Monte-Carlo Tree Search for Gomoku

Zepeng Ding
School of Data Science
18307110088@fudan.edu.cn

Yuze Ge
School of Data Science
19307130176@fudan.edu.cn

Abstract

In this project, we implemented the Gomoku AI with Monte-Carlo tree search, which requires a large number of simulation and builds up a large search tree. We use two kinds of MCTS algorithms, Heuristic Monte Carlo Tree Search (HMCTS) and Upper Confidence bounds for Tree (UCT). In the end, the two AI got around 1200 rating under Bayesian Elo, of which HMCTS performed slightly better than UCT. However, these two AIs are not as good as the minimax search AI in the previous project. And we shared some of our views on this.

1 Introduction

Monte Carlo Tree Search (MCTS) is a classic search algorithm for finding optimal decisions by taking Monte Carlo simulations in the decision space and generating a game search tree according to the results. It has a long history within the numerical algorithms and significant successes in various AI games, such as Alpha-Go. As the number of simulations increases and new nodes are expanded, the MCTS estimate of the true value becomes more and more accurate. The basic framework of MCTS is shown in Figure 1, which consists of four main steps: Selection, Expansion, Simulation and Backpropagation.

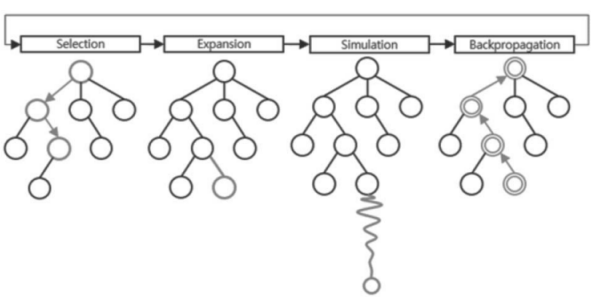

Figure 1: The basic process of MCTS.

- **Selection:** Starting from the root, successive child nodes are selected until the leaf node is reached. The root is the current state of the game. The move to be used will be chosen by the evaluation algorithm (perhaps with heuristic information) and applied to obtain the next position to be considered.
- **Expansion:** Unless a leaf node ends the game decisively (e.g., win/lose/tie), create one (or more) child nodes and select nodes from among them.
- **Simulation:** If the node has not been simulated, then do a typical Monte Carlo simulation for Gomoku. Otherwise. Generate a random chld node for the leaf node and run the simulation until the result is decided. In our code, we call the function ”rollout”[3].
- **Backpropagation:** Use the results of the rollup to update Update information about the nodes on the path from the leaf node to the root Information about the nodes on the path to the root node (in general, 0 means losing and 1 means winning). Add the number of access times for each node in the path.

Organization for the following report:

- **Section 2** Heuristic Monte Carlo Tree Search
- **Section 3** Upper Confidence bounds for Tree
- **Section 4** Code Implementation
- **Section 5** Experimental Results
- **Section 6** Conclusion and Future Work

2 Heuristic Monte Carlo Tree Search

In this paper, we present two kinds of MCTS algorithms. One is called Heuristic Monte Carlo Tree

Algorithm 1 MCTS with Heuristic Knowledge

Input: initial state s_0
Output: action a corresponding to the highest value of MCTS;
add Heuristic Knowledge;
obtain possible action moves M from state $s0$;

for each move m in moves M **do**
  reward $r_{total} \leftarrow 0$
  while simulation times<assigned times **do**
    reward $s \leftarrow$ SIMULATION$(s(m))$
    $r_{total} \leftarrow r_{total} + r$
    simulation times add one
  end while
  add (m, r_{total}) into $data$
end for
return action BEST$(data)$

function SIMULATION(state s_t)
  if s_t is win **and** s_t is terminal **then**
    return 1.0
  else if s_t is draw **and** s_t is terminal **then**
    return 0.5
  else
    return 0.0
  end if
  if s_t satisfied Heuristic Knowledge **then**
    obtain forced action a_f
    new state $s_{t+1} \leftarrow f(s_t, a_f)$
  else choose random action a_r from untried actions
    new state $s_{t+1} \leftarrow f(s_t, a_r)$
  end if
  return SIMULATION(s_{t+1})
end function

function BEST$(data)$
  return action a
  //the maximum r_{total} of m from data
end function

Search (HMCTS), and the other is called Upper Confidence bounds for Tree (UCT). The HMCTS for Gomoku is presented in Algorithm 1.

Here f is a function to generate a new board state from last board state and action. Heuristic knowledge which is common knowledge for Gomoku players[2] can save more time in simulation than random sampling. Therefore, it helps the result getting converge earlier than before. Details about the heuristic knowledge can be found in the next section.

We let the bonus 1 when the final result is a win and 0 when the final result is a loss. Bonus 0.5 (tie) rarely happens. Then, the Q value of an action can represent the expected reward for that action[3].

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} l_i(s,a) r_i$$

$N(s,a)$ is the number of times that action a has been selected from state s, $N(s)$ is the number of times that a game has been played out from s, r_i is the result of the $i-th$ simulation played out from s, and $l_i(s,a)$ is 1 if action a is selected on the $i-th$ playout from s or 0 otherwise. In our algorithm, $N(s,a)$ is a constant 85 for all action a. We select the action with the biggest Q(s,a) as out next move.

3 Upper Confidence bounds for Tree

The other widely used MCTS algorithm is UCT, which is based on Upper Confidence Bounds (UCB). UCB can balance the intention between exploration and exploitation. Our AI select the child node with the biggest UCB value in each step. The UCB is defined as follows:

$$UCB = Q_i + C \times \sqrt{\frac{\ln N_i}{n_i}}$$

Where Q_i is the average winning rate from i-th simulation defined above, N_i is the number of visits of the father of node i, and n_i is the number of visits of node i. And C is a constant value determining the trade-off between exploration and exploitation, and we set it to $\sqrt{2}$ in our algorithm.

Different from HMCTS in which we simulate each possible action the same times, UCT simulate possible actions unequally. In the formula, the first part Q_i represents the average return of the node, and the second part $C \times \sqrt{\frac{\ln N}{n_i}}$ is large if the node is rarely visited. So UCB can balance the exploration and exploitation and find out suitable leaf nodes earlier. The UCT for Gomoku is presented in Algorithm 2.

Algorithm 2 UCT for Gomoku

Input: create root node v_0 with state s_0
simulation time limit T
Output: action a corresponding to the highest value of UCT;

$t \leftarrow 0$
while $t \leq$ T **do**
  $v_l \leftarrow$ TREE-POLICY(v_0)
  Policy$\leftarrow$ Heuristic Knowledge
  $M \leftarrow$ GET-ACTION(v_l)
  EXPAND(v_l, M)
  for action a in M **do**
    reward $r \leftarrow$ Simulation$(s(v_l))$
    BACK-UPDATE$(f(s(v_l), a), r)$
  end for
  $t \leftarrow t + 1$
end while
return action a(BEST-CHILD(v_0))

function TREE-POLICY(node v)
  while v is not a leaf node **do**
    $v \leftarrow$ BEST-CHILD$(v, \sqrt{2})$
  end while
  return v // the leaf node
end function

function EXPAND(node v,actions M)
  for action a in M **do**
    add a new child v' to v, with $s(v') \leftarrow f(s(v), a)$ and $a(v') \leftarrow a$
  end for
end function

function GET-ACTION(node v)
  list $A \leftarrow$ all possible actions from node v **return** action list A
end function

function BEST-CHILD(node v, parameter c)
  return $\arg\max_{v' \in child} UCB(v)$
end function

function SIMULATION(state s)
  while s is not terminal **do**
    if s satisfied heuristic knowledge **then**
      obtain forced action a
    else choose random action $a \in A(s)$ randomly
    end if
    $s \leftarrow f(s, a)$
  end while
  return reward for state s
end function

function BACK-UPDATE(node v, reward r)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow 1 - \frac{1}{N(v)} + \frac{r}{N(v)}$
    $v \leftarrow$ parent of v
  end while
end function

v indicates a node which has four pieces of data[3]: the state $s(v)$, the next action $a(v)$, the average simulation reward $Q(v)$, the visited count $N(v)$. And v_0 is the root node corresponding to state s_0, v_l is the leaf node, the result of the overall search $a(Best - Child(v_0))$ is the action a that leads to the best child of the root node v_0. Different from HMCTS, the reward is 1 if we win, -1 if we lose and 0 if we draw.

In the algorithm, we first get the leaf node according to the tree policy. Then we expand the leaf node with possible actions. We simulate each child of the leaf node and get their rewards. Finally we do a back-update for each child. After T times of this method, we select the action with the biggest $Q(s_0, a)$ value as our next action.

4 Details in implementation

4.1 Pattern Storage and Dynamically Update the Board

We used the same method as in the last project to store the information of each position on the board and dynamically update the board.

We created a 20 $\times$ 20 information board. Each position on the board stores information about the pieces distribution at that position, such as the color of the piece, the number of its neighbors, and the pattern. Among them, the colors of the pieces are white, black, empty, and out of bounds, which are represented by the numbers 0, 1, 2, and 3 respectively.

In fig2, the number of neighbors (with a distance of no more than one square) in the position shown by the red box is 2, and the pattern in the horizontal direction is a live four (if the piece is white).

Whenever we make move or regret pieces, which may happen in the game or simulation, we will update the piece information of the neighbors at the target location immediately. In fig3, when the color of the piece at the intersection of the four red lines is

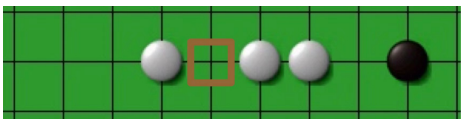

Figure 2: Each position stores information about the pieces distribution

updated, the information of the pieces on the red line will be updated.

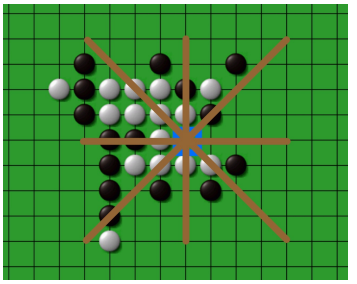

Figure 3: Dynamically update the neighbor's gomoku pattern

In this way, we don't have to create a new board for each move, which saves storage space and speeds up the algorithm.

In addition, our method of obtaining the pattern is the same as in the last project. We scan the neighbors of the target position, and obtain a 16-bit binary number through bit operation. Each 16-bit binary number corresponds to a pattern, so we only need to look it up in the pattern table to get the pattern of the piece.

4.2 Node Expansion

In HMCTS, we directly expand the root node. In UCT, we first find a leaf node based on UCB, and then expand the leaf node. Due to the excessive number of successor nodes, we adopted some strategies to select the successor nodes that need to be expanded.

If the root node or leaf node has reached the winning state, no node expansion is performed, and the reward value is directly returned.

Otherwise, we first find all positions where the number of neighbors is greater than 0 and the piece is empty, and then use the evaluation function for each position to get the evaluation value, and sort them in ascending order. Select the first M nodes for expansion. In our algorithm $M = 9$.

The evaluation function will comprehensively consider the pattern of a certain position. For example, the evaluation function will give a pattern with higher priority higher scores, and a must-win or must-lost pattern (such as live 4 or five win) will definitely get the highest score; Positions with priority patterns in multiple directions will get higher scores. In addition, the evaluation function adopts an offensive-biased strategy: the score of a same pattern on our side is slightly higher than that of the opponent's side.

4.3 Simulation Process

In order to control the time of the algorithm, we need to control the number of simulations to prevent the algorithm from overtime. We have adopted some methods in HMCTS and UCT.

4.3.1 Simulation location selection

If we always randomly select positions during the simulation, then meeting the victory conditions often requires a lot of steps. We adopted some strategies[1] to reduce the number of simulations required to meet the victory conditions.

- If four-in-a-row occurred in our side, the player will be forced to move its piece to the position where it can emerge five-in-a-row in our side.
- If four-in-a-row occurred in opponent's side, the player will be forced to move its piece to the position where it can block five-in-a-row in the opponent's side.
- If three-in-a-row occurred in our side, the player will be forced to move its piece to the position where it can emerge four-in-a-row in our side.
- If three-in-a-row occurred in opponent's side, the player will be forced to move its piece to the position where it can block four-in-a-row in the opponent's side.
- If a certain position can form a live 3 pattern, then this position will have a great advantage for a certain side. However, this advantage is not as great as the advantage in the above case, so we make a piece move in this position with a probability p. Usually p is greater than 0.5.
- If there is no special pattern, we will randomly select a position from the candidate positions for simulation.

4.3.2 Limit on the number of recursions

Even if the above strategies are used, It's still hard to reach the winning condition. For this reason, we limit the number of recursions to no more

than N times. In our algorithm, N is between 15 and 20. If the number of recursions reaches N times, it indicates that the winner has not yet been divided, and the algorithm will return a reward value slightly larger than the tie reward.

5 Experimental results

We used HMCTS,UCT and minimax search with α-β pruning AI to compete with 6 AIs on the GomoCup website. Each competition uses 3 kinds of fixed openings, alternate starting order, 6 games in total.We set the thinking time for each step not to exceed 15s, and the entire game time not to exceed 90s. The result is as follows:

Agent	Minimax	HMCTS	UCT
YIXIN17	0	0	0
WINE17	0	0	0
SPARKLE	3	1	0
NOESIS	5	2	2
FIVEROW	6	6	4
MUSHROOM	6	6	6

Table 1: Competition outcome

6 Conclusions and future work

In this project, we implemented Monte Carlo tree search using HMCTS and UCT. We continued the method of dynamically updating the chessboard in the last project, and adopted some strategies to reduce the number of recursions in the process of node expansion. In this way, we have increased the number of simulations as much as possible while the algorithm meets the time limit. Finally, HMCTS and UCT scored up to 1244 points, and HMCTS performance was slightly better than UCT.

In HMCTS, we perform approximately 100 simulations on each child node of the root, so that our AI can think in no more than 6 seconds per step. In UCT, in order to achieve the same thinking time, the total number of simulations is about 200, which is less than HMCTS. We think this is the main reason why UCT is not as good as HMCTS.

In addition, the UCB calculation formula we use in UCT may not be accurate enough. We adopted the original UCB formula $UCB = Q_i + C \times \sqrt{\frac{\ln N_i}{n_i}}$. In some literatures, in order to be suitable for Gomoku, the UCB formula often adds progressive bias like $\frac{H_i}{n_i+1}$ (H_i is the heuristic value)[1]. This may make our UCT fail to choose valuable leaf nodes.

MCTS often requires a lot of simulation to get relatively accurate results. Due to the limitations of the game rules, our AI does not have enough time to simulate, resulting in insufficient accuracy of the results. In addition, we need to balance the number of expansion nodes and the number of simulations each time. If the number of expansion nodes is small, we may miss key nodes. However, minimax search can use the evaluation function to directly select the successor nodes without simulation, which improves the efficiency of the search.

In the future, we plan to improve our MCTS from the following aspects:

- **Improve node expansion strategy** We will improve our evaluation function, not only consider the single pattern at the position, but also consider the combination of multiple patterns (such as double-live 3). To this end, we can refer to the successor selection strategy in the minimax search algorithm.
- **Improve UCB formula** We will use the UCB formula with progressive bias to strengthen the exploration of valuable nodes.
- **Special state storage** We will save some frequently encountered states, such as the state of the root node. This can reduce the number of dynamic updates and save time to increase the number of simulations. Zobrist Hashing may help.

References

[1] X. Cao and Y. Lin, "Uct-adp progressive bias algorithm for solving gomoku", in *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2019, pp. 1-7

[2] Z. Tang, D. Zhao, K. Shao, and L. Lv, "Adp with mcts algorithm for gomoku," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2016, pp. 1–7.

[3] H. J. K. Jun Hwan Kang, "Effective monte-carlo tree search strategies for gomoku ai," in *2016 International Science Press*. IEEE, 2016, pp.1–9.