Monte-Carlo Tree Search for Gomoku

Abstract—**In this project, we used Monte-Carlo Tree Search(MCTS) with Upper Confidence Bound(UCB) to implement an Agent for Gomoku. Due to the limitation of computation power and time, which are resources that MCTS depends on a lot, we adopted several enhancement strategies, both domain knowledge related and domain independent, to better implement our agents. In the end our agent got around 1070 rating under Bayesian Elo, however it can't match the performance of our previous agent based on minimax search.**

***Index Terms*—Tree Search, Monte-Carlo Method, Upper Confidence Bound, Gomoku**

I. Introduction

MCTS, which combines monte carlo methods with tree search, is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. In tree search, there's always the possibility that the current best action is actually not the most optimal action. In such cases, MCTS algorithm becomes useful as it continues to evaluate other alternatives during the learning phase by executing them, instead of the current perceived optimal strategy, which is known as the **exploration-exploitation trade-off**. It exploits the actions and strategies that is found to be the best till now but also must continue to explore the local space of alternative decisions and find out if they could replace the current best. UCT, tree search use Upper Confidence Bound, helps to balance the conflict between exploration and exploitation and find out the final result earlier. [3] In this project, we have also adopted the following methods to speed up the search speed and make the program run effectively: limitations on simulation times, heuristic knowledge before expanding and simulating, and dynamical update of the children of the board. Organization for the following report:

- Section 2 Monte-Carlo Tree Search
- Section 3 Implementation Details (representation, heuristic knowledge, dynamic update)
- Section 4 Experimental Results
- Section 5 Conclusion and Future Word

II. Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a method for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search. It has a long history within the numerical algorithms and significant successes in various AI games, such as AlphaGo [5].

To get more accurate estimate of the rewards, MCTS requires a large number of simulation to bulid up a large simulation tree [1]. The basic framework of MCTS consists of four main steps: Selection, Expansion, Simulation and Backpropagation, which is shown below.

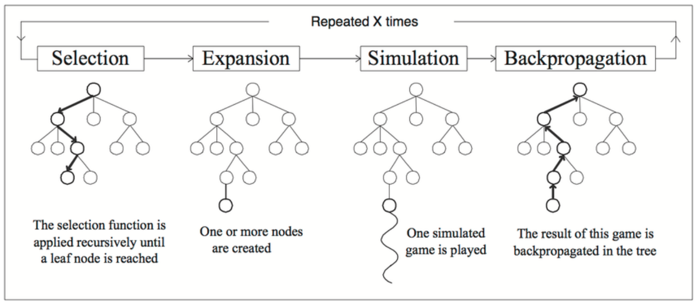

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Fig. 1. Basic Framework of MCTS

A. Selection

In this process, the algorithm will traverse the tree starting from the root, which is the current state of the game, with pre-defined $Tree\_Policy$, until it reaches the leave node (termination situations). It will use Upper Confidence Bound strategy to choose a child for fully expanded nodes, selecting the node with highest score getting from the following fomula:

$$USB = \bar{v}_i + C \times \sqrt{\frac{2 \ln N}{n_i}}$$

where $\bar{v}_i$ is the average rewards at state S_i, n_i is number of visits at state S_i, N is the total number of visits of the nodes at the same level (or in other words the number of visits at the parent of Si), C is a constant used for fine tuning, which we set $\sqrt{2}$ in our implementation.

B. Expansion

Unless a leaf node ends the game decisively, such as win, lose, or draw, this process will create one child node or more child nodes and select nodes from among them.

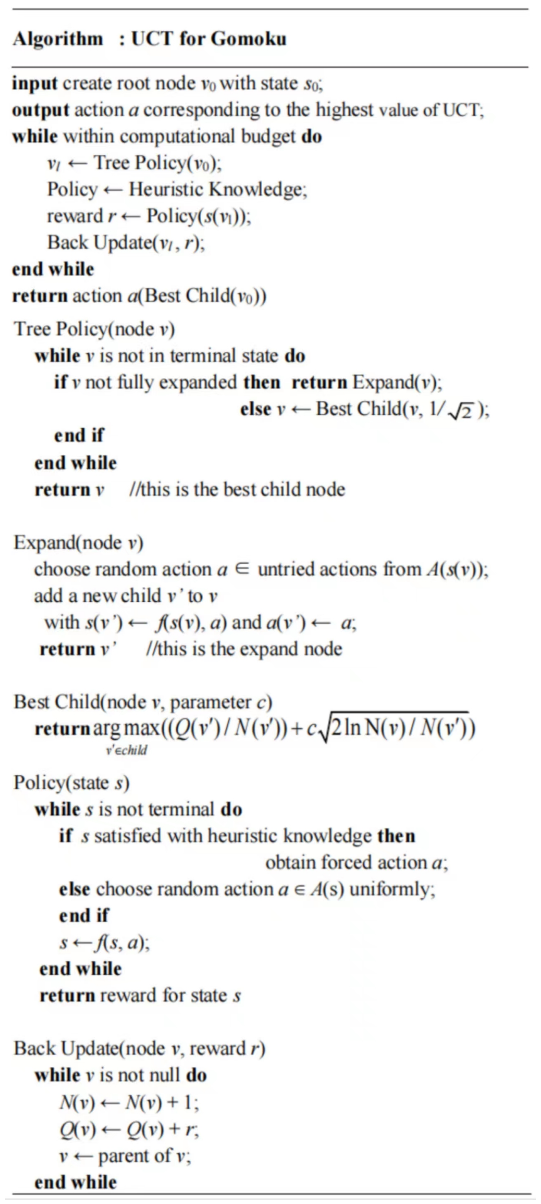
C. Simulation / Rollout

In this stage, the algorithm will run a simulation to produce a reward for this node. This process will select a child state randomly and then iterate until a terminal state is reached, at that point the value of the terminal stated is returned. Typically, -1 means losing, 1 means winning and 0 means drawing. We call this function $Rollout$ [4].

D. Back Propagation

During the process, the function gets the value (reward) of the $Rollout$ and updates the nodes from the start node of the $Rollout$ till the root node. The update consists of adding the reward to the current value of each node, and to increase by one the count of visits at each of these nodes.

The whole process of the algorithm is illustrated below.



```
Algorithm : UCT for Gomoku
input create root node $v_0$ with state $s_0$;
output action $a$ corresponding to the highest value of UCT;
while within computational budget do
    $v_l \leftarrow$ Tree Policy($v_0$);
    Policy $\leftarrow$ Heuristic Knowledge;
    reward $r \leftarrow$ Policy($s(v_l)$);
    Back Update($v_l$, $r$);
end while
return action $a$(Best Child($v_0$))

Tree Policy(node $v$)
  while $v$ is not in terminal state do
    if $v$ not fully expanded then return Expand($v$);
                    else $v \leftarrow$ Best Child($v$, $1/\sqrt{2}$);
    end if
  end while
  return $v$    //this is the best child node

Expand(node $v$)
  choose random action $a \in$ untried actions from $A(s(v))$;
  add a new child $v'$ to $v$
   with $s(v') \leftarrow f(s(v), a)$ and $a(v') \leftarrow a$;
  return $v'$    //this is the expand node

Best Child(node $v$, parameter $c$)
  return $\arg\max_{v' \in child}((Q(v')/N(v')) + c\sqrt{2\ln N(v)/N(v')})$

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

Back Update(node $v$, reward $r$)
  while $v$ is not null do
    $N(v) \leftarrow N(v) + 1$;
    $Q(v) \leftarrow Q(v) + r$;
    $v \leftarrow$ parent of $v$;
  end while
```

Fig. 2. Pseudo-code for MCTS with UCB (UCT)

The MCTS method we implemented is a structure called **UCT** mentioned in Zhentao Tang and his collaborators'work [3]. Note that MCTS with UCB (UCT) simulate possible actions unequally. In the formula of UCB, the first part represents the average return of the node, encouraging the exploitation of higher reward selection, and the second part is large if the node is rarely visited, encouraging the exploration of less visited choices. So UCB can balance the exploration and exploitation and find out suitable leaf nodes earlier.

III. Implementation Details

MCTS is required to be repeatedly carried out for enough times to ensure the prediction can be more accurate. And it's also time consuming since MCTS must spend a lot of time on searching some unnecessary feasible actions, like some possible positions that have little possibility to win if we take all empty positions on the board into consideration. So we adopted several strategies to enhance the implementation.

(representation, heuristic knowledge, dynamic update)

A. Representation of the board

We used the similar method as in the last project to repersent each state during the chess. We still re-represent the board (originally a 2-dimensional array) as a dictionary, with key being the hash value of each line and value being the string corresponding to the direction, however dropping the score calculated by the evaluation function in previous representation. So now the contents of each dictionary are {(hash value of this line):string of this line}.

What's more, we also remain the way we calculate the hash value for each string line. That is, for rows and columns, this is their corresponding row and column numbers. We use $x+y$ as the hash value for the index in the subdiagonal direction dictionary, and $width-1-x+y$ for the index in the primary diagonal dictionary.

For example, the first main diagonal of the board below will be represented in dictionary $diag\_1$ as {14:'000002101000000'} as a $StringBoard$ class.

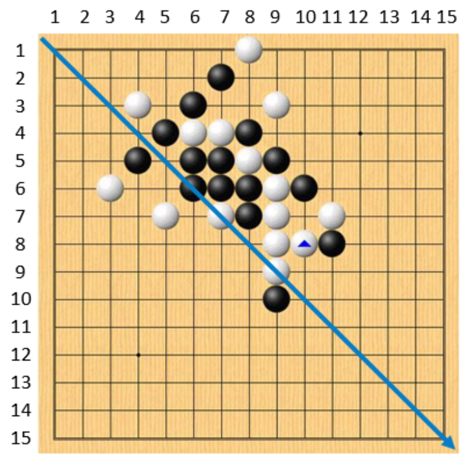


Fig. 3. Representation Example

B. Dynamic Update

This time, we set the class $StringBoard$ as a local variable rather than a global one since this won't help much. Instead, for each turn (each tree of MCTS), we will construct a variable $possible_moves$ at initial time, which will store all the possible next states for current state. This variable will be dynamically updated during one MCT search, decreasing our burden of searching for possible moves to a large extend and thus saving lots of time.

To be more specific, we will remain the possible moves we've select at initial time (which will be explained in detail later), and update these positions each time we reach a new state, that is, we select a position to place our stone. From expert knowledge for Gomoku, only the positions near the places that were already placed stones are better candidates, so we only add empty positions from 8 positions that are near the non-empty ones in to the variable.

For example, when the marked piece is newly placed during simulation, the positions on the blue line will be traversed, and the positions marked with blue stars will be added into $possible_moves$.

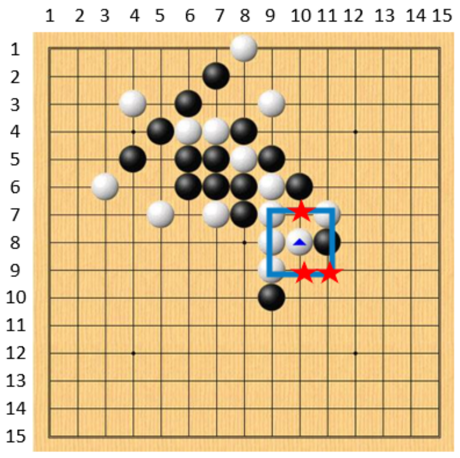


Fig. 4. Representation Example

C. Domain Knowledge

Even if we only consider the the positions that are around the current placed positions, the number of simulations sill still be too large for our agent to calculate fast enough each turn. So we use **Domain knowledge** in Gomoku to further select some better positions from $possible_moves$ to speed up the process.

1) Prior Search: At the initial time, we first check whether the positions near the ones that are already placed are empty or not. Before entering the MCT Search, our agent will check each move from $possible\_moves$ that whether it will form a live five (WIN) or live four (H4) pattern. If so, the agent will take the move directly because otherwise we will definitely lose or we will win. The specific order is:

1. Check whether our side will win
2. Check whether the other side will win (C4)
3. Check whether our side will form live 4 (H4)

If none of the special cases happen, we just go into typical MCTS.

2) Filter Moves: At the begining of MCTS, we will initial the root node with the current board situation. Before using $TreePolicy$ or going into $selection$ process, our agent will use $Filter\_move$ function to further reduce the number of $possible\_moves$, that is, it will first check whether any of the moves will form a special case with less priority than in $Prior\_Search$ but higher priority than others. The specific order is:

1. Check whether our side will form block 4 (C4) or live 3 (H3)
2. Check whether the other side will form live 3 (H3)
3. Check whether other special cases will form

If none of the special cases happen, we just keep the $possible_moves$ and let it go into the random process.

D. Domain Independent Strategy

1) Tree Policy: To better reduce the time of simulation, we find another strategy to keep decrease the number of possibly expanding nodes. During selecting process, we will decide to expand a half-expanded node either with UCB score, or select one of its children directly, both with probability 0.5, rather than keep every node in every level expanded totally, which is domain independent since it does not requires any knowledge about Gomoku. However, this is reasonable since we've already checked some special cases before we go into MCT search.

2) Limit on Number of Simulations: The larger number of simulations will generate more accurate rewards of each node for MCTS. However, limited by the computation power and time restriction, we can not afford too large a number of simulations for each turn. So we keep the maximum number of simulation as 100 and it balance the time and results best as in the current implementation.

IV. Experiment Results

In the matching, our AIs will compete with 12 AIs from Gomoku website, under 3 fixed openings with exchanges of hands. All players should limit the time of each step within 15 seconds and the entire game time should not exceed 90 seconds.

We have tested our agent several times and due to the randomness of MCTS algorithm, the Bayesian Elo score it got will fluctuate within a certain range. The highest score our agent can get up to around 1070. The results are listed as follow.

Notice that although the overall performance of MCTS agent is better than MUSHROOM, it still can not match the win:lose ratio of Minimax agent. This may be due to the limited computing power and time restrictions that bounded our maximum number of simulations. Since MCTS often requires a lot of simulation to get relatively accurate results, due to the restrictions of the game rules, our agent does

TABLE I
GOMOKU COMPETITION RESULTS

Opponent	**MCTS Agent**	**Minimax Agent**
YIXIN17	0:12	0:12
WINE	0:12	0:12
PELA17	0:12	0:12
ZETOR17	1:11	1:11
EULRING	1:11	1:11
SPARKLE	0:12	0:12
NOESIS	2:10	1:11
PISQ7	0:12	2:10
PUREROCKY	9:3	8:4
VALKYRIE	5:7	8:4
FIVEROW	4:8	10:2
MUSHROOM	12:0	12:0
Ratio	31%	43%

not have enough time to simulate enough times, resulting in insufficient accuracy of the results.

V. CONCLUSIONS AND FUTURE WORK

In this project, we implemented Gomoku Agent based on Monte-Carlo Tree Search with Upper Confidence Bound algorithm (UCT). Our representation of the board state, domain knowledge to select possible moves, and domain independent strategies all contributes to the performance of our Agent. In the future, we intend to improve our AI in the following fields:

A. Better Simulation Strategy

In our implementation, we only check the special cases before the MCTS and at the initial time of the MCTS. However, as mentioned in the zhentao Tang's work [3], we can use heuristic knowledge to reduce the possible moves at each simulation step. This may be more reasonable than randomly select the next node and thus giving more accurate results.

B. Inheritance of Search Tree

We have just set the representation of the board as a local variable, and update the board state at every simulation step. In fact, if the opponent plays a piece that has been modelled in our search tree before, we can just use this sub-tree directly as the new search state. Zobrist Hashing may be helpful.

REFERENCES

[1] C.B.Browne, E.Powley, D.Whitehouse, S.M.Lucas, P.I.Cowling, P.Rohlfshagen, S.Tavener, D.Perez, S.Samothrakis, and S.Colton. A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence Ai in Games, 4(1):1–43.

[2] L.V.Allis, H.J.Herik, and M.P.H.Huntjens. Go-moku and threat-space search. 1993.

[3] Zhentao Tang, Dongbin Zhao, Kun Shao, and Le Lv. Adp with mcts algorithm for gomoku. 2016 IEEE Symposium Series on Computational Intelligence (SSCI), IEEE, 2016:1–7

[4] H.J.K.Jun Hwan Kang, Effective montecarlo tree search strategies for gomoku ai, 2016 International Science Press. IEEE, 2016:1–9.

[5] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. Nature, 2016, 529(7587):484–489.