MILITARY INSTITUTE OF SCIENCE AND TECHNOLOGY
Department of Computer Science and Engineering

==========

CSE-404: ARTIFICIAL INTELLIGENCE SESSIONAL

ADVERSARIAL SEARCH

Section: B Group No: B03

Group Members:

1. 201614117 Lt Nayma Eisha
2. 201614123 Capt MD Rafsun Sheikh

Date of Submission: May 14, 2021

INTRODUCTION

In this introduction the rules of the game Mancala are described, as well as some nomenclature used throughout this text

1.0.1. Introduction:

Mancala is a generic name for a family of two-player turn-based strategy board games played with small stones, beans, or seeds and rows of holes or pits in the earth, a board or other playing surface. The objective is usually to capture all or some set of the opponent's pieces.

1.0.2. History:

Evidence of the game was uncovered in Israel in the city of Gedera in an excavated Roman bathhouse where pottery boards and rock cuts were unearthed dating back to between the 2nd and 3rd century AD. Among other early evidence of the game are fragments of a pottery board and several rock cuts found in Aksumite areas in Matara (in Eritrea) and Yeha (in Ethiopia), which are dated by archaeologists to between the 6th and 7th centuries AD; the game may have been mentioned by Giyorgis of Segla in his 14th century Ge'ez text Mysteries of Heaven and Earth, where he refers to a game called qarqis, a term used in Ge'ez to refer to both Gebet'a (mancala) and Sant'araz (modern sent'erazh, Ethiopian chess).

More information about the game in Wikipedia.

1.1. The Rules of the Game :

Most mancala games share a common general game play. Players begin by placing a certain number of seeds, prescribed for the particular game, in each of the pits on the game board. A player may count their stones to plot the game. A turn consists of removing all seeds from a pit, "sowing" the seeds (placing one in each of the following pits in sequence) and capturing based on the state of the board. The object of the game is to plant the most seeds in the bank. This leads to the English phrase "count and capture" sometimes used to describe the gameplay. Although the details differ greatly, this general sequence applies to all games.

If playing in capture mode, once a player ends their turn in an empty pit on their own side, they capture the opponent's pieces directly across. Once captured, the player gets to put the seeds in their own bank. After capturing, the opponent forfeits a turn.

Game Play:

1. The game begins with one player picking up all of the pieces in any one of the pockets on his/her side.

2. Moving counter-clockwise, the player deposits one of the stones in each pocket until the stones run out.

3. If you run into your own Mancala (store), deposit one piece in it. If you run into your opponent's Mancala, skip it and continue moving to the next pocket.

4. If the last piece you drop is in your own Mancala, you take another turn.

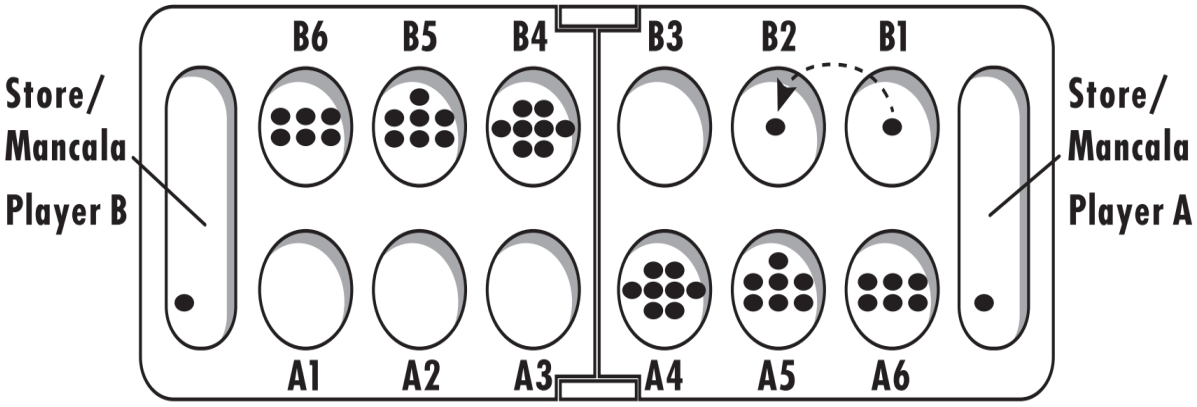


Diagram 1.1

5. If the last piece you drop is in an empty pocket on your side, you capture that piece and any pieces in the pocket directly opposite. (See figure below: In this image, player B has moved his/her piece from space B1- B2. By occupying the empty pocket on his/her side, player B earns THAT piece IN ADDITION TO the pieces in pocket A5 on the opposite side)

6. Always place all captured pieces in your Mancala (store).

7. The game ends when all six pockets on one side of the Mancala board are empty.

8. The player who still has pieces on his/her side of the board when the game ends captures all of those pieces.

9. Count all the pieces in each Mancala. The winner is the player with the most pieces

1.2. Nomenclature

In order to be able to talk about a position, it is useful to give each pit a name. We have chosen to use a nomenclature as used by general Mancala Players. The 2 rows are labeled ‘A’ and ‘B’, while the columns are numbered 1 through 6. In this way the left most bottom pit is called A1.

It is now possible to make a list of the moves made during a game. For the game of above diagram
this could have been:

1. B1 -> B2
2. A1 -> A2
3. B6 -> Mancala B
4. Player A wins

It is also easy to use the names of the pits to show which side is winning.

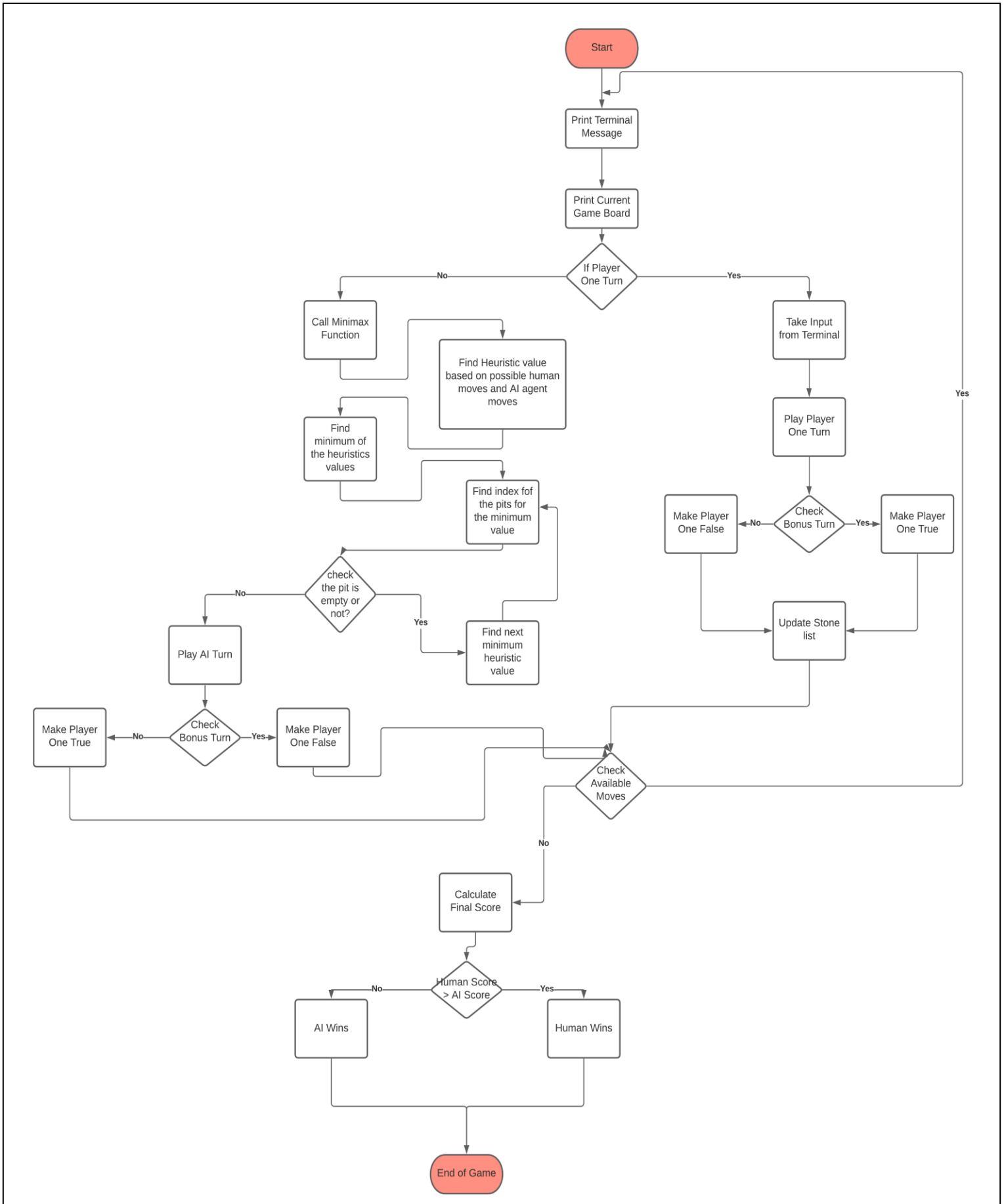
2. Methodology

For developing the Game we followed some basic steps.

a. As we know this game Mancala, is a bit difficult than many others two players games. So we tried to implement a mancala game with all working rules which is not AI based. Two different player can play the game against each other. We created the game in python and tested all possible moves whether it is working or not. We tried to fix any broken rules in our general two player mancala game.
b. Then we divided each task into different functions so that when we implement the AI agent we just can call and make the code working.
c. For implementing the AI Agent we used the help of Minimax Algorithm. In that case we can define the depth for the algorithm to run on and we can get the heuristic score of both the player for each moves.
d. For calculating the Heuristic value we used the end game player result method where after each moves we calculate the total stones number in AI agent side pits and human agent pits including their mancala pits. Then we get the difference of the scores for both the player and that is our Heuristic value for that move.
e. As we can calculate the heuristic value for each of the moves we saved all the values in a python list so that we can use the values later.
f. For implementing minimax function, we have taken a bool variable, depth and bin number. We checked for the end of depth for returning the heuristic value. Otherwise we checked the bool variable. For True value we recursively called the minimax function again and find the minimum value of the heuristic score. For the False value of the bool we recursively called the minimax function and get the maximum value of the heuristic scores.
g. So after implementing the minimax function our workflow of the code look like the diagram at figure 2 where we start the game.
h. Then we print the terminal message whether its player One's turn or AI agent turn or invalid input.
i. Then we print the current game board. For the game board we used usual print function and for the values we get the values from a list.
j. If it's the turn for the player one we take input from the terminal. And for a valid input we assigned a pit value which is taken as chosen bin pit from player one side.
k. According to the input of the player we play the turn, set the variables according to the game rules and check for either the player one gets a bonus turn or not.
l. If player one gets a bonus turn , then we make playerOne variable true, update the stone list and return back to initial position of the game.

m. If player one has no bonus move then we make the player One bool False and update the stone list and return back to the initial stage.
n. If its for the AI agents turn, we call the minimax function and then for the exact game state we get the possible moves tree for the AI agent and player one according to the depth we given.
o. Then we get the heuristic values for each of the moves pairs and save the values in a list.
p. Then we find the minimum of the heuristic values of the list.
q. We get the index of the AI Agents pits for the minimum value and check the pit has stones or not.
r. If the pit is empty we get the next minimum value and rerun the loop. If the pit is not empty then we play the AI agent turn.
s. Again we check the bonus turn for the AI agent. If it gets a bonus turn we make the playerOne bool False and get to the initial state.
t. Or we make the playerOne True and return to initial state.

MANCALA


Start
Print Terminal Message
Print Current Game Board
If Player One Turn
No
Yes
Call Minimax Function
Find Heuristic value based on possible human moves and AI agent moves
Find minimum of the heuristics values
Find index fof the pits for the minimum value
check the pit is empty or not?
No
Yes
Find next minimum heuristic value
Play AI Turn
Check Bonus Turn
No
Yes
Make Player One True
Make Player One False
Take Input from Terminal
Play Player One Turn
Check Bonus Turn
No
Yes
Make Player One False
Make Player One True
Update Stone list
Check Available Moves
Yes
No
Calculate Final Score
Human Score > AI Score
No
Yes
AI Wins
Human Wins
End of Game

u. In all the cases we check whether we have available moves or not. If we don't have available moves we calculate the final score and find out the winner. Or if we have available moves we get back to the initial position.

3. Code:

```
binAmount = [4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0]

copyBinAmount = []

tempBinAmount = []

diff_list = list()

temp_diff_list = list()

playing = True

playerOne = True

messageCode = 0

giveawayPile = -1

lastRecipient = -1

choosenBin = -2

difference = 0

def terminal_message(messageCode, playerOne):
    message = ""
    if playerOne and messageCode == 0:
        message = "Player One's Turn..."
    elif not(playerOne) and messageCode == 0:
        message = "Player Two's Turn..."
    elif playerOne and messageCode == -2:
        message = "Invalid Input. Try again Player One.."
    elif not(playerOne) and messageCode == -2:
        message = "Invalid Input. Try again Player Two.."
    elif playerOne and messageCode == -1:
        message = "You must choose a non-empty bin player One"
    elif not(playerOne) and messageCode == -1:
        message = "You must choose a non-empty bin player Ai"
```

```
    print("")
    print(message)
    print("")

def board():
    i = 0
    for element in binAmount:
        binAmount[i] = int(binAmount[i])
        if int(binAmount[i] < 10):
            binAmount[i] = " " + str(binAmount[i])
        else:
            binAmount[i] = str(binAmount[i])
        i = i + 1
    # end of for loop
    if not(playerOne):
        print("      a   b   c   d   e   f")
    print("+----+----+----+----+----+----+----+----+")
    print("|    | " + binAmount[12] + " | " + binAmount[11] + " | " + binAmount[10] + " | " + binA
mount[9] + " | " +
          binAmount[8] + " | " + binAmount[7] + " |    |")
    print("| " + binAmount[13] + " |----+----+----+----+----+----| " + binAmount[6] + " |")
    print("|    | " + binAmount[0] + " | " + binAmount[1] + " | " + binAmount[2] + " | " + binAmou
nt[3] + " | " +
          binAmount[4] + " | " + binAmount[5] + " |    |")
    print("+----+----+----+----+----+----+----+----+")
    if playerOne:
        print("      f   e   d   c   b   a")
    print("")

def player_one_user_input(userInput,messageCode,playing):
    if userInput == 'q':
        playing = False
        choosenBin = 0
    elif userInput == "a":
        choosenBin = 5
    elif userInput == "b":
        choosenBin = 4
    elif userInput == "c":
        choosenBin = 3
    elif userInput == "d":
        choosenBin = 2
    elif userInput == "e":
        choosenBin = 1
```

```
    elif userInput == "f":
        choosenBin = 0
    else:
        choosenBin = -2
        messageCode = -2  #invalid input
    return (choosenBin,messageCode,playing)


def player_one_stone_distribution(choosenBin):
    global messageCode, playerOne, giveawayPile, lastRecipeint, tempBinAmount
    if int(choosenBin) >= 0:
        giveawayPile = int(tempBinAmount[choosenBin])
        tempBinAmount[choosenBin] = 0
        if int(giveawayPile) <= 0:
            messageCode = -1 # empty bin was chosen

    recipient = choosenBin + 1
    while int(giveawayPile) > 0:
        if int(recipient) == 13:
            recipient = 0
        tempBinAmount[recipient] = int(tempBinAmount[recipient]) + 1
        giveawayPile = int(giveawayPile) - 1

        if int(giveawayPile) == 0:
            lastRecipeint = recipient
        else:
            recipient = int(recipient) + 1
            if int(recipient) > 13:
                recipient = 0

    if int(lastRecipeint) ==6:
        playerOne = True
    elif(int(tempBinAmount[lastRecipeint]) == 1 and int(lastRecipeint) < 6):
        tempBinAmount[6] = int(tempBinAmount[6]) + int(tempBinAmount[lastRecipeint]) + int(te
mpBinAmount[12 - int(lastRecipeint)])
        tempBinAmount[lastRecipeint] = 0
        tempBinAmount[12 - int(lastRecipeint)] = 0
        playerOne = not(playerOne)
    elif(int(messageCode) >= 0):
        playerOne = not (playerOne)

    return tempBinAmount
```

```
def player_one_stone_minimax_distribution(choosenBin):
    global messageCode, playerOne, giveawayPile, lastRecipeint, tempBinAmount
    if int(choosenBin) >= 0:
        giveawayPile = int(tempBinAmount[choosenBin])
        tempBinAmount[choosenBin] = 0

    recipient = choosenBin + 1
    while int(giveawayPile) > 0:
        if int(recipient) == 13:
            recipient = 0
        tempBinAmount[recipient] = int(tempBinAmount[recipient]) + 1
        giveawayPile = int(giveawayPile) - 1

        if int(giveawayPile) == 0:
            lastRecipeint = recipient
        else:
            recipient = int(recipient) + 1
            if int(recipient) > 13:
                recipient = 0

    if int(lastRecipeint) ==6:
        playerOne = True
    elif(int(tempBinAmount[lastRecipeint]) == 1 and int(lastRecipeint) < 6):
        tempBinAmount[6] = int(tempBinAmount[6]) + int(tempBinAmount[lastRecipeint]) + int(te
mpBinAmount[12 - int(lastRecipeint)])
        tempBinAmount[lastRecipeint] = 0
        tempBinAmount[12 - int(lastRecipeint)] = 0
        playerOne = playerOne
    elif(int(messageCode) >= 0):
        playerOne = playerOne

    return tempBinAmount

def player_ai_stone_distribution(choosenBin):
    global messageCode, playerOne, giveawayPile, lastRecipeint, tempBinAmount
    if int(choosenBin) >= 0:
        giveawayPile = int(tempBinAmount[choosenBin])
        tempBinAmount[choosenBin] = 0
        if int(giveawayPile) <= 0:
            messageCode = -1 # empty bin was chosen

    recipient = choosenBin + 1
    while int(giveawayPile) > 0:
        if int(recipient) == 6:
```

```
            recipient = 7
        tempBinAmount[recipient] = int(tempBinAmount[recipient]) + 1
        giveawayPile = int(giveawayPile) - 1

        if int(giveawayPile) == 0:
            lastRecipeint = recipient
        else:
            recipient = int(recipient) + 1
            if int(recipient) > 13:
                recipient = 0

    if int(lastRecipeint) ==13:
        playerOne = False
    elif(int(tempBinAmount[lastRecipeint]) == 1 and int(lastRecipeint) > 6):
        tempBinAmount[13] = int(tempBinAmount[13]) + int(tempBinAmount[lastRecipeint]) + int(
tempBinAmount[12 - int(lastRecipeint)])
        tempBinAmount[lastRecipeint] = 0
        tempBinAmount[12 - int(lastRecipeint)] = 0
        playerOne = not(playerOne)
    elif(int(messageCode) >= 0):
        playerOne = not (playerOne)

    return tempBinAmount

def player_ai_stone_minimax_distribution(choosenBin):
    global messageCode, playerOne, giveawayPile, lastRecipeint, tempBinAmount
    if int(choosenBin) >= 0:
        giveawayPile = int(tempBinAmount[choosenBin])
        tempBinAmount[choosenBin] = 0

    recipient = choosenBin + 1
    while int(giveawayPile) > 0:
        if int(recipient) == 6:
            recipient = 7
        tempBinAmount[recipient] = int(tempBinAmount[recipient]) + 1
        giveawayPile = int(giveawayPile) - 1

        if int(giveawayPile) == 0:
            lastRecipeint = recipient
        else:
            recipient = int(recipient) + 1
            if int(recipient) > 13:
                recipient = 0
```

```
    if int(lastRecipeint) ==13:
        playerOne = False
    elif(int(tempBinAmount[lastRecipeint]) == 1 and int(lastRecipeint) > 6):
        tempBinAmount[13] = int(tempBinAmount[13]) + int(tempBinAmount[lastRecipeint]) + int(
tempBinAmount[12 - int(lastRecipeint)])
        tempBinAmount[lastRecipeint] = 0
        tempBinAmount[12 - int(lastRecipeint)] = 0
        playerOne = playerOne
    elif(int(messageCode) >= 0):
        playerOne = playerOne

    return tempBinAmount


def end_game_check(playing):
    sideOne = 0
    sideTwo = 0
    for j in range(6):
        sideOne = int(sideOne) + int(binAmount[j])
        sideTwo = int(sideTwo) + int(binAmount[j + 7])

    if(int(sideOne) ==0 or int(sideTwo) == 0):
        playing = False
        binAmount[6] = int(binAmount[6]) + int(sideOne)
        binAmount[13] = int(binAmount[13]) + int(sideTwo)
        for k in range(6):
            binAmount[k] = 0
            binAmount[k + 7] = 0
    return playing

def diff():
    global difference, diff_list
    sideOne = 0
    sideAi = 0
    for j in range(7):
        sideOne = int(sideOne) + int(tempBinAmount[j])
        sideAi = int(sideAi) + int(tempBinAmount[j + 7])
    difference = sideOne - sideAi
    diff_list.append(difference)
    return difference


def minimax(maxify, depth, bins):
    global choosenBin, copyBinAmount, tempBinAmount, binAmount, playerOne
```

```
    tempBinAmount = copyBinAmount.copy()

    if depth == 0:
        choosenBin = bins
        return diff()

    if maxify:
        best_score = 999
        for bins in range(7, 13, 1):
            tempBinAmount = binAmount.copy()
            tempBinAmount = player_one_stone_minimax_distribution(bins)
            copyBinAmount = tempBinAmount.copy()
            score = minimax(not maxify, depth-1, bins)
            best_score = min(best_score, score)
        return best_score

    else:
        best_score = -999
        for bins in range(0, 6, 1):
            tempBinAmount = player_ai_stone_minimax_distribution(bins)
            score= minimax(not maxify, depth-1, bins)
            best_score = max(best_score, score)
        return best_score


while (playing):
    terminal_message(messageCode, playerOne)
    messageCode = 0

    board()

    # choosenBin = 0
    if playerOne:
        userInput = input("Enter 'q' to end the game: ")
        choosenBin,messageCode,playing = player_one_user_input(userInput,messageCode,playin
g)
        if playing:
            tempBinAmount = binAmount.copy()
            tempBinAmount = player_one_stone_distribution(choosenBin)
            binAmount = tempBinAmount.copy()

    else:
        tempBinAmount = binAmount.copy()
        copyBinAmount = binAmount.copy()
```

```
      minimax(True, 3,  int(choosenBin))

      for i in range(0, 35, 6):
         temp_diff_list.append(diff_list[i])

      min_not_ended_zero = True

      while(min_not_ended_zero):
         min_temp_diff_list = min(temp_diff_list)

         ai_choosen_bin = temp_diff_list.index(int(min_temp_diff_list))

         tempBinAmount = binAmount.copy()
         temp_bin_value_check = tempBinAmount[ai_choosen_bin + 7]
         if int(temp_bin_value_check) == 0:
            temp_diff_list[ai_choosen_bin] = 999
         else:
            tempBinAmount = player_ai_stone_distribution(ai_choosen_bin + 7)
            binAmount = tempBinAmount.copy()
            diff_list.clear()
            temp_diff_list.clear()
            min_not_ended_zero = False




   playing = end_game_check(playing)

# end of while loop
print("")
print("The Game is over!")
if int(binAmount[13]) < int(binAmount[6]):
   print("Player One has won the game!")
elif int(binAmount[13]) > int(binAmount[6]):
   print("Player Two has won the game!")
else:
   print("The game ended in a Tie!")

i = 0
for element in binAmount:
   binAmount[i] = int(binAmount[i])
   if int(binAmount[i] < 10):
      binAmount[i] = " " + str(binAmount[i])
   else:
```

```
      binAmount[i] = str(binAmount[i])
   i = i + 1
# end of for loop

print("")
print("+----+----+----+----+----+----+----+----+")
print("|    | " + binAmount[12] + " | " + binAmount[11] + " | " + binAmount[10] + " | " + binAmo
unt[9] + " | " +
   binAmount[8] + " | " + binAmount[7] + " |    |")
print("| " + binAmount[13] + " |----+----+----+----+----+----| " + binAmount[6] + " |")
print("|    | " + binAmount[0] + " | " + binAmount[1] + " | " + binAmount[2] + " | " + binAmount[
3] + " | " +
   binAmount[4] + " | " + binAmount[5] + " |    |")
print("+----+----+----+----+----+----+----+----+")
```

4. Explanation of Code:

We have developed the program in python and we didn't use any external library for our implementation of code.

```
binAmount = [4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0]

copyBinAmount = []

tempBinAmount = []

diff_list = list()

temp_diff_list = list()

playing = True

playerOne = True

messageCode = 0

giveawayPile = -1

lastRecipient = -1

choosenBin = -2

difference = 0
```

Here We have declared all the global variable for implementation of our code. binAmount[] holds the final value of our pits after each turn. copyBinAmount[] and tempBinAmount[] is required for the temporary calculation of our turns so that we don't change the real bin values mistakenly. Diff_list[] and temp_diff_list[] holds the heuristic values of our turns. Playing is a bool variable for checking of available turns, playerOne is a bool for turns of human player or AI agent. MessageCode helps us the message to print. GiveawayPile and lastRecipient for the internal stone distribution. ChoosenBin for the choice of the pit or bin.

```
def terminal_message(messageCode, playerOne):
    message = ""
    if playerOne and messageCode == 0:
        message = "Player One's Turn..."
```

```
    elif not(playerOne) and messageCode == 0:
        message = "Player Two's Turn..."
    elif playerOne and messageCode == -2:
        message = "Invalid Input. Try again Player One.."
    elif not(playerOne) and messageCode == -2:
        message = "Invalid Input. Try again Player Two.."
    elif playerOne and messageCode == -1:
        message = "You must choose a non-empty bin player One"
    elif not(playerOne) and messageCode == -1:
        message = "You must choose a non-empty bin player Ai"

    print("")
    print(message)
    print("")
```

This function is for the terminal message printed at the each of every turn. Every time we check two things: messageCode and playerOne bool variable. For messageCode we have three values as 0, -2, -1. 0 for determining player turn, -2 for if any invalid input is selected. And -1 for if a player chooses a bin which has no stones.

```
def board():
    i = 0
    for element in binAmount:
        binAmount[i] = int(binAmount[i])
        if int(binAmount[i] < 10):
            binAmount[i] = " " + str(binAmount[i])
        else:
            binAmount[i] = str(binAmount[i])
        i = i + 1
    # end of for loop
    if not(playerOne):
        print("       a    b    c    d    e    f")
    print("+----+----+----+----+----+----+----+----+")
    print("|    | " + binAmount[12] + " | " + binAmount[11] + " | " + binAmount[10] + " | " + binA
mount[9] + " | " +
          binAmount[8] + " | " + binAmount[7] + " |    |")
    print("| " + binAmount[13] + " |----+----+----+----+----+----| " + binAmount[6] + " |")
    print("|    | " + binAmount[0] + " | " + binAmount[1] + " | " + binAmount[2] + " | " + binAmou
nt[3] + " | " +
```

```
        binAmount[4] + " | " + binAmount[5] + " |    |")
    print("+----+----+----+----+----+----+----+----+")
    if playerOne:
        print("      f   e   d   c   b   a")
    print("")
```

here we print the current game board for both player one and AI agent so that we can understand the game state. In this function we iterate through the binAmount[] list and print the value inside the placeholder.

```
def player_one_user_input(userInput,messageCode,playing):
    if userInput == 'q':
        playing = False
        choosenBin = 0
    elif userInput == "a":
        choosenBin = 5
    elif userInput == "b":
        choosenBin = 4
    elif userInput == "c":
        choosenBin = 3
    elif userInput == "d":
        choosenBin = 2
    elif userInput == "e":
        choosenBin = 1
    elif userInput == "f":
        choosenBin = 0
    else:
        choosenBin = -2
        messageCode = -2  #invalid input
    return (choosenBin,messageCode,playing)
```

This portion of code is for the Player One input from the terminal. We assigned each value for each input and raised flags for invalid inputs.

```
def player_one_stone_distribution(choosenBin):
    global messageCode, playerOne, giveawayPile, lastRecipeint, tempBinAmount
    if int(choosenBin) >= 0:
        giveawayPile = int(tempBinAmount[choosenBin])
        tempBinAmount[choosenBin] = 0
        if int(giveawayPile) <= 0:
            messageCode = -1 # empty bin was chosen

    recipient = choosenBin + 1
    while int(giveawayPile) > 0:
        if int(recipient) == 13:
            recipient = 0
        tempBinAmount[recipient] = int(tempBinAmount[recipient]) + 1
        giveawayPile = int(giveawayPile) - 1

        if int(giveawayPile) == 0:
            lastRecipeint = recipient
        else:
            recipient = int(recipient) + 1
            if int(recipient) > 13:
                recipient = 0

    if int(lastRecipeint) ==6:
        playerOne = True
    elif(int(tempBinAmount[lastRecipeint]) == 1 and int(lastRecipeint) < 6):
        tempBinAmount[6] = int(tempBinAmount[6]) + int(tempBinAmount[lastRecipeint]) + int(te
mpBinAmount[12 - int(lastRecipeint)])
        tempBinAmount[lastRecipeint] = 0
        tempBinAmount[12 - int(lastRecipeint)] = 0
        playerOne = not(playerOne)
    elif(int(messageCode) >= 0):
        playerOne = not (playerOne)

    return tempBinAmount
```

Here we have implemented the game rules for the player one in such a way so that based on the game rules the stones are distributed to the bins. In the first If condition we checked choosen bin for valid input, if valid then we get the number of stones from the respected bin and save it inside giveaway variable. If the pit value is zero then raise a flag.
Inside the while loop we start distributing the stones to the respected bins. We created a round about rules so that after distributing the to the last bin it start distributing to the first bin again. Also we keep track for the stones number.

Inside the If statement we check for the bonus rule of the game. If our last distributed stone is inside own mandala we get a bonus move. Also if our last stone falls inside a empty bin in our side we get the mirror AI pits stones also but no bonus moves. Except this two condition next move will be for AI agent.

```
def player_one_stone_minimax_distribution(choosenBin):
    global messageCode, playerOne, giveawayPile, lastRecipeint, tempBinAmount
    if int(choosenBin) >= 0:
        giveawayPile = int(tempBinAmount[choosenBin])
        tempBinAmount[choosenBin] = 0

    recipient = choosenBin + 1
    while int(giveawayPile) > 0:
        if int(recipient) == 13:
            recipient = 0
        tempBinAmount[recipient] = int(tempBinAmount[recipient]) + 1
        giveawayPile = int(giveawayPile) - 1

        if int(giveawayPile) == 0:
            lastRecipeint = recipient
        else:
            recipient = int(recipient) + 1
            if int(recipient) > 13:
                recipient = 0

    if int(lastRecipeint) ==6:
        playerOne = True
    elif(int(tempBinAmount[lastRecipeint]) == 1 and int(lastRecipeint) < 6):
        tempBinAmount[6] = int(tempBinAmount[6]) + int(tempBinAmount[lastRecipeint]) + int(te
mpBinAmount[12 - int(lastRecipeint)])
        tempBinAmount[lastRecipeint] = 0
        tempBinAmount[12 - int(lastRecipeint)] = 0
        playerOne = playerOne
    elif(int(messageCode) >= 0):
        playerOne = playerOne

    return tempBinAmount
```

This portion of the code is for the minimax algorithm traversal for the possible moves for the player one is implemented here. This function is as same as the Player_one_stone_distribution() except the change of the playerOne bool value change. So that we can protect the value of the playerOne bool from any unwanted error.

```
def player_ai_stone_distribution(choosenBin):
    global messageCode, playerOne, giveawayPile, lastRecipeint, tempBinAmount
    if int(choosenBin) >= 0:
        giveawayPile = int(tempBinAmount[choosenBin])
        tempBinAmount[choosenBin] = 0
        if int(giveawayPile) <= 0:
            messageCode = -1 # empty bin was chosen

    recipient = choosenBin + 1
    while int(giveawayPile) > 0:
        if int(recipient) == 6:
            recipient = 7
        tempBinAmount[recipient] = int(tempBinAmount[recipient]) + 1
        giveawayPile = int(giveawayPile) - 1

        if int(giveawayPile) == 0:
            lastRecipeint = recipient
        else:
            recipient = int(recipient) + 1
            if int(recipient) > 13:
                recipient = 0

    if int(lastRecipeint) ==13:
        playerOne = False
    elif(int(tempBinAmount[lastRecipeint]) == 1 and int(lastRecipeint) > 6):
        tempBinAmount[13] = int(tempBinAmount[13]) + int(tempBinAmount[lastRecipeint]) + int(
tempBinAmount[12 - int(lastRecipeint)])
        tempBinAmount[lastRecipeint] = 0
        tempBinAmount[12 - int(lastRecipeint)] = 0
        playerOne = not(playerOne)
    elif(int(messageCode) >= 0):
        playerOne = not (playerOne)

    return tempBinAmount
```

This function is used for the implementation of the AI agent moves stone distribution. And like player one stone distribution this function also is written based on the mancala game rules. The difference of this portion of code from player one is condition for bonus rules. Here we check if the last recipient bin is AI agent mancala bin or not. If it is AI agent mancala bin we keep the value of playerOne bool as False so that AI Agent gets another bonus move. And also check the condition whether last recipient bin is an empty bin in AI side or not, if so then the AI agent gets the value including the mirror value of the player One side. Or else next move is for the player One.

```
def player_ai_stone_minimax_distribution(choosenBin):
    global messageCode, playerOne, giveawayPile, lastRecipeint, tempBinAmount
    if int(choosenBin) >= 0:
        giveawayPile = int(tempBinAmount[choosenBin])
        tempBinAmount[choosenBin] = 0

    recipient = choosenBin + 1
    while int(giveawayPile) > 0:
        if int(recipient) == 6:
            recipient = 7
        tempBinAmount[recipient] = int(tempBinAmount[recipient]) + 1
        giveawayPile = int(giveawayPile) - 1

        if int(giveawayPile) == 0:
            lastRecipeint = recipient
        else:
            recipient = int(recipient) + 1
            if int(recipient) > 13:
                recipient = 0

    if int(lastRecipeint) ==13:
        playerOne = False
    elif(int(tempBinAmount[lastRecipeint]) == 1 and int(lastRecipeint) > 6):
        tempBinAmount[13] = int(tempBinAmount[13]) + int(tempBinAmount[lastRecipeint]) + int(
tempBinAmount[12 - int(lastRecipeint)])
        tempBinAmount[lastRecipeint] = 0
        tempBinAmount[12 - int(lastRecipeint)] = 0
        playerOne = playerOne
    elif(int(messageCode) >= 0):
        playerOne = playerOne
```

```
    return tempBinAmount
```

This portion of code is for the simulation for the minimax algorithm in case of the move for AI agent. This code is as same as the player_ai_stone_distribution except the playerOne bool value change portion.

```
def end_game_check(playing):
    sideOne = 0
    sideTwo = 0
    for j in range(6):
        sideOne = int(sideOne) + int(binAmount[j])
        sideTwo = int(sideTwo) + int(binAmount[j + 7])

    if(int(sideOne) ==0 or int(sideTwo) == 0):
        playing = False
        binAmount[6] = int(binAmount[6]) + int(sideOne)
        binAmount[13] = int(binAmount[13]) + int(sideTwo)
        for k in range(6):
            binAmount[k] = 0
            binAmount[k + 7] = 0
    return playing
```

In this function we check the end game condition. For this checking we check whether any of the player sides all bins are empty or not. If empty we calculate the total stone numbers for both the sides including their mancala bin. And keep the stones inside their mancala and set the playing bool to False so that there is no next move.

```
def diff():
    global difference, diff_list
    sideOne = 0
    sideAi = 0
    for j in range(7):
        sideOne = int(sideOne) + int(tempBinAmount[j])
        sideAi = int(sideAi) + int(tempBinAmount[j + 7])
```

```
    difference = sideOne - sideAi
    diff_list.append(difference)
    return difference
```

This portion of code gets the difference of the stone numbers for each of the side and return the difference.

```
def minimax(maxify, depth, bins):
    global choosenBin, copyBinAmount, tempBinAmount, binAmount, playerOne
    tempBinAmount = copyBinAmount.copy()

    if depth == 0:
        choosenBin = bins
        return diff()

    if maxify:
        best_score = 999
        for bins in range(7, 13, 1):
            tempBinAmount = binAmount.copy()
            tempBinAmount = player_one_stone_minimax_distribution(bins)
            copyBinAmount = tempBinAmount.copy()
            score = minimax(not maxify, depth-1, bins)
            best_score = min(best_score, score)
        return best_score

    else:
        best_score = -999
        for bins in range(0, 6, 1):
            tempBinAmount = player_ai_stone_minimax_distribution(bins)
            score= minimax(not maxify, depth-1, bins)
            best_score = max(best_score, score)
        return best_score
```

This portion of the code implements the basic minimax algorithm. We used the following

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

pseudocode for the minimax algorithm.
Like the pseudocode we implemented the if condition for the depth, if the depth is zero we return the heuristic value. Then we check for the maxify bool value, if its for finding the player one possible moves we get inside the first if condition and get the temporary bin values for player one move. For getting the heuristic value we recursively call the minimax function with decreasing the depth and altering the maxify bool value. For the False value of maxify we get the temporary bin amount values for the AI agent stone minimax distribution function . And for the heuristic value we call minimax function recursively with decreasing depth and maxify altered.
For the heuristic value of the human player find the minimum of the player one heuristics and for the AI agent heuristic value we get the max value for the AI agent heuristics. In this way we can get the best possible AI agent move.

```
while (playing):
  terminal_message(messageCode, playerOne)
  messageCode = 0

  board()

  # choosenBin = 0
  if playerOne:
    userInput = input("Enter 'q' to end the game: ")
```

```
    choosenBin,messageCode,playing = player_one_user_input(userInput,messageCode,playin
g)
    if playing:
      tempBinAmount = binAmount.copy()
      tempBinAmount = player_one_stone_distribution(choosenBin)
      binAmount = tempBinAmount.copy()

  else:
    tempBinAmount = binAmount.copy()
    copyBinAmount = binAmount.copy()
    minimax(True, 3,  int(choosenBin))

    for i in range(0, 35, 6):
      temp_diff_list.append(diff_list[i])

    min_not_ended_zero = True

    while(min_not_ended_zero):
      min_temp_diff_list = min(temp_diff_list)

      ai_choosen_bin = temp_diff_list.index(int(min_temp_diff_list))

      tempBinAmount = binAmount.copy()
      temp_bin_value_check = tempBinAmount[ai_choosen_bin + 7]
      if int(temp_bin_value_check) == 0:
        temp_diff_list[ai_choosen_bin] = 999
      else:
        tempBinAmount = player_ai_stone_distribution(ai_choosen_bin + 7)
        binAmount = tempBinAmount.copy()
        diff_list.clear()
        temp_diff_list.clear()
        min_not_ended_zero = False



  playing = end_game_check(playing)
```

This is the main segment of the program. Here we check for possible moves, we call the terminal message function for the initial print of the message. Then we print the current game board. Then we check if its for the Player one turn. If so, we take the user input and call the player_one_user_input() function to get the choosen bin number for the player One. Then if we

have possible moves we distribute the stones of the bin by calling player_one_stone_distribution() function.
If the next turn is for AI agent then we get the heuristic values for all the possible AI agent moves. At the for loop inside this segment we collect only the non repeated heuristic values. Inside the while loop we calculate the minimum of the heuristic values because more minimum the value more beneficial for the AI agent to win the game.
After calculating the minimum value we find out whether the bin value is empty or not. If empty we find the next minimum value until we get some stones in the AI agent bins. Then for that choosen bin we distribute the AI agent stones by calling the player_ai_ston_distribution() function. And we do some post processing cleaning.

```
# end of while loop
print("")
print("The Game is over!")
if int(binAmount[13]) < int(binAmount[6]):
   print("Player One has won the game!")
elif int(binAmount[13]) > int(binAmount[6]):
   print("Player Two has won the game!")
else:
   print("The game ended in a Tie!")

i = 0
for element in binAmount:
   binAmount[i] = int(binAmount[i])
   if int(binAmount[i] < 10):
      binAmount[i] = " " + str(binAmount[i])
   else:
      binAmount[i] = str(binAmount[i])
   i = i + 1
# end of for loop

print("")
print("+----+----+----+----+----+----+----+----+")
print("|    | " + binAmount[12] + " | " + binAmount[11] + " | " + binAmount[10] + " | " + binAmo
unt[9] + " | " +
   binAmount[8] + " | " + binAmount[7] + " |    |")
print("| " + binAmount[13] + " |----+----+----+----+----+----| " + binAmount[6] + " |")
print("|    | " + binAmount[0] + " | " + binAmount[1] + " | " + binAmount[2] + " | " + binAmount[
3] + " | " +
   binAmount[4] + " | " + binAmount[5] + " |    |")
print("+----+----+----+----+----+----+----+----+")
```

This is the end of game task. If it's the end of game we calculate who won the game. Print the player as winner and print the final board again. And end the game.

5. Discussion:

During the development part our main intention was to implement the minimax algorithm. When we finished the implementation we were getting errors regarding variable declaration. The scopes of the variables were so recursively implemented that it started to break. So we planned to make all the variable global. In that case during the minimax traversal our main binAmount list were getting changed. To solve that we implemented two layers of bin stones value so its intact until we finalize the choice.

Then another problem starts to arise for the exact choosen bin. That time we were getting error like calculation was going on in the perspective of player One instead of AI agent. Then we did some breakpoint debugging and solved the problem. Another problem we faced is AI agent was too easy to lose. To solve that we increased the depth.

Lastly one big problem arised as sometimes AI agent was getting extra moves even it has no bonus moves. Line by line debugging was done to solve the problem. So finally we found out that as we were using the main player_one_stone_distribution() and player_ai_stone_distribution() function inside minimax algorithm the global playerOne value was changing continuously and sometimes it was deviating form the actual required value. To solve this problem we tried number of methods but finally we came up with the idea of using clone functions for minimax algorithm where there will be no case of playerOne bool variable change. And that solved the problem.

After the final debugging we uploaded both the two player mancala code and vs AI mancala code to git repository. And licensed the repository under MIT license. Here is the link for the git repository:

Mancala Repository.

6. Conclusion:

Before implementation of this game we have no idea about the game theory and game mechanism. Also we didn't know about that for vs player against computer we can implement the AI agent by our own. While doing the project we learned the practical implementation of minimax algorithm and implementing to a game taught us a lot.

We hope to learn more adversarial search algorithms and implement them to a practical scenario. And we will be working on the implementing games including UI instead of playing on the terminal.

We learned a lot from this project and hope to work on more projects like this.