Report for Project Reversi

Jiayu Zhang
SID:11812425
Department of Computer Science and Engineering
CS303 Artificial Intelligence
Email: 11812425@mail.sustech.edu.cn

1. Preliminary

1.1. Problem Description

Othello is a game played by two players. Othello has been popular since the beginning of the 20th century and has been used for entertainment or research. Due to the diversity of its game, Othello still has no perfect mathematical solution so far, and computers have not been able to expand further in this regard. However, because of its relatively simple rules and easy program calculations, it is often used to research and develop artificial intelligence based on game algorithms that represent other board games.

In this project, based on the game algorithm, I used python to develop an Othello AI suitable for the internal battle platform of Southern University of Science and Technology. Among them, we used the Minimax search algorithm to construct the game tree to obtain the optimal solution at a certain depth. Based on this, we made alpha-beta pruning optimization to reduce the time required for search.

1.2. Problem Applications

The research on the game algorithm of Othello has strong practical significance. The game algorithm based on Othello can be extended to most chess games. In the future, such algorithm technology can also be applied to chess education and chess competition, greatly saving human resources. In addition, in a broad sense, such an algorithm can be used wherever there is a win or loss or a comparison, so such algorithm research is of practical significance.

2. Methodology

2.1. Notation

- $pos(a, b)$: Chessboard is an 8*8 matrix. The positions in the chessboard can be represented in a tuple. $pos(a, b)$ means the position in the $a - th$ row and $b - th$ column.
- $Xpos$: $pos(1, 1)$, $pos(1, 6)$, $pos(6, 1)$ and $pos(6, 6)$.
- $Cpos$: $pos(0, 1)$, $pos(1, 0)$, $pos(0, 6)$, $pos(1, 7)$, $pos(6, 0)$, $pos(7, 1)$, $pos(6, 7)$ and $pos(7, 6)$.

2.2. Data structure

- DIR: 8*2 matrix, eight positions around a chess piece, used to find the position that can be put and the mobility of the chess piece in the a_{th} row and b_{th} column.
- $WEIGHT$: 8*8 matrix, the corresponding weight of each position in the chessboard, used to calculate part of the evaluation function
- $STATUS$: 6*4 matrix, representing the respective weights of the evaluation function components at different stages in the whole game process, used to calculate the evaluation function
- $Candidate\_list$: A list of all the positions that can be put chess, and positions with higher evaluation values will be added to this list during the search process.
- $Chessboard$: 8*8 matrix, representing the chessboard. In the corresponding position, 1 is a white chess, -1 is a black chess, and 0 is no chess.

2.3. Model design

Board games are actually a weight evaluation process. At the same time, the weights of each piece are different, and the weights of the same piece at different time are also different. Even in different chess games, the weights of the same piece at the same time are different. Throughout the game, we need to dynamically evaluate the weights of these points in order to win.

Of course, just assessing the weight of the current move is not enough. It is even more obvious in Othello. Even if most of the time one side is dominant, the last move may still lead to a reversal of the situation. Therefore, we need to use Minimax algorithm to "predict the game". The Minimax algorithm is equivalent to a depth-first search. In the search process, due to the complexity of the Othello game, the time consumed for each additional layer search is exponentially increased. Therefore, we use alpha-beta pruning algorithm on this basis, which improves the performance of the Minimax algorithm without affecting the results at all.

In this project, due to platform limitations, it is impossible for us to search many layers in a single move, and the number of layers searched by everyone is similar, so

1000	-200	60	40	40	60	-200	1000
-200	-500	-5	-5	-5	-5	-500	-200
60	-5	3	2	2	3	-5	60
40	-5	2	1	1	2	-5	40
40	-5	2	1	1	2	-5	40
60	-5	3	2	2	3	-5	60
-200	-500	-5	-5	-5	-5	-500	-200
1000	-200	60	40	40	60	-200	1000

TABLE 1. WEIGHT

the performance difference is not obvious. Therefore, it is the evaluation function that most affects performance. The evaluation function designed by each person is different, resulting in a huge difference in performance.

In the whole process of Othello, there are many factors that affect the weight: position, mobility, stability and the number of reversible chess pieces.

Position is the weight relative to position on a global chessboard. In my design, as shown in Table 1, the weight on the corner is the largest, and $Xpos$ has the smallest weight. $Cpos$ has the relatively small weight, and the weight on the edge is relatively large. The rest positions are also assigned appropriate weights according to the degree of importance. This distribution is determined by the rules of Othello. For example, the chess pieces on the corners cannot be flipped, it is easier to flip the pieces inside the chessboard when occupying the edge, and occupying the X position leads to easily being flipped by the opponent.

Mobility is also a very important factor. At the beginning of the game, the strategy of taking as many pieces as possible is actually unreasonable, because most of the pieces at the beginning are concentrated in the middle of the board (except for situations where corners are needed). At the beginning of the game, you need to improve your mobility. Let your chess pieces be surrounded by the opponent's pieces as shown in Figure 1, so that you can wait for the game to develop to the middle and late stages, and you can flip many pieces in multiple directions in one fell swoop.

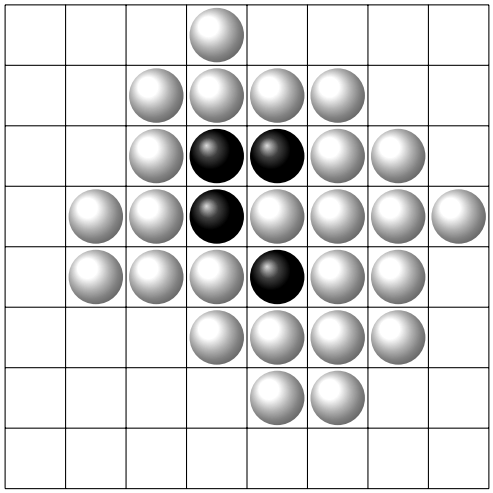

Figure 1. Mobility

The stability can be considered as the most important part of the evaluation function. A stable piece means a piece that cannot be flipped during the game. As shown in Figure 2, the triangular part on the corner cannot be flipped. Therefore, based on the rules of Othello, trying to accumulate as many pieces as possible that cannot be flipped is very conducive to winning the game. Since the stable pieces on the corners are difficult to occupy, we call the pieces on the edge of the chessboard that are clamped by the opponent's pieces as stable pieces as well. Even if they may be flipped in the future, they will generally not be in a few steps. So they relatively stable.

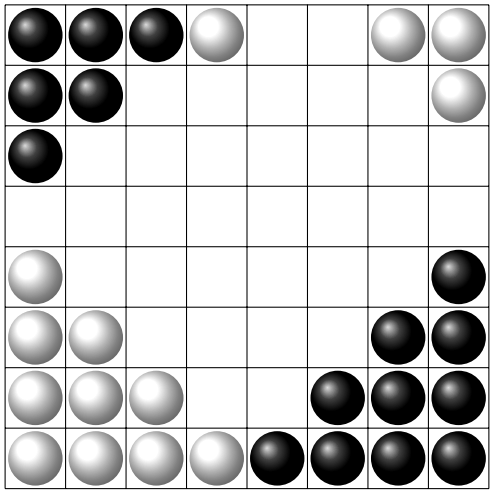

Figure 2. Stability

The number of pieces that can be flipped refers to the number of pieces that are flipped on the opposite side after one move. At the beginning of the game, this number can be set to a negative value, which has the same effect as the mobility. In the later stage of the game, this item will play a major role, because in the later stage of the game, the weights, corner positions, etc. are no longer concerned, and only the number of flipped pieces can leads to victory.

From the above we can also see that for different stages of the game, the degree of influence of each influencing factor is completely different. We need to constantly adjust the parameters of each part during the design process to make our AI more powerful.

2.4. Detail of algorithms

2.4.1. Maximum-Minimum Algorithm. The Minimax algorithm is equivalent to a depth first search. As shown in Figure 3, we need to maximize the value of A_0, then we need to select a maximum value from B_1 and B_2, and B_1 and B_2 need to minimize the value of A, so the layer where B_1 and B_2 are located is a minimal layer. That is, the minimum value is selected from the values of the next layer. Then we get that the value of B_1 is 2, and the value of B_2 is -1, so we will choose the path $A_0 - B_1 - A_3$.

2.4.2. Alpha-beta pruning. Since the tree constructed by the Minimax algorithm is very complex, we need to use

Algorithm 1 AlphaBeta

Input: $board$: chessboard, α: alpha, β: beta, $self\_color$: my color, now_color: side to move color, $depth$: depth searching now, $status$: status going now
Output: $value$: value evaluated, $move$: best move

$oppo\_color \leftarrow -self\_color$
$max \leftarrow -\infty$
$min \leftarrow +\infty$
$empty\_move \leftarrow ()$
if $depth < 0$ **then**
  return EVALUATE($board, self\_color, status$), $empty\_move$
end if
if !CANMOVE($board, now\_color$) **then**
  if !CANMOVE($board, -now\_color$) **then**
    return EVALUATE($board, self\_color, status$), $empty\_move$
  end if
  return ALPHABETA($board, \alpha, \beta, self\_color$, $-now\_color, depth, status$)
end if
for $move$ **do**
  $board, reverse\_number \leftarrow$ MAKEMOVE($board, move, now\_color$)
  $value, move\_tmp \leftarrow$ ALPHABETA($board, \alpha, \beta, self\_color$, $-now\_color, depth - 1, status$)
  $value \leftarrow value + reverse\_number * STATUS[status][3]$
  if $now\_color == self\_color$ **then**
    if $value > \alpha$ **then**
      if $value > \beta$ **then**
        return $value, move\_tmp$
      end if
      $alpha \leftarrow value$
    end if
    if $value > max$ **then**
      $max = value$
      $best\_move = move$
    end if
  else
    if $value < \beta$ **then**
      if $value < \alpha$ **then**
        return $value, move\_tmp$
      end if
      $beta \leftarrow value$
    end if
    if $value < min$ **then**
      $min = value$
      $best\_move = move$
    end if
  end if
end for
if $now\_color = self\_color$ **then**
  return $max, best\_move$
else
  return $min, best\_move$
end if

Algorithm 2 Evaluate

Input: $board$: chessboard, $self\_color$: my color, $status$: status going now
Output: $value$: value evaluated

$value1 \leftarrow 0$
$value2 \leftarrow 0$
$value3 \leftarrow 0$

for $point \in board$ **do**
  if $self\_color == point.color$ **then**
    $value1+ = Weight[point]$
  else
    $value1- = Weight[point]$
  end if
end for

$value2+ =$MOBILITY($self\_color$)
$value2- =$MOBILITY($-self_color$)

$value3+ =$STABILITY($self\_color$)
$value3- =$STABILITY($-self_color$)

$weight \leftarrow STATUS[status]$
$value \leftarrow value1 * weight[0] + value2 * weight[1] + value3 * weight[2]$
return value

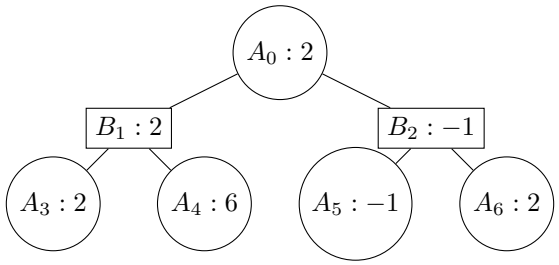

Figure 3. Minimax

alpha-beta pruning to optimize performance. As shown in Figure 4, if the A_3 node has been searched and its value is 2, then if a value greater than or equal to 2 is found in the child nodes of A_4, you can stop searching for the A_4 node. This is beta pruning. Alpha pruning is used at the maximum level.

2.4.3. Evaluation function. In the evaluation function, we first calculate the weight of the position. Then calculated their respective mobility, if it is our own mobility then this is positive, if it is the other side's mobility then this is negative. Finally, we calculated the stability, and also add if it is our own stability, and subtract if it is the opponent's stability. Finally we return a total weight multiplied by the weight. Note that here we have written the number of reversible pieces in the internal Minimax algorithm.

In the process of calculating the stability, we set the weights on the four corners to be larger. The stabilizer on

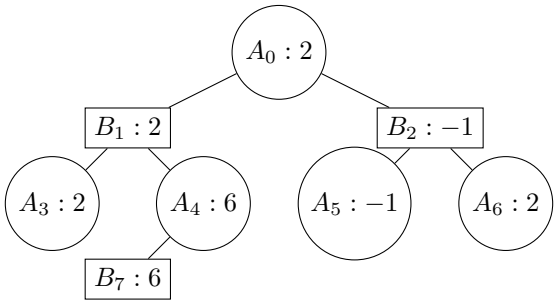

Figure 4. AlphaBeta

the angle is the smallest isosceles right triangle with the angle as the vertex of the right angle. The stable pieces on the side can only be counted as stable pieces if they are clamped by the opponent's pieces. As shown in Figure 5, the black chess pieces on the upper edge are considered stable, and the black chess pieces on the lower edge are not.

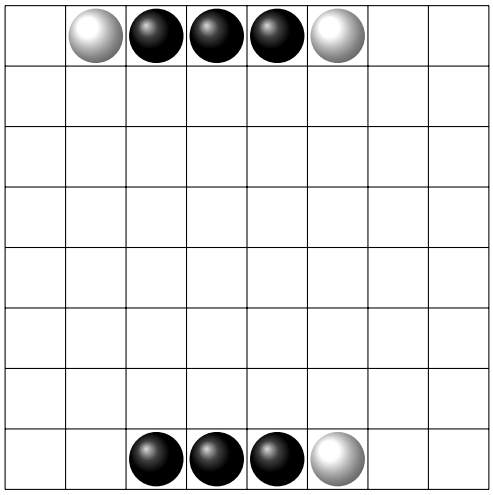

Figure 5. Edge Stability

3. Empirical Verification

3.1. Dataset

In the usability test phase, in addition to the 8 given test samples, I also tested a number of self-constructed samples to ensure that the code is universal.

In the points race and round-robin phases, many functions need to be tested separately. I wrote additional python files to test and output the phased results of each function. When a bug occurs, I insert some print functions into the source code to observe the intermediate values of some internal variables to determine where the problem is.

Before the playto function is closed, I judge the effect of my algorithm through playto and autoplay against my classmates; after the playto function is closed, I set up my own AI bot on Botzone, and each update is played against the previous strongest AI bot version in order to improve the effect of their algorithm.

3.2. Performance measure

Search consumption time: When the server resources are sufficient, when searching the fourth layer, most cases will not time out. But in order to ensure that the program can return a relatively good result in all cases, we first search for 2 layers, then 3 layers, and then search for the corresponding number of layers according to the game stage.

Integral ranking: Under normal circumstances, I will measure the strength of my algorithm based on the ranking of the integral. When I did not join the stabilizer, I was in the top 20, but with the progress of the project and the teacher's lecture, everyone's algorithm is getting stronger and stronger. Even if the stabilizer is added, the integral is always stable at 100. It's hard to get up and down. Later, I subdivided the game stage and greatly adjusted the weight distribution, so that the ranking rose all the way, and finally stabilized at about 30.

3.3. Hyperparameters

- $WEIGHT$: The corresponding weight of each position in the chessboard, the weight on the corner is the largest, the weight of the edge is larger, the weight of X is the smallest, and the weight of C is smaller. See Table 1 for details.
- $STATUS$: The whole chess game is divided into six stages, each stage has four parameters, namely the weight table, action force, stable disc and the number of flipped stones. As the game progresses, the importance of the first three parameters gradually decreases, and the weight of the number of flipped pieces gradually increases.
- $DEPTH$: In the third search, select the corresponding search level according to the game stage. Most of the time the game cannot be searched for more than 3 floors, and when the game progresses to the end, since there are very few places to go, it can even search for 9 floors.

The parameters were constantly adjusted during the battle with the old version of ai, and finally stabilized on a better version.

3.4. Experimental results

The test cases I pass in usability test is 10.
My rank in the points race is $32nd$.
My rank in the round robin is $25th$.

3.5. Conclusion

Under the algorithm I designed, the strategy of AI's chess is generally to give away the pieces at the beginning,

and try to take them if there are corners. Finally flip as many pieces as possible. This can increase the chance of victory in the later game, but it is also possible that my chess pieces are all flipped by the opponent shortly after the start. Although the parameters can be adjusted to make the initial sub-strategy more rigorous, it will also lead to a greater chance of losing to other types of AI.

In a project, the ranking may not fully reflect the strength of the algorithm, and it is even possible that the lower-ranked algorithm can beat the top ten algorithms. This is caused by the pertinence of everyone's parameters. Therefore, for algorithm optimization, I can start by guessing the opponent's chess strategy based on the opponent's historical decision, and adjust my own parameters according to the opponent's strategy to obtain a more targeted strategy.

At the same time, because the alpha-beta algorithm is not good at optimizing performance at all times, we can propose to use the history table to change the structure of the tree in the Minimax algorithm to achieve faster results with earlier pruning. In addition, after learning a deep learning course, I may be able to try to use the knowledge of deep learning to design algorithms.

Acknowledgments

I would like to thank Prof. $TangKe$ for his course. Also, I want to thank TA $ZhaoYao$ for her guidance.

References

[1] M. Buro, Experiments with Multi-Probcut a new high-quality evaluation function for Othello, in: H.J. vanden Herik, H. Iida (Eds.), Games in AI Research, Proceedings of a Workshop on Game-Tree Search Held in1997 at NECI in Princeton, NJ, Universiteit Maastricht, The Netherlands, 2000, pp. 77–96.

[2] Shoham, Y., Toledo, S. (2002). Parallel randomized best-first minimax search. *Artificial Intelligence, 137*(1-2), 165-196.

[3] Olivito, J., Resano, J., Briz, J. L. (2017). Accelerating board games through Hardware/Software codesign. *IEEE Transactions on Computational Intelligence and AI in Games, 9*(4), 393-401.

[4] Reversi. (2014, August 11). In $BotzoneWiki$. Retrieved November 1, 2020, from https://wiki.botzone.org.cn/index.php?title=Reversiaction=info