Gomoku: A Fundamental Application for Adversarial Learning

Kemiao Huang, 11610728, *Undergraduate, CSE*

***Abstract*—Gomoku is a fundamental chess game with a clear-defined game rule. This work focuses on designing the adverarial algorithms for playing Gomoku automatically. The proposed methods mainly include heuristic searching, negemax and alpha-beta pruning. In the SUSTech Gomoku benchmark, this implementation shows competitive results.**

***Index Terms*—Artificial intelligence, Gomoku, adversarial learning, hurisitic searching, pruning**

I. Preliminaries

The goal of this project is to realize a Gomoku AI by using classical algorithms for game AI. To check how much intelligence does the AI have, the codes are submitted to the competition platform and evaluated as scores.

A. Software

This project is written by Python so PyCharm IDE was used because of its usefulness for editing and debug. The libraries contain numpy and functools.

B. Algorithm

The algorithms used in this project contain heuristic search, negamax and alpha-beta pruning.

The evaluation function is to compute the value which is associated with each position or state of the game. The value indicates how good it would be for a player to reach that position. The heuristic search is to pick out a number of empty positions which will most certainly win without far prediction.

Minimax assumes that the opponent will always make the best move. It let the player make the move that maximizes the minimum value of the position resulting from the opponent's possible following moves[1].

Alpha-beta pruning reduces the number of nodes that need to be evaluated in the search tree by the negamax algorithm.

Negamax algorithm relies on the fact that max(a,b) = -min(-a,-b) to simplify the implementation of the minimax algorithm. Algorithm optimizations for minimax are also equally applicable for Negamax. In this project, negamax with alpha-beta pruning is used.

II. Methodology

The details of the design of the Gomoku game AI will be discussed in this section.

A. Representation

The chessboard is represented as a numpy array. In the array, 0 means empty, -1 means black and 1 means white. The pieces in chessboard can be grouped as different patterns or threat types. Different patterns are labelled as different scores.

B. Architecture

The basic parameters are defined as global variables. The AI class contains attributes of chessboard, board size, color, time out, candidate list, count of total pieces, score caches for score evaluation of self and the opponent. Most functions are defined in the class except some static functions such as compare functions.

C. Details of Algorithms

1) Opening: To avoid heuristic search failure, the number of total pieces in the chessboard is counted at the beginning of the 'go' function. To be clear, a simple opening function is written.

Algorithm 1 simple opening

```
Input: $chessboard$
Output: position of next move
  if $chessboard$ is None then
    return None
  end if
  $count$ = number of pieces in $chessboard$
  if ($count > 1$) then
    return None
  end if
  if ($count == 0$) then
    index = center of $chessboard$
    return $index$
  end if
  return index next to the first piece at the opposite side respect to the chessboard
```

2) Score Evaluation: After the opening, the scores of the empty places are initialized for heuristic search. To reduce the time, all the empty places whose close neighbours are less than 2 are discarded when doing evaluation. The *evaluation* function is to evaluate the score for one index.

Algorithm 2 evaluation

```
Input: $piece\_index$, $color$, $direction$
Output: score of one empty space
  for $dir$ in the 4 directions do
    if $direction$ is None or $dir == direction$ then
      $cnt_1 = cnt_2 = 1$, $empty\_loc = 0$, $end_1 = end_2 = 0$, $empty_2$ = False
      do the first half traverse for $cnt_1$, $end_1$, $empty_2$, $empty\_loc$
      if $empty_2$ is not None then
        do the second half traverse for $cnt_2$, $end_2$, $empty_2$, $empty\_loc$
      end if
      use the values above to match the patterns and store the score in $cache$
    end if
  end for
  return sum of scores at index in $cache$
```

Generally, the evaluation for each direction is divided into two parts from the piece we focus. I use this kind of approach because it helps to check the pattern when there is a more than five loose connected with one empty space in it. Simply, if that situation occurs, it is either a connected five or a live four or a flush four. I can just check each part and the empty space location to get the pattern straightforward.

To reduce the time cost, the parameter 'direction' is used. The reason is that each time adding or removing a piece from the chessboard, the scores of spaces around the piece should be updated. The score of each piece is only changed through the direction of the line connected by itself and the new added or removed piece. On the other directions, it doesn't need to update.

Additionally, the $empty_2$ variable is used for check 'big jump live two', which is a pattern equivalent to live two and jump live two.

3) Heuristic Search: Heuristic function is to group up the empty pieces by their score levels and return the pieces list with best score level. Obviously, it needs some strategies to group the pieces to control the balance of keeping the number of the output small enough and ensuring the output contains the piece which will have the biggest threat.

Algorithm 3 heuristic search

```
Input: $color$
Output: pieces list with biggest threat
  initialize the empty lists: $five$, $live\_four$, $flush\_four$, $two\_three$, $three$, $two$, $one$ for self and opponent respectively.
  for empty $piece$ in chessboard do
    discard the too sparse pieces
    get the scores from caches for self and opponent
    append $piece$ to one right pattern list
  end for
  sort the lists
  check each list whether it is empty for self first and then opponent
  if $five$ or $live\_four$ is not empty then
    return $five$ or $live\_four$
  end if
  $result$ = combine the left non-empty lists in order by different priorities
  if $len(result) > max\_len$ then
    cut the tail of $result$
  end if
  return $result$
```

4) Negamax and Alpha-Beta Pruning: Negamax algorithm is executed to deepen the search. Although the alpha-beta pruning is used, the time complexity is still very large compared to the other functions in the game. Actually, the performance of using negamax is greatly depends on the evaluation function.

5) Sort and Comparison: After adjusting the scores by negamax algorithm, The candidate list should be sorted to get the best move. The sort() function in python can be used but the comparison function in sort in not allowed in Python 3. Therefore, I use the library 'functools' to convert the comparison function to key.

Moreover, the equal, greater, less et.al functions are given to approximately compare the value instead of using absolute comparison. The goal is to reserve the approximate results for further comparisons in algorithms of negamax and sorting. This method is referred from [2].

The two big difficulties in the algorithm implementation are the pattern matching and using recursion in pruning function. The number of code lines for matching and score evaluation is about 500 and pruning recursion is difficult to debug since the moves of pieces are too many.

Algorithm 4 negemax with alpha-beta pruning

```
Input: $self$, $depth$, $alpha$, $beta$, $color$
Output: best score with step
  $pieces\_list$ = pieces given by heuristic search
  if $depth$ == 0 or $pieces\_list$ is empty or $five$ score piece $\in$ $pieces\_list$ then
    $self\_max$ = $rival\_max$ = 0
    for $piece$ in empty places do
      $self\_max$ = max($self\_max$, $piece$ score in self score cache)
      $rival\_max$ = max($rival\_max$, $piece$ score in rival score cache)
    end for
    if $color$ == $self.color$ then
      return $self\_max - rival\_max$
    else
      return $rival\_max - self\_max$
    end if
  end if
  $best$ = -$\infty$
  for $piece$ in $pieces\_list$ do
    put $piece$
    $v$ = -$alphabeta(depth - 1, -beta, -alpha, -color)$
    if $v > best$ then
      $best$ = $v$
    end if
    $alpha$ = max($alpha$, $v.score$)
    remove $piece$
    if $v$ $greater\ than\ beta$ then
      return $beta$
    end if
  end for
  return $best$
```

Algorithm 5 compare scores

```
Input: $a$, $b$
Output: comparison value
  if $a.max\_score$ $equal$ $b.max\_score$ then
    return $b.totalScore$ - $a.totalScore$
  end if
  return $b.maxScore$ - $a.maxScore$
```

Algorithm 6 equal

```
Input: $a$, $b$
Output: comparison value
  $b$ = $b$ or 0.01
  if $b \geq 0$ then
    return $b/threshold \leq a \leq b * threshold$
  end if
  return $b * threshold \leq a \leq b/threshold$
```

Algorithm 7 greater

```
Input: $a$, $b$
Output: comparison value
  if $b \geq 0$ then
    return $a \geq (b + 0.1) * threshold$
  end if
  return $a \geq (b + 0.1)/threshold$
```

III. Empirical Verification

This section will discuss the test and result part of this project.

A. Design

Although the preliminary test file is provided, the test data is not enough for performance guarantee. Followed by the provided 'code_check.py' file, modify the '_check_advance_chessboard()' function and the expected result list can test all the specific chessboard as desired. Fortunately, the school has built a perfect on-line competition platform for us to upload the code and play against each other. The chess logs are reserved to modify the parameters the strategy for the algorithms, especially the evaluation function and the constant scores for different patterns.

B. Data

According to experience, the basic test should be able to check whether it can defence the two-live-three or flush-four-live-three threats. The number of the good way to attack is usually not only one so the test actually can only focus on defence.

C. Performance

The speed of negamax is the bottleneck of the entire program. The required time for one move is less than five seconds. In this project, interrupt function is not set so to ensure the result comes before time out the search depth for negamax is set as two.

D. Result

The evaluation of different patterns are summarized as a table. The self part should be larger than the opponent because attack has higher priority than defence. The scores are mostly set by experience. The learning algorithm is not considered in this project.

TABLE I: Scores for Chess Patterns

pattern	self	opponent
sleep one	15	10
live one	40	25
sleep two	120	75
(big)jump live two	260	240
live two	450	415
sleep three	650	500
jump live three	1550	1150
live three	1730	1425
flush four	2450	1750
live four	4750	3600
five	20000	15000

The other parameters in the program are all by testing.

TABLE II: Other Parameters

parameter	value
search depth	2
threshold	1.4
heuristic max len	10

The speed of each move with negamax algorithm is about 2 seconds. If the search depth is set as zero, the time cost is usually less than 0.5 second.

E. Analysis

Five seconds are usually sufficient for one move. However, when the search depth is set as 4 or larger, the result sometimes goes wrong and the AI is not much smarter. It is inferred that there may be some bugs when implementing the negamax algorithm. To further the searching algorithm, some strategies such as continuously finding moves for flush four and finding the check pieces for two-threes and flush-four-live-three can be used. Actually, those strategies can be ignored if the search depth is large enough but the benefit for those strategies is to save some time when doing search.

References

[1] Kuan Liang Tan, C. H. Tan, K. C. Tan and A. Tay, "Adaptive game AI for Gomoku," 2009 4th International Conference on Autonomous Robots and Agents, Wellington, 2000, pp. 507-512.

[2] Li, H. (2018). lihongxun945/gobang. [online] GitHub. Available at: https://github.com/lihongxun945/gobang [Accessed 12 Oct. 2018].