Artificial Intelligence Design for Oware with Alpha-Beta pruning and MiniMax algorithm

HE Chenchen

December 2018

Abstract

The objective of this project is to design an artificial intelligence (AI) for an Oware game with modified playing and capturing rules as well as board configuration. The end goal of the project is to design AI-based player which will always win the game. The AI-player will need to perform the search with a time limitation. In this project, we utilized Alpha-Beta pruning MiniMax algorithm in order to search for its best move with simple a evaluation function. As part of the project, we will be exploring different evaluation metrics and adding more heuristics for the new game.

Keywords: *Oware, alpha-beta pruning, minimax, heuristic*

1 Introduction

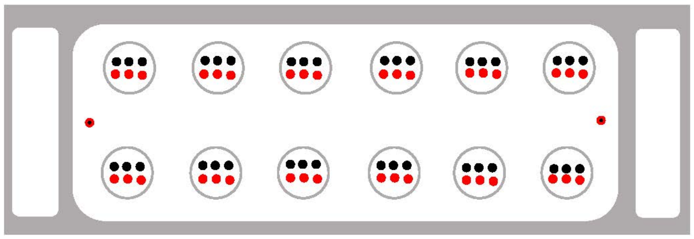
Oware is a two-player strategy game whose aim is to capture seeds from opposite player's side as many as possible under certain play and capture rules. The game starts with a game board which has a predefined number of holes for each player's hole and a predefined number of seeds with different colors for each hole. In our game design of Oware, the two players will be the human and the AI-based computer player. From the artificial intelligence perspective, the aim of this game is achieve a state where computer will always win the game by applying Mini-max Search and Alpha-beta Pruning. In this case, computer player uses a minimax search tree with a depth of several steps to foresee every valid move of the game. Mini-max search algorithm is used for determining which move should the computer take. Meanwhile, alpha-beta pruning is also used for reducing the time and memory consuming and thus increasing the efficiency of the move decision. Another fundamental and essential part in this artificial intelligence designing is the evaluation function since the value taken by the mini-max search is calculated via the evaluation function. More than that, the evaluation function also have impact on the efficiency. For instance, when given a large depth for minimax search, time consuming will increase along with the increasing of the complexity of evaluation function. In this report, we also make some explorations for different evaluation function to test the difference.

This report is structured as follows. In section 2, we introduce the basic rules of our new oware game. In section 3, we show the artificial intelligence algorithms used in our project, i.e., minimax and alpha-beta pruning, our implementation and present the pseudo-codes. In section 4, we explored different evaluation function. In section 5, we show and discuss the results that we have. In section 6, we conclude what we have done in the project.

2 Game Rules

Oware is a board game which has many variants since it can have different board configuration with changes in the initial number and the color of seeds in a hole, and the number of holes for each player. The variants can also come from the capture rules i.e., under what condition we can capture[1]. In this report, we have a different rules for the oware game. There are 12 holes, 6 seeds including 3 red and 3 black per player at the beginning. Each player has 1 special seed which is black and red. The first player decide the position of the special seed and then the second player.

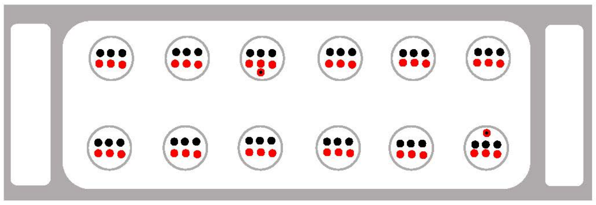
Figure 1: Oware Game Board



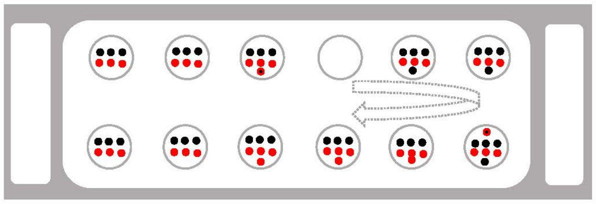
2.1 Sowing rules

Players take turns moving the seeds, they choose one of their valid holes, which means the hole is not empty, named as starting-hole. They remove all seeds from the starting-hole, and distribute them, dropping seeds one by one in clockwise turn starting from the starting-hole. This processing is called sowing. Seeds are not allowed to be distributed into the end scoring holes, nor into the starting hole. The starting-hole is always left empty when current player finish its sowing. If the starting-hole contains more than 12 holes, when it sowing the twelfth seed, it skipped the starting-hole moving to the next hole.

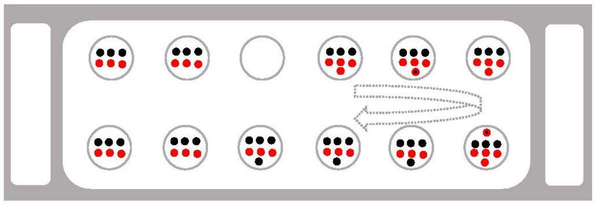
Figure 2: Sowing examples



4B2: The 4th hole, Black seed first, no special seeds (ignore the last “2”)



3R2: The 3rd hole, Red seed first, special seed is played on the second sowing step



Moves are made according to the colors. Once the color is decided by the player, all this kind of seeds are played first and then the other color are played (if applicable). Thus, a move is expressed by **NCS**, where **N** is the index of the hole, **C** is the first color which is played, **S** is the first position of the seed. For instance, **3R2** means that we choose the hole 3 as the starting-hole, remove all the seeds and distributed it to clockwise holes and sowing Red seeds first and then the other color. If we have special seed, we sow it in the second step, otherwise there's nothing to do with special seeds.

2.2 Capture rules

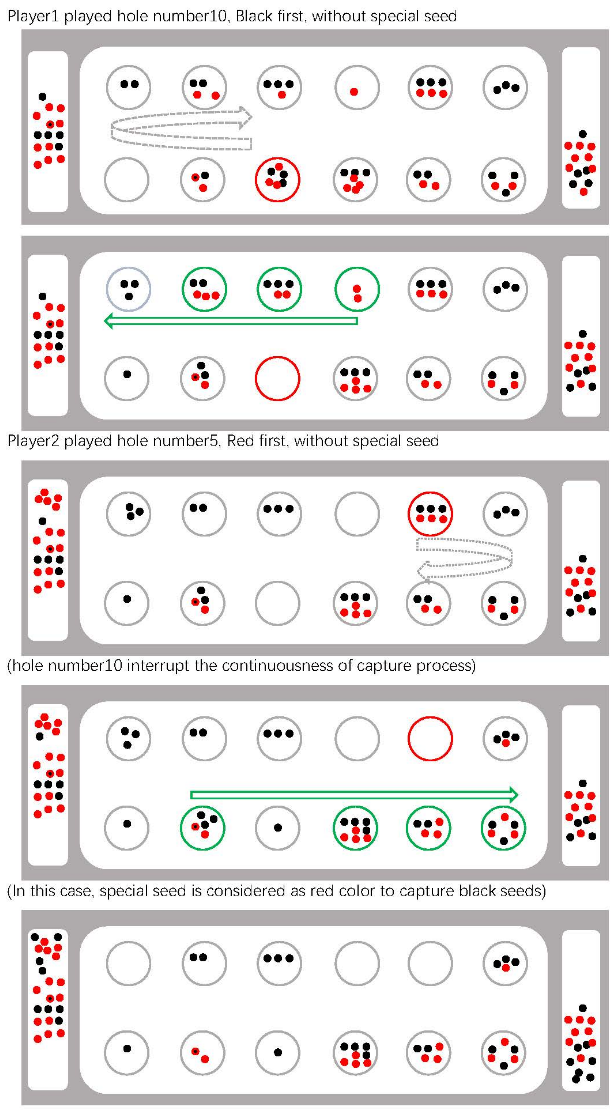
In oware, capturing occurs only when a player brings the account of an opponent's hole to exactly two or three seeds. Here are some rules of capturing for the new game.

- The last seed determine the color of the capture seeds, e.g., if the last seed is red then only red seeds can be captured
- If the last seed is special then red and black seeds can be captured,The color of the seed that has been captured in the hole i determines the color of the seed that can be captured in the previous hole $(i-1)$

- Example: 2R2B1special,2R2B,3R,2B4R,2R4B1S,
  2R2B1S: the color is R and B: all the seeds are taken
  2R2B: the color is R and B : all the seeds are taken
  3R: the color is R and B: all the seeds are taken
  2B4R: the color is R. No seed is taken

This always captures the seeds in the corresponding hole,and possibly more: If the previous to last state seed also brought an opponent's hole to two or three seeds of the same color as the final seed, these are captured as well and so on until a hole is reached a break condition which does not contain two or three seeds of the same color of the final seed or does not belong to opponent. Taking all the seeds of the opponent is not allowed. In case of starving all the seeds are taken by the last player. When there's no more valid move, the game stops and each player takes the sees of its side.

Figure 3: Capturing examples


Player1 played hole number10, Black first, without special seed
Player2 played hole number5, Red first, without special seed
(hole number10 interrupt the continuousness of capture process)
(In this case, special seed is considered as red color to capture black seeds)

2.3 Winning condition

The game is over when one player has captured 38 or more seeds, or each player has taken 37 seeds. If both players agree that the game has become reduced to an endless cycle, the game ends when each player has seeds in his holes and then each player captures the seeds on their side of the board.

2.4 Time limitation

The computation time of the computer player should less than 2 seconds. This constraint will impact the depth that is able to be performed for the Minimax search, which will not be allowed to exceed the time limitation.

With the new Oware game design in this project has two colors i.e., red and black, and a special seed for each player, the computing and memory consumption is a lot more than the original game design. NCS is the input of each player. N is the hole that player wants to play which has 6 possible cases for each player; C is the color of the seed that you want to first sow in current round which has 2 possible cases; s is the index of which step the player wants to play it's special seed if current play hole has special seed, which has the number of total seeds in current play hole possible cases. So when generating children nodes for a node, it has $Numberofvalidmoves \times Numberofcolortype \times Numberofseedsforcurrentmove$ which is very time and memory consume. Therefore, it may cannot go very deep when doing the minimax search not only because of the time limitation in this new game but also because of the hardware of computer.

3 Artificial Intelligence Algorithms

In this section, we introduce two artificial intelligence algorithms[2] that are adopted in our designing for oware game. That is minimax and alpha-beta pruning minimax. *Minimax* is a tree search algorithm. The root node of the tree is the current state of the board. From the root node, the algorithm considers all the valid moves and generates them one by one to obtain a new state of board representing the child of the root node. From this child node, the algorithm uses the same mechanism to get its children node until it reaches the leaves, that is to say, the state of the node is a terminal state of the game or the the depth of the current node has reached the predefined depth of the tree. In the leaf layer, the algorithm applies an evaluation function which can compute the scores or the rewards of the leaf node state and return to its parent node and repeat this process until reach the root node and thus, in the end, the root node can get the best next move and the score it will get if the root follow the move.

Since the minimax search algorithm is implemented recursively, it must be completely expanded and visited which is very time-consuming and memory consuming especially when the branch and the depth are both large. *Alpha-beta pruning* is a strategy that can be used to reduce the number of visited nodes. Alpha-beta pruning can improve the efficiency of the minimax by stopping a move evaluation when it finds out at least one possibility that prove the move will be worse than the previous examined one and thus it will reduce the time and memory consumption.

3.1 Mini-max Search Algorithm

In order to design our game, we first implement a minimax without alpha-beta pruning to generate the search tree correctly.

Algorithm 1 Minimax pseudo-code

```
1:  function MINIMAX(node, MaximizingPlayer)
2:      if node is terminal then
3:          return the reward of the node
4:      end if
5:      if maximizingPlayer then
6:          bestValue ← $-\infty$
7:          for all child of node do
8:              val ← MINIMAX(child, False)
9:              bestValue ← MAX(bestValue,val)
10:         end for
11:         return bestValue
12:     else
13:         bestValue ← $+\infty$
14:         for all child of node do
15:             val ← MINIMAX(child, True)
16:             bestValue ← MIN(bestValue,val)
17:         end for
18:         return bestValue
19:     end if
20: end function
```

The pseudo-code of the implementation of minimax algorithm is shown in Algorithm1. From Line 1 to Line 4, the algorithm is checking if the current node is leave node, if it is, we will return the value of evaluation function. From Line 5 to Line 11, the algorithm will check which player is using the minimax, its best value will be initialized to $-\infty$. For every child node, it calls the minimax function and saves the return value and when this return value is greater than best value, we will update the best value. After we finish the evaluation of all children nodes, we return the best value. As for Line 12 and Line 20, they will be implemented only when it is the opponent player. In this case, its best value will be set into $+\infty$. For every child node, it will call the minimax function, and saves the return value and if this return value is smaller than the best value, the algorithm will update best value. After finishing the evaluation of all the children nodes, we return the best value.

3.2 Alpha-beta Pruning

After we implement minimax algorithm correctly, we add alpha-beta pruning to reduce the visited nodes keep the accuracy at the mean time since we only pruning those who will not update the best value.

Algorithm 2 Alpha-beta pruning minimax pseudo-code

```
1: function $\alpha - \beta -$MINIMAX($node, MaximizingPlayer$)
2:     if $node$ is terminal then
3:         return $the\ reward\ value\ of\ the\ node$
4:     end if
5:     if $maximizingPlayer$ then
6:         $bestValue \leftarrow -\infty$
7:         for all $child\ of\ node$ do
8:             $val \leftarrow$ MINIMAX($child$, False)
9:             $bestValue \leftarrow$ MAX($bestValue,val$)
10:            $\alpha \leftarrow$ MAX($\alpha,bestValue$)
11:            if $\alpha \leq \beta$ then
12:                break
13:            end if
14:        end for
15:        return $bestValue$
16:    else
17:        $bestValue \leftarrow +\infty$
18:        for all $child\ of\ node$ do
19:            $val \leftarrow$ MINIMAX($child$, True)
20:            $bestValue \leftarrow$ MIN($bestValue,val$)
21:            $\beta \leftarrow$ MAX($\beta,bestValue$)
22:            if $\beta \leq \alpha$ then
23:                break
24:            end if
25:        end for
26:        return $bestValue$
27:    end if
28: end function
```

In alpha-beta pruning minimax algorithm, α is the best alternative for the max, β is the best alternative for min. From Line 1 to Line 4, the algorithm is checking whether the node is a terminal node or not, if it is, just terminate the function. Line 5 to Line 15 is checking who is using the minimax function.The best value is initialized as $-\infty$. Every children node calls the minimax function and store the return value if the value is greater than the best value, then update the best value. After that, if the best value is greater than the α, then update the α. On top of the α and β value we can stop evaluating children node, if $\alpha \leq \beta$. After finishing all necessary evaluation for children nodes, return the best value. Line 16 to Line 28 will be implemented if it is the opponent player, its best value is set as $+\infty$. For every children node, it calls minimax function to get its evaluation score and if the score is smaller than best value, update the best value. If best value is greater than β, then we can update β. After that, The algorithm will stop visiting the rest children nodes when $\beta \leq \alpha$. After finish all necessary evaluation of children nodes, return the best value.

3.3 Algorithm implementation

This is our algorithm implementation for oware game.

Algorithm 3 Alpha-beta pruning minimax pseudo-code

```
function $\alpha - \beta -$MINIMAX($node, \alpha, \beta, player, depth$)
    if $node$ is terminal OR $depth = 0$ then //leaf node
        return Bestmove // the index of this node
    end if
    if $Player = 0$ then // Computer's turn
        $bestValue \leftarrow (-74)$
        $isValidChildren \leftarrow$ isValid($node$)
        $Bestmove \leftarrow$ firstValidmove//initialized to firstValidmove
        for all $child$ in $isValidChildren$ do
            $move \leftarrow$ MINIMAX($child, \alpha, \beta$, player, depth $- 1$)
            $val \leftarrow$ EVALUATION($node$)
            $bestValue \leftarrow$ MAX$(bestValue, val)$
            //Bestmove should have a better score
            $Bestmove \leftarrow$ BETTER$(move, Bestmove)$
            $\alpha \leftarrow$ MAX$(\alpha, bestValue)$
            if $\alpha \leq \beta$ then
                break
            end if
        end for
        return $Bestmove$
    else
        $bestValue \leftarrow (+74)$
        $isValidChildren \leftarrow$ isValid($node$)
        $Bestmove \leftarrow$ firstValidmove//initialized to firstValidmove
        for all $child$ in $isValidChildren$ do
            $move \leftarrow$ MINIMAX($child, \alpha, \beta$, player, depth $- 1$)
            $val \leftarrow$ EVALUATION($node$)
            $bestValue \leftarrow$ MIN$(bestValue, val)$
            $Bestmove \leftarrow$ WORSE$(move, Bestmove)$
            $\beta \leftarrow$ MAX$(\beta, bestValue)$
            if $\beta \leq \alpha$ then
                break
            end if
        end for
        return $Bestmove$
    end if
end function

function EVALUATE SCORE($Node$)
    return computer.seeds-player.seeds
end function
```

From the AI perspective, the goal of the game to let the computer win the game. Therefore, when we implement the algorithms it's computer who want to maximize score, and human player is the opponent who wants to minimize computer's score when we doing the minimax. Therefore, when it is computer's turn, we will choose a best next move for it via alpha-beta minimax search with a certain search depth. Since there are $12 holes \times 6 seeds + 2 special seeds = 74 seeds$ in total, so we initialize best value to be -74 and $+74$ instead of $+\infty$ and $-\infty$. More than that, we also checked the validation of the move when we generating the search tree nodes because we don't have to generate an node which we won't reach it. In our case, every time, we will return the best move for current player. If it is computer player, the best move will let computer get the biggest score, otherwise, the best move will let the opponent get the smallest score.

4 Explorations for Heuristics

In this project, different experiments have been carried out in order to improve the game playing at different stages. A Minimax algorithm was first written with the evaluation function in Algorithm 3 which evaluates according to the score difference between two. Alpha-Beta Pruning was then implemented in order to reduce the number of nodes traversed to search for the best move for the player in consideration of the time limitations. Despite the improvements, the agent was not able to go deeper than depth 4 within the time limit of 2 due to the complexity of the game rules and hence a better heuristics function to initialize the best move in the event where none of the move is better is implemented.

Algorithm 4 Selection heuristic function for all equal move

```
1: function SELECT BEST NODE($Nodes$)
2:     $min \leftarrow -999$
3:     $mincell \leftarrow 0$
4:     for index 6 to 1 do
5:         $difference \leftarrow Nodes[index] - Nodes[index - 1]$
6:         if $difference < min$ then
7:             min $\leftarrow$ MIN*difference, min*
8:             min*cell* $\leftarrow index - 1$
9:         end if
10:    end for
11:    return min cell
12: end function
```

This new heuristics added into the game playing system has provided better initialization for the game at the start of the game. The goal of this heuristics is to ensure that the move returned is one that optimizes the differences in the number of seeds between the holes on the computer side. This will make sure that opponents would not be able to capture seeds in consecutive holes. This heuristics is implemented throughout the game when there are no best move based on the evaluation score to ensure that the seeds are optimized for this purpose.

Another heuristics that has been implemented in the game playing system is an algorithm to induce starving state for the opponent but not starving them by choosing the move that would leads to more seeds on our side than the opponent side. This heuristics will kick in when the game has only less than 25 seeds left.

Algorithm 5 Evaluation function

```
1: function EVALUATE SEEDS($Node$)
2:     $totalplayerseeds \leftarrow 0$
3:     $totalcomputerseeds \leftarrow 0$
4:     for all $player\ cells$ in $Node.cells$ do
5:         $total(player/computer)seeds \leftarrow \sum Node.cells$
6:     end for
7:     return total computer seeds - total player seeds
8: end function
```

According to the rules of the game, taking all the seeds of the opponent is allowed and in the case of starving all the seeds are captured by the last player. This heuristics aims to reach to the state where capturing all the seeds of the opponents could happen or starving could happen.

5 Results

In this section, the experimental observations and results will be discussed. Firstly, the implementation of Alpha beta pruning has enabled the algorithm to increase its depth 3 to 4 as shown in the table below. Implementation of Algorithm 4 has improved the initialization of the game and the computer has started to gain seeds at much earlier steps than the implementation without a proper initialization strategy.

Max Depth for each algorithm	
Algorithm	Depth
Minimax	3
Minimax + Alpha Beta	4
Minimax + Alpha Beta + Initial heuristics	4
Minimax + Alpha Beta + Initial heuristics + Seeds difference	4

6 Conclusion

In conclusion, besides the implementation of the game playing strategies with the various game algorithms, heuristics plays an important part in the the game strategy initialization and at different stages of the game. More research can be studied for the different strategies to best optimize for this game.

References

[1] Colin Divilly, Colm O'Riordan, and Seamus Hill. Exploration and analysis of the evolution of strategies for mancala variants. In *Computational Intelligence in Games*, pages 1–7, 2013.

[2] G. Rovaris. *Design of Artificial Intelligence for Mancala Games*. POLITESI, 2017.