AI hw2 report

1. Introduction

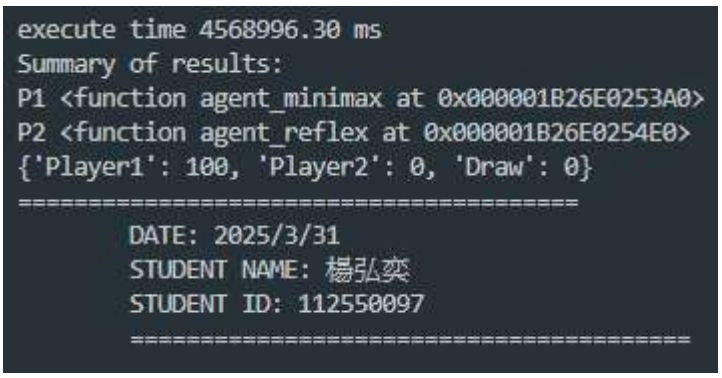
本次作業透過在Connect Four中實作Minimax演算法及Alpha-Beta剪枝，深入了解人工智慧的Adversarial Search。最後設計了一個更強的agent_strong，透過改良heuristic function，有效判斷與阻擋對手的進攻，並掌握棋盤的關鍵位置，目標是表現可以穩定優於depth=4的Alpha-Beta。

2. Implementation

(1) Minimax Search

1. **How it recursively explores the game tree?**
    a. 從當前的棋盤狀態開始，探索每個可能的move與對應的後續棋盤狀態，形成一個game tree，每個節點都是從父節點延伸的狀態。
    b. 探索的深度最多是4層，當depth降為0即會回傳heuristic function。
2. **How it selects moves for the maximizing and minimizing players?**
    - maximizing player：從所有子節點中選擇能達到最大分數的節點。
    - minimizing player：從所有子節點中選擇能達到最小分數的節點。
3. **The role of get_heuristic (board) in evaluation?**

    用於在terminal node評估狀態價值並回傳分數。由於所有決策都基於此heuristic function的回傳值，它是決定策略成效的關鍵環節。
4. **Results & Evaluation**



(2) Alpha-Beta Pruning

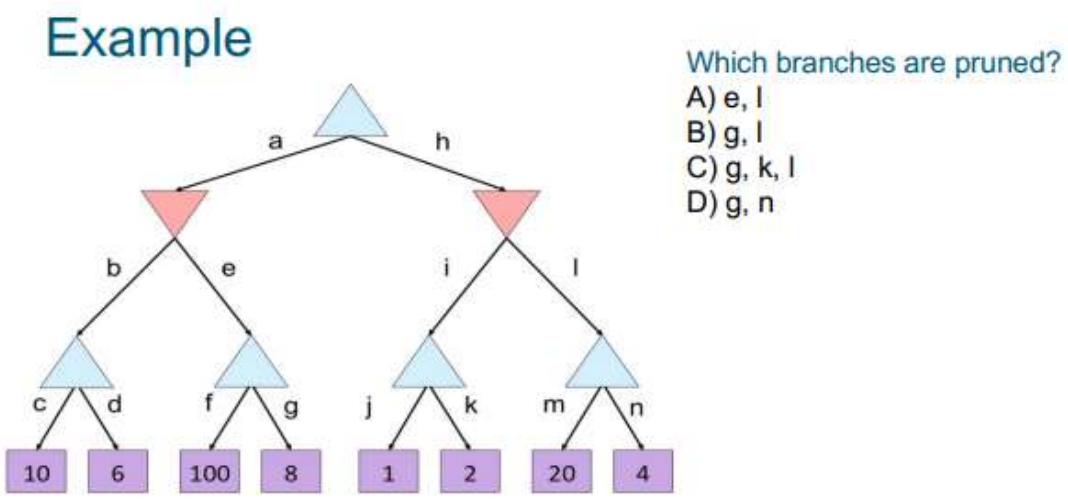
1. **How α-β pruning reduces unnecessary node expansions?**

   透過兩個參數：

   - **α**：best option so far from any MAX node to the root
   - **β**： best option so far from any MIN node to the root

   在搜尋過程中，當發現某個節點的後續結果必定不會優於先前評估過的選擇（即 `alpha` ≥ `beta`）時，就可以停止搜尋該節點以下的所有分支，快速減少搜尋空間，避免不必要的計算。

   舉例：



   砍掉g

   1. c、d選擇最大值10
   2. 當我們看到f是100時，就不用往後看了，因為(100≥10)已經對上一層的min player沒有貢獻

   砍掉I

   1. 左子樹處理完後，b、e選擇最小值10
   2. j、k選擇最大值2
   3. 回傳到min player節點，因此後面的不用看了，因為(2≤10)已經對上一層的max player沒有貢獻

2. **When and how pruning occurs in your implementation?**

- **當輪到 maximizingPlayer（MAX節點）時：**

```
alpha = max(alpha, value)
# 後面的不用看了
if alpha >= beta:
    break
```

  - `alpha` 隨著節點探索到更大的value而提高。
  - 當這個新獲得的 `alpha` 值大於或等於目前節點的 `beta` 時，表示後續的探索對上層的min player來說已經沒有貢獻，因此觸發pruning。
- **當輪到 minimizingPlayer（MIN節點）時：**

```
beta = min(beta, value)
if alpha >= beta:
    break
```

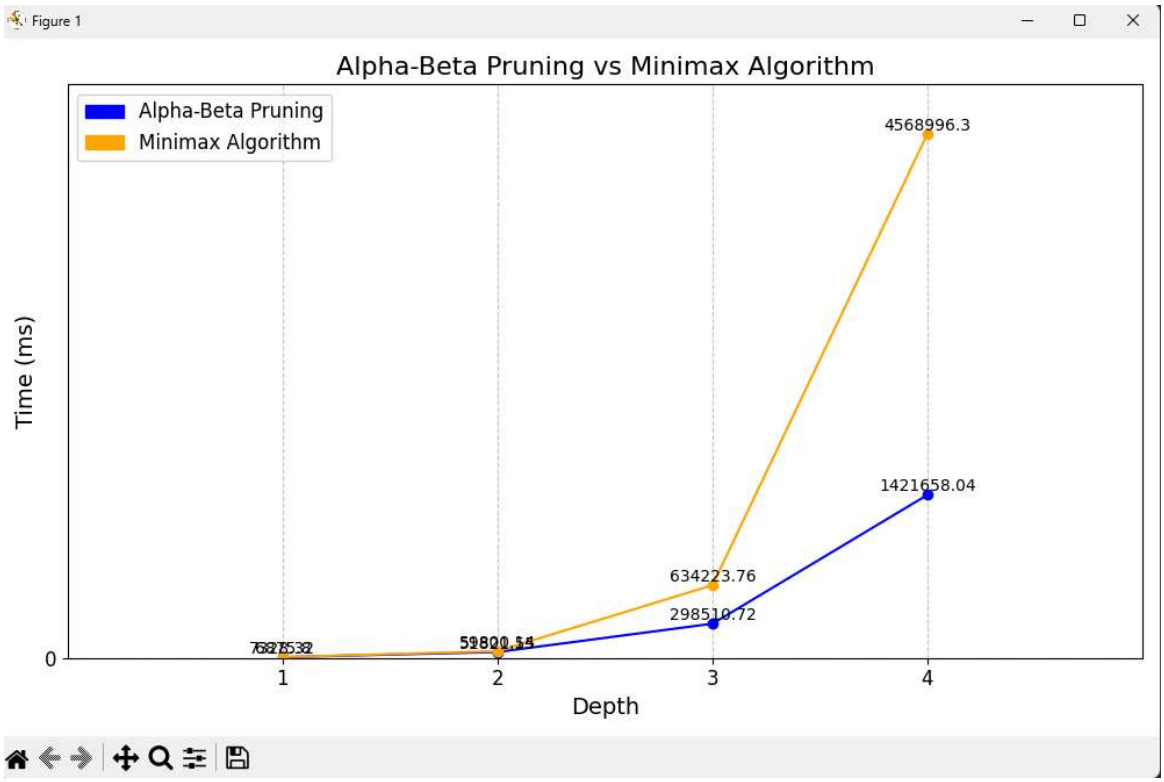
  - `beta` 隨著節點探索到更小的value而提高
  - 當新獲得的 `beta` 值小於或等於目前節點的 `alpha` 時，表示後續的探索對上層的max player來說已經沒有貢獻，因此觸發pruning。

3. **Execution time of Minimax vs. Alpha-Beta**

   a. winning rate against reflex agent

```
execute time 1421658.04 ms
Summary of results:
P1 <function agent_alphabeta at 0x000001F4AED554E0>
P2 <function agent_reflex at 0x000001F4AED55580>
{'Player1': 97, 'Player2': 3, 'Draw': 0}
========================================
        DATE: 2025/3/31
        STUDENT NAME: 楊弘奕
        STUDENT ID: 112550097
        ========================================
```

   b. Execution Time vs Minimax



(3) Strong AI Agent

1. **Techniques Used**

   - **更精確的Heuristic Function設計**

     使用`get_heuristic_strong()`作為評估棋盤狀態的函式，更好的啟發函式可以讓AI做出更有利的決策。

   - **Alpha-Beta剪枝條件的細微調整**
     我發現原本的Alpha-Beta演算法在剪枝條件設定為`alpha >= beta`時，可能會忽略一些潛在的更優行動。在實驗與觀察之後，我將此條件調整為更嚴格的`alpha > beta`，目的在於：

     - 當`alpha`與`beta`數值相同時，若過早進行剪枝，可能會錯失更佳或至少是同等價值的策略性走法。
     - 此調整有助於AI在決策過程中能更全面地評估節點，使AI在面對邊界條件時仍能維持優異的判斷力。

2. **Advanced Heuristic Function**

   heuristic_strong的具體策略包含以下四點：

   1. **Immediate Win/Lose (即時勝負判斷)**

```python
# if win or lose
if board.win(1):
    return 1e12
if board.win(2):
    return -1e12
```

若當前棋盤狀態已有勝負產生，則直接回傳極端分數：

- 若已獲勝，則回傳極大值（10^12）。
- 若已輸掉，則回傳極小值（−10^12）。

2. **Forced Win/Block Lose (強制勝利或防止失敗)**

```python
# forced win/ block lose
winning_moves_p1 = [col for col in board.valid if game.check_winning_move(board, col, 1)]
winning_moves_p2 = [col for col in board.valid if game.check_winning_move(board, col, 2)]
if winning_moves_p1:
    return 1e10 + 1000 * len(winning_moves_p1)
if winning_moves_p2:
    return -1e10 - 1000 * len(winning_moves_p2)
```

偵測下一步棋是否能夠立即產生勝負：

- 若下一步棋可以立即獲勝，則給予非常高的分數（101010^{10}），並依據可獲勝走法數量給予額外加分。
- 若對手下一步棋能夠立即獲勝，則給予非常低的分數（−101010^{10}），並依據對手可獲勝的走法數量額外扣分。

3. **Potential Wins (潛在的勝利機會)**

```python
# potential win/loss
num_threes = game.count_windows(board, 3, 1)
num_threes_opp = game.count_windows(board, 3, 2)
num_twos   = game.count_windows(board, 2, 1)
num_twos_opp   = game.count_windows(board, 2, 2)
win_potential = (num_threes - num_threes_opp) * 1e6 + (num_twos - num_twos_opp) * 100
```

針對潛在的連線機會進行評估：

- 若有潛在的三子連線，給予較高分數（每個三子連線機會 10610^6 分）。
- 若有潛在的兩子連線，給予適當的加分（每個兩子連線機會 100 分）。
- 同時也考慮對手的潛在機會，以相同分數的負值扣分。

4. **Center Control (中心控制權)**

```
# center control
center_column = board.column // 2
 (variable) center_count_opp: Literal[0]
center_count_opp = 0
for r in range(board.row):
    if board.table[r][center_column] == 1:
        center_count += 1
    elif board.table[r][center_column] == 2:
        center_count_opp += 1
center_score = 5000 * (center_count - center_count_opp)
```

由於棋盤的中心列能夠提供更多的未來贏棋機會，因此對於中心列的掌控給予額外分數：

- 計算中心列我方與對方棋子的數量差，將其乘以5000，作為中心控制分數的加權。

5. **總分**

```
# final score
score = win_potential + center_score
return score
```

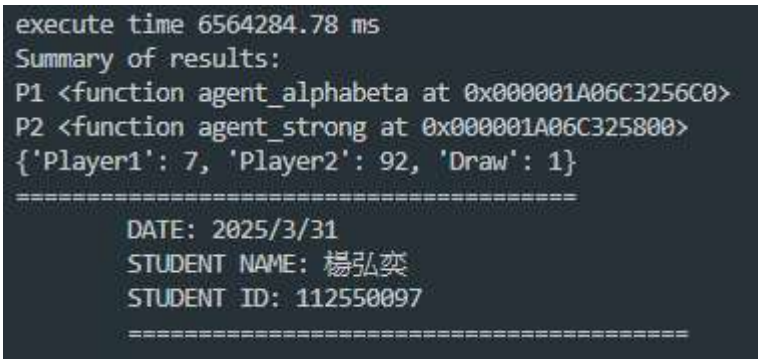
總分是Potential Win的分數加上Center Control的分數

6. **Conlusion**

   a. Immediate Win/Lose 以及 Forced Win/Block Lose都是透過極端的數值差距來引導agent_strong做出優先決策，讓它不會錯失獲勝的機會，也可以阻擋對手的勝利。

   b. 而Potential wins以及Center Control則是透過相對於get_heurisitc()更詳細以及更有策略的評估局面，回傳更符合當前狀態獲勝價值的分數，同樣可以更好的引導agent_strong。

3. **Results & Evaluation**



3. Analysis & Discussion

(1) Difficulties in designing a strong heuristic:

一開始的困難在於，即使仔細閱讀了spec，還是沒什麼想法要怎麼將策略具體轉化成程式碼。但是在`game.py`中找到一些函式以及屬性幫助我可以更好的存取並分析當前的局面。

另外有一個問題是在一開始完全清空並重寫了`agent_strong()`函式，且錯誤地將呼叫`your_function()`中的`maximizingPlayer`參數設為`True`（與先前Minimax、Alpha-Beta的做法相同），導致`agent_strong()`在測試時一直被打爛。後來反向調整了`get_heuristic_strong()`內部對於玩家編號的判斷，也就是讓max player以編號2作為判斷根據，在測試中才獲得較高的勝率。

之後跟同學討論，才了解到作業中的`agent_strong()`設定後手（Player 2）為實際上的max player，因此必須在呼叫`your_function()`時將`maximizingPlayer`參數設為`False`，以讓後手取得最大化的value。所以`get_heuristic_strong()`中對於玩家編號的判斷就可以保持不變。

(2) Weaknesses of agent_strong():

在設計過程中遇到的主要弱點之一是計算時間相當長，由於更複雜的評估函式以及增加了額外的條件判斷，導致運行效率降低。

(3) Further enhancements for agent_strong():

- **Depth-Aware Evaluation**

  在回傳value時，可能可以根據當前的depth作權重的調整。

- **優先處理有較多win-potential的行動：**

  在決策時，除了考量單一走法的立即勝利外，還可以評估每個可能走法產生的多種後續獲勝機會。透過事先計算與評估這些潛在的「多重獲勝走法」，agent能更有效地布局策略。

- **動態深度調整**

  未來可考量在關鍵局勢時（如發現對手可能連成三子）動態增加搜尋深度，更有效地處理緊急防守或快速致勝的情況。

4. Conclusion

這次作業透過實際實作Minimax和AIpha-Beta pruning演算法，幫助我更清楚地了解遊戲搜尋樹的結構，以及如何透過剪枝來提升搜尋效率。此外，為了設計一個更強大的AI，優化了heuristic function ，加入了即時判斷勝負局勢、主動攻擊與防守、以及控制棋盤中心等策略，進一步提升決策品質與勝率。