Breakthrough

Python 3 was used to implement this game.

A **configuration of the board** is a node in a search tree. Each node is implemented as a class "State."

State class contains members:

Member name:	Type:	Purpose:
Value	int	Stores Min or Max value of the State
White_positions	List of (int,int)	2-D coordinate of each white pieces
Black_positions	List of (int,int)	2-D coordinate of each black pieces
Subtree_size	Int	Number of nodes expanded, by the State
Extreme_child	Pointer to a State	Stores pointer to next optimal State

In our implementation, **White player is Max-player** and **black player is Min-player.**

Minimax search:

```
8   def Max_value(CurNode, heuristic, level, level_threshold):
9   
10      maxval= -infinity
11  
12      # For each White peice, consider all 3 moves
13      for x,y in CurNode.White_Positions:
14          #Consider all three positions
15          for position in {Left-Up, Right-up, Up}:
16              x_new = position.x
17              y_new = position.y
18  
19              if( (x_new,y_new) is legal move ):
20                  ChildA= Max_Child( x_new,y_new, CurNode, heuristic, "minimax", level, level_threshold )
21  
22                  if( ChildA.value > maxval):
23                      maxval= ChildA.value
24                      CurNode.Extreme_child= ChildA
25  
26      return maxval
```

Max_value(CurNode) calculates max value of CurNode. The heuristic used in Max is defined in parameter. Search depth is defined by "level_threshold". The parameter "level" keeps track of level.

Flow:

- For each white piece, each 3 moves are considered: Left-up, Right-up, and up (line 12-15). (The maximum possible number of moves is 16*3=48)
- If the considered move is legal, a child node is created (line 19 and 20.) The child node is created using helper function Max_Child(). (see below for pseudocode of Max_Child)
- Each time a child node is created, the child node's value is checked whether if it's the new maximum. The maximum valued child is stored in CurNode.Extreme_child (line 22 – 24.)

```
49 def Max_Child(x_new,y_new, CurNode, heuristic, searchtype, level, level_threshold, alpha, beta):
50 
51     # Create newChild (a new state of the board)
52     newChild = State(new_White_Positions, new_Black_Positions)
53 
54     # Calculate "Max value" of newChild
55     # if search haven't reached leaf node, call use Max() to calculate "Max value"
56     # If search has reached leaf node, use the heuristic function to calculate "Max value"
57     # Note: "level" increases by 1 when Max() is used.
58     if( level < level_threshold):
59 
60         if( searchtype=="alphabeta"):
61             newChild.value= AlphaBeta_Min_value(newChild, heuristic, level+1, level_threshold, alpha, beta)
62         else if( searchtype== "minimax"):
63             newChild.value= Min_value(newChild, heuristic, level+1, level_threshold)
64 
65     else if( level == level_threshold):
66         newChild.value = heuristic( newChild )
67 
68     return newChild
```

Max_Child() creates a child state and calculates its min value.

<u>Flow:</u>

- A node representing new state of the board, newChild, is created. In this state, the positions of white and black pieces are updated (line 52.) (details are trivial and omitted in pseudocode.)
- The newChild's min value is calculated by recursively calling Min() or using heuristic (if the node is a leaf) (line58-66.) When Min() is used, the level increases by 1. The newChild uses parent's alpha and beta value for minimum calculation, in Alpha-Beta search.
- The newChild is returned (line 68.)

```
def Min_value(CurNode, heuristic, level, level_threshold):
    minval = infinity

    # For each Black peice, consider all 3 moves
    for x,y in CurNode.Black_Positions:
        #Consider all three positions
        for position in {Left-down, Right-down, Down}
            x_new = position.x
            y_new = position.y

            if( x_new, y_new is a legal move ):
                ChildA = Min_Child(x_new,y_new, CurNode, heuristic, "minimax", level, level_threshold )

                if( ChildA.value < minval):
                    minval= ChildA.value
                    CurNode.Extreme_child= ChildA
    return minval
```

Min_value() follows similar flow as Max_value().

```
#level_threshold is depth of the search
def Min_Child( x_new,y_new, CurNode, heuristic, searchtype, level, level_threshold, alpha, beta):

    # Create newChild (a new state of the board)
    newChild = State(new_White_Positions, new_Black_Positions)

    # Calculate "Min value" of newChild
    # if search haven't reached leaf node, call use Mix() to calculate "Min value"
    # If search has reached leaf node, use the heuristic function to calculate "Min value"
    # Note: "level" increases by 1 when Min() is used.
    if( level < level_threshold):

        if(searchtype=="alphabeta"):
            newChild.value= AlphaBeta_Max_value(newChild, heuristic, level+1, level_threshold, alpha, beta)
        else if (searchtype =="minimax")
            newChild.value= Max_value(newChild, heuristic, level+1, level_threshold)

    else if(level == level_threshold):

        newChild.value = hueristic( newChild )

    return newChild
```

Min_Child() follows similar flow as Max_Child()

Alpha-Beta Search

```
75  def AlphaBeta_Max_value(CurNode, heuristic, level, level_threshold):
76
77      maxval= -infinity
78
79      # For each White peice, consider all 3 moves
80      for x,y in CurNode.White_Positions:
81          #Consider all three positions
82          for position in {Left-Up, Right-up, Up}:
83              x_new = position.x
84              y_new = position.y
85
86              if( (x_new,y_new) is legal move ):
87                  ChildA= Max_Child( x_new,y_new, CurNode, heuristic, "minimax", level, level_threshold )
88
89                  if( ChildA.value > maxval):
90                      maxval= ChildA.value
91                      CurNode.Extreme_child= ChildA
92
93                  if( maxval >=beta):
94                      return maxval #end the forloop.
95                  alpha= max( alpha, maxval)
96
97      return maxval
```

Psuedocode for AlphaBeta_MaxValue() is identical to Max_value(), except 3 additional lines at 93-95.

<u>Flow:</u>

- The search tree is stops creating branches (search is pruned) if maxval exceeds beta (beta is the highest maxval can be.) (line 93-94)
- Alpha value is maximized if higher child value is found (line 95.)

```
def AlphaBeta_Min_value(CurNode, heuristic, level, level_threshold):
    minval = infinity

    # For each Black peice, consider all 3 moves
    for x,y in CurNode.Black_Positions:
        #Consider all three positions
        for position in {Left-down, Right-down, Down}
            x_new = position.x
            y_new = position.y

            if( x_new, y_new is a legal move ):
                ChildA = Min_Child(x_new,y_new, CurNode, heuristic, "minimax", level, level_threshold )

                if( ChildA.value < minval):
                    minval= ChildA.value
                    CurNode.Extreme_child= ChildA

                if( minval <=alpha):
                    return minval
                beta = min(beta, minval)

    return minval
```

Follows similar flow as AlphaBeta_Max_value()

<u>Running a game</u>

```
122 def Game(searchtype_white, heuristic_white, searchtype_black, heuristic_black):
123     CurNode = Initial_Node()
124     player= "white"
125 
126     while( game_ended(CurNode) ==False ):
127         if( player=="white" ):
128 
129             if( searchtype_white=="alphabeta"):
130                 CurNode.value = AlphaBeta_Max_value(CurNode, heuristic_white, level=0, level_threshold=4, alpha= -inf, beta = inf)
131             else:
132                 CurNode.value = Max_value(CurNode, heuristic_white, level=0, level_threshold=3)
133 
134             # go to next state
135             CurNode = CurNode.Extreme_child
136             # switch player
137             player = "black"
138 
139         else if( player == "black")
140 
141             if(searchtype_black=="alphabeta"):
142                 CurNode.value = AlphaBeta_Min_value( CurNode, heuristic_black, level=0, level_threshold=4, alpha= -inf, beta = inf)
143             else:
144                 CurNode.value = Min_value(CurNode, heuristic_white,level=0, level_threshold=3)
145 
146             # go to next state
147             CurNode = CurNode.Extreme_child
148             # switch player
149             player = "white"
150     #end of while loop
151 
152     print_game_status(CurNode)
153 
154     return
```

In a game, white player finds optimal move using Max and black player find optimal move using Min.

<u>Flow:</u>

- A game start with all black pieces at row 1 and 2, and all white pieces at row 7 and 8 (line 123) and white player starts (line 124.)
- If the game has not ended, players alternate taking moves (line 126.)
- The player color is identified then search type is identified.
- In alpha-beta search, depth of search is 4 (level_threshold is 4) (line 130)
- After the search, state of the game is updated with optimal move (line 135) and the player is switched (line 137.)
- After game ends, the game status is printed.

Heuristic Functions:

Remark: White player chooses maximum heuristic value, while black player chooses minimum heuristic value.

In beginning of a game, black player's pieces start at row 1 and 2, and white player's pieces start at row 7 and 8.

Offensive 2:

$$Offensive2(Node) = 10000 - (\#\ of\ black\ pieces) + (\text{White Offensive value}) + random$$

Where:

$$White\ Offensive\ value = \sum_{i=1}^{8} weight_i * (\#\ of\ white\ pieces\ at\ row_i)$$

$$weight = (10, 5, 3, 1,1,1,1,1)$$

Offensive2 strategy has two goals: minimize (# of black pieces) and place white pieces close to row 1, 2, or 3.

"*White offensive value"* measures how good white position is. When white pieces are at row 1, 2, or 3, white player has good chance of winning. Therefore, "weight" have high value in row 1,2, and 3.

Defensive 2:

$$Defensive2\ (Node) = 10000 - (\ Black\ Threat\ value) + (White\ Defense\ value) + random$$

$$Black\ Threat\ value = \sum_{i=1}^{8} BTweight_i * (\#\ of\ black\ pieces\ at\ row_i)$$

$$White\ Defense\ value = \sum_{i=1}^{8} WDweight_i * (\#\ of\ white\ pieces\ at\ row_i)$$

$$BTweight = (1,1,1,1,1,1,5,10)$$

$$WDweight = (1,1,1,1,1,1,5,10)$$

Defensive2 strategy has two goals: minimize (Black Threat value) and maximize (White defense value.)

BTweight has highest values at row 7 and 8 because those are very threatening position to white player.

WDweight has highest values at row 7 or 8 because white pieces placed at row 7 or 8 forms good defense.

Result

1.White Minimax (Offensive Heuristic 1) vs Black Alpha-beta (Offensive Heuristic 1)

Shown below:

```
. . . . . . . .

. W B . . B . .

. B B B . B B .

. . . . . . . .

. . . . . . W .

. B . . W . . .

B . B . . W . .

. . . . . B W .

Number of White moves:  47
Number of Black moves:  47
White: Avg nodes expanded per move:  550.3404255319149
Black: Avg nodes expanded per move:  2716.0851063829787
White: Avg time per move:  0.3631872623524767
Black: Avg time per move:  0.7477707913581361
Number of White captured:  11
Number of Black captured:  5
Total time of Game:  52.215028524398804
Black Player wins!
```

2.White Alpha-beta (Offensive Heuristic 2) vs Black Alpha-beta (Defensive Heuristic 1)

Shown below:

```
. . . W B . B .

. . . . . . . .

. B . . . B . .

. B . . . . . .

. . . . . . . .

. . . . . . . .

. . . . W . W W

W W W W W W W W

Number of White moves:  25
Number of Black moves:  24
White: Avg nodes expanded per move:  2312.84
Black: Avg nodes expanded per move:  2659.4166666666665
White: Avg time per move:  0.5848392009735107
Black: Avg time per move:  0.7306151489416758
Number of White captured:  4
Number of Black captured:  11
Total time of Game:  32.15574359893799
White Player wins!
```

3. White Alpha-beta (Defensive Heuristic 2) vs Black Alpha-beta (Offensive Heuristic 1)

Shown below:

```
B B . B B . B W

. B B . B B . B

B . . . B . B W

B . B . . . . .

. . . . . . . .

. . . . . . . .

W W W W . W W .

W W W W W W W W

Number of White moves:  10
Number of Black moves:  9
White: Avg nodes expanded per move:  4514.6
Black: Avg nodes expanded per move:  4526.333333333333
White: Avg time per move:  1.0964405298233033
Black: Avg time per move:  1.7907399071587458
Number of White captured:  0
Number of Black captured:  1
Total time of Game:  27.081064462661743
White Player wins!
```

4.White Alpha-beta (Offensive Heuristic 2) vs Black Alpha-beta (Offensive Heuristic 1)

Shown below:

```
W . . . . . . .

. . . B . B B .

B . B . B B . .

B B . B . B . B

. . B . B B . B

. . . . . . . W

. . . . . . . .

W . W W W W W W

Number of White moves:  35
Number of Black moves:  34
White: Avg nodes expanded per move:  2881.114285714286
Black: Avg nodes expanded per move:  4194.911764705882
White: Avg time per move:  0.6622197832380022
Black: Avg time per move:  1.458213532672209
Number of White captured:  7
Number of Black captured:  0
Total time of Game:  72.75695252418518
White Player wins!
```

5.White Alpha-beta (Defensive Heuristic 2) vs Black Alpha-beta (Defensive Heuristic 1)

Shown below:

```
. . . B . . W .

B . . . . . B .

. . . B . . . .

. B . . . . . .

. B . . . . . .

. . . . . . . W

. . . W . W . .

W W W W W W W W

Number of White moves:  25
Number of Black moves:  24
White: Avg nodes expanded per move:  3964.72
Black: Avg nodes expanded per move:  3236.0833333333335
White: Avg time per move:  1.0269276046752929
Black: Avg time per move:  0.8496142228444418
Number of White captured:  4
Number of Black captured:  10
Total time of Game:  46.063931465148926
White Player wins!
```

6.White Alpha-beta (Offensive Heuristic 2) vs Black Alpha-beta (Defensive Heuristic 2)

Shown below:

```
. B B . . B . .

. B . . . . . B

. . . . B . . .

. B W B B . B B

. . . . . . . .

. . . . . W . .

. . . . . . . .

W . W . W . W B

Number of White moves:  41
Number of Black moves:  41
White: Avg nodes expanded per move:  3519.512195121951
Black: Avg nodes expanded per move:  4700.048780487805
White: Avg time per move:  0.9741740808254336
Black: Avg time per move:  1.760015144580748
Number of White captured:  10
Number of Black captured:  4
Total time of Game:  112.10175824165344
Black Player wins!
```