Part2 Frozen lake 8x8

Q-learning演算法優化

地圖特性

模式：slippery

地圖大小：8 x 8（固定size）

移動方式：上下左右一步

難題

一、slippery會讓移動具有隨機性

二、獎勵少：終點為1，其餘皆為0

核心演算法 Q-learning

核心機制：Q table

創建一個大小為64 x 4的矩陣Q(s,a)
記錄在狀態s採取動作a能獲得的預期累積獎勵
公式為：

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max Q(s',a') - Q(s,a)]$$

其中：α為學習率、R為當下獲得的獎勵、γ為折扣因子（對未來回報的重視程度）

實作細節

平手策略

```python
if is_training and rng.random() < epsilon:
    action = env.action_space.sample()
else:
    # 遇到 Q 值都一樣的情況（例如初期），隨機選擇以增加隨機性
    if np.all(q[state, :] == q[state, 0]):
        action = env.action_space.sample()
    else:
        action = np.argmax(q[state,:])
```

為了避免 Q-table 初始化為全 0 時，argmax 函數總是優先回傳索引 0（向左）導致的探索偏差，我們加入了判斷機制：

1.全相等情況（如 [0,0,0,0]）：強制隨機選擇動作，促進初期探索。

2.部分相等情況（如 [0, 0.5, 0.5, 0]）：維持 argmax 特性，優先選擇索引順序靠前的最大值（如選擇 Action 1 下），確保決策的確定性。

參數設定

調整參數

Learnign rate : 0.1
環境很滑，設太高 (如 0.9) 會導致 Q 值震盪；
設低一點能取長期平均，穩定收斂。

Discount factor : 0.99
路徑長且只有終點有獎勵，必須極度重視Future Reward

Epsilon decay : 動態衰減
設定衰減週期為總回合數的 60%，確保有充分的時間進行隨機探索，避免過早收斂於局部解。

實驗結果

測試條件：
Training 15000 episodes
testing 1000 episodes
結果：60%-62%

難題

依舊無法突破70%

解法：保持地圖尺寸，產生隨機地圖並把坑洞數量減少

```
# 訓練時：生成一張 8x8 的隨機地圖（p=0.8 代表 80% 是冰面，20% 是洞）
map_desc = generate_random_map(size=8, p=0.9)
```

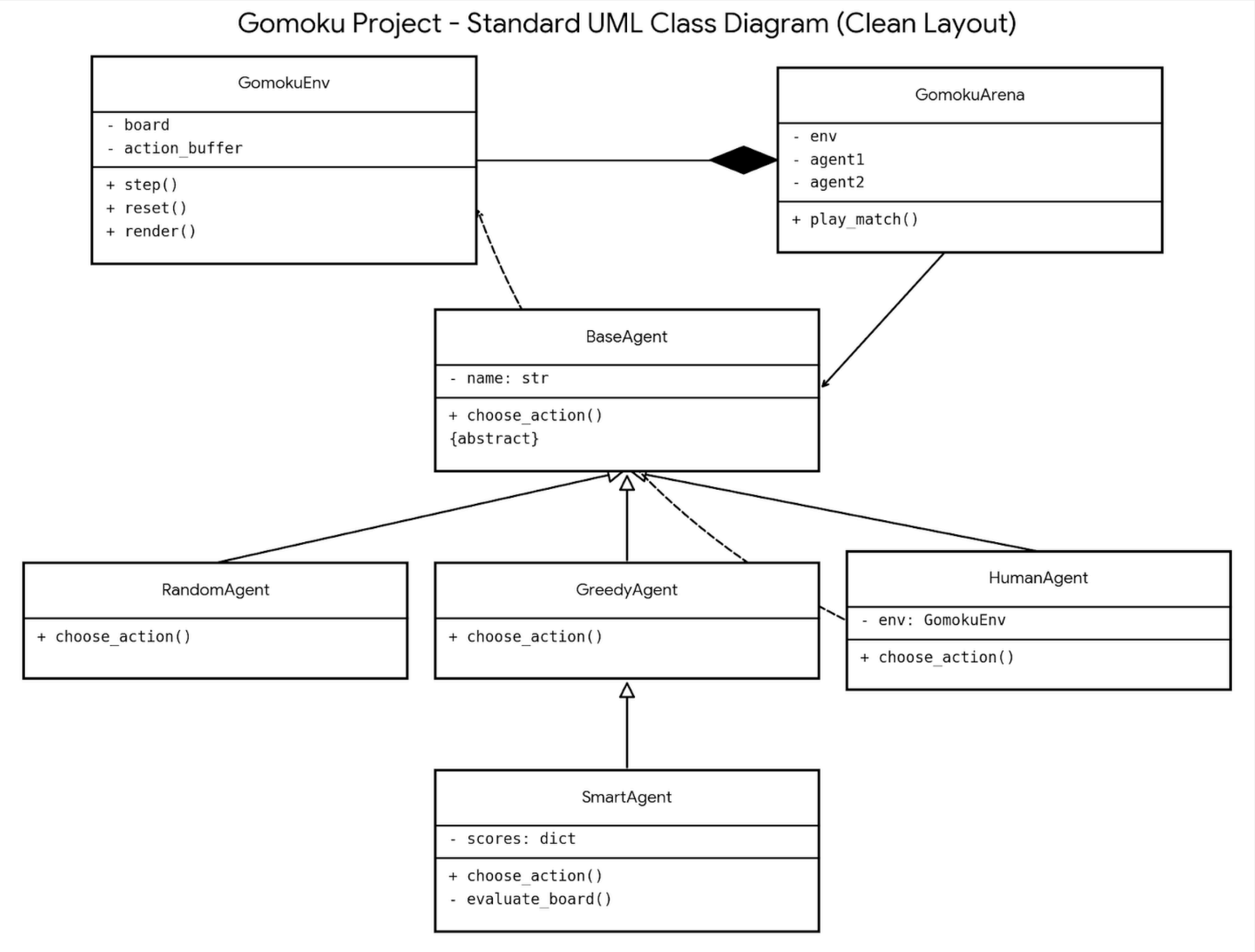
實驗結果(有更改地圖)

測試條件：
Training 15000 episodes
testing 1000 episodes
結果：0%-99%

因為地圖是隨機生成 因此他有可能產生走不到終點的地圖

Part3 五子棋 AI 對戰系統

架構


Gomoku Project - Standard UML Class Diagram (Clean Layout)
GomokuEnv
- board
- action_buffer
+ step()
+ reset()
+ render()
GomokuArena
- env
- agent1
- agent2
+ play_match()
BaseAgent
- name: str
+ choose_action()
{abstract}
RandomAgent
+ choose_action()
GreedyAgent
+ choose_action()
HumanAgent
- env: GomokuEnv
+ choose_action()
SmartAgent
- scores: dict
+ choose_action()
- evaluate_board()

架構

三大模組：

- **Model(GomokuEnv)：**
  負責遊戲規則、棋盤狀態、畫面渲染 (封裝)

- **Controller (GomokuArena)：**
  負責管理比賽流程、輪替玩家 (組合)

- **Agent(BaseAgent)：**
  定義 AI 的思考邏輯 (抽象/繼承)

OOP實踐 - 抽象

- 定義 BaseAgent 為抽象類別 (ABC)

```python
class BaseAgent(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def choose_action(self, board, valid_moves):
        pass
```

- 強制所有子類別實作 choose_action() 方法

  舉例其中一個子類別:

```python
class RandomAgent(BaseAgent):
    def choose_action(self, board, valid_moves):
        # 修正：檢查 NumPy 陣列是否為空
        if valid_moves.size == 0:
            return None
        return random.choice(valid_moves.tolist()) # 確保這裡使用列表
```

- 目的：制定統一的溝通介面

OOP實踐 - 繼承

- RandomAgent、GreedyAgent 繼承自 BaseAgent

```
class RandomAgent(BaseAgent):
```

```
class GreedyAgent(BaseAgent):
```

- 重點：SmartAgent 繼承自 GreedyAgent
- 目的：SmartAgent 直接沿用父類別的winning_move & blocking_move可減少 30% 以上的重複程式碼

```
# 1. 進攻檢查 (一步致勝)
winning_move = self._find_winning_move(board, valid_moves_list, my_id)
if winning_move is not None:
    return winning_move

# 2. 防守檢查 (擋住對手致勝)
opponent_id = 3 - my_id
blocking_move = self._find_winning_move(board, valid_moves_list, opponent_id)
if blocking_move is not None:
    return blocking_move
```

OOP實踐 - 多型

- Arena 中呼叫 agent.choose_action() 時，不需要知道對手是誰

```
action = current_agent.choose_action(self.env.board, valid_moves)
```

- 隨機 AI -> 亂下；聰明 AI -> 算分後下

- 目的：提升系統彈性，程式碼解耦

OOP實踐 - 封裝

- 複雜邏輯隱藏在class內部
- 例如：_evaluate_board()、_check_win_simulation()

```python
def _evaluate_board(self, board, player_id):
    total_score = 0

    for r in range(self.board_size):
        for c in range(self.board_size):
            if board[r, c] == player_id:
                total_score += self._get_position_score(board, r, c, player_id)

    return total_score
```

```python
def _check_win_simulation(self, board, row, col, player):
    """ 檢查在 (row, col) 落子後是否達成連線。 """
    directions = [(0, 1), (1, 0), (1, 1), (1, -1)]

    for dr, dc in directions:
        count = 1
        for i in range(1, self.win_streak):
            r, c = row + dr * i, col + dc * i
            if 0 <= r < self.board_size and 0 <= c < self.board_size and board[r, c] == player:
                count += 1
            else:
                break
        for i in range(1, self.win_streak):
            r, c = row - dr * i, col - dc * i
            if 0 <= r < self.board_size and 0 <= c < self.board_size and board[r, c] == player:
                count += 1
            else:
                break

        if count >= self.win_streak:
            return True
    return False
```

- 目的：外部只需取得結果，不需干涉運

AI 演算法 - Greedy

特色：規則式 (Rule-based)，反應快速但短視

1. 進攻：這一步能贏嗎？ -> 下這裡 (Win)
2. 防守：對手下一步會贏嗎？ -> 擋這裡 (Block)
3. 搶地：天元 (中心點) 是空的嗎？ -> 佔領中心
4. 隨機：以上皆非 -> 隨機下

AI 演算法 - Smart

特色：啟發式搜尋 (Heuristic Evaluation)，模擬人類判斷

棋型評分表 (scores)：
連五：10,000,000 分 (必勝)
活三 (兩頭空)：10,000 分
眠三 (一頭擋)：1,000 分

攻守權衡公式：
Score = (MyScore) - (OpponentScore * 0.9)

同時考慮「自己得分」與「阻礙對手」。
係數 0.9 表示稍微偏重進攻，但絕不忽視防守

THANK YOU!