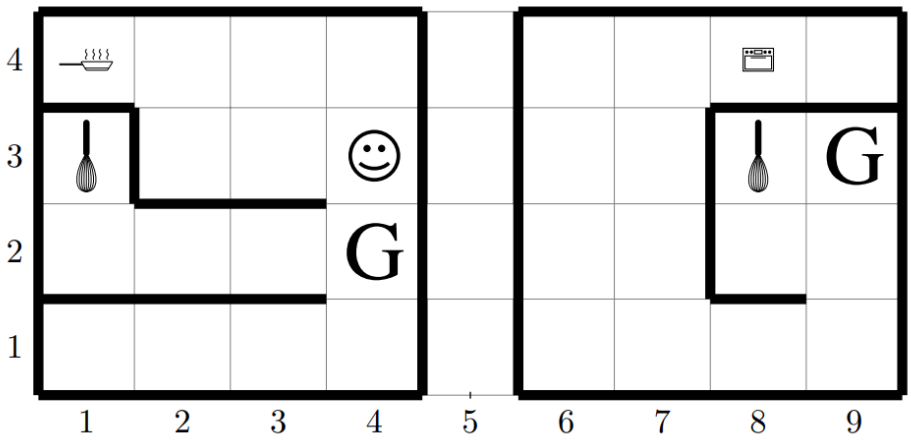

Figure 1: Given instance of the cooking chef problem

a) Since the given instance of the cooking chef problem in figure 1 shows two individual grids each of the size 4X4 and has solid walls between them, so each of the grids should have 16 states and the two grids therefore together has 32 states in total.

It is also important to provide a concise description of the actions for a better understanding of the solution.

Overall, the agent can have 6 actions actions: {left, right, up, down, useGateForRightGrid, useGateForLeftGrid}.

```
Environment representation:
+----+----+----+----+
|  1 |  2 |  3 |  4 |
+----+----+----+----+
|  5 |  6 |  7 |  8 |
+----+----+----+----+
|  9 | 10 | 11 | 12 |
+----+----+----+----+
| 13 | 14 | 15 | 16 |
+----+----+----+----+
+----+----+----+----+
| 17 | 18 | 19 | 20 |
+----+----+----+----+
| 21 | 22 | 23 | 24 |
+----+----+----+----+
| 25 | 26 | 27 | 28 |
+----+----+----+----+
| 29 | 30 | 31 | 32 |
+----+----+----+----+
```

Figure 2: Solution's state representation

d) Figure 2 shows the state representation of the problem in the solution. Overall, there are two grids each consisting of 16 cells. The cells of the first grid have cells numbered from 1 to 16 corresponding

to the states of the left grid, while the other grid has cells numbered from 17 to 32 corresponding to the states of the right grid.

In the submitted project folder there is a file named “transition_function_representation.xlsx” file showing the overall transition of the states for the given actions.

e) In the reward function, the agent is encouraged to get the beaters before going to the stove or oven. Keeping this in mind, the following reward function was defined.

There is a default negative incentive of -1 for all the states that are without beaters or stove/oven. For the encouragement of getting the beater before starting the cooking process, a much higher incentive of +50 was set. For reaching the end states, that is, the states with the stove or oven there are even higher incentive of +100. However, if the agent reaches the end state without the beaters, there is a negative reward of -100 as a penalty. The discount factor was set as 0.6 as it provided a good balance in choosing the tradeoff between immediate reward and future reward. The chosen value suggests overall, the agent has an ever so slight bias for the future reward.

h) The “code_implementation.py” contains the python code to solve the cooking chef problem. The code has the following classes and functions.

class State: Represents a state in the environment of the reinforcement learning model. The class’s constructor initializes the position of the agent and whether the agent has a beater or not. The ‘__eq__’ method compares two instances of the State class. It overrides the default equality behavior and defines those two states are considered equal if and only if they have the same position and the same ‘has_beater’ status. This method in identifying whether the agent has encountered a particular state before. The ‘__hash__’ method provides a hash value for an instance of the State class. The hash value is a unique integer calculated from the ‘position’ and ‘has_beater’ attributes. This allows storing and retrieving state instances in dictionaries used to store Q-values.

class Agent: This class represents the learning entity of the reinforcement model. It has three class attributes, namely, ‘epsilon_value’, which is the exploration rate that determines how often the agent will choose a random action (exploration) versus choosing the best-known action (exploitation); ‘discount_factor’ often represented as gamma determines how much the agent values future rewards compared to immediate rewards; and ‘learning_rate’ is the rate at which the agent updates the knowledge, that is, it's a factor in how much new information affects the existing Q-values. The constructor of the class Initializes the agent's current state (present_state) based on the provided initial_position; initializes an empty dictionary (q_values) to store the Q-values, which represent the expected utility of taking a certain action in a given state; and defines the set of possible actions (actions) the agent can take. ‘update_q_value’ method updates the Q-values using the Bellman equation. The Q-value for a given state and action pair is updated based on the reward received and the maximum predicted Q-value of the next state.

class Environment: This class represents the setting or the "world" in which the agent operates in your reinforcement learning model. The constructor of the class defines the oven, gates, beaters’ positions,

while 'illegal_moves' is the set of illegal moves computed by the 'get_illegal_moves' method, which defines the moves that would take the agent off the grid (e.g., moving left from the left edge and specific moves that are disallowed within the grid, for example moving from position 1 to 5 is illegal because of a barrier. 'get_reward' defines the reward structure for the environment according to the definitions mentioned in part e.

function get_next_state: This function determines the next state of the agent based on its current state and the action it decides to take. Overall, this function responsible for transitioning the agent from one state to another based on its actions while respecting the rules and constraints of the environment. This includes moving around the grid, picking up items like beaters, and using gates to teleport between different areas of the environment.

function print_optimal_policy: The function prints the optimal policy determined by the reinforcement learning agent.

function run_episodes: This function simulates a single episode of the agent's interaction with the environment. It initializes the agent in a random state without a beater and iteratively selects actions based on either exploration (choosing randomly) or exploitation (choosing the best-known action), updating the agent's knowledge (Q-values) after each action. The episode continues until the agent reaches the oven with a beater, accumulating rewards along the way, which are returned as the total reward for the episode.

function main: The 'main' function serves as the entry point for the reinforcement learning simulation. It initializes the environment and the agent (starting without a beater at position 8), and then runs multiple training episodes (10,000 in this case). In each episode, the agent interacts with the environment, and the total reward is calculated and printed. After completing all episodes, the function calculates the optimal policy for each state when the agent has a beater, displaying the best action for each state.

A sample of the printed optimal policy is given below.

```
Optimal Policy With Beater:

Optimal Policy - Has Beater: True
State (Position: 1, Has Beater: True): Best Action -> left
State (Position: 2, Has Beater: True): Best Action -> up
State (Position: 3, Has Beater: True): Best Action -> useGateForRightGrid
State (Position: 4, Has Beater: True): Best Action -> down
State (Position: 5, Has Beater: True): Best Action -> down
State (Position: 6, Has Beater: True): Best Action -> left
State (Position: 7, Has Beater: True): Best Action -> right
State (Position: 8, Has Beater: True): Best Action -> down
State (Position: 9, Has Beater: True): Best Action -> right
State (Position: 10, Has Beater: True): Best Action -> right
State (Position: 11, Has Beater: True): Best Action -> right
State (Position: 12, Has Beater: True): Best Action -> down
State (Position: 13, Has Beater: True): Best Action -> down
```

```
State (Position: 14, Has Beater: True): Best Action -> right
State (Position: 15, Has Beater: True): Best Action -> down
State (Position: 16, Has Beater: True): Best Action -> left
State (Position: 17, Has Beater: True): Best Action -> right
State (Position: 18, Has Beater: True): Best Action -> right
State (Position: 19, Has Beater: True): Best Action -> left
State (Position: 20, Has Beater: True): Best Action -> left
State (Position: 21, Has Beater: True): Best Action -> down
State (Position: 22, Has Beater: True): Best Action -> up
State (Position: 23, Has Beater: True): Best Action -> right
State (Position: 24, Has Beater: True): Best Action -> useGateForLeftGrid
State (Position: 25, Has Beater: True): Best Action -> down
State (Position: 26, Has Beater: True): Best Action -> left
State (Position: 27, Has Beater: True): Best Action -> up
State (Position: 28, Has Beater: True): Best Action -> up
State (Position: 29, Has Beater: True): Best Action -> right
State (Position: 30, Has Beater: True): Best Action -> up
State (Position: 31, Has Beater: True): Best Action -> up
State (Position: 32, Has Beater: True): Best Action -> up
```