Artificial Intelligence of Reversed Reversi

Tianao Wang

Southern University of Science and Technology
12011014

1. Introduction

1.1. Background

Reversi is a strategy board game for two players, played on an 8x8 uncheckered board.[3] The rules of reversed reversi are almost identical to the rules of reversi but the only difference is that the object of the game is to have the least number of disk spins to show your color when the last playable empty square is filled.

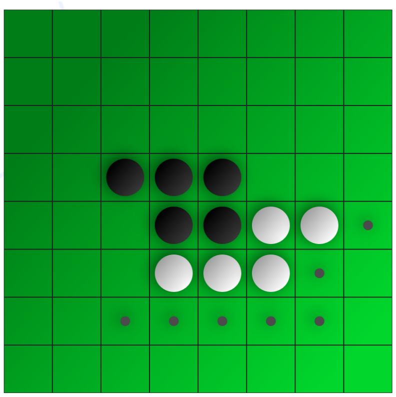


Figure 1. Reversi Game

In this project, we aim to implement reverse Reversi's AI algorithm according to interface requirements and submit it to the online system for usability testing and turn-based combat as required within a month.

1.2. Algorithms

To complete this project, I have used the following algorithms

1) *MiniMax Algorithm with A-B pruning*: Main algorithm for the Reversed Reversi AI
2) *Sorting Preprocessing*: Use a shallow depth to sort the candidate list in order to reduce subsequent searching.
3) *Heuristic Function with dynamic adjusting parameters*: An optimization for MiniMax Algorithm

Those algorithms are referred to the textbook[4] we learned.

1.3. Application

This project can help us in the following applications:

- Build an application for human-computer fighting of Reversed Reversi.
- Adversarial AI competition: Gobang, Halma, Reversi, I-go and more advanced competition.
- Limited Time Fast Effect AI competition: Competition held by TenCent, Alibaba, HuaWei, etc.

2. Methodology

2.1. Notation

There are some notations that I will use in my essay:

Symbol	Definition
w	weight board value
d	depth of solution tree
e	execution
k	weight coefficient
α	α in Minimax
β	β in Minimax
N	Numbers of stages need to explore
$b^\star$	Effective branching factor

TABLE 1. NOTATIONS LIST

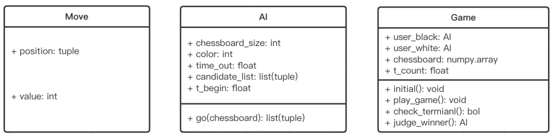
2.2. Problem Formulation

This problem can be formulated into a search problem[2]

1) *States*: The two-dimensional 8x8 board, and all the arrangement of pieces on the board
2) *Initial states*: The 8x8 board with 4 pieces placed on the middle.
3) *Player*: Which color player has the move in $State\ S$.
4) *Actions*: Add some possible candidate positions to the candidate list.
5) *Transition*: $S \times A \Rightarrow S$ Defines the result of a move.
6) *Utility(s,p)*: Utility that each time place a piece onto the board evaluated by the last few steps.
7) *Terminal test*: Determine who has minimum pieces on the board when the game is finished.

2.3. Data Structure

There are three main data structures used in this project(Figure 2):



(a) Move (b) AI (c) Game

Figure 2. Data Structures

2.4. General workflow

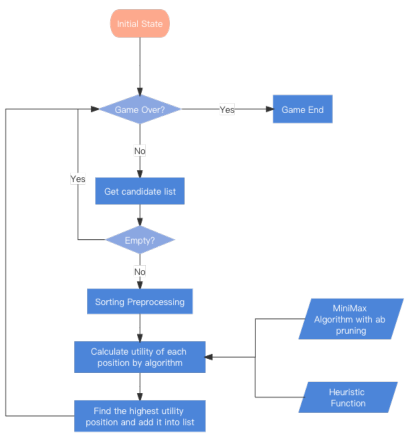


Figure 3. WorkFlow

2.5. Details of Algorithms

2.5.1. Get valid candidate list.

The basic require of this project is to pass the usability cases which means AI must find valid positions to set down the piece.What's more, *ChangeBoard* is also one of the most important function built to help me judge the state of the game. Basic methods are listed here:

Algorithm 1 *JudegCanGo*

Input: chessboard , color
Output: cangolist

```
cangolist ← find all the pieces whose color equals to none
for all blank_index in cangolist do
    if not CanMove(chessboard,color,blank_index) then
        remove this index from cangolist
    end if
end for
return cangolist
```

Algorithm 2 *CanMove*

Input: chessboard, color, index
Output: boolean

```
initialize boolean=False
for i, neighbour in (index+directions) do
    if neighbour is not valid or color of neighbour isn't another color then
        continue;
    end if
    while neighbour is valid do
        if color of neighbour is given color then
            boolean = True
            break;
        else if color of neighbour is blank then
            break;
        end if
        Update neighbour by directions[i]
    end while
end for
return boolean
```

Algorithm 3 *ChangeBoard*

Input: chessboard , chess, color
Output: chessboard_copy

```
copy chessboard_copy using chessboard
initialize eight directions, an empty change_list
change the chess in the chessboard_copy into the color
for all direction in directions do
    get a new chess by adding last chess and direction
    initialize an empty temporary list
    while new chess color is another color do
        append the new chess into the temporary list
        get a new chess again by same way
        if the chess_index is not valid then
            break;
        end if
    end while
    if the chess_index is not valid then
        continue;
    end if
    if new chess color is given color then
        append chess of temporary list into change_list
    end if
end for;
for all chess in the change_list do
    chessboard_copy[chess_new]=color
end for;
return chessboard_copy
```

2.5.2. MiniMax Algorithm with A-B pruning.

Minimax algorithm is the main logic algorithm widely used in the Adversarial Search. The alphabeta algorithm is a method for speeding up the Minimax searching routine by pruning off cases that will not be used anyway. This method takes advantage of the fact that every other level in the tree will maximize and every other level will minimize.[1] So I implement the MiniMax Algorithm, and add more pruning to it in order to get "best candidate" more quickly.

The principle of this algorithm displays as (Figure 4) vividly.

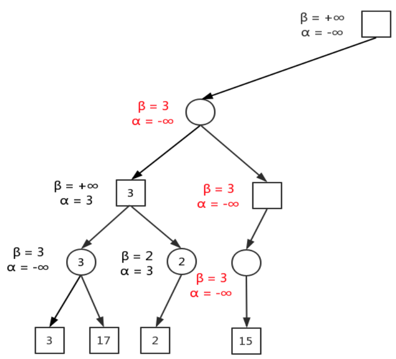


Figure 4. Minimax with ab pruning

Algorithm 4 *$\alpha\beta$MiniMax*

Input: is_behind, depth, α, β, chessboard, color
Output: value

cangolist=*JudgeCanGo(chessboard,color)*
if depth==0 or cangolist is empty **then**
 return *Evaluate(chessboard,cangolist,color)*
end if
if not is_behind **then**
  initialize value = -inf
  for chess in cangolist **do**
    chessboard_change=*ChangeBoard(chessboard, chess ,color)*
    value=*Max*(value,*$\alpha\beta$MiniMax(True,depth-1, α , β, chessboard_change , -color)*)
    α =*Max*(α, value)
    if $\beta <= \alpha$ **then**
      break;
    end if
  end for
 return value
else
  initialize value = inf
  for chess in cangolist **do**
    chessboard_change=*ChangeBoard(chessboard, chess , color)*
    value=*Min*(value,*$\alpha\beta$MiniMax(False,depth-1, α , β, chessboard_change , -color)*)
    β =*Min*(β, value)
    if $\beta <= \alpha$ **then**
      break;
    end if
  end for
 return value
end if

For different situations, I also use MiniMax algorithm with ab pruning by differentiating depth. What'more, I use *CheckTime()* method to get the execution time of the code.Finally, I call *WorstChoice()* Method in *Go()* to implement function in my AI.

Algorithm 5 *WorstChoice*

Input: chessboard
Output: chess

initialize min_score,min_chess,depth
calculate the step l played
if have possibility **then**
  increase depth
end if
for all chess in candidate_list **do**
  chessboard=*ChangeBoard(chessboard,chess,self.color)*
  score=*$\alpha\beta$Minimax()*
  if score<min_score **then**
    min_score=score,min_chess=chess
  end if
  CheckTime()
end for
return min_chess

2.5.3. Heuristic Evaluate Function.

Heuristic function is used to calculate the specific utility of one state. In my method, it not only contains some heuristic criteria, but also uses some dynamic parameters change strategy.Specifically, at beginning I randomly play in the center of the chessboard.Then at the middle of the game, I use a board weight map (Figure 5) and execution to evaluate the position. Finally, I use the number difference of the two players' chess as an evaluate method.

Algorithm 6 *Evaluate*

Input: chessboard, cangolist, color
Output: value

calculate chessboard weight w by chessboard
calculate execution e by cangolist, color
Judge the *Situation* to determine k
value = $w + ke$ // k is different in each step
return value

500	-25	10	5	5	10	-25	500
-25	-45	1	1	1	1	-45	-25
10	1	3	2	2	3	1	10
5	1	2	1	1	2	1	5
5	1	2	1	1	2	1	5
10	1	3	2	2	3	1	10
-25	-45	1	1	1	1	-45	-25
500	-25	10	5	5	10	-25	500

Figure 5. Weight Map[5]

3. Empirical Verification

3.1. Software and Hardware

3.1.1. Software.

- Reversed Revesi AI code: Python (Editor: Pycharm CE 2022.2.2)
- Online usability and round robin test: SUSTech Reversed Reversi AI platform
- Report writing: LATEX (Editor: Overleaf)
- Python: Version 3.8
- Numpy: Version 1.22.2
- Numba: Version 0.56.3

3.1.2. HardWare.

- Basic code and report writing: *MacBook Pro Intel Core i5 CPU*
- Test Platform: *Sustech OJ Server CPU with 32 cores 64 threads*

3.2. Usability Test

3.2.1. Basic Test Cases.

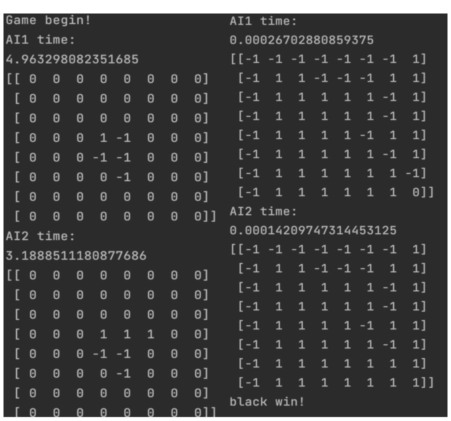
SUSTech Reversed Reversi AI platform provide ten basic test cases, which can prove the validity of our method.

3.2.2. Self Battle.

Class *Game()* is designed for the following reasons:

- Display each step as the AI chooses where to place the pieces.
- Let two AI class from different versions battle against each other and judge which is better.
- Show the progressiveness of each version of my AI design and choose the best one.

One example game whose players is my last version and penult version played on my platform shows as (Figure 6):



```
Game begin!
AI1 time:
4.963298082351685
[[ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  1 -1  0  0  0]
 [ 0  0  0 -1 -1  0  0  0]
 [ 0  0  0  0 -1  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]
AI2 time:
3.1888511180877686
[[ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  1  1  1  0  0]
 [ 0  0  0 -1 -1  0  0  0]
 [ 0  0  0  0 -1  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]
AI1 time:
0.00026702880859375
[[-1 -1 -1 -1 -1 -1 -1  1]
 [-1  1  1 -1 -1 -1  1  1]
 [-1  1  1  1  1  1 -1  1]
 [-1  1  1  1  1  1 -1  1]
 [-1  1  1  1  1 -1  1  1]
 [-1  1  1  1  1  1 -1  1]
 [-1  1  1  1  1  1  1 -1]
 [-1  1  1  1  1  1  1  0]]
AI2 time:
0.00014209747314453125
[[-1 -1 -1 -1 -1 -1 -1  1]
 [-1  1  1 -1 -1 -1  1  1]
 [-1  1  1  1  1  1 -1  1]
 [-1  1  1  1  1  1 -1  1]
 [-1  1  1  1  1 -1  1  1]
 [-1  1  1  1  1  1 -1  1]
 [-1  1  1  1  1  1  1  1]
 [-1  1  1  1  1  1  1  1]]
black win!
```

Figure 6. Game

Different version and their rankings on my platform shows as (Figure 7):

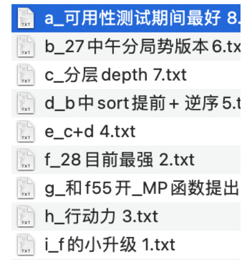


Figure 7. Ranking(version+ranking)

3.3. Time Performance Measure

3.3.1. Time Complexity Analysis.

Due to time constraints, my AI can only search the deepest 4, and find A-B pruning is less effective in practice.

$$T_{MiniMax} = O(b^d) \tag{1}$$

In practice, for each stages, AI needs to calculate it's utility. Suppose we need to check n grids. Then the time complexity of the utility is:

$$T_u = O_w(n) + O_m(n^2) = O(n^2) \tag{2}$$

Thus the total time complexity during the search of function *go()* of AI is $O(b^d n^2)$

3.3.2. Time Optimization.

- Use *numba* to optimized Python functions at runtime by using the industry standard LLVM compiler library. The speed has increased more than tenfold.
- Use *Sorting Preprocessing* to get the best result given by ab pruning Minimax earlier.The speed has increased about three times.
- Use *time.time()* to get a current optimal solution in the time limit.
- Use different *Evaluate()* like difference of chess number is better than weight map in pruning. Maybe it can help the function pruning useless branch earlier.

4. Conclusion

4.1. Advantages and Disadvantages of the Algorithms

In the whole projects, I have tried several algorithms to improve the intelligence of my AI – MiniMax algorithm with ab pruning, sorting preprocessing and Heuristic function and have successfully implemented them. What's more, I use numpy and numba to optimize the time cost, making the depth from 3 to 4 even 5. However, compared to other students I know, maybe my code is weak or optimization is less, the depth and decision of my AI still have many aspects needed to be improved.

4.2. Experiment and Experience

From this project, I study a lot in adversarial search and realize the essence of AI is mathematics. The Online Playing Platform bring me a lot fun by 'fighting against' with my classmate.Additionally, I find although generally with more optimizations the AI is stronger, sometimes an AI with very little optimization also can beat an AI with relatively strong performance and many optimizations.It teaches me that everyone has weakness and never be discouraged. All in all, I am appreciated for this experience.

4.3. Deficiencies and Possible Improvement Directions

If given more time, I will improve my *JudgeCanGo* function by *bitwise operation* and using two ways(begin from my pieces or blank grid) in different period to search. What's more, I will add some other evaluate methods in my *Evaluate()* such as stable factors. Additionally, I will use Genetic Algorithm even Neural network to train my AI and get more suitable coefficient in my adversarial search algorithm.

References

[1] Hendrawan Armanto et al. "Evolutionary Neural Network for Othello Game". In: *Procedia - Social and Behavioral Sciences.* 57 (2012), pp. 419–425.

[2] Wikipedia contributors. *Computer Othello — Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Computer_Othello. 2022.

[3] Wikipedia contributors. *Reversi — Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Reversi. 2022.

[4] J.R.Stuart and N.Peter. "Artificial Intelligence A Mordern Approach." In: 2010.

[5] Orion Nebula. *Black and White AI: Situation evaluation + AlphaBeta pruning pre-search*. https://zhuanlan.zhihu.com/p/35121997. 2018.