Reversi AI Based on Alpha-Beta pruning

Jinrui Zhang 11810124
Department of Computer Science and Engineering
Southern University of Science and Technology
11810124@mail.sustech.edu.cn

1. Preliminaries

1.1. Problem Description

Reversi is a game with a long history. The main content of this project is to design an AI of reversi. The main purpose of this project is to help us learn more about game algorithms.It is very complex to design a good AI. Firstly, Designer need to know the basic knowledge about reversi. Then, designer need to choose a proper algorithm. Moreover, good arguments are also very important. There are many factors need to be considered in a game, so this is a good way to learn about game algorithms.

1.2. Problem Application

There are a lot applications of this project. First, reversi AI can help to training human player. The level of AI can be easily adjusted so it can face players at all levels. Second, reversi AI can help human to do research about game algorithms cause the level of AI is a very direct way to evaluate the algorithm.

1.3. Software and Code

The language of this project is python and the version is python37. The IDE used in this project are Pycharm and Sublime Text. The code of this project can be divided into two parts, one is the AI part and the other is Checking part. AI part contains the logic of placing pieces and algorithm. Checking part is used to check the step detail and step time.

1.4. Algorithm

The basic algorithm of this AI is Alpha-Beta pruning.The evaluation function of this AI mainly consider the following parts: mobility, corner, "x" location, "C" location, border pieces, inside pieces, wall pieces and stable pieces. The weight of each part will be assigned according to the game status dynamically. To improve the efficiency, iterative deepening and zobrist hashing are added to this AI.

TABLE 1. NOTATIONS

Notation	Meaning
Node	Class to keep current complexion
xLocation	Number of pieces next to corner
depth	Searching depth
lower	Lower limit of current complexion
upper	Upper limit of current complexion
switch	Zobrist hashing value for white side
go(chessboard)	Decision-making
iterativeDeepening(size, maxDepth,...)	Iterative deepening
isValid(size, board, color, x, y)	Function to judge legal move
count(size, board)	Function to count total pieces
preProcess(list, size,...)	Function to process legal moves
canMove(size, ...)	Function to judge whether can move
evaluate(size, board)	Function to evaluate board
getEdge(board, color, size)	Function to count Edge pieces
getStable(board, color, size)	Function to count stable pieces
place(size, board, color, x, y)	Function to place piece at (x, y)
alphaBeta(size, board, alpha, ...)	Alpha-Beta Pruning function
update(hashCode, ...)	Function to update hash table
getHashCode(board, color)	Function to get hash value
getHash(hashcode, depth)	Function to get complexion

2. Methodology

2.1. Representation

Table 1 shows the main functions in AI and some variables which may lead to misunderstandings. The parameters of some functions are omitted for convenience of representation.

2.2. Data Structure

The data structures used in the code are listed below.

WEIGHTS A two dimension array to keep the evaluation weight of each location of board.

DIR A array to keep eight direction vectors.

hashTable A node list thar acts as a hash table.

white A two dimension array to keep zobrist hashing value of each location for white side.

black A two dimension array to keep zobrist hashing value of each location for black side.

candidate list A list to keep all legal moves

chessboard, board A two dimension array to represent current chess board.

oldBoard A two dimension array to represent chess board after placing piece.

score A list to keep simple evaluation value of each possible move.

isStable A two dimension Boolean array to maintain the information of stable pieces.

rowJudge A Boolean array to judge whether there is a blank in some row.

colJudge A Boolean array to judge whether there is a blank in some column.

digJudge1 A Boolean value to judge whether there is a blank in some diagonal line from top left to bottom right.

digJudge2 A Boolean value to judge whether there is a blank in some diagonal line from top right to bottom left.

2.3. Model Design

Designing a AI for reversi basically is a searching problem. The AI need to predicting which location the enemy most likely to place piece and which is the best location. Then using the Minimax game tree and designing a good evaluation function is a smooth conclusion. A good evaluation function need some basic knowledge to predict the best move. [1] has mentioned some good ways to understand the current chessboard. Thus some factors are considered in evaluation function: board weight, stable pieces, border pieces, inside pieces, corner location, "X" location, and corner pieces. An accurate assessment function is very important, it will not only enable you to accurately assess your own move, but also can accurately predict the enemy's move. For now the basic framework of the AI is in place. Another important factor which can affect the AI is searching depth. The larger the depth is the further the prediction is. To increase its searching depth, Alpha-Beta Pruning is used in this AI, because Alpha-Beta Pruning can cut off branches that have no impact on the final result. This will reduce the number of useless searches, thus making the search deeper. In addition, in order to achieve the effect of pruning, feasible moves are ordered so that better points could be searched first as far as possible. But using alpha beta pruning alone is not enough to improve efficiency, because some of the historical information generated in the game is not used.It is good way to do memorized searches with hashing. This AI using Zobrist hashing learned from [2]. In addition, in order to be able to adjust the search depth according to the step time, the AI uses the iterative deepening method. This is also a good method to improve time utilization.

2.4. Details of Algorithm

Alpha-Beta Pruning is the basic frame of this AI. α and β two value are added to help pruning.According to [3], for a root node, $\alpha = -\infty$ and $\beta = \infty$. And the value of α and β come from their parent node and update value according to their child node. If current node value $v < \alpha$, this node won't be the best choice. If $\alpha < v < \beta$, the value of α may need to be update. If $\beta \leq \alpha$, pruning happens.

Algorithm 1 Alpha Beta Pruning

Input: size, board, alpha, beta, color, depth
Output: bestValue

$bestValue \leftarrow -\infty$
$sign \leftarrow -1$
if color is my side color **then**
  $sign \leftarrow 1$
end if
if $depth \leq 0$ **then**
  return $sign * Evaluation(size, board)$
end if
if my side has no move **then**
  if opposite side has no move **then**
    return $sign * Evaluation(size, board)$
  end if
  return $-alphaBeta(size, board, -beta, -alpha, -color, depth)$
end if
$hashCode \leftarrow getHashCode(board, size)$
if Can find current complexion if hash table **then**
  return vaue in hash table
end if
$moves \leftarrow getMoves(size, board, color)$
$moves \leftarrow preProcess(moves, size, board, color)$
for each move **do**
  $MakeMove$
  $val \leftarrow -alphaBeta(size, board, -beta, -alpha, -color, depth - 1)$
  $UndoMove$
  if $val > alpha$ **then**
    if $val \geq beta$ **then**
      $UpdateHashTable$
      return val
    end if
    $alpha = val$
  end if
  $bestValue = MAX(val, bestValue)$
end for
$UpdateHashTable$
return $bestValue$

Evaluation function is a very important part of this AI. There are many factors are considered: board weight, stable pieces, mobility, border pieces, inside pieces, number Of pieces, corner location, "X" location, and corner pieces. The weight of different factors will be changed according to the status of game. The details of each factors are given below. **The "outside"** or boundary pieces are the piece which has blank next to it. The more the "outside" is the bad the complexion is, because the "outside" will give enemy possible mobility. On the contrary, **the "inside"** is the piece which has no blank next to it and it is very good cause it won't give extra mobility to enemy. **Number of pieces** is also a factor should be considered cause it is the final target of the whole game, but it is only important at later stage. **Edge** pieces are piece at edge. Edge pieces sometimes are also good because they decrease the mobility of enemy.All of these factors are indirect ways to increase one's own mobility and limit the enemy's. **Mobility** is a very important factor of reversi. When a party has a high level of mobility, it can choose a better position and force the opponent into a worse position. Therefore, the direct consideration of mobility is also an important aspect of evaluation. **Stable** pieces are pieces which can not be reversed. This is a factor directly relative to final target of this game. The stable part of this AI can be divided into two parts: pieces at full edge and pieces which has no blank at all directions. **Corner** is the most important location in this game, it can help one side get stable very easily. **"X"location** are the locations next to corner, if one side place at this kind of location firstly, it is easily for other side to get corner, so x location is very bad location.

Algorithm 2 Evaluation

Input: size, board
Output: value

$oppOut, myOut, oppIn, myIn, myMob, oppMob \leftarrow 0$
$oppEdge, myEdge, myNum, oppNum \leftarrow 0$
Calculate all factors listed above
$outside \leftarrow 0$
if $myOut > oppOut$ **then**
  $outside = +100 * myOut/(myOut + oppOut + 0.1)$
else
  $outside = 100 * oppOut/(myOut + oppOut + 0.1)$
end if
Calculate inside, mobility, edge and number with the same way like calculating outside
$corner \leftarrow numOfMyCorner - numOfOppCorner$
$stable \leftarrow numOfMyStable - numOfOppStable$
$xLocation \leftarrow 0$
for each location next corner and not at boundary **do**
  if corner next to this is not empty **then**
    $continue$
  end if
  if this location is my piece **then**
    $xLocation - -$
  end if
  if this location is oppsite piece **then**
    $xLocation + +$
  end if
end for
for each location next corner and at boundary **do**
  if corner next to this is not empty **then**
    $continue$
  end if
  if this location is my piece **then**
    $xLocation - = 0.29$
  end if
  if this location is oppsite piece **then**
    $xLocation + = 0.29$
  end if
end for
$weight \leftarrow 0$
for each location **do**
  if this location is my piece **then**
    $weight + = weightOfThisLocation$
  end if
  if this location is oppsite piece **then**
    $weight - = weightOfThisLocation$
  end if
end for
return Linear combination of these factors

Hashing is a very effective method to improve search efficiency. It can keep the complexion in hash table. When next time it meet the same complexion again, it can get the value directly. The hashing method of this AI is zobrist hashing. It is a very efficient way for game board hashing. It use 64 bit number as hashing key for both side at every location and using exclusive or get final hashing value of one complexion. The zobrist key value for each location is generated by C++ in advance.

Algorithm 3 Get hash code

Input: board, color
Output: hashValue

$hashValue \leftarrow 0$
for each location **do**
  $hashValue = hashValue \oplus keyValueOfLocation$
end for
if color is white **then**
  $hashValue = hashValue \oplus keyValueOfSwitch$
end if
return hashValue

To maintain the complexion, this a AI create a class node, which contains lock, depth, lower value, upper value and best move. When it update the complexion, if it is a historical complexion and depth is the same, only lower value, upper value and best move need to be updated. If the depth of current complexion is larger or this complexion is not a historical complexion, all the value in the hash table need to be update. This is called a deeper substitution, which means if the depth of same complexion is larger, the complexion is better.

Algorithm 4 Update hash table

Input: hashcode, lower, upper, bestMove, depth

1: $index \leftarrow hashcode\&TABLESIZE$
2: **if** $hashcode == table[index].lock$ and depth is same **then**
3: Update lower upper and bestMove
4: **else if** $hashcode == table[index].lock\&\&depth < table[index].depth$ **then**
5: Update all value
6: **else**
7: Update all value
8: **end if**

Iterative deepening is method to improve time utilization. It will adjust the searching depth according to the time limit. It will begin to search from depth equal to 2 and increase to 3. Every time when it finish one depth searching, it will update the best move. If time up or not enough time to continue the next search, it will end the searching. Because the use of hash, it won't waste a lot of time for a new searching. Thus, the searching depth is dynamically adjusted and it is more powerful.

Algorithm 5 Iterative deepening

Input: size, maxDepth, canditate list, board

1: $starTime \leftarrow currentTime$
2: **for** depth from 2 to maxDepth and time is enough **do**
3: **if** current complexion is in hash table **then**
4: $bestMove \leftarrow$ best move keep in hash table
5: **else**
6: $bestMove \leftarrow$ alphabeta searching for this depth
7: Update hash table
8: **end if**
9: Update best move
10: **end for**

Preprocess is also a important part of this AI. It will sort all legal moves and make the better moves be searched firstly.

Algorithm 6 Preprocess

Input: list, size, board, color
Output: list

1: $score[] \leftarrow empty$
2: **for** each move **do**
3: $MakeMove$
4: $score[move] \leftarrow Evaluation(size, board)$
5: $UndoMove$
6: **end for**
7: **if** color is enemy corlor **then**
8: Sort the list from smallest to largest by scores
9: **else**
10: Sort the list from largest to smallest by score
11: **end if**
12: **return** list

3. Empirical Verification

3.1. Data set

The data set of this AI project is totally comes from course platform. For usability test , except pass all test cases, I will randomly check the log for some game to check the if the program can find all legal moves. Most of my methods to improve AI rely on repeating a losing game, trying to find the move that caused the loss, then importing the situation of this step into the local area, and trying to find some problems with the method of one-step debugging. Then adjust the parameters and timelines of the evaluation function according to the problems found.

3.2. Performance

To test the efficiency of the algorithm, I wrote some tests locally. When checking the battle logs, I select some of the scenarios and enter them into the local application, then test the maximum search depth. Because this AI USES an iterative deepening algorithm, the maximum search depth can represent the search efficiency. The final performance of this AI is not bad, the rank in points race is 25 and the rank in round robin is 24.

3.3. Hyperparameters

The most important part of arguments in this AI is in evaluation function. Corner and "X" location are both very important so the weights of them are very large. Other arguments are listed in TABLE2. In the early and middle stages, the mobility and Angle are very important, and they should be given more weight. Boundary pieces, edge pieces, and inside pieces are all used to indirectly promote the mobility. In the middle and late stage, more attention should be paid to stabilizers, which are the key to win the final victory. Therefore, the weight of stabilizers in this stage should be increased.At the end of the game, the AI will only think about quantity of pieces.

TABLE 2. ARGUMENTS

state	number	mobility	stable	inside	edge	outside
$(0, 15]$	0	86	67	0	0	78
$(15, 18]$	0	98	100	0	22	76
$(18, 21]$	0	108	200	4	24	76
$(21, 24]$	5	116	300	5	22	72
$(24, 37]$	7	124	1000	6	22	70
$(37, 40]$	7	124	1500	7	16	66
$(40, 49]$	8	124	2000	0	0	60
$(50, 60]$	9	120	2250	0	0	56
$(61, 64]$	100	0	0	0	0	0

3.4. Result

This AI can search at most 4 level and it can pass all test cases in usability test. The final performance of this AI is not bad, the rank in points race is 25 and the rank in round robin is 24.

3.5. Conclusion

The advantage of this AI is that it use a lot of methods to improve time utilization, which is very important for improving searching depth. But there are a lot of things that need to be improved, such as the fact that the replacement strategy is too simple when updating the hash table, and the fact that the hash method has a low probability of collisions, which is not considered in the algorithm. In addition, the parameters used to evaluate the function are not perfect. Through this project, I have a deeper understanding of game algorithms and have my own understanding of designing better evaluation functions. Also, I learned that it is very difficult to design an appropriate set of parameters manually. In the future, I will try to use machine learning to solve problems when the same parameters need to be adjusted

References

[1] B. Rose, *Othello: A Minute to Learn... A Lifetime to Master*. Anjar, 2005.

[2] Zobrist hashing. [Online]. Available: {http://www.soongsky.com/othello/computer/zobrist}

[3] Application of some intelligent algorithms in reversi program. [Online]. Available: {https://wenku.baidu.com/view/6b740d497fd5360cbb1adb0b.html}