AI ON CHECKERS GAME

SHUBHAM SONI, 5th semester, B.Tech (IT)(IIT2018177)

Indian Institute of Information Technology Allahabad, Allahabad, India

Abstract:. In this paper we have made a GUI using pygame library of python and implemented a virtual opponent using minimax algorithm and AI .

I. INTRODUCTION

Checkers is a game for two players in which involve diagonals moves of uniform game pieces and mandatory captures by jumping over opponent pieces. Checkers is played on an 8×8 board.

The question is:-
Make an automated player for the game of checkers. Use a GUI to test your computer player. Use a good heuristic function that you can design. (code+ report)

II. General rules in checkers(basic rules)

It is a game played by two players on opposite side of the board. The pieces can be moved on the diagonal dark sqares. If a player wants to kill the piece of the other he needs to make a jump over it when opponent's piece is on the adjecent digonal square. One piece may jump over many opponents piece in a single move.

The pieces are of two types:

Normal Piece: It is the status of a piece untill it reaches the last row of other side. It can move only in forward daigonal.

King Piece: It is the status of the piece once it reaches the last row on other side. It can move in all possible daigonal places.

III. ALGORITHM DESCRIPTION

For GUI and window creation i used:

```
import pygame
win = pygame.display.set_mode((800,800))
pygame.display.set_caption('Checkers GUI')
```

for copying the array to change it i used

```
from copy import deepcopy
```

for delaying the move between computer player and human player i used

```
import time
```

initial state of the pieces is

```
[[(B,F),(W,F),(B,F),(W,F),(B,F),(W,F),(B,F),(W,F)],
 [(W,F),(B,F),(W,F),(B,F),(W,F),(B,F),(W,F),(B,F)],
 [(B,F),(W,F),(B,F),(W,F),(B,F),(W,F),(B,F),(W,F)],
 [(B,F),(B,F),(B,F),(B,F),(B,F),(B,F),(B,F),(B,F)],
 [(B,F),(B,F),(B,F),(B,F),(B,F),(B,F),(B,F),(B,F)],
 [(R,F),(B,F),(R,F),(B,F),(R,F),(B,F),(R,F),(B,F)],
 [(B,F),(R,F),(B,F),(R,F),(B,F),(R,F),(B,F),(R,F)],
 [(R,F),(B,F),(R,F),(B,F),(R,F),(B,F),(R,F),(B,F)]]
```

Where B = blank squares , W = sqaures with white pieces and R = squares with red pieces.

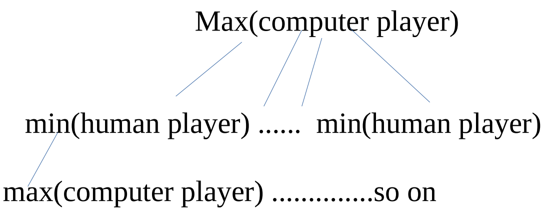
Here F implies that no piece is king initially.

So min max is finding the best move based on

```
def heuristic(piece):
cost = 0
for i in range(0,8):
for j in range(0,8):
if piece[i][j][0] == WHITE:
if piece[i][j][1]:
            for move in get_valid_pos(piece,i,j):
             cost+=2
            else:
            for move in get_valid_pos(piece,i,j):
            cost+=1
            elif piece[i][j][0] == RED:
            if piece[i][j][1]:
            for move in get_valid_pos(piece,i,j):
            cost-=2
            else:
            for move in get_valid_pos(piece,i,j):
            cost-=1
```

So in simple terms cost is total moves the AI have subtracted from total moves human player have.

For computer player , we implemented an algorithm based on which it may the computer player thinks and then operates the best move upto the depth of 3(in my code).



Max(computer player)

min(human player) ...... min(human player)

max(computer player) ....so on

so the pseudo code for this is like:-

```
def minimax(piece,depth,maxplayer):
   piece = deepcopy(piece)
   if depth == 0 or any one won:
     return heuristic(piece),piece
   if maxplayer:
       max_val = -1000000
        best_state = None
        for all pieces of computer player
           for move in valid_moves of each piece:
             x,y = move
piece[x][y],piece[i][j]=piece[i][j],piece[x][y]
for pos in valid_chance[move]{
             xx,yy = pos
          piece[xx][yy]=(BLACK,False)
}
state = deepcopy(piece)
temp = minimax(piece,depth-1, False)[0]
piece[x][y],piece[i][j]=piece[i][j],piece[x][y]
if max_val < temp:
    max_val = temp
    best_state = state
return max_val, best_state

else{
                          min_val = 1000000
                          best_state = None
for i in range(0,8):
   for j in range(0,8):
   if piece[i][j][0] == RED:
valid_chance = get_valid_pos(piece,i,j)
for move in valid_chance:
   x,y = move
piece[x][y],piece[i][j]=piece[i][j],piece[x][y]
for pos in valid_chance[move]:
xx,yy = pos
piece[xx][yy]=(BLACK,False)
state = deepcopy(piece)
temp = minimax(piece,depth-1, True)[0]
piece[x][y],piece[i][j]=piece[i][j],piece[x][y]
if min_val > temp:
   min_val = temp
   best_state = state
return min_val, best_state
```

and i called the function minimax(self.piece,3,True) when the human player have done with his part.

IV. ALGORITHM AND ANALYSIS

time complexity :

as here the algorithm is cheking every aspect of moves possible , so in worst case :-

all pieces are supposed to be king to get the worst case:

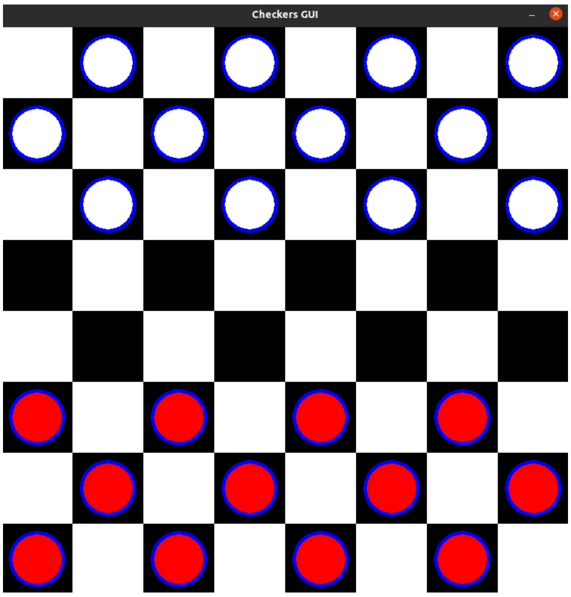
for every piece of player :
for all moves of that piece :

this combination of for loop takes almost 4*n where n is the number of pieces of the player.

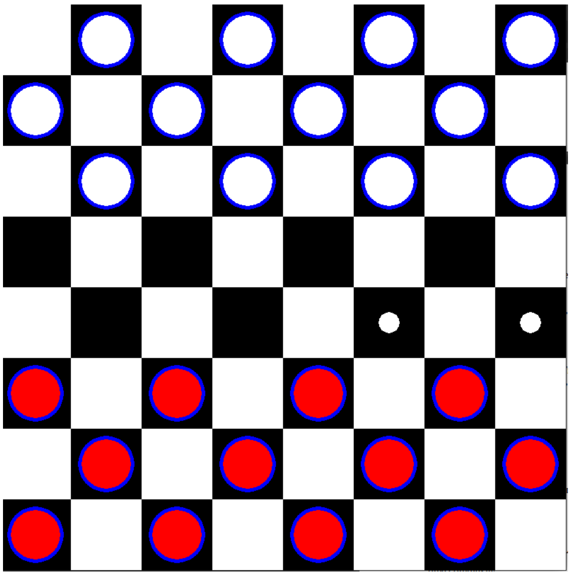
Now inside every iteration 4*n we again go through 4*n * 4*n ...... upto(depth time) so it is O(n^depth) without alpha beta pruning with aplha beyta pruning it may reduce .

V. Some sample of the code output:

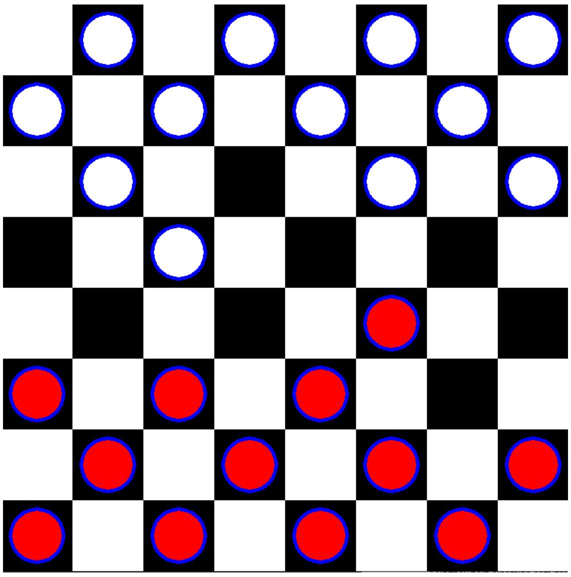
Initial state:



When human player clicks the pieces:



When human player move the piece:



VI. CONCLUSION

From this paper we can conclude that the alpha-beta prunning is helpuful in reducing the time complexity as well as space complexity of the game playing algorithm like min max approach.

VII. REFERENCES

[1]https://www.javatpoint.com/mini-max-algorithm-in-ai

[2]https://en.wikipedia.org/wiki/Draughts

[3]https://www.pygame.org/news