Group 25
莊尹安、王芊婷、陳冠穎

Part 2: Frozen Lake

1.目標與方法

此部份目標為透過調整參數在 FrozenLake 8x8 中實現70% 的平均成功率。最後我們選擇 gene algorithm 作為最佳化工具，希望透過此方法的全局搜索能力來找到較好的結果。其中利用多次獨立運行平均來計算成功率，以確保結果不會被單一次的結果導致偏差。

2.實施與挑戰

a.初始階段的問題

一開始是手動調整參數，但是發現成功率一直停留在60%左右，期間也嘗試了一些方法，最終只留下了動態調整learning rate的結構。之後想到利用gene algorithm來調整參數，以求此結構下的最佳化參數。

b.gene algorithm最佳化結果與瓶頸

最後透過gene algorithm找到的最佳參數組合和對應的最高平均成功率（60.56%）。由此得知在這個方式底下60%成功率的確是瓶頸,其中FrozenLake的高隨機性 (2/3機率滑向錯誤方向)應該是此架構下性能上限的主因。

c.其他策略嘗試與無效分析

(1)Sarsa algorithm

結果：成功率降至 10%~20%。大概是因為Sarsa 追求安全策略，但在 8×8 長路徑中幾乎走不到成功路徑。

(2)reward shaping

結果：成功率無顯著提升，甚至下降。推測是因為微小的中間獎勵被高隨機性帶來的巨大最終懲罰掉洞所抵消，使結果無法穩定收斂，。

3.學習與洞察

a.強化學習算法選擇學習：得知Q-learning和Sarsa在高隨機性的環境下，Sarsa會更保守的選擇。

b.超參數關係：理解其中參數的意義。像是gamma必須非常接近 1.0 才能讓代理人關注長遠目標，是手動調整時提升成功率最多的參數。

c.gene algorithm應用：了解如何利用gene algorithm來得到最佳化結果，並利用結果來精進下一次的學習（example：縮小搜索範圍）。

d.分析結果

得知真正的瓶頸是算法與環境的匹配度。確認在這個問題上，算法的選擇會決定成功率的上限。

4.結論

學習到可以利用gene algorithm能夠了解到選擇的演算法是否為環境下的優質選項。利用此專案的經驗，在未來專案中優化使用gene algorithm的效率。也了解到在不同的環境之下，學習方法的上限會因而變化。

Part 3: Tic-Tac-Toe

1. 使用的物件導向程式設計概念

   a. Abstraction（抽象化）

      我們透過抽象類別 Player 將「所有玩家必須能夠決定下一步」這件事抽象化，而不關心玩家究竟是人類還是 AI。這降低了高層程式碼的複雜度，使 GameManager 可以專注於流程控制。

   b. Inheritance（繼承）

      HumanPlayer 與 AIPlayer 都繼承自 Player，並根據需要覆寫 select_action()。AI 也透過 AIStrategy 形成一套策略的繼承架構，使得不同的 AI 行為能獨立開發與切換。

   c. Polymorphism（多型）

      不論是使用 RandomStrategy 或 MinimaxStrategy，AIPlayer 都透過同一個方法呼叫策略，使得：

      ```
      action = self.strategy.choose_action(env)
      ```

      可以自動依照策略類型執行不同邏輯。這讓“切換難度”變成完全不需要修改 AIPlayer 本身的行為，展示了多型的強大。

   d. Encapsulation（封裝）

      我們將棋盤規則封裝到 Environment 中，將玩家行為封裝到 Player 中，並將人機互動邏輯封裝在 GUI 內，使得後端邏輯與介面分離，維護時不易互相干擾。

   e. Strategy Pattern（策略模式）

      AIPlayer 不負責決定 AI 怎麼下棋，而是委託給某一個策略物件：

      - RandomStrategy（初級）
      - MediumStrategy（中級）
      - MinimaxStrategy（高級）

      這讓難度切換、或是之後要擴充都比較容易。

2. Minimax 演算法

Minimax 是一種在二人零和遊戲中常用的 AI 演算法。其核心概念是AI 模擬每一種可能的未來棋局，假設對手會走最強的反擊，並從中選出對自己最有利的一步，其流程如下：

a. 列出 AI 可以下的所有位置

b. 對每個位置模擬對手會怎麼走

c. 對手再模擬 AI 的回應

d. 一直往後模擬直到結束

e. 回傳每個結果的分數（贏 +1、輸 -1、平手 0）

f. AI 選擇「能最大化自己分數」的一步

在 3×3 圈圈叉叉中，棋盤小、可能局面有限，Minimax 可以做到搜尋完整棋局，因此使用 Minimax 的 AI 永遠不會輸（最差平局）。

3. 遇到的困難

a. 如何將遊戲邏輯拆成合適的類別

初期我們難以判斷哪些功能應該被拆出來，如環境是否需要知道玩家？Player 是否要知道棋盤結構？GUI 是否要負責遊戲規則？後來透過不斷重構，我們採用了「環境、玩家、流程管理、介面」四層分離，使架構更清楚，也更加符合理想的 OOP 設計。

b. AI 設計過程的挑戰

最初我們直接把 random 與 minimax 的邏輯寫在 Player 內，但當我們想加入難度選擇時，發現 Player 類別未來會變得越來越臃腫。採用 Strategy Pattern 之後，AI 的邏輯得以獨立發展，大幅改善了程式的可擴充性。