MCTs and rapid action value estimation in GOMOKU

Midterm-2 report for final project

Xuetian Chen 19307110189 Yifan Li 19307110499

November 21th 2021

Abstract

In this implementation, we use MCTs as the framework of search algorithm. Considering the 'exploration-exploitation trade-off', the variant UCTs of MCTs was used. In addition, we have tried several optimization algorithms such as UCT-RAVE balancing, heuristic prior knowledge, forced actions reduction and progressive strategy in order to improve the speed of the search algorithm and the accuracy of the estimation of node's utility value. The above improvements make the performance of our agent satisfactory.

1 Introduction

In the last report, we have implemented the Gomoku agent using Alpha-Beta pruning with Minimax search. But the search branching factors of board game is too high that the number of iteration layers is small. In addition, the evaluation function built by experience is difficult to ensure its accuracy. This time, we will use Monte Carlo Tree search to build our second Gomoku agent.
MCTs is a probabilistic and heuristic driven search algorithm. MCTs algorithm becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the current perceived optimal strategy. This is known as the 'exploration-exploitation trade-off'. It exploits the actions and strategies that is found to be the best till now, and continue to explore the local space of alternative decisions at the same time and find out if they could replace the current best.

2 Monte Carlo Tree Search

2.1 Four Main Stages

Monte Carlo Tree Search (MCTS) requires a large number of simulation (also called playouts) and builds up a search tree and the estimated value will converge to the true value. The basic process of MCTS is shown in Fig. 1 [2]. It consists of four main stages: Selection, Expansion, Simulation, and Backpropagation.

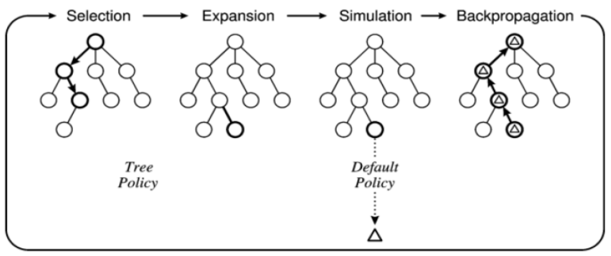


Figure 1: Four main stages of MCTS

We need to maintain a search tree whose root is the current board state. In the beginning of a playout, a child node A is selected. Then we choose a possible move in node A,

generating a child B and attach B to the current tree in the expansion stage. A simulation process will be taken in B and returns a value indicating who (black or white) won the game. Finally, the result will be back-propagated to its ancestor nodes in the path we selected.
Note that an appropriate method to select A with expanding B and simulate the result plays an important role in the efficiency of estimation and convergence. We call them as **Tree policy** (as a combination of selection policy and expanding policy) and **Default policy** (also called playout policy), respectively.

2.1.1 Selection

The selection policy should focus the computational resources on the important parts of the game tree. A simple idea is: choose the node that maximize the estimated winning probability. Assign a node with two values: number of playouts and number of victories. Then the estimate probability of an action a is:

$$\hat{p}(s,a) = \frac{Q(s,a)}{N(s,a)},$$

.

where $Q(s,a) = \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i(s)$ is the number of victories and $N(s,a) = \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a)$ is the number of playouts, both for state s, action a. $\mathbb{I}_i(s,a)$ is the indicator of i^{th} playout of s choosing action a, and $z_i(s)$ is the result $(0/1)$ of i^{th} playouts with respect to s: $z(s) = 1$ if the current term of player s wins, otherwise $z(s) = 0$.
Then the simple selecting policy is:

$$a_{\text{best}}(s) = \arg\max_a \hat{p}(s,a).$$

However, simply choose the action with highest estimated probability will typically avoid searching actions after one or more poor outcomes, even if there is significant uncertainty about the value of those actions. The excluded actions may be winning actions, given the true prior knowledge.
To avoid this, we need a selection policy that balances the two factors: **exploration** of states that have had few simulations and **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value. A solution, known as **UCT**, will be discussed in section 2.2.

2.1.2 Expansion

In this stage, we randomly choose an untried action a on the selected node s, initialize $Q(s,a) = N(s,a) = 0$, and attached its child node to the current search tree as a leaf to expand the tree. See section 3.2.1 about nonzero initialization.

2.1.3 simulation

In this stage, We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are not recorded in the search tree (that is, a complete MC simulation will only attach 1 node to the search tree). After the game is terminated (with a winner or with a draw), the playout terminates and return a flag of the winner.
Note that if we simply moves randomly, the number of playouts needed to guarantee convergence is large. And a best action in this sense is the best action if both players act randomly rather than rationally. Some heuristic prior knowledge will be discussed in section 3.2.2.

2.1.4 Back propagation

Use the return value in the simulating process and update both $N(s,a)$ and $Q(s,a)$ of the nodes in the selected path. The update is bottom-up, costing $O(1)$ time in each node and $O(d)$ time, $O(1)$ space totally, where d is the depth of the expanded child.
Note that d is smaller than tree height h, and even if 10,000 playouts are taken in the

root (which means the search tree contains 10,000 nodes in the end), the height h is still small (roughly less than 10), due to the high branching factor in Gomoku.

Going bottom-up from the leaf to the root, for each node we meets, its number of playouts added by 1, and if the rule of node (corresponding to current turn of the board) is the same as the winner, its number of victories added by 1, otherwise 0. That is:

$$N(s,a) \leftarrow N(s,a) + 1,$$
$$Q(s,a) \leftarrow Q(s,a) + \begin{cases} 1, & \text{if win} \\ 0, & \text{otherwise} \end{cases}.$$

3 UCT

As illustrated in section 2.1.1, the simple selecting policy

$$a_{\text{best}}(s) = \arg\max_{a} \hat{p}(s,a),$$

is inefficient, as discussed before. In order to handle the exploration-exploitation tradeoff, the method "Upper Confidence Bounds applied to Trees" (UCT) is introduced. The policy ranks each possible move based on an upper confidence bound formula called UCB1 [2]:

$$UCB1(s,a) = \frac{Q(s,a)}{N(s,a)} + C \times \sqrt{\frac{log\ N(s)}{N(s,a)}}.$$

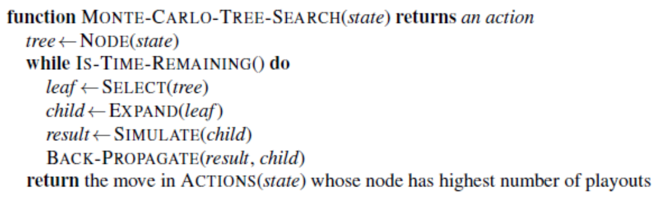
Here $Q(s,a)$ is defined as the number of victories. We find an action a that maximize the UCB1 value:

$$a_{\text{best}}(s) = \arg\max_{a} UCB1(s,a).$$

Note that the first term in UCB1 represents the exploitation value: the higher estimated winning probability action a has, the higher priority of a is. And the second term stands for exploration: If number of playouts in a is small, we should try a for its large uncertainty. With the number of playouts increases, the second term (exploration) term go down to 0 and we only consider the exploitation term.

In the above formula, C is a constant that balances exploitation and exploration, control how large the confidence interval is. There is a theoretical argument that C should be $\sqrt{2}$ [3], but in practice, we can try multiple values and choose the one performs best.

The UCT-MCTS algorithm is shown in Fig.2 [3]. Instead of selecting the action with highest $Q(s,a)$ in the end of MCTS, the one with highest number of playouts is selected. Intuitively, the reason can be explained in this way: The action with value 65/100 is better than the one with 2/3 because of less uncertainty. In fact, the UCB1 formula ensures that the node with the most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes large [3].



```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 2: The complete algorithm of UCT-MCTS

4 Optimization

4.1 UCT-RAVE Balancing

Instead of using the MC estimate $\frac{Q(s,a)}{N(s,a)}$ as the measurement of exploitation, RAVE algorithm uses the *all-moves-at-first (AMAF)* heuristic. Rapid action value estimation (RAVE) provides a simple way to share knowledge between related nodes in the search tree, resulting in a rapid, but biased estimate of the action values [4].

4.1.1 AMAF and RAVE

The AMAF heuristic, based on the assumption that the value of an action is often unaffected by actions played elsewhere on the board, assigns a general value for each action a played anywhere after the state s is reached. The formula of AMAF count and utility is:

$$\tilde{N}(s,a) = \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s,a),$$

$$\tilde{Q}(s,a) = \sum_{i=1}^{N(s)} \tilde{\mathbb{I}}_i(s,a) z_i(s).$$

Here $\tilde{\mathbb{I}}_i(s,a)$ is an indicator of action a is chosen in any time step **after** state s, in the i^{th} playout [4].
RAVE applies the AMAF into MCTS. Intuitively, it can be illustrated by Fig. 3 [4]. If we compute $Q(s,a)$ and $Q(s,b)$ in MC's way, we get $Q(s,a) = 0/2, Q(s,b) = 2/3$, indicating that we should take action b. However, AMAF's way use the information that a and b are taken in some successors of state s, under which a is better.

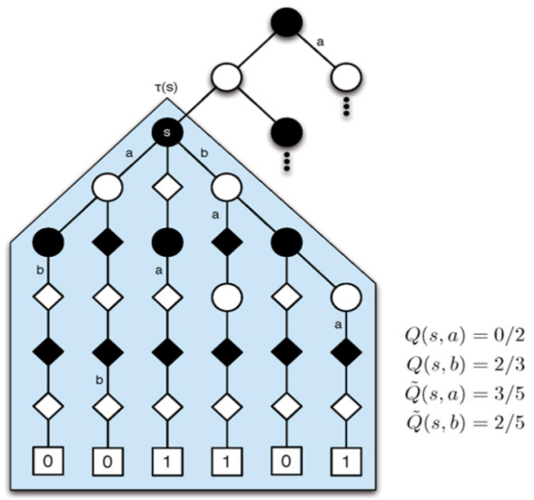


Figure 3: The explanation of RAVE (Q is defined as winning rate here.)

4.1.2 UCT-RAVE

As an incremental game, if an action a, taken exactly in state s or in successors of s, leads to a victory, they both mean that a is a key action to win. Therefore, the AMAF rule suits Gomoku in some sense. However, the value of actions often change after some enemy's actions. After one turn, the potential action may be more important, or even of no use, which means that the RAVE assumption may be wrong.
In order to combine the feature of fast convergence of AMAF and unbiased accuracy of MC, the UCT-RAVE can be used to balance the trade-off. The UCT-RAVE value is defined as the weighted sum of them:

$$Q_*(s,a) = (1-\beta(s,a))Q(s,a) + \beta(s,a)\tilde{Q}(s,a),$$

$$UCB1_*(s,a) = \frac{Q_*(s,a)}{N(s,a)} + C \times \sqrt{\frac{log\ N(s)}{N(s,a)}}.$$

where $\beta(s,a)$ is a balancing parameter for (s,a) [4].
In order to compute $Q_*(s,a)$, the count and utility in both MC and AMAF's way need to be recorded in nodes. When updating AMAF value from leaf to root, one node should update all actions taken after it, which can be implemented by maintaining a list of "history" actions (in fact, future actions) in $O(d)$ time (instead of $O(1)$ in MC) per node in the back-propagation process. Just little extra computational cost.

4.1.3 Balancing Factor

The balancing factor β should have these properties:

1. When $N(s,a)$ is small, we want to accelerate the information gathering, and AMAF value is weighted higher, indicating $\beta = 1$.

2. When $N(s,a)$ is large, the information is ample, and we need to guarantee the convergence to the true value, and MC value is weighted higher, indicating $\beta = 0$.

In our implementation, we use the following formula to determine a $\beta(s,a)$ that has this two properties:

$$\beta(s,a) = \max\left\{0, \frac{N_0 - N(s)}{N_0}\right\},$$

where N_0 is a hand-selected threshold according to the max number of simulation of the root, $N(s)$ is the number of playouts played in state s.

4.2 Heuristic Prior Knowledge

4.2.1 Expansion Heuristic

When expanding a new child node to the search tree, we randomly choose an untried action and initialize the $N(s,a) = Q(s,a) = 0$. The values are updated only after a playout is done. However, the results of playouts are random. An good action may make mistakes and get bad results. If we can add some prior knowledge to the expanded action, that is, initialize nonzero values, then update them as the same procedure, the effect of randomness can be reduced and the value can converge faster.

We redefine $Q(s,a)$ as the winning rate $Q(s,a) = \frac{1}{N(s,a)}\sum_{i=1}^{N(s)} \mathbb{I}_i(s,a)z_i(s)$. Another thing we need to change is the update formula, to the incremental form

$$Q(s,a) \leftarrow Q(s,a) + \frac{z(s) - Q(s,a)}{N(s,a)}.$$

Then we can use the board estimator implemented in minimax search to evaluate the expanded board. The evaluated value v can be transformed to a probability in $(0,1)$ by sigmoid function:

$$Q(s,a) = \frac{1}{1 + \mathrm{e}^{-(kv+b)}},$$

where we can hand-select coefficient k and b to fit our values' scale. When v is positive, the value is larger than 0.5 and finally goes to 1 as $v \to +\infty$.

4.2.2 Simulation Heurisitic

As discussed above, random simulation is not a good idea. We try to add heuristic into simulation. For every action in the simulation process, we evaluate the action score (discussed in our last report), if an action's score is higher than the scores of some patterns (such as five, four), this action will be taken immediately. This method can reduce the effect by randomness and accelerate convergence.

However, evaluating a score is much slower than generating a random number. If adding heuristic in this way, the max number of simulations will go down to less than 20 (compared to 2000 without heuristic), which can not get a good estimation. So we reject this idea.

4.3 Forced Actions Reduction

It is another way to add heuristic. We put the heuristic out of main MC procedure. When we need to consider an action, before entering MC procedure, we check (by evaluating single actions) if there are some forced actions to take, such as five and four. If there is, immediately take the most emergent action and skip MCTS. Note that the emergency means both offensive and defensive actions. This heuristic saves much time from preventing taking useless simulations in some special cases.

4.4 Progressive Strategy

When a node is visited only a few times, its utility value has not converged. At this time, the selection of actions will be better than the selection of policies. However, when a node is visited enough times, its utility value is more reliable. At this time, the selection of the policy will be more accurate than the selection of action. We hope to use these two selection methods at the same time, and perform transition from one to the other gradually. Therefore, we have implemented the following two strategies: progressive bias and progressive unpruning.[1]

4.4.1 Progressive Bias

The purpose of progressive bias is to guide the selection strategy at the beginning of the search by using heuristic knowledge, but when the search times increases, we hope this influence will gradually become smaller in order to convergence to the policy strategy. With this idea, we modified UCB fomula:

$$UCB(s,a) = \frac{Q(s,a)}{N(s,a)} + C \times \sqrt{\frac{logN(s)}{N(s,a)}} + f(a)$$

In implement, $f(a) = \frac{H(a)}{N(s,a)+1}$,where $H(a)$ is a heuristic value associated with the chessboard after action a.
In the selection of $H(a)$, we have made two attempts: one is to calculate weighted scores according to the chess types of both players of the current chessboard; Another is calculate the single point score, equals to the total player score minus opponent score in eight directions on the new chess piece after action a (The specific implementation details were shown in the previous report). Since calculate heuristic value of chessboard takes more time, we choose the latter method to calculate the single point score. The revised UCB has the following properties:

1. If the node has not been explored, selected it.

2. If the node has been simulated only a few times, and the heuristic value $H(a)$ is high enough, then $f(a)$ is dominant.

3. When the simulation times increases, both the utility value and the heuristic value have an balanced impact on the selection strategy.

4. If the node has been simulated many times, the previous simulations have a greater impact on selection (because $f(a)$ decreases by $O(\frac{1}{N(s,a)})$). At this stage, the selection strategy is close to the origin UCT.

4.4.2 Progressive unpruning

In the previous strategy, although we chose the heuristic function with less time consumption, the time consumption can not be underestimated when the number of simulations increases. For Gomoku, with the increase of playing rounds, the number of chess increases, and the branching factor of each simulation also increases significantly. In this regard, we use the pruning idea to filter the child nodes and retain the part with the highest heuristic value. In addition, unprun child node when there is some time left.
The implementation idea is to gradually unprun child nodes when the visit times of the node reaches a certain threshold.

References

[1] Chaslot, G., Winands, M.H., Herik, H.J., Uiterwijk, J., & Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation, 04, 343-357.

[2] Tang, Z., Zhao, D., Shao, K., & Lv, L. (2016). ADP with MCTS algorithm for Gomoku. 2016 IEEE Symposium Series on Computational Intelligence (SSCI), 1-7.

[3] Russell, S.J., & Norvig, P. (1995). Artificial Intelligence: A Modern Approach.

[4] Gelly, S., & Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer Go. Artif. Intell., 175, 1856-1875.