Ayesha Lala
Khushal Jobanputra
Kishan Shah

FourPlay with minimax.

Game overview:

- Connect four is 2 player game where players take turns dropping game pieces into a 6X6 grid board.
- Game pieces stack vertically from bottom row.
- At each turn, players drop a game piece into the column of their choosing. The game piece falls to the lowest available spot in the game board at that column.
- The game ends when one player has aligned 4 of their colored game in a row on the game board (vertically, diagonally, and horizontally)
- There is a tie if all pieces are placed on the board without 4 game pieces of the same color aligned.

Domain

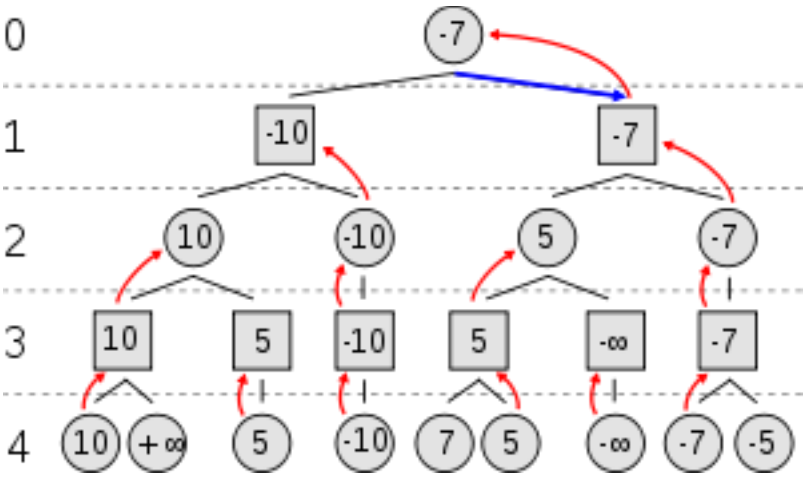
Different AI techniques were studied and an algorithm was chosen. There are many ways to solve the Connect 4 game. There are several levels of AI difficulties. They are random, defensive and aggressive AI. Random is just like it sounds; it randomly picks a play and is the easiest to beat. Defensive AI makes blocking a win a priority, while aggressive makes winning a priority. Both are harder to beat than the random AI. Solution algorithms for AI are numerous and can be complex. Some that were considered are minimax, minimax (with alpha-beta pruning), A* and influence maps. Minimax is a recursive tree that uses backtracking to find the optimal move and is difficult to beat. The opponents are referred to as MIN and MAX. MIN tries to minimize MAX's score and MAX tries to maximize his score. The algorithm then takes this and looks several moves ahead for the best move possible. Minimax may be the best way of getting the optimal move, but it takes a lot of processing, so pruning methods are used. The pruning method that was considered is the alpha-beta method. Since minimax looks at all possible plays including ones that can be ignored, alphabeta pruning is used to improve the efficiency of the minimax algorithm . It scores the possible plays and if it is at a MAX node, it only looks down the branches that have a score greater than or equal to that of the MAX node and if it's a MIN node, it looks for a score that is less than or equal to the MIN node . Even with these pruning methods found in, this type of AI is too complex for the existing program. The A* algorithm was also looked at. A* is a best first search that combines the path cost from the start to the end and the estimated cost of the cheapest path. This method would not be a good fit for the Connect 4 game. The algorithm used in this program is based on using heuristics with influence mapping that is seen in. This was chosen because it is the method that fits in with the original code for the game.

Ayesha Lala
Khushal Jobanputra
Kishan Shah

On starting with the implementation of connect four with python and worked on it further by modifying and the implementing the game mode to allow for simulation.

We constructed algorithms to play against the computer. These included a random playing heuristic and a variation of Minimax heuristic.

Now here, for every two-person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that:
(a) Given player 2's strategy, the best payoff possible for player 1 is V, and
(b) Given player 1's strategy, the best payoff possible for player 2 is −V.



A game with just two possible moves on each turn will have a search tree as shown in the above picture. The circles represent possible moves by the current player (max), while the squares represent the opponent's moves (min).

Ayesha Lala
Khushal Jobanputra
Kishan Shah

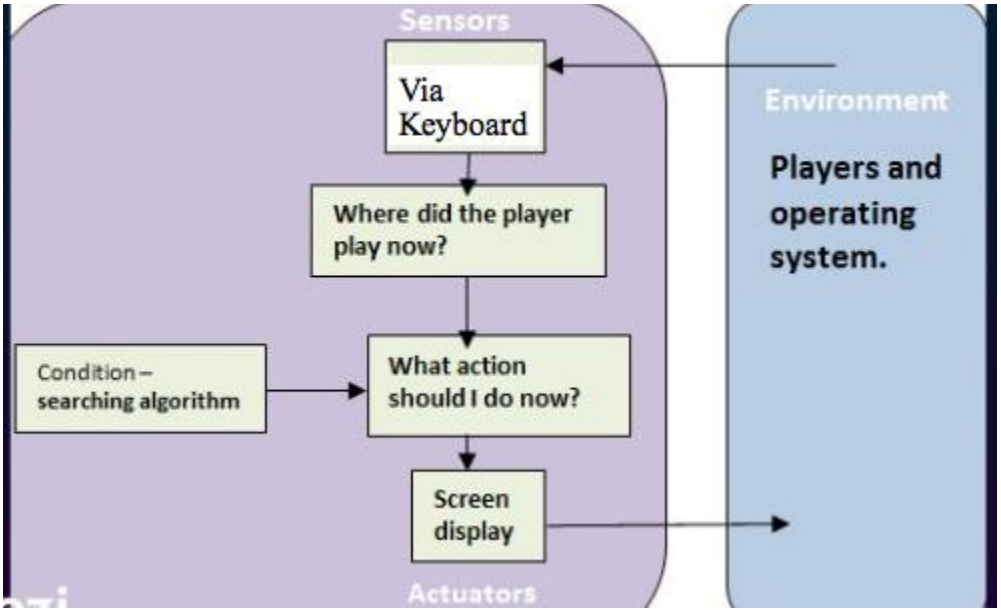
Problem representation

This program improvised on the random algorithm by adding minimax algorithm.

PEAS of applications:

Performance: Winning the game in best possible manner.
Environment: Players (Human and computer) and the operating system.
Actuators: Displaying the connect four.
Sensors: The inputs via keyboard.

Connect four is:
Fully
Deterministic
Sequential
Static
Discrete
Single



Ayesha Lala
Khushal Jobanputra
Kishan Shah

Goals and General approach

Player set: Minimax algorithm with minimax difficulty level : 1,2,3,4,5
There were 3 sets of experiments that we performed
First experiment: Test each player in a series of simulations against an uninformed random heuristic.
Second experiment: Play every player vs minimax of depth or difficulty 1
Third experiment: Play every player in the set vs a minimax of depth or difficulty 2
We tested the 5 depth levels of minimax against minimax with various depth levels.
We also considered the case where which player made the first move to look at the effects.
Additional our other goals were to measure the effectiveness of the minimax at different depth levels by looking at differences in efficiency, number of moves it takes to finish the game and who wins the game to determine this effectiveness
Goals of experiments: Look at differences in time efficiency, number of moves taken to finish the game, winner.

Solution design

Here we've used the minimax procedure and how it's applied to an AI opponent in a game such as Connect 4, there moves and counter moves made by each player. As a human player we think ahead and try to counter a move based on what we think the opponent will do next. So the problem in developing an artificially intelligent agent is how that how can we as programmers make a computer simulate the same forward thinking. For us humans it comes naturally but for AI it needs to be programmed to think ahead. One way to do this is for the computer to generate every possible game moves, from the starting move all the way to the very last move and based on this information choose the best move that will lead to a loss or draw for the human player. For games like connect 4 searching every possible game move is feasible and doesn't require many resources but for a game like chess, even though there is a finite set of moves, searching all of them would be impossible because there are more chess moves than there are atoms in the entire universe, but for Connect 4 the number of moves is around 400,000. This move list can be represented in the computer as a game tree. Each level of the game tree represents the next player's turn. Now as the game progresses, the number of possible moves becomes less until we reach the final game states or the human wins, loses or draws. To search the tree and choose the best move we use an algorithm called Minimax. In Minimax the human player is trying to maximize the chances of winning and the computer is trying to minimize the human's chances of winning. Each final game there can be assigned value through a heuristic function and these values can be returned to the intermediate game states.
The heuristic value determines how good a possible game move is for the human player, the higher the score the better for the human player. The computer opponent's goal is to lead the human player to a lower score.

```
Function minimax (node, depth, maximizingPlayer ) //returns terminal
value, return the best value for max, best value for minimax

//base case
If depth = 0 or the node is a terminal node
    Return the heuristic value of that node

If maximinzingPlayer is TRUE
{
    bestValue = -infinity
    For each child of the node
    {
    Val = minimax(child, depth -1, FALSE)
    bestValue = max(bestValue, Val)
}
}
else if maximizingPlyer is FALSE
{
    bestValue = +infinity
    For each child of the node
    {
      Val = minimax(child, depth -1, TRUE)
    bestValue = min(bestValue,Val)
            }
    Return bestValue
}
```

Now, the same minimax algorithm with the same game tree of depth 5 but we do it with alpha beta pruning. For each node, Alpha represents the maximum score that the max player can receive and beta represents the minimum score the minimizing player can receive, the goal of the max player is to get the highest score while the goal of the min player is to get the lowest score. Alpha and beta are initialized with the worst possible scores for each player. Thus we can use them prune the branches of the game tree or in other words we can stop exploring the nodes of the subtree that have not been visited yet. The condition to prune is when alpha becomes greater than or equal to beta. When this happens, we can prune.

```
Function alphabeta(node, depth, a,b, minimizingPlayer) //heuristic value, alpha for max, return the beta from min

//base case

If depth = 0 or node is terminal node
    return the heuristic value of thr node

If minimizingPlayer is TRUE
{
    For each child of the node
    {
        a = max(a, alphabeta(child, depth-1, a,b,FALSE))
        if a >= b
            break //Prune
    }

    return a
}
else if minimizingPlayer is FALSE
{
    for each child of the node
    {
        b = min(b, alphabeta(child, depth-1, a, b,TRUE))

        if a >= b
        break
    }
    return b
}
```

Ayesha Lala
Khushal Jobanputra
Kishan Shah

Solution implementation

Game Environment

We've created a simple AI to test our environment.

What we needed:

- A function to get the player's move
- A function to get the computer's move
- A function to decide who plays when and draws the board

The computer's move function will:

1. Check if it can win, if so it will play there
2. Check if the opponent can win, if so it will play there
3. Pick a random move in all the available moves

Our algorithm is a simple minimax algorithm with a little extra added:

- If the computer can win immediately it will play there
- If the opponent can win immediately it will play there

Description of the files

1. main.py

As the name suggests, this is the main component of the program. The main contains a loop that initiates the game class—ConnectFour from the file- connect_four.py

Initially, the menu_choice= 1, this starts with the first round of the game. Once the game is completed, the user is asked to enter their choice. The while loop checks if the menu_choice=1. If it is, then it starts a new game. Else the game will quit.

2. connect_four.py

This is the file that contains code that is used for the implementation of Connect Four. This file contains various functions that are used play the game. The function names and its job has been elaborated below.

Function name	Functionality
start_new(self)	· Initializes the game by setting finished status of the game to false and by setting the winner to none · Selects which player will play first at random · Initializes the grid for the game
Switch_player(self)	· Switched the current player i.e. if the current player is human then switches the current player to AI

Next_move(self)	· Takes the column number where current player wants to input, i.e. performs get move of the current player, and then checks if that column is empty. If it is empty then marks that spot with the current player's color i.e. X/0 · After this it performs the functions check_status(),print_state() and switch_player() · It then increments the round number
Check_status(self)	· Checks if the grid gets full on the current players move. Makes the finished status of the game true if the grid gets full as number of rounds cannot be more than the number of cases. · Checks if the current player move makes four connecting colors. Makes the current player the winner if there are four connecting colors of same player
Is_full(self)	· Checks if the grid we are playing the game in is full
Is_connect_four(self)	· Checks if there are four connecting colors in vertical, horizontal or diagonal
Print_state(self)	· Prints the round number · Prints the grid with the latest move · Prints the result of the game
Player(object)	· It is an abstract class
HumanPlayer(player)	· This is a class which is executed when the human player is the current player · Initializes the color for the human player · Defines the function of get move when the human player is the current player

ComputerPlayer(player)	· This is a class which is executed when the computer is the current player · This class defines the next best move that the AI makes. · There are difficulty levels that can be set in the game implementation which basically change the depth at which the AI performs search for the best move.
Get_best_move(self,grid)	· Works out the best move that the AI can make by calculating the alpha values of each column using find_move, simulate_move and eval_move · Returns the best_move
Find(self, depth, grid, curr_player_color)	· Finds the next move and then returns the next move along with its alpha value to the get_best_move function
Is_legal_move(self, column, grid)	· Runs through every column in the grid to make a list of all the legal moves
Game_is_over(self, grid)	· Checks for streak of 4 for either of the players and returns true
Simulate_move(self, grid, column, color)	· Carries out a temp next possible move by referring to the legal move list
Eval_game(self, depth, grid, playercolor)	· Evaluates the alpha value of the simulated move by evaluating the current game condition
Find_streak(self, grid, color, streak)	· Looks for streaks in the came · The parameter streak decides how long streak is detected by this function

Explanation of the Code

The code begins with start_new() which is the function where the game begins from everytime play again is selected in the menu_choice at the end of the game. start_new()calls start which then calls next_move()
next_move() gets the move from the current player using get_move and then checks status of the game to check if any player has won the game. It then prints the state of the game that is the round number, etc and then switches the player.
The get_move function if the current player is human then just simply takes the input from the user and returns it to the next_move function.
But when the get_move function is executed when the current player is computer at that time the use of minimax and alpha beta pruning is made to get the best move that the AI can make using the get_best_move().

Code snippets and pseudo codes of important functions

1. get_best_move()

Code snippet:

```
get_best_move()
        for col in xrange(CONNECT_FOUR_GRID_WIDTH):
                if self._is_legal_move(col, grid):
                        tmp_grid = self._simulate_move(grid, col, self._color)
                        legal_moves[col] = -self._find(self._DIFFICULTY - 1, tmp_grid,
                        human_color)
…
        for move, alpha in moves:
                 if alpha >= best_alpha:
                        best_alpha = alpha
                        best_move = move
```

Pseudo code:

```
get_best_move()
        for (all columns in the range of the grid width)
                if (legal move in the col is true)
                        Simulates the legal move and stores the new grid in tmp_grid.
Runs find() to store the alpha value of that col in legal_moves[col] where col is the current col in the for loop.
        Now checks each col move from legal_moves[col] to check for the highest alpha value
        Returns the move with the highest alpha value
```

2. find()

Code snippet:

find()

```
for i in xrange(CONNECT_FOUR_GRID_HEIGHT):
        if self._is_legal_move(i, grid):
                tmp_grid = self._simulate_move(grid, i, curr_player_color)
                legal_moves.append(tmp_grid)

            if depth == 0 or len(legal_moves) == 0 or self._game_is_over(grid):
                return self._eval_game(depth, grid, curr_player_color)
…

alpha = -99999999
for child in legal_moves:
if child == None:
        print("child == None (search)")
        alpha = max(alpha, -self._find(depth - 1, child, opp_player_color))
        return alpha
```

Pseudo code:

find()
Simulates all the legal moves of a column at different heights to get the highest alpha value of that column i.e. the value at the bottom most empty slot of the column.
It then evaluates the game to look for streaks in the game using eval_game()

3. eval_game()

Code Snippets:

eval_game()

```
ia_fours = self._find_streak(grid, player_color, 4)
ia_threes = self._find_streak(grid, player_color, 3)
ia_twos = self._find_streak(grid, player_color, 2)
human_fours = self._find_streak(grid, opp_color, 4)
human_threes = self._find_streak(grid, opp_color, 3)
human_twos = self._find_streak(grid, opp_color, 2)
if human_fours > 0:
        return -100000 - depth
else:
        return (ia_fours * 100000 + ia_threes * 100 + ia_twos * 10) - (human_threes * 100 +
        human_twos * 10) + depth
```

Pseudo code:

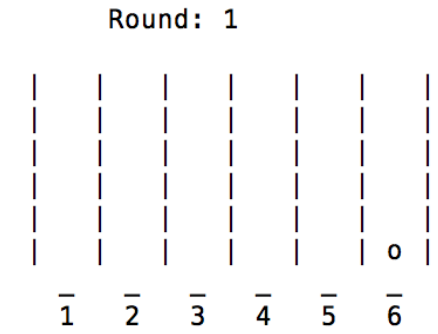
```
eval_game()
        Checks for the streaks of 4,3 and 2 of the AI
        Checks for the streaks of 4,3 and 2 of the Human
        if (the streaks of 4 of human >0)
                return -100000 –depth
        else
                Return ”(ia_fours * 100000 + ia_threes * 100 + ia_twos * 10) - (human_threes *
                100 + human_twos * 10) + depth” alpha value to that column move
```

Game simulation

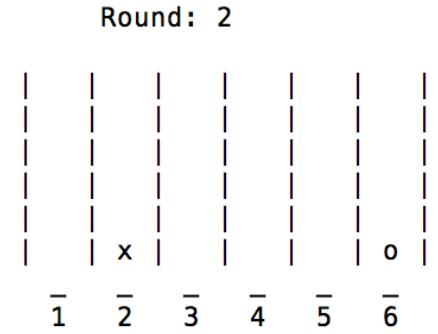
Just to give a better idea of how the game works, here are a few instances that I have explained showing how the IA makes the best move using the eval_move() function explained above

Round 1:
IA moves possible with their alpha values calculated by the eval_move() function
(row, column) → alpha value
(1,1) → -99999998
(1,2) → -99999997
(1,3) → -99999996
(1,4) → -99999995
(1,5) → -99999994
(1,6) → -99999993

Since the values are negative the value at (1,6) is the highest hence the IA makes a move there

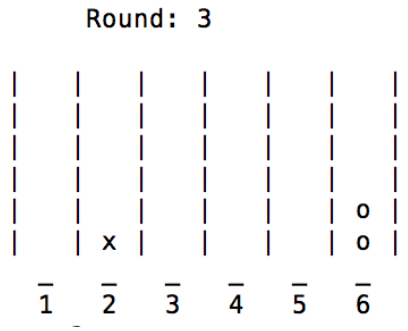


Round 2:
The user makes his move at (1,2)

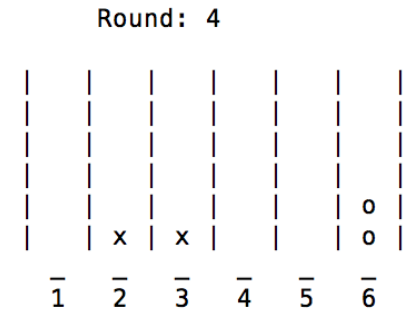


Round 3:
IA moves possible with their alpha values calculated by the eval_move() function
(row, column) → alpha value
(1,1) → -99999998
(2,2) → -99999997
(1,3) → -99999996
(1,4) → -99999995
(1,5) → -99999984
(2,6) → -99999983

Since the values are negative the value at (2,6) is the highest hence the IA makes a move there



Round 4:
The user makes his move at (1,3)



Round 5:

IA moves possible with their alpha values calculated by the eval_move() function
(row, column) → alpha value
(1,1) → -99999998
(2,2) → -99999997
(2,3) → -99999996
(1,4) → -99999995
(1,5) → -99999984
(3,6) → -99999903

Since the values are negative the value at (3,6) is the highest hence the IA makes a move there

```
      Round: 5

|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   | o |
|   |   |   |   |   |   | o |
|   | x | x |   |   |   | o |

  1   2   3   4   5   6
```

After a few more rounds the IA makes the best moves to win the game

```
          Round: 7

|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   | o |
|   |   |   |   |   |   | o |
|   |   |   |   |   |   | o |
|   | x | x | x |   |   | o |

  1   2   3   4   5   6
Game Over!
IA is the winner!
```

Ayesha Lala
Khushal Jobanputra
Kishan Shah

Evaluation and conclusion

There is a disadvantage in looking at too many depth levels, resulting in exponential increase in the processing time of and boards analyzed for each move and an increased number of turns taken to win the game.

Depth level of 4 or 5 seems to be the optimal searching depth.

Looking at higher depth levels is necessary when playing smarter opponents with the better informed search algorithms in order to minimize number of turns.

In some cases, when the weaker player goes first ie. while not playing optimally, it can extend the game length to more turns, and in some cases even cause a tie, where as if you play optimally, stronger player has a higher probability of winning

We have implemented improvement features over standard Minimax and Alpha-Beta searches. For Minimax, we found that the addition of a transposition table added significant pruning. Similarly, move ordering schemes contributed to pruning of Alpha-Beta search. The best algorithm we have found to play the game of Connect-4 is an alpha-beta search with move ordering towards the center of the board. It was not clear whether taking into account threat features into the heuristic significantly improved performance. While an efficient algorithm was found to compute the number of threats, we preferred to have a standard (own winning rows) - (opponent winning rows) heuristic to minimize the time taken to evaluate heuristic values. While this algorithm just described performs well when played against a human player on a game of Connect-4, it does not perform reasonably when gravity is turned off or on larger boards. We believe that simple improvements as suggested in this paper may impact the performance of our algorithm in a positive way.