<u>TOPIC – TREASURE HUNT GAME AI PROJECT</u>

CLASS – TYBSCIT

DIV – B

TEAM MEMBER –
SRUSHTI CHAUHAN – 53003230097
DEV SHETA – 53003230073
HETVI PARMAR – 53003230067
KASAK MEHTA - 53003230071

Treasure Hunt AI Project Documentation

1. Problem Statement

This project involves developing an interactive two-player treasure hunt game set on a **10x10 grid**. Each player controls an agent aiming to locate and reach a treasure placed at a significant distance from both agents.

The players can:

- **Manually navigate** their agents using keyboard controls.
- **Strategically place obstacles** to block their opponent's path.

Additionally, agents can **automatically move step-by-step** toward the treasure using the **Breadth-First Search (BFS)** pathfinding algorithm, which finds the shortest path while avoiding obstacles and the opponent.

The challenge is to **reach the treasure first** while navigating around dynamically placed obstacles and competing against the opponent's strategic moves.

2. Scope

- **Grid Environment**:
  - The game operates on a fixed 10x10 grid representing discrete cells.
- **Agents**:
  - Two distinct agents start at opposite corners of the grid:
    - Agent 1 → Top-left corner.
    - Agent 2 → Bottom-right corner.
- **Treasure Placement**:
  - The treasure is placed randomly but at a minimum **Manhattan distance** from both agents.
- **Manual Control**:
  - Players can manually move agents using keyboard controls:
    - WASD → Agent 1.
    - Arrow Keys → Agent 2.
- **AI-Assisted Movement**:
  - Agents can perform an automatic single-step BFS move toward the treasure.
- **Obstacle Mechanics**:
  - Obstacles block paths.
  - Cannot overwrite agents or the treasure.
  - Must be placed adjacent to the agent.
- **Win Conditions**:
  - The game ends when either or both agents reach the treasure.
- **Visualization**:
  - The game uses **Pygame** for real-time graphical rendering and input handling

3. Brief Overview of Existing Approaches or PriorWork

- **Search Algorithms**:
  BFS, DFS, Dijkstra, and A* are fundamental algorithms used for path finding in grid environments. BFS guarantees the shortest path in an unweighted grid, making it suitable for this project.
- **Competitive Multi-Agent Systems**:
  AI agents competing on grids often use adversarial search algorithms such as Minimax or employ reinforcement learning techniques to improve strategies.
- **Obstacle Navigation**:
  Dynamic environments with obstacles require agents to adapt their paths or strategies, sometimes recalculating routes in real-time.
- **Interactive AI Games**:
  Pygame is a popular framework for creating educational AI demonstrations and interactive grid-world games, providing easy-to-use tools for rendering and input handling.

This project leverages BFS as a reliable and straightforward approach for shortest pathfinding and combines manual interaction for obstacle placement and movement, enabling human vs AI or human vs human gameplay.

4. Theoretical Concepts for Solving the Problem

Breadth-First Search (BFS)

- Explores nodes layer by layer, ensuring the shortest path in unweighted grids.
- Maintains a queue for exploration and a map for predecessors (for path reconstruction).
- Once the treasure cell is reached, BFS reconstructs the sequence of moves.

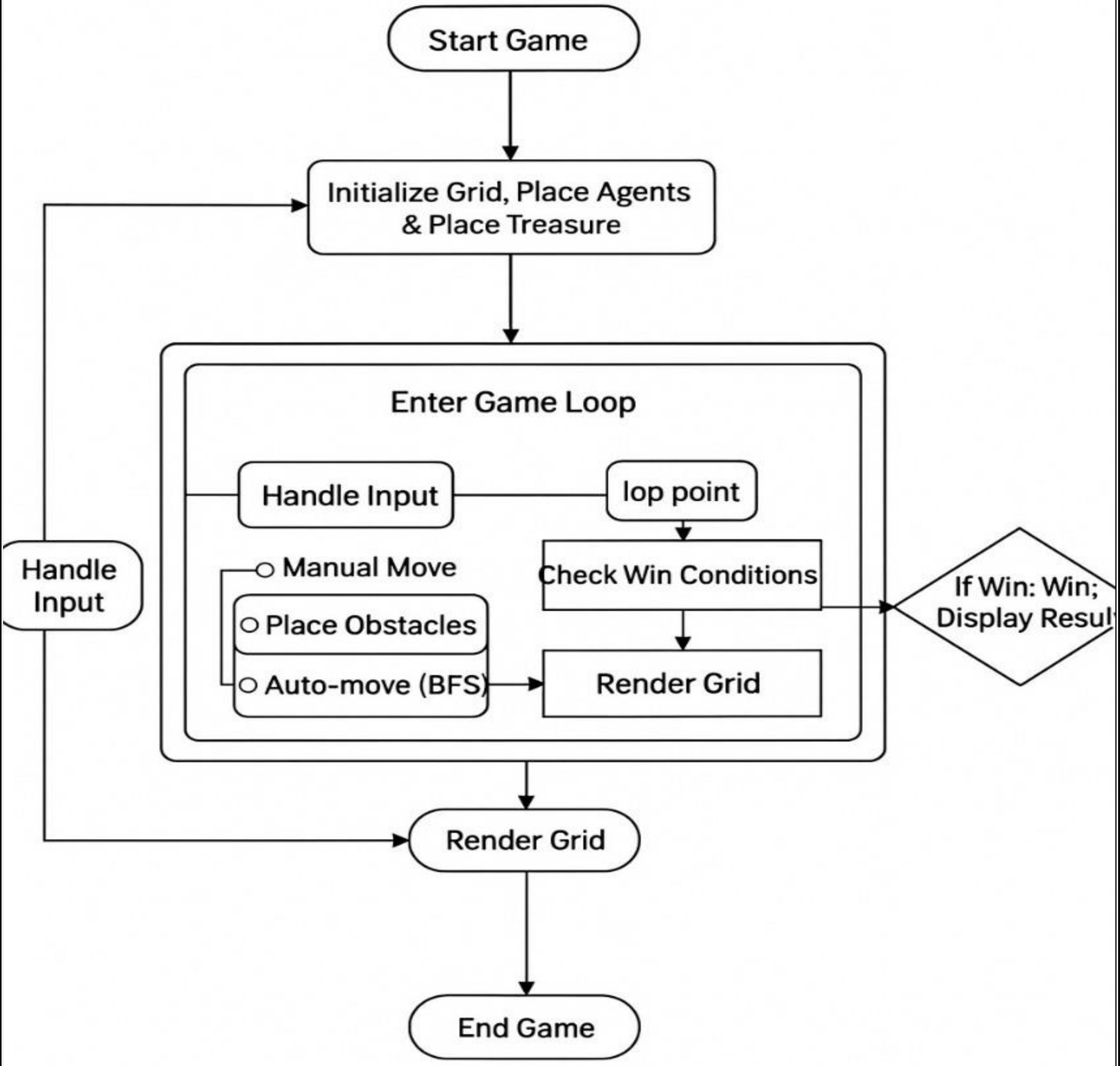
Manhattan Distance

- Measures distance as the sum of absolute differences in coordinates.
- Used to ensure treasure placement is **not too close** to agents.

Manual Controls & Obstacle Placement

- Agents move one cell at a time in four directions (up, down, left, right).
- Obstacles:
  - Can only be placed adjacent to the agent.
  - Cannot overwrite the treasure, another agent, or outside the grid.

5. Flowchart


Start Game
Initialize Grid, Place Agents & Place Treasure
Enter Game Loop
Handle Input
lop point
Manual Move
Check Win Conditions
Place Obstacles
Auto-move (BFS)
Render Grid
If Win: Win; Display Resul
Handle Input
Render Grid
End Game

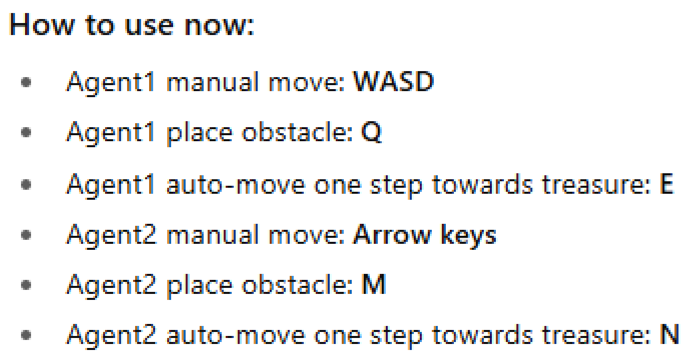
6. Data Collection/Dataset Description

This project does not utilize external datasets. The game state-including grid cells, agent positions, obstacle placements, and treasure location-is managed internally and dynamically updated at run time. Treasure placement is randomized each game start but constrained by Manhattan distance to the agents.

7. Evaluation Metrics

- **Win Condition Accuracy**: Detect when agents reach treasure.
- **Pathfinding Efficiency**: BFS guarantees shortest paths.
- **Game Responsiveness**: Smooth input handling and visualization.
- **Strategic Complexity**: Effective obstacle placement.
- **User Engagement**: Clear graphics and intuitive controls.

- **Controls:**



How to use now:

- Agent1 manual move: **WASD**
- Agent1 place obstacle: **Q**
- Agent1 auto-move one step towards treasure: **E**
- Agent2 manual move: **Arrow keys**
- Agent2 place obstacle: **M**
- Agent2 auto-move one step towards treasure: **N**

8. AlgorithmComplexities

- BFS Pathfinding

- **Time Complexity**: O(V + E)
  - V = number of cells = 100 (10x10 grid).
  - E = number of edges ≈ 4V (each cell connects to ~4 neighbors).
- **Space Complexity**: O(V) (for visited nodes and queue).

- Movement & Obstacle Placement

- Constant time O(1) checks for valid moves or obstacle placement.

-PEAS parameter

- **P (Performance Measure):**
- Reach treasure in minimum number of moves.
- Fewer turns = better performance.

- **E (Environment):**
- 10×10 grid with treasure hidden in one cell.
- May include obstacles if extended.
- **A (Actuators):**
- Player’s possible moves (up, down, left, right).
- Guessing/selecting a cell.
- **S (Sensors):**
- Feedback system: Manhattan distance
- Player knows current position and hints.

9. TestCases

- **TC1**: Treasure placed at minimum Manhattan distance from both agents – Treasure not too close to agents.

- **TC2**: Agent 1 moves into empty adjacent cell – Agent 1 successfully moves.

- **TC3**: Agent 1 attempts to move into obstacle or agent – Movement blocked, position unchanged.

- **TC4**: Agent 1 places obstacle adjacent to self – One obstacle placed in an empty adjacent cell.

- **TC5**: Agent 2 performs auto-move with clear BFS path – Agent 2 moves one step towards treasure.

- **TC6**: Agent 2 auto-move with no path available due to obstacles – Agent 2 stays in place.

- **TC7**: Both agents reach treasure simultaneously – Game ends with message indicating both reached.

- **TC8**: Agents attempt to move outside grid bounds – Movement prevented.

10. Conclusion

This project successfully developed a fully functional and interactive two-player treasure hunt game, effectively demonstrating the integration of fundamental AI pathfinding with strategic gameplay. The core objective — to create a competitive environment on a 10x10 grid where players locate a treasure while navigating and placing obstacles — was fully achieved.

The implementation of the Breadth-First Search (BFS) algorithm proved to be an excellent choice for the AI-assisted movement feature. Given the unweighted and relatively small grid, BFS consistently and efficiently provides the guaranteed shortest path, allowing for responsive, real-time gameplay without performance overhead. The successful integration of manual controls (movement and obstacle placement) with the automated BFS step creates a unique and engaging dynamic. This hybrid approach allows the game to serve as both an entertaining strategic challenge and a clear, practical demonstration of a classic search algorithm.

The use of Pygame provided a robust framework for visualization and user interaction, making the game mechanics intuitive and the agents' behavior easy to observe. The project met all its specified scope requirements, from the randomized treasure placement using Manhattan distance to the accurate handling of win conditions and game rules.

11.Future Work

While the current implementation is a complete success as a proof-of-concept, several avenues exist for future enhancement:

- **Advanced Adversarial AI**:
  The AI could be improved by incorporating algorithms like Minimax to not only find its own best path but also to strategically place obstacles to optimally block the human opponent.
- **Dynamic Environments**:
  Introducing elements like moving obstacles, different terrain types with varying movement costs (requiring Dijkstra's or A* algorithm), or power-ups could add more complexity.
- **Reinforcement Learning**:
  An agent could be trained using reinforcement learning to develop sophisticated strategies for both movement and obstacle placement over thousands of simulated games.

12. CODE

```
import pygame

import random

import time

from collections import deque



# Constants

GRID_SIZE = 10

CELL_SIZE = 60

WIDTH = HEIGHT = GRID_SIZE * CELL_SIZE

FPS = 10

MIN_TREASURE_DIST = 6  # minimum Manhattan distance from agents



# Colors

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

TREASURE_COLOR = (255, 215, 0)

AGENT1_COLOR = (0, 128, 255)
```

```
AGENT2_COLOR = (255, 0, 0)

OBSTACLE_COLOR = (128, 128, 128)


# Game symbols

EMPTY, TREASURE, AGENT1, AGENT2, OBSTACLE = '.', 'T', 'A', 'B', 'X'


# Initialize pygame

pygame.init()

screen = pygame.display.set_mode((WIDTH, HEIGHT))

pygame.display.set_caption("Treasure Hunt - Manual + BFS Auto-Move")

clock = pygame.time.Clock()


def manhattan_dist(a, b):

    return abs(a[0] - b[0]) + abs(a[1] - b[1])


class Game:

    def __init__(self):

        self.grid = [[EMPTY for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]

        self.agent1 = (0, 0)

        self.agent2 = (GRID_SIZE - 1, GRID_SIZE - 1)

        self.place_treasure_far()

        self.grid[self.agent1[0]][self.agent1[1]] = AGENT1

        self.grid[self.agent2[0]][self.agent2[1]] = AGENT2

        self.running = True

    def place_treasure_far(self):
```

```
    while True:
        pos = (random.randint(0, GRID_SIZE - 1), random.randint(0, GRID_SIZE - 1))
        d1 = manhattan_dist(pos, self.agent1)
        d2 = manhattan_dist(pos, self.agent2)
        if d1 >= MIN_TREASURE_DIST and d2 >= MIN_TREASURE_DIST:
            self.treasure = pos
            self.grid[pos[0]][pos[1]] = TREASURE
            break

def can_move(self, pos, dx, dy):
    nx, ny = pos[0] + dx, pos[1] + dy
    if 0 <= nx< GRID_SIZE and 0 <= ny< GRID_SIZE:
        if self.grid[nx][ny] not in [OBSTACLE, AGENT1, AGENT2]:
            return True
    return False

def move_agent(self, agent_pos, dx, dy, symbol):
    if self.can_move(agent_pos, dx, dy):
        nx, ny = agent_pos[0] + dx, agent_pos[1] + dy
        self.grid[agent_pos[0]][agent_pos[1]] = EMPTY
        agent_pos = (nx, ny)
        self.grid[nx][ny] = symbol
    return agent_pos

def place_obstacle(self, agent_pos):
    directions = [(-1,0), (1,0), (0,-1), (0,1)]
    for dx, dy in directions:
        nx, ny = agent_pos[0] + dx, agent_pos[1] + dy
        if 0 <= nx< GRID_SIZE and 0 <= ny< GRID_SIZE:
            if self.grid[nx][ny] == EMPTY and (nx, ny) != self.treasure:
                self.grid[nx][ny] = OBSTACLE
                break

def bfs(self, start, goal, agent_symbol):
    queue = deque([start])
    visited = {start: None}

    while queue:
        current = queue.popleft()
        if current == goal:
            # Reconstruct path from goal to start
            path = []
            while current is not None:
                path.append(current)
                current = visited[current]
            path.reverse()
            return path  # path includes start and goal

        for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:
            nx, ny = current[0] + dx, current[1] + dy
            if 0 <= nx< GRID_SIZE and 0 <= ny< GRID_SIZE:
```

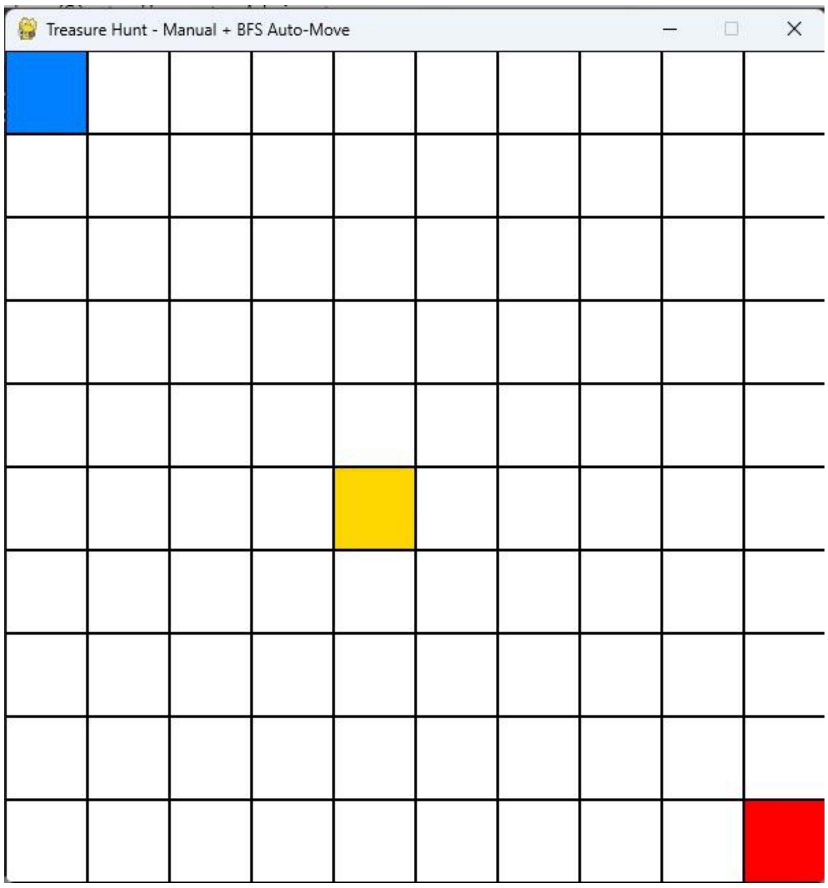
```
if (nx, ny) not in visited:
                    cell = self.grid[nx][ny]
                    # Can step on EMPTY or TREASURE but NOT on obstacles or other agent
                    if cell in [EMPTY, TREASURE] or (nx, ny) == goal:
                        # Avoid blocking own position or opponent's position
                        # But allow moving into treasure cell
                        if agent_symbol == AGENT1 and cell != AGENT2:
                            visited[(nx, ny)] = current
                            queue.append((nx, ny))
                        elifagent_symbol == AGENT2 and cell != AGENT1:
                            visited[(nx, ny)] = current
                            queue.append((nx, ny))
        return None  # no path found

    def auto_move_agent(self, agent_pos, symbol):
        path = self.bfs(agent_pos, self.treasure, symbol)
        if path and len(path) > 1:
            next_step = path[1]
            dx = next_step[0] - agent_pos[0]
            dy = next_step[1] - agent_pos[1]
            return self.move_agent(agent_pos, dx, dy, symbol)
        return agent_pos

    def draw_grid(self):
        screen.fill(WHITE)
```

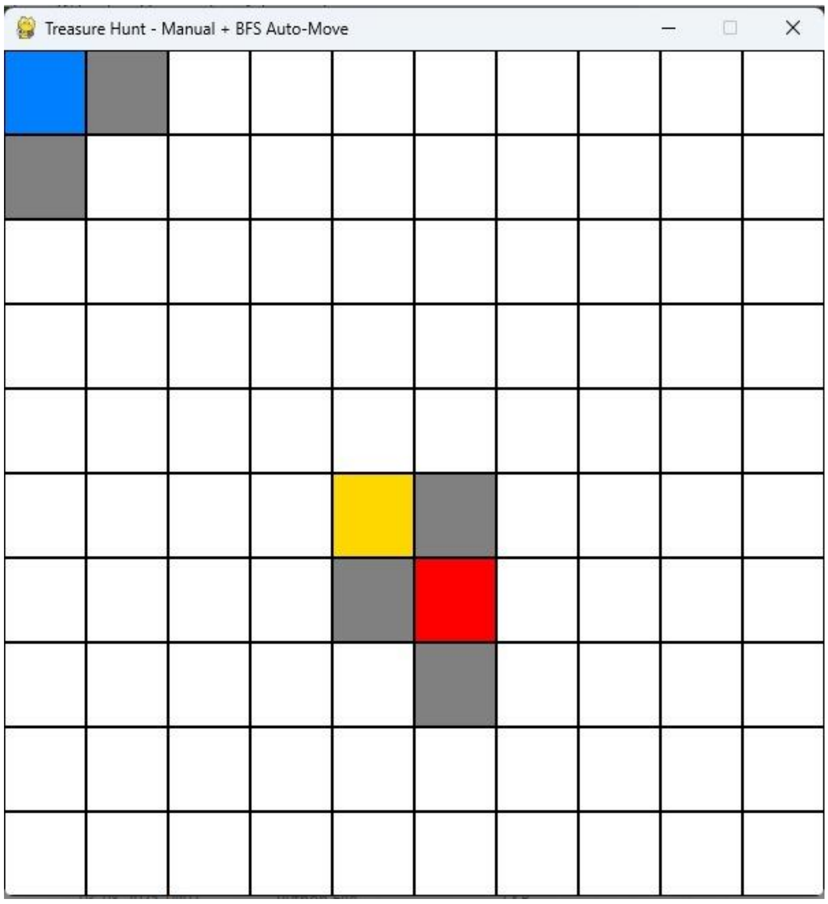
```
for row in range(GRID_SIZE):
     for col in range(GRID_SIZE):
       rect = pygame.Rect(col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE, CELL_SIZE)
       cell = self.grid[row][col]
       if cell == TREASURE:
         pygame.draw.rect(screen, TREASURE_COLOR, rect)
       elif cell == AGENT1:
         pygame.draw.rect(screen, AGENT1_COLOR, rect)
       elif cell == AGENT2:
         pygame.draw.rect(screen, AGENT2_COLOR, rect)
       elif cell == OBSTACLE:
         pygame.draw.rect(screen, OBSTACLE_COLOR, rect)
       pygame.draw.rect(screen, BLACK, rect, 1)

 def play(self):
   while self.running:
     clock.tick(FPS)
     for event in pygame.event.get():
       if event.type == pygame.QUIT:
         self.running = False
       elifevent.type == pygame.KEYDOWN:
         # Agent1 manual move WASD
         if event.key == pygame.K_w:
           self.agent1 = self.move_agent(self.agent1, -1, 0, AGENT1)
         elifevent.key == pygame.K_s:
```

```
self.agent1 = self.move_agent(self.agent1, 1, 0, AGENT1)
            elifevent.key == pygame.K_a:
                self.agent1 = self.move_agent(self.agent1, 0, -1, AGENT1)
            elifevent.key == pygame.K_d:
                self.agent1 = self.move_agent(self.agent1, 0, 1, AGENT1)
            # Agent1 place obstacle Q
            elifevent.key == pygame.K_q:
                self.place_obstacle(self.agent1)
            # Agent1 auto move step E
            elifevent.key == pygame.K_e:
                self.agent1 = self.auto_move_agent(self.agent1, AGENT1)

            # Agent2 manual move Arrows
            elifevent.key == pygame.K_UP:
                self.agent2 = self.move_agent(self.agent2, -1, 0, AGENT2)
            elifevent.key == pygame.K_DOWN:
                self.agent2 = self.move_agent(self.agent2, 1, 0, AGENT2)
            elifevent.key == pygame.K_LEFT:
                self.agent2 = self.move_agent(self.agent2, 0, -1, AGENT2)
            elifevent.key == pygame.K_RIGHT:
                self.agent2 = self.move_agent(self.agent2, 0, 1, AGENT2)
            # Agent2 place obstacle M
            elifevent.key == pygame.K_m:
                self.place_obstacle(self.agent2)
            # Agent2 auto move step N
```

```
elifevent.key == pygame.K_n:
                        self.agent2 = self.auto_move_agent(self.agent2, AGENT2)

            # Win check
            if self.agent1 == self.treasure and self.agent2 == self.treasure:
                print("□ Both reached the treasure!")
                self.running = False
            elifself.agent1 == self.treasure:
                print("□ Agent1 Wins!")
                self.running = False
            elifself.agent2 == self.treasure:
                print("□ Agent2 Wins!")
                self.running = False

            self.draw_grid()
            pygame.display.flip()

        time.sleep(2)
        pygame.quit()

if __name__ == "__main__":
    Game().play()
```

OUTPUT:-


Treasure Hunt - Manual + BFS Auto-Move


Treasure Hunt - Manual + BFS Auto-Move