SD LAB1: DESIGNING AN INTELLIGENT AGENT

NAME: EBA BOKALLI ANNA

DEPARTMENT: SOFTWARE ENGINEERING

Campus B

Introduction

Main objective: Creating a simple intelligent agent that can navigate a 2D grid world, sense obstacles, and find a target using decision-making techniques.

Steps:

1. Environment Creation
2. Agent Design
3. Agent Simulation Loop
4. Pathfinding Enhancement

1. Environment Creation

```python
import pygame
import random
from collections import deque

# Constants
GRID_SIZE = 20
CELL_SIZE = 30
WINDOW_SIZE = GRID_SIZE * CELL_SIZE
OBSTACLE_COUNT = 40
REWARD = 10  # Reward for moving towards the target
PENALTY = -5  # Penalty for hitting an obstacle

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
BLUE = (0, 0, 255)

# Initialize Pygame
pygame.init()
window = pygame.display.set_mode((WINDOW_SIZE, WINDOW_SIZE))
pygame.display.set_caption("Grid World Agent with Rewards")

# Create the grid
grid = [[0 for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]

# Place obstacles
for _ in range(OBSTACLE_COUNT):
    x, y = random.randint(a: 0, GRID_SIZE - 1), random.randint(a: 0, GRID_SIZE - 1)
    if grid[y][x] == 0:  # Ensure no overlap with existing obstacles
```

- **Importing Libraries**: We import Pygame for graphics and deque from collections for BFS.
- **Constants:** We define constants for grid size, cell size, and colors.
- **Initializing Pygame**: We set up the display window using Pygame.
- Creating **the Grid**: We create a 2D list to represent the grid, initializing all cells to 0 (open space).
- **Placing Obstacles**: The `place_obstacles` function randomly places a set number of obstacles (marked as 1) in the grid.

• **Agent and Target Positions**: The agent starts at the top-left corner (0,0), and the target is at the bottom-right corner (GRID_SIZE-1, GRID_SIZE-1), which is marked with a 2.

2) Agent Design

```
1 usage
class Agent:
    def __init__(self, grid, start_pos):
        self.grid = grid
        self.position = start_pos
        self.score = 0  # Initialize score

    1 usage
    def decide(self):
        """Use BFS to find the next move towards the target."""
        target = target_pos
        queue = deque([self.position])
        visited = {self.position: None}

        while queue:
            current = queue.popleft()
            if current == target:
                break

            x, y = current
            for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                neighbor = (x + dx, y + dy)
                if 0 <= neighbor[0] < GRID_SIZE and 0 <= neighbor[1] < GRID_SIZE:
                    if self.grid[neighbor[1]][neighbor[0]] != 1 and neighbor not in visited:
                        visited[neighbor] = current
                        queue.append(neighbor)
```

We define an `Agent` class to encapsulate its behaviour.

- **Initialization**: The agent is initialized with the grid and its starting position.
- **Sensing**: The `sense` method checks surrounding cells (up, down, left, right) and returns their values.
- **Moving**: The `move` method updates the agent's position based on the given direction.
- **Decision-Making**: The `decide` method uses BFS to find the path to the target. It explores neighbouring cells and tracks visited cells to avoid cycles. Once it reaches the target, it backtracks to construct the path and returns the next position.

3) Agent Simulation Loop

```
                neighbor = (x + dx, y + dy)
                if 0 <= neighbor[0] < GRID_SIZE and 0 <= neighbor[1] < GRID_SIZE:
                    if self.grid[neighbor[1]][neighbor[0]] != 1 and neighbor not in visited:
                        visited[neighbor] = current
                        queue.append(neighbor)

        # Backtrack to find the path
        path = []
        while current:
            path.append(current)
            current = visited[current]
        path.reverse()

        # Move to the next position in the path
        if len(path) > 1:
            next_pos = path[1]  # Next position towards the target
            # Update score for moving towards the target
```

```python
# Main Loop
agent = Agent(grid, agent_pos)

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    window.fill(WHITE)

    # Draw the grid
    for y in range(GRID_SIZE):
        for x in range(GRID_SIZE):
            rect = pygame.Rect(x * CELL_SIZE, y * CELL_SIZE, CELL_SIZE, CELL_SIZE)
            if grid[y][x] == 1:
                pygame.draw.rect(window, BLACK, rect)  # Obstacle
            elif grid[y][x] == 2:
                pygame.draw.rect(window, RED, rect)  # Target
            elif (x, y) == agent.position:
                pygame.draw.rect(window, BLUE, rect)  # Agent
            pygame.draw.rect(window, GREEN, rect, width: 1)  # Grid lines
```

Main Loop: The loop continues until the user closes the window.

- **Event Handling**: It checks for quit events to stop the loop.
- **Drawing**: The grid is drawn based on the state of each cell (obstacle, target, and agent).
- **Agent Decision**: The agent decides its next move using the `decide` method.
- **Display Update**: The display is updated, and a delay is added for visual effect.

OUTPUT

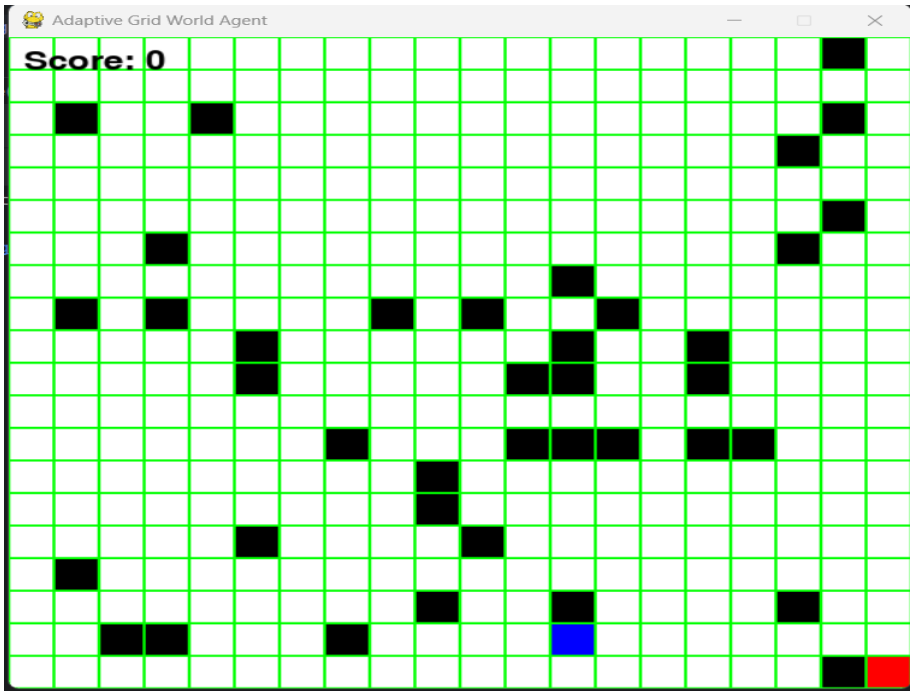

Figure 1: showing the agent on grid with obstacles

Reward Allocations

```
def update_score(self, next_pos):
    """Update the agent's score based on the next position."""

    if next_pos == target_pos:  # Check if the next position is the target
        self.score += REWARD  # Increase the score by the defined reward amount for reaching the target
    elif self.grid[next_pos[1]][next_pos[0]] == 1:  # Check if the next position is an obstacle
        self.score += PENALTY  # Decrease the score by the penalty amount for hitting an obstacle
```

• **Function Purpose**: The `update score` method determines how the agent's score should change based on its next position.

• **Reward for Target**:

- `If next_pos == target_pos:`: Checks if the agent's next move is to the target position.
- `Self. Score += REWARD`: If true, the agent receives a reward for reaching the target, increasing its score.

• **Penalty for Obstacles**:

- `elif self.grid[next_pos[1]][next_pos[0]] == 1:`: Checks if the next position is an obstacle.
- `self.score += PENALTY`: If true, the agent incurs a penalty, reducing its score.

OUTPUT

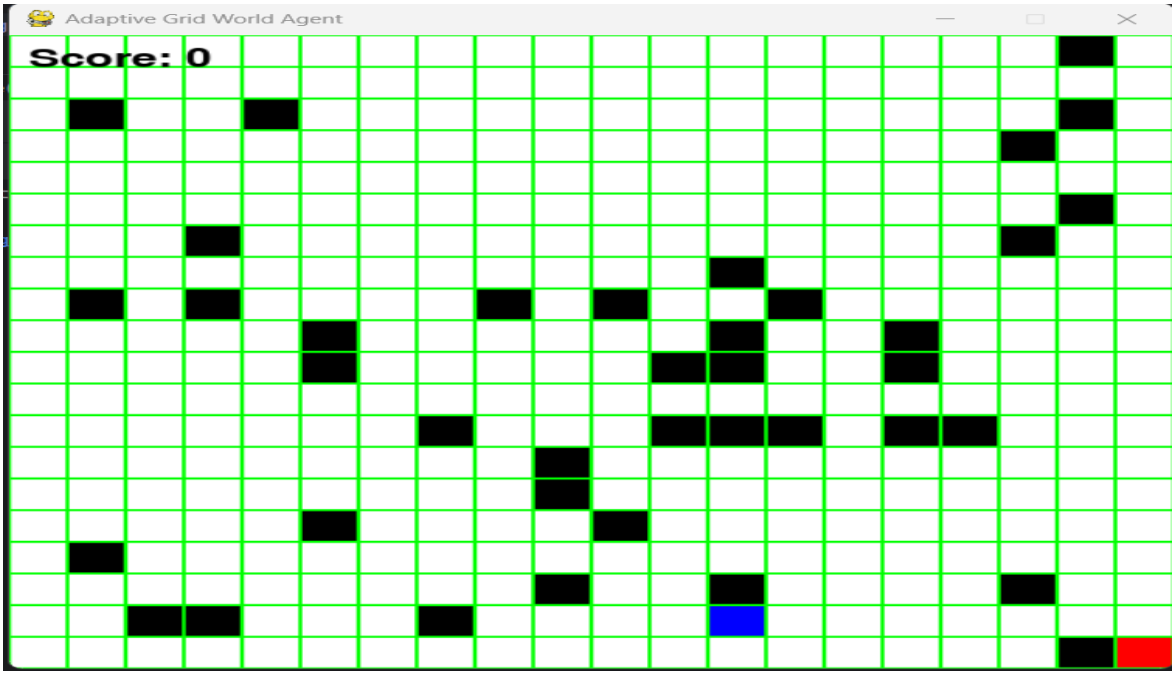

Figure2: showing the reward in red allocated for the agent.

Pathfinding Enhancements

1. **Breadth-First Search (BFS) Algorithm**:
    - **Purpose**: Used to find the shortest path from the agent's current position to the target.
    - **Exploration**: BFS explores all possible paths layer by layer, ensuring that the shortest path is found first.
2. **Dynamic Path Evaluation**:
    - **Real-Time Decisions**: The agent re-evaluates its path each time it makes a move, allowing it to adapt to changes in the grid (e.g., if obstacles are encountered).
    - **Queue Structure**: Utilizes a queue (`deque`) to efficiently manage the current positions being explored.
3. **Visited Tracking**:
    - **Tracking Visited Cells**: A dictionary (`visited`) keeps track of cells that have already been explored, preventing the agent from revisiting them and thus avoiding infinite loops.
4. **Backtracking for Path Reconstruction**:
    - **Path Creation**: Once the target is reached, the algorithm backtracks through the `visited` dictionary to reconstruct the path from the target to the agent's current position.
    - **Next Move Selection**: The agent selects the next position to move based on the reconstructed path, ensuring it moves towards the target.
5. **Obstacle Avoidance**:
    - **Conditional Movement**: The BFS algorithm avoids cells marked as obstacles (`1`), ensuring that the agent does not attempt to move into these cells.
    - **Adaptive Learning**: If the agent encounters an obstacle, the BFS will explore alternative paths in subsequent iterations.
6. **Feedback Integration**:
    - **Score Updates**: The agent receives rewards for reaching the target and penalties for hitting obstacles, encouraging it to choose paths that maximize its score over time.

```python
def decide(self):
    """Use BFS to find the next move towards the target."""

    target = target_pos  # Set the target position to the defined target
    queue = deque([self.position])  # Initialize the queue with the agent's current position
    visited = {self.position: None}  # Track visited cells and their predecessors

    # Start the BFS loop
    while queue:
        current = queue.popleft()  # Dequeue the first position to explore
        if current == target:  # Check if the current position is the target
            break  # Exit the loop if the target is reached

        x, y = current  # Deconstruct the current position into x and y coordinates
        # Explore each of the four possible directions (up, down, left, right)
        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            neighbor = (x + dx, y + dy)  # Calculate the neighbor's position
            # Check if the neighbor is within grid bounds
            if 0 <= neighbor[0] < GRID_SIZE and 0 <= neighbor[1] < GRID_SIZE:
                # Ensure the neighbor is not an obstacle and hasn't been visited
                if self.grid[neighbor[1]][neighbor[0]] != 1 and neighbor not in visited:
                    visited[neighbor] = current  # Mark the neighbor as visited and record its predecessor
                    queue.append(neighbor)  # Add the neighbor to the queue for further exploration

    # Backtrack to find the path from target to agent's current position
    path = []  # Initialize an empty list to store the path
    while current:  # Continue until there are no predecessors left
        path.append(current)  # Add the current position to the path
        current = visited[current]  # Move to the predecessor of the current position
    path.reverse()  # Reverse the path to get it from the agent to the target
    # Choose the next position based on the path
```

```python
    # Choose the next position based on the path
    if len(path) > 1:  # Ensure there is a next position to move to
        next_pos = path[1]  # Set the next position to the second position in the path
        self.update_score(next_pos)  # Update the score based on the next position
        return next_pos  # Return the next position for the agent to move to
    return self.position  # If no valid move, stay in the current position
```

OUTPUT

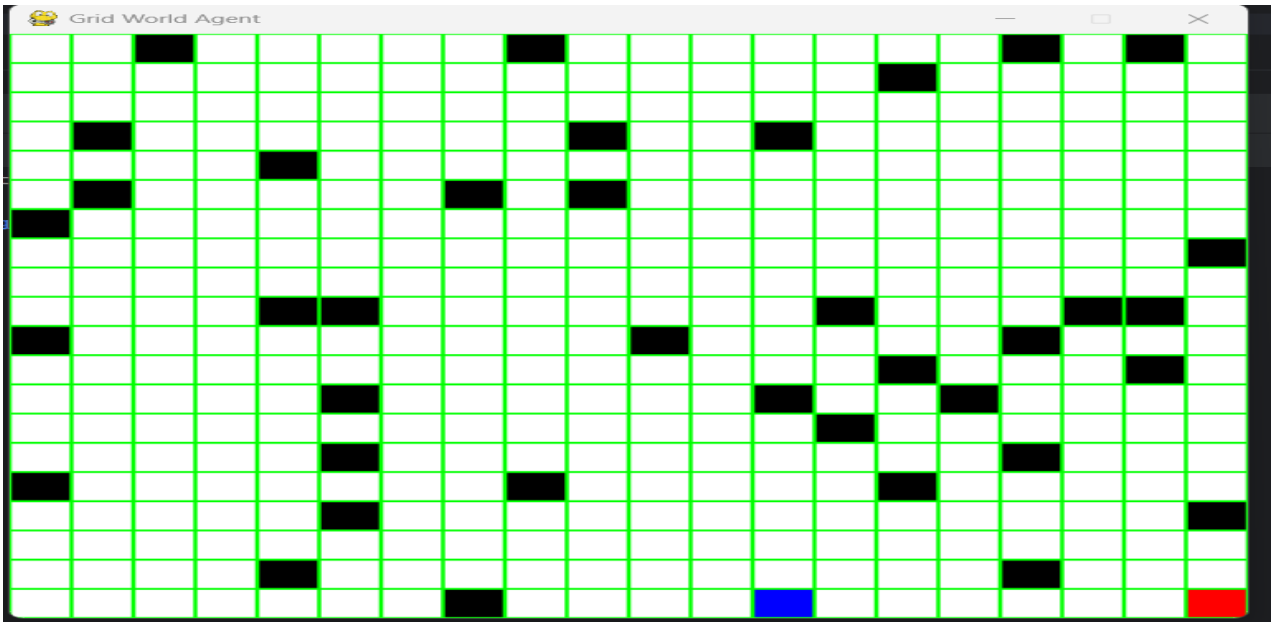

Figure 3: Agent moving toward the reward

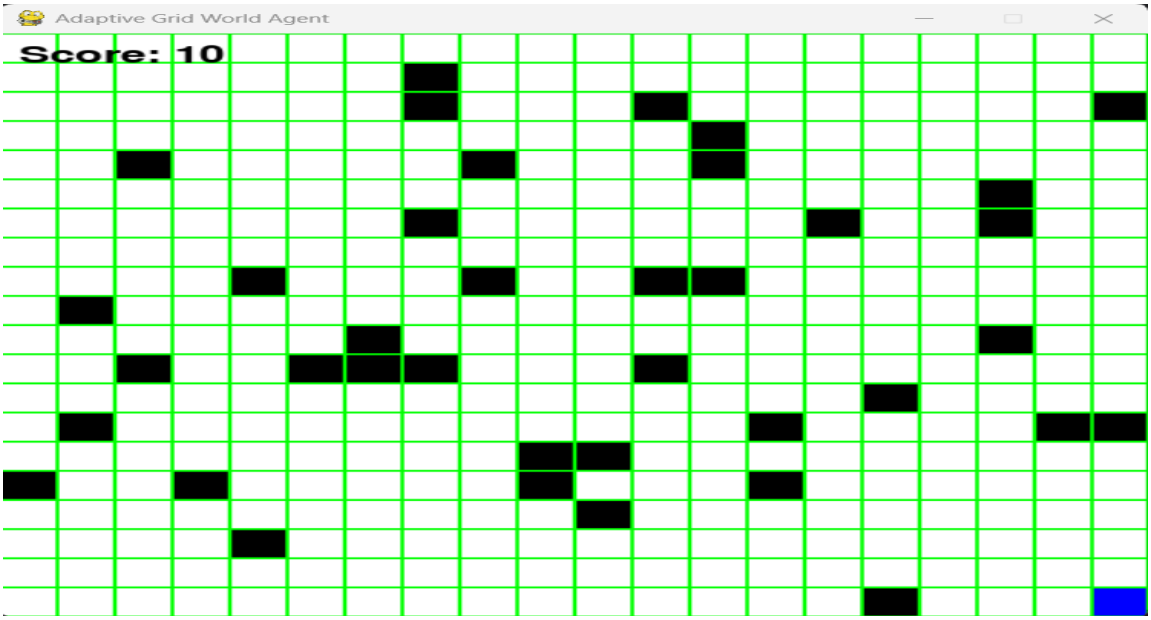

Figure4: Agent has found the reward and score a 10.