A Project Activity Report
Submitted for Artificial Intelligence

TOPIC: CONNECT 4 GAME

Submitted by:
Priyanshi Modi - 102117116
Riya Raizada - 102297009
Akanksha Solanki - 102297010

Submitted to:
Dr Paluck

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY,
(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB INDIA

Jan - May 2023

Introduction

Connect Four is a logical two-player connection game in which the players each choose a color and take turns dropping one colored disc at a time from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game is to get four discs in a row either vertically, horizontally or diagonally before your opponent.

Various different AI techniques and algorithms were considered in the development of this game, including the minimax algorithm, alpha-beta pruning as well as the A* algorithm. Minimax is a recursive tree that uses backtracking to find the optimal move, which makes it the hardest to beat (almost guarantees AI's win over the opponent). MIN tries to minimize MAX's score and MAX tries to maximize his score with their moves. The algorithm then takes this move and looks several moves ahead for the best move possible. Minimax, for this reason, may be the best way of getting the optimal move, but it takes a lot of processing, so pruning methods are used instead that improves efficiency by not exploring the entire tree but only looking 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.

Still, however, this type of AI was too complex and so the A* algorithm was used. 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 algorithm was not a good fit for the Connect 4 game, because the A* algorithm works on a fully observable environment, meaning its success was dependent on the view of the entire connect 4 board to evaluate its next move, which isn't true for this game since only the top available spot in each column needs to be looked at before the next move.

For these reasons of inefficiency and unfitness, these algorithms are no longer used. The most suitable algorithm used now is the Monte Carlo Method that estimates possible outcomes based on simulations because of its 'strong way of estimating uncertainty' to efficiently suggest optimal moves.

Literature Survey

Connect 4 was first published by Milton Bradley in 1974. This project was originally done as a real-time project for Embedded Systems. The program was written in C using LabWindows/CVI by National Instruments. It was written for a windows environment. The game was 2 players and had a five second timer for play. The plays were made on an interactive GUI. The current player was displayed on the top of the GUI. Once the current player makes a play, the chip is placed in that column within one second. When the chip is in place, the timer is reset. When the timer expires, the current player lost their turn. Due to the time constraints the computer player was not implemented.

"Mastering the game of Connect Four: Human-like performance with deep neural networks" (2015) by Tesauro et al.

This paper proposes an AI approach using deep neural networks (DNNs) to achieve human-like performance in Connect 4. The authors train a DNN to evaluate game states using supervised learning with a large dataset of expert game positions. They also introduce a novel reinforcement learning algorithm called Temporal Difference with Asynchronous Critics (TDAC), which combines supervised learning and reinforcement learning to fine-tune the DNN. The proposed approach achieves superhuman performance, outperforming previous Connect 4 AI methods.

"Playing Atari with Deep Reinforcement Learning" (2013) by Mnih et al.

Although this paper focuses on Atari games, it introduced the concept of using Q-learning with neural networks, also known as Deep Q-Networks (DQNs). This approach has been widely applied to various board games, including Connect 4. DQNs use a neural network to approximate the Q-function, which represents the expected cumulative reward for taking actions from different states. DQNs have been used to train AI agents for Connect 4 with good performance, and they have been further improved with modifications such as Double Q-networks and Dueling Q-networks.

"Solving Connect Four: Constructing an Expert-Level Player from Monte Carlo Sampled Knowledge" (2007) by Helmstetter et al.

This paper presents an AI approach that combines Monte Carlo tree search (MCTS) with domain-specific knowledge to create an expert-level Connect 4 player. The authors use MCTS, a popular search algorithm in AI, to explore the game tree and select moves based on simulations. They also incorporate domain-specific knowledge, such as opening book moves and pattern-based heuristics, to guide the search and improve performance. The proposed approach achieves high-level play, rivaling human experts in Connect 4.

"Evolutionary Reinforcement Learning in a Connect-Four Playing Neural Network" (1999) by Liebana et al.

This paper proposes an AI approach that combines evolutionary algorithms with reinforcement learning to train a neural network to play Connect 4. The authors use a genetic algorithm to evolve the neural network's weights and architecture, and then use Q-learning for reinforcement learning to fine-tune the network. The proposed approach shows promising results in training a neural network to play Connect 4 effectively, and it demonstrates the potential of combining evolutionary algorithms with reinforcement learning in game playing.

"Optimal Play of Connect-Four" (1988) by Victor Allis.

This seminal paper provides an in-depth analysis of the game of Connect 4 and presents an algorithm that can solve the game and determine the optimal moves. The algorithm, known as the alpha-beta algorithm with heuristic evaluation functions, can search through the game tree and find the best move based on a heuristic evaluation of game states. This paper has served as the foundation for many subsequent AI approaches in Connect 4 and provides important insights into the game's complexity and optimal play.

"Efficiently Searching the Game Tree of Connect-Four" (1989) by Allis and Schaeffer.

This paper presents an optimized version of the alpha-beta algorithm for Connect 4, which significantly reduces the search space by exploiting the game's symmetries and properties. The authors introduce several techniques, such as transposition tables, iterative deepening, and move ordering, to speed up the search process and improve the algorithm's efficiency.

This paper provides important insights into how to effectively search the game tree of Connect 4, which has influenced subsequent AI approaches for the game.

In summary, the literature on AI approaches for Connect 4 is diverse and encompasses various techniques, including optimized search algorithms, reinforcement learning, evolutionary algorithms, and machine learning. These studies have advanced the field of Connect 4 AI and have contributed to our understanding of the game's strategic properties, optimal play, and effective search and decision-making techniques.

Methodology

Technique

Technique used is the **Monte Carlo Method.**
It is a mathematical technique, which is used to estimate the possible outcomes of an uncertain event. Monte Carlo Simulation predicts a set of outcomes based on an estimated range of values versus a set of fixed input values. In other words, a Monte Carlo Simulation builds a model of possible results by leveraging a probability distribution, such as a uniform or normal distribution, for any variable that has inherent uncertainty. It, then, recalculates the results over and over, each time using a different set of random numbers between the minimum and maximum values. In a typical Monte Carlo experiment, this exercise can be repeated thousands of times to produce a large number of likely outcomes. Monte Carlo Simulations are also utilized for long-term predictions due to their accuracy. As the number of inputs increases, the number of forecasts also grows, allowing us to project outcomes farther out in time with more accuracy. When a Monte Carlo Simulation is complete, it yields a range of possible outcomes with the probability of each result occurring.

Algorithm

1. Initialize the game board with an empty grid of 6 rows and 7 columns.
2. Choose which player goes first.
3. Set the number of simulations that the Monte Carlo AI will run for each possible move.
4. Begin the game loop:
   a. If it is the AI player's turn:
   i. For each possible move, simulate the game with the given move using Monte Carlo method.
   ii. Calculate the win rate of each move by averaging the results of the simulations.
   iii. Choose the move with the highest win rate and apply it to the game board.
   iv. Check for a win or draw condition.

   b. If it is the other AI player's turn:
   i. For each possible move, simulate the game with the given move using Monte Carlo method.
   ii. Calculate the win rate of each move by averaging the results of the simulations.
   iii. Choose the move with the highest win rate and apply it to the game board.

iv. Check for a win or draw condition.
c. If the game is over, exit the loop.

5. Print the final state of the game board and the winner (if any).

Code and Explanation

```
def Get(b, level, col):
    if col < 0 or col > BX or level < 0 or level > BY:
        return CIRCLE_INVALID
    return b[level][col]
```

This function retrieves the value at a specific level and column on the game board (b), which represents the current state of the Connect Four game. It returns the value at that position, which can be CIRCLE_INVALID (if the level or column is out of bounds), CIRCLE_EMPTY (if the position is empty), CIRCLE_YELLOW (if the position is occupied by a Yellow player's disc), or CIRCLE_RED (if the position is occupied by a Red player's disc).

```
def Set(b, level, col, value):
    b[level][col] = value
```

This function sets the value at a specific level and column on the game board (b) to a given value. It takes as input the level, column, and value to be set.

```
def ColIsFull(b, col):
            return (Get(b, BY, col) != CIRCLE_EMPTY)
```

This function checks if a specific column on the game board (b) is full, i.e., if all the positions in that column are occupied by discs. It returns True if the column is full and False otherwise.

```
def Drop(b, col, value):
    if (ColIsFull(b,col) == CIRCLE_INVALID):
        return 0
    for level in range(0,BOARD_HEIGHT):
```

```
        if Get(b, level, col) == CIRCLE_EMPTY:
            Set(b, level, col, value)
            break
    return 1
```

This function simulates dropping a disc of a given value (representing the player) into a specific column on the game board (b). It checks for the first available position from the bottom of the column and sets the value at that position. It returns 1 if the drop is successful and 0 if the column is full.

```
def printGameBoard(b):
    export = ""
    if CLEARABLE:
        os.system('cls||echo -e \\\\033c')
    if COLORAMA:
        export = export + colorama.Fore.YELLOW + SMALL_BANNER + colorama.Style.RESET_ALL + "\n"
    else:
        export = export + SMALL_BANNER + '\n'+ '\n' +'\n'
    for level in range(BY, -1, -1):
        export = export + str(level)
        for col in range(0, BOARD_WIDTH):
            color = Get(b, level, col)
            if COLORAMA:
                if color == CIRCLE_YELLOW:
                    export = export + colorama.Fore.YELLOW"[Y]"+colorama.Style.RESET_ALL
                elif color == CIRCLE_RED:
                    export = export + colorama.Fore.RED+"[R]"+colorama.Style.RESET_ALL
                else:
                    export = export + "[ ]"
            else:
                export = export + "[" + str(cellOptions[color]) + "]"
        export = export + "\n"
    export = export + " "
    for col in range(0, BOARD_WIDTH):
        export = export + " " + str(col) + " "
    print(export)
```

This function prints the current state of the game board (b) in a human-readable format, showing the discs of Yellow and Red players at their respective positions.

```
def GetWinner(b):
    empty = 0
    sp = 0
    for level in range(BY, -1, -1):
        for col in range(0, BX):
            color = Get(b, level, col)
            if (color == CIRCLE_EMPTY):
                empty = empty + 1
                continue
            directory = [[1,0],[0,1],[1,1],[-1,1]]

            for d in range(0, 4):
                start_col = col
                start_level = level
                while(Get(b, start_level-directory[d][1], start_col-directory[d][0]) == color):
                    start_col = start_col - directory[d][0]
                    start_level = start_level - directory[d][1]

                count = 0
                while(Get(b, start_level, start_col) == color):
                    count = count + 1
                    start_col = start_col + directory[d][0]
                    start_level = start_level + directory[d][1]
                if (count >= 4):
                    return color
    if(empty <= BOARD_HEIGHT * BOARD_WIDTH):
        return CIRCLE_EMPTY
    return CIRCLE_DRAW
```

This function determines the winner of the game by checking the current state of the game board (b). It checks for four consecutive discs of the same color in horizontal, vertical, and diagonal directions. It returns CIRCLE_EMPTY (if the game is still ongoing), CIRCLE_YELLOW (if Yellow player wins), CIRCLE_RED (if Red player wins), or CIRCLE_DRAW (if the game is a draw).

```
def RandomGame(b, tomove):
    for i in range(0, BOARD_HEIGHT * BOARD_WIDTH):
        potentialMoves = [x for x in range(0,BOARD_WIDTH)]
        shuffle(potentialMoves)
        for move in potentialMoves:
            if(not ColIsFull(b, move)):
                nextMove = move
                break
        if (Drop(b, nextMove, tomove)):
            if(tomove == CIRCLE_YELLOW):
                tomove = CIRCLE_RED
            else:
                tomove = CIRCLE_YELLOW
        winner = GetWinner(b)
        if (winner != CIRCLE_EMPTY):
            return winner
    return CIRCLE_DRAW
```

This function simulates a random game between two players (Yellow and Red) on the game board (b). It randomly selects columns to drop discs into and keeps track of the winner based on the final state of the game board. It returns the winner of the game (CIRCLE_YELLOW, CIRCLE_RED, or CIRCLE_DRAW).

```
def SuggestMove(b, tomove, simulations=AI_STRENGTH):
    best = -1
    best_ratio = 0
    if COLORAMA:
        if tomove == CIRCLE_YELLOW:
            print(colorama.Fore.YELLOW + "YELLOW IS THINKING" + colorama.Style.RESET_ALL)
        elif tomove == CIRCLE_RED:
            print(colorama.Fore.RED + "RED IS THINKING" + colorama.Style.RESET_ALL)
    for move in range(0,BX+1):
        ttm = time.time()
        if (ColIsFull(b,move)):
            continue
        won = 0
        lost = 0
```

```
        draw = 0
        print_neutral = 1
        for j in range(0, simulations):
            copy = deepcopy(b)
            Drop(copy, move, tomove);
            if (GetWinner(copy) == tomove):
                return move
            if (tomove == CIRCLE_YELLOW):
                nextPlayer = CIRCLE_RED
            else:
                nextPlayer = CIRCLE_YELLOW
            winner = RandomGame(copy, nextPlayer)
            if (winner == CIRCLE_YELLOW or winner == CIRCLE_RED):
                if (winner == tomove):
                    won = won + 1
                else:
                    lost = lost + 1
            else:
                draw = draw + 1
            if j == AI_STRENGTH/2 and RUSH == True:
                ratio = float(won)/(lost+won+1);
                if(ratio+0.05 <= best_ratio and best != -1):
                    if COLORAMA:
                        print(colorama.Fore.RED + "X" + colorama.Style.RESET_ALL, end = " ")
                        print_neutral = 0
                    break
        ratio = float(won)/(lost+won+1);
        if(ratio > best_ratio or best == -1):
            best = move
            best_ratio = ratio
            if COLORAMA:
                print(colorama.Fore.GREEN + "$" + colorama.Style.RESET_ALL, end = " ")
                print_neutral = 0
        if COLORAMA and print_neutral == 1:
            print(colorama.Fore.YELLOW + "?" + colorama.Style.RESET_ALL, end = " ")
        ttmList.append(time.time() - ttm)
        print("Move", move, ":", round(ratio*100,1), "draws:", draw, "ttm:", round(ttmList[-1],1), "attm:",
round(sum(ttmList)/len(ttmList),1))
```

```
    return best
```

This function suggests the next best move for the AI player whose symbol is tomove on the game board b, by simulating multiple random games (the number of simulations is specified by the simulations parameter, with a default value of 200) and choosing the move that results in the highest win ratio for the AI player. The function returns the column (0 to 6) of the suggested move.
The code checks if the current environment supports colorama by setting the COLORAMA flag.

```
def sim():
    board = deepcopy(emptyBoard)
    while(1):
        printGameBoard(board)
        if showPotentialWinner(board):  return
        Drop(board,SuggestMove(board, CIRCLE_RED, AI_STRENGTH), CIRCLE_RED)
        if showPotentialWinner(board):  return
        printGameBoard(board)
        Drop(board, SuggestMove(board, CIRCLE_YELLOW, AI_STRENGTH), CIRCLE_YELLOW)
```

The sim() is a recursive function that simulates a game by playing random moves for both players (Yellow and Red) until a terminal state (win, lose, or draw) is reached. It then returns the outcome of the game (CIRCLE_YELLOW, CIRCLE_RED, or CIRCLE_DRAW) from the perspective of the player whose turn it is at the beginning of the simulation.
The sim() function is used in the SuggestMove() function to evaluate the best move for the AI player by running simulations and choosing the move with the highest win ratio. This allows the AI player to make informed decisions about which move to make in order to increase its chances of winning the game.

```
def showPotentialWinner(board):
    winner = GetWinner(board)
    if winner != CIRCLE_EMPTY and winner != CIRCLE_DRAW:
```

```
        printGameBoard(board)
        if COLORAMA:
            if winner == CIRCLE_YELLOW:
                print(colorama.Fore.YELLOW+" winner: YELLOW" + colorama.Style.RESET_ALL)
            else:
                print(colorama.Fore.RED+" winner: RED" + colorama.Style.RESET_ALL)
            return True
        else:
            print("winner: ", winner)
            return True
    elif winner == CIRCLE_DRAW:
        printGameBoard(board)
        print("DRAW!")
    return False
```

This function is used to display the potential winning moves for the current player (either Yellow or Red).The function simulates potential moves for the current player and checks if any of those moves result in a winning condition.

```
if __name__== "__main__":


    if "--easy" in sys.argv:
        AI_STRENGTH = 250
    elif "--medium" in sys.argv or "--med" in sys.argv:
        AI_STRENGTH = 500
    elif "--hard" in sys.argv:
        AI_STRENGTH = 1000

    elif "--insane" in sys.argv:
        AI_STRENGTH = 2500

    elif "--master" in sys.argv:
        AI_STRENGTH = 5000

    elif "--demigod" in sys.argv:
        AI_STRENGTH = 10000
```

```
elif "--god" in sys.argv:
    AI_STRENGTH = 100000

if "--norush" in sys.argv:
    RUSH = False

elif "--rush" in sys.argv:
    RUSH = True

if "--pretty" in sys.argv:
    try:
        import colorama
        colorama.init()
        COLORAMA = 1
    except ImportError:
 if "--clearable" in sys.argv:
    import os
    CLEARABLE = 1
   else:
    sim()
```

This function is the entry point of the Connect Four game. It initializes the game board, displays the game board, and takes input from the players to make moves. It calls other functions to handle different aspects of the game, such as dropping discs, checking for a winner, and suggesting moves using AI. It also handles game over conditions and prompts for restarting the game.

Result

The result of this project is that the AI for a Connect 4 game was able to be implemented. It met original requirements and was updated as necessary during programming.
The screenshots shown in this section, is according to the game flow from the starting to the end of the game until one player won. Here the game is played between two players AI vs AI which is trained through the algorithm we discussed in methodology section. Players alternate turns.
There are *4,531,985,219,092* possible unique states of the board. This makes it difficult to brute force the game tree. Thus, we used the Monte Carlo method.
At the end of the game either one of the two players wins or it's a draw (no more checkers can be added because the grid is full).

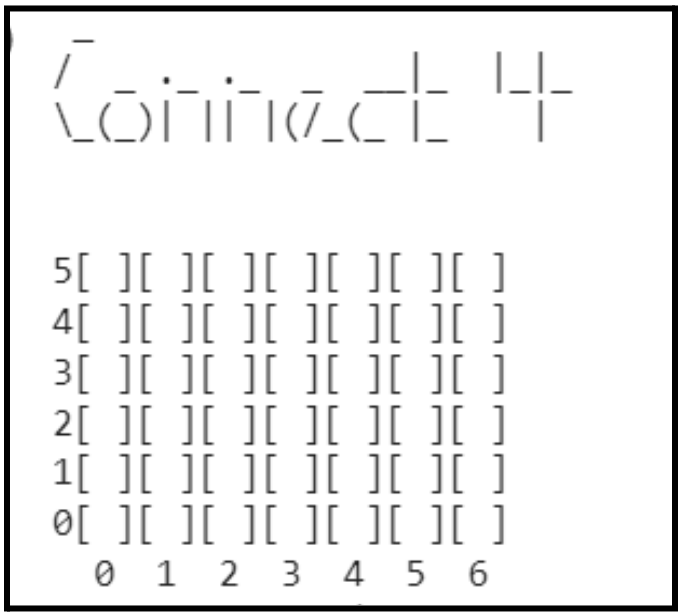

Fig 1. At the beginning of the game the screen will somewhat look like this.

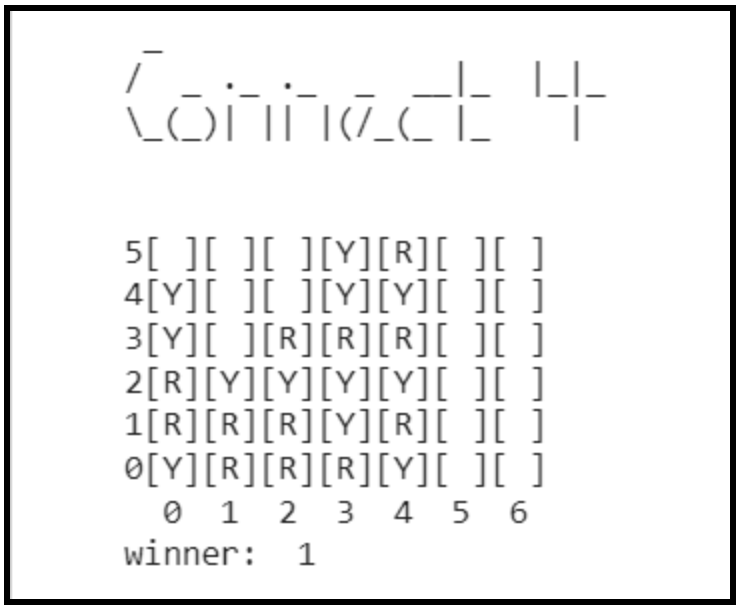

Fig 2. Winner 1 wins the match

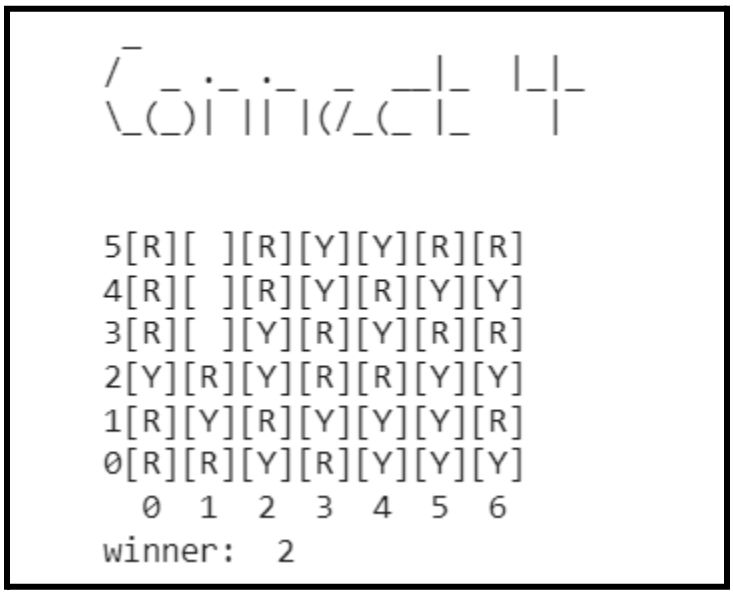

Fig 3. Winner 2 wins the match

Future scope

Monte Carlo method is currently being used in such gaming techniques because of its nature of estimation by simulations. This technique has great future scope aside from gaming algorithms in areas ranging from the sciences to the business world.

Its use in physics can be greatly fruitful in the area of quantum mechanics. The prediction on the basis of simulations can result in many new inventions and completions of current research areas. The problem, however, is that for such research, highly efficient algorithms are needed and so the Monte Carlo method too will need to be modified to reduce its time-complexity for proper use in these science fields.

In business the monte carlo method has great scope in the area of risk analysis. This technique can greatly help businesses better forecast and manage their finances and sales, meaning that with better prediction the businesses can be better prepared for the changes to come in the future and therefore the chances of success are greater, just like the greater chances of success in the game of connect 4.

References

https://www.ibm.com/in-en/topics/monte-carlo-simulation

https://www.sciencedirect.com/topics/medicine-and-dentistry/monte-carlo-method

Connect Four - Wikipedia

https://www.askpython.com/python/examples/connect-four-game

http://faculty.otterbein.edu/wittman1/games/connect4/