* **Problem Name: Solving 8-queen problem using Hill climbing and simulated annealing algorithm**

import numpy

import copy

import random

import pygame #For drawing

class HillClimbing:

def sameRow(self, row, col, a): #checking for same row attacking queen

count = 0

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

if (int(a[row][i]) == -1):

count += 1

return count

def upperDiagonal(self, row, col, a): #checking for upper diagonal attacking queen

count = 0

row -= 1

col += 1

while (row >= 0 and col <= 7):

if (int(a[row][col]) == -1):

count += 1

row -= 1

col += 1

return count

def lowerDiagonal(self, row, col, a): #checking for lower diagonal attacking queen

count = 0

row += 1

col += 1

while (row <= 7 and col <= 7):

if (int(a[row][col]) == -1):

count += 1

row += 1

col += 1

return count

def countCostForOneIndex(self, row, col, a): #cost count for each queen

count = 0

count += self.sameRow(row, col, a)

count += self.upperDiagonal(row, col, a)

count += self.lowerDiagonal(row, col, a)

return count

def hillClimbingStateSingleCell(self, a): #cost for one cell

count = 0

for i in range(8):

for j in range(8):

if int(a[j][i]) == -1:

count += self.countCostForOneIndex(j, i, a)

return(count)

def replace(self, a, index, j): #replace queen from one cell to another

for row in range(8):

if a[row][index] == -1:

a[row][index] = 0

a[j][index] = -1

return a

def move(self,a,hue,cost): #move the queen

dummyA = copy.deepcopy(a)

for i in range(8):

for j in range(8):

if cost[j][i] == hue:

dummyA = self.replace(dummyA, i, j)

break

return dummyA

def cost(self, a, hue): #total cost

cost = numpy.zeros((8, 8))

for i in range(8):

dummyA = copy.deepcopy(a)

for j in range(8):

dummyA = self.replace(dummyA, i, j)

cost[j][i] = self.hillClimbingStateSingleCell(dummyA)

if cost[j][i] < hue:

hue = cost[j][i]

return cost, hue

def hillClimbingState(self, a, step): #decision making

solve = False

hue = self.hillClimbingStateSingleCell(a)

b = copy.deepcopy(a)

s = step

while True:

cost, value = self.cost(b, hue)

if int(value) >= hue:

break

elif int(self.hillClimbingStateSingleCell(b)) == 0:

solve = True

print(cost)

print(b)

break

hue = value

b = self.move(b,hue,cost)

print("Step ", s)

s += 1

print(cost)

print(b)

return solve, b, hue, s

def randomValue(self, a): #random value generation

numberOfColumn = random.randint(0,7)

for i in range(numberOfColumn):

column = random.randint(0,7)

row = random.randint(0, 7)

a = self.replace(a, row, column)

return a

def stimulation(self, a, hue): #Simulated annealing

for i in range(10000):

current = hue

rand = self.randomValue(a)

new = self.hillClimbingStateSingleCell(a)

difference = new - current

if difference > 0:

return rand

# Define some colors

BLUE = (50, 120, 150)

WHITE = (240, 255, 240)

RED = (255, 100, 100) # represent QUEEN

def draw\_the\_queen(queen\_matrix, no):

WIDTH = 61

HEIGHT = 61

MARGIN = 1

grid = []

for row in range(8):

grid.append([])

for column in range(8):

grid[row].append((row + column) % 2) # Append a cell

# print(queen\_matrix)

for column in range(len(queen\_matrix)):

current\_row = list(queen\_matrix[:, column]).index(-1)

grid[current\_row][column] = 2

pygame.init()

WINDOW\_SIZE = [500, 500]

screen = pygame.display.set\_mode(WINDOW\_SIZE)

pygame.display.set\_caption("Solution at step: " + str(no+1))

done = False

while not done:

for event in pygame.event.get(): # User did something

if event.type == pygame.QUIT: # If user clicked close

done = True # Flag that we are done so we exit this loop

# Set the screen background

screen.fill(BLUE)

# Draw the grid

for row in range(8):

for column in range(8):

color = WHITE

if grid[row][column] == 1:

color = BLUE

elif grid[row][column] == 2:

color = RED

pygame.draw.rect(screen,

color,

[(MARGIN + WIDTH) \* column + MARGIN,

(MARGIN + HEIGHT) \* row + MARGIN,

WIDTH,

HEIGHT])

# Go ahead and update the screen with what we've drawn.

pygame.display.flip()

pygame.quit()

return

a = numpy.zeros((8, 8))

a[4][0] = -1

a[5][1] = -1

a[6][2] = -1

a[3][3] = -1

a[4][4] = -1

a[5][5] = -1

a[6][6] = -1

a[5][7] = -1

draw\_the\_queen(a, -1)

h = HillClimbing()

solve , b, hue, step = h.hillClimbingState(a, 1)

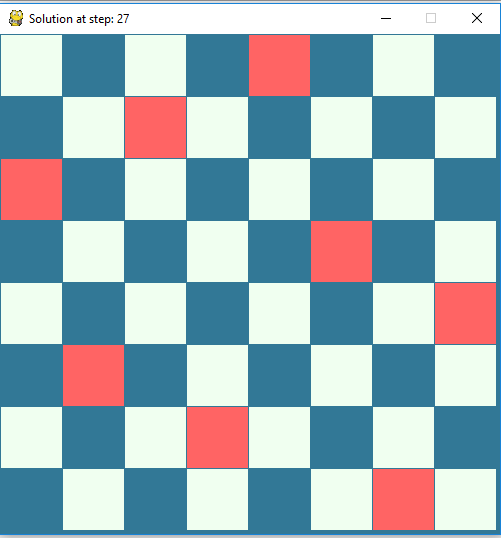
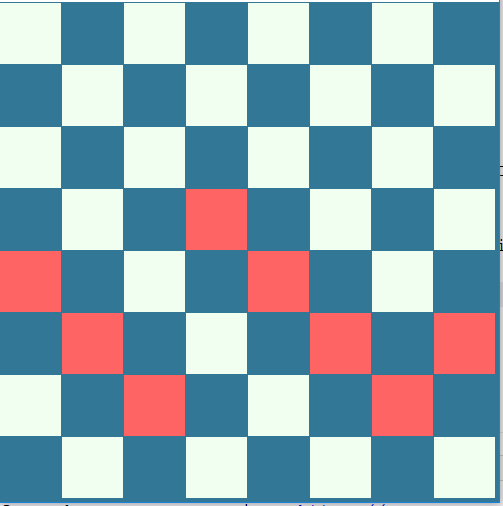
while solve == False:

b = h.stimulation(b, hue)

solve, b, hue, step = h.hillClimbingState(b, step)

if solve:

draw\_the\_queen(b, step)



Solution

Input