Report of Reversed Reversi

Li Jiajun 11912021
Department of Computer Science and Engineering
Southern University of Science and Technology
11912021@mail.sustech.edu.cn

1. Preliminaries

Reversed Reversi is a two-player chess game. The two players take turns to place white or black disks on the board. When two disks of the same color block the opponent's disks in a straight line, the opposite disks between the two disks will be flipped to the other color side. The game will continue until there is no empty spot on the board, and the winner would be the one whose number of disks existing on the board is smaller.

1.1. Software

The project uses *Python* with editor *Pycharm*. Operating on *Windows 10*.

1.2. Algorithm

The project focuses on Minimax Search algorithm with alpha-beta pruning, which is one of the adverserial search algorithms that is widely applied in multi-agent competitive games such as chess games.

1.3. Applications

The core algorithm applied in this project is Minimax Search algoritm and alpha-beta pruning, this algorithm is very common when it comes to competitive chess games because it predicts future moves and evaluates which move is most beneficial to our side and destructive to the opponent, therefore increase the odds of winning.

Apart from chess games, Minimax algoritm is useful in route planning. In reality, there are often multiple route choices when we travel from one position to another, but some streets may be blocked or jammed due to heavy traffic while some may be too far away but with less traffic load. Minimax is capable of estimating all possible routes and find one that costs the least time.

2. Methodology

This section demonstrates the design idea of the minimax algorithm that is applied in Reversed Reversi.

2.1. Representation

2.1.1. Notation.

This section introduces the meaning of functions in the project.

get_all_actions(chessboard, color): This function searches for available positions to put down the disk of the selected color.

evaluate(chessboard): The function evaluates the situation of the board by invoking several other sub evaluate functions.

alpha_beta(chessboard, color, a, b): This function performs Minimax Search algorithm with alpha-beta pruning to search for the best position for the next move.

get_total_number(chessboard, color): This function counts the total number of a certain colored disk.

evaluate_weight: This function evaluate the total weight sum based on the weight matrix and the disks' positions.

stable(chessboard): This function counts the total number of stable disks of our side.

frontier(chessboard): This function counts the total number of frontier disks of our side.

2.1.2. Data Structure.

This section introduces the functions of significant data structures in the project.

chessboard: A two-dimensional array, containing integer elements that demonstrates "vacancy", "black disk" and "white disk".

candidate_list: A list of tuples that represents the available moves' coordinates.

weight: A two-dimensional array displaying the weight value to evaluate the corresponding chessboard position.

2.2. Architecture

To decide each move, a grading criteria is adopted. The evaluation criteria is based on frontier disks, stable disks, remaining steps and weight matrix. Frontier disk is a disk with empty space nearby, so such disks are easy to be flipped. Stable disk represents the disk that can not be flipped, such disks usually appear on the corners and edges. Remaining steps is the length of the action list. To win the game, we need more frontier disks and less stable disks, so that my disk can be easily transferred to the opponent's color. Also, more remaining steps is beneficial to my side because more steps means more choices to increase my winning odds. Furthermore, the weight matrix is the most important criteria, because it introduces the priority of each position. For reversed reversi, the corner must have the lowest priority while the positions near the corners enjoy higher priority, so that the program could force the opponent to capture the corner and therefore creating a stable disk.

When a board is presented, the first step is to find all positions available to put down the disks. According to the game rule, the player must choose an empty spot where he can flip the opponents disks after he put down his own disk.

To perform the task efficiently, two approaches are adopted. The first one is to search from the existing disk of current player's color, and expand the searching distance at eight searching directions until an available empty position is spotted. The second approach is to search from the current empty positions and determine whether the current empty block is available. If the disks of our side outnumbers the empty blocks, then the program perform the first search approach, otherwise it chooses the second one.

After finding all candidates, the program performs Minimax with alpha-beta pruning. Minimax regards the opponent as a "smart" agent who would make the best decisions to gain most advantage, therefore the program would search for the most favorable step for its side and restrain the opponent at the same time. What determines the capability of the algorithm is the maximun depth it could reach. The greater the maximun search depth, the more time it costs. Since there is a time limit of five seconds, the depth should not be too big, but if the depth is too small, we may risk wasting computation resource and lose the game. Therefore an innovative mechanism—-dynamic depth planning is adopted. Based on the number of candidate positions, we adjust the maximun depth. If the number of candidates is small, the depth is given a small value, and if the candidate number is big, then the depth would be bigger. Meanwhile, when the number of remaining empty positions is small, the depth is also given a value big enough to predict the following situations until the game is over.

2.3. Detail of Algorithm

This section illustrates the pseudocode of important functions in the project. The pseudocode fragments are distributed in the following pages due to the inconvenient layout.

Algorithm 1 evaluate_weight

Input: chessboard
Output: score

```
score ← 0
for i = 0 → 8 do
    for j = 0 → 8 do
        if chessboard[i][j] = my disk then
            score+ = weight[i][j]
        else if chessboard[i][j] = opponent then
            score− = weight[i][j]
        end if
    end for
end for
return score
```

Algorithm 2 stable

Input: chessboard
Output: number

```
number ← 0
drow ← [−1, −1, −1, 0, 0, 1, 1, 1]
dcol ← [−1, 0, 1, −1, 1, −1, 0, 1]
for each corner position c do
    if c ← mycolor then
        number += 1
        for j = 0 → 7 do
            row = c.row + drow[j]
            col = c.col + dcol[j]
            while (row, col) is within the board do
                if chessboard[row][col] = mycolor then
                    number += 1
                    row += drow[j]
                    col += dcol[j]
                end if
            end while
        end for
    end if
end for
return number
```

3. Empirical Verification

3.1. Data Set

To verify the performance of the program, I chose some specific opponents to match. The opponents are my familiar classmates whose algorithms' general architecture is described to me. Some used random generated move at the early stage, some applied Minimax without alpha-beta pruning, and some used alpha-beta pruning but with different search depth.

Also, the logs of certain perticular games are useful. For instance, after consulting with my friend, we both used our own weight matrix to battle, and after each game of ours, I would download the log to review the steps and try to ameliorate the weight matrix and other hyperparameters.

Algorithm 3 frontier

Input: chessboard
Output: number

$number \leftarrow 0$
$drow \leftarrow [-1, -1, -1, 0, 0, 1, 1, 1]$
$dcol \leftarrow [-1, 0, 1, -1, 1, -1, 0, 1]$
for $i = 1 \rightarrow 6$ **do**
  for $j = 1 \rightarrow 6$ **do**
    if chessboard[i][j] = mycolor **then**
      for $i = 0 \rightarrow 7$ **do**
        row = i + drow[k]
        col = j + dcol[k]
        if chessboard[row][col] == empty **then**
          number += 1
        end if
      end for
    end if
  end for
end for
return number

Algorithm 4 min_value

Input: chessboard, alpha, beta, color
Output: value

if current depth ¿ max depth **then**
  return evaluate(chessboard)
end if
if action_list is empty **then**
  if opponent action_list is empty **then**
    return evaluate(chessboard)
  end if
  return max_value(chessboard, alpha, beta, -color)
end if
$value \leftarrow infinity$
for each $acts \in action\_list$ **do**
  move and change the board
  current depth increase by 1
  value ← min(value, max_value(chessboard, alpha, beta, -color))
  current depth decrease by 1
  if value ¿= beta **then**
    return value
  end if
  alpha ← min(alpha, value)
end for
return value

Algorithm 5 get_all_actions

Input: chessboard, color
Output: action_list

$disks \leftarrow$coordinates with value color
$empty_spots \leftarrow$empty coordinates
$drow \leftarrow [-1, -1, -1, 0, 0, 1, 1, 1]$
$dcol \leftarrow [-1, 0, 1, -1, 1, -1, 0, 1]$
if $get\_total\_number(chessboard, color) <= get\_total\_number(chessboard, empty)$ **then**
  for each $disk \in disks$ **do**
    for $i = 0 \rightarrow 7$ **do**
      $(row, col) \leftarrow (disk[0] + drow[i], disk[1] + dcol[i])$
      $permission \leftarrow False$
      while (row, col) is within the board **do**
        if chessboard[row][col] = -1 * color **then**
          $row+ = drow[i]$
          $col+ = dcol[i]$
          $permission \leftarrow True$
        else if chessboard[row][col] = 0 and permission is True **then**
          action_list adds (row, col)
        end if
      end while
    end for
  end for
else
  for each $empty\_spot \in empty\_spots$ **do**
    for $j = 0 \rightarrow 7$ **do**
      $(row, col) \leftarrow (empty\_spot[0] + drow[i], empty\_spot[1] + dcol[i])$
      $permission \leftarrow False$
      while (row, col) is within the board **do**
        if chessboard[row][col] = -1 * color **then**
          $row+ = drow[i]$
          $col+ = dcol[i]$
          $permission \leftarrow True$
        else if permission is True and chessboard[row][col] = color **then**
          action_list adds (row, col)
        end if
      end while
    end for
  end for
end if

3.2. Performance

To measure the performance, my first step is to utilize the usability test. Judging from the failed test cases, the approximate time cost could be inferred. When the maximun search depth is too large, the program would pass fewer test cases. Therefore the appropriate search depth is estimated.

The second step is to estimate the winning percentage in auto playing and judge the performance based on the ranking on the platform. After some program optimization, my ranking has improved.

Algorithm 6 max_value

Input: chessboard, alpha, beta, color
Output: value

if current depth ¿ max depth **then**
  return evaluate(chessboard)
end if
if action_list is empty **then**
  if opponent action_list is empty **then**
    return evaluate(chessboard)
  end if
  return min_value(chessboard, alpha, beta, -color)
end if
$value \leftarrow -infinity$
for each $acts \in action\_list$ **do**
  move and change the board
  current depth increase by 1
  value ← max(value, min_value(chessboard, alpha, beta, -color))
  current depth decrease by 1
  if value ¿= beta **then**
    return value
  end if
  alpha ← max(alpha, value)
end for
return value

Algorithm 7 alpha-beta

Input: chessboard, color
Output: action

best_score ← -infinity
beta ← infinity
action ← empty tuple
for each act ∈ action_list **do**
  move and change the board
  current depth increase by 1
  value ← min(value, max_value(chessboard, alpha, beta, color))
  current depth decrease by 1
  if value ¿ best_score **then**
    best_score ← value
    action $gets$ act
  end if
end for
return action

Lastly is to measure the performance based on competitions with certain opponents. For instance, after optimizing the action searching function, the program is able to run with more search depth, and the competitions with my classmate's Minimax algorithm with only two-level searching depth proved that my efforts of increasing search depth is worthwhile.

3.3. Hyperparameters

There are multiple hyperparameters. The first one is the two-dimensional array named weight, storing the weight value in each position. The second parameter is the current search depth, which is initially zero and will change during the Minimax searching procedure. The third parameter is max_depth, meaning the maximun search depth of the Minimax algorithm. Other hyperparameters acts as the weight value of the sub functions in the final evaluate function.

To achieve better performance, I focused on three approaches, the first one is to improve the evaluation methodology, which requires adjustments to the weight matrix. During the verification process, I observed the fact that the four corners' value were supposed to be as low as possible so that the program would make the most effort to avoid occupying the corners. The second approach is to reduce the time cost, which is mostly controled by the search depth of Minimax algorithm. By adjusting the parameter max_depth, the time cost could be regulated at an appropriate range. By fine tuning the evaluation parameters manually based on the outcomes of multiple games, the program could achieve higher grades.

3.4. Result

The program passed all ten usability test cases.

The ranking changed several times after multiple updates. The ranking was around one hundred when I first introduced Minimax with alpha-beta pruning. My ranking advanced to around sixty after the update on get_all_actions, because I managed to increase the search depth from 2 to 3. The ranking then stablized at around forty after the modulation on the weight matrix.

The final Round Robin ranking is 80/208.

3.5. Conclusion

Based on the test results, it is obvious that my optimization on searching method and weight matrix has had potent effects. And the effort to increase the maximun search depth has proven to be vital in competing with other alpha-beta pruning algorithms. Taking stable disks and other factors into consideration is a significant approach to win the game.

As for the disadvantages, the primary one is that my evaluation algorithm is optimized manually without solid scientific bases. Also, the process to acquire the number of stable disk is so simplified that not all stable disks are included.

From the project, I have gained a better understanding about adverserial search and the essence of chess games. After reflecting on the project, I think one possible improvement might be switching the evaluation function to a dynamic one that can change its evaluation metrics based on the current situation. Another improvement is that the parameters can be adjusted more precisely and effectively if self battling and machine learning can be involved, so that the optimal parameters can be acquired.

Acknowledgments

Many thanks to my fellow classmates and friends who enlightened me when I encounter problems.

References

[1] J.S.Fulda, ”Alpha-Beta pruning,” ACM SIGART Bulletin, (94), pp. 26, 1985. Available: https://www.proquest.com/scholarly-journals/alpha-beta-pruning/docview/29759060/se-2?accountid=162699. DOI: http://dx.doi.org/10.1145/1056313.1056315.

[2] D.Billman and D.Shaman, ”Strategy knowledge and strategy change in skilled performance: a study of the game Othello,” Am.J.Psychol., vol.103, (2), pp.145-166, 1990. Available:https://www.proquest.com/scholarly-journals/strategy-knowledge-change-skilled-performance/docview/79818776/se-2?accountid=162699.