21AIE311 – Reinforcement Learning

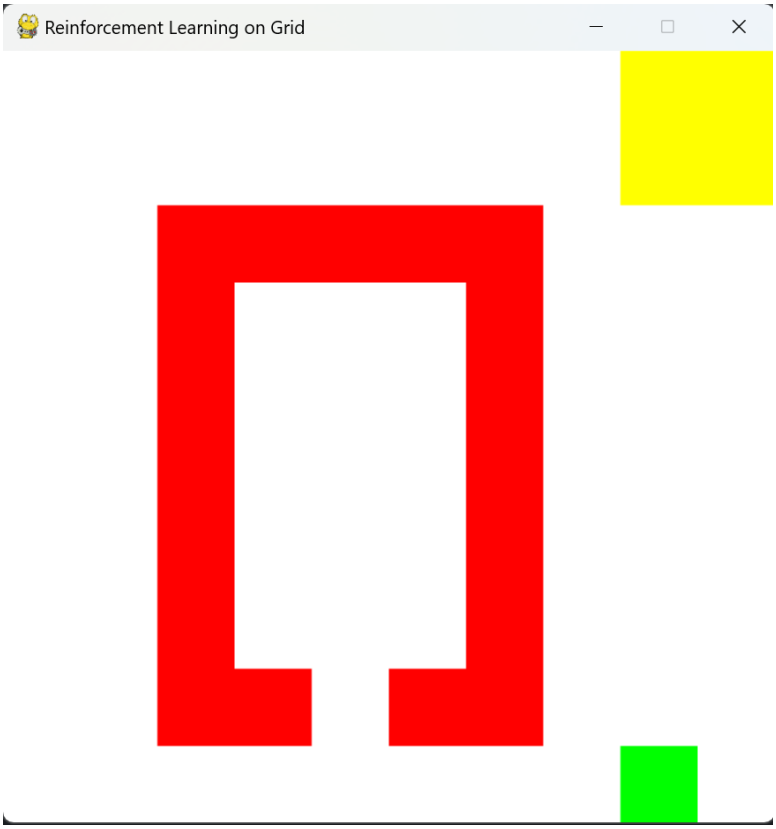
J Viswaksena - AM.EN.U4AIE21035

1.

```
import pygame
import sys
import numpy as np
import time
from collections import deque

# Initialize Pygame
pygame.init()

# Constants
WIDTH, HEIGHT = 500, 500  # Adjusted width and height
ROWS, COLS = 10, 10
CELL_SIZE = WIDTH // COLS  # Adjusted cell size
GRID_COLOR = (0, 0, 0)
BACKGROUND_COLOR = (255, 255, 255)
SPECIAL_CELL_COLOR_YELLOW = (255, 255, 0)
SPECIAL_CELL_COLOR_RED = (255, 0, 0)
SPECIAL_CELL_COLOR_GREEN = (0, 255, 0)
SPECIAL_CELLS_YELLOW = [(0, 8), (0, 9), (1, 8), (1, 9)]
SPECIAL_CELLS_RED = [(2,2),(2,3),(2,4),(2,5),(2,6),(3,2),(4,2),(5,2),(6,2),(7,2),
                     (8,2),(3,6),(4,6),(5,6),(6,6),(7,6),(8,6),(8,5),(8,3)]
CELL_VALUES = {}

# Set values for special cells
for cell in SPECIAL_CELLS_YELLOW:
    CELL_VALUES[cell] = 20  # Reward for reaching the goal

for cell in SPECIAL_CELLS_RED:
    CELL_VALUES[cell] = -1  # Penalty for hitting red cells

# Set value for all remaining cells
for row in range(ROWS):
    for col in range(COLS):
        if (row, col) not in CELL_VALUES:
            CELL_VALUES[(row, col)] = 1  # Default value for white cells

# Map actions to movements
ACTIONS = {"up": (-1, 0), "down": (1, 0), "left": (0, -1), "right": (0, 1)}

# Set up the display
```

```
win = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Reinforcement Learning on Grid")

def draw_grid():
    for x in range(0, WIDTH, CELL_SIZE):
        pygame.draw.line(win, GRID_COLOR, (x, 0), (x, HEIGHT))
    for y in range(0, HEIGHT, CELL_SIZE):
        pygame.draw.line(win, GRID_COLOR, (0, y), (WIDTH, y))

def draw_cell(row, col):
    cell_rect = pygame.Rect(col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE, CELL_SIZE)
    if (row, col) in SPECIAL_CELLS_YELLOW:
        pygame.draw.rect(win, SPECIAL_CELL_COLOR_YELLOW, cell_rect)
    elif (row, col) in SPECIAL_CELLS_RED:
        pygame.draw.rect(win, SPECIAL_CELL_COLOR_RED, cell_rect)
    else:
        pygame.draw.rect(win, BACKGROUND_COLOR, cell_rect)

def draw_agent(agent_position):
    agent_rect = pygame.Rect(agent_position[1] * CELL_SIZE, agent_position[0] * CELL_SIZE,
CELL_SIZE, CELL_SIZE)
    pygame.draw.rect(win, SPECIAL_CELL_COLOR_GREEN, agent_rect)

def main():
    agent_position = (4, 4)  # Initial position of the agent
    visited_cells = set()  # Keep track of visited cells
    queue = deque([agent_position])  # Queue for breadth-first search
    parent = {agent_position: None}  # Parent dictionary for storing the shortest path
    reached_yellow_cell = False  # Flag to indicate if the agent reached a yellow cell

    while queue:
        current_position = queue.popleft()
        visited_cells.add(current_position)

        if current_position in SPECIAL_CELLS_YELLOW:
            reached_yellow_cell = True
            break

        for action, movement in ACTIONS.items():
            next_row, next_col = current_position[0] + movement[0], current_position[1] +
movement[1]
            next_position = (next_row, next_col)

            if (0 <= next_row < ROWS and 0 <= next_col < COLS and
                CELL_VALUES[next_position] != -1 and next_position not in visited_cells):
                queue.append(next_position)
                parent[next_position] = current_position

    # Trace back the path if the agent reached a yellow cell
    if reached_yellow_cell:
```

```
        path = []
        while current_position:
            path.append(current_position)
            current_position = parent[current_position]

        path.reverse()  # Reverse the path to start from the agent's position

        for next_position in path[1:]:  # Exclude the agent's initial position
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()

            # Fill the background
            win.fill(BACKGROUND_COLOR)

            # Draw the grid
            draw_grid()

            # Draw cells
            for row in range(ROWS):
                for col in range(COLS):
                    draw_cell(row, col)

            # Draw the agent
            draw_agent(next_position)

            pygame.display.update()

            # Add a delay to visualize movement
            time.sleep(0.5)

    # Keep the window open until the user closes it manually
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

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



2.

```
import pygame
import sys
import numpy as np
import time
from collections import deque


pygame.init()


WIDTH, HEIGHT = 500, 500
ROWS, COLS = 10, 10
CELL_SIZE = WIDTH // COLS  # Adjusted cell size
GRID_COLOR = (0, 0, 0)
BACKGROUND_COLOR = (255, 255, 255)
SPECIAL_CELL_COLOR_YELLOW = (255, 255, 0)
SPECIAL_CELL_COLOR_RED = (255, 0, 0)
SPECIAL_CELL_COLOR_GREEN = (0, 255, 0)
SPECIAL_CELLS_YELLOW = [(0, 8), (0, 9), (1, 8), (1, 9)]
SPECIAL_CELLS_RED = [(2,2),(2,3),(2,4),(2,5),(2,6),(3,2),(4,2),(5,2),(6,2),(7,2),
                     (8,2),(3,6),(4,6),(5,6),(6,6),(7,6),(8,6),(8,5),(8,3)]
CELL_VALUES = {}

# Set values for special cells
for cell in SPECIAL_CELLS_YELLOW:
    CELL_VALUES[cell] = 20  # Reward for reaching the goal
```

```
for cell in SPECIAL_CELLS_RED:
    CELL_VALUES[cell] = -1  # Penalty for hitting red cells

# Set value for all remaining cells
for row in range(ROWS):
    for col in range(COLS):
        if (row, col) not in CELL_VALUES:
            CELL_VALUES[(row, col)] = 1  # Default value for white cells

# Map actions to movements
ACTIONS = {"up": (-1, 0), "down": (1, 0), "left": (0, -1), "right": (0, 1)}

# Set up the display
win = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Reinforcement Learning on Grid")

class Bandit:
    def __init__(self, k, epsilon):
        self.k = k  # Number of arms/actions
        self.epsilon = epsilon  # Exploration rate
        self.q_estimates = np.zeros(k)  # Estimated values of actions
        self.action_counts = np.zeros(k)  # Number of times each action is chosen

    def choose_action(self):
        if np.random.rand() < self.epsilon:
            return np.random.choice(range(self.k))  # Choose a random action index with
probability epsilon
        else:
            # Choose the action with the highest estimated value (breaking ties randomly)
            max_actions = np.where(self.q_estimates == np.max(self.q_estimates))[0]
            return np.random.choice(max_actions)

    def update_estimates(self, action_index, reward):
        self.action_counts[action_index] += 1
        # Update the estimated value of the chosen action using the incremental sample
average formula
        self.q_estimates[action_index] += (1 / self.action_counts[action_index]) * (reward -
self.q_estimates[action_index])

    def bandit_environment(self, action_index):
        movement = ACTIONS[list(ACTIONS.keys())[action_index]]
        next_row = 0 + movement[0]
        next_col = 0 + movement[1]

        # Ensure the next position falls within the grid boundaries
        if 0 <= next_row < ROWS and 0 <= next_col < COLS:
            next_position = (next_row, next_col)
            return np.random.normal(loc=CELL_VALUES[next_position], scale=1.0)
        else:
```

```
            # If the next position is out of bounds, return a default reward
            return 0  # You can adjust the default reward as needed




def draw_grid():
    for x in range(0, WIDTH, CELL_SIZE):
        pygame.draw.line(win, GRID_COLOR, (x, 0), (x, HEIGHT))
    for y in range(0, HEIGHT, CELL_SIZE):
        pygame.draw.line(win, GRID_COLOR, (0, y), (WIDTH, y))

def draw_cell(row, col):
    cell_rect = pygame.Rect(col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE, CELL_SIZE)
    if (row, col) in SPECIAL_CELLS_YELLOW:
        pygame.draw.rect(win, SPECIAL_CELL_COLOR_YELLOW, cell_rect)
    elif (row, col) in SPECIAL_CELLS_RED:
        pygame.draw.rect(win, SPECIAL_CELL_COLOR_RED, cell_rect)
    else:
        pygame.draw.rect(win, BACKGROUND_COLOR, cell_rect)

def draw_agent(agent_position):
    agent_rect = pygame.Rect(agent_position[1] * CELL_SIZE, agent_position[0] * CELL_SIZE,
CELL_SIZE, CELL_SIZE)
    pygame.draw.rect(win, SPECIAL_CELL_COLOR_GREEN, agent_rect)

def main(num_steps, k, epsilon):
    agent_position = (4, 4)  # Initial position of the agent
    visited_cells = set()  # Keep track of visited cells
    queue = deque([agent_position])  # Queue for breadth-first search
    parent = {agent_position: None}  # Parent dictionary for storing the shortest path
    reached_yellow_cell = False  # Flag to indicate if the agent reached a yellow cell

    bandit = Bandit(k, epsilon)
    rewards = []

    while queue:
        current_position = queue.popleft()
        visited_cells.add(current_position)

        if current_position in SPECIAL_CELLS_YELLOW:
            reached_yellow_cell = True
            break

        for action_index in range(k):  # Iterate over all possible action indices
            movement = ACTIONS[list(ACTIONS.keys())[action_index]]
            next_row, next_col = current_position[0] + movement[0], current_position[1] +
movement[1]
            next_position = (next_row, next_col)
```

```
            if (0 <= next_row < ROWS and 0 <= next_col < COLS and
                CELL_VALUES[next_position] != -1 and next_position not in visited_cells):
                queue.append(next_position)
                parent[next_position] = current_position

        # Choose action from Bandit algorithm
        action_index = bandit.choose_action()
        reward = bandit.bandit_environment(action_index)
        bandit.update_estimates(action_index, reward)
        rewards.append(reward)

    # Trace back the path if the agent reached a yellow cell
    if reached_yellow_cell:
        path = []
        while current_position:
            path.append(current_position)
            current_position = parent[current_position]

        path.reverse()  # Reverse the path to start from the agent's position

        for next_position in path[1:]:  # Exclude the agent's initial position
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()

            # Fill the background
            win.fill(BACKGROUND_COLOR)

            # Draw the grid
            draw_grid()

            # Draw cells
            for row in range(ROWS):
                for col in range(COLS):
                    draw_cell(row, col)

            # Draw the agent
            draw_agent(next_position)

            pygame.display.update()

            # Add a delay to visualize movement
            time.sleep(0.5)

    # Keep the window open until the user closes it manually
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
```

```
                sys.exit()

if __name__ == "__main__":
    main(num_steps=100, k=len(ACTIONS), epsilon=0.1)
```