FINAL REPORT FOR GOMOKU

Wen Deng
School of Data Science
Fudan University
17307130171

Hongou Liu
School of Data Science
Fudan University
17307130204

January 12, 2020

1 Introduction

In the second part of the final project, my teammate and I mainly accomplished the following things.

- First, we fulfilled our mid-term-versioned Agent
- Secondly we tried two other methods MCTS and VCX(VCT + VCF) for Gomoku agent.
- At last, we combined all the agents we have now and see if we could come to a final-versioned one that could outperform our previous ones.

In the latter part, we will introduce our work in detail.

2 MCTS: Monte Carlo Tree Search Method

2.1 Main Thoughts and Improvements

Monte carlo simulation is to let two players randomly move from a specific game until the game achieves to the end. The victory is contingent. But if you let thousands of this game being played, then the statistics of the result can still give the game's inherent winning rate and the best winning move.

The following is the pseudo code of MCTS method, which is from the paper IEEE 2016 ADP with MCTS algorithm for Gomoku.

- **Input** original state s0
- **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 ← 0;
   while simulation times < assigned times do
      reward r ← Simulation(s(m));
      r_total ← r_total + r;
      simulation times add one;
   end while
   add (m, r_total) into data;
   end for each
return action Best(data)
```

```
Simulation(state s_t)
   if (s_t is win and s_t is terminal) then return 1.0;
                                       else return 0.0;
   end if
   if (s_t satisfied with Heuristic Knowledge)
     then obtain forced action a_f;
          new state s_{t+1} ← f(s_t, a_f);
     else choose random action a_r ∈ untried actions;
          new state s_{t+1} ← f(s_t, a_r);
   end if
   return Simulation(s_{t+1})

Best(data)
   return action a   //the maximum r_total of m from data
```

In our game, each time we are given a board contains our AI's and opponent's previous pieces, and the goal of us is to run MCTS algorithm to simulate the game and choose the action that is the most likely to win.

In this process, Monte carlo tree search expands the scale of game tree step by step through iteration. UCT tree grows asymmetrically and its growth order is unpredictable. The direction of the extension based on the performance of the child node, which is the UCB value. It means that in the search process, we should not only make full use of the existing knowledge to give more opportunities to nodes with high probability of winning, but also consider exploring those brother nodes with low probability of winning. Such a balance between Exploitation and Exploration is reflected in the definition of UCT method selection function, that is, the UCB of the child node $N_i's$ is computed by:

$$UCB = \frac{W_i}{N_i} + \sqrt{\frac{C \ln N}{N_i}}$$

where W_i is the number of wins of child nodes.

N_i is the number of times of child nodes participate in simulation.

N is the number of times of the current nodes participate in simulation.

C is the weighting coefficient.

It can be seen that the UCB formula consists of two parts, the former part is the use of existing knowledge, and the latter part is the exploration of incomplete simulation nodes. If C is small, the it prefers for utilization; If C is big, then it will attaches great importance to exploration. Parameters need to be set experimentally to control the number of visits to the node and the threshold value of the extended node.

It can be seen later that, when the code is actually written, the current node refers not to the specific move, but to the current chess game, and its children are the specific moves. It is bound to participate in the simulation that each child node participates in, so N is equal to the sum of the times that all its children participate in the simulation. When C is 1.96, the confidence of the confidence interval reaches 95%, which is also the value actually selected.

Monte carlo tree search (MCTS) expands only the nodes calculated according to the UCB formula, and USES an automated way to search more for nodes with good performance metrics. Specific steps are summarized as follows:

- Establish the root node from the current situation, generate all the child nodes of the root node, and simulate the game respectively;
- Starting from the root node, search for the best first;
- Calculate the UCB value of each child node by using the UCB formula, and select the maximum child node;
- If this node is not a leaf node, take this node as the root node and repeat step 2;
- Until the leaf node is encountered, if the leaf node has not been simulated before, simulate the game for this leaf node; Otherwise, child nodes are randomly generated for this leaf node and simulated game is performed.
- Update this node and ancestor nodes at all levels according to the corresponding color for the benefit of simulation game (generally, 1 and 0), and increase the visits of all nodes above this node;
- Return to 2, unless the search time of this round ends or the preset number of cycles is reached;
- Select the one with the highest average return from the child nodes of the current situation to give the best approach.

Therefore, it can be seen that the UCT algorithm is to continuously complete the process from the root node to a certain leaf node according to the guidance of UCB within the set time. The basic flow of the algorithm includes four parts: Selection, Expansion, Simulation and Backpropagation.

Another significant advantage of UCT tree search is that the search can be ended at any time and the results returned. At each moment, there is a relatively optimal result for the UCT tree.

Even now the agent still performs very bad in 20×20 board, for it takes two many time to simulate, and if we consider all the action that is available in the board, it very hard to simulate so many situations. So an intuitive improvement is to only consider the actions in the neighborhood(which means we only consider the available positions that is within a radius from the positions that already have a piece).

Another important improvement is to reduce the situations that need to simulate, because if our AI already has a pattern that contains four pieces, or opponent already has a pattern that already has four pieces, we should directly makes the move to win or block the opponent's four.

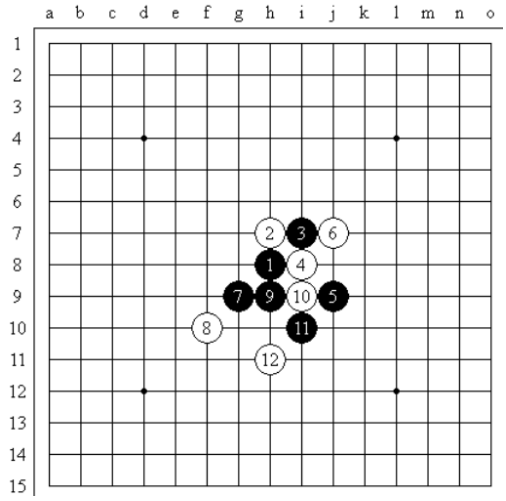
2.2 Results and Analysis

After I finished the improvement, the agent performs better than the original one, but the agent is still unable to beat most of the AIs that provided by the website(can only beat mushroom). After check every block of code, I found that the simulation takes a relatively long time, and I set the total time constraint to be 7.0 seconds. The agent can only simulate several hundred times in average. This is much less than our expectation. But the simulation process contains a lot of deepcopy uses, which lead to the time loss. But to optimalize it is not necessary, because even we optimalize the run time we can't guarantee the performance. Though the MCTS's idea is powerful and we have learned a lot from it, but our abilities and time is limited. So the main idea of our team is to use mini-max method as our final version.

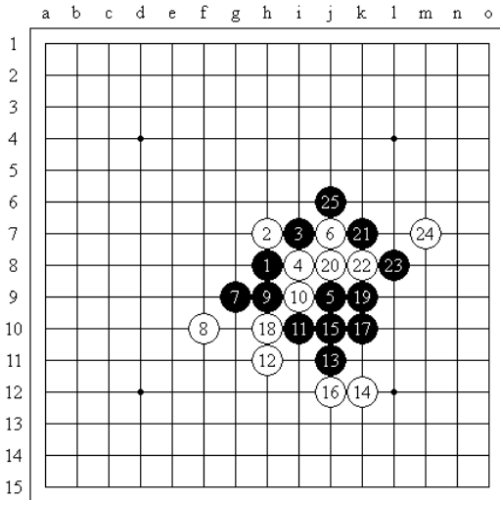
3 Kill2Win(VCX)

The main idea of VCX is simple. In our term, we want to find a series of "killing moves"(S-FOUR and Live THREE) that our opponent will have to block and see if this chain of moves could lead to some circumstance that the opponent could not defend. Actually, it is just like a special kind of mini-max search, however, could search much deeper as each parent Node only have very few children Node(typically 3 or 4). Generally, we could search for up to 10-layers within acceptable cost of time.

A simple example:



(a)



(b)

Problem:

- As this is a DFS, it would always return the deepest target-node, hence is not optimal. It actually takes the longest path of all available to kill our opponent. And what could be worse is that after we set a time limit to the search, the killing moves in lower layers could be cast away if it hit the time limit while strugling to search for a "bad killing move"in deeper layers.
- Still not efficient and fast enough, hence could not support deeper search – 12 layers or more.

Solution:

- The first problem could be easily solved by implementing an iterative deepening algorithm which would allow us to return an action found in the lowest layer.
- As for the second problem, we did not come up with many good ideas. What we do is to add more knowledge to the winning state detection. For example, for some certain kinds of situations we could directly determine we would win, such as when we have more than one S-FOUR, or we have one S-FOUR and a live THREE but our opponent has no S-FOUR or live FOUR. This could let us detect the ending state two or three layers earlier.

4 Improve Previous Minimax

4.1 Dynamic Board Evaluation

As we said in our mid-term report, to avoid a lot of time waste caused by repeated computation, we plan to change the evaluation function into a dynamic one, which means we store the information of the board and with each action made, we update the information in real-time.

To realize this target, we add a variable $self.count$ to keep the pattern record of the board, and every time we take an action, we update the information of the record, instead of computing it again. And when the action is canceled(that is the search backtracks), we make the variable revert to the previous value by maintaining a temp variable each step.

Then we designed a function $dynamic\_evaluate$ to update the count, and rewrite the evaluation method in the leaf nodes of mini-max search by converting the count of each pattern to score(For this we redefined a $getScore$ function to make the evaluation more reasonable).

4.2 Improve the Heuristic Function for Single Steps

In our real experiments, we found the heuristic function before is too aggressive that ignores many occasion that may need very far-sighted defence. So we rewrite the heuristic function by using the dynamic evaluation function defined in above. By doing this, the strategy will only focus the actions that certainly win if the action is made or lose if not made. And other actions will not get very high score.

4.3 Brief Summary

With the above improvements, we have optimalized our mini-max algorithm. It has achieved better performance when fighting with other AIs in both time cost and result quality. Besides, it becomes more adaptive to combine the VCX together.

5 Minimax + VCX

We have already shown that VCX-agent is a considerably good offender. However, there is still a severe problem of it, as we have mentioned, the VCX probably could not return with a result most of time. Therefore, we have to set up a back-up agent to make sure we could still return a reasonable action when VCX fails. And this is where the mini-max agent come into effect. Concretely, in our turn, we would first run a VCX to see if there is any killing move in limited time. If we fail to find one, our mini-max agent would take it over and return an action. The idea is simple, but the performance is impressive.

Typically, our agent could detect a killing move in 5 rounds within a time limit of 5 secs.

- Defect:

Nevertheless, the defense of our agent is not as powerful as our offense. This is because that our mini-max agent is only 5-6 layers deep, and is implemented with unsafe pruning. Hence, we could at most discover "danger" in three rounds later, which is obviously not enough to defend some good offenders like "NOESIS" and "SPARKLE".

- Possible solution:
  To handle the problem above, we thought, why don't we run a VCX detection for our opponent in our turn then we could discover the threat move to us and block it off. Concretely, in our turn, we first run a VCX agent for ourselves, if it fails to return a move, then we run a VCX agent for our opponent; again if it detect nothing, our mini-max agent come to play.

6 Defect and Potential Improvements

6.1 MCTS

As I have said before, our MCTS agent works not so well, it has the following drawbacks.

- Too many copy methods makes simulation times even not able to reach a thousand level.
- The actions to choose may also be optimalized by heuristic function like minimax.
- The end state can be computed a few steps earlier sometimes, with which we can save a lot time.

And due to the time limits, we don't have time for modifying these problems, we plan to put it in the near future, we still think MCTS is a powerful way to build a wise agent in gomoku.

6.2 Mini-max and VCX

- There is still space for us to do some time cost reduction to our VCX. Repeated computations still gets in our way of speeding up, but we do not have time to change and test it.
- We discovered that running VCX for our opponent, not as we have expected, did not bring us better performance. We think the reason is that adding one more VCX detection squeezed out some time allocated for the previous VCX. Before, we arrange 5 secs to one VCX to "think", but now we give each of two VCX agents only maybe 3 secs to "think"; then neither of the two agents could come out with a "good"action. Hence time allocation is an important issue that deserves our further consideration.

References

[1] Russell, Stuart J. and Norvig, Peter. Artificial Intelligence: A Modern Approach (2nd ed) *Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2*

[2] marble_xu Method of Evaluation Function in Gomoku In *https://blog.csdn.net/marble_xu/article/details/90450436*

[3] Zhentao Tang, Dongbin Zhao, Kun Shao, and Le lv. ADP with MCTS algorithm for Gomoku In *2016 IEEE Symp.Ser.Comput.Intel.SSCI 2016 (61273136) 2017*

[4] xmwd Gomoku based on MCTS and UCT RAVE in python. In *https://www.cnblogs.com/xmwd/p/python_game_based_on_MCTS_and_UCT_RAVE.html*