# Report

## COSC276, Fall2021, PA3, Xuedan Zou

## A. Description

We implemented minimax search algorithm, the evaluation funcion, the Alpha-Beta pruning and the iterative deepening in this project, all based on the chess game.

Firstly, minimax search algorithm is implemented with an recursive structure. Minimax search algorithm always computes the best achievable utility against a rational adversary. However, sometimes searching the optimal solution could be very time confusing so we add a max depth to limit its deepest searching depth. For each node of the minimax search tree, it could be either a max node or a min mode, we should use a variable to memorize this property during each time of the recursive calling. Besides, we count the depth of each recursive calling so that when the depth meets the maximal deepth we set before, even possibly our tree has not reached the terminal state, we should stop our minimax search and begin backtracking. We also need the *board* to determine the current state. However, since we want to avoid initializing the board for each recursive calling, we save the board as a property in MinimaxAI class and use *board.push()* with *board.pop* to change the state of the game to express the successive states. We use the following code to pass the successive board to the recursive child.

```
                self.board.push(move)
                res = self.minimax(depth+1, False)
                # we dont want to initialize another new board which costs a lot so we
just pop the previous move to return to the current's board
                self.board.pop()
```

Secondly, we develope an evaluation function to help our minimax search algorithm finding whether the utility is good. We first check whether the game had ended under the current state and if not we continue with our evaluation function. Our center idea of this evaluation function is based on the textbook: give each type of chess piece a weighted value and do the weighted linear sum of all chess pieces. We dont need to count for the King here since the game will end if either of the King vanished. Based on our knowledge of chess, queen is the most important type of chess piece, then is ROCK, follwed by KNIGHT and BISHOP, with PAWN in the last. We can use some interface provided by the chess package to count the number of each type of the chess. For example we count the number of white PAWN as follows:

```
wp = len(self.board.pieces(chess.PAWN, chess.WHITE))
```

Similarly we can have the number of other types of chess and finally get our heuristic function where wp/bp stands for the number of white/black PAWN, wn/bn for the number of white/black KNIGHT, wb/bb for the number of white/black BISHOP, wr/br for the number of white/black ROCK, and bq/wq for the number of black/white QUEEN. We use the weighted values 1, 3, 3, 5, 9 for each of them respectively. We then check whether the current turn is BLACK or WHITE and we want to keep the higher heuristic number means the better state. Core code is shown below:

```
 if self.turn == chess.WHITE:
            material = 1 * (wp - bp) + 3 * (wn - bn) + 3 * (wb - bb) + 5 * (wr - br) + 9 *
(wq - bq)
        else:
            material = 1 * (bp - wp) + 3 * (bn - wn) + 3 * (bb - wb) + 5 * (br - wr) + 9 *
(bq - wq)
```

Thirdly, we use Alpha-Beta pruning method to make our original Minimax algorithm faster and so we can explore deeper depth in a given time. The core idea of Alpha-Beta pruning is using variable Alpha to memorize the MAX"s best option along the path from root and Beta to memorize the MIN's. We can then prove that as long as Alpha is greater than Beta there is no need for us to expand deeper from the current node and that is when we do the pruning. The challenge here is to use move-reordering method to improve our Alpha-Beta pruning. We want to improve the possibility to first expand those nodes which their possible final utilities are most likely to have greatest/smallest heuristic values. We use a very simple and normal move-reordering method to do so: by ordering all the legal unexpanded nodes of current depth based on  values calculated by the previous evalution function. However, since there is no direct relationship with the current node's evaluation value with its children's values, our move-reordering method may not work so obviously sometimes. Note that here we use **list** to save all the legal unexpanded nodes which are stored in tuple and we use the sort method provided by list to re-order shown as below:

```
 if maxnode == False:
           pairs.sort(
              key = lambda x : x[1] ,
              reverse = False
           )
      else:
          pairs.sort(
              key=lambda x: x[1],
              reverse=True
          )
```

Finally, to better fit our AI with the given time limit, we implement the iterative deepening algorithm which is mainly based on a *while* structure of the previous Alpha-Beta pruning algorithm. We add the max value of depth gradually and for each step calculate the best move result using Alpha-Beta pruning algorithm. Though ideally the deeper the result comes from ,the better the result could be, we still return each best move with its heuristic number and choose the best of those best moves based on the heuristic value.. We add the heurstic value as a property to our AlphaBetaAI instead of a return value of AlphaBetaAI's searching algorithm to minimize our code modificaiton.

## B. Evaluation

All of these algorithms work properly in general. Our iterative deepening algorithm is implemented based on our alphabeta method. As the result of our tests, all of them can beat the RandomAI for most of the time. However, when we add our max depth to 5 or larger, it will be obviously time confusing. Thus we test our algorithms mainly with a depth of 3 and 4. Besides, with the help of pruning process, alphabeta algorithm is much more faster than minimax method with a deeper depth. Since the evaluation function we provided is simple, our AI is not that kind of smart but can still win the RandomAI most of the time. Also since our move-reordering  method is simple, the Alpha-Beta puring may not be too faster than Minimax searching.

But note that there might be some difference in the results between alpha-beta and minimax algorithm with all the same given conditions.  We test our minimax algorithm and the alpha-beta pruning with the same given board and the same maximum depth the result may vary sometimes.  After checking the implementation of our alpha-beta method several times carefully, we believe both the different results given by alpha-beta and minimax are correct, and those different chosen results are with the same evaluation value from our evaluation function.

## C. Discussion

### 1. minimax and cutoff test

By varying the maximum depth, our minimax algorithm works fine when the maximum depth is below 3, but begins to be much slower from 4 and extremely slow over 5. We then keep track of the cost (number of calls for minimax algorithm during the iteractive process). We notice that when the depth is 1 the cost is in general under 100 times. As the depth increases, the cost increases appretenly, with in general under 1000 with the maximum depth 2, in general between 10,000 and 100,000 with the maximum depth 3 and over 200,000 while the maximum depth is 4.

### 2. evaluation function

We have discussed our evaluation function in the **Description** part. The core idea in general is based on the textbook: give each type of chess piece a weighted value and do the weighted linear sum of all chess pieces. Our evaluation function only counts the number of each type of chess piece without only consideration based on their locations. Obviously, their are many other more complex and effective methods like taking the position of each type of chess piece into account and this requires a deeper understanding of how to play chess.

By utilizing our evaluation function, our AI almost wins every time when playing with the given RandomAI. With the maximum depth keeps rising, it becomes extremely time confusing and we have discussed this in the previous part.

## 3. alpha-beta

We simply implement our move-reordering method by resorting the current legal unexplored nodes with their evaluation values, choosing the larger ones to explore first for max node and the lower ones to explore first for min node. Since the realtionship between the current node's evaluation value and the leaf node's evaluation value has no significant relationship, our move-reordering method works better for the deeper nodes and may not be so effective in most cases (compared with alphabeta method without using move-reordering method).

We have discussed in **Evaluation** part about why sometimes the result given by alpha-beta and minimax could be different with the same input. It is most likely that both results are equally valued by our evaluation function.

We test some cases using our alpha-beta method and minimax method seperately with the same given board, the same maximum depth. As for the the very beginning state of the board, both makes the same decision. With the maximum depth 3, Minimax AI costs around 9000 times to make the decision while alphabeta costs only around 600 times. It makes the searching process much faster and make the maximum of depth greater for a limited given time.

```
r . b q k b . r
p p p p . Q p p
. . n . . n . .
. . . . p . . .
. . B . P . . .
. . . . . . . .
P P P P . P P P
R N B . K . N R
AlphaBeta AI recommending move c4e6
The number of calls it made to alphabeta is 2

Minimax AI recommending move c4e6
The number of calls it made to minimax is 6412
```

For the status which can lead to winning of the game shown above, alpha-beta is much faster than minimax.

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
AlphaBeta AI recommending move g1h3
The number of calls it made to alphabeta is 581

Minimax AI recommending move g1h3
The number of calls it made to minimax is 9323
```

```
. . b . . . . .
p . N . . . N .
. . . . . . . p
. p . p . . . k
. . p . . p . .
. . . . . . . .
P P Q P P P P P
R . B . K B . R
AlphaBeta AI recommending move c2f5
The number of calls it made to alphabeta is 483

Minimax AI recommending move c2f5
The number of calls it made to minimax is 5718
```

## 4. iterative deepening

We have tested our iterative deepening in the game and with the board state becomes more complex compared to the beginning status, our iterative deepening algorithm (based on alphabeta algorithm) changes its decision as the depth increases more. Sometimes the final chosen move may not be the move made by the greatest maximum depth, similarly as the reason of moving difference between alphabeta and minimax we discussed before, those moves seem to result to the same evaluation value by our given evaluation function.

We show some of the cases which the decision changed as the maximum depth getting deeper below.

```
. . . . . k . N
. . . B . . p .
N . . . p . . n
```

```
p . . . . . . q
. . . . . p . .
. . . . P . . .
P P Q P . P P P
R . B . K . . R
----------------
a b c d e f g h

White to move

AlphaBeta AI recommending move d7e6
The number of calls it made to alphabeta is 46
AlphaBeta AI recommending move d7e6
The number of calls it made to alphabeta is 117
AlphaBeta AI recommending move e3f4
The number of calls it made to alphabeta is 3501
Interactive Deepening AI recommending move e3f4
```

```
. . . . . k . N
. . . B . . . .
N . . . p . . n
p . . . . . p .
. . . . . P . .
. . . . . P . .
P P Q P . P . P
R . B . K . . R
----------------
a b c d e f g h

White to move

AlphaBeta AI recommending move d7e6
The number of calls it made to alphabeta is 45
AlphaBeta AI recommending move f4g5
The number of calls it made to alphabeta is 106
AlphaBeta AI recommending move c2h7
The number of calls it made to alphabeta is 704
Interactive Deepening AI recommending move c2h7
```

```
. . r . . k n N
. . . b . . p .
N . . . p . . .
p p . . . p q .
. . . . . . . .
. . . . . . . .
```

```
P P P P P P P P
R . B Q K B . R
----------------
a b c d e f g h

White to move

AlphaBeta AI recommending move h8f7
The number of calls it made to alphabeta is 25
AlphaBeta AI recommending move h1g1
The number of calls it made to alphabeta is 118
AlphaBeta AI recommending move e2e3
The number of calls it made to alphabeta is 1426
Interactive Deepening AI recommending move e2e3
```