**Artificial Intelligence Course**

**Assignment 03**



**Dr.Hashim Yasin**

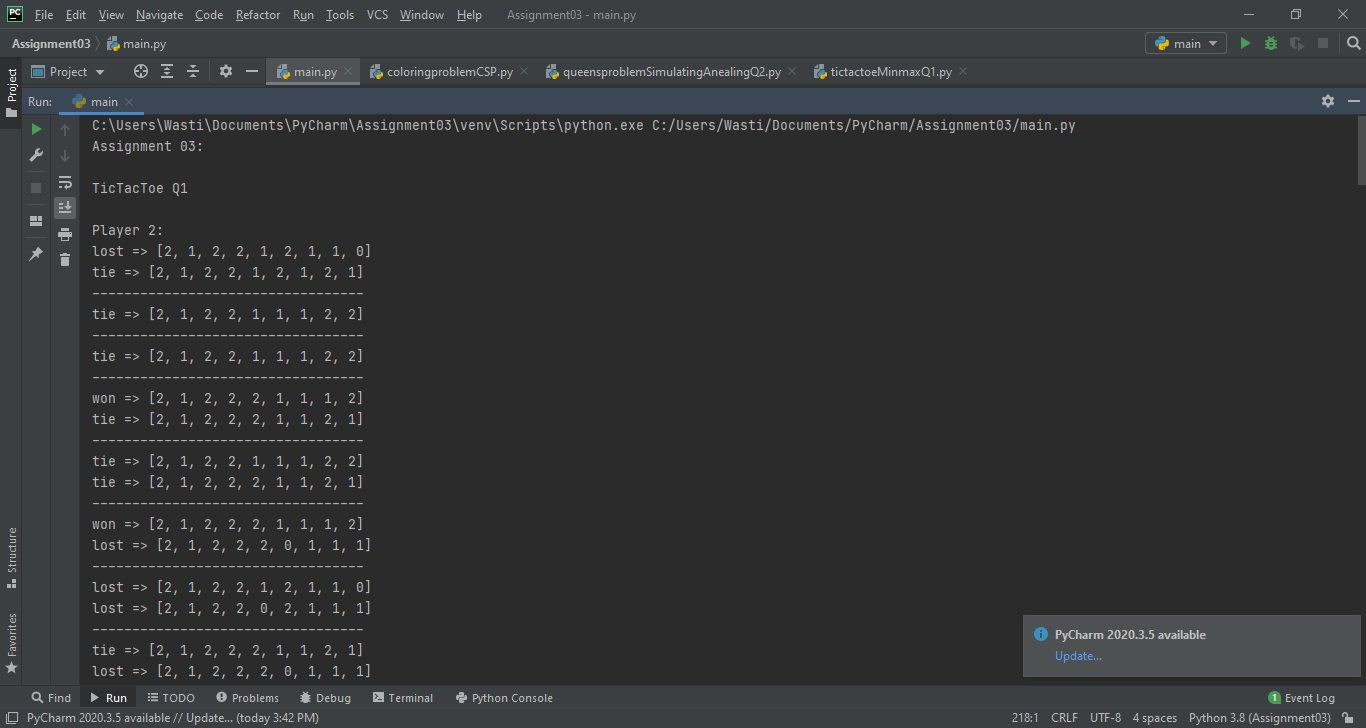
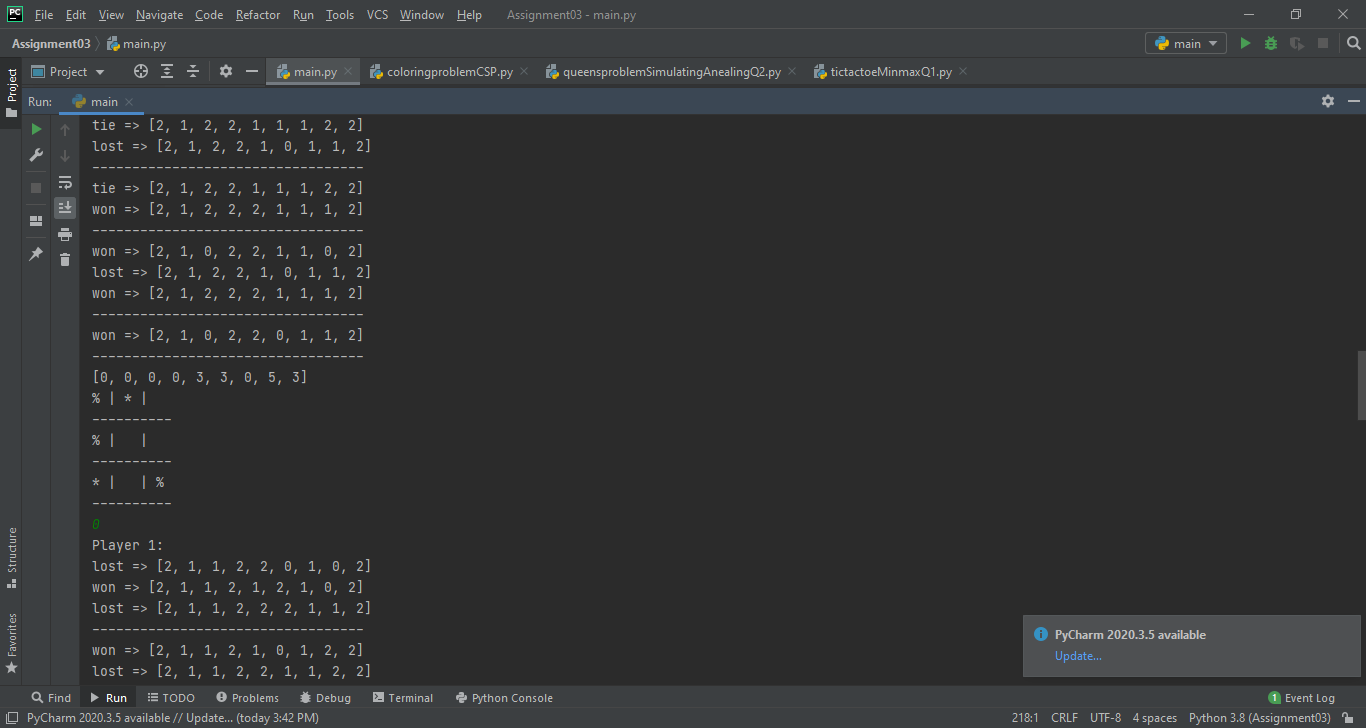
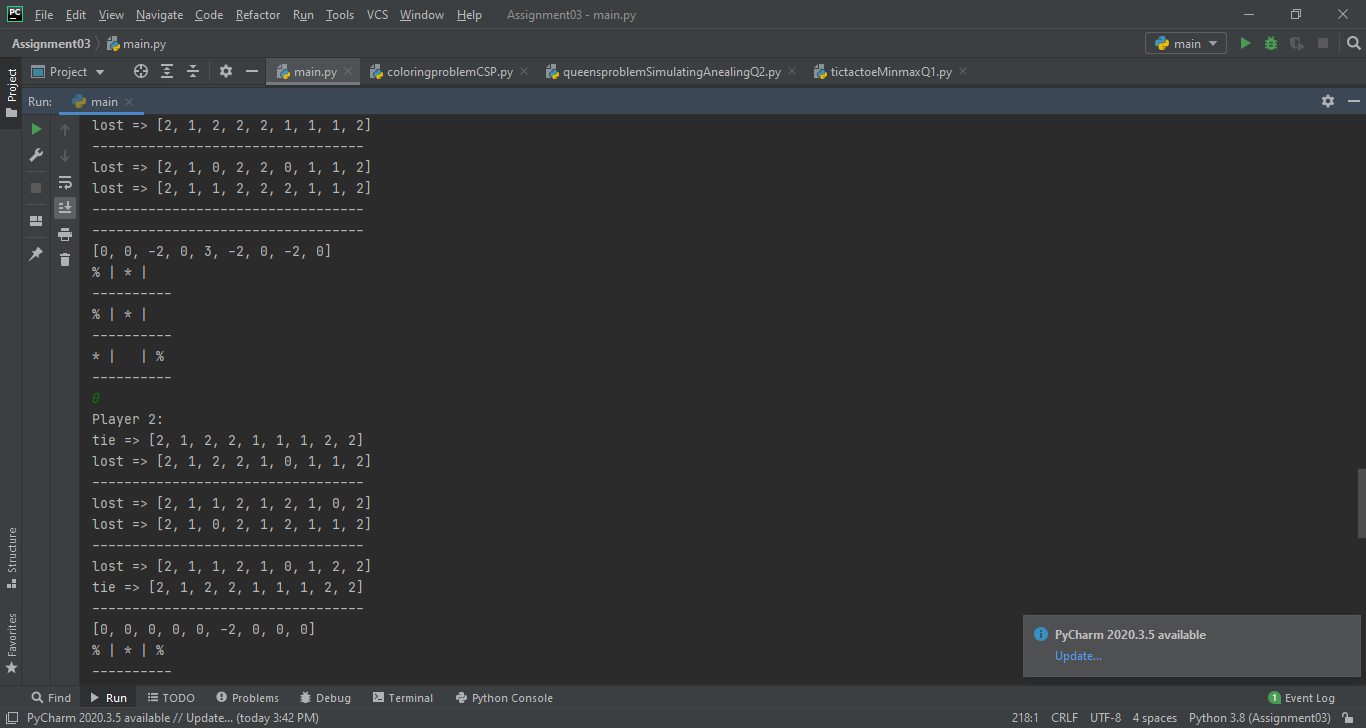
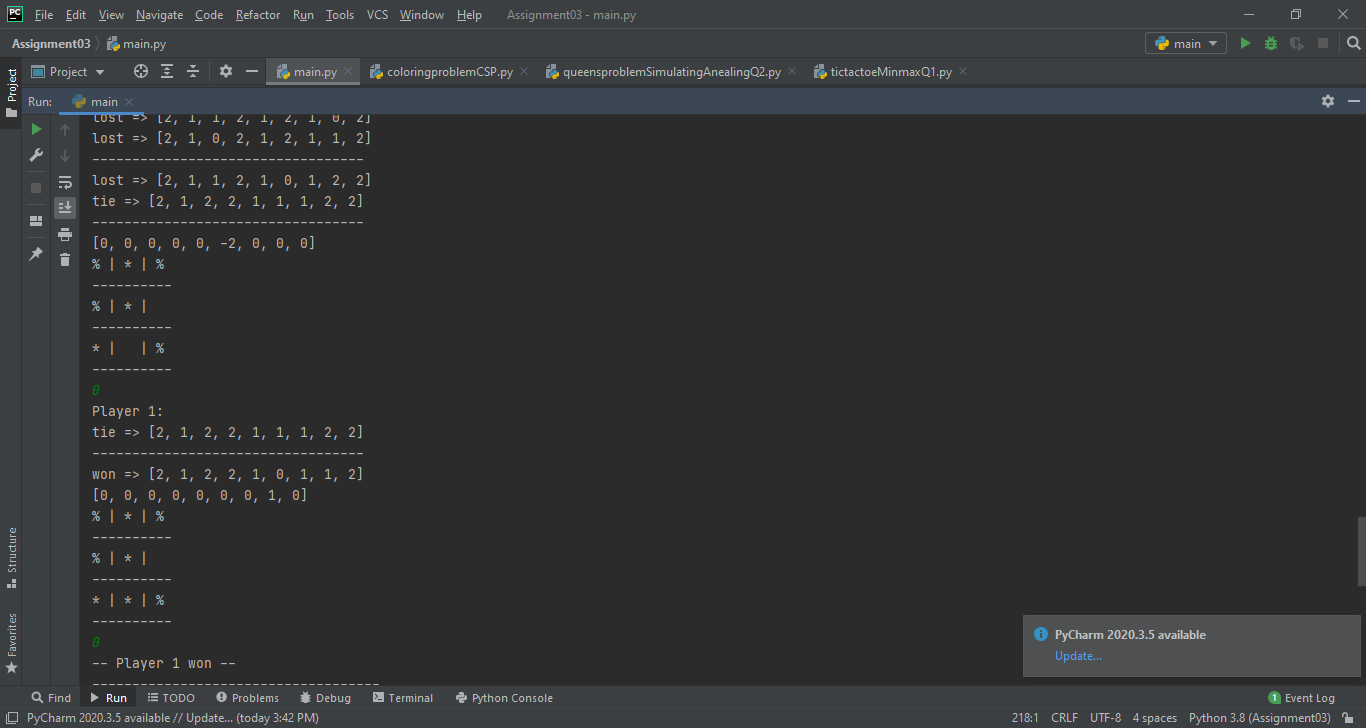
Student Name: Abdul Salam Wasti & Taha Raja

Roll No: 18F-0326 & 18F-0137 Section: CS(6B)

Date performed: 17/05/2021

# Question no 01:

Screenshot:

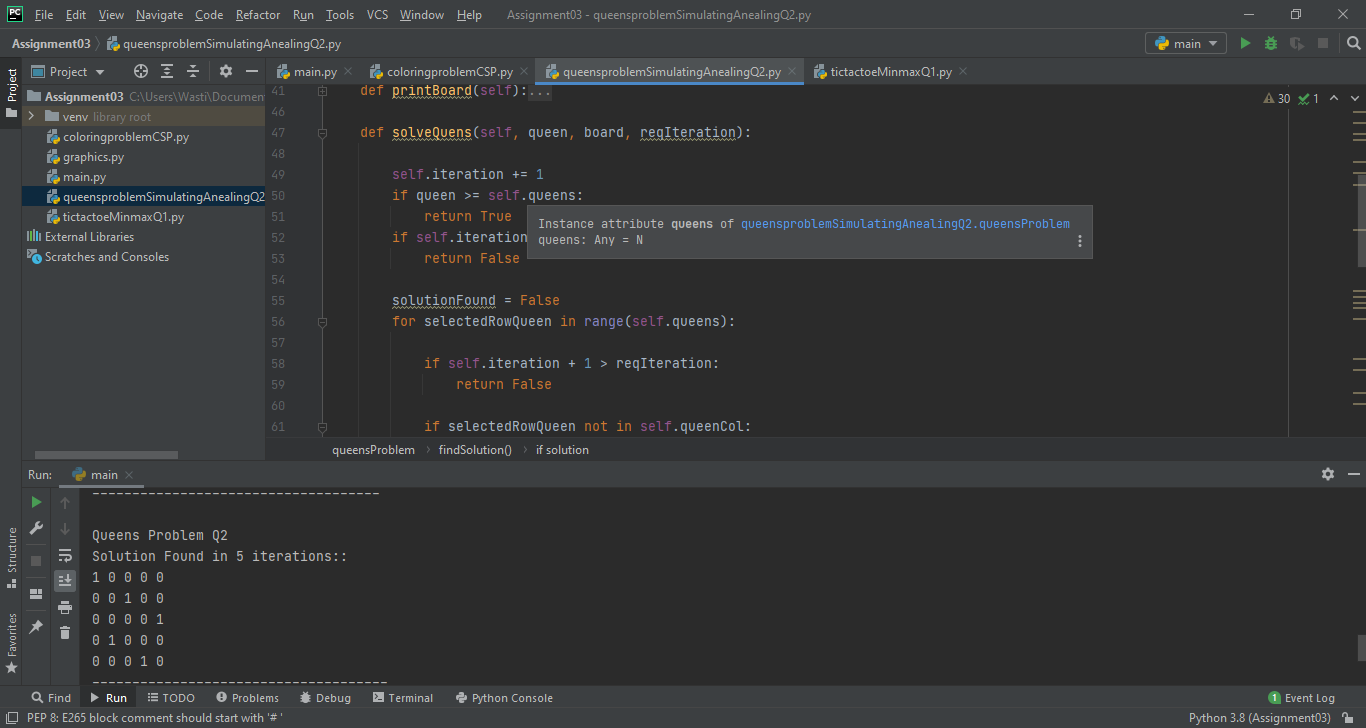
   

Code:

import copy  
import random  
from graphics import \*  
  
  
#using 1 for player 1 & and 2 for player 2  
  
def playerWin(board):  
 winMask = [[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6]]  
 result = 0 #tie  
 for i in winMask:  
 if board[i[0]] == board[i[1]] and board[i[1]] == board[i[2]] and board[i[0]] != 0:  
 result = board[i[0]]  
  
 if result == 0 and 0 not in board:  
 result = 3  
  
 return result  
  
def minmax(board, depth, alpha, beta, maximingPlayer, player):  
 result = playerWin(board)  
  
 if result != 0:  
  
 if result == 3:  
 print("tie => ", end="")  
 print(board)  
 result = 0  
 elif result == player:  
 print("won => ", end="")  
 print(board)  
 result = 1 \* depth  
 else:  
 print("lost => ", end="")  
 print(board)  
 result = -1 \* depth  
 return result  
  
 if maximingPlayer:  
 p = 1 if player == 2 else 2  
 chilList = []  
 for i in range(len(board)):  
 if board[i] == 0:  
 child = copy.deepcopy(board)  
 child[i] = p  
 chilList.append(child)  
  
 #print(chilList)  
 maxEval = -999  
 for children in chilList:  
 eval = minmax(children, depth+1, alpha, beta, False, player)  
 maxEval = max(maxEval,eval)  
 alpha = max(alpha, eval)  
 if beta <= alpha:  
 break  
 print("----------------------------------")  
 return maxEval  
  
 else:  
 p = 1 if player == 2 else 2  
 chilList = []  
 for i in range(len(board)):  
 if board[i] == 0:  
 child = copy.deepcopy(board)  
 child[i] = player  
 chilList.append(child)  
  
 minEval = 999  
 for children in chilList:  
 eval = minmax(children, depth+1 , alpha, beta, True, player)  
 minEval = min(minEval,eval)  
 beta = min(beta, eval)  
 if beta <= alpha:  
 break  
 return minEval  
  
class player:  
  
 def \_\_init\_\_(self, id):  
 self.playerID = id  
  
 def chooseMove(self,board):  
 moves = []  
  
 for i in range(len(board)):  
 if board[i] == 0:  
 newBoard = copy.deepcopy(board)  
 newBoard[i] = self.playerID  
 x = minmax(newBoard,1, -999, 999, True, self.playerID)  
 moves.append(x)  
 else:  
 moves.append(0)  
 #print("--------------------------------------------------------------")  
  
 sameMoves = []  
 win = 999  
 winIndex = -1  
 tieIndex = -1  
 counter = 0  
 for x in moves:  
 if win == x:  
 sameMoves.append(counter)  
 if x > 0 and x < win:  
 win = x  
 winIndex = counter  
 sameMoves.clear()  
 sameMoves.append(winIndex)  
 if board[counter] == 0 and x == 0 and tieIndex == -1:  
 tieIndex = counter  
 counter +=1  
  
 #print("----------------------------------------------------------")  
 print(moves)  
  
  
 if winIndex != -1:  
 winIndex = random.choice(sameMoves)  
 return winIndex  
 return tieIndex  
  
  
class ticTacToe:  
  
 board = [2, 1, 0, 2, 0, 0, 1, 0, 0]  
 #board = [0, 0, 0, 0, 0, 0, 0, 0, 0]  
  
 def printBoard(self):  
 counter = 0  
 for x in self.board:  
 if x == 0:  
 x = " "  
 elif x == 1:  
 x = "\*"  
 elif x == 2:  
 x = "%"  
  
 print(x, end="")  
 counter += 1  
 if (counter%3 == 0):  
 print("\n----------")  
 else:  
 print(" | ", end="")  
  
 def playGame(self, player1,player2):  
 playerTurn = 2  
  
 result = playerWin(self.board)  
 while result == 0:  
 newBoard = copy.deepcopy(self.board)  
 if playerTurn == 1:  
 print("Player "+ str(playerTurn) +":")  
 blockChoosed = player1.chooseMove(newBoard)  
 self.board[blockChoosed] = 1  
 playerTurn = 2  
 else:  
 print("Player " + str(playerTurn) + ":")  
 blockChoosed = player2.chooseMove(newBoard)  
 self.board[blockChoosed] = 2  
 playerTurn = 1  
 result = playerWin(self.board)  
 self.printBoard()  
 f = input()  
 #print("----------------------------------------")  
 if result == 3:  
 print("Tie - No one wins")  
 else:  
 if playerTurn == 1:  
 print("-- Player 2 won --")  
 else:  
 print("-- Player 1 won --")

# Question no 02:

Screenshot:

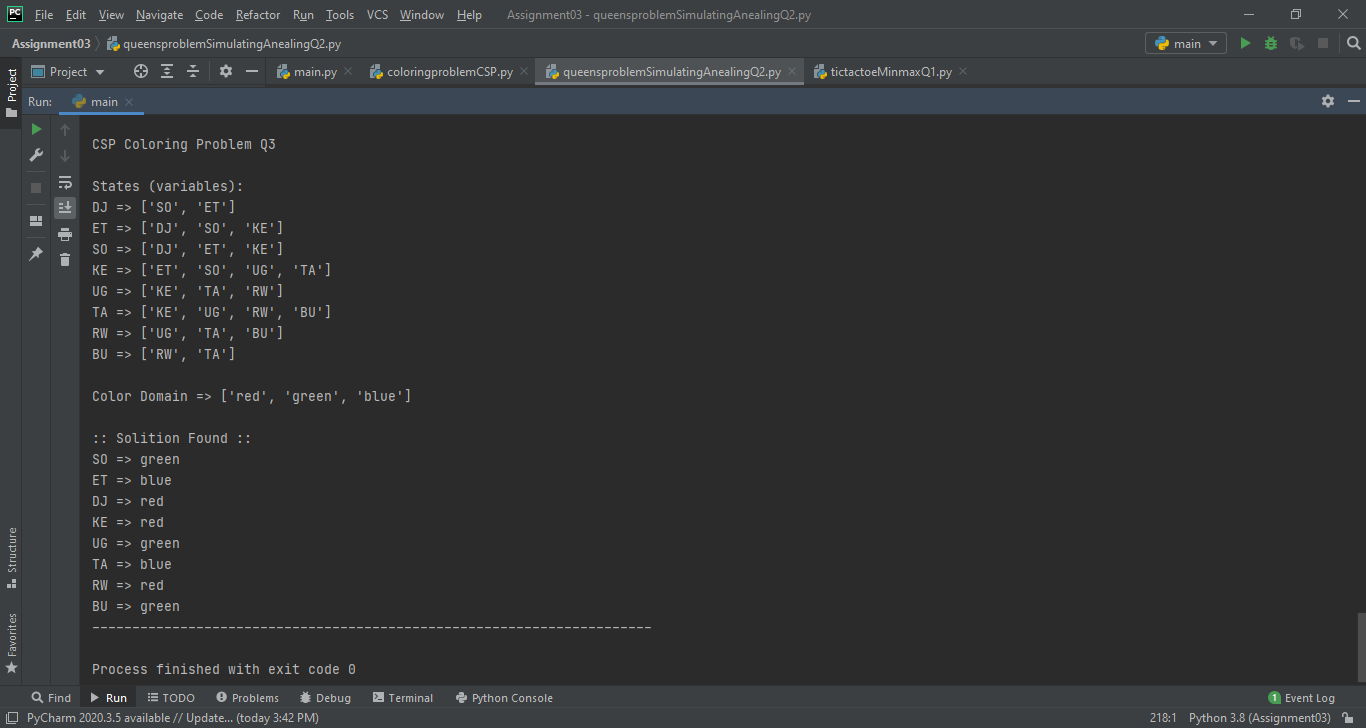


Code:

class queensProblem:  
  
 def \_\_init\_\_(self, N):  
 self.queens = N  
 self.board = [[0 for y in range(N)] for x in range(N)]  
 self.queenCol = [-1 for y in range(N)]  
 self.iteration = 0  
  
  
 def conditionCheck(self, queenCol, queenRow):  
 noConflict = True  
  
 for x in range(self.queens):  
 if x+queenCol < self.queens:  
 col = x + queenCol  
 if x+queenRow < self.queens:  
 row = x+queenRow  
 if self.board[row][col] != 0 and (row != queenRow and col != queenCol):  
 noConflict = False  
 return noConflict  
 if queenRow-x >= 0:  
 row = queenRow-x  
 if self.board[row][col] != 0 and (row != queenRow and col != queenCol):  
 noConflict = False  
 return noConflict  
  
 if queenCol-x >= 0:  
 col = queenCol - x  
 if x+queenRow < self.queens:  
 row = x+queenRow  
 if self.board[row][col] != 0 and (row != queenRow and col != queenCol):  
 noConflict = False  
 return noConflict  
 if queenRow-x >= 0:  
 row = queenRow-x  
 if self.board[row][col] != 0 and (row != queenRow and col != queenCol):  
 noConflict = False  
 return noConflict  
 return noConflict  
  
 def printBoard(self):  
 for x in range(self.queens):  
 for y in range(self.queens):  
 print(self.board[x][y], end=" ")  
 print(" ")  
  
 def solveQuens(self, queen, board, reqIteration):  
  
 self.iteration += 1  
 if queen >= self.queens:  
 return True  
 if self.iteration+1 > reqIteration:  
 return False  
  
 solutionFound = False  
 for selectedRowQueen in range(self.queens):  
  
 if self.iteration + 1 > reqIteration:  
 return False  
  
 if selectedRowQueen not in self.queenCol:  
 self.queenCol[queen] = selectedRowQueen  
 self.board[queen][selectedRowQueen] = 1  
  
  
 solutionFound = self.conditionCheck(selectedRowQueen,queen)  
 #self.printBoard()  
 #j = input()  
 #print("----------------------------------------------")  
 if solutionFound:  
 solutionFound = self.solveQuens(queen+1, board, reqIteration)  
 if not solutionFound:  
 self.queenCol[queen] = -1  
 self.board[queen][selectedRowQueen] = 0  
  
 return solutionFound  
  
 def findSolution(self, totalIteration):  
  
 solution = self.solveQuens(0,self.board, totalIteration)  
 if solution:  
 print("Solution Found in " + str(self.iteration-1) + " iterations::")  
 self.printBoard()  
 #print("--------------------------------")  
 else:  
 print("Solution Not-Found in " + str(self.iteration) + " iterations::")

# Question no 03:

Screenshot:



Code:

from collections import defaultdict  
  
class colorProblem:  
  
 def \_\_init\_\_(self):  
 self.stateSpace = defaultdict(list)  
 self.colorDomain = []  
 self.assignedColor = defaultdict(list)  
 self.setValues()  
 def addEdge(self, u, v):  
 self.stateSpace[u].append(v)  
 def printStateSpace(self):  
 for x in self.stateSpace:  
 print(x,end=" => ")  
 print(self.stateSpace[x])  
 def setValues(self):  
  
 self.addEdge("DJ", "SO")  
 self.addEdge("DJ", "ET")  
 self.addEdge("ET", "DJ")  
 self.addEdge("ET", "SO")  
 self.addEdge("ET", "KE")  
 self.addEdge("SO", "DJ")  
 self.addEdge("SO", "ET")  
 self.addEdge("SO", "KE")  
 self.addEdge("KE", "ET")  
 self.addEdge("KE", "SO")  
 self.addEdge("KE", "UG")  
 self.addEdge("KE", "TA")  
 self.addEdge("UG", "KE")  
 self.addEdge("UG", "TA")  
 self.addEdge("UG", "RW")  
 self.addEdge("TA", "KE")  
 self.addEdge("TA", "UG")  
 self.addEdge("TA", "RW")  
 self.addEdge("TA", "BU")  
 self.addEdge("RW", "UG")  
 self.addEdge("RW", "TA")  
 self.addEdge("RW", "BU")  
 self.addEdge("BU", "RW")  
 self.addEdge("BU", "TA")  
  
 self.colorDomain.append("red")  
 self.colorDomain.append("green")  
 self.colorDomain.append("blue")  
  
 print("\nStates (variables):")  
 self.printStateSpace()  
  
 print("\nColor Domain", end=" => ")  
 print(self.colorDomain)  
  
  
  
 def checkCondition(self, v, color):  
 if self.assignedColor[v] == color:  
 return False  
 return True  
  
 def findSolution(self, startingNode):  
  
 observable = []  
 assigned = []  
 visited = []  
  
 observable.append(startingNode)  
  
 constrain = False  
 while observable:  
  
 node = observable.pop(0)  
 visited.append(node)  
 neighboursNodes = self.stateSpace[node]  
  
 constrain = False  
 for color in self.colorDomain:  
  
 for neighbour in neighboursNodes:  
 constrain = self.checkCondition(neighbour, color)  
 if not constrain:  
 break  
  
 if constrain:  
 self.assignedColor[node] = color  
 break  
  
 for neighbour in neighboursNodes:  
 if neighbour not in visited and neighbour not in observable:  
 observable.append(neighbour)  
  
  
 if not constrain:  
 print(":: Solition not possible ::")  
 else:  
 print("\n:: Solition Found ::")  
 for x in self.assignedColor:  
 print(x, end=" => ")  
 print(self.assignedColor[x])  
 print("----------------------------------------------------------------------")