CS580 Project 2
Justin Yang
jyang52
G01135050

## Introduction:

Connect Four is an adversarial, turn-based game that ends when a number of one player's pieces form a certain pattern on the board. When playing against an algorithm-based agent, the agent would apply an adversarial search to find the best moves to satisfy the win condition. An adversarial search is a search algorithm that involves multiple players competing against each other. It considers the possible moves both players can make and evaluates them using a heuristic based on the likelihood of winning. The algorithm then picks the moves that maximizes the chances of winning while minimizing the opponent's ability to win. It is commonly used in games such as chess or checkers. We, in this paper, will example the application of the minimax algorithm with alpha-beta pruning to the Connect Four game. First, the paper will explain the minimax and alpha-beta pruning algorithms, and the modified win conditions of this implementation of Connect Four. Then it will detail the proposed approach and the strategy to assign points for the depth of the minimax algorithm. Finally, it will present the experimental results for varying values of depth and different columns.

## Background:

The minimax algorithm is a recursive algorithm used to determine the best move for a player to make in a deterministic, complete game. The algorithm builds a game state tree and recursively searches through the player's possible moves. The algorithm alternates between maximizing and minimizing the score at each level of the tree. It continues this alternating search until it reaches an end state, at which point it assigns a heuristic score to the state dependent on whether a player won, lost, or tied. The scores are then propagated back up. Finally, the algorithm chooses the move that leads to the highest score for the maximizing player or the lowest score for the minimizing player.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

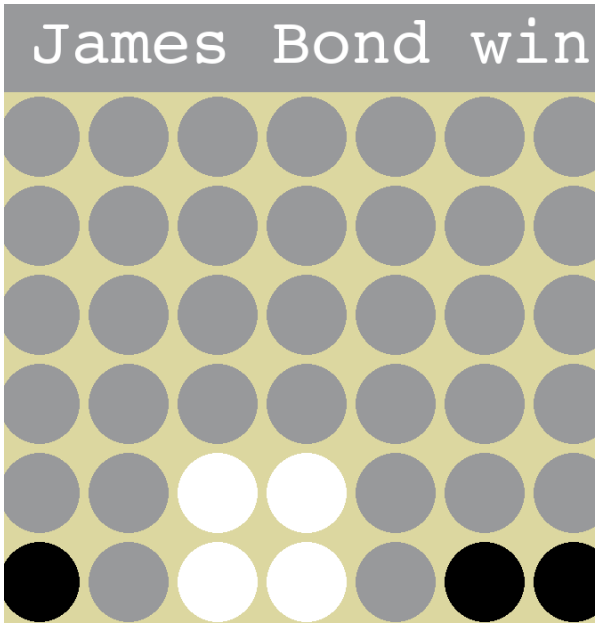
The alpha-beta pruning algorithm is a technique used within the minimax algorithm. It saves both time and resources by reducing the number of game states that need to be evaluated. It keeps track of two values: the alpha and beta. The alpha value is the lowest score that the maximizing player will achieve, and the beta value is the highest value that the minimizing player can achieve. The algorithm updates the alpha and beta values at each of the game state tree levels based on the scores of the states that have been evaluated. If it visits a node that is guaranteed to be worse than the current alpha or beta value, then it trims the subtree from the set of states to be evaluated, since there is no possible way that any of the subtree could have a better value than what has already been encountered.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            if value > β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            if value < α then
                break (* α cutoff *)
            β := min(β, value)
        return value
```

## Proposed Approach:

In this implementation of minimax with alpha-beta pruning, we will apply it to the game of Connect Four. There are several key adaptations of the standard minimax algorithm to apply it to Connect Four. These include adding a heuristic function for evaluating the positions of the pieces on the board. We will also modify the win conditions of this game. While Connect Four

traditionally ends when a player has placed four consecutive pieces on the board, either linearly or diagonally, in this implementation, the game will end when a player has put the pieces in a square pattern. That is, for any given space *(x, y)*, a player will win if there are also pieces in *(x + 1, y)*, *(x, y + 1)*, *(x + 1, y + 1)* that are of the same color and all coordinates fall within the board space.




The approach used in the algorithm will resemble the minimax algorithm above with some key modifications to adapt it for use with evaluating Connect Four. First will be the inclusion of win condition detection. This function will check whether four pieces for a single color exist in a square on the board. It, for any given (x, y) coordinate of the board, will check in (x + 1, y), (x, y + 1), and (x + 1, y + 1) for matching pieces. If it is a winning move, then the minimax algorithm will return a boolean value

```
def winning_move(board, piece):
    #Square configuration win condition
    for c in range(COLUMN_COUNT - 1):
        for r in range(ROW_COUNT - 1):
            if board[r][c] == piece and board[r][c+1] == piece and board[r+1][c] == piece and board[r+1][c+1] == piece:
                return True
```

The heuristic that evaluates the position of the board is split into two functions: evaluate_window and score_position. Score_position assigns a single number that evaluates which player has the advantage on the board. It gives the center of the board the advantage, since there are more possibilities to achieve the win conditions when not limited by the edge of the board. It then iterates through the board and uses the evaluate_window function to assign points to a subsection of the board it is considering. Evaluate_window assigns the highest score to a subsection of the board that has four pieces of a player. This represents a victory condition, so the algorithm would pursue this more intensely. It gives higher preference to more pieces in

the subsection. However, if it detects a high number of opponent pieces, it subtracts points in order to indicate that this board position may be an undesirable outcome.

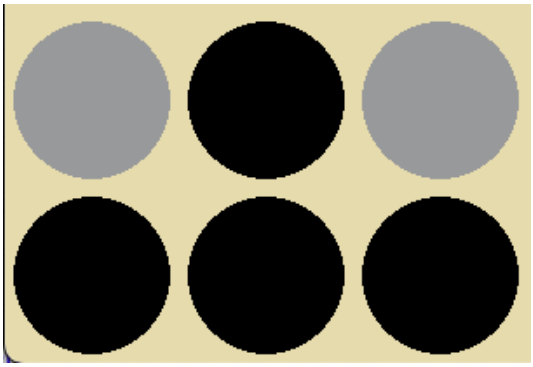
```
def evaluate_window(window, piece):
    #Need to compare the opponent's state with
    score = 0
    #print(window)
    opp_piece = PLAYER_PIECE
    if piece == PLAYER_PIECE:
        opp_piece = AI_PIECE

    if window.count(piece) == 4:
        score += 100
    elif window.count(piece) == 3 and window.count(EMPTY) == 1:
        score += 5
    elif window.count(piece) == 2 and window.count(EMPTY) == 2:
        score += 2

    if window.count(opp_piece) == 3 and window.count(EMPTY) == 1:
        score -= 4

    return score

def score_position(board, piece):
    score = 0

    center_array = [int(i) for i in list(board[:, COLUMN_COUNT//2])]
    center_count = center_array.count(piece)
    score += center_count * 3

    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT - 3):
            window = [board[r + i][c + i] for i in range(WINDOW_LENGTH)]
            score += evaluate_window(window, piece)

    return score
```

## Experimental Results:

The game was tested using the following method. 35 plays were executed. In each play, each of the seven different columns and each of the five depths from 1 - 5 were used for a total of 35 runs. Two different strategies were used by the human player against the algorithm. The first we will call the “Immediate Payoff” strategy. In this strategy, I prioritized two actions as the player:

1. If the computer was about to win, block it.
2. Attempt to build a square as directly as possible. Almost a “brute force” method in that no planning for additional actions in the future was made

The second strategy I employed is what I will refer to as the "Absolute Win" strategy. When not in imminent danger of losing, I would prioritize the formation of a certain formation of pieces such that when it was created, it was sure to end in my victory. For example, the following formation is sure to win, since I have two opportunities to win while the opponent can only block one.



The following tables specify win/loss from the player perspective. That is, when the table indicates a win it means that the human playing the game was victorious, rather than the computer.

Immediate Payoff strategy

| | Column | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Depth | | | | | | | | |
| 1 | | Draw | Draw | Draw | Draw | Draw | Draw | Draw |
| 2 | | Draw | Draw | Draw | Draw | Draw | Draw | Draw |
| 3 | | Draw | Draw | Draw | Draw | Draw | Draw | Draw |
| 4 | | Draw | Draw | Draw | Draw | Draw | Draw | Draw |
| 5 | | Draw | Draw | Draw | Draw | Draw | Draw | Draw |

Absolute Win strategy

| | Column | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Depth | | | | | | | | |
| 1 | | Win | Win | Win | Win | Win | Win | Win |
| 2 | | Win | Win | Win | Win | Win | Win | Win |
| 3 | | Win | Win | Win | Win | Win | Win | Win |
| 4 | | Draw | Draw | Draw | Draw | Draw | Draw | Draw |
| 5 | | Draw | Draw | Draw | Draw | Draw | Draw | Draw |

## Conclusions:

During the play using the immediate payoff strategy, it resulted in draws for all combinations of depth and starting column. Due to the heuristic, the computer would immediately block any imminent win, and I would do the same. Also due to the heuristic, the computer would also attempt to build a square as directly as possible. In these ways, the algorithm and the player using this strategy were mirrored, which resulted in draws.

The effects of the difference in depth become much more evident in the absolute win strategy. To the low-depth opponent, since the absolute win formation is not strictly an imminent win, the algorithm would not be able to detect the danger. It would only react after it is too late. However, for a higher-depth opponent, they would be able to spot the danger since they look farther into the future. This difference is highlighted by the variation in results from games played with depths lower than four versus higher than 3. While depths of 1, 2, and 3 ended in loss for the algorithm, the algorithm with depths of 4 and 5 were able to successfully spot and defend the impending danger into a draw. Clearly, for situations that give an opponent an advantage in board position that does not result in an immediate end game condition, a higher depth is necessary in order to look further ahead in the future and anticipate possible outcomes. Higher depths would presumably result in victories for the algorithm, but the runtime was too long beyond a depth of 5. Future improvements to the algorithm could include modifying the heuristic to identify formations that would result in the “Absolute Win” and score them with increased weight in order to both pursue and defend against them. Another improvement would be implementing this code in a compiled language rather than an interpreted language in order to increase performance to a point where higher depths could be effectively tested.