DRL HW2 說明報告

1. 作業要求

(1) 目標

- 使用價值迭代算法 (Value Iteration Algorithm) 計算 Gridworld 環境的最佳政策 (Optimal Policy)。
- 在給定起點 (Start)、終點 (End) 和障礙物 (Obstacles) 的條件下，推導最佳行動策略。
- 計算價值函數 (Value Function V(s))，表示在最佳政策下，每個狀態的期望回報。
- 透過動畫模擬學習過程，並在動畫結束後將最優路徑標記成 黃色。

(2) 預期結果

- 最佳政策顯示：
  - 每個格子顯示最佳行動 (↑、↓、←、→)，代表最優移動方向。
- 價值函數顯示：
  - 每個格子顯示 價值函數 V(s)，表示該狀態的最優回報。
- 動畫模擬路徑：
  - 根據策略矩陣，模擬學習過程。
  - 動畫結束後，將最終最佳路徑標記為黃色 🏆。

---

2. 作法說明

ChatGPT Prompt

> 📝 我現在的程式碼功能如下：
>
> 1. 預設生成 5×5 的 square，允許使用者選擇 start, end, obstacles cells，點擊 "Start Game" 後，計算並顯示正確的 Value Matrix 和 Policy Matrix。
> 2. 使用者可輸入 6~9 之間的數字，點擊 "Generate Square" 生成對應大小的 square，並選擇 start, end, obstacles cells，然後點擊 "Start Game" 正確產生 Value Matrix 和 Policy Matrix。
>
> 現在，我希望新增以下新功能，請基於我的原始程式碼 (不要移除原有功能) 進行擴展：
>
> 1. 新增一個視窗 (modal)，當點擊 "Start Game" 後彈出此視窗，視窗內包含一個與原 square 大小相同的 grid，並保留使用者選擇的 start, end, obstacles cells 的顏色。視窗樣式可參考附圖一二。
> 2. 當執行 find_best_path() 時，每次 iteration 都會產生 Value Matrix 和 Policy Matrix，並根據該次 iteration 找到一條或多條最佳路徑 (從 start cell 到 end cell)。
> 3. 在視窗中模擬這些最佳路徑，每次從 start cell 開始，start cell 以綠色表示，根據 Policy Matrix，若往右走是最好的選擇，則將下一格設為綠色，並依此類推，每一步停留 0.3 秒，直到走到 end cell。
> 4. 確保所有的顯示與運算邏輯都與原功能相容，不要影響原先的 Value Matrix 和 Policy Matrix 計算與顯示功能。
>
> 請基於我的程式碼實現這個功能，並提供完整的程式碼，確保視窗能夠正確顯示與運行，並詳細說明你的實作方式！

本專案透過 Flask (後端) + HTML/CSS/JavaScript (前端) 搭配 價值迭代算法 (Value Iteration Algorithm) 來計算 最佳政策 (Optimal Policy)，並進一步模擬最佳路徑的移動過程。以下是主要的實作步驟：

---

(1) Flask 後端

負責計算最佳政策 (Policy Matrix) 和價值函數 (Value Matrix)，主要步驟如下：

1. 接收前端輸入的 Gridworld 配置 (起點、終點、障礙物)。
2. 執行價值迭代算法 (Value Iteration)
   - 迭代更新每個狀態 sss 的 價值函數 V(s)V(s)V(s)，直到收斂。
   - 根據每個狀態的最佳行動，建立 最佳政策矩陣 (Policy Matrix)。
3. 返回計算結果至前端：
   - Value Matrix：顯示每個格子的期望回報值。
   - Policy Matrix：標示每個格子最優行動的方向 (↑、↓、←、→)。
   - 所有迭代過程中的最佳路徑集合，供動畫模擬。

---

(2) JavaScript 前端

負責動態更新 UI 並模擬最佳路徑移動動畫，主要步驟如下：

1. 建立互動式 Gridworld
   - 使用 `div` 元素產生 n×nn \times nn×n 的網格 (`grid`)，並允許使用者點擊選擇 起點、終點、障礙物。
   - 點擊「Generate Square」時，重新生成不同大小的網格。
2. 發送請求至 Flask 後端，取得最佳政策與價值函數
   - 在點擊「Start Game」後，向 `/find_best_path` 端點發送請求。
   - 伺服器返回 Policy Matrix、Value Matrix 和最佳路徑，更新 UI。
3. 更新 UI
   - 顯示計算結果
     - 使用 `innerHTML` 將 最佳政策 (箭頭) 和價值函數 顯示於網格內。
   - 動態模擬最佳路徑
     - 動畫顯示最佳移動路徑 (0.3s 間隔)
       - 使用 `setTimeout()` 依序高亮每一步移動方向，從 起點 (綠色) → 沿最佳政策移動 → 終點 (紅色)。
     - 最終標記最佳路徑
       - 動畫結束後，將最優路徑標記為金黃色 (🏆)，視覺化最短路徑。

(新功能擴展)：當點擊 "Start Game" 時，彈出 模擬最佳路徑的視窗，其內部顯示：

1. 與主網格相同大小的 square，保留 start, end, obstacles 顏色。
2. 執行 100 次 Value Iteration，每次都會產生一組 Policy Matrix 和 Value Matrix。
3. 從這 100 次 iteration 中，每次隨機選擇一條最佳路徑，並以 動畫方式顯示移動過程。
4. 每一步移動間隔 0.3 秒，當走到終點後，開始下一條最佳路徑的模擬，直到 100 條路徑模擬完成。

---

3. 程式碼說明

(1) `app.py` (後端)

🔷 1. Gridworld 設定與初始化

- `set_size()`：
  - 允許使用者輸入 Grid 大小 (5×5 至 9×9)，並初始化 起點 (Start)、終點 (End)、障礙物 (Obstacles)。

🔷 2. 執行價值迭代 (Value Iteration)

- `find_best_path()`：
  - 建立 `values` (價值函數矩陣)：
    - 終點 (`end`) 設為 1.0，代表最終獎勵值。
    - 障礙物 (`obstacles`) 設為 -1.0，表示無法通行的區域。
    - 其他格子的 `values` 初始為 0.0。
  - 建立 `policy` (策略矩陣)：
    - 每個格子預設為 " " (空白)，稍後透過價值迭代決定最優行動 (↑、↓、←、→)。

🔷 3. 進行價值迭代 (最多 100 次)

- 遍歷所有格子，更新 `values` 和 `policy`：
  1. 計算每個格子的最佳行動：
     - 嘗試向 四個方向 (↑, ↓, ←, →) 移動，並計算預期回報。
     - 選擇 回報值最高的行動，更新 `policy[r, c]`。
     - 使用折扣因子 γ=0.9\gamma = 0.9γ=0.9 來平衡 短期 vs. 長期回報。
  2. 判斷是否收斂：
     - 如果 `values` 無變化，則提前結束迭代，避免不必要的運算。

🔷 4. 計算最佳路徑

- 對每次迭代 (`100 iterations`)：
  - 根據 `policy` 找出一條最佳路徑 (從 `start` 到 `end`)。
  - 將所有 `100` 次迭代產生的路徑儲存 (`all_iterations_paths`)。
  - 最後一次迭代的最佳路徑 (`final_best_path`) 作為動畫播放的主路徑。

🔷 5. 回傳結果至前端

當計算完成後，後端會回傳：

1. `value_matrix`：每個格子的價值函數 V(s)V(s)V(s)。
2. `policy_matrix`：最佳政策 (箭頭標示方向)。
3. `paths`：100 次價值迭代中，不同的最佳路徑。
4. `final_best_path`：最終最佳路徑 (動畫顯示用)。

```
from flask import Flask, render_template, request, jsonify
import numpy as np
import random

app = Flask(__name__)

GRID_SIZE = 5
GAMMA = 0.9  # 折扣因子
ACTIONS = {
    "↑": (-1, 0),
    "↓": (1, 0),
    "←": (0, -1),
```

```
    "→": (0, 1)
}

@app.route('/')
def index():
    return render_template('index.html', grid_size=GRID_SIZE)

@app.route('/set_size', methods=['POST'])
def set_size():
    """更新 GRID_SIZE"""
    global GRID_SIZE, START_CELL, END_CELL, OBSTACLES
    try:
        data = request.get_json()
        size = data.get("size")

        if 5 <= size <= 9:
            GRID_SIZE = size
            START_CELL = None
            END_CELL = None
            OBSTACLES = set()
            return jsonify({"status": "success", "grid_size": GRID_SIZE})

        return jsonify({"status": "error", "message": "Grid size must be between 5 and 9"}), 400

    except Exception as e:
        return jsonify({"status": "error", "message": str(e)}), 500

@app.route('/find_best_path', methods=['POST'])
def find_best_path():
    """計算 Value Matrix 和 Policy Matrix，當找到 end cell 就結束當前 iteration"""
    global GRID_SIZE, START_CELL, END_CELL, OBSTACLES

    try:
        data = request.get_json()
        if not data or 'start' not in data or 'end' not in data or 'obstacles' not in data or 'grid_size' not in data:
            return jsonify({"status": "error", "message": "Missing required parameters"}), 400

        start = tuple(data['start'])
        end = tuple(data['end'])
        obstacles = set(tuple(obs) for obs in data['obstacles'])
        GRID_SIZE = data['grid_size']

        values = np.zeros((GRID_SIZE, GRID_SIZE))
        policy = np.full((GRID_SIZE, GRID_SIZE), " ", dtype=object)

        values[end] = 1.0  # 設定終點獎勵

        for obs in obstacles:
            values[obs] = -1.0  # 設定障礙物

        all_iterations_paths = []
        final_best_path = []

        # Value Iteration
        for iteration in range(100):
```

```
        new_values = values.copy()
        updated = False  # 追蹤是否有更新

        for r in range(GRID_SIZE):
            for c in range(GRID_SIZE):
                if (r, c) == end or (r, c) in obstacles:
                    continue
                max_value = float('-inf')
                best_actions = []
                for action, (dr, dc) in ACTIONS.items():
                    nr, nc = r + dr, c + dc
                    if 0 <= nr < GRID_SIZE and 0 <= nc < GRID_SIZE and (nr, nc) not in obstacles:
                        value = GAMMA * values[nr, nc]
                        if value > max_value:
                            max_value = value
                            best_actions = [action]
                        elif value == max_value:
                            best_actions.append(action)
                if new_values[r, c] != max_value:
                    updated = True  # 如果有變化，設為 True
                new_values[r, c] = max_value
                policy[r, c] = "".join(best_actions)

        values = new_values

        # 如果 values 沒有變化，則提早結束迭代
        if not updated:
            break

        # 取得最佳路徑 (每次計算一條，並儲存最後一條)
        cur_pos = start
        path = []
        while cur_pos != end:
            path.append(cur_pos)
            actions = policy[cur_pos]
            if not actions:
                break
            chosen_action = random.choice(actions)
            dr, dc = ACTIONS[chosen_action]
            cur_pos = (cur_pos[0] + dr, cur_pos[1] + dc)
            if cur_pos in obstacles:
                break
            if cur_pos == end:
                path.append(end)
                break  # ✅ 抵達 end cell，結束當前 iteration

        all_iterations_paths.append(path)
        final_best_path = path  # 🚀 記錄最後一條最優路徑

    return jsonify({
        "status": "success",
        "value_matrix": values.tolist(),
        "policy_matrix": policy.tolist(),
        "paths": all_iterations_paths,
        "final_best_path": final_best_path  # 🔥 新增這個欄位！
```

```
        })

    except Exception as e:
        return jsonify({"status": "error", "message": str(e)}), 500


if __name__ == '__main__':
    app.run(debug=True)
```

(2) `index.html` (前端)

這是一個 Gridworld 可視化介面，使用 HTML + CSS + JavaScript (jQuery)，允許使用者 設定 Gridworld，執行 價值迭代 (Value Iteration)，並動態顯示最佳政策與模擬最佳路徑。

🔷 1. 主要功能

✅ 生成 Gridworld (5×5 到 9×9)，使用者可 選擇起點、終點、障礙物

✅ 執行價值迭代 (Value Iteration) 計算 最佳政策 (Policy Matrix) 和 價值函數 (Value Matrix)

✅ 動畫模擬最佳路徑移動過程，逐步顯示從 起點 → 終點 的最佳行動

🔷 2. 主要結構

1. 輸入 Grid 大小，點擊 "Generate Square" 生成 Gridworld
2. 點擊格子設定：
   - 起點 (Start)：綠色
   - 終點 (End)：紅色
   - 障礙物 (Obstacles)：灰色
3. 點擊 "Start Game"：
   - 發送 AJAX 請求至 Flask
   - 計算最佳政策 & 價值函數
   - 更新 Gridworld 顯示結果
   - 動畫模擬最佳路徑 (0.3 秒間隔)

🔷 3. 主要函數

1. `generateGrid()`：建立新的 Gridworld，允許使用者輸入 Grid 大小 (5~9)
2. `renderGrid()`：動態生成 Grid，允許使用者點擊設定 起點、終點、障礙物
3. `handleCellClick()`：處理點擊事件，設定 起點、終點、障礙物
4. `startGame()`：發送 AJAX 請求至 Flask，計算 最佳路徑
5. `renderMatrix()`：顯示 價值函數與最佳政策
6. `animatePaths()`：動畫顯示從 起點 → 終點 的移動過程
7. `highlightFinalPath()`：最終標記 最佳路徑 (金黃色🏆)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Gridworld</title>
```

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<style>
  body {
    font-family: 'Arial', sans-serif;
    text-align: center;
    margin: 40px;
    background-color: #f4f4f4;
  }

  h2 {
    font-size: 24px;
    font-weight: bold;
    color: #333;
  }

  h3 {
    font-size: 20px;
    font-weight: bold;
    color: #444;
  }

  input[type="number"] {
    width: 50px;
    text-align: center;
    padding: 5px;
    margin-right: 5px;
  }

  button {
    font-size: 16px;
    background-color: #888888;
    color: white;
    border: none;
    padding: 8px 15px;
    cursor: pointer;
    border-radius: 5px;
    transition: background 0.3s ease;
  }

  button:hover {
    background-color: #5a5a5a;
  }

  .grid-container {
    display: grid;
    column-gap: 18px; /* 只調整水平間距 */
    row-gap: 5px;    /* 可選：控制垂直間距 */
    margin-top: 10px;
    padding: 10px;
    border-collapse: collapse;
  }

  .grid-item {
    width: 50px;
    height: 50px;
```

```
    display: flex;
    justify-content: center;
    align-items: center;
    border: 1px solid black;
    font-size: 18px;
    cursor: pointer;
    font-weight: bold;
    transition: all 0.2s ease;
    border-radius: 5px; /* 設置圓角，數值越大圓角越明顯 */
}

/* ✅ 讓 Gridworld 置中 */
.grid-item:hover {
    transform: scale(1.1);
}

#grid-container {
    display: flex;
    justify-content: center;
    margin-top: 20px;
}

#matrix-container {
    display: flex;
    justify-content: center;
    align-items: flex-start;
    gap: 40px;
    margin-top: 30px;
}
#simulation-container {
    display: flex;
    justify-content: center;
    align-items: flex-start;
    gap: 40px;
    margin-top: 30px;
}
@media (max-width: 768px) {
    #matrix-container {
        flex-direction: column;
        align-items: center;
    }
}

.start {
    background-color: #64a866;
    color: white;
}

.end {
    background-color: #f5736a;
    color: white;
}

.obstacle {
    background-color: #9E9E9E;
```

```
        color: white;
      }
    </style>
</head>
<body>

<h2>Gridworld</h2>
<p>Enter a number between 5 and 9:
    <input type="number" id="grid-size" min="5" max="9">
    <button onclick="generateGrid()">Generate Square</button>
</p><br>
<p>
    a. Click on a cell to set up the start grid as green <br>
    b. Click on a cell to set up the end grid as red <br>
    c. Click on n-2 cells to set up the obstacle grid as gray
</p>

<!-- Gridworld 區域 -->
<div id="grid-container">
    <div>
        <h3><span id="grid-title">5 × 5 Square:</span></h3>
        <div id="grid" class="grid-container"></div><br>
        <button id="start-game" onclick="startGame()">Start Game</button>
    </div>
</div>

<div id="simulation-container">
    <!-- Maze Simulation -->
    <div id="maze-container">
        <h3>Simulation:</h3>
        <div id="maze-grid" class="grid-container"></div>
    </div>

    <!-- Value Matrix -->
    <div>
        <h3>Value Matrix (Final):</h3>
        <div id="value-matrix" class="grid-container"></div>
    </div>

    <!-- Policy Matrix -->
    <div>
        <h3>Policy Matrix (Final):</h3>
        <div id="policy-matrix" class="grid-container"></div>
    </div>
</div>

<script>
let gridSize = 5;
let startCell = null;
let endCell = null;
let obstacles = new Set();

function generateGrid() {
    gridSize = parseInt(document.getElementById("grid-size").value);
    if (gridSize < 5 || gridSize > 9 || isNaN(gridSize)) {
```

```
        alert("Please enter a valid number between 5 and 9.");
        return;
    }

    $.ajax({
        url: "/set_size",
        type: "POST",
        contentType: "application/json",
        data: JSON.stringify({ size: gridSize }),
        success: function(response) {
            if (response.status === "success") {
                startCell = null;
                endCell = null;
                obstacles.clear();
                renderGrid();
            } else {
                alert(response.message);
            }
        },
        error: function(xhr) {
            console.log("AJAX Error:", xhr.responseText);
            alert("Error setting grid size.");
        }
    });
}

function renderGrid() {
    let grid = document.getElementById("grid");
    grid.innerHTML = "";
    grid.style.gridTemplateColumns = `repeat(${gridSize}, 40px)`;

    for (let row = 0; row < gridSize; row++) {
        for (let col = 0; col < gridSize; col++) {
            let cell = document.createElement("div");
            cell.classList.add("grid-item");
            cell.textContent = row * gridSize + col + 1;
            cell.dataset.row = row;
            cell.dataset.col = col;
            cell.addEventListener("click", () => handleCellClick(cell));
            grid.appendChild(cell);
        }
    }
    document.getElementById("grid-title").textContent = `${gridSize} x ${gridSize} Square:`;
}

function handleCellClick(cell) {
    let row = parseInt(cell.dataset.row);
    let col = parseInt(cell.dataset.col);
    let cellPos = [row, col];

    if (startCell && startCell[0] === row && startCell[1] === col) {
        startCell = null;
        cell.className = "grid-item";
    } else if (endCell && endCell[0] === row && endCell[1] === col) {
        endCell = null;
```

```
        cell.className = "grid-item";
    } else if (obstacles.has(JSON.stringify(cellPos))) {
        obstacles.delete(JSON.stringify(cellPos));
        cell.className = "grid-item";
    } else if (!startCell) {
        startCell = cellPos;
        cell.className = "grid-item start";
    } else if (!endCell) {
        endCell = cellPos;
        cell.className = "grid-item end";
    } else if (obstacles.size < gridSize - 2) {
        obstacles.add(JSON.stringify(cellPos));
        cell.className = "grid-item obstacle";
    } else {
        alert(`You can only place up to ${gridSize - 2} obstacles.`);
    }
}

function startGame() {
    if (!startCell || !endCell) {
        alert("Please select both a start and end cell.");
        return;
    }

    let obstacleList = Array.from(obstacles).map(JSON.parse);

    $.ajax({
        url: "/find_best_path",
        type: "POST",
        contentType: "application/json",
        data: JSON.stringify({
            start: startCell,
            end: endCell,
            obstacles: obstacleList,
            grid_size: gridSize
        }),
        success: function(response) {
            if (response.status === "success") {
                renderMatrix("value-matrix", response.value_matrix, false);
                renderMatrix("policy-matrix", response.policy_matrix, true);

                let paths = response.paths;
                let finalPath = response.final_best_path; // 🆕 取得最優路徑

                showMaze();

                // ✅ 等待動畫播放完畢，再顯示最優路徑
                animatePaths(paths, () => {
                    highlightFinalPath(finalPath);
                });

            } else {
                alert(response.message);
            }
        },
```

```
        error: function(xhr) {
            alert("Error calculating matrices.");
        }
    });
}

function highlightFinalPath(finalPath) {
    for (let i = 0; i < finalPath.length; i++) {
        setTimeout(() => {
            let cell = finalPath[i];
            let nextCell = i < finalPath.length - 1 ? finalPath[i + 1] : null; // 下一步
            let cellElement = document.querySelector(`#maze-grid [data-row="${cell[0]}"][data-col="${cell[1]}"]`);

            if (cellElement) {
                // ✅ 確保起點 & 終點顏色不變，且終點不顯示箭頭
                if (JSON.stringify(cell) !== JSON.stringify(startCell) && JSON.stringify(cell) !== JSON.stringify(endCell
                    cellElement.style.backgroundColor = "#FFD700"; // 🏆 最優路徑標記為金黃色
                } else if (JSON.stringify(cell) === JSON.stringify(startCell)) {
                    cellElement.style.backgroundColor = "#64a866"; // ✅ 保持起點綠色
                } else if (JSON.stringify(cell) === JSON.stringify(endCell)) {
                    cellElement.style.backgroundColor = "#f5736a"; // ✅ 保持終點紅色
                    cellElement.textContent = ""; // ✅ 確保終點不顯示箭頭
                }

                cellElement.style.color = "black"; // ✅ 設定箭頭顏色為黑色

                // ✅ 只有當 `nextCell` 存在時，才計算箭頭方向
                if (nextCell) {
                    let arrow = getArrowDirection(cell, nextCell);
                    cellElement.textContent = arrow; // 🏹 顯示箭頭
                }
            }
        }, i * 150);
    }
}

function getArrowDirection(current, next) {
    let [currRow, currCol] = current;
    let [nextRow, nextCol] = next;

    if (nextRow < currRow) return "↑";  // 往上
    if (nextRow > currRow) return "↓";  // 往下
    if (nextCol < currCol) return "←";  // 往左
    if (nextCol > currCol) return "→";  // 往右
    return "";
}

function showMaze() {
    let maze = document.getElementById("maze-grid");
    maze.innerHTML = "";
    maze.style.gridTemplateColumns = `repeat(${gridSize}, 40px)`;
    maze.style.display = "grid";

    for (let row = 0; row < gridSize; row++) {
        for (let col = 0; col < gridSize; col++) {
```

```
            let cell = document.createElement("div");
            cell.classList.add("grid-item");
            cell.dataset.row = row;
            cell.dataset.col = col;

            let cellPos = JSON.stringify([row, col]);
            if (startCell && startCell[0] === row && startCell[1] === col) {
                cell.classList.add("start");
            } else if (endCell && endCell[0] === row && endCell[1] === col) {
                cell.classList.add("end");
            } else if (obstacles.has(cellPos)) {
                cell.classList.add("obstacle");
            }

            maze.appendChild(cell);
        }
    }
}

function closeMaze() {
    document.getElementById("maze-container").style.display = "none";
}

function animatePaths(paths, callback) {
    let iteration = 0;

    function animatePath() {
        if (iteration >= paths.length) {
            if (callback) callback(); // ✅ 動畫結束後執行 callback (顯示最優路徑)
            return;
        }

        let path = paths[iteration];
        let previousCell = null;

        path.forEach((cell, index) => {
            setTimeout(() => {
                let cellElement = document.querySelector(`#maze-grid [data-row="${cell[0]}"][data-col="${cell[1]}"]`

                // ✅ 恢復前一步的顏色 & 清除箭頭
                if (previousCell) {
                    let prevElement = document.querySelector(`#maze-grid [data-row="${previousCell[0]}"][data-col='
                    if (prevElement) {
                        if (JSON.stringify(previousCell) === JSON.stringify(startCell)) {
                            prevElement.classList.add("start");
                            prevElement.style.backgroundColor = "#64a866"; // ✅ 起點綠色不變
                            prevElement.textContent = ""; // 清除箭頭
                        } else if (JSON.stringify(previousCell) === JSON.stringify(endCell)) {
                            prevElement.classList.add("end");
                            prevElement.style.backgroundColor = "#f5736a"; // ✅ 終點紅色不變
                            prevElement.textContent = ""; // 確保終點不顯示箭頭
                        } else if (obstacles.has(JSON.stringify(previousCell))) {
                            prevElement.classList.add("obstacle");
                            prevElement.textContent = ""; // 清除箭頭
                        } else {
```

```
                    prevElement.style.backgroundColor = "#FFFFFF"; // ✅ 普通路徑恢復白色
                    prevElement.textContent = ""; // 清除箭頭
                }
            }
        }

        // ✅ 更新當前步驟的顏色 & 顯示箭頭
        if (cellElement) {
            // ✅ 普通路徑顏色變為 lightgreen
            if (JSON.stringify(cell) === JSON.stringify(startCell)) {
                cellElement.style.backgroundColor = "#64a866"; // ✅ 起點綠色
            } else if (JSON.stringify(cell) === JSON.stringify(endCell)) {
                cellElement.style.backgroundColor = "#f5736a"; // ✅ 終點紅色不變
                cellElement.textContent = ""; // ✅ 確保終點不顯示箭頭
            } else {
                cellElement.style.backgroundColor = "#90EE90"; // ✅ 普通路徑 lightgreen
            }

            cellElement.style.color = "black"; // ✅ 設定箭頭為黑色

            if (previousCell && JSON.stringify(cell) !== JSON.stringify(endCell)) {
                let arrow = getArrowDirection(previousCell, cell);
                cellElement.textContent = arrow; // 🏹 顯示箭頭
            }
        }

        previousCell = cell; // ✅ 更新前一步的 cell
    }, index * 100);
  });

  iteration++;
  setTimeout(animatePath, path.length * 100);
 }

 animatePath();
}

function renderMatrix(id, matrix, isPolicy) {
 let container = document.getElementById(id);
 container.innerHTML = "";
 container.style.gridTemplateColumns = `repeat(${gridSize}, 40px)`;

 for (let row = 0; row < gridSize; row++) {
  for (let col = 0; col < gridSize; col++) {
   let cell = document.createElement("div");
   cell.classList.add("grid-item");

   // ✅ 保留格子顏色 (start, end, obstacle)
   let cellPos = JSON.stringify([row, col]);
   if (startCell && startCell[0] === row && startCell[1] === col) {
    cell.classList.add("start");
   } else if (endCell && endCell[0] === row && endCell[1] === col) {
    cell.classList.add("end");
   } else if (obstacles.has(cellPos)) {
    cell.classList.add("obstacle");
```

```
            }

            // ✅ 設定數值或箭頭
            if (isPolicy) {
                let actions = matrix[row][col];
                cell.innerHTML = actions.replace(/↑/g, "↑ ")
                                        .replace(/↓/g, "↓ ")
                                        .replace(/←/g, "← ")
                                        .replace(/→/g, "→ ");
            } else {
                cell.textContent = matrix[row][col].toFixed(2);
            }

            container.appendChild(cell);
        }
    }
}

window.onload = renderGrid;
</script>

</body>
</html>
```

4. 結果展示

當使用者點擊 "Start Game" 按鈕後，系統會執行價值迭代 (Value Iteration) 並根據計算結果動態更新 最佳政策 (Optimal Policy)、價值函數 (Value Function)，並模擬最佳路徑的移動過程。

(1) 最佳政策顯示 (Policy Matrix)

- 每個格子標示最優行動 (↑、↓、←、→)：
  - 根據策略矩陣 (Policy Matrix)，顯示該格子處於最佳決策時應該選擇的行動。
  - 箭頭方向代表從當前格子移動的最佳選擇，例如：
    - "→"：建議往右移動
    - "↓"：建議往下移動
    - "←"：建議往左移動
    - "↑"：建議往上移動
  - 起點 (Start) 與終點 (End) 仍保留原本的顏色 (綠色、紅色)。

(2) 價值函數顯示 (Value Matrix)

- 每個格子顯示對應的 V(s) 值：
  - 這些值代表該狀態的最佳期望回報 (Expected Return)，由價值迭代計算得出。
  - 終點 (End Cell) 的數值最大，表示其具有最高回報 (通常為 1.0)。
  - 障礙物 (Obstacles) 的數值為 -1.0，表示不能移動的區域。
  - 其他格子的值代表該狀態在最佳決策下的潛在回報，數值越大代表越接近目標。

(3) 動態更新與最佳路徑模擬

- 模擬最佳移動路徑：
  - 透過動畫展示智能體的移動過程，從 起點 (綠色) 出發，沿著 最佳路徑 移動至 終點 (紅色)。

- 每一步移動延遲 0.3 秒，讓使用者能夠清楚觀察決策過程。

- 動畫結束後，標記最優路徑 (黃色🏆)：
  - 當動畫執行完畢，最終最佳路徑上的所有格子會變成 金黃色，用來標示智能體的最佳移動路徑。
  - 確保起點 (綠色)、終點 (紅色) 和障礙物 (灰色) 的顏色不變，僅對最佳路徑標記。