Midterm Project Report

Group 21
資工 17 102012072 林永昕
電資 17 102030015 廖穎毅
電資 16 101060020 劉冠佑

MCTS Algorithm Introduction:

The key concept of MCTS algorithm is simple: According to the results of simulated playouts, we can build a search tree node by node and use it to determine which move is the best for us to play. It is based on random sampling and mainly aims at solving problems with large search spaces. More importantly, this method is domain-independent, which means that MCTS could be applied to different games without having to modify the algorithm according to game specific knowledge, so we can implement multiple games with a single MCTS class. The algorithm includes a UCT formula and four stages.

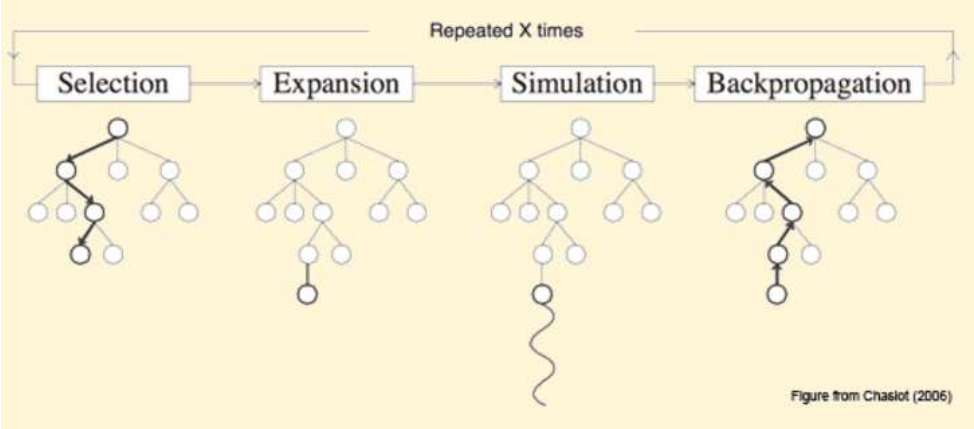
UCT:

From the view point of a tree, we want to select child nodes from the current root by finding a balance between the exploitation of known rewards vs the exploration of unvisited nodes in the tree. The reward estimate for a node is based on a random simulation from that node. Before the move can be seen as reliable enough to play from, each node must be visited a certain amount of times. We choose the node which has the highest score according to the following formula:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\log t}{n_i}}$$

Where:
wi - the number of wins for the current node
ni - the number of visits for the current node
c - the exploration parameter which is usually chosen empirically
t - the total number of simulations for the nodes considered, i.e. the total sum of all ni

Four Stages:



1. Selection

Starting at root node R, recursively select optimal child nodes until a leaf node L is reached.

The selection stage is where the UCT formula is used. The move is selected by the use of the UCT formula rather than randomly, and applied to the next position to be considered. The selection process would then continue until a child is reached where no statistics have yet been generated for the children.

2. Expansion

If L is a not a terminal node (i.e. it does not end the game) then create one or more child nodes and select one C.

The expansion stage begins after Selection has finished. During the expansion phase, of the currently unplayed children a move is randomly selected and played.

3. Simulation

Run a simulated playout from C until a result is achieved.

The simulation stage begins after a random child node has been selected from expansion. The simulation phase works by continually playing with itself on a copy of the current board state until a winner has been found.

4. Backpropagation

Update the current move sequence with the simulation result.

The backpropagation stage begins after the simulation stage has finished. The board should now have a winner, loser or a draw. This result now needs to reflect in the tree, to show the path that has been taken to get to this node. The back propagation stage is done recursively to make its way back up the tree and score the nodes with the result that has been obtained from the simulation on this node.

Implementation:

Language: Python

Structure:

2 classes – Gomoku : Maintain the board status and implement game actions

Node: Implement the nodes in the search tree of MCTS

2 global function – UCT() : Implement 4 stages of MCTS and return the optimal move

UCTPlayGame() : Control the game flow and print the board

Game Flow:

Our Gomoku AI program starts from calling the function “UCTPlayGame()”. In this function, first we create a Gomoku object called “state”, and we will use this object to access the game properties such as board coordination, which player’s turn, the steps that have been taken…etc, in addition, we also use this object to call methods that implement the actions in game, like play move, regret move or check if AI or player has already won after each move. Then, we print out a message to ask the player to choose if he/she would like to go first or second. The game officially starts after the player has chosen. If player goes first, we print out a message to ask the board coordination that he/she wants to play. On the contrary, if AI goes first, we make it play [H,7] which is the center of the board as AI’s first move. Whenever either side has finished their move, we give the turn to the other, and the game keep this flow until a winner comes out.

The things we do is determined by the variable ”playerJustMoved”. If it equals to 2, which means AI just moved and it is now the player’s turn, then we simply ask player to input his/her move, calling the function “DoMove()” to renew the board by adding player’s stone to the position he/she inputted, moreover, we also append the move to a list “steps”, so that each move can be recorded. On the other hand, if the variable ”playerJustMoved” is equal to 1, which means player just moved and it is now the AI’s turn, then the tasks we have to deal with are much more complicated, and this is the main part of this project, the most effort we have paid and the most time we have spent on. We have to run a well-designed algorithm, i.e. MCTS plus other supported algorithms in this project, in order to decide an optimal move to play. In the algorithm, we observe the situation of the board first, to see if there is a move that AI MUST play. In other words, AI would immediately win or obviously increasing the chance of winning if it plays that move. Or, AI would immediately lose or obviously going to be in great danger if do not play that move. If no necessary move exists, we call the function “UCT()” to run the MCTS algorithm and return a best move for AI to play.

MCTS Implementation:

Our MCTS algorithm is constructed by two parts. The first part is a class “Node”. It is the data structure of nodes in the search tree. In our Gomoku project, a Node object represents a move. The other part is a global function “UCT()”. We write the whole MCTS process, that is, 4 stages: Selection, Expansion, Simulation and Backpropagation, in this function. It passes the Gomoku object “state” and the number of iteration as parameter, so we can adjust the MCTS iterations by giving different parameter. UCT() returns an optimal move determined by MCTS algorithm. As a result, although we have done lots of things in this function, we can easily obtain the result we want by simply calling this function, so this is a very domain-independent. We can apply this algorithm into any game project without difficulty.

In the class “Node”, it maintains a parent node, list of child nodes, wins, visits, move(move that got us to this node), untriedMoves(future child nodes), playerJustMoved, and state. There are three methods in this class. The first one is “UCTSelectChild()”. It is called in Selection stage and there is only one thing to be done in this method. It sorts all the children of the given node by the evaluated score according to the UCT formula, and return the highest one. The second method is “AddChild()”. It is called in Expansion stage. First its creates a new Node object, and then removes the move passed by parameter from the untriedMoves, and at last appends the newly created node to the lists of childNodes. The third method is “Update()”. It is called in Backpropagation stage. Whenever it is called, visits plus 1 ,and wins plus the result returned from GetResult(). GetResult() returns 1 if winner is playerJustMoved, 0.5 if no winner, 0 if winner is the opponent of playerJustMoved.

In the function “UCT()”, we first assign the Node created by rootstate to an object “rootnode”, and then use for loop to iterate the 4 stages of MCTS for the number of times given. In the for loop, first we assign rootnode to a variable “node” and assign “rootstate” to a variable “state”. Then, we can start from Selection stage, we call UCTSelectChild() to get a optimal node according to UCT formula. However, it will not executed until nodes are fully expanded, so at the beginning the process will start from Expansion. In Expansion stage, we randomly choose a move from untriedMoves, calling DoMove() for that chosen move to renew board state and call AddChild() by passing chosen move and new state as parameter. Next, the process enters Simulation stage, keeps randomly playout from one of all possible moves at this state until a winner is appeared. At last, in the Backpropagation stage, we trace back through the path from the terminal node to the root, and call Update() to count the visits and wins. Finally, after the iterations of these 4 stages finished, then we sort all the children of root by the number of visits, returning the highest child as optimal move.