Lab 5

8-Queen Problem

# Code with comments

# -\*- coding: utf-8 -\*-

"""

Created on Fri Aug 26 10:28:13 2022

@author: aelbadra

"""

# This import statement imports EightQueenGameSolver from eight\_queen

from eight\_queen import EightQueenGameSolver

EightQueenGameSolver.brute\_force\_solution()

#### Using a Brute-Force Approach to finding a solution to the 8-Queen problem. Notice the amount of time taken.

# This statement calls the brute\_force\_solution\_2 method on EightQueenGameSolver

# EightQueenGameSolver.brute\_force\_solution\_2()

#### Using a hill climbing approach to find a solution from an initial state very close to a goal state.

# This statement calls the hill\_climbing method on EightQueenGameSolver

# EightQueenGameSolver.hill\_climbing([0, 4, 7, 5, 2, 6, 1, 3])

#### Using a hill climbing approach when the initial state is that all eight queens are on the first row.

# This statement calls the hill\_climbing method on EightQueenGameSolver with 0s for starting values

#EightQueenGameSolver.hill\_climbing([0, 0,0,0,0,0,0,0])

#### Using a repeated hill-climbing approach.

#### Whenever hill climbing gets stuck at a local minimum, a new initial state is generated and hill climbing

#### is recalled until a solution is found.

# This statement calls the restarting\_hill\_climbing method on EightQueenGameSolver with 0s for starting values

#EightQueenGameSolver.restarting\_hill\_climbing([0,0,0,0,0,0,0,0])

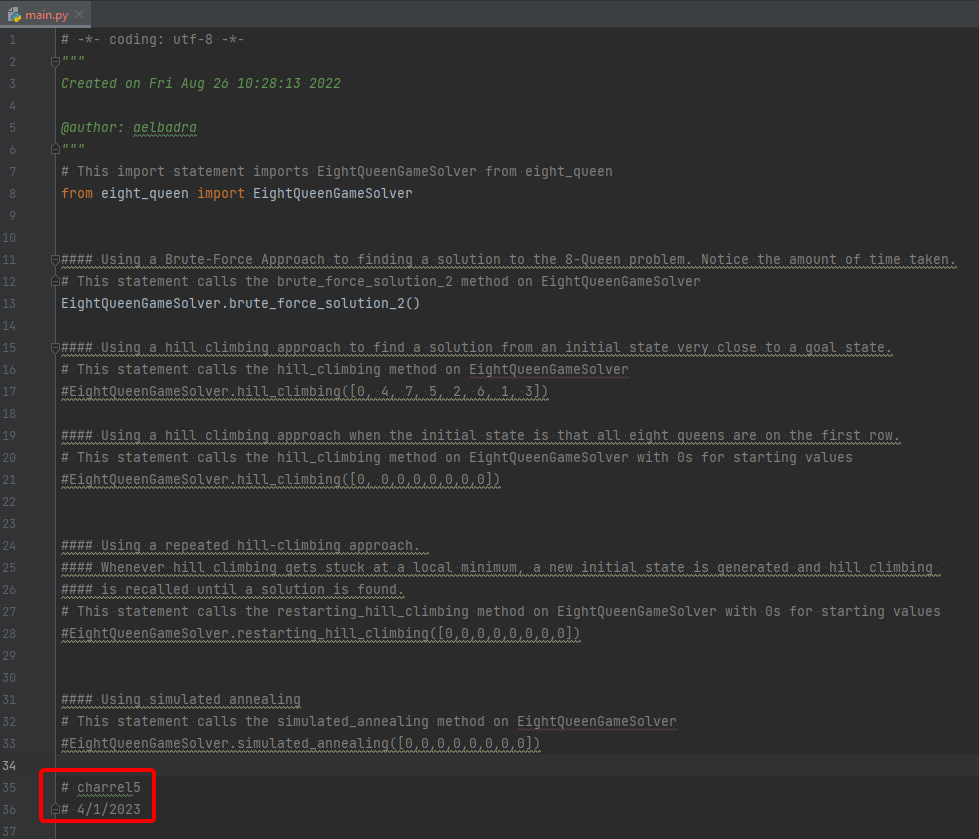
#### Using simulated annealing

# This statement calls the simulated\_annealing method on EightQueenGameSolver

#EightQueenGameSolver.simulated\_annealing([0,0,0,0,0,0,0,0])

# charrel5

# 4/1/2023



# -\*- coding: utf-8 -\*-

"""

Created on Fri Aug 26 10:06:03 2022

@author: aelbadra

"""

#QUEENS have indices 0-7

#QUEEN POSITIONS are indicated as (x,y) coordinates in the range [0,7]

# This statement imports the random module

import random

# This class defines the EightQueenGame which is solved by the EightQueenGameSolver

class EightQueenGame:

# This method sets positions to be p, which is an argument to the method

def \_\_init\_\_(self, p):

self.positions = p

return

#find how many attacks are in a given setting

#p is a list of size 8 holding the 8 queens positions from row 0 to row 7

# This method uses nested loops to get the total number of attacks

def get\_num\_attacks(p):

nattacks = 0

for i in range(0,8):

for j in range(i+1,8):

i\_x = i

j\_x = j

i\_y = p[i]

j\_y = p[j]

if (i\_x == j\_x) or (i\_y == j\_y) or ( abs(i\_x-j\_x) == abs(i\_y-j\_y) ):

nattacks += 1

return nattacks

#set the position of one queen

# This method sets the position p for queen within the position dict

def set\_position(self, queen, p):

self.positions[queen] = p

return

#set the positions of all queens

# This method sets the positions dict to be p

def set\_positions(self, p):

self.positions = p

return

#for a given position, find all neighboring positions that can be

#obtained by moving one queen to any location along its column.

# This method accepts the argument p and uses it to find all neighboring positions that the queen can be moved to

def extend\_position(p):

extended\_positions=[]

for col in range(8):

for row in range(8):

if row != p[col]:

new\_p = p[:]

new\_p[col]=row

extended\_positions.append(new\_p)

return extended\_positions

#Class that contains different methods for solving an 8-queen problem:

# - Brute Force Method

# - Hill Climbing

# - Repeating Hill Climbing

# - Simulated Annealing

# This class defines the EightQueenGameSolver which solves the EightQueenGame

class EightQueenGameSolver:

# This method does nothing to initialize the class and can be removed

def \_\_init\_\_(self):

return

#brute-force method tries out all positions until a solution is found

#eight nested for loops are used to loop over the positions of the 8 queens

# This method incorrectly lacks the self argument and uses nested loops to try all possible positions for a solution

def brute\_force\_solution():

positions = [0,0,0,0,0,0,0,0]

for i in range(0,8):

for j in range(0,8):

for k in range(0,8):

for l in range(0,8):

for m in range(0,8):

for n in range(0,8):

for o in range(0,8):

for p in range(0,8):

positions = [i,j,k,l,m,n,o,p]

a = EightQueenGame.get\_num\_attacks(positions)

print(a, ":", positions)

if a == 0:

print("Solution Found:", positions)

return

#instead of writing 8 nested for loops for brute force solution,

#we can use the itertools package to write more compact code for nested

#for loops.

# This method incorrectly lacks the self argument and uses generators to try all possible positions for a solution

def brute\_force\_solution\_2():

import itertools

for i,j,k,l,m,n,o,p in itertools.product(range(8), range(8), range(8), range(8), range(8), range(8), range(8), range(8)):

positions = [i,j,k,l,m,n,o,p]

a = EightQueenGame.get\_num\_attacks(positions)

print(a, ":", positions)

if a == 0:

print("Solution Found:", positions)

break

#hill climbing method

# This method uses an indefinite loop, test all neighboring positions, check if they are solutions,

# and update the state. It returns true if a solution is found, and returns false if in a local minimum, or stuck

def hill\_climbing(initial\_position):

#check if initial position is a solution!

old\_p = initial\_position[:]

old\_n\_attacks = EightQueenGame.get\_num\_attacks(initial\_position)

if old\_n\_attacks == 0:

print("Initial State is a solution!", initial\_position)

return True

while True:

#extend current position by checking all neighboring positions

new\_positions = EightQueenGame.extend\_position(old\_p)

new\_n\_attacks = []

#get number of attacks for each neighboring position

for i in range(len(new\_positions)):

n = EightQueenGame.get\_num\_attacks(new\_positions[i])

new\_n\_attacks.append(n)

#get index of the neighboring position min # attacks

min\_attacks = min(new\_n\_attacks)

new\_p\_indx = new\_n\_attacks.index(min\_attacks)

#check if that is a solution

if min\_attacks == 0:

print("Solution found:", new\_positions[new\_p\_indx])

return True

#if no neighboring position has a lower number of attacks, hill climbing gets stuck!

if old\_n\_attacks<=min\_attacks:

print("Hill Climbing Reached a Local Minimum at Position:",

new\_positions[new\_p\_indx])

return False

#update current position in case new position has less number of attacks

if old\_n\_attacks>min\_attacks:

old\_p = (new\_positions[new\_p\_indx])[:]

old\_n\_attacks = EightQueenGame.get\_num\_attacks(old\_p)

print(old\_n\_attacks, old\_p)

#same like hill climbing, except that each time hill climbing gets stuck,

#it is restarted with a different initial state until a solution is found.

# This method uses the above defined hill\_climbing method to try to find a solution while using a new starting postition when stuck

def restarting\_hill\_climbing(initial\_position):

count=0

found = EightQueenGameSolver.hill\_climbing(initial\_position)

if found:

return

while not found:

count +=1

#generate a new initial state

new\_p = [random.randrange(8) for i in range(8)]

#re-call the hill climibing method

found = EightQueenGameSolver.hill\_climbing(new\_p)

print("Number of Hill Climbing restarts = ", count)

return

#simulated annealing is very similar to hill climbing. Only difference is that when we

#cannot find any neighboring positions with a lower number of attacks than the current

#position, then we can

def simulated\_annealing(initial\_position):

# check if initial position is a solution!

old\_p = initial\_position[:]

old\_n\_attacks = EightQueenGame.get\_num\_attacks(initial\_position)

if old\_n\_attacks == 0:

print("Initial State is a solution!", initial\_position)

return True

T = 0.70

while True:

# extend current position by checking all neighboring positions

new\_positions = EightQueenGame.extend\_position(old\_p)

new\_n\_attacks = []

# get number of attacks for each neighboring position

for i in range(len(new\_positions)):

n = EightQueenGame.get\_num\_attacks(new\_positions[i])

new\_n\_attacks.append(n)

# get index of # attacks for some random neighbor

new\_p\_indx = random.randrange(len(new\_n\_attacks))

n\_attacks = new\_n\_attacks[new\_p\_indx]

# check if that is a solution

if n\_attacks == 0:

print("Solution found:", new\_positions[new\_p\_indx])

return True

# only accept worse positions with some probability T, then reduce T

if old\_n\_attacks <= n\_attacks:

# roll a die

d = random.random()

if d < T:

old\_p = (new\_positions[new\_p\_indx])[:]

old\_n\_attacks = EightQueenGame.get\_num\_attacks(old\_p)

if T > 0.1:

T = T - 0.01

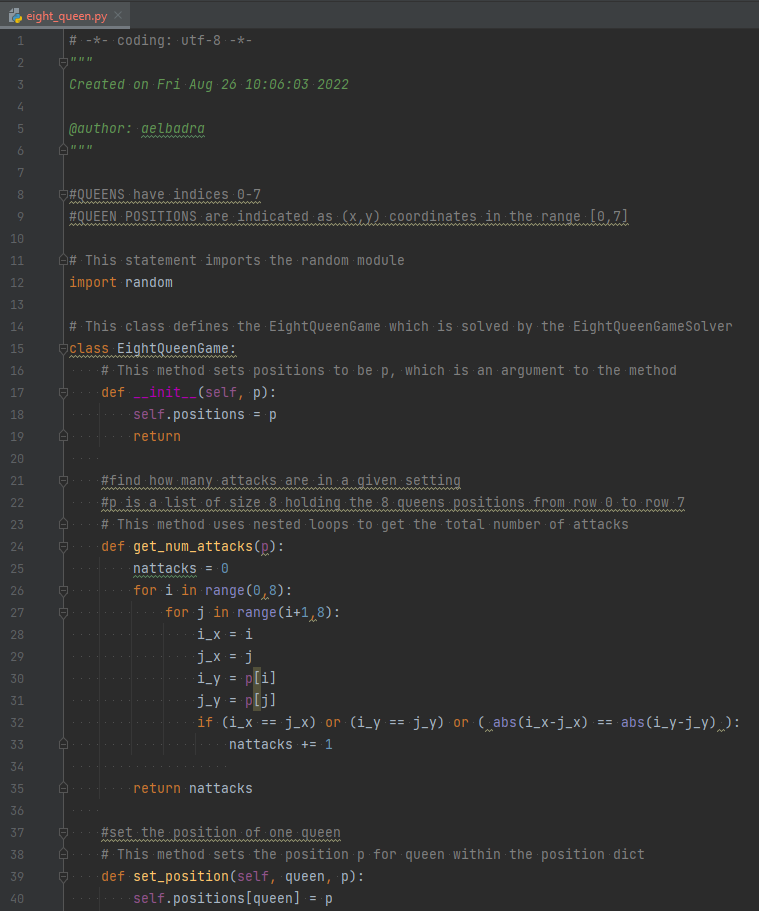
# update current position in case new position has less number of attacks

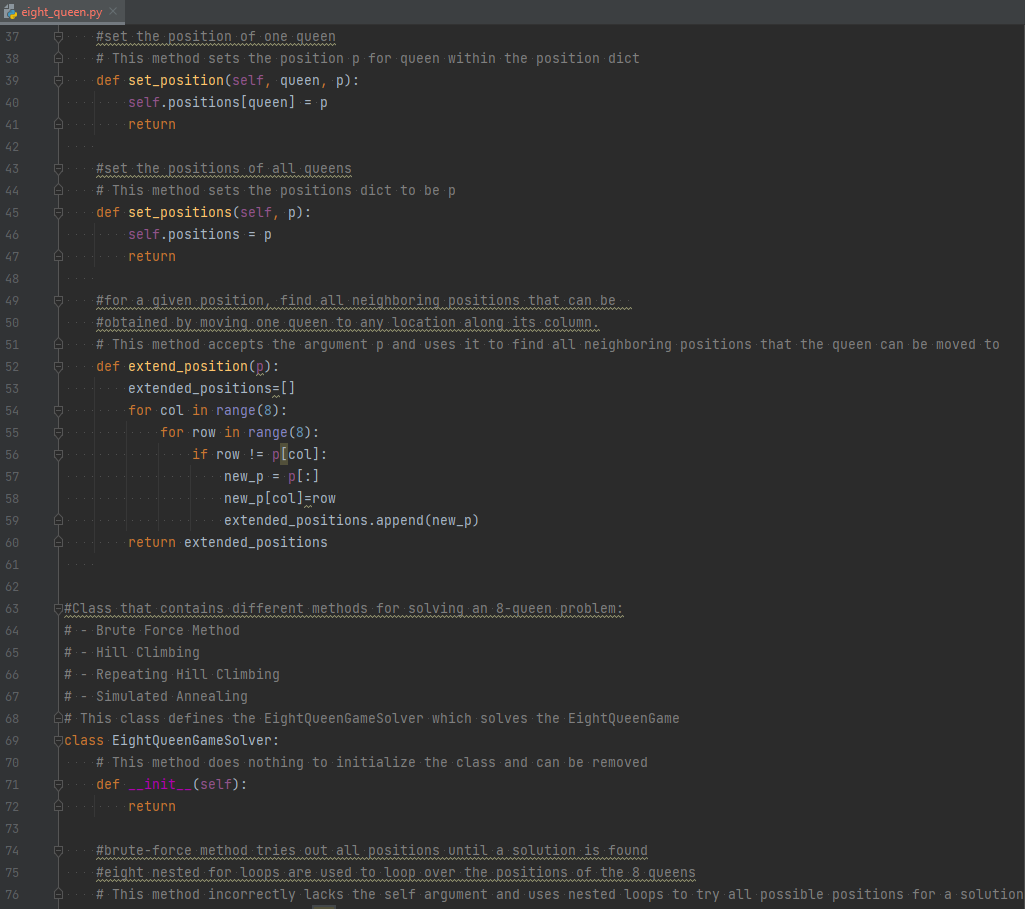
if old\_n\_attacks > n\_attacks:

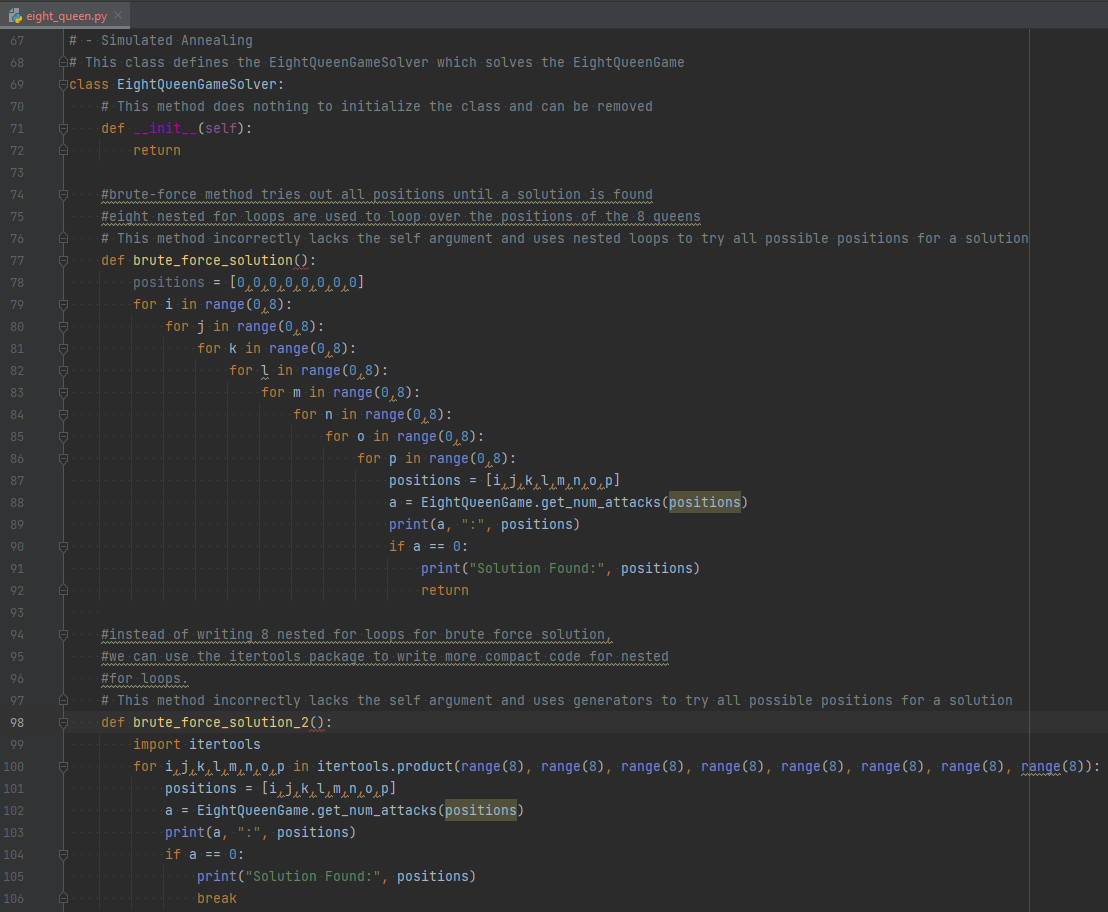
old\_p = (new\_positions[new\_p\_indx])[:]

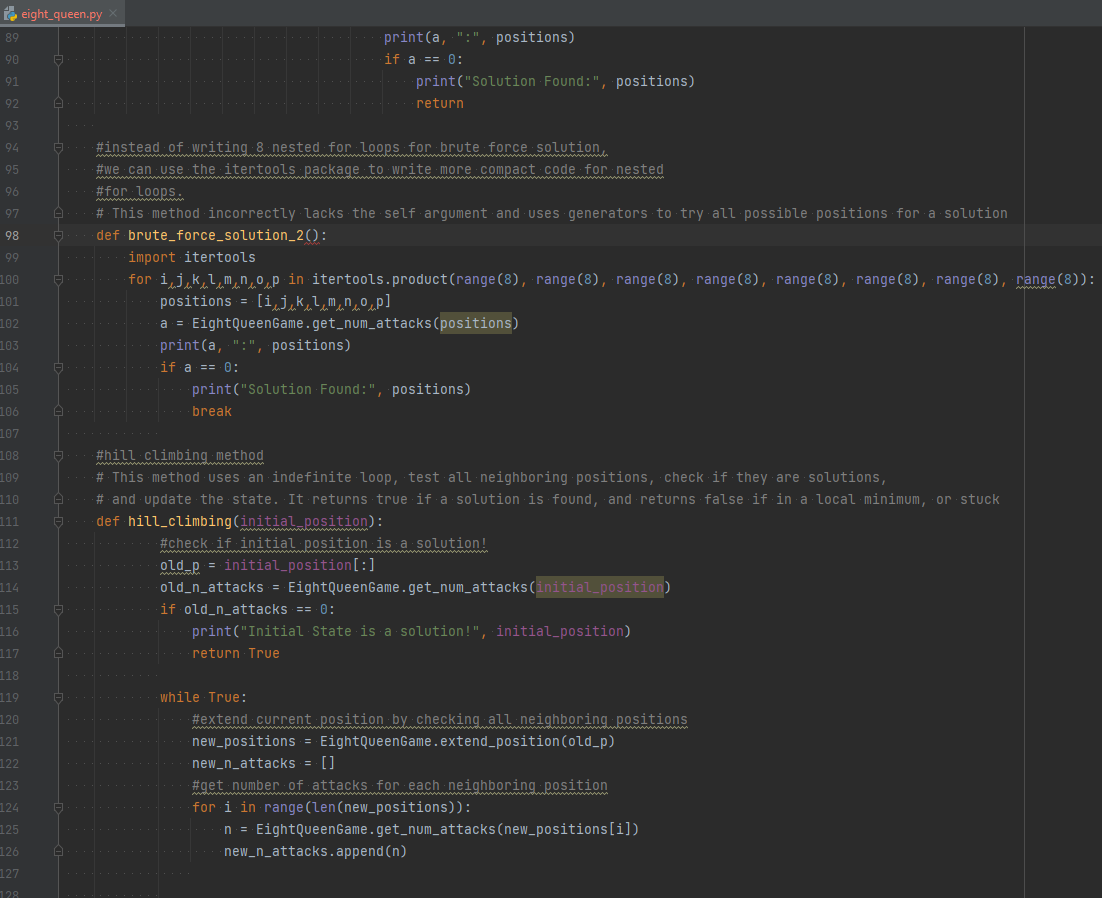
old\_n\_attacks = EightQueenGame.get\_num\_attacks(old\_p)

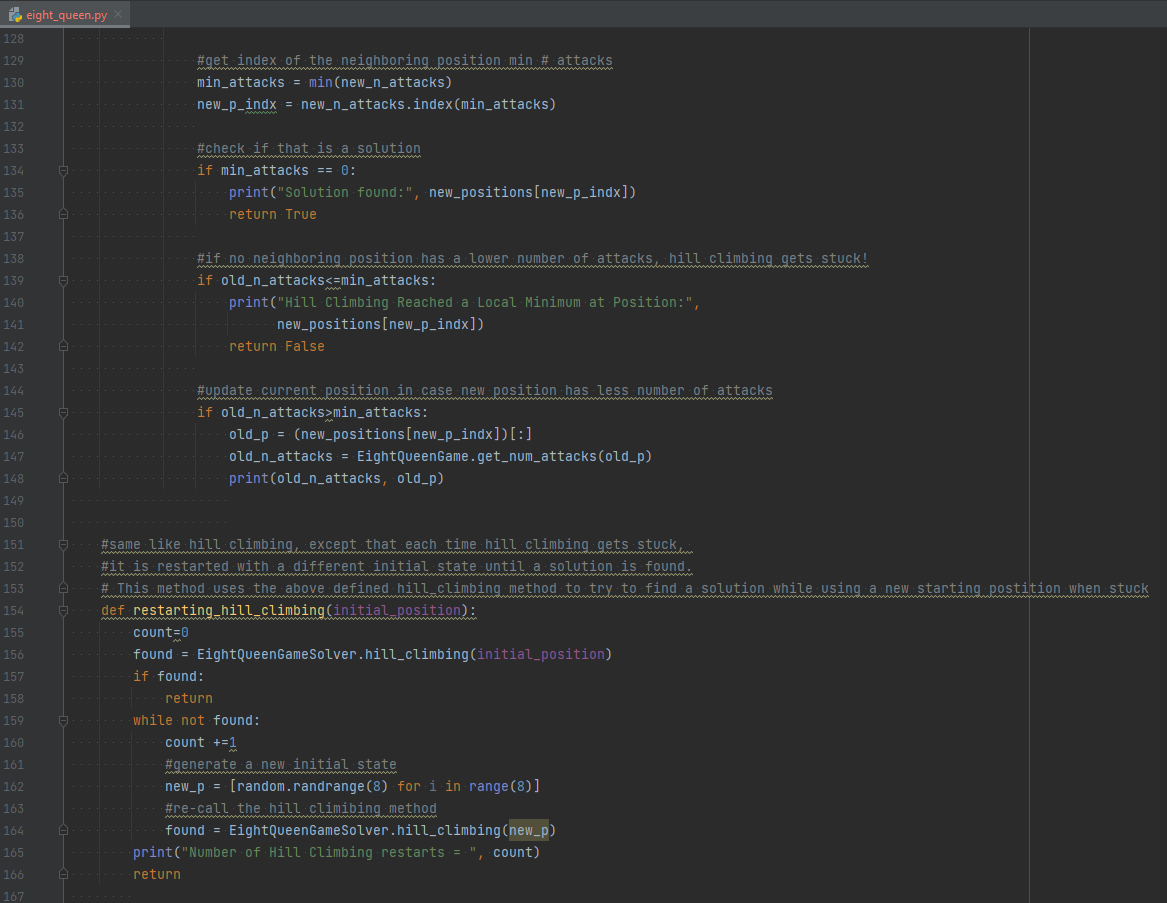
print(old\_n\_attacks, old\_p)

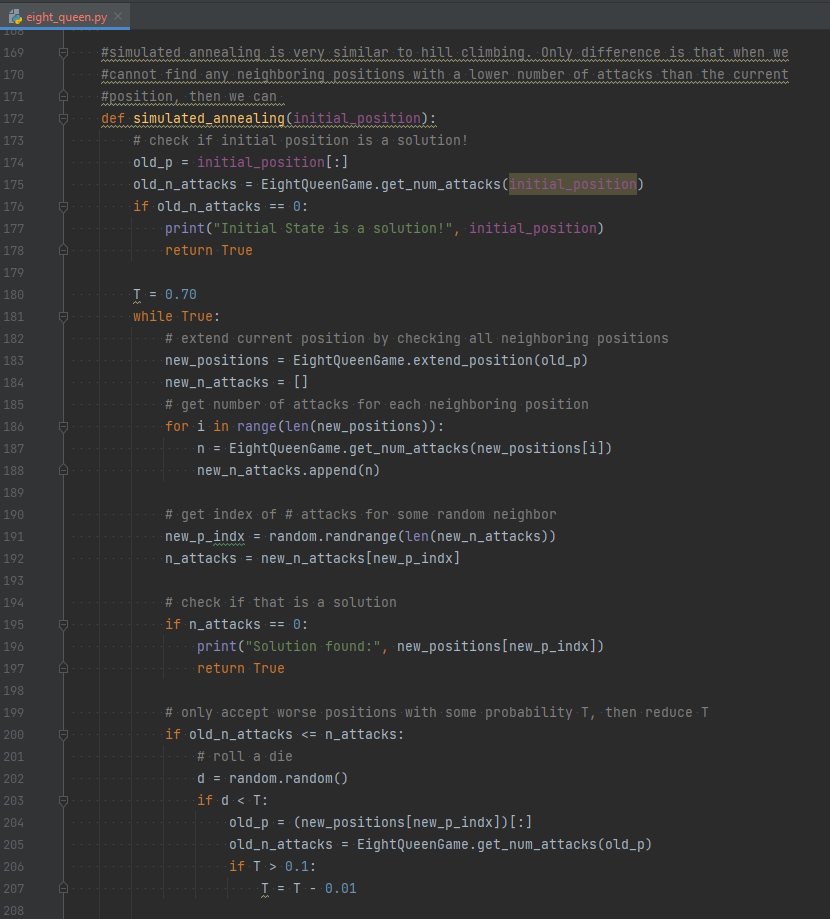


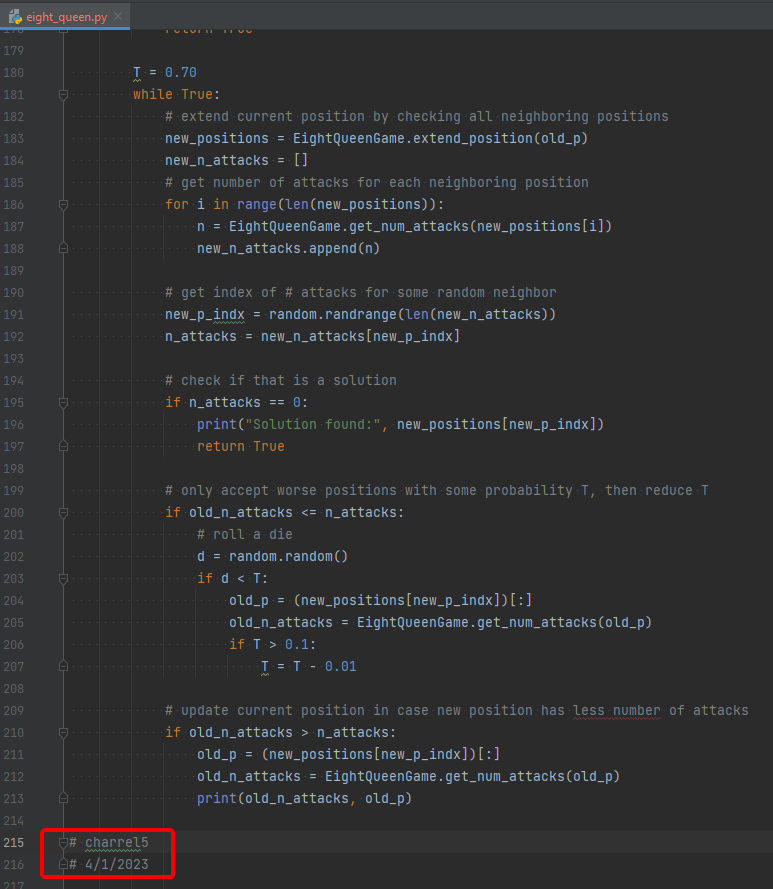




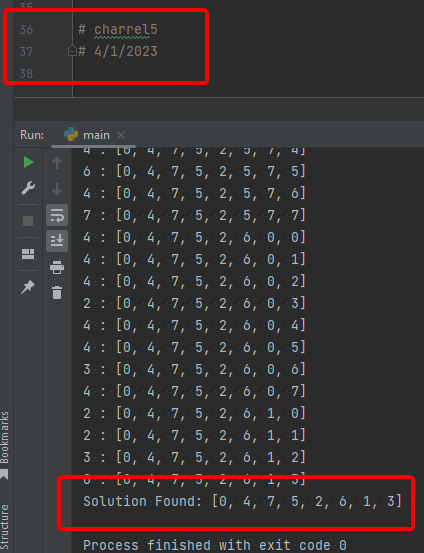
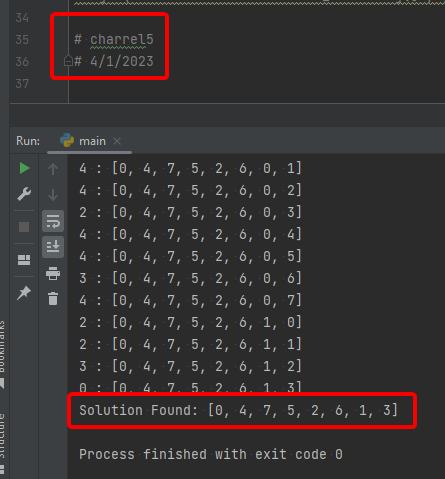
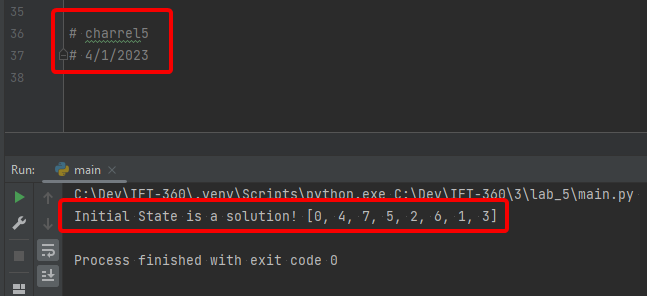
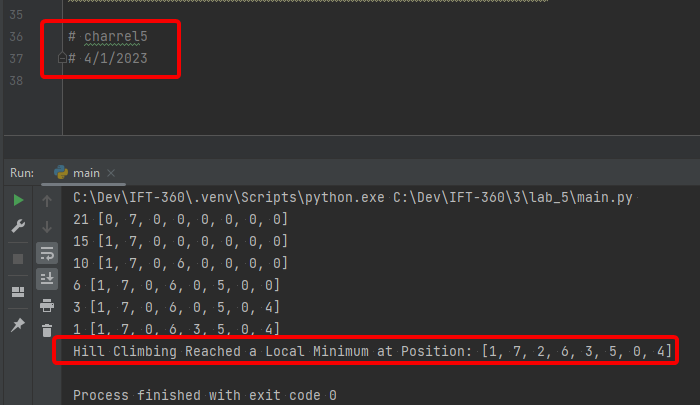
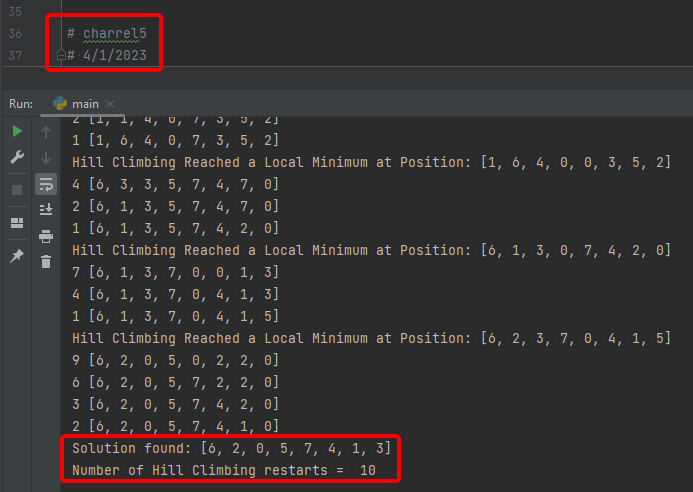
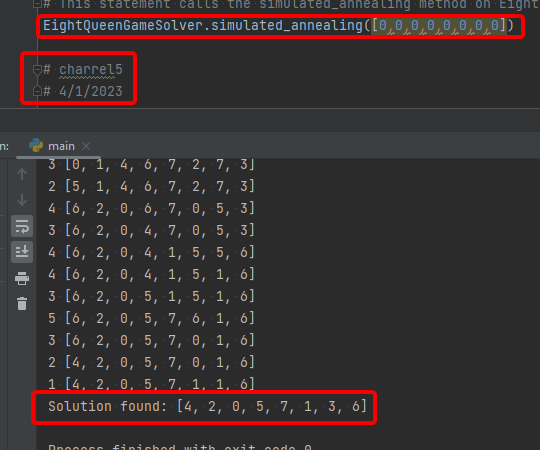
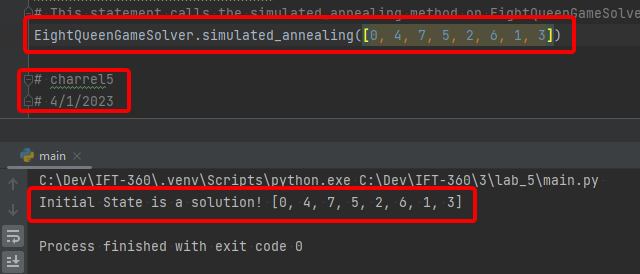








# Questions

1. Uncomment the call to the brute-force method & run it. Notice how this method takes a very long time to find a solution. (It should take a few minutes)
   1. 
2. Comment the previous method and uncomment the second call to the brute-force method and run it. Notice how this one also takes a very long time to run. The only difference between it and the first method is in how the code is written in a more compact way.
   1. 
3. Comment the previous method and uncomment the first hill climbing call which starts from this state: [0, 4, 7, 5, 2, 6, 1, 3]. What is the obtained result?
   1. Initial State is a solution! [0, 4, 7, 5, 2, 6, 1, 3]
   2. 
4. Comment the first hill climbing call and uncomment the second hill climbing call that starts from this state: [0, 0, 0, 0, 0, 0, 0, 0]. What is the obtained result?
   1. Hill Climbing Reached a Local Minimum at Position: [1, 7, 2, 6, 3, 5, 0, 4]
   2. 
5. Comment the previous call and uncomment the call to the Restarting Hill Climbing method that starts from this state: [0, 0, 0, 0, 0, 0, 0, 0]. Does this method get stuck? Try running it two more times. What is the number of hill climbing restarts for each run?
   1. No, it does not get stuck
   2. Number of Hill Climbing restarts = 10
   3. Number of Hill Climbing restarts = 2
   4. Number of Hill Climbing restarts = 19
   5. 
6. Check if you face any errors. Try to fix them and rerun the code.
   1. No errors
   2. 
7. Try running the code starting at different initial states. Do you notice a change in time when you run the code multiple times?
   1. The second time was much faster
   2. 
8. How do you explain that?
   1. The initial state I tried was a solution so it finished right away