CS 4320/5314

Programming Assignment 2:

Game Playing Agents

Group Members:

Menda W Dorji

Sonam S Tshering

**Contributions**

Menda Dorji – Part I: Algorithms for Selecting Moves, Algorithm 1: Uniform Random (UR), Algorithm 4: Upper Confidence bound for Trees (UCT), Part II: Algorithm Tournaments and Evaluation.

Sonam Seldon Tshering – Part I: Algorithms for Selecting Moves, Algorithm 2: Depth-Limited MinMax (DLMM), Algorithm 3: Pure Monte Carlo Game Search (PMCGS), Part II: Algorithm Tournaments and Evaluation, Part III: Enhancements.

Both of us contributed to the group meeting. We both ran and tested the code. We utilized a different test file with different test code to prove our functionality.

 Both of us contributed to documentation, testing, and report writing.

Note: the functions were assigned individually but we both helped equally on each other’s part and gave review as well.

**Introduction**

In this assignment we developed AI agents to play the game Connect Four. The first part we implement four algorithms for selection moves for given board configuration which is the algorithm.py which implements the algorithms.

ConnectFour.py is implementing the game logic and functionality. The Game class encapsulates the state of the game, including the game board, the current player (RED or YELLOW), and parameters such as the number of columns, rows, and the required number of consecutive discs to win. The class includes methods for making moves, checking for a winner or a draw, printing the game board, and creating a copy of the game state.The game board is represented as a 2D list, with each cell containing the color of the disc placed in that position (RED, YELLOW, or NONE for an empty cell). The class provides methods to interact with the game, such as making moves, checking for a win or draw, and printing the current state of the board. Additionally, utility functions like diagonalsPos and diagonalsNeg generate positive and negative diagonals, contributing to the comprehensive win-checking mechanism. Overall, this code serves as the backbone for a Connect Four game, facilitating the implementation of various algorithms and strategies for playing the game.

The main.py is where the main function serves as the entry point for executing the Connect Four game. It takes command-line arguments to specify the input file containing the initial game state and the verbosity level for algorithm output. The function reads the board information from the input file, creates an instance of the Connect Four game using the Game class, and initializes an algorithm instance based on the specified algorithm type. Depending on the chosen algorithm, it calls for the corresponding method to make moves or decisions within the game. The final move selected by the algorithm is printed, considering the verbosity level specified. If any exception occurs during the execution, it is caught, and an appropriate error message is displayed. This script enables the testing and evaluation of various Connect Four playing algorithms by providing a flexible interface for input and output.

What we implemented for the four algorithms are given below:

**Algorithm 1: Uniform Random (UR)**

For this algorithm we implemented a simple strategy to make a move in the connect four games. It begins by obtaining a list of legal moves (columns where a disc can be placed) from the current game state. Then, it randomly selects one move from the list using the random.choice function from the Python standard library. Finally, it returns the randomly chosen to move, representing the column where the player decides to drop their disc. This method essentially represents a strategy where the player makes a move without considering any specific game strategy or evaluation, choosing randomly among the available legal moves.

**Results**

UR

0

R

OOOOOOO

OOOOOOO

OOYOOOY

OOROOOY

OYRYOYR

YRRYORR



UR

0

Y

OOOOOOO

OOOOROO

ROYOOOY

OOROOOY

OYRYOYR

YRRYORR

A screen shot of a computer screen

Description automatically generated

UR

0

R

OOOOOOO

OOOOOOO

OOYOYYY

OOROOOY

OYRYOYR

YRRYYRR

A screen shot of a game

Description automatically generated

**Algorithm 2: Depth-Limited MinMax (DLMM)**

This code defines the dLMinMax method within the Algorithms class for the Connect Four game. The method implements the Deep Learning Minimax (DLMM) algorithm, which is a variant of the Minimax algorithm enhanced with deep learning techniques. It takes two parameters: param, which determines the depth of the search, and verboseType, which controls the verbosity of the output during the algorithm's execution. The method initializes an empty list score to store the evaluation scores for each possible move. It iterates over all legal moves in the current game state, applies the minmax recursive function to calculate the desirability of each move, and appends the resulting score to the scores list. If the verboseType is set to "Verbose," it prints out the move and its corresponding score. Finally, the method selects the move with the maximum score if there are scores available, otherwise, it returns None.

The minmax function is a recursive helper function for dLMinMax that performs the Minimax algorithm with alpha-beta pruning. It evaluates the desirability of a given game state by recursively exploring possible moves up to a specified depth. The function considers both maximizing and minimizing players, updating alpha and beta values accordingly to prune the search tree and improve efficiency. The evaluate method calculates the desirability of a given board state by counting the occurrences of each player's color in various directions (columns, rows, positive diagonals, and negative diagonals). It then computes a simple heuristic as the difference between the current player's score and the opponent's score, scaling the result between -1 and 1. This heuristic serves as the evaluation function used by the Minimax algorithm to guide the decision-making process.

**Results**

DLMM

2

Y

OOOOOOO

OOOOOOO

OOYOOOY

OOROOOY

OYRYOYR

YRRYORR

**A screen shot of a computer

Description automatically generated**

**Algorithm 3: Pure Monte Carlo Game Search (PMCGS)**

For this algorithm the algorithm initiates by creating a duplicate of the current game state to prevent modifications to the original state during simulations. It then repeatedly selects random legal moves and updates the game state, accordingly, simulating potential game outcomes. The method returns 1 if the current player wins, 0 in case of a draw, or continues the loop until one of these conditions is met. By conducting these simulations, PMCGS assesses the desirability of different moves, providing a simple yet effective heuristic for move selection in situations where exhaustive search is impractical due to the vastness of the state space.

**Results**

**A screenshot of a computer

Description automatically generated**

A screenshot of a computer

Description automatically generated

**Algorithm 4: Upper Confidence bound for Trees (UCT)**

The method takes the current game state (board) and the number of simulations conducted for the parent node (parent\_simulations). It first checks if this is the first simulation (when parent\_simulations is zero) and returns positive infinity, ensuring the initial exploration of nodes.

The method then calculates the UCT values for each possible move by iterating over the legal moves of the game. It creates a temporary game instance to simulate the effect of each move on the current state, generating a new board. The child simulations variable, initially set to zero, would typically represent the actual number of simulations conducted for the child node. If this count is zero, the UCT value is set to infinity to prioritize unexplored moves.

The UCT value is calculated as the sum of a heuristic evaluation of the new board state and an exploration term, which balances exploration and exploitation. The exploration term considers the ratio of simulations for the parent and child nodes, influencing the UCT value. Finally, the method selects the move with the highest UCT value, determining the optimal move for the current game state within the bounds of the UCT algorithm.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

**Part II: Algorithm Tournaments and Evaluation**

For this part, we altered our main.py to take different arguments through command line where if user inputs tournament, we run the tournament method instead of doing the part I.  The run\_tournament method conducts the round-robin tournament between 6 algorithms variations and runs 100 games for each pair. The results are printed at the end. The play\_game, get\_next\_move and print\_results methods are helper functions. These are the results, we couldn't get the appropriate results. The code wasn't taking two algorithms and running it against each other.

**Results**

**A screenshot of a computer program

Description automatically generated**

**Part III: Enhancements**

We implemented the second part where we used a simple evaluation function, we used for DLMM to run heuristic function for UCT as well. This heuristic function returns a value that shows the desirability of the state. The results show the output for basis UCT and UCT with heuristics for the same txt file with same number of simulations.

A screenshot of a computer

Description automatically generated

**Reference**

<https://gist.github.com/poke/6934842>

<https://github.com/Alfo5123/Connect4/blob/master/game.py>

<https://www.youtube.com/watch?v=gvlO_-Fdk9w&ab_channel=DennisRoof>

<https://github.com/MohithMarisetti/Connect-4-using-Depth-Limited-MiniMax-and-Alpha-Beta-pruning>

<https://www.youtube.com/watch?v=P7WQUBLKDmo>

<https://www.youtube.com/watch?v=l-hh51ncgDI>