Monte Carlo Tree Search (MCTS) Algorithm for Connect Four

***Abstract*—In this project we using MCTS for Connect 4. MCTS algorithm is based on Monte Carlo Tree simulation method, which goes through lots of simulation for current state and generate a game search tree. In order to avoid redundant calculations, we use the 'Tree Recycling' optimization strategy.**

***Index Terms*—MCTS, Tree Recycling, Connect 4**

I. Introduction

Connect Four is a classic two-player strategy game played on a 7×6 grid. The game's objective is simple yet challenging – players take turns to place their colored discs into any unfilled column, and the discs will fall to the lowest available position within the chosen column. The primary goal is to be the first to achieve a sequence of four of your own discs in a row, either horizontally, vertically, or diagonally. Connect Four's simplicity and competitiveness make it a favorite pastime for players of all ages. In this project, we delve into the implementation of the Monte Carlo Tree Search (MCTS) algorithm to explore the strategic aspects of this game and improve our understanding of MCTS as an AI technique.

II. Monte Carlo Tree Search

A. The advantages of using MCTS

I preferred using the Monte Carlo Tree Search algorithm because of its flexibility to solve games with large branching factors. The main bottleneck for games involving large action space (7 in case of connect4) is that it requires an extensive search considering the different permutation and combination of the given board. MCTS tries to overcome this problem in an efficient way as explained below by reducing the search space while at the same time maintaining efficiency.

B. Theory

MCTS requires a large number of simulation to bulid up a large simulation tree with n nodes with each node annotated with the win count and the visit count in order to get more accurate to its reward. It starts with a single root node and performs iterations as long as resources are not exhausted. MCTS consists of four main steps: Selection, Expansion, Simulation and Backpropagation.

a) Selection: In this phase, the agent starts from the root node, selects the most urgent node, performs the selected operation and continues until it reaches the end state. To select the most urgent node the upper confidence limit of the node needs to be used. The node with the largest UCB will be used as the next node. The UCB process helps to overcome the exploration and exploitation dilemma. For bandit problems, it is useful to know the upper confidence bound (UCB) that any given arm will be optimal. The simplest UCB policy proposed by Auer et al[1] is called UCB1, which has an expected logarithmic growth of regret uniformly over n (not just asymptotically) without any prior knowledge regarding the reward distributions (which have to have their support in [0, 1]). The policy dictates to play arm j that maximises:

$$UCB_1 = \overline{X}_j + \sqrt{\frac{2\ln n}{n_j}}. \tag{1}$$

b) Expansion: When UCB can no longer be applied to find the next node, the game tree is expanded further to include an unexplored child by appending all possible nodes from the leaf node.

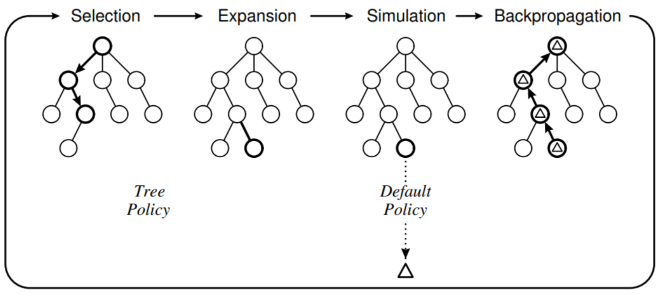


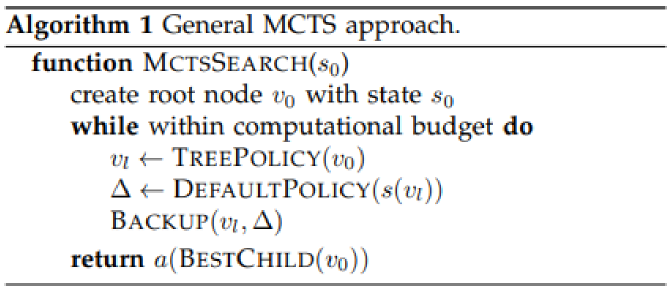
Fig. 1. The basic process of MCTS

c) Simulation: Once expanded the algorithm selects the child node either randomly or with a policy until it reaches the final stage of the game.

d) Backpropagation: When the agent reaches the final state of the game with a winner, all the traversed nodes are updated. The visit and win score for each node is updated.

C. *MCTS Summary*

The process of each iteration can be summarized as: 1) Each iteration starts at the root.
2) Follows tree policy to reach a leaf node.
3) Node N is added.
4) Perform a random rollout.
5) Value backpropagated up the tree.



Algorithm 1 General MCTS approach.

```
function MCTSSEARCH($s_0$)
    create root node $v_0$ with state $s_0$
    while within computational budget do
        $v_l \leftarrow$ TREEPOLICY($v_0$)
        $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
        BACKUP($v_l, \Delta$)
    return $a$(BESTCHILD($v_0$))
```

Fig. 2. MCTS Algorithm

As can be seen from Fig.2., there are two important policies: 1) Tree policy: From the existing search tree, either select an existing leaf node or generate a new one.(Selection and expansion) 2) Default Policy: Starting from a specified non-ending state, play through the domain to generate an estimated value.(Simulation)

III. Algorithm Implementation

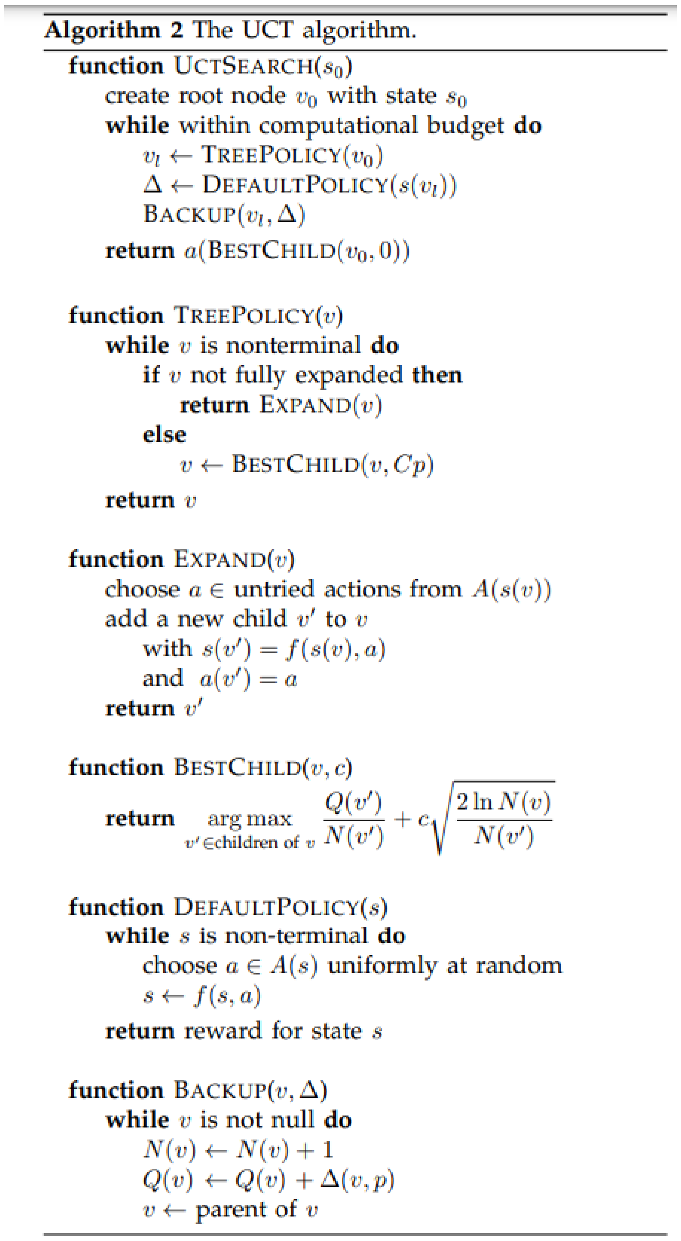
A. *Helper Functions*

To apply this algorithm to the game, in addition to the main part of the MCTS algorithm, we also need to define some helper functions. We defined some functions to update the board state, check for a winner, determine if there's a draw, and calculate the UCB value, among others.

In order to check if a player has won the game after a move, we use four types of patterns/checks to determine if the player has won: Horizontal (h), Vertical (v), Positive diagonal (p), and Negative diagonal (n). Depending on the type of check, the function will iterate over rows and/or columns and slice a window of the size configuration.inarow. After the function get the window, it checks the window to see if the player has all their marks and no other marks in it. If any of these checks results in a win, the function returns True. If none of the checks indicate a win, it returns False (i.e., is_win which is initialized to False).

B. *MCTS*

In this project, we implemented a MCTS structure proposed by Browne et al[2]. We use the UCT(Upper Confidence Bound 1 applied to Trees) strategy, and the algorithm is presented in Figure.2. We defined a Node class to represent a state node in the search tree, and it contains some important attributes, including the game board, player, parent node, score and so on. Some methods are also contained with the 'Node' class, including some functions to help the processes of expand, simulate and backpropagate.



Algorithm 2 The UCT algorithm.

```
function UCTSEARCH($s_0$)
    create root node $v_0$ with state $s_0$
    while within computational budget do
        $v_l \leftarrow$ TREEPOLICY($v_0$)
        $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
        BACKUP($v_l, \Delta$)
    return $a$(BESTCHILD($v_0, 0$))

function TREEPOLICY($v$)
    while $v$ is nonterminal do
        if $v$ not fully expanded then
            return EXPAND($v$)
        else
            $v \leftarrow$ BESTCHILD($v, Cp$)
    return $v$

function EXPAND($v$)
    choose $a \in$ untried actions from $A(s(v))$
    add a new child $v'$ to $v$
        with $s(v') = f(s(v), a)$
        and $a(v') = a$
    return $v'$

function BESTCHILD($v, c$)
    return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

function DEFAULTPOLICY($s$)
    while $s$ is non-terminal do
        choose $a \in A(s)$ uniformly at random
        $s \leftarrow f(s, a)$
    return reward for state $s$

function BACKUP($v, \Delta$)
    while $v$ is not null do
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta(v, p)$
        $v \leftarrow$ parent of $v$
```

Fig. 3. UCT Algorithm

C. *Optimization*

We use 'Tree Recycling' in our algorithm: by saving the tree between moves, the agent avoids redundant calculations and can continue its search deeper into the tree.

After every move, the agent retains its search tree and its current root node. Upon receiving a new game state after the opponent's move, the agent examines the new board state to determine which move the opponent chose. Based on the opponent's move, the agent locates the child node corresponding to that move and designates it as the new root node. All other non-relevant child nodes (and their subtrees) are discarded, allowing the agent to continue its search from this new root.

In our code, tree recycling is implemented by: Utilizing a global variable current_state to store the current search tree. When the agent receives a new observation, it first tries to locate the child node corresponding to the opponent's move and set it as the new current_state. If it can't find a corresponding child node (for instance, when a new game starts or for some reason the opponent's move can't be determined), the agent starts a brand-new search tree. Since the agent already has results from prior searches, it can continue a deeper search on this tree instead of starting anew.

This approach allows the agent to reuse and build upon its search tree across consecutive moves, enhancing its overall search efficiency.

Besides, in order to avoid exceed time which may lead to disqualification or forced random moves in many environments, we implemented 'Time Management'. Thus, the agent can keep track of time and ensures it doesn't exceed the limit.

We established a specific computational constraint based on the total game time and the time limit for each move. And we employs a while loop to repeatedly perform iterations of MCTS until the time limit is reached. During each iteration, the MCTS explores more game states and updates its estimates for each potential move. Within the while loop, the code continually checks if the elapsed time surpasses the time limit. As soon as this limit is breached, the loop terminates, halting further search by the MCTS. Subsequently, the algorithm chooses an action based on the current search results.

The strength of this time management approach is that it ensures that the MCTS doesn't exceed the designated time constraint, allowing it to select an appropriate action for the game within the allocated time. Through iterative deepening, the MCTS can explore as many game states as possible within a limited time frame, yielding a more comprehensive and accurate action recommendation.

IV. EXPERIMENT

A. Parameters

a) Exploration Parameter: This is the exploration parameter for the UCB1 formula. The UCB1 formula balances exploration (looking at unvisited or less-visited nodes) and exploitation (choosing nodes with high average rewards). The value of it determines the weight of the exploration term. A higher value encourages more exploration, while a lower value emphasizes exploitation. The choice of 1 means that both exploration and exploitation are equally important.

b) Reward: The value 1 is the score assigned to a winning outcome. The value 0.5 is the score assigned to a tied game. This score suggests that a tie is considered better than a loss but not as good as a win.

Awarding a tie a reward value between 0 and 1 (such as 0.5) signifies that it's better than a loss but not as good as a win. This can motivate the learning algorithm to aim for a tie when it cannot secure a win, rather than easily conceding a loss. Losing a match carries a penalty (i.e., the reward is -1). This means, from a reward perspective, that losing isn't equivalent to taking no action at all. This encourages the algorithm to explore various strategies and try to avoid a penalty, knowing that the worst-case scenario is a penalty, rather than a tie.

B. Result

We first run multiple episodes to estimate its performance. We let our agent play against Kaggle's Negamax Agent for 25 rounds each time and record the results to preliminarily determine our parameter settings. The results can be seen in Table 1.

From the results, we can see that the choice of reward values can greatly influence the learning process. If the rewards aren't set appropriately, it may lead to suboptimal learning. Based on our preliminary simple experiments, we determined some appropriate reward values. Then, we made submissions on Kaggle, adjusted the explore_parameter, and modified the code structure and stopping conditions. According to the evaluation results, the performance was best when the explore_parameter was set to 1, and there was a significant improvement in performance after introducing tree recycling.

V. CONCLUSION

A. Weeknesses

a) High computational cost: MCTS requires a large amount of computational resources in the search tree construction and simulation phases, especially in complex games and decision problems. This may limit its real-time performance.

b) Randomness: The random nature of MCTS may lead to unstable performance, especially during the initial search. This may require extensive simulations to improve robustness. Need for tuning: The performance of MCTS is highly dependent on the tuning of parameters, including exploration parameters and termination conditions. Inappropriate parameter settings can lead to performance degradation.

B. Possible ways to improve the performance of the MCTS algorithm

a) Heuristic Functions: Use smarter heuristic functions to guide the search, especially when selecting nodes. This can help improve the performance of MCTS by reducing the number of unnecessary simulations.

b) Move Ordering: In our code, When expanding a node, a random unexplored move is selected. By leveraging domain-specific knowledge, we can prioritize more promising moves. For instance, in Connect X, the central columns are often more valuable due to more winning opportunities. So, expanding them first might lead to more efficient tree searches.

c) Iterative Deepening: Instead of running MCTS for a fixed duration, consider an iterative deepening approach where we start with a shallow search and gradually deepen it. This approach can help in situations where an immediate, strong move is detected early.

In this report, we implemented MCTS, after several experiments, we recognized that the performance of MCTS is greatly influenced by the reward settings. And the observed performance is greatly improved after applying the tree recycling strategy and removing the limit on the number of

reward(win,tie,loss)	explore_parameter	My Agent vs Negamax Agent
(1,0,-10)	1	0.7
(1,0,-5)	1	0.8
(1,0,-1)	1	1.0
(1,0,0)	1	1.0
(1,0.5,0)	1	1.0
(1,0.5,0)	2	1.0

TABLE I
RESULT.

simulations, this may indicates that it is important to guarantee the search can explore deeper into the game tree and examine more branches.

REFERENCES

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3):235–256, may 2002.

[2] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.