Final Project: Smart Gomoku Agent

Tianxiao Hu
School of Computer Science
Fudan University
txhu14@fudan.edu.cn

Hui Xu
School of Data Science
Fudan University
xuhui14@fudan.edu.cn

Bing Zhang
School of Data Science
Fudan University
14307130338@fudan.edu.cn

Abstract

This is abstract.

1 Introduction

This project is aimed to develop a smart agent for gomoku game.

2 Evaluation Function

First of all, in order to quantify the properties of the current situation, we construct an evaluation function to estimate the win-probability. And naturally, the evaluation function will incorporate a great deal of the knowledge about the Gomoku game. Therefore, the evaluation can be either naive or complicated, which depends on the your understanding towards the Gomoku game. In general, the more complex the evaluation is, the slower the program will get over time.

In this section, we mainly propose two versions of the evaluation function.

2.1 The First Version

The general idea of the first version is fairly simple. As is known to all, the object of the game is to be the first player to achieve five pieces in a row, horizontally, vertically or diagonally. Therefore, we can focus only on the five continuous positions on the chess-board, hereinafter called ”five-tuple”. Generally, the chess-board is 15×15, having 572 five-tuples in all. Then we can assign a score to every five-tuple based on the number of the black and white pieces in it. Here we ignore the relative position of pieces. Then the evaluation to a given situation is the sum of the score of all 572 five-tuples. Suppose we take piece ”x” and opponent takes piece ”o”, then the score table for me is displayed as follow,

Five-Tuple	Score
x	15
xx	400
xxx	1800
xxxx	100000
xxxxx	10000000
o	-35
oo	-800
ooo	-15000
oooo	-800000
ooooo	-10000000
Blank	7
Polluted	0

Note the score to blank five-tuple is 7 rather than 0. This is because there is worse condition, where the five-tuple is polluted, i.e., both black and white pieces in the tuple.

The first version of the evaluation function is fairly simple and works rapidly. Nevertheless, after trying dozens of man-machine games, we find the agent can make some stupid mistakes, which result from its excessively simple evaluation function. Actually, the defect can be well solved by applying minimax algorithm, but at a high price of the computing resource.

2.2 The Second Version

Then we intend to add more knowledge to the evaluation function to make it ”smarter”.

In the second version of the evaluation function, we take more account of chess-types, or in other words, the relative position of the pieces.

For every non-blank position on the chess-board, we take it as the center and extend four positions to both sides horizontally, vertically and diagonally. Then we can obtain four nine-tuples. For each nine-tuple, we check its chess-type and assign a score according to the new score table. Add up these four scores and we can get the score of this position.

Finally, our score is the sum of all positions occupied by our pieces while the opponent's score is the sum of all positions occupied by his pieces. And the evaluation to current situation is the difference of our score and opponent's score.

The new score table is stated below,

Chess-Type	Self-Score	Opponent-Score
Five	1000000	1000000
Alive-Four	20000	100000
CoDash-Four	6100	65000
GapDash-Four	6000	65000
CoAlive-Three	1100	5500
GapAlive-Three	1000	5000
Asleep-Three	300	200
Alive-Two	100	90
Asleep-Two	10	9
One	3	4

Note the names of the chess-types are fabricated by ourselves because we can't find accurate translation to these terms. If you are interest in the exact chess mode corresponding to the chess-types, please refer to the code or contact us.

3 Greedy Algorithm

Greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage, with the hope of finding a global optimum.

For our Gomoku agent, the evaluation function is exactly the so-called problem solving heuristic. And the implementation of the greedy search is elaborated as follows,

Algorithm 1: Greedy Search

```
Input: chess board
Output: position of move
1 for each position (i,j) on the board do
2     if position (i,j) is blank then
3         suppose place my stone on (i,j);
4         calculate my score: myScore;
5         calculate opponent score: opScore;
6         score(i,j) = myScore - opScore;
7     end
8 end
9 Return the position with the maximum score
```

However, after dozens of games, we find the greedy strategy does not in general produce an optimal solution. On the one hand, owing to the limitation of the evaluation function, the evaluated score of the current situation can't describe the win-probability precisely. On the other hand, this is also the inherent defect of the greedy algorithm. But nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time, which is appealing when solving many other problems.

4 Minimax Algorithm

4.1 Introduction to Minimax and α-β Prunning

4.2 Performance on Different Evaluation Functions

4.3 Speed Optimization

5 Monte Carlo Tree Search

5.1 Introduction to MCTS and UCT

Monte Carlo Tree Search (MCTS) is a tree search technique expanding the search tree based on random sampling of the search space. The application of Monte Carlo tree search in games is based on many playouts. In each playout, the game is played out to the very end by selecting moves at random. The final game result of each playout is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts. The strategy is going to have to balance playing all of the machines to gather that information, with concentrating the plays on the observed best machine. One strategy, called UCB1, does this by constructing statistical confidence intervals for each machine

$$\bar{x}_i \pm \sqrt{\frac{C \ln n}{n_i}}$$

$\bar{x}_i$: the mean playout for machine *i*
n_i: the number of plays of machine *i*
n: the total number of plays Then, the strategy is to pick the machine with the highest upper bound each time. Upper Confidence bound applied to Trees (UCT) is MCTS with UCB strategy.

For Gomoku game, MCTS starts with a chess board and walk chess randomly until the end. The process is repeated many times which eliminates the best move for the current chess board. Each round of Monte Carlo tree search consists of four steps:

- Selection: Build the root node based on the current chess board and generate all of its

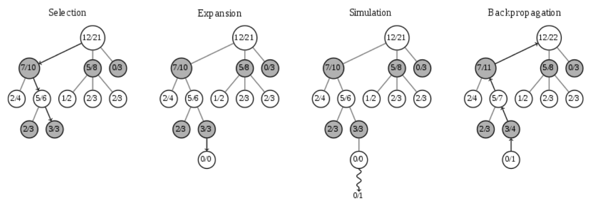

Figure 1: something

child nodes. The move to use would be chosen by the UCB1 algorithm and applied to obtain the next position to be considered.

- Expansion: Selection would then proceed until reach a position where not all of the child positions have statistics recorded.
- Simulation: If the node hasn't been simulated, then do a typical Monte Carlo simulation for chess game. Else, generate a random child node for the leaf node and do the simulation.
- Backpropagation: Update the reward of the simulation (generally 0 for lose and 1 for win) to the leaf node and its ancestor node. Meanwhile add the number of view for every node in the search path.

Repeat the playouts for many times until reach the search time or max search times and we can get the best move by selecting the maximum reward child node for the current root board. The pseudocode is showed as follow.

6 Round Robin for Agents

Acknowledgments

The acknowledgments should go immediately before the references.

A Supplemental Material