (4,4) to/from (4,5)
    elif ((current_state == (4,4)) and (next_state == (4,5))):
        return True
    elif ((current_state == (4,5)) and (next_state == (4,4))):
        return True

    #From/To (4,5) to/from (4,6)
    elif ((current_state == (4,5)) and (next_state == (4,6))):
        return True
    elif ((current_state == (4,6)) and (next_state == (4,5))):
        return True

    else :
        return False
```

In [117…

```
#check Grid Boundaries
def OutOfBoundaries(next_state, grid_size):
    if ((next_state[0] < 1) or (next_state[0] >= grid_size[0]) or (next_state[1] < 1) or (next_state[1] >= grid_size[1]) ):
        return True
    else:
        return False
```

In [118…

```
#Moving Due to Special Gate
def MoveSpecialGate(next_state, gate_pos):
    if next_state in gate_pos:
        if next_state == gate_pos[0]:
            return gate_pos[1], True
        elif next_state == gate_pos[1]:
            return gate_pos[0], True

    return next_state, False
```

In [119…

```
#Check finish game yet
def check_terminal_state(current_state, pass_requirement):
    #Always need to pass egg_beater first
    if (pass_requirement == False):
        return False

    terminal_states = [frying_pan_pos, oven_pos]
    if current_state in terminal_states:
        return True
    else:
        return False
```

In [120…

```
# Implement a function to choose the next action based on the Q-table
def choose_action(current_state, prev_state, q_table, possible_actions, epsilon, isRandom, final_pos):
    #if epsilon high therefore learning, or when there are no learning history
    if (np.sum(q_table) == 0) or (np.random.uniform(0, 1) < epsilon):
        if (isRandom == True):
            #either take from random
            action = np.random.choice(possible_actions)
        else :
            #or use auclean to nearer
            action = move_towards_target(current_state, prev_state, final_pos, possible_actions)
    #if Epsilon low therefore not learning and get from q_table
    else:
        state_action = q_table[current_state[0], current_state[1]]
        action = possible_actions[np.argmax(state_action)]

    return action
```

In [121…

```
def bfs(start, goal):
    queue = deque([start])
    visited = set([start])

    while queue:
        current = queue.popleft()

        if current == goal:
            return True  # goal found

        for next_state in get_valid_states(current):
            if next_state not in visited:
                visited.add(next_state)
                queue.append(next_state)
```

```
    return False  # goal not found
```

In [122…
```
def get_valid_states(current_state):
    valid_states = []
    for action in actions:
        next_state = get_next_step(action, current_state)
        if next_state and not check_walls(current_state, next_state):
            valid_states.append(next_state)
    return valid_states
```

In [123…
```
#Decide which beater to be taken
def move_towards_beater(current_state, egg_beater_pos):
    egg_beater_no = 0
    egg_beater_distance = (grid_size[0] * grid_size[1])

    for egg_beater_no_iter in range(len(egg_beater_pos)):
        distance = np.linalg.norm(np.array(current_state) - np.array(egg_beater_pos[egg_beater_no_iter]))
        if (distance < egg_beater_distance):
            egg_beater_no = egg_beater_no_iter
            egg_beater_distance = distance

    return egg_beater_no
```

In [124…
```
#Push the action to closer to the oven/frying pan/beater -> final_pos = oven_pos for oven and frying_pan_pos for frying pan
def move_towards_target(current_state, prev_state, final_pos, possible_actions):
    best_action = None
    best_distance = float("inf")

    for action in possible_actions:
        if action == "up":
            next_state = (current_state[0]-1, current_state[1])
        elif action == "down":
            next_state = (current_state[0]+1, current_state[1])
        elif action == "left":
            next_state = (current_state[0], current_state[1]-1)
        elif action == "right":
            next_state = (current_state[0], current_state[1]+1)

        if ((check_walls(current_state, next_state) == False) and (bfs(next_state, final_pos))):
            distance = np.sqrt((next_state[0] - final_pos[0])**2 + (next_state[1] - final_pos[1])**2)
            if (prev_state == next_state):
                continue
            if distance < best_distance:
                best_distance = distance
                best_action = action

    return best_action
```

In [125…
```
#get Next Step of Action :
def get_next_step(action, current_state):
    next_state = None

    if action == "up": # up
        next_state = (current_state[0]-1, current_state[1])
    elif action == "down": # down
        next_state = (current_state[0]+1, current_state[1])
    elif action == "left": # left
        next_state = (current_state[0], current_state[1]-1)
    elif action == "right": # right
        next_state = (current_state[0], current_state[1]+1)

    return next_state
```

In [126…
```
#Move agent
def transition_function(current_state, prev_state, action, num_steps, fulfiled_prev, is_terminal):
    next_state = None
    reward = None
    #is_terminal = None
    #fulfiled_prev = True

    next_state = get_next_step(action, current_state)

    # Check if the next state is a wall
    if check_walls(current_state, next_state):
        #print("Hit the wall | " + str(num_steps) + " | prev_state = " + str(prev_state) + " current_state = " + str(current_stat
        if (prev_state == (0,0)):
            next_state = current_state #For the 1st time if the next state hit walls therefore you won't move instead of backtrac
        else :
            next_state = prev_state

    #Skip if out of boundaries
    if OutOfBoundaries(next_state, grid_size):
        #print("Out of Boundaries | " + str(num_steps) + " | prev_state = " + str(prev_state) + " current_state = " + str(current
        if (prev_state == (0,0)):
            next_state = current_state #For the 1st time if the next state hit walls therefore you won't move instead of backtrac
        else :
```

```
            next_state = prev_state

    # Check if the next state is a special gate and update the state accordingly
    next_state, useSpecialGate = MoveSpecialGate(next_state, gate_pos)

    # Check if pick up egg_beater
    if (next_state in egg_beater_pos):
        #print("EGG BEATER FOUND | " + str(num_steps) + " | prev_state = " + str(prev_state) + " current_state = " + str(current_
        fulfiled_prev = True

    #check the game is final yet
    is_terminal = check_terminal_state(next_state, fulfiled_prev)

    # Get the reward for the current state, action and next state
    reward = get_reward(current_state, next_state, num_steps, prev_state, fulfiled_prev)

    return next_state, reward, is_terminal, fulfiled_prev, useSpecialGate
```

In [127…

```
# Load the Q-table if it exists
try:
    q_table = np.load("q_table.npy")
except:
    q_table = np.zeros(grid_size)

for episode in range(num_episodes):
    savePathArray = [] #Save path to be printed in the end

    #define starting value
    current_state = starting_state
    num_of_steps = 0
    prev_state = (0,0)

    #Define starting check toggle
    is_terminal = False #isFoundStove
    fulfiled_prev = False #isFoundBeater

    #get egg beater that be targeted (nearest egg beater with euclian distance)
    egg_beater_target = egg_beater_pos[move_towards_beater(current_state, egg_beater_pos)]

    while not is_terminal:
        final_pos = egg_beater_target if (fulfiled_prev == False) else frying_pan_pos #oven_pos

        # Choose an action based on the current state
        action = choose_action(current_state, prev_state, q_table, possible_actions, epsilon, isRandom, final_pos)

        # Perform the action and observe the next state and reward
        next_state, reward, is_terminal, fulfiled_prev, useSpecialGate = transition_function(current_state, prev_state, action, r

        # Update the Q-value for the current state and action
        next_row, next_col = next_state
        q_table[current_state[0], current_state[1]] = (1 - alpha) * q_table[current_state[0], current_state[1]] + alpha * (reward

        savePathArray.append(current_state)

        prev_state = current_state
        current_state = next_state

        if (is_terminal):
            # Save the Q-table after training
            print("\nFinish in : " + str(num_of_steps) + " steps.\n" + str(savePathArray))
            np.save("q_table.npy", q_table)

        num_of_steps += 1

        #If using special gate the steps was added 1 for declaring the usage of special gate
        if (useSpecialGate):
            num_of_steps += 1
            savePathArray.append('UseGate')
```

```
Finish in : 35 steps.
[(3, 4), (2, 4), (2, 3), (2, 4), (2, 3), (2, 2), (2, 3), (2, 2), (2, 1), (2, 2), (2, 1), (2, 2), (2, 1), (2, 2), (2, 1), (2, 2),
(2, 1), (3, 1), (2, 1), (3, 1), (2, 1), (2, 2), (2, 3), (2, 4), (3, 4), (3, 3), (3, 4), (3, 3), (3, 2), (3, 3), (4, 3), (4, 2),
(4, 3), (4, 4), (4, 3), (4, 2)]

Finish in : 38 steps.
[(3, 4), (3, 4), (3, 4), (3, 4), (2, 4), (2, 3), (1, 3), (2, 3), (2, 2), (2, 1), (3, 1), (2, 1), (3, 1), (2, 1), (3, 1), (2, 1),
(3, 1), (2, 1), (2, 2), (2, 1), (2, 2), (2, 1), (2, 2), (2, 1), (3, 1), (2, 1), (2, 2), (2, 1), (2, 2), (2, 3), (2, 4), (3, 4),
(3, 3), (4, 3), (4, 2), (3, 2), (4, 2), (3, 2), (4, 2)]

Finish in : 42 steps.
[(3, 4), (2, 4), (3, 4), (2, 4), 'UseGate', (1, 6), (2, 4), (3, 4), (2, 4), (2, 3), (2, 2), (2, 1), (3, 1), (2, 1), (2, 2), (2,
1), (2, 2), (2, 3), (1, 3), (2, 3), (2, 4), (3, 4), (4, 4), (3, 4), (4, 4), (3, 4), (3, 3), (3, 2), (3, 3), (3, 4), (4, 4), (4,
3), (4, 4), (4, 3), (3, 3), (3, 2), (4, 2), (3, 2), (4, 2), (4, 3), (4, 2), (3, 2), (4, 2)]
```

In [ ]:

i) Is the computed policy deterministic or stochastic?

The computed policy is deterministic if the agent always takes the same action in the same state, based on the information available to it (agent always chooses the action with the highest expected value).
However, if we use epsilon-greedy exploration strategy (agent will choose a random action with probability epsilon). This means that there is a chance that the agent will choose a different action than the one with the highest value in the Q-table, making the policy non-deterministic.

In other words, ***The computed policy is stochastic*** *because there is a probability that the agent will choose a random action rather than the action with the highest value in the Q-table. This* ***allows the agent to explore the action space and improve its knowledge of the environment****.*

j) Is there any advantage to having a stochastic policy? Explain.

Yes, there are advantages to having a stochastic policy.
Some of stochastic policy examples :

- *Allows the agent to explore the action space and improve its knowledge of the environment.*
- *Agent with a stochastic policy is more likely to explore different actions, even if they have not been previously selected, this means that it will not get stuck in a local optimum and can learn the optimal policy for the environment.*
- *Can help to avoid getting stuck in suboptimal solutions. In some cases, there may be multiple optimal solutions, and a deterministic policy may only find one of them. A stochastic policy allows the agent to explore different solutions, increasing the chances of finding the global optimum.*
- *A stochastic policy allows the agent to adapt to changes in the environment and uncertainty, making it more robust. Because, in the real-world problems, the environment is not deterministic, and the agent cannot rely on a strict deterministic policy.*

Part b

Now consider that your agent, because of his tiredness, might goes in the wrong direction. Then each action has a 60% chance of going in the chosen direction and 40% chance of going perpendicular to the right of the direction chosen. Accordingly, with these new settings, answer the following questions:

a) Report the transition function P for any state s and action a ∈ A.

The transition function P for any state s and action a ∈ A in this scenario would be as follows:
- *For any action a that moves the agent in the intended direction, there is a 60% chance that the agent will end up in the intended state sJ and a 40% chance that the agent will end up in a state that is perpendicular to the right of the intended direction.*

For any action a that would result in the agent hitting a wall, the agent will remain in the same state S.
- *For any action a that would result in the agent reaching the special gate cells (4,1) or (6,1), there is a 100% chance that the agent will transition to the corresponding special gate cell on the opposite side of the map.*
- *For any action a that would result in the agent reaching the egg beater or the stove, there is a 100% chance that the agent will transition to the state where the egg beater or stove is located and pick it up.*

So for any state s and action a, the transition function P would be represented as a probability distribution over the possible next states sJ, with the probabilities determined by the conditions above.

b) Does the optimal policy change compared to Part a? Justify your answer.

With this new policy, the agent's movements will be less predictable and there is a higher chance of the agent deviating from its intended direction. This will make it more difficult for the agent to reach its goal and make the cooking process longer. The optimal policy will have to take this into account and may require more steps or a more complex strategy to reach the final goal.

c) Will the value of the optimal policy change? Explain how.

The value of the optimal policy may also change as the agent will likely receive lower rewards due to the increased chance of deviating from the intended path. The agent may also have a higher chance of getting stuck in certain states as it deviates from its intended direction, which will also affect the value of the optimal policy. Overall, the new policy makes the problem more challenging and requires a more robust solution to find the optimal policy.
However, it's also possible that the new policy allows the agent to explore more states and therefore find a better optimal policy with higher value. The optimal policy may also change as the agent may now have a higher chance of ending up in a state that was not previously considered optimal due to the increased probability of deviation.