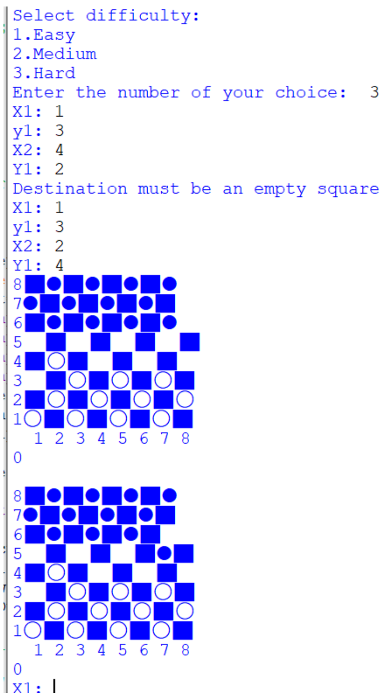
Knowledge & Reasoning Programming Project

Introduction

In this project I have implemented a program that simulates the game of checkers, the game is text based, making use of Unicode characters to display the board and pieces. The opponent of the player is a minimax search algorithm that seeks to minimise its losses and maximise its gains on the board. The difficulty of the game is adjustable by changing the depth to which the minimax algorithm recurses.



```
Select difficulty:
1.Easy
2.Medium
3.Hard
Enter the number of your choice:  3
X1: 1
y1: 3
X2: 4
Y1: 2
Destination must be an empty square
X1: 1
y1: 3
X2: 2
Y1: 4
8■●■●■●■●
7●■●■●■●■
6■●■●■●■●
5 ■ ■ ■ ■
4■○■ ■ ■
3 ■○■○■○■
2■○■○■○■○
1○■○■○■○■
  1 2 3 4 5 6 7 8
0

8■●■●■●■●
7●■●■●■●■
6■●■●■●■
5 ■ ■ ■●■
4■○■ ■ ■
3 ■○■○■○■
2■○■○■○■○
1○■○■○■○■
  1 2 3 4 5 6 7 8
0
X1: 
```

Figure 1 A screenshot showing gameplay at the beginning

Three different classes are implemented and used for this program. There is a class for the board, moves, and checkers pieces.

The board class stores the layout of the board, and piece locations within the board, as well as handling anything related to the layout of the board in its methods.

The piece class stores relevant variables for each game piece, such as the player a piece is associated with, the character representing the piece when the board is displayed, and whether the piece is a king, as well as methods to update attributes of the piece (e.g. when the piece becomes a king)

The move class is the most complex of all of the classes. It holds the starting and ending x and y positions of a proposed move, translates player co-ordinate entries to the index position in the board state representation , validates, and executes each move.

I chose not to implement a class for the AI player, since its implementation was possible within a single function, and I didn't want to over complicate things for myself.

Description of Program Functionality

Gameplay

This program is an implementation of a turn based simulation of checkers, figure 1 shows the starting co-ordinate entry for the human player, the first board displayed shows the move has been executed and the AI move is shown below that.

The difficulty is entered by the user when the program starts. The difficulty is regulated using the depth of the minimax algorithm. With a low depth, one would expect the AI to be more greedy when it comes to taking opposing pieces, much as an inexperienced player of checkers would be.

The depth increases with difficulty so the AI player can maximise its gains over more turns given the human player makes the best choice of move.

The structure of the game is outlined under Main in the appendices, while the game is running, the human player takes a turn by inputting the co-ordinate of the piece they would like to move and the co-ordinate of the square they want the piece to move to. These co-ordinates are translated to the index positions for the board state representation in the Move class (Appendices: Move: Line 117 – 3(on next page)).

Search Algorithm

My search algorithm is an implementation of the minimax algorithm, that searches for the move that, provided the opposing player makes the best possible moves on their turns, selects the move that will guarantee the best possible score for the player over the series of turns that the algorithm searches.

Code for the search algorithm is available in Appendices: Main: lines 10-50.

The algorithm receives as parameters, the current board state (whether this is the true state of the game or a simulated board affected by simulated moves during the search), the player (to determine whether it should maximise or minimise the score, the depth(the number of moves ahead that the algorithm will evaluate) and the values needed for alpha-beta pruning to make the algorithm more efficient by allowing it to guarantee when pruning requirements are satisfied that it has the best possible move already evaluated, increasing the runtime efficiency of the algorithm.

When the algorithm is called for a given player, it calls a method of the board object that it was passed (Appendix:Board:Lines 84-96) that returns every possible move the player can make.

Each of these moves are executed on the simulated board in a for loop, and the board is passed to the minimax function to evaluate moves that the opposing player can make.

Once the depth is zero, the minimax function returns the simulated board score (calculated by Appendices:Board:Lines50-67). When the function receives the scores from the recursion, it evaluates the scores to find the minimising or maximising move, and returns the move and the score.

Validation of Moves

There are three sets of tests that my validation system uses to validate a given move.

The first of these are hard coded basic test that ensures players don't enter moves that, for example, attempt to move the other players piece, start at an empty square, or start and/or finish at a black square, and that the destination square is empty. If these tests aren't passed, the validation method returns False, indicating the move was invalid, an empty list of skips, and a string to be displayed that prompts the user on why their move was invalid.(Appendices:Images:1) (The code for these tests is at Appendices:Move:Lines70-85)

If the move passes these tests, it is passed to the pathsearch method (Appendices:Move:Lines 41-60). The pathsearch function handles validity checks for all moves that don't involve taking a player(skipping). To do this, it checks that the adjacent squares that are forward of the piece in its assigned direction of travel are clear, if this is true, it returns True for move validity, an empty skips list and an empty string.

If the move involves skipping over opposing players pieces, the move is passed to a depth first recursive search method (Appendices:Move:Lines 13-38) that determines if a possible path can be established between the moves starting and destination square. A skip always involves a move of two squares to the left or right and two squares forward (depending on the direction of travel determined by the factor variable). This function receives the pieces current position and the proposed destination position, if the end position is reached, the function returns True for valid move an array of skipped piece locations (half way between start and end position of each leg) and an empty string(which would otherwise be the prompt if an invalid move was passed. If traversing down the left side doesn't return true, the algorithm attempts to traverse right. If the values for current position fall outside the bounds of the board dimensions, the function returns False for validity, an empty skips list and a relevant prompt for the user. (Appendices:Images:2)

All player moves are validated this way, and the function is used in generating all valid moves for a given player for the purposes of the AI player. The validation function is run a second time on the prevailing move from the minimax algorithm, not for the purposes of validation, but for determining which pieces are skipped in the move so that they can be removed when the move is executed.

It is worth noting that a flaw of my validation implementation is that you cannot take a piece at the leftmost or rightmost squares of the board, so any piece in these areas is safe from being taken. This is because my validation system assumes a skip leg of a move is a one directional move. An attempt to take a piece on the edge would flag in the validation system as impossible since taking the piece would require the leg of the move ending out of the bounds of the board.

Other features

Multi leg capturing:

This implementation allows for multi leg capturing as the section of the validation system responsible for moves involving skips will recursively search all possible directions the piece can take for each possible leg until the proposed end square is reached. Provided the end square is validly reachable, a multi leg move is valid.

This works for both the human user and the AI, since the method that determines all valid AI moves tests the validity of the AI moving all pieces from their start location to all other squares of the board.

King conversion at baseline:

There is a section of code in the move execute method (Appendices:Move:Lines 112-115) that crowns a piece when it reaches the baseline of the opposing player.

Regicide:

In the execute method, if one of the pieces in the skip list (pieces to be deleted) is a king, the piece that took the player is instantly crowned. (Appendices:Move:Lines 100-101, 105-106)

Human – Computer interface

Board display:

Figure 1 shows the display of the board and numbers to help the user determine the correct co-ordinates to enter. It is clear that the board is not uniformly spaced. This is because I had trouble finding Unicode characters for the pieces that were equal diameters, however in the event that I did find characters of equal width, the board would display uniformly.

The display is updated between moves:

After any move is executed, the display method is called to display the boards current state.

Appendices

Board

```python
from piece import Piece
from move import Move
class Board():

    def __init__ (self, dimension):
        initstate =[]
        for i in range(0,dimension):
            initstate.append([])
            for j in range(0,dimension):
                if (i%2==0 and j%2 == 0)or(i%2==1 and j%2 == 1) :#determine black squares and white squares
                    initstate[i].append("█")
                else:
                    initstate[i].append("  ")
        self.state=initstate
        self.dimension = dimension
    def getState(self):#return 2d array representing the board at current state of play
        return self.state

    def display(self): #print the board in ascii/unicode characters

        for i in range(0,len(self.state)):
            tempstring=str(self.dimension - i)#string for horizontal display
            for j in self.state[i]:
                if j != "  " and j!="█":
                    tempstring=tempstring+j.getChar()
                else:
                    tempstring=tempstring + j
            print (tempstring+"")
        print("  1 2 3 4 5 6 7 8")#print x axis chars

    def populate(self):#set board to regular state for beginning a game
        for i in range(0,self.dimension):
            for j in range(0,self.dimension):
                if i < 3 and self.state[i][j]=="  ":#set black pieces
                    self.state[i][j]=Piece("b", False)
                if i>4 and self.state[i][j]=="  ":#set white pieces
                    self.state[i][j]=Piece("w", False)
    def piecesLeft(self,player):#how many pieces of a given player remain?
        string="string"
        count=0
        debugcount=0
        for i in self.state:
            for j in i:
                debugcount+=1
                if j!="  " and j!= "█":
                    if j.getPlayer == player:
                        count=count+1
        print(count)
        return count
    def score(self):#score evaluation
        bcount=0#initialise score counts
        wcount=0
        for i in self.state:#iterate through whole board state
            for j in i:
                if type(j)!=type(" "):#if not a string (i.e. if given square is a piece)
                    if j.getPlayer()=="w":#if piece is white
                        if j.getKing():#if piece is a king
                            wcount=wcount+1.5
                        else:
                            wcount=wcount+1
                    elif j.getPlayer()=="b":#if piece is black
                        if j.getKing():#if piece is a black king
                            bcount=bcount+1.5
                        else:
                            bcount=bcount+1
        count = wcount-bcount
        return count
    def gameOver(self):#returns true if one player has no pieces left
        bcount=0
        wcount=0
        for i in range(0,self.dimension):
            for j in range(0,self.dimension):
                if self.state[i][j]!="  " and self.state[i][j]!="█":
                    if self.state[i][j].getPlayer()=="w":
                        wcount=wcount+1
                    else:
                        bcount=bcount+1
        if wcount==0:
            return "b"
        elif bcount==0:
            return "w"
        else:
            return 0
    def getvalidmoves(self,player):#finds every possible valid move for a give player
        moves=[]
        for i in range(0,self.dimension):#search for players pieces
            for j in range(0,self.dimension):
                if self.state[i][j]!="  " and self.state[i][j]!="█":
                    if self.state[i][j].getPlayer()==player:
                        for k in range(0,self.dimension):#generate a move starting at the pieces position
                            for l in range(0,self.dimension):
                                move=Move(i,j,k,l)
                                valid,skips,string=move.validate(player,self)#check move is valid
                                if valid:
                                    moves.append([move,skips])#add valid moves to valid moves array
        return moves
```

Piece

```python
class Piece():
    def __init__(self,player,king):
        self.player=player
        self.king=king
        if player == "b":#set player characters and characters for display
            self.char="●"
        elif player =="w":
            self.char = "◯"
        else:
            self.char = "broken"

    def getPlayer(self): #get player
        return self.player
    def getKing(self):#return king status
        return self.king
    def getChar(self):#get character for display
        return self.char
    def crown(self):#converts player to king
        self.king =True
```

Move

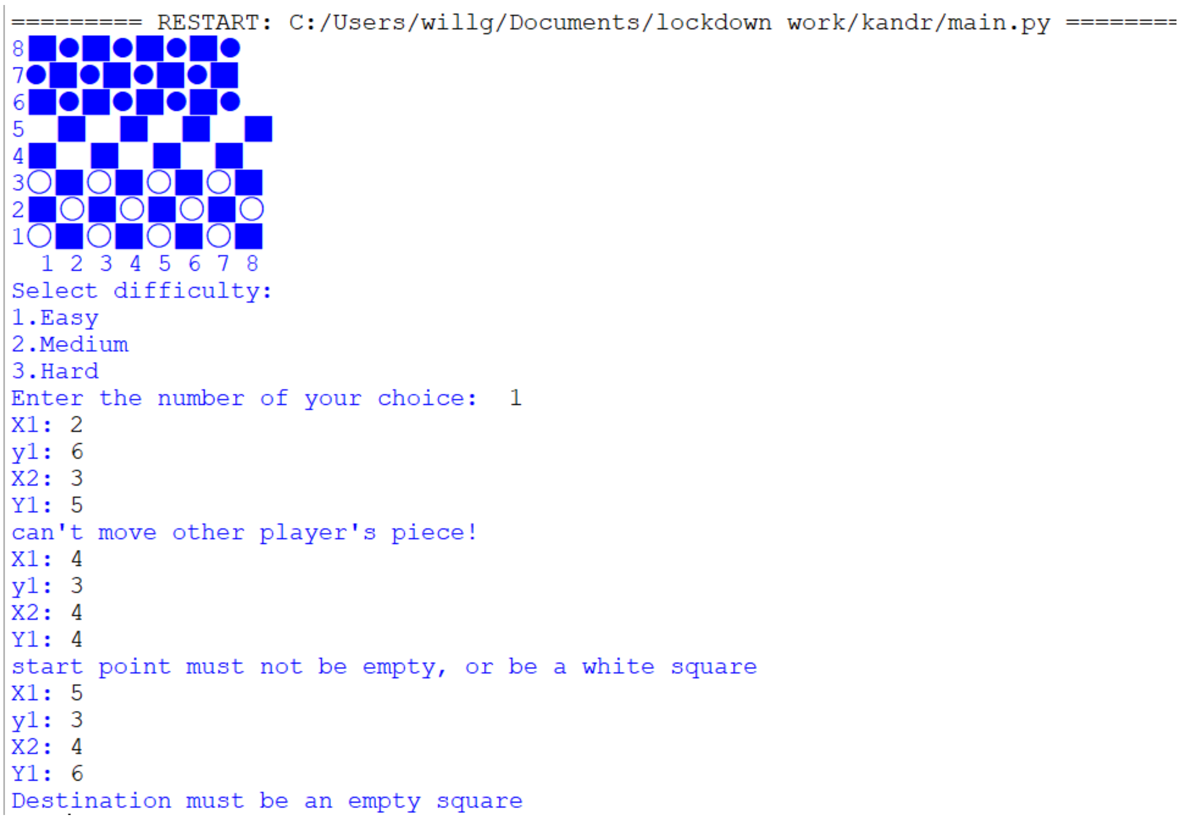
```python
from piece import Piece
#from board import Board
class Move:
    def __init__ (self,x1,y1,x2,y2):
        self.x1=x1
        self.y1=y1
        self.x2=x2
        self.y2=y2

    def getCoOrds(self):#return coords (only used in debugging)
        return self.x1,self.y1,self.x2,self.y2

    def skipsearch(self,state,player,factor,currentpos,endpos):#a depth first search for valid routes that involve taking opposing player piece
        found = False
        if (0>currentpos[0]or 7<currentpos[0])or(0>currentpos[1]or 7<currentpos[1]):
            return False,[],"no valid path"
        if currentpos==endpos:
            return True, [],"  "
        else:
            leftshift = [currentpos[0]+(2*factor),currentpos[1]-2]
            rightshift= [currentpos[0]+(2*factor),currentpos[1]+2]
            #print(currentpos)
            if min(currentpos[0]+(1*factor), currentpos[1]-1)>-1 and max(currentpos[0]+(1*factor), currentpos[1]-1)<8:#traverse left
                if type(state[currentpos[0]+(1*factor)][currentpos[1]-1])!=type("  "):
                    if state[currentpos[0]+(1*factor)][currentpos[1]-1].getPlayer()!=player:
                        path,skips,string =self.skipsearch(state,player,factor,leftshift,endpos)#recurse for multi leg moves
                        if path:
                            skips.append([currentpos[0]+(1*factor),currentpos[1]-1])
                            return True, skips, "  "
            if min(currentpos[0]+(1*factor), currentpos[1]+1)>-1 and max(currentpos[0]+(1*factor), currentpos[1]+1)<8:#traverse right
                if type(state[currentpos[0]+(1*factor)][currentpos[1]+1])!=type("  "):
                    if state[currentpos[0]+(1*factor)][currentpos[1]+1].getPlayer()!=player:
                        path,skips,string=self.skipsearch(state,player,factor,rightshift,endpos)#recurse for multi leg moves
                        if path:
                            skips.append([[currentpos[0]+(1*factor),currentpos[1]+1]])
                            return True, skips, "  "

        return False,[],"no valid path"


    def pathSearch(self,state,player):#determines if the move requires taking opposing pieces
        currentpos=[self.x1,self.y1]
        endpos=[self.x2,self.y2]
        reached=False
        if player == "b":#factor regulates the direction up or down the board the piece can move
            factor=1
        else:
            factor=-1
        if state[self.x1][self.y1].getKing():#if the player is a king, the factor depends on the direction of the move itself
            if currentpos[0]<endpos[0]:
                factor=1
            else:
                factor=-1
        #print(self.y2,self.y1+(1*factor), self.x2,self.x1+1,self.x2 , self.x1-1)
        if (endpos[0]==currentpos[0]+(1*factor))and(endpos[1]==currentpos[1]+1 or endpos[1] == currentpos[1]-1):
            return True,[],"  "
        else:
            #print(state,player,factor,currentpos,endpos)
            reached,skips,string= self.skipsearch(state,player,factor,currentpos,endpos)#enter search for moves that involve skipping a piece
            return reached,skips,string

    def validate(self, player, board):
        state=board.getState()
        valid=True
        x1=self.x1
        y1=self.y1
        x2=self.x2
        y2=self.y2

        if min(x1,x2,y1,y2)<0 or max(x1,x2,y1,y2)>8:#ensure entered coords are within the boards dimensions
            return False,[],"must be within the dimensions of the board"
        if state[x1][y1]=="  " or state[x1][y1]=="█": #start must not be empty, must be a playable square

            return False,[],"start point must not be empty, or be a white square"
        elif state[x2][y2]!="  ": #end point must be empty

            return False,[],"Destination must be an empty square"
        elif state[x1][y1].getPlayer()!=player:#player cannot move other players piece

            return False,[], "can't move other player's piece!"

        else:#if above tests are passed, search for path
            valid,skips,err = self.pathSearch(state,player)

            return valid,skips,err






    def execute(self, board,skips):#executes a move when given a board state
        state=board.getState()
        #print(state)
        for skip in skips:#remove skipped pieces from the board
            #print (skip)
            if type(skip[0])==type([1,2]):
                skip=skip[0]
                if type(state[skip[0]][skip[1]])!=type(" "):
                    if state[skip[0]][skip[1]].getKing():#if piece takes a king, the piece is crowned instantly
                        state[self.x1][self.y1].crown()
                    state[skip[0]][skip[1]] = "  "
            else:
                if type(state[skip[0]][skip[1]])!=type(" "):
                    if state[skip[0]][skip[1]].getKing():
                        state[self.x1][self.y1].crown()
                    state[skip[0]][skip[1]] = "  "

        state[self.x2][self.y2] = state[self.x1][self.y1]#move player to destination
        state[self.x1][self.y1]="  "#clear starting square
        #print(state)
        if self.x2==0 and state[self.x2][self.y2].getPlayer()=="w":#if piece reaches king baseline, crown it
            state[self.x2][self.y2].crown()
        if self.x2==7 and state[self.x2][self.y2].getPlayer()=="b":
            state[self.x2][self.y2].crown()
        return state
    def translateCoOrds(self):#turns input coords into correct format to index board state
        x1=self.x1
        x2=self.x2
        y1=self.y1
        y2=self.y2
        self.y1=x1-1
self.y2=x2-1
self.x1=(y1*-1)+8
self.x2=(y2*-1)+8
```

Main (and minimax function)

```python
from board import Board
from piece import Piece
from move import Move
from copy import deepcopy
import random
board=Board(8)



def minimax(board,depth,alpha,beta,maximum):
    boardcopy=deepcopy(board)#copy board to avoid changing game board object
    if depth==0 or board.gameOver():#base case
        return board.score(),0
    if maximum:
        bestmove=[]
        val=float('-inf')
        for move in boardcopy.getvalidmoves("w"):#get all valid moves for white
            boardcopy=deepcopy(board)
            move[0].execute(boardcopy,move[1])
            value,prevailingmove=minimax(boardcopy,depth-1,True,True,False)#recurse for next set of moves
            if val ==value:#if values are the same, pick a random one so game isn't deterministic
                i=random.randint(0,1)
                if i == 0:
                    bestmove=move[0]
            if val<value:#if value greater, return the value
                val=value
                alpha=value
                bestmove=move[0]
            if alpha >= beta:
                break
        return val,bestmove
    else:
        bestmove=[]
        val=float('inf')
        for move in boardcopy.getvalidmoves("b"):#get all valid moves for black
            boardcopy=deepcopy(board)
            move[0].execute(boardcopy,move[1])
            value,prevailingmove=minimax(boardcopy,depth-1,True,True,True)
            if val ==value:#if values are the same, pick a random one so game isn't deterministic
                i=random.randint(0,1)
                if i == 0:
                    bestmove=move[0]
            if val>value:#if value greater, return the value
                val=value
                beta=value
                bestmove=move[0]
            if beta<=alpha:
                break

        return val,bestmove

new = Board(8)#initialise new board
new.populate()
new.display()

depth=0
ease=int(input("Select difficulty:\n1.Easy\n2.Medium\n3.Hard\nEnter the number of your choice:  "))#select difficulty
if ease == 1:
    depth=2
elif ease == 2:
    depth=5
elif ease == 3:
    depth =7
else:
    print("restart program and enter valid number")


game=True
while game and depth:#game loop
    correct=False
    while correct==False:#loop in case of invalid input
        x1=int(input("X1: "))
        y1=int(input("y1: "))
        x2=int(input("X2: "))
        y2=int(input("Y1: "))
        move=Move(x1,y1,x2,y2)
        move.translateCoOrds()
        valid,skips,string =move.validate("w",new)
        if valid:
            correct=True
        else:
            print(string)#print reason why not valid input
    if valid:
        move.execute(new,skips)#execute player move
        new.display()
        print(new.score())
    end=new.gameOver()#if game over end loop
    if end=="b":
        print("Black wins!")
        break
    elif end=="w":
        print("White wins!")
        break

    else:
        print(string)
    val,move=minimax(new,depth,float('-inf'),float('inf'),False)#AI search for best move
    valid,skips,string=move.validate("b",new)#validate move and get skips
    if valid:#if valid execute
        move.execute(new,skips)
    else:
        print(string)
    new.display()#display board after AI move
    print(new.score())
    end=new.gameOver()#if game over break out of loop
    if end=="b":
        print("Black wins!")
        break
    elif end=="w":
        print("White wins!")
        break
```

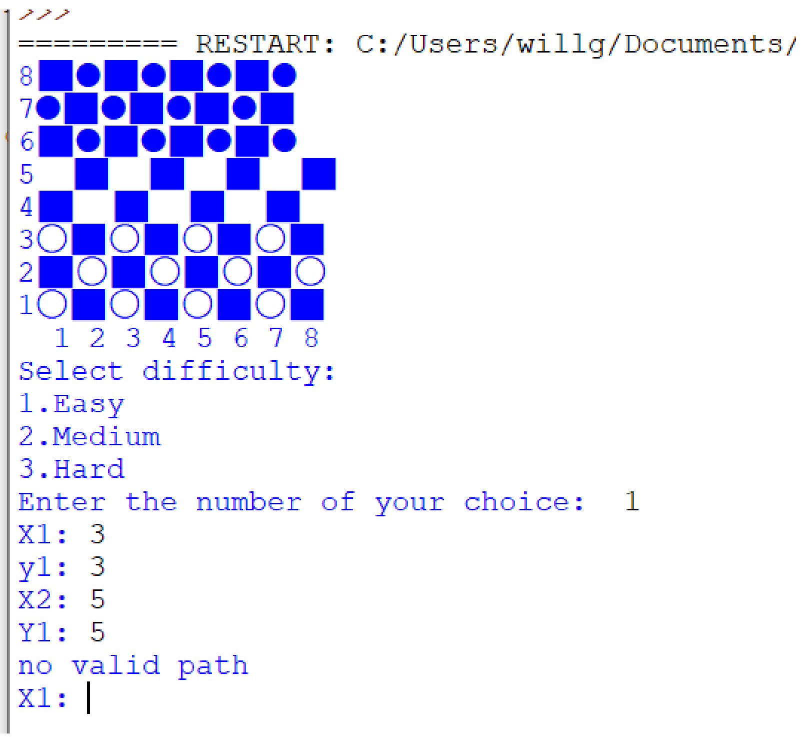
Images

1) Basic validity test fail cases



```
========= RESTART: C:/Users/willg/Documents/lockdown work/kandr/main.py ========
8■●■●■●■●
7●■●■●■●■
6■●■●■●■●
5 ■ ■ ■ ■
4■ ■ ■ ■
3○■○■○■○■
2■○■○■○■○
1○■○■○■○■
 1 2 3 4 5 6 7 8
Select difficulty:
1.Easy
2.Medium
3.Hard
Enter the number of your choice:  1
X1: 2
y1: 6
X2: 3
Y1: 5
can't move other player's piece!
X1: 4
y1: 3
X2: 4
Y1: 4
start point must not be empty, or be a white square
X1: 5
y1: 3
X2: 4
Y1: 6
Destination must be an empty square
```

2) Invalid case for pathfinder(No blue piece in between start and end square)



```
>>>
========= RESTART: C:/Users/willg/Documents/
8■●■●■●■●
7●■●■●■●■
6■●■●■●■●
5 ■ ■ ■ ■
4■ ■ ■ ■
3○■○■○■○■
2■○■○■○■○
1○■○■○■○■
 1 2 3 4 5 6 7 8
Select difficulty:
1.Easy
2.Medium
3.Hard
Enter the number of your choice:  1
X1: 3
y1: 3
X2: 5
Y1: 5
no valid path
X1: 
```