AI Homework 2

112550184 陳璽安

1 Introduction

In this assignment, I have implemented adversarial search techniques for the game Connect Four. It focuses on developing intelligent agent that are capable of decision-making by using searching algorithm. The algorithm I implemented include Minimax algorithm, Alpha-Beta algorithm and strong agent with advanced heuristic function.

2 Implementation

2.1 Minimax Search

It follows a recursively approach to explore the game tree and determine the optimal move.

Parameters:

A. grid: the current board state

B. depth : how many move we want to look ahead

C. maximizingPlayer: If == True then the current player is maximizing player, if == False it's minimizing player.

D. dep : the maximum depth (=4)

Code explanation:

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

terminate condition: if we reach the maximum depth(depth==0) or the game is ended (grid.terminate()), evaluate the current board status by using get_heuristic(grid) and return the value and empty set.

```
move = grid.valid

    if not move:
      return 0, set()

    candidate=set()
    value=0
```

Get the valid movements, if we don't have any valid movement (the board is full), return 0 and empty set.

Initialize the candidate move with an empty set and the boardValue with 0.

```
if maximizingPlayer:
      value=float('-inf')
      for col in move:
        next=game.drop_piece(grid, col)
        next_value, _ = minimax(next, depth-1, False, dep)
        if next_value>value:
          value=next_value
          candidate={col}
        elif next_value==value:
          candidate.add(col)
```

If it's maximizing player's turn, initialize the board value with negative infinity. Simulate each valid movement and get the next board value (Recursively call Minimax function for the opponent's turn and drop a piece in that column).

If next board value is bigger than current value, update value with next board value, and reset the candidate move to the current move. (It means the move is the best move for the maximizing player)

If next board value equals to the current value, add this movement to candidate set. (t's one of the best moves)

```
 else:
      value=float('inf')
      for col in move:
        next=game.drop_piece(grid, col)
        next_value, _= minimax(next, depth-1, True, dep)
        if next_value<value:
          candidate={col}
          value=next_value
        elif next_value==value:
          candidate.add(col)
```

If it's minimizing player's turn, initialize board value with infinity and recursively call minimax function to simulate opponent's possible next move.

If the next board value is less than current board value, update the value with next board value and reset the candidate set with current movement. (It's the best move for minimizing player)

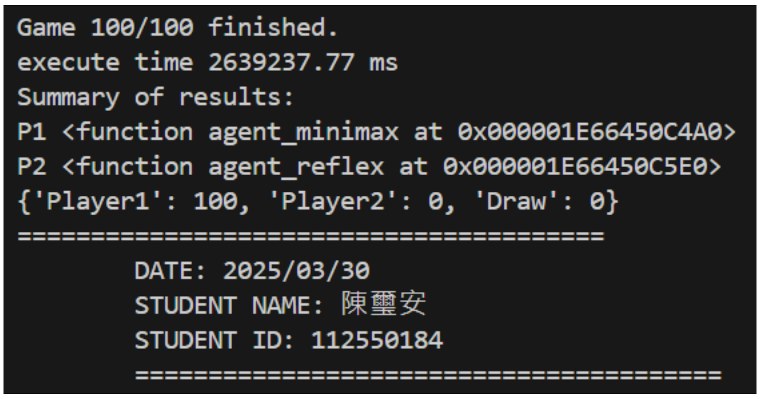
If the next board value equals to the current board value, add the current movement to the candidate set.(It's one of the best move for minimizing player)

```
return value, candidate
```

return the best value found for the current player and candidate movement set.

Results and Evaluation:

Based on the screenshot below, the winning rate is 100% and the execution time is 2639237.77 ms. We can observe that if we used minimax searching to make our decision, it's very possible to win the game.



```
Game 100/100 finished.
execute time 2639237.77 ms
Summary of results:
P1 <function agent_minimax at 0x000001E66450C4A0>
P2 <function agent_reflex at 0x000001E66450C5E0>
{'Player1': 100, 'Player2': 0, 'Draw': 0}
========================================
        DATE: 2025/03/30
        STUDENT NAME: 陳璽安
        STUDENT ID: 112550184
        ========================================
```

2.2 Alpha-Beta Pruning

It enhances the minimax algorithm by eliminating branches that cannot influence the final decision. Alpha is the best value so far for maximizing player and beta is the best value for minimizing player. We cut the branch if alpha>=beta for the reasons below.

1. From the maximizing player's perspective:

   They already found a move that guarantees at least alpha. If the minimizing player can choose a value that is less than alpha after maximizing player's move, then the maximizing player will not choose that move. So it doesn't need to explore the branch any further.

2. From the minimizing player’s perspective:

   They already found a move that guarantees at most beta. If the maximizing player can choose a value that is more than beta after minimizing player’s move, the minimizing player will not choose that move. So we don’t need to explore that branch any further.

Parameters:

A. grid : the current board status
B. depth : how many we want to move forward.
C. maximizing player: If == True then the current player is the maximizing player, and if == False it’s minimizing player.
D. alpha : the best value so far for the maximizing player.
E. beta : the best value found so far for the minimizing player.
F. dep : maximum depth (=4)

Code explanation:

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

If depth reaches 0 (it’ s the last step we want to search) or if the game has terminated, returns the heuristic value of current board and an empty set of candidate moves.

```
move = grid.valid

    if not move:
      return 0, set()

    candidate=set()
    value=0
```

Gets all valid moves. If it’s empty, returns value of 0 and empty set.
Initialize set of candidate move be empty set and board value with 0.

```
 if maximizingPlayer:
      value=float('-inf')
      for col in move:
        next=game.drop_piece(grid, col)
        next_value, _ = alphabeta(next, depth-1, False, alpha, beta,
dep)
        if next_value>value:
          value=next_value
          candidate={col}
        elif next_value==value:
          candidate.add(col)
```
When it’s maximizing player’s turn, it performs the similar operation as in maximizing player in minimax (but recursively call alphabeta function).

```
if value>alpha:
          alpha=value
        if beta<=alpha:
          break
```
Update alpha (best(max) value for maximizing player)

Pruning the remaining branches if beta <= alpha (Since the remaining branches will not affect the final decision)

```
else:
      value=float('inf')
      for col in move:
        next=game.drop_piece(grid, col)
        next_value, _= alphabeta(next, depth-1, True, alpha, beta, dep)
        if next_value<value:
          candidate={col}
          value=next_value
        elif next_value==value:
          candidate.add(col)
          candidate.add(col)
```
When it’s minimizing player’s turn, it performs the similar operation as in minimizing player in minimax (but recursively call alphabeta function).

```
if value<beta:
          beta=value
        if beta<=alpha:
          break
```
Update beta (best(min) value for minimizing player)

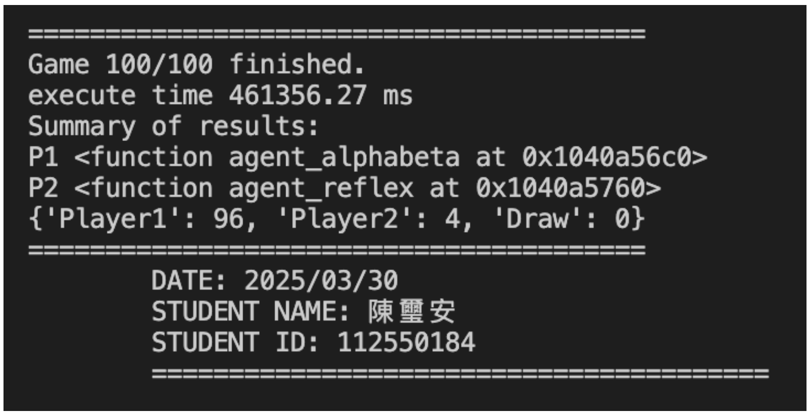
Pruning the remaining branches if beta <= alpha

```
return value, candidate
```

return the best value and candidate move set.

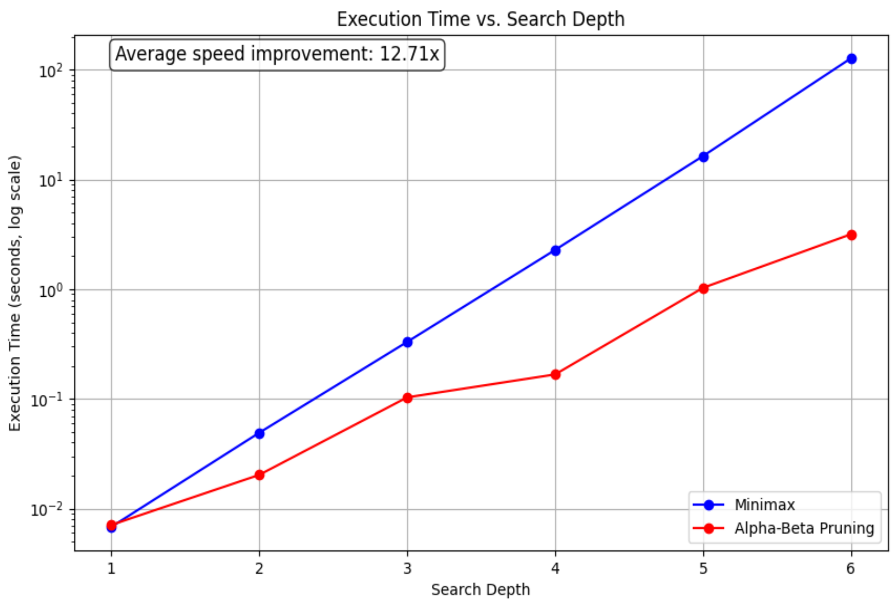
Results and Evaluation:

As we can see from the screenshot below, alphabeta pruning is much faster than minimax searching(alphabeta pruning runs 100 cases in 461356.27 ms while minimax searching runs 100 cases in 2639237.77 ms). But the winning rate of alphabeta pruning is a little bit lower than minimax(won 96 games out of 100 games).



```
========================================
Game 100/100 finished.
execute time 461356.27 ms
Summary of results:
P1 <function agent_alphabeta at 0x1040a56c0>
P2 <function agent_reflex at 0x1040a5760>
{'Player1': 96, 'Player2': 4, 'Draw': 0}
========================================
        DATE: 2025/03/30
        STUDENT NAME: 陳璽安
        STUDENT ID: 112550184
        ========================================
```

Below is the graph of execution time vs. search depth of minimax searching and alphabeta pruning. Alphabeta pruning is faster than minimax searching with the searching depth from 1 to 7, and has an overall speed up of 12.71x compared to minimax searching. Besides, we can observe that alphabeta pruning has better improvement with bigger depth number.



2.3 Strong AI Agent

Parameters:

A. grid : the current board status

B. depth : how many we want to move forward.

C. maximizing player: If == True then the current player is the maximizing player, and if == False it's minimizing player.

D. alpha : the best value so far for the maximizing player.

E. beta : the best value found so far for the minimizing player.

F. dep : maximum depth (=4)

1. **your_function:**

   The function is based on alphabeta pruning, the main differences is I used redesigned get_heuristic_strong instead of get_heuristic function. Besides I sort the moves by their column, the agent will choose the move that is closer to the center column.

Code explanation:

If reach depth 0 or the game is terminated, evaluate the current board status by function get_heuristic_strong and return the value and an empty set of candidate move.

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

    move = grid.valid

    if not move:
      return 0, set()
```

Sort the column by their distance from the center, so the column that closer to the center will be chosen first.

```
center = grid.column // 2
    # Sort columns by distance from center (center first)
    move = sorted(move, key=lambda col: abs(col - center))
```

The remaining code does the same implementation as alpha-beta pruning.

(skip the detail code)

2. get_heuriscit_strong

Set the current player as player 2 and opponent as player 1. If player 2 wins, return a large negative value (1e-10). If the opponent (opponent=1) wins, return a large positive value (1e10). This ensures immediate wins/losses are prioritized

```
now=2
opponent=1

#we win
if board.win(now):
  return -1e10
#opponent win
if board.win(opponent):
  return 1e10
```

Initialize the evaluated board value as zero, and check the immediate winning moves for both players. Adds/subtracts large values to prioritize winning or blocking.

```
score=0

   immediate_win = False
   immediate_loss = False

   for col in board.valid:
       # Check if we can win in the next move
       if game.check_winning_move(board, col, now):
           immediate_win = True
           score -= 1000000

       # Check if opponent can win in the next move
       if game.check_winning_move(board, col, opponent):
           immediate_loss = True
           score += 1200000  # Higher penalty to prioritize blocking
```

Evaluate center control: give higher weight for lower rows

```
center=board.column//2

    center_score = 0
    for row in range(board.row):
        if board.table[row][center] == now:
            center_score -= 7 * (row + 1)  # Higher weight for lower 
positions
        elif board.table[row][center] == opponent:
            center_score += 8 * (row + 1)  # Even higher penalty for 
opponent center control

    score -= center_score * 10
```

Evaluate center control: weight the column with the distance to the central column.

```
for col in range(board.column):
      col_weight = 6 - abs(col - center)
      for row in range (board.row):
        row_weight= row +1
        if board.table[row][col] == now:
            score -= col_weight *row_weight* 3
        elif board.table[row][col] == opponent:
            score += col_weight *row_weight* 3.5
```

Count 2 and 3 pieces windows for both players, and gives higher weight to 3-piece windows.

```
for piece_count in [2, 3]:
      # Our pieces
      window_count = game.count_windows(board, piece_count, now)
      score -= window_count * (80 if piece_count == 2 else 800)

      # Opponent pieces
      opp_window_count = game.count_windows(board, piece_count, 
opponent)
      score += opp_window_count * (100 if piece_count == 2 else 1000)
```

Identifies moves that create multiple potential winning threats by count 3-piece in current board and next board. Adds a significant bonus for creating multiple 3-

piece windows.

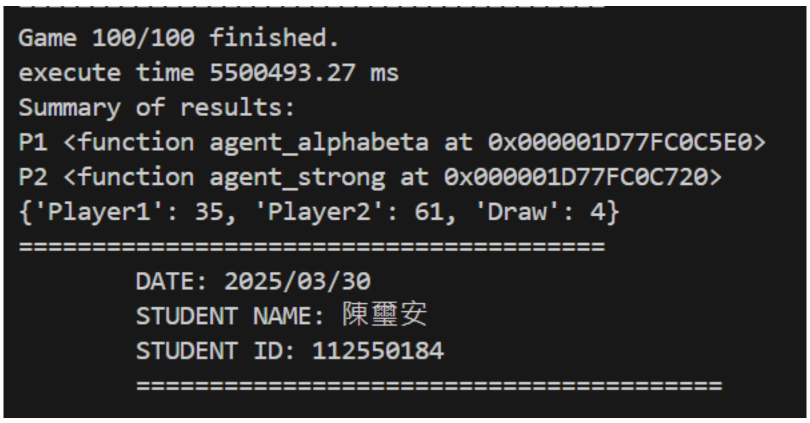
```
three_in_row_cols = 0
    for col in board.valid:
      next_board = game.drop_piece(board, col)
      if game.count_windows(next_board, 3, now) >
game.count_windows(board, 3, now):
        three_in_row_cols += 1

    if three_in_row_cols > 1:
      score -= 10000  # Bonus for creating multiple threats
```

return the final evaluated value.

```
    return score
```

Result & Evaluation:



```
Game 100/100 finished.
execute time 5500493.27 ms
Summary of results:
P1 <function agent_alphabeta at 0x000001D77FC0C5E0>
P2 <function agent_strong at 0x000001D77FC0C720>
{'Player1': 35, 'Player2': 61, 'Draw': 4}
========================================
        DATE: 2025/03/30
        STUDENT NAME: 陳璽安
        STUDENT ID: 112550184
        ========================================
```

The strong agent successfully beat alpha-beta agent and won over 50 games out of 100 games (alpha-beta agent won 35 games while strong agent won 61 games)

But the execution time of strong agent much more than alpha-beta agent and minimax agent.

3 Analysis & Discussion

Difficulties in designing a strong heuristic:

1. It required long experiment time to find the right balance, since execution time of strong agent is longer than other agents.
2. Need to consider both offensive and defensive strategies simultaneously.
3. Challenging to weight different aspect (center control, winning

sequence, piece position....)

Weakness of agent strong:

1. Complex heuristic function increase computation time: execution time of agent strong is much longer than agents used original heuristic function.
2. Potential performance overhead due to multiple scoring calculations.
3. Heuristic function may be predicable.

Potential enhancement:

1. Machine learning integration: use reinforcement learning to dynamically adjust heuristic function or train the agent against multiple playing styles.
2. Use advanced search techniques such as MCTS.
3. Doing more move ordering from different aspects.
4. Stored computed board status to avoid repeated calculation to reduced execution time.
5. Simplify the heuristic function to shorten the execution time.

4 Conclusion

In this homework, we explored advanced adversarial search techniques for Connect Four, implementing three key components: Minimax search, Alpha-Beta pruning, and a stronger AI agent.

Key achievement

Successfully implemented recursive Minimax and Alpha-Beta search algorithms and design a stronger heuristic function that can beat original heuristic function with over 50% winning rate.

Future improvement:

Although the strong agent has higher winning rate compared to the agents with original heuristic function, its computation time is much longer. To enhance the heuristic function, we need to shorten the execution time. Besides, we can improve the function by using advanced searching technique.