HW2

112550109 楊榮竣

1 Introduction

這次作業主要以 Connect Four 作為主要測試的遊戲，讓不同的 agent 進行遊戲，統計勝負的次數，並以此作為依據，評斷 agent 的表現。首先利用 Minimax Algorithm 來設計 agent 打敗 Reflex Agent。接著利用 Alpha-Beta Purning 來縮短搜尋時間。之後在通過加強 heuristic function 或是改動結構來強化 agent。最終讓 Strong Agent 能夠在和 AlphaBeta Agent 的對局中保有 50% 以上的勝率。

2 Implementation

Minimax Search

利用 recursive function 來將樹的節點展開，並在一開始先判斷目前節點的深度是否到達條件以及遊戲是否結束，若是符合條件則計算 heuristic 並回傳計算結果。先將目前遊戲可以進行的移動儲存在 valid_move 中，並透過 drop_piece() 來模擬下一步。再將模擬的結果輸入進 minmax()，並且將 maximizingPlayer 設為相反數值，將執行完的結果存入 board_val，接著依照 maximizingPlayer 來判斷目前是要選擇最大值還是最小值，而當 board_val 與目前最大或最小值相同時，將目前正在計算的移動選擇加入 CandidateMove 中。等到所有可能的移動結果都計算完後，回傳 board 及 CandidateMove。

```
if depth == 0 or grid.terminate():
    return game.get_heuristic(grid) , set()
```

判斷深度及遊戲進行狀況

```
for col in valid_move:
    grid_next = game.drop_piece(grid,col)
    board_val , result = minimax(grid_next,depth-1,not maximizingPlayer)

    if maximizingPlayer:
        if board_val > board:
            CandidateMove = {col}
        elif board_val == board:
            CandidateMove.add(col)
        board = max(board,board_val)
    else:
        if board_val < board:
            CandidateMove = {col}
        elif board_val == board:
            CandidateMove.add(col)
        board = min(board,board_val)
```

將遊戲中所有可能的下一步進行搜尋，並依照 maxigPlayer 的數值選擇目前要抓取最大值還是最小值

執行結果:

```
========================================
Game 100/100 finished.
execute time 1695589.38 ms
Summary of results:
P1 <function agent_minimax at 0x0000028DBE1849A0>
P2 <function agent_reflex at 0x0000028DBE184AE0>
{'Player1': 100, 'Player2': 0, 'Draw': 0}
========================================
        DATE: 2025/03/25
        STUDENT NAME: 楊榮竣
        STUDENT ID: 112550109
        ========================================
```

Result of MiniMax agent

Alpha-Beta Pruning

Alpha 代表到目前為止，Agent 能夠產生的最大值。Beta 則代表到目前為止，Agent 產生的最小值。在執行過程中，如果尋找最小值時，發現目前的最小值比 Alpha 小，如果之後還回改傳值還會改動的話，也只回傳回比目前值在更小的值。當把結果往上一層傳遞時，因為要選擇對大值，但是在下一層檢查時，就已經確定回傳值不會比目前的最達值大，所以不會對上一層產生的結果有任何影響。同理，當在尋找最大值時，就要判斷在這一層的最大值是否會比目前找到的最小值還大，若是符合就停止搜尋，因為接下來的搜尋已經確定不會再影響結果。

```python
if maximizingPlayer:
    if board > beta:
        break
    alpha = max(alpha,board)
else:
    if board < alpha:
        break
    beta = min(beta,board)
```

依照 maximizingPlayer 判斷目前要尋找的是最大值還是最小值，並做出不同選擇

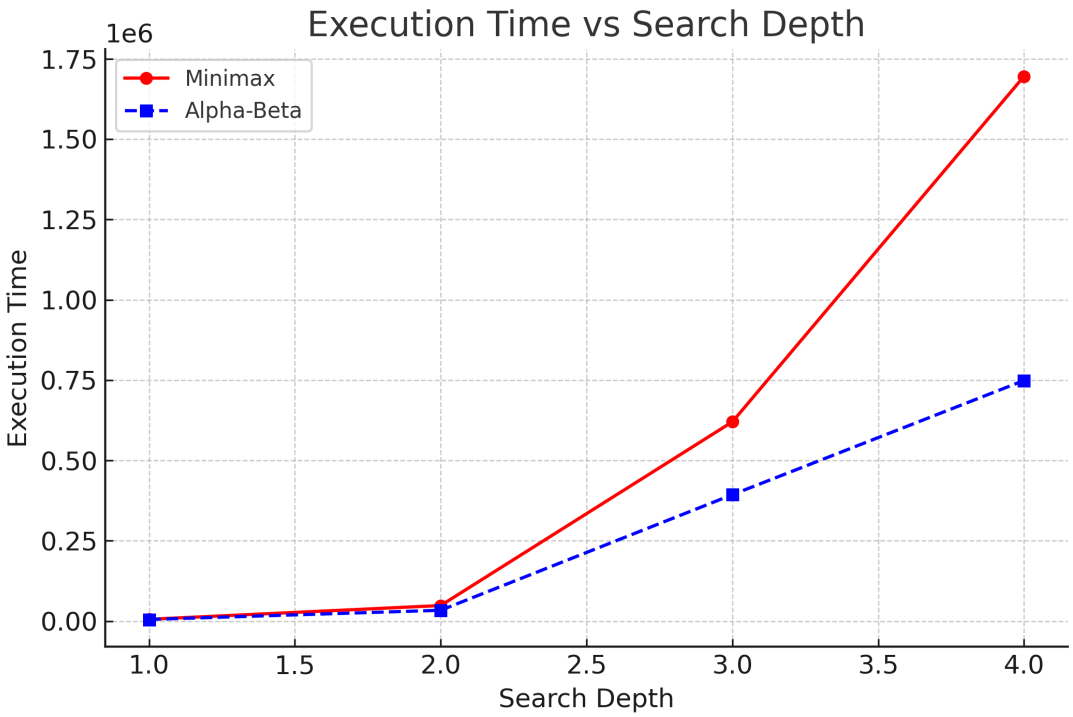
執行結果:

```
========================================
Game 100/100 finished.
execute time 748913.98 ms
Summary of results:
P1 <function agent_alphabeta at 0x000002128E944C20>
P2 <function agent_reflex at 0x000002128E944CC0>
{'Player1': 100, 'Player2': 0, 'Draw': 0}
========================================
        DATE: 2025/03/25
        STUDENT NAME: 楊榮竣
        STUDENT ID: 112550109
        ========================================
```

Result of AlphaBeta Agent

執行時間比較

Depth	1	2	3	4
Minimax	5734.41	48492.33	620261.42	1695589.38
Alpha-Beta	5104.48	33768.19	393416.11	748913.98



通過上圖可以觀察出，當搜尋深度較小時，Alpha-Beta Pruning 較無法產生明顯的差距，而當深度增加時，差距會逐漸顯著。

Strong AI Agent

以原來的 heuristic_function 為基礎，再增加條件判斷，加強 Agent 的能力，我制定了以下幾個策略來增加勝率：

- **依據遊戲進行的步數調整不同項目的重要性**
- **遊戲初期會先放在中心位置**
- **判斷可能連線或威脅性較強的狀況**
- **調整參數在攻擊與防守中取得平衡**
- **控制第一步下的位置**
- **優先搜尋中心可放置的位置**

依據遊戲進行的步數調整不同項目的重要性

從 board.cnt 判斷目前前遊戲進行的狀況，並將整個遊戲分成三個部分。並依照現在遊戲進行的狀況傳回段應的數值。因為在遊戲前期時，兩個連線的重要性會比三個連線的重要性高，因此在前期時，會分配比較高的數值。但在中後期，三個連線會比較重要。因此中後期時我將三個連線的比率調高。

```
def priority_adjustment(board):
    game_stage = board.cnt
    if game_stage < 20:
        return 1.5
    elif game_stage < 30:
        return 1
    else:
        return 0.5
```

```
num_twos       = game.count_windows(board, 2, 1) * priority_adjustment(board)
num_threes     = game.count_windows(board, 3, 1) * (2-priority_adjustment(board))
num_twos_opp   = game.count_windows(board, 2, 2) * priority_adjustment(board)
num_threes_opp = game.count_windows(board, 3, 2) * (2-priority_adjustment(board))
```

遊戲初期會先放在中心位置

因為只希望只在前期以中間部分為主，所以會先利用目前放置的個數判斷目前遊戲進行到哪個部分。之後再利用計算中間部分 (第三、四、五行) 兩個玩家放置的個數，計算完成後相減並乘上比率。

```
# Center First Motion
center_column = [2,3,4]
center_count = 0
if board.cnt < 8:
    for col in center_column:
        for row in range(board.row):
            if board.table[row][col] == 1:
                center_count += 1
            elif board.table[row][col] == 2:
                center_count -= 1
    score += center_count * center_p
```

center_p = 1000

判斷可能連線或威脅性較強的狀況

在原來的 heuristic_function 中，只要是三個連線都會判斷為一樣的狀況，但是在遊戲中會因為是否被阻擋，而有不同的重要程度，因此我將原來的計算方式進行修改，並在額外增加威脅性較高的判斷，並給予其不同的比重。

可能會連線狀況：

player , player , 0 , player

player , 0 , player , player

威脅性高的狀況：

0 , player , player , player , 0

```python
def count_windows_three(game_list,player):
    score = 0
    patterns = [
        [player,player,0,player],
        [player,0,player,player],
        [player,player,player,0],
        [0,player,player,player],
    ]
    for pattern in patterns:
        for i in range(1,len(game_list)-len(pattern)):
            if game_list[i:i+len(pattern)] == pattern:
                score += 1
    return score
```

```python
def count_strong_three(game_list,player):
    score = 0
    pattern = [0,player,player,player,0]
    for i in range(0,len(game_list)-len(pattern)+1):
        if game_list[i:i+len(pattern)] == pattern:
            score += 1
    return score
```

直接將特定情況儲存在函式中，檢查傳入陣列中是否有特定狀況

```python
def horizontal_test(board,player):
    score = 0
    for row in board.table:
        score += count_windows_three(row,player)
    return score

def horizontal_test_strong(board,player):
    count = 0
    for row in board.table:
        count += count_strong_three(row,player)
    return count
```

將上面的函式放入 heuristic_strong，將計算出的結果乘上比率，加入 score 中，影響輸出結果

調整參數在攻擊與防守中取得平衡

在 Heuistic_strong 中，透過在計算分數時，對兩個玩家分配不同的比重來達到，傾向進攻或是傾向防禦的效果。我將 Player2_p 作為比重，數值越大則代表會越以攻擊作為選擇的目標。為了找出最佳的參數，我修改了助教給的遊戲，並在每次遊戲結束時，依照勝負調整參數。總共進行五次測試，每一次測試都進行 100 次遊戲，在測試中 Player2_p 會逐步調整。最終依照測試結束的結果取其平均做為最後設置的參數。

控制第一步下的位置

在經過多次觀察後，我發現在我設計出的 Agent 中，第一步下的位置會大幅度的影響接下來獲勝的可能性。因此通過判斷敵方機器人第一步下的位置，並做出最佳應對，來提高勝率。

```
if grid.cnt == 2:
    return 0 , first_move_choose(grid)
```

在 your_function 中的部分

```
def first_move_choose(board):
    # Find first move
    first_move = 0
    for r in range(board.column):
        if board.table[5][r] == 1:
            first_move = r
            break
    # Decision
    if first_move == 4:
        return {3}
    elif first_move == 2:
        return {3}
    elif first_move == 3:
        return {2}
    else:
        return {2,3,4}
```

會先搜尋 Player1 第一步下的位置，再傳回對應的選擇

優先搜尋中心可放置的位置

在這個遊戲中，中心的位置會有比較高的機率出現最佳解法，因此在展開點時，有先從中心位置開始展開，接著再逐步往兩側檢查。利用這種方法搜尋，有機會減少搜尋時間，如果最佳走法出現在中心。

```
valid_moves = []
columns_sequence = [3,2,4,5,1,0,6]

for col in columns_sequence:
    if col in grid.valid:
        valid_moves.append(col)
```

在 your_function 中，依照順序逐個檢查是否時可以執行的下法，並放入 valid_moves 中

執行結果:

```
Game 100/100 finished.
execute time 1609569.08 ms
Summary of results:
P1 <function agent_alphabeta at 0x0000025B04B34860>
P2 <function agent_strong at 0x0000025B04B349A0>
{'Player1': 11, 'Player2': 63, 'Draw': 26}
========================================
        DATE: 2025/03/25
        STUDENT NAME: 楊榮竣
        STUDENT ID: 112550109
        ========================================
```

3 Analysis & Discussion

參數調整

在完成這次作業的過程中，我認為最大的困難在於調整參數。因此我透過更改助教提供的遊戲程式，並讓程式能夠記錄一些在遊戲的過程，作為分析與調整的參考依據。

調整攻守比率：

進行五次測試，每次測試都進行一百次遊戲，初始值設定為 10，隨著遊戲進行逐漸調整，並取五次的平均作為最後結果。

測試結束時最終值	1	2	3	4	5	Average
player2_p	8.2	7.2	8.1	8.0	8.5	8.0

計算第一步的最佳下法：

在執行許多測試後，我觀察到 AlphaBeta Agent 第一步只會出現三種可能性。我測試當 Strong Agent 在出現某中情況時，選擇哪一個位置會出現比較高的勝率。在測試階段，我會隨機讓 Strong Agent 選擇中間三行中的任一行。在執行時，我會記錄兩個參數 (value,total)。value 一開始設為零，在 Player1 獲勝時加一，而 Player2 獲勝時則減一，平手時則維持不變。toal 則代表總共發生的局數，兩者相除就是比率。

測試結果比率 Player2(x) vs Player1(y)

測試結果比率	2	3	4
2	-0.410	-0.483	-0.244
3	-0.641	-0.604	-0.580
4	-0.367	-0.558	-0.403

從這張表中，可以看出當 Strong Agent 在以下情況時會有較高的勝率：

- Player1 第一步放在 2，Player2 第一步放在 3
- Player1 第一步放在 3，Player2 第一步放在 2
- Player1 第一步放在 4，Player2 第一步放在 3

放棄的方案

在完成 Strong_Agent 的期間，我構思了許多的期間不同的方案，在這部分我會分析幾個曾經構思過，但因為效果不佳而放棄的想法。

1. 計算棋子密度：檢查某個位置的四周是否有出現過多相同玩家下的棋子。當初會想要嘗試，是因為發現出現必勝圖型時，周圍常常會出現六個以上的同一玩家的棋子。但在經過嘗試後，發現這方法並不能有效阻止必勝圖型的產生。我認為要有效處理必勝圖型產生，可能還是需要盡量去阻擋敵方的三個連線棋子。

2. 隨機模擬一局可能狀況：這是在看到 MCTS 演算法後，產生的想法。一開始的作法是在計算 heuristic 時，開始隨機下棋，直到遊戲結束，依照結果來影響 score。接著我調整下棋的模式，會傾向於下棋在中間附近，並只在遊戲中局後才開始進行，以加計算快速度並提升模擬的精準度。後來在反覆測試後，我認為這個方法並不可行，可能原因在於，這個方法的隨機性過高，無法利用這個方法去使 Agent 下出較佳的棋路。

缺點

在我設計的 Agent 中，最明顯的缺點就是會執行的時間較長。因為需要搜尋的狀況增加，雖然讓每一步的精確度增加，但也因此需要比較長的時間去搜尋。此外，在雖然能夠在讓勝率超越 50%，但還是在在某些情況下，會輸給 AlphaBeta Agent，我認為是缺少了收尾的判斷。觀察輸給 AlphaBeta 的那幾場，我發現到幾乎都是遊戲快結束時，只剩下一兩個位置可以選擇，或許可以設定一個收尾的演算法，負責處理當指甚下幾個位置放置時，應該要如何才能選擇最佳的位置。此外，也可以在前期時調高 Player2_p，或許能夠讓遊戲在前期或是中期內就結束。但這也可能造成，忽視敵人獲勝的情形，所以可能會需要精細的調整參數。

4 Conclusion

在本次作業先以實作 Minimax Agent 及 AlphaBeta Agent 為基礎，再設計 Strong Agent 在遊戲中打敗 AlphaBeta Agent。在這次實作 Strong Agent 我主要以加強 heuristic function 來完成。而利用中心優先、依照時程分配比重等策略強化 Agent，讓 Agent 能夠選擇出最佳的解法。其中在調整參數實利用反覆測試並記錄數據，在依據數據下去修改。雖然強化過後的 Agent 能夠有一定的勝率，但仍然存在些許問題，像是計算時間過長，或是遊戲收尾的情況未考慮。