Monte Carlo Tree Search project report

Xiang Zheng	Jiaao Wu	Zihao Cheng
21307110169	21307130203	21307130080
mango789	Julius Woo	football prince

October 28, 2023

Abstract

In this report, we delve into the application of MCTS algorithm for developing a proficient *ConnectX* game-playing agent. Initially, we will give a brief introduction to MCTS plus *ConnectX* and explain the theoretical framework of MCTS, breaking it down into its four core components. Following this, we will propose our refinement strategies that have substantially improved our agent's performance, assisting it in achieving a high score. We will then present the results of our agent's performance in *Kaggle* competitions to demonstrate its effectiveness. In conclusion, we will reflect on the limitations encountered during the development of our agent and explore potential avenues for future improvements.

1 Introduction

Monte Carlo Tree Search (MCTS) is a versatile algorithm that has gained significant popularity in the field of artificial intelligence, particularly for games and decision-making processes. It excels in scenarios where the search space is rather large and complex, and can be even applied to brand-new games in which there is no prior experience to define an evaluation function. The core idea is to estimate the value of a state as the average utility over a number of simulations of complete games starting from the state.
Connect Four is a competitive two-player game where participants alternately drop their colored checkers from the top into a grid consisting of seven columns and six rows. Each checker falls straight down, occupying the lowest available position within the selected column. The primary goal for each player is to be the first to form a horizontal, vertical, or diagonal line of four consecutive checkers of their color. If the grid is completely filled without any player achieving this, the game ends in a draw.

2 Framework

MCTS evaluates a state's value by conducting simulations or playouts, progressively building a search tree to represent the state space of the problem. The MCTS algorithm can be broken down into the following four stages, as Figure 1 illustrates:

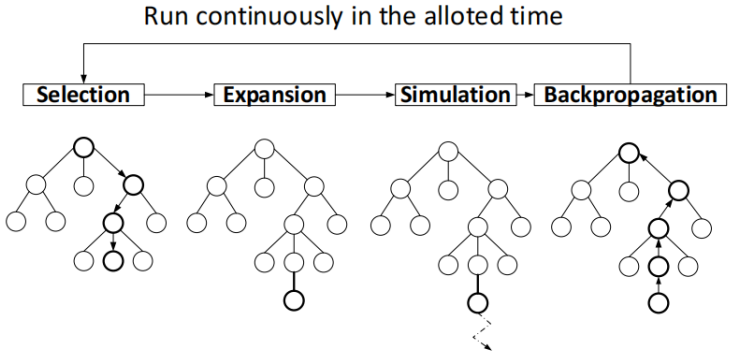

Figure 1: Monte Carlo Search Phases

2.1 Selection

The selection phase starts from the root and navigates down the tree to a leaf node based on a selection policy. The selection policy is designed to balance exploration and exploitation, ensuring that the algorithm both investigates states with few playouts, and refines the estimates of the states that outperform in past simulations.
A very effective selection policy is called "upper confidence bounds applied to trees" or **UCT**. The policy ranks each possible move based on a **UCT** formula called **UCB1**: [1]

$$\text{UCB1} = \frac{U(n)}{N(n)} + C\sqrt{\frac{2\log N(n.parent)}{N(n)}}$$

where $U(n)$ is the total score of the node n, $N(n)$ the number of playouts though n, and C is a constant. Our agent in *Connect Four* game adopts UCB1 as selection policy.

2.2 Expansion

Once a leaf node is reached during the selection phase, the expansion phase is initiated. In this stage, we grow the search tree by generating one or more new child nodes which represent potential future states of the game. In our program, we randomly choose one board column to drop the checker from all legal moves for the selected node.

2.3 Simulation

After expansion, a playout is performed from the newly added child node(s). Moves are chosen for all players involved based on a predefined playout policy, which might be random or biased in some way. The simulation continues until a terminal state, and the outcome of the game is recorded. The result provides an estimate of the newly expanded node's value. Our agent simply takes a random move action as the default playout policy.

2.4 Back-propagation

The final stage of MCTS is back-propagation. In this phase, the results of the simulation are used to update all the nodes visited during the selection phase, as well as the newly expanded node. The updates propagate back up the tree to the root, modifying the value estimates and win/loss counts at each node. This updated information will then be used

to guide future selections and improve the overall performance of the algorithm.

In summary, MCTS builds and refines a search tree through repeated iterations of selection, expansion, simulation, and back-propagation, aiming to accurately estimate the value of each state and ultimately make optimal decisions. These four steps are repeated either for a set number of iterations, or until the allotted time has expired.
In our implementation, we mainly focus on the time which the intelligent agent consumes every simulation as there is time limits. In view of this, we devise some refinement techniques to reduce time cost, and enhance our agent's behavior.

3 Refinement

3.1 Tunable number

Upon successfully implementing our initial agent for *ConnectX*, our immediate inclination was to fine-tune the parameter C within the UCB1 calculation. Previous studies of MCTS have suggested 1 as the optimal value for this parameter, hence, we established $C = 1$ as our baseline. Thereafter, we varied C within the range of $\frac{1}{\sqrt{2}}$ to $\sqrt{2}$ to empirically ascertain the optimal value. The performance of varying C values within *ConnectX* is shown in Figure 2.

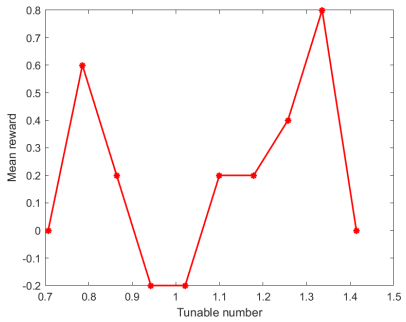

Figure 2: Results of different tunable numbers

From Figure 2 we can find that the agent with tunable number around 1 outperforms other agents, so we choose 1 as our optimal tunable number.

3.2 Scores v.s. Visited times

There are two different principles in the choice of best child when our agent finished its search in a given time: child with the highest score; child with the most visited times. Since we don't find a plausible theory to assess the performances of different principles, the method used in section3.1 is inherited here. With scores principle as the baseline, we display the experiment results of all versions in Figure 3.
Clearly scores principle is preferred in *ConnectX*, hence it is implemented in the final selection of best action.

3.3 Check way of winning state

It may seem trivial to check the winning state of a given board like Algorithm 1:

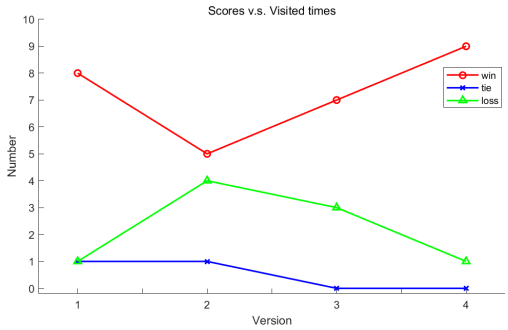

Figure 3: Scores v.s. Visited times results

Algorithm 1 Traditional check way of win state

```
for direction in directions do
  for row in rows do
    for col in cols do
      window ← board[col+i,row+j]
      if window.length=inarow then
        return true
      else
        return false
      end if
    end for
  end for
end for
```

However, the time complexity of this algorithm is $T = O(4 \cdot (\text{cols-inarow}) \cdot (\text{rows-inarow}))$ and the check function will be called numerous times in search process, leading to a waste of search time and an inefficient search agent. Can we make some improvements in this check function?
Notice that the state of board changes if and only if a new piece is put into the current board, so we can check the state of board the instant we put a new piece into the board. The time complexity of the check algorithm is only $O(4 \cdot \text{inarow})$, which saves a lot time for our agent. Therefore, our agent will be able to search deeper in a fixed given time, contributing to a better search result. The pseudocode of the algorithm is shown in Algorithm 2.
The performance of our agent improved dramatically from 900+ to 1900+ after the implementation of refined check way. Details of the algorithm can be found in the submitted *python* file.

3.4 Global variable

A property of global variable in *python* is to store the current information for later use. Since there is only one step difference between our current act and next act, the Monte Carlo Tree formed by our agent may be quite similar. Hence it is natural for us to use a global variable to store the current tree for reusing in next act to reduce time complexity. Besides, the action taken by our opponent can be easily found by checking the difference between the new board and old board. We set the action taken by our opponent as the root node and search from our opponent's perspective to retaliate our opponent's action.

Algorithm 2 Refined check way of win state

```
(row,col) ← piece
for direction in directions do
  (offsetrow,offsetcol) ← direction
  for i in 1 to inarow-1 do
    for j in 1 to inarow-1 do
      window ← board[row+offsetrow*i,col+offsetcol*j]
      if mark in window≠our_mark then
        return counted number
      end if
    end for
  end for
  return inarow-1
end for
```

The try-except structure is implemented in the initialization of our global variable. If the global variable doesn't exist, we need to initialize a class MCTS based on the current board, else we just need to get the child node of the existed global variable based on opponent's action.
The *Kaggle* score of our agent improved a little from 1950+ to 2000+ after the implementation of the global variable.

3.5 Allocation of OverageTime

When we were at our wits' end finding ways to optimize our agent and decided to give up, the parameters stated in **Final Project 1.pdf** caught our eyes. After a throughout view of the file, we found that we made a fatal mistake: the parameter remainingOverageTime is ingored by us! It means that our agent is given less time to search than our opponent and this mistake may be the only difference between our agent and the best one. Consider that more biases will be introduced to MCT as the search time increases, we just allocate 0.3s to each step taken by our agent.
The strategy is proofed correct as our *Kaggle* score improved from 2000+ to 2300+. However, the improvement don't meet our anticipation and it may due to the lower convergence rate of Monte Carlo Tree as the search times increase.

4 Experiment Results

The *Kaggle* scores of our different versions of agents are listed below.

Agent	Characteristic	Score
Version 1	Traditional agent	903.7
Version 2	Refined check way	2008.7
Version 3	Add global variable	2043.6
Version 4	Considering remainingOverageTime	2375.2

From the table, it is observed that by attributing refined features to our agent, we are able to achieve higher scores, thereby demonstrating the efficacy of our refinement techniques.

5 Discussion

5.1 Limitations

1. We have only assigned constant scores to winning or tying states, ignoring the importance of different terminal states.
2. The global variable used in finding the root node may affect the inherent randomness of MCTS.
3. Various refinement methods might negate each other, and we have not conducted a comprehensive comparison of these variables across all agent versions.

5.2 Future work

1. Utilizing the `numpy` and `numba` module could enhance the execution speed of our search. The inspiration for this approach stems from a specific *GitHub* repository [2].
2. Since MCTS constructs a highly selective tree, it may overlook crucial moves or fall into tactical traps. In contrast, full-width minmax search does not exhibit this weakness [3]. Employing a hybrid of MCTS and minmax search could potentially yield superior search outcomes.
3. In MCTS, we only expand the MCT and may then take some suboptimal nodes in the final action selection. Tree pruning may assist in deriving more accurate results [4].
4. The current use of a constant "tunable-number" for our UCB policy might lead to overestimation or underestimation of certain nodes. Implementing variants of UCB estimation, based on empirical findings within *ConnectX*, could prove beneficial [5].

References

[1] Russell, Stuart, and Peter Norvig. Artificial Intelligence: A Modern Approach. 4th ed., Pearson, 2020.

[2] https://github.com/mziele1/connectx_mcts

[3] H. Baier and M. H. M. Winands, "Monte-Carlo Tree Search and minimax hybrids," 2013 IEEE Conference on Computational Inteligence in Games (CIG), Niagara Falls, ON, Canada, 2013, pp. 1-8, doi: 10.1109/CIG.2013.6633630.

[4] Joris Duguépéroux, Ahmad Mazyad, Fabien Teytaud, Julien Dehos. Pruning playouts in Monte-Carlo Tree Search for the game of Havannah. The 9th International Conference on Computers and Games (CG2016), Jun 2016, Leiden, Netherlands. ffhal-01342347f

[5] Świechowski, M., Godlewski, K., Sawicki, B. et al. Monte Carlo Tree Search: a review of recent modifications and applications. Artif Intell Rev 56, 2497–2562 (2023). https://doi.org/10.1007/s10462-022-10228-y