HW3
EE562
Prithvi Shetty

Assignment 3 Kalah Game

Overview:

Kalah is a two – player game which is turn-based.
The board has 12 holes and one goal hole for each player.

Each player has 6 holes on their side assigned to themselves. Each hole has 6 stones at start in each thus there are total 72 stones at the start.

The player whose turn it is chooses one of the 6 holes they own which is not empty. The stones in the hole are held completely and distributed one by one into the subsequent holes moving anti-clockwise, skipping only the goal of the opposite player.

Aim: The final aim is to collect as many stones as possible in their own goal or Kalah.

Rules:

1.The game will end when one of the sides (6 holes) are completely empty.

2.The player exceeds a score of 36

3.If the last stone lands on the goal of the player, the player gets an extra chance.

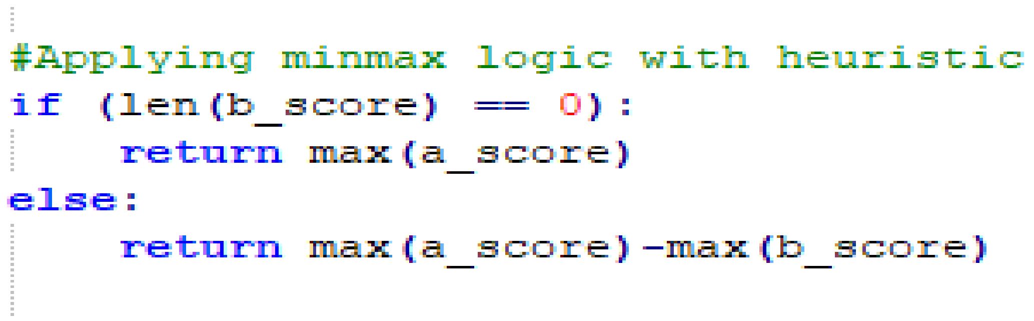
4.If the last stone lands on an empty goal, then the player gets all the stones from the exact opposite hole to his goal.

HW3
EE562
Prithvi Shetty

Heuristic function used:

1. Difference in the maximum score possible by the AI and the maximum score possible by the human.

   Reason: The reason I have chosen this heuristic is that it will consider the best possible move of the AI after taking into account that the human has taken the best possible move to increase its score.



```
#Applying minmax logic with heuristic
if (len(b_score) == 0):
    return max(a_score)
else:
    return max(a_score)-max(b_score)
```

2. The number of stones on each side of the player.

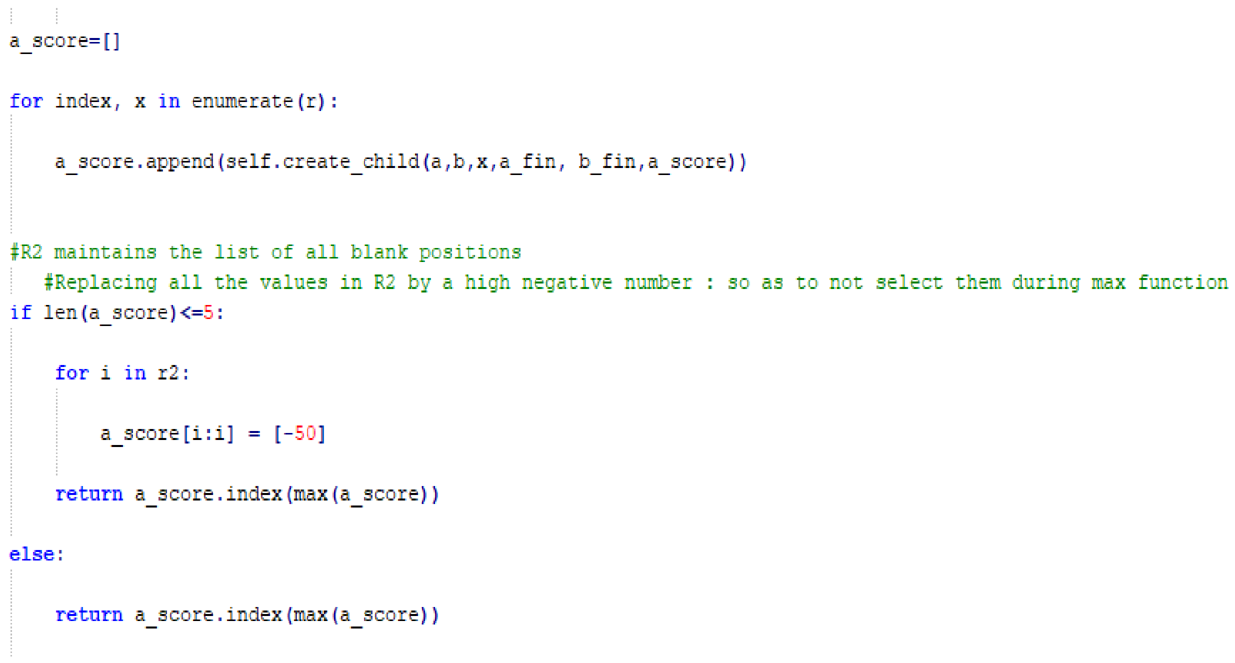
   Reason: The more the number of stones on each side, the more the moves for that player. This directly increases the probability for winning.

HW3
EE562
Prithvi Shetty

Working of the heuristic:

1. The first step is to take the input move by the human and branch out the possible states.

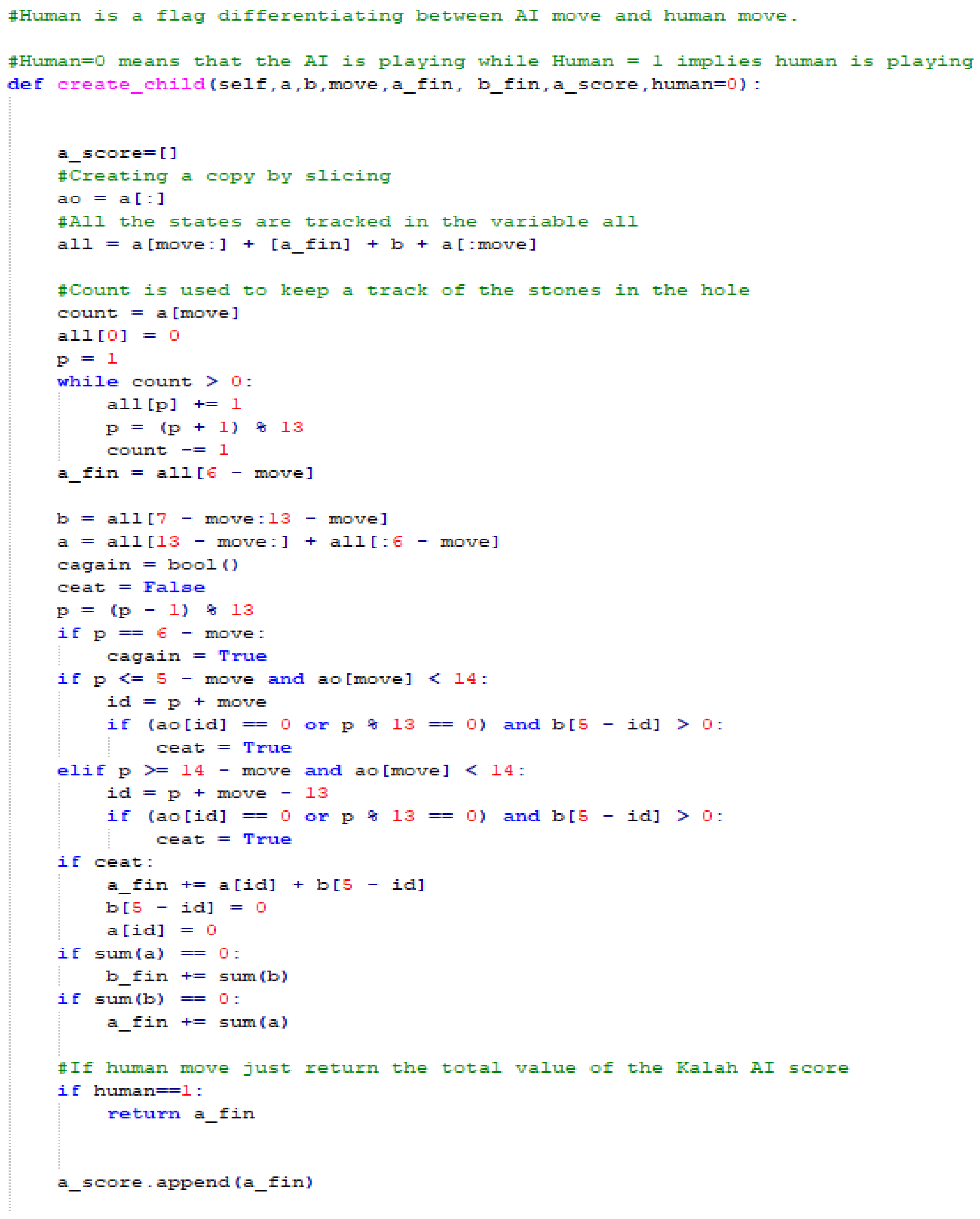
   The blank stones are maintained in another list which is later used for selection of the maximum score and the best possible move.



```
a_score=[]

for index, x in enumerate(r):

    a_score.append(self.create_child(a,b,x,a_fin, b_fin,a_score))


#R2 maintains the list of all blank positions
   #Replacing all the values in R2 by a high negative number : so as to not select them during max function
if len(a_score)<=5:

    for i in r2:

        a_score[i:i] = [-50]

    return a_score.index(max(a_score))

else:

    return a_score.index(max(a_score))
```

HW3
EE562
Prithvi Shetty

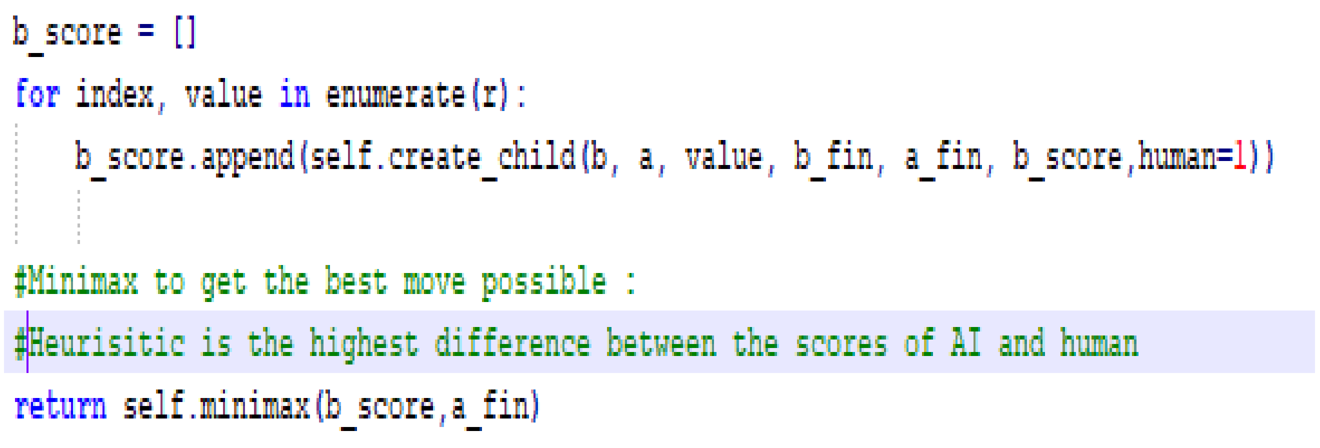
2. All the possible children are created till the depth provided. For example, if the depth is 3. The AI will consider all the possibilities from the action it takes till the next two turns, which is the human turn and the next AI possible move.



```
#Human is a flag differentiating between AI move and human move.

#Human=0 means that the AI is playing while Human = 1 implies human is playing
def create_child(self,a,b,move,a_fin, b_fin,a_score,human=0):


    a_score=[]
    #Creating a copy by slicing
    ao = a[:]
    #All the states are tracked in the variable all
    all = a[move:] + [a_fin] + b + a[:move]

    #Count is used to keep a track of the stones in the hole
    count = a[move]
    all[0] = 0
    p = 1
    while count > 0:
        all[p] += 1
        p = (p + 1) % 13
        count -= 1
    a_fin = all[6 - move]

    b = all[7 - move:13 - move]
    a = all[13 - move:] + all[:6 - move]
    cagain = bool()
    ceat = False
    p = (p - 1) % 13
    if p == 6 - move:
        cagain = True
    if p <= 5 - move and ao[move] < 14:
        id = p + move
        if (ao[id] == 0 or p % 13 == 0) and b[5 - id] > 0:
            ceat = True
    elif p >= 14 - move and ao[move] < 14:
        id = p + move - 13
        if (ao[id] == 0 or p % 13 == 0) and b[5 - id] > 0:
            ceat = True
    if ceat:
        a_fin += a[id] + b[5 - id]
        b[5 - id] = 0
        a[id] = 0
    if sum(a) == 0:
        b_fin += sum(b)
    if sum(b) == 0:
        a_fin += sum(a)

    #If human move just return the total value of the Kalah AI score
    if human==1:
        return a_fin


    a_score.append(a_fin)
```

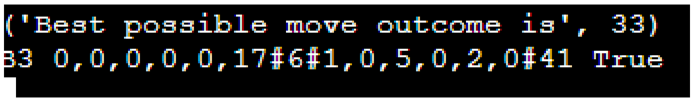
HW3
EE562
Prithvi Shetty

3. After the possible states have been created. The minmax function finds out if the value returned satisfies the best possible heuristic and correspondingly returns the best possible move for the AI.



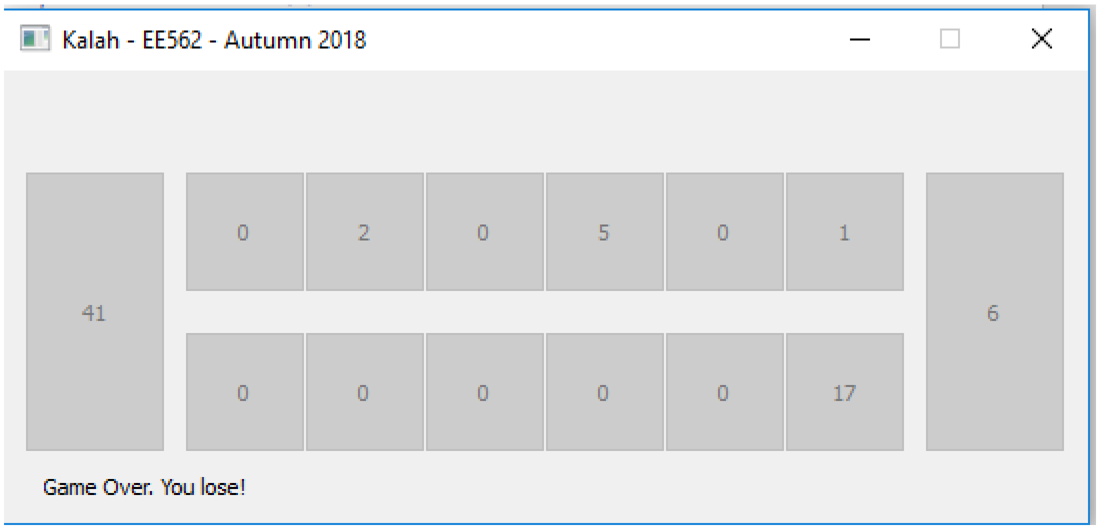
```
b_score = []
for index, value in enumerate(r):
    b_score.append(self.create_child(b, a, value, b_fin, a_fin, b_score,human=1))

#Minimax to get the best move possible :
#Heurisitic is the highest difference between the scores of AI and human
return self.minimax(b_score,a_fin)
```

3.Example:



```
('Best possible move outcome is', 33)
33 0,0,0,0,0,17#6#1,0,5,0,2,0#41 True
```

The final move of 33 was chosen which resulted in the AI winning the game against the human.



Game Over. You lose!

4.**Experiments:**

Infrastructure : 2.4 GHz CoreI5 , 16 GB RAM.

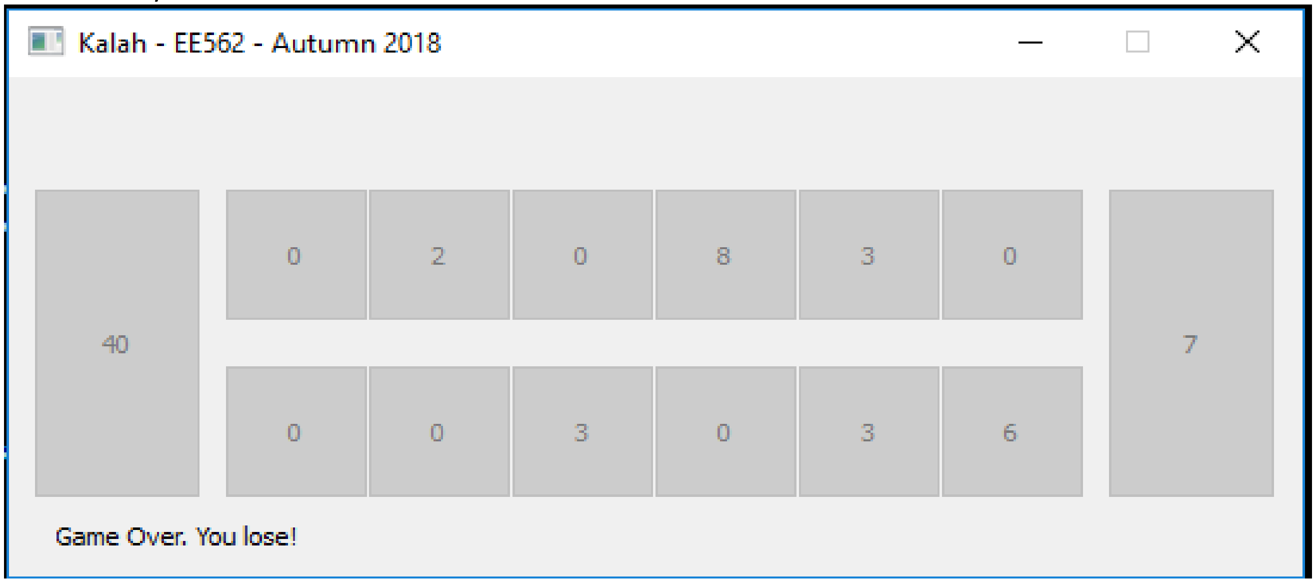
Level	Time taken for each move
1	.0009s
2	.0045
3	.01
4	.03

Heuristics

I considered my first heuristic and competed it against a heuristic of just choosing the maximum score for best possible move and not the difference between human and AI scores.

Result: The AI was better and won every time against the old AI.

HW3
EE562
Prithvi Shetty



Game Over. You lose!

Deduction:

1)Heuristic of the game must always consider the worst possible move of the opponent for the AI to perform the best.

2) As depth increase, the performance decreases as well. Alpha beta pruning plays a great role to increase the efficiency and skip unwanted traversals.