CS 6364 ARTIFICIAL INTELLIGENCE

PROGRAMMING PROJECT

Name: Yashwanth Devireddy

Net ID: YXD210008

PART I: MINIMAX

First Program: MiniMaxOpening

```
positions_evaluated = 0

# Static estimation function
# Calculates the static estimate value of the board position when the leaf nodes are reached
# Input: Board position is given as the input
# Output: A value is returned for the given board position
def static_estimation(b):
    global positions_evaluated
    positions_evaluated += 1
    numWhitePieces = 0
    numBlackPieces = 0
    for i in b:
        if i == "W":
            numWhitePieces += 1
        elif i == "B":
            numBlackPieces += 1
    return numWhitePieces - numBlackPieces

# closeMill function
# To check if the current move makes a mill
# Input: a location j in the array representing the board and the board b
# Output: returns True if the move to j closes a mill or else returns False
def closeMill(j, b):
    C = b[j]

    def zero():
        if (b[2] == b[4] == C):
            return True
        else:
            return False

    def one():
        if (b[3] == b[5] == C) or (b[8] == b[17] == C):
            return True
        else:
            return False

    def two():
        if (b[0] == b[4] == C):
            return True
        else:
            return False
```

```
def three():
    if (b[1] == b[5] == C) or (b[7] == b[14] == C):
        return True
    else:
        return False

def four():
    if (b[0] == b[2] == C):
        return True
    else:
        return False

def five():
    if (b[1] == b[3] == C) or (b[6] == b[11] == C):
        return True
    else:
        return False

def six():
    if (b[5] == b[11] == C) or (b[7] == b[8] == C):
        return True
    else:
        return False

def seven():
    if (b[3] == b[14] == C) or (b[6] == b[8] == C):
        return True
    else:
        return False

def eight():
    if (b[6] == b[7] == C) or (b[1] == b[17] == C):
        return True
    else:
        return False

def nine():
    if (b[10] == b[11] == C) or (b[12] == b[15] == C):
        return True
    else:
        return False

def ten():
    if (b[9] == b[11] == C) or (b[13] == b[16] == C):
        return True
```

```
    else:
        return False

def eleven():
    if (b[5] == b[6] == C) or (b[9] == b[10] == C) or (b[14] == b[17] == C):
        return True
    else:
        return False

def twelve():
    if (b[9] == b[15] == C) or (b[13] == b[14] == C):
        return True
    else:
        return False

def thirteen():
    if (b[12] == b[14] == C) or (b[10] == b[16] == C):
        return True
    else:
        return False

def fourteen():
    if (b[3] == b[7] == C) or (b[12] == b[13] == C) or (b[11] == b[17] == C):
        return True
    else:
        return False

def fifteen():
    if (b[16] == b[17] == C) or (b[12] == b[9] == C):
        return True
    else:
        return False

def sixteen():
    if (b[15] == b[17] == C) or (b[13] == b[10] == C):
        return True
    else:
        return False

def seventeen():
    if (b[1] == b[8] == C) or (b[15] == b[16] == C) or (b[14] == b[11] == C):
        return True
    else:
        return False

switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
```

```
        6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
        12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# generateRemove function
# Removes the Blackpieces from the given board which are not in the mill and adds that position to the list
# Input: a board position and a list L
# Output: positions are added to L by removing black pieces
# If no blackpieces can be removed the given board is added to the list
def generateRemove(b, L):
    a1 = len(L)
    for i in range(len(b)):
        if b[i] == "B":
            if not closeMill(i, b):
                b1 = b[:]
                b1[i] = 'x'
                L.append(b1)
    a2 = len(L)
    if a1 == a2:
        L.append(b)

# GenerateAdd function
# adds a whitepiece in available places and adds that position to the list
# Input: a board position
# Output: a list L of board positions
def GenerateAdd(b):
    L = []
    for i in range(len(b)):
        if b[i] == 'x':
            b1 = b[:]
            b1[i] = 'W'
            if closeMill(i, b1):
                generateRemove(b1, L)
            else:
                L.append(b1)
    return L

# WtB function
# Converts all the whitepieces to black and blackpieces to white
# Input: a board position
# Output: a board position where all the whites are swapped to black and vice-versa
def WtB(b):
    b1 = b[:]
    for i in range(len(b1)):
```

```
        if b1[i] == "W":
            b1[i] = "B"
        elif b1[i] == "B":
            b1[i] = "W"
    return b1

# leaf function
# Checks if the given board posiition is a leaf
# Input: a board position
# Output: returns True if the given board position is a leaf or else returns False
def leaf(b):
    numWhitepieces = 0
    numBlackpieces = 0
    for i in b:
        if i == 'W':
            numWhitepieces += 1
        elif i == 'B':
            numBlackpieces += 1
    if (numWhitepieces == 8 or numBlackpieces == 8):
        return True
    else:
        return False

# MinMax function
# Generates the moves for the current min node position and returns the minimum value for all the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: Generates the min node moves for the current board position by swapping the blacks and whites using WtB function
#        and compares the current minimum value with MaxMin value of all generated moves and returns the minimum value.
def MinMax(b, ply):
    if ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        b = WtB(b)
        v = 10000
        x = GenerateAdd(b)
        for i in x:
            i = WtB(i)
            v = min(v, MaxMin(i, ply))
        return v

# MaxMin function
```

```
# Generates the moves for the current max node position and returns the maximum value for all the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static estimation value of that board position
#        else  Generates the max node moves for the current board position and compares the current maximum value with MinMax value of all
#              generated moves and returns the maximum value. If the depth is 1, the board position for the max value is also returned.
def MaxMin(b, ply):
    if leaf(b) or ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        v = -10000
        y = GenerateAdd(b)
        x = 0
        for i in y:
            m = MinMax(i, ply)
            if m > v:
                x = i
                v = m
        if ply == 1:
            if x == 0:
                x = y[0]
            return x, v
        return v

# Input of the program
(f1, f2, depth) = list(input().split())
depth = int(depth)
file1 = open(f1)
l1 = list(file1.read())
s = ""
count = 0
for i in l1:
    s += i
    if i == 'W':
        count += 1

# Checks if the given depth is 0
if depth == 0:
    file2 = open(f2, "w")
    file2.write("No moves are calculated" + "\n\n" + "Board position is: " + s)

# Checks if the opening game is ended for the current player
```

```
elif count == 8:
    file2 = open(f2, "w")
    file2.write("No further moves are there" + "\n\n" + "The opening game is completed for the
player" + "\n\n" + "Board position is: " + s )

# If none of the above conditions are satisfied MaxMin algorithm is executed
else:
    (A1, A2) = MaxMin(l1, 0)
    s = ""
    for i in A1:
        s += i
    file2 = open(f2, "w")
    file2.write("Board position is: " + s + "\n\n" + "Positions evaluated by static estimation: " +
str(positions_evaluated) + "\n\n"
                + "MINIMAX estimate: " + str(A2))
```

Second Program: MiniMaxGame

```
positions_evaluated = 0

# Static estimation function
# Calculates the static estimate value of the board position when the leaf nodes are reached
# Input: Board position is given as the input
# Output: A value is returned for the given board position
def static_estimation(b):
    global positions_evaluated
    positions_evaluated += 1
    numWhitePieces = countofpieces(b, 'W')
    numBlackPieces = countofpieces(b, 'B')
    if numBlackPieces <= 2:
        return 10000
    elif numWhitePieces <= 2:
        return -10000
    elif numBlackMoves(b) == 0:
        return 10000
    else:
        return (1000 * (numWhitePieces - numBlackPieces)) - numBlackMoves(b)

# countofpieces function
# Counts the number of specified pieces (White or Black) in the given board position
# Input: a board position and a specific piece(black or white)
# Output: returns the number of specified pieces (White or Black) in the given board position
def countofpieces(b, x):
    count = 0
    for i in b:
        if i == x:
            count += 1
    return count

# numWhiteMoves function
# calculates the number of possible moves for the white for the given board
# Input: a board position
# Output: returns the number of moves possible for white for the given board position
def numWhiteMoves(l):
    n = GenerateMovesMidgameEndgame(l)
    if n == 3:
```

```
        x = GenerateHopping(l)
    else:
        x = GenerateMove(l)
    return len(x)

# numBlackMoves function
# calculates the number of possible moves for the black for the given board
# Input: a board position
# Output: returns the number of moves possible for black for the given board position
def numBlackMoves(l):
    ll = WtB(l)
    n = GenerateMovesMidgameEndgame(ll)
    if n == 3:
        x = GenerateHopping(ll)
    else:
        x = GenerateMove(ll)
    return len(x)

# neighbours function
# Returns the neighbours of the given location
# Input: a location j in the array representing the board
# Output: a list of locations in the array corresponding to j's neighbors
def neighbours(j):
    def zero():
        return [1, 2, 15]

    def one():
        return [0, 3, 8]

    def two():
        return [0, 3, 4, 12]

    def three():
        return [1, 2, 5, 7]

    def four():
        return [2, 5, 9]

    def five():
        return [3, 4, 6]
```

```
def six():
    return [5, 7, 11]

def seven():
    return [3, 6, 8, 14]

def eight():
    return [1, 7, 17]

def nine():
    return [4, 10, 12]

def ten():
    return [9, 11, 13]

def eleven():
    return [6, 10, 14]

def twelve():
    return [2, 9, 13, 15]

def thirteen():
    return [10, 12, 14, 16]

def fourteen():
    return [7, 11, 13, 17]

def fifteen():
    return [0, 12, 16]

def sixteen():
    return [13, 15, 17]

def seventeen():
    return [8, 14, 16]

switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
          6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
          12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}
```

```
    return switch[j]()

# closeMill function
# To check if the current move makes a mill
# Input: a location j in the array representing the board and the board b
# Output: returns True if the move to j closes a mill or else returns False
def closeMill(j, b):
    C = b[j]

    def zero():
        if (b[2] == b[4] == C):
            return True
        else:
            return False

    def one():
        if (b[3] == b[5] == C) or (b[8] == b[17] == C):
            return True
        else:
            return False

    def two():
        if (b[0] == b[4] == C):
            return True
        else:
            return False

    def three():
        if (b[1] == b[5] == C) or (b[7] == b[14] == C):
            return True
        else:
            return False

    def four():
        if (b[0] == b[2] == C):
            return True
        else:
            return False
```

```
def five():
    if (b[1] == b[3] == C) or (b[6] == b[11] == C):
        return True
    else:
        return False

def six():
    if (b[5] == b[11] == C) or (b[7] == b[8] == C):
        return True
    else:
        return False

def seven():
    if (b[3] == b[14] == C) or (b[6] == b[8] == C):
        return True
    else:
        return False

def eight():
    if (b[6] == b[7] == C) or (b[1] == b[17] == C):
        return True
    else:
        return False

def nine():
    if (b[10] == b[11] == C) or (b[12] == b[15] == C):
        return True
    else:
        return False

def ten():
    if (b[9] == b[11] == C) or (b[13] == b[16] == C):
        return True
    else:
        return False

def eleven():
    if (b[5] == b[6] == C) or (b[9] == b[10] == C) or (b[14] == b[17] == C):
        return True
    else:
```

```
        return False

def twelve():
    if (b[9] == b[15] == C) or (b[13] == b[14] == C):
        return True
    else:
        return False

def thirteen():
    if (b[12] == b[14] == C) or (b[10] == b[16] == C):
        return True
    else:
        return False

def fourteen():
    if (b[3] == b[7] == C) or (b[12] == b[13] == C) or (b[11] == b[17] == C):
        return True
    else:
        return False

def fifteen():
    if (b[16] == b[17] == C) or (b[12] == b[9] == C):
        return True
    else:
        return False

def sixteen():
    if (b[15] == b[17] == C) or (b[13] == b[10] == C):
        return True
    else:
        return False

def seventeen():
    if (b[1] == b[8] == C) or (b[15] == b[16] == C) or (b[14] == b[11] == C):
        return True
    else:
        return False

switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
          6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
```

```
        12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# generateRemove function
# Removes the Blackpieces from the given board which are not in the mill and adds that position
to the list
# Input: a board position and a list L
# Output: positions are added to L by removing black pieces
# If no blackpieces can be removed the given board is added to the list
def generateRemove(b, L):
    a1 = len(L)
    for i in range(len(b)):
        if b[i] == "B":
            if not closeMill(i, b):
                b1 = b[:]
                b1[i] = 'x'
                L.append(b1)
    a2 = len(L)
    if a1 == a2:
        L.append(b)

# GenerateMove function
# generates moves created by moving a white piece to an adjacent location
# Input: a board position
# Output: a list L of board positions
def GenerateMove(b):
    L = []
    for i in range(len(b)):
        if b[i] == "W":
            n = neighbours(i)
            for j in n:
                if b[j] == "x":
                    b1 = b[:]
                    b1[i] = "x"
                    b1[j] = "W"
                    if closeMill(j, b1):
                        generateRemove(b1, L)
                    else:
                        L.append(b1)
```

```
    return L

# GenerateHopping function
# generates moves created by hopping a white piece
# Input: a board position
# Output: a list L of board positions
def GenerateHopping(b):
    L = []
    for i in range(len(b)):
        if b[i] == "W":
            for j in range(len(b)):
                if b[j] == "x":
                    b1 = b[:]
                    b1[i] = "x"
                    b1[j] = "W"
                    if closeMill(j, b1):
                        generateRemove(b1, L)
                    else:
                        L.append(b1)
    return L

# GenerateMovesMidgameEndgame
# Determines whether to use GenerateMoves or GenerateHopping
# Input: a board position
# Output: the number of whitepieces
# if it is equal to 3 hopping is executeed or else GenerateMoves is executed in the MaxMin or MinMax function
def GenerateMovesMidgameEndgame(b):
    n = 0
    for i in b:
        if i == "W":
            n += 1
    return n

# WtB function
# Converts all the whitepieces to black and blackpieces to white
# Input: a board position
# Output: a board position where all the whites are swapped to black and vice-versa
def WtB(b):
    b1 = b[:]
```

```
    for i in range(len(b1)):
        if b1[i] == "W":
            b1[i] = "B"
        elif b1[i] == "B":
            b1[i] = "W"
    return b1

# leaf function
# Checks if the given board posiition is a leaf
# Input: a board position
# Output: returns True if the given board position is a leaf or else returns False
def leaf(b):
    numWpieces = countofpieces(b, 'W')
    numBpieces = countofpieces(b, 'B')
    if numWpieces < 3 or numBpieces < 3:
        return True
    else:
        return False

# MinMax function
# Generates the moves for the current min node position and returns the minimum value for all the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static estimation value of that board position
#         else  Generates the min node moves for the current board position and compares the current minimum value with MaxMin value of all
#               generated moves and returns the minimum value.
def MinMax(b, ply):
    if leaf(b) or ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        b = WtB(b)
        v = 10000
        n = GenerateMovesMidgameEndgame(b)
        if n == 3:
            x = GenerateHopping(b)
        else:
            x = GenerateMove(b)
```

```
        for i in x:
            i = WtB(i)
            v = min(v, MaxMin(i, ply))
        return v

# MaxMin function
# Generates the moves for the current max node position and returns the maximum value for all the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static estimation value of that board position
#         else  Generates the max node moves for the current board position and compares the current maximum value with MinMax value of all
#               generated moves and returns the maximum value. If the depth is 1, the board position for the max value is also returned.
def MaxMin(b, ply):
    if leaf(b) or ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        v = -10000
        n = GenerateMovesMidgameEndgame(b)
        if n == 3:
            y = GenerateHopping(b)
        else:
            y = GenerateMove(b)
        x = 0
        for i in y:
            m = MinMax(i, ply)
            if m > v:
                x = i
                v = m
        if ply == 1:
            if x == 0:
                x = y[0]
            return x, v
        return v

# Input of the program
(f1, f2, depth) = list(input().split())
```

```
depth = int(depth)
file1 = open(f1)
l1 = list(file1.read())
s = ""
Whitepieces = countofpieces(l1, 'W')
Blackpieces = countofpieces(l1, 'B')
for i in l1:
    s += i

# Checks if the given depth is 0
if depth == 0:
    file2 = open(f2, "w")
    file2.write("No moves are calculated" + "\n\n" + "Board position is: " + s)

# checks if the board position is valid
elif Whitepieces < 3 and Blackpieces < 3:
    file2 = open(f2, "w")
    file2.write("Not a vaid board position, PLEASE check it")

# checks if we have lost already
elif Whitepieces < 3 or numWhiteMoves(l1) == 0:
    file2 = open(f2, "w")
    file2.write("You have lost the game man, SORRY!!!" + "\n\n" + "Final Board position is: " +
s)

# checks if we have won already
elif Blackpieces < 3 or numBlackMoves(l1) == 0:
    file2 = open(f2, "w")
    file2.write("CONGRATULATIONS!!!, You have won the game" + "\n\n" + "Final Board
position is: " + s)

# If none of the above conditions are satisfied MaxMin algorithm is executed
else:
    (A1, A2) = MaxMin(l1, 0)
    s = ""
    for i in A1:
        s += i
    file2 = open(f2, "w")
    file2.write("Board position is: " + s + "\n\n" + "Positions evaluated by static estimation: " +
str(positions_evaluated) + "\n\n"
```

```
+ "MINIMAX estimate: " + str(A2))
```

PART II: ALPHA-BETA

Third Program: ABOpening

```
positions_evaluated = 0

# Static estimation function
# Calculates the static estimate value of the board position when the leaf nodes are reached
# Input: Board position is given as the input
# Output: A value is returned for the given board position
def static_estimation(b):
    global positions_evaluated
    positions_evaluated += 1
    numWhitePieces = 0
    numBlackPieces = 0
    for i in b:
        if i == "W":
            numWhitePieces += 1
        elif i == "B":
            numBlackPieces += 1
    return numWhitePieces - numBlackPieces

# closeMill function
# To check if the current move makes a mill
# Input: a location j in the array representing the board and the board b
# Output: returns True if the move to j closes a mill or else returns False
def closeMill(j, b):
    C = b[j]

    def zero():
        if (b[2] == b[4] == C):
            return True
        else:
            return False

    def one():
        if (b[3] == b[5] == C) or (b[8] == b[17] == C):
            return True
        else:
            return False

    def two():
        if (b[0] == b[4] == C):
```

```
        return True
    else:
        return False

def three():
    if (b[1] == b[5] == C) or (b[7] == b[14] == C):
        return True
    else:
        return False

def four():
    if (b[0] == b[2] == C):
        return True
    else:
        return False

def five():
    if (b[1] == b[3] == C) or (b[6] == b[11] == C):
        return True
    else:
        return False

def six():
    if (b[5] == b[11] == C) or (b[7] == b[8] == C):
        return True
    else:
        return False

def seven():
    if (b[3] == b[14] == C) or (b[6] == b[8] == C):
        return True
    else:
        return False

def eight():
    if (b[6] == b[7] == C) or (b[1] == b[17] == C):
        return True
    else:
        return False

def nine():
    if (b[10] == b[11] == C) or (b[12] == b[15] == C):
        return True
    else:
        return False
```

```
def ten():
    if (b[9] == b[11] == C) or (b[13] == b[16] == C):
        return True
    else:
        return False

def eleven():
    if (b[5] == b[6] == C) or (b[9] == b[10] == C) or (b[14] == b[17] == C):
        return True
    else:
        return False

def twelve():
    if (b[9] == b[15] == C) or (b[13] == b[14] == C):
        return True
    else:
        return False

def thirteen():
    if (b[12] == b[14] == C) or (b[10] == b[16] == C):
        return True
    else:
        return False

def fourteen():
    if (b[3] == b[7] == C) or (b[12] == b[13] == C) or (b[11] == b[17] == C):
        return True
    else:
        return False

def fifteen():
    if (b[16] == b[17] == C) or (b[12] == b[9] == C):
        return True
    else:
        return False

def sixteen():
    if (b[15] == b[17] == C) or (b[13] == b[10] == C):
        return True
    else:
        return False

def seventeen():
    if (b[1] == b[8] == C) or (b[15] == b[16] == C) or (b[14] == b[11] == C):
        return True
    else:
```

```
            return False

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# generateRemove function
# Removes the Blackpieces from the given board which are not in the mill and adds that position
to the list
# Input: a board position and a list L
# Output: positions are added to L by removing black pieces
# If no blackpieces can be removed the given board is added to the list
def generateRemove(b, L):
    a1 = len(L)
    for i in range(len(b)):
        if b[i] == "B":
            if not closeMill(i, b):
                b1 = b[:]
                b1[i] = 'x'
                L.append(b1)
    a2 = len(L)
    if a1 == a2:
        L.append(b)

# GenerateAdd function
# adds a whitepiece in available places and adds that position to the list
# Input: a board position
# Output: a list L of board positions
def GenerateAdd(b):
    L = []
    for i in range(len(b)):
        if b[i] == 'x':
            b1 = b[:]
            b1[i] = 'W'
            if closeMill(i, b1):
                generateRemove(b1, L)
            else:
                L.append(b1)
    return L

# WtB function
# Converts all the whitepieces to black and blackpieces to white
# Input: a board position
# Output: a board position where all the whites are swapped to black and vice-versa
```

```
def WtB(b):
    b1 = b[:]
    for i in range(len(b1)):
        if b1[i] == "W":
            b1[i] = "B"
        elif b1[i] == "B":
            b1[i] = "W"
    return b1

# leaf function
# Checks if the given board posiition is a leaf
# Input: a board position
# Output: returns True if the given board position is a leaf or else returns False
def leaf(b):
    numWhitepieces = 0
    numBlackpieces = 0
    for i in b:
        if i == 'W':
            numWhitepieces += 1
        elif i == 'B':
            numBlackpieces += 1
    if (numWhitepieces == 8 or numBlackpieces == 8):
        return True
    else:
        return False

# MinMax_alb function
# Generates the moves for the current min node position and returns the minimum value for all
the generated moves based on static estimation value
# Input: a board position, depth of the current node posiition, alpha and beta values
# Output: Generates the min node moves for the current board position by swapping the blacks
and whites using WtB function
#        and compares the current minimum value with MaxMin_alb value of all generated moves
and returns the minimum value.
def MinMax_alb(b, ply, al, beta):
    if ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        b = WtB(b)
        v = 10000
        x = GenerateAdd(b)
        for i in x:
            i = WtB(i)
            v = min(v, MaxMin_alb(i, ply, al, beta))
            if v <= al:
```

```
            return v
        else:
            beta = min(v, beta)
    return v
```

MaxMin_alb function
Generates the moves for the current max node position and returns the maximum value for all the generated moves based on static estimation value
Input: a board position, depth of the current node posiition, alpha nad beta values
Output: if a leaf node or a node at the maximum depth is reached it returns the static estimation value of that board position
else Generates the max node moves for the current board position and compares the current maximum value with MinMax_alb value of all
generated moves and returns the maximum value. If the depth is 1, the board position for the max value is also returned.

```
def MaxMin_alb(b, ply, al, beta):
  if leaf(b) or ply == depth:
    return static_estimation(b)
  else:
    ply += 1
    v = -10000
    y = GenerateAdd(b)
    x = 0
    for i in y:
      m = MinMax_alb(i, ply, al, beta)
      if m > v:
        x = i
        v = m
      if v >= beta:
        if ply == 1:
          return x, v
        else:
          return v
      else:
        al = max(v, al)
    if ply == 1:
      if x == 0:
        x = y[0]
      return x, v
    return v
```

```
# Input of the program
(f1, f2, depth) = list(input().split())
depth = int(depth)
file1 = open(f1)
l1 = list(file1.read())
```

```
s = ""
count = 0
for i in l1:
    s += i
    if i == 'W':
        count += 1

# Checks if the given depth is
if depth == 0:
    file2 = open(f2, "w")
    file2.write("No moves are calculated" + "\n\n" + "Board position is: " + s)

# Checks if the opening game is ended for the current player
elif count == 8:
    file2 = open(f2, "w")
    file2.write("No further moves are there" + "\n\n" + "The opening game is completed for the
player" + "\n\n" + "Board position is: " + s )

# If none of the above conditions are satisfied Alpha beta pruning algorithm is executed
else:
    (A1, A2) = MaxMin_alb(l1, 0, -10000, 10000)
    s = ""
    for i in A1:
        s += i
    file2 = open(f2, "w")
    file2.write("Board position is: " + s + "\n\n" + "Positions evaluated by static estimation: " +
str(positions_evaluated) + "\n\n"
                + "MINIMAX estimate: " + str(A2))
```

Fourth Program: ABGame

```
positions_evaluated = 0

# Static estimation function
# Calculates the static estimate value of the board position when the leaf nodes are reached
# Input: Board position is given as the input
# Output: A value is returned for the given board position
def static_estimation(b):
    global positions_evaluated
    positions_evaluated += 1
    numWhitePieces = countofpieces(b, 'W')
    numBlackPieces = countofpieces(b, 'B')
    if numBlackPieces <= 2:
        return 10000
    elif numWhitePieces <= 2:
        return -10000
    elif numBlackMoves(b) == 0:
        return 10000
    else:
        return (1000 * (numWhitePieces - numBlackPieces)) - numBlackMoves(b)

# countofpieces function
# Counts the number of specified pieces (White or Black) in the given board position
# Input: a board position and a specific piece(black or white)
# Output: returns the number of specified pieces (White or Black) in the given board position
def countofpieces(b, x):
    count = 0
    for i in b:
        if i == x:
            count += 1
    return count

# numWhiteMoves function
# calculates the number of possible moves for the white for the given board
# Input: a board position
# Output: returns the number of moves possible for white for the given board position
def numWhiteMoves(l):
    n = GenerateMovesMidgameEndgame(l)
    if n == 3:
        x = GenerateHopping(l)
    else:
        x = GenerateMove(l)
    return len(x)

# numBlackMoves function
```

```
# calculates the number of possible moves for the black for the given board
# Input: a board position
# Output: returns the number of moves possible for black for the given board position
def numBlackMoves(l):
    ll = WtB(l)
    n = GenerateMovesMidgameEndgame(ll)
    if n == 3:
        x = GenerateHopping(ll)
    else:
        x = GenerateMove(ll)
    return len(x)

# neighbours function
# Returns the neighbours of the given location
# Input: a location j in the array representing the board
# Output: a list of locations in the array corresponding to j's neighbors
def neighbours(j):
    def zero():
        return [1, 2, 15]

    def one():
        return [0, 3, 8]

    def two():
        return [0, 3, 4, 12]

    def three():
        return [1, 2, 5, 7]

    def four():
        return [2, 5, 9]

    def five():
        return [3, 4, 6]

    def six():
        return [5, 7, 11]

    def seven():
        return [3, 6, 8, 14]

    def eight():
        return [1, 7, 17]

    def nine():
        return [4, 10, 12]
```

```
    def ten():
        return [9, 11, 13]

    def eleven():
        return [6, 10, 14]

    def twelve():
        return [2, 9, 13, 15]

    def thirteen():
        return [10, 12, 14, 16]

    def fourteen():
        return [7, 11, 13, 17]

    def fifteen():
        return [0, 12, 16]

    def sixteen():
        return [13, 15, 17]

    def seventeen():
        return [8, 14, 16]

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# closeMill function
# To check if the current move makes a mill
# Input: a location j in the array representing the board and the board b
# Output: returns True if the move to j closes a mill or else returns False
def closeMill(j, b):
    C = b[j]

    def zero():
        if (b[2] == b[4] == C):
            return True
        else:
            return False

    def one():
        if (b[3] == b[5] == C) or (b[8] == b[17] == C):
```

```
        return True
    else:
        return False

def two():
    if (b[0] == b[4] == C):
        return True
    else:
        return False

def three():
    if (b[1] == b[5] == C) or (b[7] == b[14] == C):
        return True
    else:
        return False

def four():
    if (b[0] == b[2] == C):
        return True
    else:
        return False

def five():
    if (b[1] == b[3] == C) or (b[6] == b[11] == C):
        return True
    else:
        return False

def six():
    if (b[5] == b[11] == C) or (b[7] == b[8] == C):
        return True
    else:
        return False

def seven():
    if (b[3] == b[14] == C) or (b[6] == b[8] == C):
        return True
    else:
        return False

def eight():
    if (b[6] == b[7] == C) or (b[1] == b[17] == C):
        return True
    else:
        return False
```

```
def nine():
    if (b[10] == b[11] == C) or (b[12] == b[15] == C):
        return True
    else:
        return False

def ten():
    if (b[9] == b[11] == C) or (b[13] == b[16] == C):
        return True
    else:
        return False

def eleven():
    if (b[5] == b[6] == C) or (b[9] == b[10] == C) or (b[14] == b[17] == C):
        return True
    else:
        return False

def twelve():
    if (b[9] == b[15] == C) or (b[13] == b[14] == C):
        return True
    else:
        return False

def thirteen():
    if (b[12] == b[14] == C) or (b[10] == b[16] == C):
        return True
    else:
        return False

def fourteen():
    if (b[3] == b[7] == C) or (b[12] == b[13] == C) or (b[11] == b[17] == C):
        return True
    else:
        return False

def fifteen():
    if (b[16] == b[17] == C) or (b[12] == b[9] == C):
        return True
    else:
        return False

def sixteen():
    if (b[15] == b[17] == C) or (b[13] == b[10] == C):
        return True
    else:
```

```
            return False

    def seventeen():
        if (b[1] == b[8] == C) or (b[15] == b[16] == C) or (b[14] == b[11] == C):
            return True
        else:
            return False

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# generateRemove function
# Removes the Blackpieces from the given board which are not in the mill and adds that position
to the list
# Input: a board position and a list L
# Output: positions are added to L by removing black pieces
# If no blackpieces can be removed the given board is added to the list
def generateRemove(b, L):
    a1 = len(L)
    for i in range(len(b)):
        if b[i] == "B":
            if not closeMill(i, b):
                b1 = b[:]
                b1[i] = 'x'
                L.append(b1)
    a2 = len(L)
    if a1 == a2:
        L.append(b)

# GenerateMove function
# generates moves created by moving a white piece to an adjacent location
# Input: a board position
# Output: a list L of board positions
def GenerateMove(b):
    L = []
    for i in range(len(b)):
        if b[i] == "W":
            n = neighbours(i)
            for j in n:
                if b[j] == "x":
                    b1 = b[:]
                    b1[i] = "x"
                    b1[j] = "W"
```

```
            if closeMill(j, b1):
                generateRemove(b1, L)
            else:
                L.append(b1)
    return L

# GenerateHopping function
# generates moves created by hopping a white piece
# Input: a board position
# Output: a list L of board positions
def GenerateHopping(b):
    L = []
    for i in range(len(b)):
        if b[i] == "W":
            for j in range(len(b)):
                if b[j] == "x":
                    b1 = b[:]
                    b1[i] = "x"
                    b1[j] = "W"
                    if closeMill(j, b1):
                        generateRemove(b1, L)
                    else:
                        L.append(b1)
    return L

# GenerateMovesMidgameEndgame
# Determines whether to use GenerateMoves or GenerateHopping
# Input: a board position
# Output: the number of whitepieces
# if it is equal to 3 hopping is executeed or else GenerateMoves is executed in the MaxMin-alb
or MinMax_alb function
def GenerateMovesMidgameEndgame(b):
    n = 0
    for i in b:
        if i == "W":
            n += 1
    return n

# WtB function
# Converts all the whitepieces to black and blackpieces to white
# Input: a board position
# Output: a board position where all the whites are swapped to black and vice-versa
def WtB(b):
    b1 = b[:]
    for i in range(len(b1)):
        if b1[i] == "W":
```

```
            b1[i] = "B"
        elif b1[i] == "B":
            b1[i] = "W"
    return b1

# leaf function
# Checks if the given board posiition is a leaf
# Input: a board position
# Output: returns True if the given board position is a leaf or else returns False
def leaf(b):
    numWpieces = countofpieces(b, 'W')
    numBpieces = countofpieces(b, 'B')
    if numWpieces < 3 or numBpieces < 3:
        return True
    else:
        return False

# MinMax_alb function
# Generates the moves for the current min node position and returns the minimum value for all the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static estimation value of that board position
#         else  Generates the min node moves for the current board position and compares the current minimum value with MaxMin_alb value of all
#               generated moves and returns the minimum value.
def MinMax_alb(b, ply, al, beta):
    if leaf(b) or ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        b = WtB(b)
        v = 10000
        n = GenerateMovesMidgameEndgame(b)
        if n == 3:
            x = GenerateHopping(b)
        else:
            x = GenerateMove(b)
        for i in x:
            i = WtB(i)
            v = min(v, MaxMin_alb(i, ply, al, beta))
            if v <= al:
                return v
            else:
                beta = min(v, beta)
        return v
```

```
# MaxMin_alb function
# Generates the moves for the current max node position and returns the maximum value for all the generated moves based on static estimation value
# Input: a board position, depth of the current node posiition, alpha nad beta values
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static estimation value of that board position
#         else  Generates the max node moves for the current board position and compares the current maximum value with MinMax_alb value of all
#               generated moves and returns the maximum value. If the depth is 1, the board position for the max value is also returned.
def MaxMin_alb(b, ply, al, beta):
    if leaf(b) or ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        v = -10000
        n = GenerateMovesMidgameEndgame(b)
        if n == 3:
            y = GenerateHopping(b)
        else:
            y = GenerateMove(b)
        x = 0
        for i in y:
            m = MinMax_alb(i, ply, al, beta)
            if m > v:
                x = i
                v = m
            if v >= beta:
                if ply == 1:
                    return x, v
                else:
                    return v
            else:
                al = max(v, al)
        if ply == 1:
            if x == 0:
                x = y[0]
            return x, v
        return v

# Input of the program
(f1, f2, depth) = list(input().split())
depth = int(depth)
file1 = open(f1)
l1 = list(file1.read())
```

```
s = ""
Whitepieces = countofpieces(l1, 'W')
Blackpieces = countofpieces(l1, 'B')
for i in l1:
    s += i

# Checks if the given depth is
if depth == 0:
    file2 = open(f2, "w")
    file2.write("No moves are calculated" + "\n\n" + "Board position is: " + s)

# checks if the board position is valid
elif Whitepieces < 3 and Blackpieces < 3:
    file2 = open(f2, "w")
    file2.write("Not a vaid board position, PLEASE check it")

# checks if we have lost already
elif Whitepieces < 3 or numWhiteMoves(l1) == 0:
    file2 = open(f2, "w")
    file2.write("You have lost the game man, SORRY!!!" + "\n\n" + "Final Board position is: " +
s)

# checks if we have won already
elif Blackpieces < 3 or numBlackMoves(l1) == 0:
    file2 = open(f2, "w")
    file2.write("CONGRATULATIONS!!!, You have won the game" + "\n\n" + "Final Board
position is: " + s)

# If none of the above conditions are satisfied Alpha beta pruning algorithm is executed
else:
    (A1, A2) = MaxMin_alb(l1, 0, -10000, 10000)
    s = ""
    for i in A1:
        s += i
    file2 = open(f2, "w")
    file2.write("Board position is: " + s + "\n\n" + "Positions evaluated by static estimation: " +
str(positions_evaluated) + "\n\n"
            + "MINIMAX estimate: " + str(A2))
```

PART III: PLAY A GAME FOR BLACK

Fifth Program: MiniMaxOpeningBlack

```
positions_evaluated = 0

# Static estimation function
# Calculates the static estimate value of the board position when the leaf nodes are reached
# Input: Board position is given as the input
# Output: A value is returned for the given board position
def static_estimation(b):
    global positions_evaluated
    positions_evaluated += 1
    numWhitePieces = 0
    numBlackPieces = 0
    for i in b:
        if i == "W":
            numWhitePieces += 1
        elif i == "B":
            numBlackPieces += 1
    return numWhitePieces - numBlackPieces

# closeMill function
# To check if the current move makes a mill
# Input: a location j in the array representing the board and the board b
# Output: returns True if the move to j closes a mill or else returns False
def closeMill(j, b):
    C = b[j]

    def zero():
        if (b[2] == b[4] == C):
            return True
        else:
            return False

    def one():
        if (b[3] == b[5] == C) or (b[8] == b[17] == C):
            return True
        else:
            return False

    def two():
        if (b[0] == b[4] == C):
            return True
        else:
```

```
        return False

def three():
    if (b[1] == b[5] == C) or (b[7] == b[14] == C):
        return True
    else:
        return False

def four():
    if (b[0] == b[2] == C):
        return True
    else:
        return False

def five():
    if (b[1] == b[3] == C) or (b[6] == b[11] == C):
        return True
    else:
        return False

def six():
    if (b[5] == b[11] == C) or (b[7] == b[8] == C):
        return True
    else:
        return False

def seven():
    if (b[3] == b[14] == C) or (b[6] == b[8] == C):
        return True
    else:
        return False

def eight():
    if (b[6] == b[7] == C) or (b[1] == b[17] == C):
        return True
    else:
        return False

def nine():
    if (b[10] == b[11] == C) or (b[12] == b[15] == C):
        return True
    else:
        return False

def ten():
    if (b[9] == b[11] == C) or (b[13] == b[16] == C):
```

```
        return True
    else:
        return False

def eleven():
    if (b[5] == b[6] == C) or (b[9] == b[10] == C) or (b[14] == b[17] == C):
        return True
    else:
        return False

def twelve():
    if (b[9] == b[15] == C) or (b[13] == b[14] == C):
        return True
    else:
        return False

def thirteen():
    if (b[12] == b[14] == C) or (b[10] == b[16] == C):
        return True
    else:
        return False

def fourteen():
    if (b[3] == b[7] == C) or (b[12] == b[13] == C) or (b[11] == b[17] == C):
        return True
    else:
        return False

def fifteen():
    if (b[16] == b[17] == C) or (b[12] == b[9] == C):
        return True
    else:
        return False

def sixteen():
    if (b[15] == b[17] == C) or (b[13] == b[10] == C):
        return True
    else:
        return False

def seventeen():
    if (b[1] == b[8] == C) or (b[15] == b[16] == C) or (b[14] == b[11] == C):
        return True
    else:
        return False
```

```
    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# generateRemove function
# Removes the Blackpieces from the given board which are not in the mill and adds that position
to the list
# Input: a board position and a list L
# Output: positions are added to L by removing black pieces
# If no blackpieces can be removed the given board is added to the list
def generateRemove(b, L):
    a1 = len(L)
    for i in range(len(b)):
        if b[i] == "B":
            if not closeMill(i, b):
                b1 = b[:]
                b1[i] = 'x'
                L.append(b1)
    a2 = len(L)
    if a1 == a2:
        L.append(b)

# GenerateAdd function
# adds a whitepiece in available places and adds that position to the list
# Input: a board position
# Output: a list L of board positions
def GenerateAdd(b):
    L = []
    for i in range(len(b)):
        if b[i] == 'x':
            b1 = b[:]
            b1[i] = 'W'
            if closeMill(i, b1):
                generateRemove(b1, L)
            else:
                L.append(b1)
    return L

# WtB function
# Converts all the whitepieces to black and blackpieces to white
# Input: a board position
# Output: a board position where all the whites are swapped to black and vice-versa
def WtB(b):
    b1 = b[:]
```

```
    for i in range(len(b1)):
        if b1[i] == "W":
            b1[i] = "B"
        elif b1[i] == "B":
            b1[i] = "W"
    return b1

# leaf function
# Checks if the given board posiition is a leaf
# Input: a board position
# Output: returns True if the given board position is a leaf or else returns False
def leaf(b):
    numWhitepieces = 0
    numBlackpieces = 0
    for i in b:
        if i == 'W':
            numWhitepieces += 1
        elif i == 'B':
            numBlackpieces += 1
    if (numWhitepieces == 8 or numBlackpieces == 8):
        return True
    else:
        return False

# MinMax function
# Generates the moves for the current min node position and returns the minimum value for all
the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static
estimation value of that board position
#         else  Generates the min node moves for the current board position and compares the
current minimum value with MaxMin value of all
#               generated moves and returns the minimum value.
def MinMax(b, ply):
    if ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        b = WtB(b)
        v = 10000
        x = GenerateAdd(b)
        for i in x:
            i = WtB(i)
            v = min(v, MaxMin(i, ply))
        return v
```

```
# MaxMin function
# Generates the moves for the current max node position and returns the maximum value for all
the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: Generates the max node moves for the current board position and compares the current
maximum value with MinMax value of all
#       generated moves and returns the maximum value. If the depth is 1, the board position for
the max value is also returned.
def MaxMin(b, ply):
    if leaf(b) orply == depth:
        return static_estimation(b)
    else:
        ply += 1
        v = -10000
        y = GenerateAdd(b)
        x = 0
        for i in y:
            m = MinMax(i, ply)
            if m > v:
                x = i
                v = m
        if ply == 1:
            if x == 0:
                x = y[0]
            return x, v
        return v

# Input of the program
(f1, f2, depth) = list(input().split())
depth = int(depth)
file1 = open(f1)
l1 = list(file1.read())

# In the given board position all the whitepieces and blackpieces are swapped
# The Minimax algorithm is performed on the swapped board
l1 = WtB(l1)
s = ""
count = 0
for i in l1:
    s += i
    if i == 'W':
        count += 1

# Checks if the given depth is 0
if depth == 0:
    file2 = open(f2, "w")
```

```
    file2.write("No moves are calculated" + "\n\n" + "Board position is: " + s)

# Checks if the opening game is ended for the current player
elif count == 8:
    file2 = open(f2, "w")
    file2.write("No further moves are there" + "\n\n" + "The opening game is completed for the
player" + "\n\n" + "Board position is: " + s )

# If none of the above conditions are satisfied MaxMin algorithm is executed
else:
    (A1, A2) = MaxMin(l1, 0)

# The final board position is swapped again
    A1 = WtB(A1)
    s = ""
    for i in A1:
        s += i
    file2 = open(f2, "w")
    file2.write("Board position is: " + s + "\n\n" + "Positions evaluated by static estimation: " +
str(positions_evaluated) + "\n\n"
            + "MINIMAX estimate: " + str(A2))
```

Sixth Program: MiniMaxGameBlack

```
positions_evaluated = 0

# Static estimation function
# Calculates the static estimate value of the board position when the leaf nodes are reached
# Input: Board position is given as the input
# Output: A value is returned for the given board position
def static_estimation(b):
    global positions_evaluated
    positions_evaluated += 1
    numWhitePieces = countofpieces(b, 'W')
    numBlackPieces = countofpieces(b, 'B')
    if numBlackPieces <= 2:
        return 10000
    elif numWhitePieces <= 2:
        return -10000
    elif numBlackMoves(b) == 0:
        return 10000
    else:
        return (1000 * (numWhitePieces - numBlackPieces)) - numBlackMoves(b)

# countofpieces function
# Counts the number of specified pieces (White or Black) in the given board position
# Input: a board position and a specific piece(black or white)
# Output: returns the number of specified pieces (White or Black) in the given board position
def countofpieces(b, x):
    count = 0
    for i in b:
        if i == x:
            count += 1
    return count

# numWhiteMoves function
# calculates the number of possible moves for the white for the given board
# Input: a board position
# Output: returns the number of moves possible for white for the given board position
def numWhiteMoves(l):
    n = GenerateMovesMidgameEndgame(l)
    if n == 3:
        x = GenerateHopping(l)
    else:
        x = GenerateMove(l)
    return len(x)

# numBlackMoves function
```

```
# calculates the number of possible moves for the black for the given board
# Input: a board position
# Output: returns the number of moves possible for black for the given board position
def numBlackMoves(l):
    ll = WtB(l)
    n = GenerateMovesMidgameEndgame(ll)
    if n == 3:
        x = GenerateHopping(ll)
    else:
        x = GenerateMove(ll)
    return len(x)

# neighbours function
# Returns the neighbours of the given location
# Input: a location j in the array representing the board
# Output: a list of locations in the array corresponding to j's neighbors
def neighbours(j):
    def zero():
        return [1, 2, 15]

    def one():
        return [0, 3, 8]

    def two():
        return [0, 3, 4, 12]

    def three():
        return [1, 2, 5, 7]

    def four():
        return [2, 5, 9]

    def five():
        return [3, 4, 6]

    def six():
        return [5, 7, 11]

    def seven():
        return [3, 6, 8, 14]

    def eight():
        return [1, 7, 17]

    def nine():
        return [4, 10, 12]
```

```
    def ten():
        return [9, 11, 13]

    def eleven():
        return [6, 10, 14]

    def twelve():
        return [2, 9, 13, 15]

    def thirteen():
        return [10, 12, 14, 16]

    def fourteen():
        return [7, 11, 13, 17]

    def fifteen():
        return [0, 12, 16]

    def sixteen():
        return [13, 15, 17]

    def seventeen():
        return [8, 14, 16]

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# closeMill function
# To check if the current move makes a mill
# Input: a location j in the array representing the board and the board b
# Output: returns True if the move to j closes a mill or else returns False
def closeMill(j, b):
    C = b[j]

    def zero():
        if (b[2] == b[4] == C):
            return True
        else:
            return False

    def one():
        if (b[3] == b[5] == C) or (b[8] == b[17] == C):
```

```
        return True
    else:
        return False

def two():
    if (b[0] == b[4] == C):
        return True
    else:
        return False

def three():
    if (b[1] == b[5] == C) or (b[7] == b[14] == C):
        return True
    else:
        return False

def four():
    if (b[0] == b[2] == C):
        return True
    else:
        return False

def five():
    if (b[1] == b[3] == C) or (b[6] == b[11] == C):
        return True
    else:
        return False

def six():
    if (b[5] == b[11] == C) or (b[7] == b[8] == C):
        return True
    else:
        return False

def seven():
    if (b[3] == b[14] == C) or (b[6] == b[8] == C):
        return True
    else:
        return False

def eight():
    if (b[6] == b[7] == C) or (b[1] == b[17] == C):
        return True
    else:
        return False
```

```
def nine():
    if (b[10] == b[11] == C) or (b[12] == b[15] == C):
        return True
    else:
        return False

def ten():
    if (b[9] == b[11] == C) or (b[13] == b[16] == C):
        return True
    else:
        return False

def eleven():
    if (b[5] == b[6] == C) or (b[9] == b[10] == C) or (b[14] == b[17] == C):
        return True
    else:
        return False

def twelve():
    if (b[9] == b[15] == C) or (b[13] == b[14] == C):
        return True
    else:
        return False

def thirteen():
    if (b[12] == b[14] == C) or (b[10] == b[16] == C):
        return True
    else:
        return False

def fourteen():
    if (b[3] == b[7] == C) or (b[12] == b[13] == C) or (b[11] == b[17] == C):
        return True
    else:
        return False

def fifteen():
    if (b[16] == b[17] == C) or (b[12] == b[9] == C):
        return True
    else:
        return False

def sixteen():
    if (b[15] == b[17] == C) or (b[13] == b[10] == C):
        return True
    else:
```

```
        return False

    def seventeen():
        if (b[1] == b[8] == C) or (b[15] == b[16] == C) or (b[14] == b[11] == C):
            return True
        else:
            return False

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# generateRemove function
# Removes the Blackpieces from the given board which are not in the mill and adds that position
to the list
# Input: a board position and a list L
# Output: positions are added to L by removing black pieces
# If no blackpieces can be removed the given board is added to the list
def generateRemove(b, L):
    a1 = len(L)
    for i in range(len(b)):
        if b[i] == "B":
            if not closeMill(i, b):
                b1 = b[:]
                b1[i] = 'x'
                L.append(b1)
    a2 = len(L)
    if a1 == a2:
        L.append(b)

# GenerateMove function
# generates moves created by moving a white piece to an adjacent location
# Input: a board position
# Output: a list L of board positions
def GenerateMove(b):
    L = []
    for i in range(len(b)):
        if b[i] == "W":
            n = neighbours(i)
            for j in n:
                if b[j] == "x":
                    b1 = b[:]
                    b1[i] = "x"
                    b1[j] = "W"
```

```
                if closeMill(j, b1):
                    generateRemove(b1, L)
                else:
                    L.append(b1)
    return L

# GenerateHopping function
# generates moves created by hopping a white piece
# Input: a board position
# Output: a list L of board positions
def GenerateHopping(b):
    L = []
    for i in range(len(b)):
        if b[i] == "W":
            for j in range(len(b)):
                if b[j] == "x":
                    b1 = b[:]
                    b1[i] = "x"
                    b1[j] = "W"
                    if closeMill(j, b1):
                        generateRemove(b1, L)
                    else:
                        L.append(b1)
    return L

# GenerateMovesMidgameEndgame
# Determines whether to use GenerateMoves or GenerateHopping
# Input: a board position
# Output: the number of whitepieces
# if it is equal to 3 hopping is executeed or else GenerateMoves is executed in the MaxMin or
MinMax function
def GenerateMovesMidgameEndgame(b):
    n = 0
    for i in b:
        if i == "W":
            n += 1
    return n

# WtB function
# Converts all the whitepieces to black and blackpieces to white
# Input: a board position
# Output: a board position where all the whites are swapped to black and vice-versa
def WtB(b):
    b1 = b[:]
    for i in range(len(b1)):
        if b1[i] == "W":
```

```
            b1[i] = "B"
        elif b1[i] == "B":
            b1[i] = "W"
    return b1

# leaf function
# Checks if the given board posiition is a leaf
# Input: a board position
# Output: returns True if the given board position is a leaf or else returns False
def leaf(b):
    numWpieces = countofpieces(b, 'W')
    numBpieces = countofpieces(b, 'B')
    if numWpieces < 3 or numBpieces < 3:
        return True
    else:
        return False

# MinMax function
# Generates the moves for the current min node position and returns the minimum value for all the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static estimation value of that board position
#         else  Generates the min node moves for the current board position and compares the current minimum value with MaxMin value of all
#               generated moves and returns the minimum value.
def MinMax(b, ply):
    if leaf (b) or ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        b = WtB(b)
        v = 10000
        n = GenerateMovesMidgameEndgame(b)
        if n == 3:
            x = GenerateHopping(b)
        else:
            x = GenerateMove(b)
        for i in x:
            i = WtB(i)
            v = min(v, MaxMin(i, ply))
        return v

# MaxMin function
# Generates the moves for the current max node position and returns the maximum value for all the generated moves based on static estimation value
```

```
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static
estimation value of that board position
#         else  Generates the max node moves for the current board position and compares the
current maximum value with MinMax value of all
#              generated moves and returns the maximum value. If the depth is 1, the board position
for the max value is also returned.
def MaxMin(b, ply):
    if leaf(b) or ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        v = -10000
        n = GenerateMovesMidgameEndgame(b)
        if n == 3:
            y = GenerateHopping(b)
        else:
            y = GenerateMove(b)
        x = 0
        for i in y:
            m = MinMax(i, ply)
            if m > v:
                x = i
                v = m
        if ply == 1:
            if x == 0:
                x = y[0]
            return x, v
        return v

# Input of the program
(f1, f2, depth) = list(input().split())
depth = int(depth)
file1 = open(f1)
l1 = list(file1.read())

# In the given board position all the whitepieces and blackpieces are swapped
# The Minimax algorithm is performed on the swapped board
l1 = WtB(l1)
s = ""
Whitepieces = countofpieces(l1, 'W')
Blackpieces = countofpieces(l1, 'B')
for i in l1:
    s += i

# Checks if the given depth is 0
```

```
if depth == 0:
   file2 = open(f2, "w")
   file2.write("No moves are calculated" + "\n\n" + "Board position is: " + s)

# checks if the board position is valid
elif Whitepieces < 3 and Blackpieces < 3:
   file2 = open(f2, "w")
   file2.write("Not a vaid board position, PLEASE check it")

# checks if we have lost already
elif Whitepieces < 3 or numWhiteMoves(l1) == 0:
   file2 = open(f2, "w")
   file2.write("You have lost the game man, SORRY!!!" + "\n\n" + "Final Board position is: " +
s)

# checks if we have won already
elif Blackpieces < 3 or numBlackMoves(l1) == 0:
   file2 = open(f2, "w")
   file2.write("CONGRATULATIONS!!!, You have won the game" + "\n\n" + "Final Board
position is: " + s)

# If none of the above conditions are satisfied MaxMin algorithm is executed
else:
   (A1, A2) = MaxMin(l1, 0)

# The final board position is swapped again
   A1 = WtB(A1)
   s = ""
   for i in A1:
      s += i
   file2 = open(f2, "w")
   file2.write("Board position is: " + s + "\n\n" + "Positions evaluated by static estimation: " +
str(positions_evaluated) + "\n\n"
            + "MINIMAX estimate: " + str(A2))
```

PART IV: STATIC ESTIMATION

Seventh Program: MiniMaxOpeningImproved

```
positions_evaluated = 0

# Static estimation function
# Calculates the static estimate value of the board position when the leaf nodes are reached
# Input: Board position is given as the input
# Output: A value is returned for the given board position
def static_estimation(b):
    global positions_evaluated
    positions_evaluated += 1
    possiblemill = 0
    numWhitePieces = 0
    numBlackPieces = 0
    numWhiteMills = 0
    numBlackMills = 0
    numW2piece = 0
    Blackblockedpieces = 0

# Calculating number of white and black pieces, White and Black Mills
    for i in range(len(b)):
        if b[i] == 'W':
            numWhitePieces += 1
            if closeMill(i, b):
                numWhiteMills += 1
        elif b[i] == 'B':
            numBlackPieces += 1
            if closeMill(i, b):
                numBlackMills += 1

# Calculating number of Black pieces blocked
                n = neighbours(i)
                for j in n:
                    if b[j] == 'x':
                        break
                    else:
                        Blackblockedpieces += 1

# Calculating the number of 2 piece configurations for the White
        elif b[i] == 'x':
            b1 = b[:]
            b1[i] = 'W'
            if closeMill(i,b1):
```

```
            numW2piece += 1
    if numWhiteMills % 3 != 0:
        numWhiteMills = int(numWhiteMills/3) + 1
    else:
        numWhiteMills = int(numWhiteMills/3)
    if numBlackMills % 3 != 0:
        numBlackMills = int(numBlackMills/3) + 1
    else:
        numBlackMills = int(numBlackMills/3)
    if len(b) == 19:
        possiblemill = b[-1]
    return  10 * (numWhitePieces - numBlackPieces) + 10 * (numWhiteMills - numBlackMills) +
5 * (possiblemill) + 3 * numW2piece + 1 * Blackblockedpieces

# neighbours function
# Returns the neighbours of the given location
# Input: a location j in the array representing the board
# Output: a list of locations in the array corresponding to j's neighbors
def neighbours(j):
    def zero():
        return [1, 2, 15]

    def one():
        return [0, 3, 8]

    def two():
        return [0, 3, 4, 12]

    def three():
        return [1, 2, 5, 7]

    def four():
        return [2, 5, 9]

    def five():
        return [3, 4, 6]

    def six():
        return [5, 7, 11]

    def seven():
        return [3, 6, 8, 14]

    def eight():
        return [1, 7, 17]
```

```
    def nine():
        return [4, 10, 12]

    def ten():
        return [9, 11, 13]

    def eleven():
        return [6, 10, 14]

    def twelve():
        return [2, 9, 13, 15]

    def thirteen():
        return [10, 12, 14, 16]

    def fourteen():
        return [7, 11, 13, 17]

    def fifteen():
        return [0, 12, 16]

    def sixteen():
        return [13, 15, 17]

    def seventeen():
        return [8, 14, 16]

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# closeMill function
# To check if the current move makes a mill
# Input: a location j in the array representing the board and the board b
# Output: returns True if the move to j closes a mill or else returns False
def closeMill(j, b):
    C = b[j]

    def zero():
        if (b[2] == b[4] == C):
            return True
        else:
            return False
```

```
def one():
    if (b[3] == b[5] == C) or (b[8] == b[17] == C):
        return True
    else:
        return False

def two():
    if (b[0] == b[4] == C):
        return True
    else:
        return False

def three():
    if (b[1] == b[5] == C) or (b[7] == b[14] == C):
        return True
    else:
        return False

def four():
    if (b[0] == b[2] == C):
        return True
    else:
        return False

def five():
    if (b[1] == b[3] == C) or (b[6] == b[11] == C):
        return True
    else:
        return False

def six():
    if (b[5] == b[11] == C) or (b[7] == b[8] == C):
        return True
    else:
        return False

def seven():
    if (b[3] == b[14] == C) or (b[6] == b[8] == C):
        return True
    else:
        return False

def eight():
    if (b[6] == b[7] == C) or (b[1] == b[17] == C):
        return True
    else:
```

```
        return False

def nine():
    if (b[10] == b[11] == C) or (b[12] == b[15] == C):
        return True
    else:
        return False

def ten():
    if (b[9] == b[11] == C) or (b[13] == b[16] == C):
        return True
    else:
        return False

def eleven():
    if (b[5] == b[6] == C) or (b[9] == b[10] == C) or (b[14] == b[17] == C):
        return True
    else:
        return False

def twelve():
    if (b[9] == b[15] == C) or (b[13] == b[14] == C):
        return True
    else:
        return False

def thirteen():
    if (b[12] == b[14] == C) or (b[10] == b[16] == C):
        return True
    else:
        return False

def fourteen():
    if (b[3] == b[7] == C) or (b[12] == b[13] == C) or (b[11] == b[17] == C):
        return True
    else:
        return False

def fifteen():
    if (b[16] == b[17] == C) or (b[12] == b[9] == C):
        return True
    else:
        return False

def sixteen():
    if (b[15] == b[17] == C) or (b[13] == b[10] == C):
```

```
            return True
        else:
            return False

    def seventeen():
        if (b[1] == b[8] == C) or (b[15] == b[16] == C) or (b[14] == b[11] == C):
            return True
        else:
            return False

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
             6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
             12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# generateRemove function
# Removes the Blackpieces from the given board which are not in the mill and adds that position to the list
# Input: a board position, a list L and the depth
# Output: positions are added to L by removing black pieces
# If no blackpieces can be removed the given board is added to the list
def generateRemove(b, L, ply):
    a1 = len(L)
    for i in range(len(b)):
        if b[i] == "B":
            if not closeMill(i, b):
                b1 = b[:]
                b1[i] = 'x'

# If a closed mill is formed in the last level append a 1 or -1 based on the closed mill is of White's or Black's respectively
# We use this as a evaluating factor in the static estimation function
                if ply == depth:
                    if depth % 2 == 0:
                        b1.append(int(-1))
                    else:
                        b1.append(int(1))
                L.append(b1)
    a2 = len(L)
    if a1 == a2:
        if ply == depth:
            if depth % 2 == 0:
                b.append(int(-1))
            else:
                b.append(int(1))
```

```
        L.append(b)

# GenerateAdd function
# adds a whitepiece in available places and adds that position to the list
# Input: a board position and the depth
# Output: a list L of board positions
def GenerateAdd(b, ply):
    L = []
    for i in range(len(b)):
        if b[i] == 'x':
            b1 = b[:]
            b1[i] = 'W'
            if closeMill(i, b1):
                generateRemove(b1, L, ply)
            else:
                L.append(b1)
    return L

# WtB function
# Converts all the whitepieces to black and blackpieces to white
# Input: a board position
# Output: a board position where all the whites are swapped to black and vice-versa
def WtB(b):
    b1 = b[:]
    for i in range(len(b1)):
        if b1[i] == "W":
            b1[i] = "B"
        elif b1[i] == "B":
            b1[i] = "W"
    return b1

# leaf function
# Checks if the given board posiition is a leaf
# Input: a board position
# Output: returns True if the given board position is a leaf or else returns False
def leaf(b):
    numWhitepieces = 0
    numBlackpieces = 0
    for i in b:
        if i == 'W':
            numWhitepieces += 1
        elif i == 'B':
            numBlackpieces += 1
    if (numWhitepieces == 8 or numBlackpieces == 8):
        return True
    else:
```

```
        return False

# MinMax function
# Generates the moves for the current min node position and returns the minimum value for all the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: Generates the min node moves for the current board position by swapping the blacks and whites using WtB function
#        and compares the current minimum value with MaxMin value of all generated moves and returns the minimum value.
def MinMax(b, ply):
    if ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        b = WtB(b)
        v = 10000
        x = GenerateAdd(b, ply)
        for i in x:
            i = WtB(i)
            v = min(v, MaxMin(i, ply))
        return v

# MaxMin function
# Generates the moves for the current max node position and returns the maximum value for all the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static estimation value of that board position
#        else  Generates the max node moves for the current board position and compares the current maximum value with MinMax value of all
#              generated moves and returns the maximum value. If the depth is 1, the board position for the max value is also returned.
def MaxMin(b, ply):
    if leaf(b) or ply == depth:
        return static_estimation(b)
    else:
        ply += 1
        v = -10000
        y = GenerateAdd(b, ply)
        x = 0
        for i in y:
            m = MinMax(i, ply)
            if m > v:
                x = i
                v = m
```

```
        if ply == 1:
            if x == 0:
                x = y[0]
            return x, v
        return v

# Input of the program
(f1, f2, depth) = list(input().split())
depth = int(depth)
file1 = open(f1)
l1 = list(file1.read())
s = ""
count = 0
for i in l1:
    s += i
    if i == 'W':
        count += 1

# Checks if the given depth is 0
if depth == 0:
    file2 = open(f2, "w")
    file2.write("No moves are calculated" + "\n\n" + "Board position is: " + s)

# Checks if the opening game is ended for the current player
elif count == 8:
    file2 = open(f2, "w")
    file2.write("No further moves are there" + "\n\n" + "The opening game is completed for the
player" + "\n\n" + "Board position is: " + s )

# If none of the above conditions are satisfied MaxMin algorithm is executed
else:
    (A1, A2) = MaxMin(l1, 0)
    s = ""
    for i in range(18):
        s += A1[i]
    file2 = open(f2, "w")
    file2.write("Board position is: " + s + "\n\n" + "Positions evaluated by static estimation: " +
str(positions_evaluated) + "\n\n"
                + "MINIMAX estimate: " + str(A2))
```

Eighth Program: MiniMaxGameImproved

```
positions_evaluated = 0

# Static estimation function
# Calculates the static estimate value of the board position when the leaf nodes are reached
# Input: Board position and the depth is given as the input
# Output: A value is returned for the given board position
def static_estimation(b, ply):
    global positions_evaluated
    positions_evaluated += 1
    possiblemill = 0
    numWhitePieces = countofpieces(b, 'W')
    numBlackPieces = countofpieces(b, 'B')
    numWhiteMills = 0
    numBlackMills = 0
    numW2piece = 0
    Blackblockedpieces = 0
    Bmoves = numBlackMoves(b)
    if numBlackPieces <= 2 or Bmoves == 0:
        return 10000 - ply
    elif numWhitePieces <= 2:
        return -10000 + ply

# Calculating number of White and Black Mills
    for i in range(len(b)):
        if b[i] == 'W':
            if closeMill(i, b):
                numWhiteMills += 1
        elif b[i] == 'B':
            if closeMill(i, b):
                numBlackMills += 1
            if numBlackPieces > 3:

# Calculating number of Black pieces blocked
                n = neighbours(i)
                for j in n:
                    if b[j] == 'x':
                        break
                    else:
                        Blackblockedpieces += 1

# Calculating the number of 2 piece configurations for the White
        elif b[i] == 'x':
            b1 = b[:]
            b1[i] = 'W'
```

```
            if closeMill(i,b1):
                numW2piece += 1
    if numWhiteMills % 3 != 0:
        numWhiteMills = int(numWhiteMills/3) + 1
    else:
        numWhiteMills = int(numWhiteMills/3)
    if numBlackMills % 3 != 0:
        numBlackMills = int(numBlackMills/3) + 1
    else:
        numBlackMills = int(numBlackMills/3)
    if len(b) == 19:
        possiblemill = b[-1]

    return 100 * (numWhitePieces - numBlackPieces) + 100 * (numWhiteMills - numBlackMills)
+ 50 * possiblemill + 40 * Blackblockedpieces + 30 * numW2piece - 10 * Bmoves

# countofpieces function
# Counts the number of specified pieces (White or Black) in the given board position
# Input: a board position and a specific piece(black or white)
# Output: returns the number of specified pieces (White or Black) in the given board position
def countofpieces(b, x):
    count = 0
    for i in b:
        if i == x:
            count += 1
    return count

# numWhiteMoves function
# calculates the number of possible moves for the white for the given board
# Input: a board position
# Output: returns the number of moves possible for white for the given board position
def numWhiteMoves(l):
    n = GenerateMovesMidgameEndgame(l)
    if n == 3:
        x = GenerateHopping(l, 0)
    else:
        x = GenerateMove(l, 0)
    return len(x)

# numBlackMoves function
# calculates the number of possible moves for the black for the given board
# Input: a board position
# Output: returns the number of moves possible for black for the given board position
def numBlackMoves(l):
    ll = WtB(l)
    n = GenerateMovesMidgameEndgame(ll)
```

```
    if n == 3:
        x = GenerateHopping(ll, 0)
    else:
        x = GenerateMove(ll, 0)
    return len(x)

# neighbours function
# Returns the neighbours of the given location
# Input: a location j in the array representing the board
# Output: a list of locations in the array corresponding to j's neighbors
def neighbours(j):
    def zero():
        return [1, 2, 15]

    def one():
        return [0, 3, 8]

    def two():
        return [0, 3, 4, 12]

    def three():
        return [1, 2, 5, 7]

    def four():
        return [2, 5, 9]

    def five():
        return [3, 4, 6]

    def six():
        return [5, 7, 11]

    def seven():
        return [3, 6, 8, 14]

    def eight():
        return [1, 7, 17]

    def nine():
        return [4, 10, 12]

    def ten():
        return [9, 11, 13]

    def eleven():
        return [6, 10, 14]
```

```
    def twelve():
        return [2, 9, 13, 15]

    def thirteen():
        return [10, 12, 14, 16]

    def fourteen():
        return [7, 11, 13, 17]

    def fifteen():
        return [0, 12, 16]

    def sixteen():
        return [13, 15, 17]

    def seventeen():
        return [8, 14, 16]

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# closeMill function
# To check if the current move makes a mill
# Input: a location j in the array representing the board and the board b
# Output: returns True if the move to j closes a mill or else returns False
def closeMill(j, b):
    C = b[j]

    def zero():
        if (b[2] == b[4] == C):
            return True
        else:
            return False

    def one():
        if (b[3] == b[5] == C) or (b[8] == b[17] == C):
            return True
        else:
            return False

    def two():
        if (b[0] == b[4] == C):
```

```
        return True
    else:
        return False

def three():
    if (b[1] == b[5] == C) or (b[7] == b[14] == C):
        return True
    else:
        return False

def four():
    if (b[0] == b[2] == C):
        return True
    else:
        return False

def five():
    if (b[1] == b[3] == C) or (b[6] == b[11] == C):
        return True
    else:
        return False

def six():
    if (b[5] == b[11] == C) or (b[7] == b[8] == C):
        return True
    else:
        return False

def seven():
    if (b[3] == b[14] == C) or (b[6] == b[8] == C):
        return True
    else:
        return False

def eight():
    if (b[6] == b[7] == C) or (b[1] == b[17] == C):
        return True
    else:
        return False

def nine():
    if (b[10] == b[11] == C) or (b[12] == b[15] == C):
        return True
    else:
        return False
```

```
def ten():
    if (b[9] == b[11] == C) or (b[13] == b[16] == C):
        return True
    else:
        return False

def eleven():
    if (b[5] == b[6] == C) or (b[9] == b[10] == C) or (b[14] == b[17] == C):
        return True
    else:
        return False

def twelve():
    if (b[9] == b[15] == C) or (b[13] == b[14] == C):
        return True
    else:
        return False

def thirteen():
    if (b[12] == b[14] == C) or (b[10] == b[16] == C):
        return True
    else:
        return False

def fourteen():
    if (b[3] == b[7] == C) or (b[12] == b[13] == C) or (b[11] == b[17] == C):
        return True
    else:
        return False

def fifteen():
    if (b[16] == b[17] == C) or (b[12] == b[9] == C):
        return True
    else:
        return False

def sixteen():
    if (b[15] == b[17] == C) or (b[13] == b[10] == C):
        return True
    else:
        return False

def seventeen():
    if (b[1] == b[8] == C) or (b[15] == b[16] == C) or (b[14] == b[11] == C):
        return True
    else:
```

```
            return False

    switch = {0: zero, 1: one, 2: two, 3: three, 4: four, 5: five,
              6: six, 7: seven, 8: eight, 9: nine, 10: ten, 11: eleven,
              12: twelve, 13: thirteen, 14: fourteen, 15: fifteen, 16: sixteen, 17: seventeen}

    return switch[j]()

# generateRemove function
# Removes the Blackpieces from the given board which are not in the mill and adds that position
to the list
# Input: a board position, a list L and the depth
# Output: positions are added to L by removing black pieces
# If no blackpieces can be removed the given board is added to the list
def generateRemove(b, L, ply):
    a1 = len(L)
    for i in range(len(b)):
        if b[i] == "B":
            if not closeMill(i, b):
                b1 = b[:]
                b1[i] = 'x'

# If a closed mill is formed in the last level append a 1 or -1 based on the closed mill is of
White's or Black's respectively
# We use this as a evaluating factor in the static estimation function                       \
                if ply == depth:
                    if depth % 2 == 0:
                        b1.append(int(-1))
                    else:
                        b1.append(int(1))
                L.append(b1)
    a2 = len(L)
    if a1 == a2:
        if ply == depth:
            if depth % 2 == 0:
                b.append(int(-1))
            else:
                b.append(int(1))
        L.append(b)

# GenerateMove function
# generates moves created by moving a white piece to an adjacent location
# Input: a board position and the depth
# Output: a list L of board positions
def GenerateMove(b, ply):
    L = []
```

```
    for i in range(len(b)):
        if b[i] == "W":
            n = neighbours(i)
            for j in n:
                if b[j] == "x":
                    b1 = b[:]
                    b1[i] = "x"
                    b1[j] = "W"
                    if closeMill(j, b1):
                        generateRemove(b1, L, ply)
                    else:
                        L.append(b1)
    return L

# GenerateHopping function
# generates moves created by hopping a white piece
# Input: a board position and the depth
# Output: a list L of board positions
def GenerateHopping(b, ply):
    L = []
    for i in range(len(b)):
        if b[i] == "W":
            for j in range(len(b)):
                if b[j] == "x":
                    b1 = b[:]
                    b1[i] = "x"
                    b1[j] = "W"
                    if closeMill(j, b1):
                        generateRemove(b1, L, ply)
                    else:
                        L.append(b1)
    return L

# GenerateMovesMidgameEndgame
# Determines whether to use GenerateMoves or GenerateHopping
# Input: a board position
# Output: the number of whitepieces
# if it is equal to 3 hopping is executeed or else GenerateMoves is executed in the MaxMin or
MinMax function
def GenerateMovesMidgameEndgame(b):
    n = 0
    for i in b:
        if i == "W":
            n += 1
    return n
```

```
# WtB function
# Converts all the whitepieces to black and blackpieces to white
# Input: a board position
# Output: a board position where all the whites are swapped to black and vice-versa
def WtB(b):
    b1 = b[:]
    for i in range(len(b1)):
        if b1[i] == "W":
            b1[i] = "B"
        elif b1[i] == "B":
            b1[i] = "W"
    return b1

# leaf function
# Checks if the given board posiition is a leaf
# Input: a board position and the depth
# Output: returns True if the given board position is a leaf or else returns False
def leaf(b, ply):
    numWpieces = countofpieces(b, 'W')
    numBpieces = countofpieces(b, 'B')
    if numWpieces < 3 or numBpieces < 3:
        return True
    elif ply % 2 == 0:
        if numWhiteMoves(b) == 0:
            return True
    elif ply % 2 != 0:
        if numBlackMoves(b) == 0:
            return True
    else:
        return False

# MinMax function
# Generates the moves for the current min node position and returns the minimum value for all
the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static
estimation value of that board position
#         else  Generates the min node moves for the current board position and compares the
current minimum value with MaxMin value of all
#               generated moves and returns the minimum value.
def MinMax(b, ply):
    if leaf(b, ply) or ply == depth:
        return static_estimation(b, ply)
    else:
        ply += 1
        b = WtB(b)
```

```
        v = 10000
        n = GenerateMovesMidgameEndgame(b)
        if n == 3:
            x = GenerateHopping(b, ply)
        else:
            x = GenerateMove(b, ply)
        for i in x:
            i = WtB(i)
            v = min(v, MaxMin(i, ply))
        return v

# MaxMin function
# Generates the moves for the current max node position and returns the maximum value for all
the generated moves based on static estimation value
# Input: a board position and the depth of the current node posiition
# Output: if    a leaf node or a node at the maximum depth is reached it returns the static
estimation value of that board position
#         else  Generates the max node moves for the current board position and compares the
current maximum value with MinMax value of all
#               generated moves and returns the maximum value. If the depth is 1, the board position
for the max value is also returned.
def MaxMin(b, ply):
    if leaf(b, ply) or ply == depth:
        return static_estimation(b, ply)
    else:
        ply += 1
        v = -10000
        n = GenerateMovesMidgameEndgame(b)
        if n == 3:
            y = GenerateHopping(b, ply)
        else:
            y = GenerateMove(b, ply)
        x = 0
        for i in y:
            m = MinMax(i, ply)
            if m > v:
                x = i
                v = m
        if ply == 1:
            if x == 0:
                x = y[0]
            return x, v
        return v

# Input of the program
(f1, f2, depth) = list(input().split())
```

```
depth = int(depth)
file1 = open(f1)
l1 = list(file1.read())
s = ""
Whitepieces = countofpieces(l1, 'W')
Blackpieces = countofpieces(l1, 'B')
for i in l1:
    s += i

# Checks if the given depth is 0
if depth == 0:
    file2 = open(f2, "w")
    file2.write("No moves are calculated" + "\n\n" + "Board position is: " + s)

# checks if the board position is valid
elif Whitepieces < 3 and Blackpieces < 3:
    file2 = open(f2, "w")
    file2.write("Not a vaid board position, PLEASE check it")

# checks if we have lost already
elif Whitepieces < 3 or numWhiteMoves(l1) == 0:
    file2 = open(f2, "w")
    file2.write("You have lost the game man, SORRY!!!" + "\n\n" + "Final Board position is: " +
s)

# checks if we have won already
elif Blackpieces < 3 or numBlackMoves(l1) == 0:
    file2 = open(f2, "w")
    file2.write("CONGRATULATIONS!!!, You have won the game" + "\n\n" + "Final Board
position is: " + s)

# If none of the above conditions are satisfied MaxMin algorithm is executed
else:
    (A1, A2) = MaxMin(l1, 0)
    s = ""
    for i in A1:
        s += i

# Checks if the next move the player makes will finish the game
    if A2 == 9999:
        file2 = open(f2, "w")
        file2.write("CONGRATULATIONS!!!, You have won the game" + "\n\n" + "Final Board
position is: " + s)
    else:
        file2 = open(f2, "w")
```

```
file2.write("Board position is: " + s + "\n\n" + "Positions evaluated by static estimation: " + str(positions_evaluated) + "\n\n"
            + "MINIMAX estimate: " + str(A2))
```