Monte Carlo Tree Search Algorithm

Angel Salges
Department of Electrical and
Computer Engineering
Florida State University
Tallahassee, Florida
ads19n@fsu.edu

***Abstract*—In this project we aim to use the Monte Carlo Tree Search algorithm to act as an agent in a connect 4 game and give us the best possible moves for any state we give to it. We use some easy states to test the reliability which it passed and tried it with sets of data of different complexity of remaining amount of moves. It reliably gave possible moves to the states which we can infer they are good moves as it behaved in a positive way before.**

Keywords—Monte Carlo, MCTS, Connect 4, Artificial Intelligence

I. INTRODUCTION

The ever expanding field of AI is in constant need of finding new and more efficient ways to do things. This causes algorithms to be developed, studied and improved at a very quick pace. So it is essential to know the basics of algorithm development, which ones exists, what are they used for so we can understand what other uses we can give them. One of the many techniques that is used to test the efficiency of algorithms is to test it in board game settings as they are usually fairly simple games with clear parameters and little outside influence. This festers the perfect environment to test and train artificial intelligence.

Connect 4 is one of those games that researchers use to test the algorithms. It is based on a simple vertical grid of 6 rows by 7 columns in which players alternate playing their identical pieces on top of each other. The idea is to connect 4 of your pieces vertically, horizontally or diagonally before your opponent. With a perfect player, if they go first it is guaranteed that they always win. This makes it a good test for artificial intelligence to see if they can get close or do the perfect play.

Amongst the many algorithms used by researchers to test in board games there is the Monte Carlo Tree Search. It is a stochastic search algorithm that has been used efficiently in solving many other games by using its features of random exploration and strategic planning. In this paper we will discuss how does the Monte Carlo Tree Search Algorithm works, how was it implemented, the data that was used, what were the results and what did we learn from this experience.

II. BACKGROUND

A. Problem Description

The project is a program that solves the best move for you to take in a game of connect 4. It sees the state of the board, observes which pieces are placed and evaluates which is the best row to place your piece. The purpose of using the Monte Carlo tree search is to evaluate multiple possibilities down the line and observe the child nodes and predict a utility of each possible move we can make. Then we compare this utilities and choose the move with the best possible outcome for us.

B. Monte Carlo Tree Search Description

According to Tim Miller, a professor of artificial intelligence of the University of Queensland, The Monte Carlo Tree Search algorithm has few key components: Select, expand, simulate and backpropagate [4]. The foundation for this algorithm is a Markov Decision Process which can be represented by an Expectimax Tree.

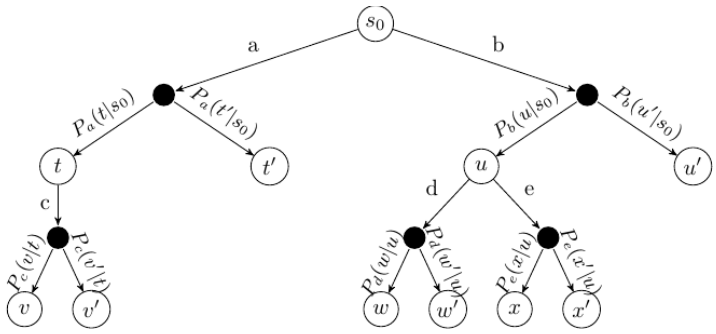

Figure 1:Expectimax Tree Example[4]

The first step in the Monte Carlo Tree Search is the selection. We start at the root node and find a node that has not yet been expanded, in this example, " *t'* ". We then move into the expansion in which we expand this node with an allowed action and create more nodes with the outcomes of this action, unless it's an end state. Then comes the simulation in which we choose one of the child nodes we expanded randomly and simulate it to an end state. Finally we end up with the backpropagation that gets a reward depending on the outcome of the final state and goes back the path it came from updating the expected values of each state along the path [4].

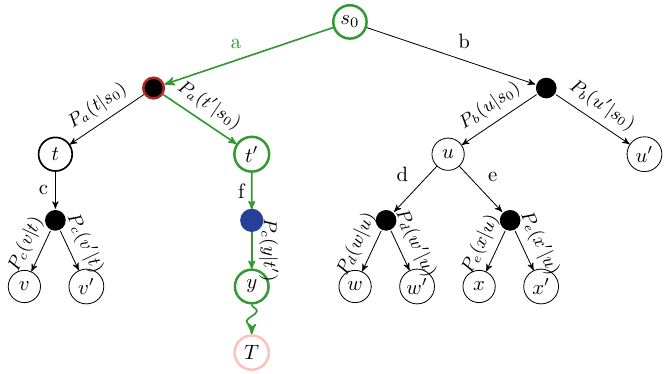

Figure 2: Expansion, simulation and backpropagation of node t' [4]

III. Project Description

The project has 3 main parts, the algorithm file called MCTS.py, code that takes the input and gives best move called main.py, and a text file with the test data. The algorithm file has the theoretical implementation of the Monte Carlo Tree search algorithm [2] in which it does the process mentioned in the previous section with the state for each node being the actual state of the board that is passed to the algorithm. The main.py file establishes the board we will be using for the algorithm as well as the interface we will be inputting our states in and a function to convert data from game trials of other connect 4 games to a format we can use for our project. The data file includes the already converted data to our format as well as some of the examples of raw data that came in the files. We will be using test cases that I designed, as well as part from the Test_L3_R1, Test_L2_R2 from Pons [1].

A. *MCTS.py*

First, we will need imports such as random, time, math, json and os.path. This file sets hyper parameters of max actions, limiting memory and time limit to prevent expansions of every nodes as this can be counterproductive as it can go for an incredibly long time depending on the amount of moves that can be played. This file includes 2 main classes we will be using in the project UCT and Stats. Stats contains information about the values for each node and the nodes that have already been visited inside a dictionary. The UCT class is the main class in the file. The objects of this class contain the board, history, player to move and the max depth as well as limits from the hyper parameters. It uses the update function to set the current state of the board and search for possible actions with the get_action method while calling the simulate method to expand nodes to terminal node. Then it evaluates all possible actions with the function evaluate_actions method with the rewards assigned by the simulation. It also has a save and load functionality that we will not be using as that is intended for continuous usage of the simulation and we will just be doing by state. Lastly there is a child class UCT_ver1 that we will call in our main function to process our board.

B. *Main.py*

First, we import the UCT_ver1 class and deepcopy function. We then set some hyperparameters such as the rewards, length and height of the board and search directions for the board to find winning moves. It has a Board class we will make our object from that has the board and the player to play. We will always use "X" as the player to play for convenience. It has a possible_actions method to search for the first empty space of each column in the board and list them as possible moves. It also has a next state function that shows the search tree what state comes after each move for evaluation purposes. It also has a compacting and un-compacting methods to make the board smaller and format it to process by the search tree. It also has the check_winning function to assign rewards to end states and feed it back to the search tree. The processInput function takes user input in. The input has to be a 42 length string that contains only the characters 'X','O' and '0'. The first character represents us and what the algorithm will try to make moves for. The second character represents our opponent that we have to prevent from winning. The third character represent empty spaces. The 42 characters represent the 42 squares in which pieces can be played. Starting from the top of the board. The first pieces that can legally be played are the last 7 characters. We can see in the following figure a description of each character in order of input and where would they be placed.

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42

Figure 3: Indexing of input pieces

With the given input we first validate that the characters are only 'X','O' or '0' and remove any white spaces between. We then create a 2D-array that we will feed into the board by starting a counter and iterating through the 2D-array and placing the data into it. We will then start the board validation. We will check that there are no floating pieces, as it is a vertical game, by checking that any 'X' and 'O' do not have a '0' below them. Then, it checks that it is not a winning state, as there are no legal moves if a player has already won. If all is good, we create a board object with the data and create an AI player that will give us our next move with the board state. The action is then stored into an array for later use and the player is deleted as when testing multiple cases the history attribute of the Monte Carlo Search Tree implementation will accumulate previous moves and become inaccurate.

Lastly we have the convertData function. This is done to handle a history of moves of a game. We get a sequence of numbers without spaces that represent the columns in which a piece has been placed. We can change who starts by changing the ch variable to a '0' for and 'X' start or vice-versa. It takes the sequence and then maps it to a board format and prints it. This format we can use in our previously made processInput() function.

C. *Data.txt*

This file contains the test data, the list of resulting moves and a few examples of raw data. The test data is going to be first some edge cases to test for invalid inputs,, the starting move and the winning move. Then I will include test data from Pons [1] that has been classified by difficulty based on remaining moves

for end game. The classification is L3_R1 with 28 < moves and remaining < 14. L2_R2 14 < moves <= 28 14 <= remaining < 2. The subsequent tests get progressively harder. I will be using 10 cases from each of the data sets and recording the results. The examples for raw data will be the first 5 of the L3_R1 data set. The raw data is the order of moves in a game. We will start with us making the first move for demonstration but the code works as well with us making the second move. Although it sets us in a impossible path to victory as the second move always losses if played perfectly.

IV. Results

Figure 4: Example result of invalid input

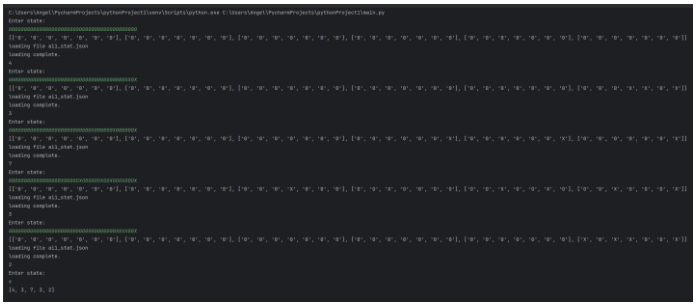

Figure 5:Results from testing best plays (direct wins)

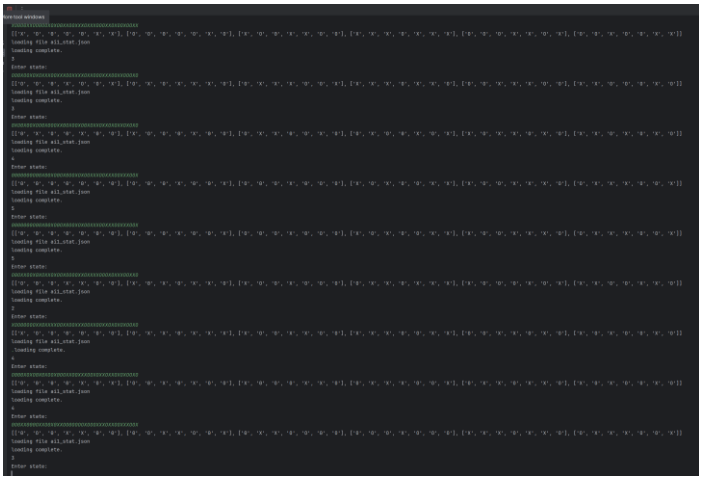

Figure 6: Results from data set Test_L3_R1

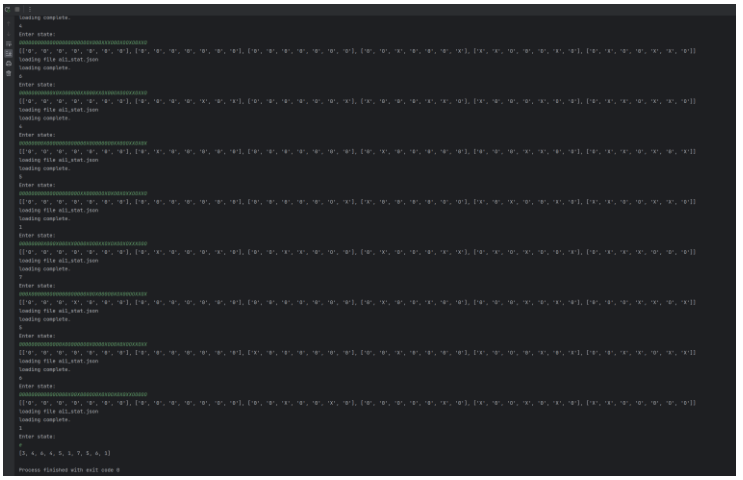

Figure 7:Results data set Test_L2_R2

Discussion

I tried to measure the quality of the moves that the algorithm gave. In the guaranteed best move cases it went for the win. When the board was empty it always went for the middle. This is objectively the best play as it basically guarantees a win if played right. It also prevents the opponent from winning if the following move for them is a winning move. I varied the inputs as start cases where the board was set up in a particular game state. The algorithm gave us reliable moves we could use to prevent losing and aim for wins. It is hard to know which exactly the best move in each situation is, but the algorithm does a reliable job in giving moves that at the very least keep the match going for every move if it does not make you win. The input data has very little impact in terms of performance. Having more pieces on the board does not seem to affect that much in how the process last. More tests should be done in comparison with other algorithms to see the performance and see if their answers are similar and that could indicate that they are on the best possible plays.

Experience

The biggest issue I encountered was finding an implementation of the algorithm that was usable. Most of the algorithm implementation do not have documentation nor comments to be modified. I would have chosen a more common algorithm or system to study as I would have more reference and would ease the workload and research requirements. Also, having python experience would have helped immensely as I would not have to deal with syntax and typecasting errors that made my experience really difficult. I would create more code on a different algorithm to see how it goes against the MCTS. I would compare the answers and make them play games in different starting orders to see who reliably wins the most. This could help me understand the difference in efficiency and implementation as there are easier algorithms to implement too.

I learned in this class about search algorithms the most but did not see how they are implemented and that is what I would like to keep researching. I also learned about what an AI takes into consideration and what are effective ways to design one to fit our needs. With this project I learned a great amount about

the Monte Carlo Tree Search Algorithm and how it works. It gave me great detail into stochastic models and how it uses probability to make decisions inside a tree which made it really interesting as I had only seen binary trees without probability incorporated into them for decision making. I also learned a lot of python while developing this project. I learned what tuples and dictionaries are, I learned how to process data and format it to our needs and improved my skills on how to do object oriented programming. It also improved my research skills as I found a vast amount of information although most of it was not useful for my particular case.

REFERENCES

[1] [1] P. Pons, “Part 2 – benchmarking solvers,” Solving Connect 4: how to build a perfect AI, http://blog.gamesolver.org/solving-connect-four/02-test-protocol/ (accessed Dec. 8, 2023).

[2] Somewheref, “SOMEWHEREF/Connect4_MCTS: An implementation of Connect4 agent using MCTS(Monte Carlo Tree Search),” GitHub, https://github.com/Somewheref/Connect4_MCTS/ (accessed Dec. 8, 2023).

[3] T. Cazenave, "Monte Carlo Beam Search," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 68-72, March 2012, doi: 10.1109/TCIAIG.2011.2180723.

[4] T. Miller, “Monte-Carlo Tree Search (MCTS),” Monte-Carlo Tree Search (MCTS) Introduction to Reinforcement Learning, https://gibberblot.github.io/rl-notes/single-agent/mcts.html (accessed Nov. 1, 2023).J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.