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
bingzhang14
@fudan.edu.cn

Abstract

Our Gomoku game supports two types of games, including HUMAN VS AI and AI VS AI. A well-designed user interface is provided and 3 versions of AI algorithms are implemented: **Greedy**, **Minimax Search** and **Monte Carlo Tree Search**. For source code used in the project, please visit our code respository[1].

1 Introduction

Gomoku, also called "five in a row", is a board game which originates from Japan. It is a game played on a Go board typically of size 15 x 15. In the game, players will take turns placing pieces until a player has managed to connect 5 in a row.

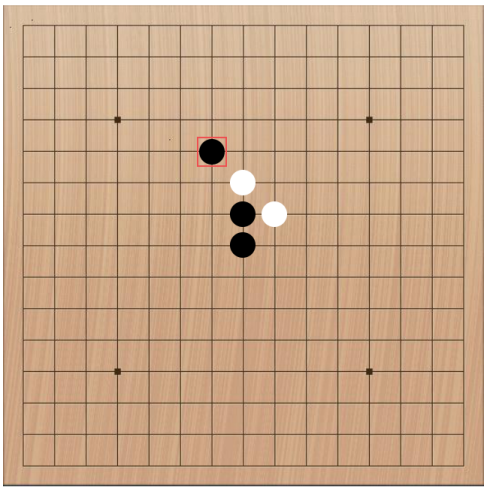


Figure 1: A Typical Gomoku Game

This project is aimed to develop a smart agent for Gomoku game. We use 3 main strategies to implement the agent: **Greedy**, **Minimax Search** and **Monte Carlo Tree Search(MCTS)**. We will introduce them in details in the following sections.
We also developed a user interface for Gomoku game. Special thanks to open source project gobang[2] for the beautiful design of the web page!

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

[1] https://github.com/TianxiaoHu/GomokuAgent

[2] https://github.com/lihongxun945/gobang

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	Opp-Score
Five	1000000	1000000
Alive-Four	20000	100000
CoDash-Four	6100	65000
GapDash-Four	6000	65000
CoAlive-Three	1100	5500
GapAlive-Three	1000	5000
CoAsleep-Three	300	200
GapAsleep-Three	290	200
TeAsleep-Three	290	200
FalseAlive-Three	290	200
Alive-Two	100	90
Asleep-Two	10	9
One	3	4
NoThreat	1	1

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.

2.3 Using Genetic Algorithm to find Better Evaluation Function

You may find in the score table above, the chess-types and corresponding self-scores and opponent scores are quite subjective. Thus, we apply genetic algorithm to find a better evaluation function, which is inspired by the process of natural selection(Bishop, 2007).

We initialize the algorithm by setting the original score table as the very first generation. Afterwards, it can breed a new generation by randomly plus or minus 3% specific chess-type score. The parent agent will play with each child agent 100 times and find the best children as the new parent. When the new generation can't defeat their parent, the algorithm comes to an end.

The results are as follow:

Chess-Type	Self-Score	Opp-Score
Five	1229874	1304773
Alive-Four	27685	119405
CoDash-Four	8198	84810
GapDash-Four	7164	75353
CoAlive-Three	1202	7176
GapAlive-Three	1426	6720
CoAsleep-Three	415	219
GapAsleep-Three	378	239
TeAsleep-Three	336	253
FalseAlive-Three	357	246
Alive-Two	127	111
Asleep-Two	11	12
One	4	5
NoThreat	1	1

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
  for each position (i, j) on the chess board **do**
    if position (i, j) is blank **then**
      suppose place my piece on (i, j)
      calculate my score: myScore
      calculate opponent score: opScore
      score(i, j) = myScore - opScore
  return the position with the max 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 Learn Openings from Game Record

After testing our agents for a number of games, we found opening is fairly important for Gomoku. If our agent is playing with a experienced human, it will easily lose the advantage during the opening. However, there are hundreds of different openings and they are hard to compute or collect. To solve the problem, we let our agent learn from 5581 top-level game records[3]. If current game matches some sequences of initial moves in the game records, it will compute scores for records and find the next step having the highest score.

5 Minimax Algorithm

5.1 Introduction to Minimax and Alpha-Beta Prunning

In general games, the maximin value of a player is the largest value that the player can be sure to get without knowing the actions of the other players; equivalently, it is the smallest value the other players can force the player to receive when they know his action(Russell and Norvig, 2009).

Calculating the maximin value of a player is done in a worst-case approach: for each possible action of the player, we check all possible actions of the other players and determine the worst possible combination of actions - the one that gives player i the smallest value. Then, we determine which action player i can take in order to make sure that this smallest value is the largest possible.

In Gomoku game, we use **evaluation function** to score each chess board. After a player has placed a piece on the chess board, a new state is created right after the old. Afterwards, agents can search the tree and return the best choice. However, due to the large search space of Gomoku, the tree will expand rapidly

[3]http://game.onegreen.net/Soft/HTML/47233.html

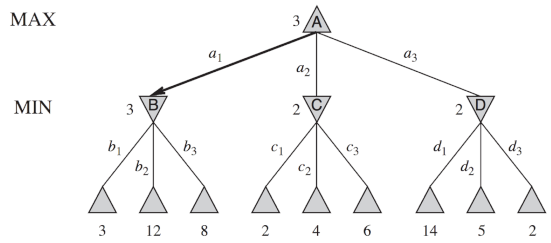


Figure 2: Example: Minimax Search

and lead to a unbearable waiting time. We employed Alpha-Beta Prunning to accelerate our agent.

Alpha-Beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the Minimax algorithm in its search tree. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further.

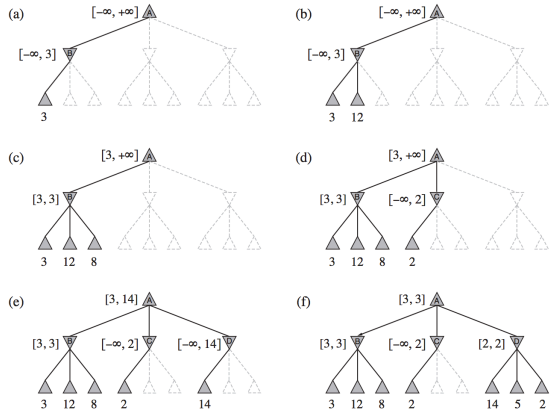


Figure 3: Example: Minimax Search

Our pseudocode is listed as right-above.

5.2 Speed Optimization

The depth of search tree is an important parameter in Minimax Search. If layers searched are not enough, our agent will be really short-sighted. However, for the sake of Python's efficiency, our naive implementation can only search for 2 layers in 1 second. If we force it to search 4 layers, the waiting time will become 10 seconds. Further optimization is needed.

Algorithm 2 Alpha-Beta Pruning algirithm

function alphabeta(node, depth, α, β, maximizingPlayer)
  if depth = 0 or node is a terminal node **then**
    return the heuristic value of node
    if maxmizingPlayer **then**
      v $\leftarrow -\infty$
      for each child of node **do**
        v $\leftarrow$ max(v, alphabeta(child, depth - 1, α, β, FALSE)
        $\alpha \leftarrow max(\alpha, v)$
        if $\beta \leq \alpha$ **then**
          break
        return v
    else
      $v \leftarrow +\infty$
      for each child of node **do**
        v $\leftarrow$ min(v, alphabeta(child, depth - 1, α, β, TRUE)
        $\beta \leftarrow max(\beta, v)$
        if $\beta \leq \alpha$ **then**
          break
        return v

We use several methods to accelerate our program:

- **Less Search States**
  We explore less search state for each player. Instead of search all possible place for a piece, we only consider places near pieces which are already placed in the chess board.

- **Zobrist Hashing States in Memory**
  During the expanding of the search tree, some nodes may share the same state. Their scores will be the same so we need not calculate them twice. We save each state and its score in memory to accelerate the agent.
  However, a state is hard to perform hashing. We use Zobrist Hashing to represent states. Zobrist Hashing starts by randomly generating bitstrings for each possible element of a board game, i.e. for each combination of a piece and a position. Now any board configuration

can be broken up into independent position components, which are mapped to the random bitstrings generated earlier. The final Zobrist Hashing is computed by combining those bitstrings using bitwise XOR. When updating positions, rather than computing the hash for the entire chess board every time, the hash value of a chess board can be updated simply by XOR out the bitstring for positions that have changed, and XOR in the bitstrings for the new positions.

- **Python's Numba Package**
  Our implementation is based on Python's scientific computing packages Numpy. We use Numba package to accelerate it. Numba is an Open Source NumPy-aware optimizing compiler for Python. It uses the LLVM compiler infrastructure to compile Python to machine code. This optimized function runs 200 times faster than the interpreted original function on a long NumPy array; and it is 30% faster than NumPy's builtin *sum()* function[4].

Our optimized Alpha-Beta Prunning can search 8 layers within 1 second. And it turned out to be really hard to defeat especially when the agent goes first.

6 Monte Carlo Tree Search

6.1 Introduction to MCTS and UCT

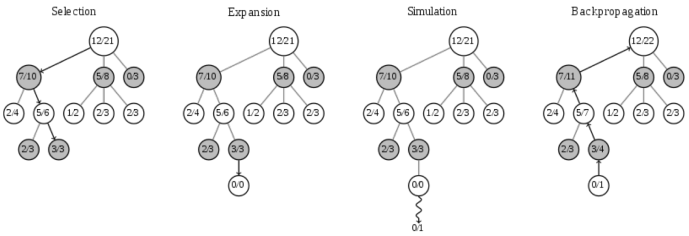
Monte Carlo Tree Search (MCTS) is a tree search technique expanding the search tree based on random sampling of the search space. The application of Monte Carlo tree search in games is based on many playouts. In each playout, the game is played out to the very end by selecting moves at random. The final 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 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.

[4] https://en.wikipedia.org/wiki/Numba

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.

Algorithm 3 The UCT algorithm

function UCTSearch(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow TreePolicy(v_0)$
    $\Delta \leftarrow DefaultPolicy(s(v_l))$
    $BackUp(v_l, \Delta)$
  return $a(BestChild(v_0, 0))$

function TreePolicy(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return Expand(v)
    else
      $v \leftarrow BestChild(v, Cp)$
  return v

function Expand(v)
  choose a $\in$ untried actions from A(s(v))
  add a new child v' to v with s(v') = f(s(v),a)
  and a(v') = a
  return v'

function BestChild(v,c)
  return $\arg\max \frac{Q(v')}{N(v')} + c\sqrt{\frac{2lnN(v)}{N(v')}}$

function DefaultPolicy(s)
  while s is non-terminal **do**
    choose a $\in$ A(s) uniformly at random
    s $\leftarrow$ f(s,a)
  return reward for state s

function BackUp(v,Δ)
  while v is not null **do**
    N(v) $\leftarrow$ N(v) + 1
    Q(v) $\leftarrow$ Q(v) + $\delta(v, p)$
    v $\leftarrow$ parent of v

Obviously the tree structure for a 15x15 Gomoku game chess board is too large, so the machine needs too many simulations for each choice of move and it take a long time for AI's move. Thus, the game experience is not very well.

Since we cannot speed up our algorithm by using cluster environment, we conduct a 4-in-row game in a 7x7 game chess board. And here is a view for a game result.

```
   a b c d e f g
  ==============
1| x - - - - - -
2| - - - o - - -
3| x - - x - - -
4| - o - x - - -
5| - - o x - - -
6| - - - o - - -
7| - - - - o - -
##########    AI3 is the WINNER!    #########
```

Figure 4: Result for MCTS algorithm

7 Round Robin for Agents

Several agents are implemented and we hold a round robin for them. From the competition result we can make a detailed analysis on different strategies.
Information of agents is listed here:

- AI1: Greedy Algorithm with evaluation function version 1
- AI2: 2-layer Minimax Search with evaluation function version 1
- AI3: Monte Carlo Tree Search
- AI4: 8-layer Minimax Search with evaluation function version 1
- AI5: Greedy Algorithm with evaluation function version 2
- AI6 Greedy Algorithm with evaluation function version 1 but also known better openings

Note that AI3 runs quite slow and due to the large search space, it's performance is not satisfying. Besides, **Monte Carlo Tree Search** approach sometimes attempts to place a few

pieces away from current pieces and build up "combination moves" that way, which is different from all other agents.

- **AI1 vs. AI2**
  The difference is between Greedy Algorithm and 2-layer Minimax Search. That's to say, whether the agent can consider the opponent's next step matters. We find 2-layer Minimax Search has a slight advantage. If AI2 goes first, it will win all the 100 games. But if AI1 goes first, things become quite different. AI1 can win 14 games and 6 games turn out to be tie. AI2 still has 80% winning percentage regardless of advantage of the upper edge.

- **AI1 vs. AI5**
  The difference between the two is the evaluation function. Contrasted to AI1, the evaluation function of AI5 is more reasonable, considering more possible cases. If AI1 goes first, it will win 75%games and if AI5 goes first, AI5 will win 89% games. The competition result has relevance to the order for placing pieces. But in total, AI5 performs better.

- **AI2 vs. AI5**
  When AI2 encounters with AI5, only one more layer for searching seems to be of no use. AI5 beats AI2 in each game and the result is independent of which AI goes first.

- **AI1 vs. AI6**
  AI6 can learn next step from 5570 top-level game records. When AI6 goes first, it can beat AI1 easily in a short time. But when AI1 goes first, AI6 can only win 50% games. The reason is that AI1 plays straight forward to carry out "five in a row" while AI6 will find a best place for latter pieces.

- **AI2 vs. AI6**
  Interestingly, AI6 has a very high winning percentage(90%) against AI2. We guess the reason is top-level openings can provide more opportunities for latter pieces. The games also end quickly, all of which less than 30 pieces in total.

- **AI5 vs. AI6**
  After AI1 has learned better openings, it performs better in competitions with AI5. If AI6 goes first, it can win about 50% games(AI1: 25%). However, it nearly has nothing to do with the winning percent when AI5 goes first. The winning percent only rise from 11% to 13%.

- **AI4 vs. all other AI**
  The depth of searching tree begins to dominate in the competitions of AI4. AI4 can easily beat any other AIs, even when AI4 is not the one goes first.
  Although AI4's evaluation function is not the best, it can gain an advantage step by step. If other AI goes first, AI4 may not perform very well at the very beginning of the game(always be busy defending). However, after about 30 pieces are placed, the situation will become balanced. After about 50 pieces are placed, AI4 will win.
  By the way, we invited several students in Fudan University that are good at Gomoku game to test our agents. According to their feedback, AI4 plays the best among all AIs especially AI4 goes first. "It's hard to defeat", one of the players told us.

References

Christopher M. Bishop. 2007. *Pattern Recognition and Machine Learning*. Springer.

Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach*. Prentice Hall.