**Aim: Write a program to simulate N-Queen problem using Heuristic Search Technique and recursive backtracking method.**

**Code: Heuristic Approach :-**

import time

import queue

import random

import numpy as np

import matplotlib.pyplot as plt

from heapq import heappush, heappop, heapify

N = 8

q = queue.Queue()

class PriorityQueue:

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

self.pq = []

def add(self, item):

heappush(self.pq, item)

def poll(self):

return heappop(self.pq)

def peek(self):

return self.pq[0]

def remove(self, item):

value = self.pq.remove(item)

heapify(self.pq)

return value is not None

def \_\_len\_\_(self):

return len(self.pq)

class queen:

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

self.row = -1

self.col = -1

def \_\_cmp\_\_(self, other):

return self.row == other.row and self.col == other.col

def \_\_eq\_\_(self, other):

return self.\_\_cmp\_\_(other)

def \_\_hash\_\_(self):

return hash(str(self.list\_()))

def list\_(self):

return [self.row,self.col]

class state:

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

self.nQueen = [queen() for i in range(N)]#

index is col, value is row

if(data != None):

self.moves = data.moves + 1

self.heuristicVal = data.heuristicVal

for i in range(N):

self.nQueen[i].row =data.nQueen[i].row

self.nQueen[i].col =data.nQueen[i].col

else:

self.moves = 0

self.initQueens()

self.parent = data

def isSafe(self,row,col):

for i in range(N):

if(self.nQueen[i].row == row):

return False

for i in range(N):

if(self.nQueen[i].col == col):

return False

for i in range(N):

if(abs(self.nQueen[i].row - row) ==abs(self.nQueen[i].col - col)):

return False

return True

def getConflictCount(self,row,col):

count = 0

conflictCount = 0

ConflictSet = []

for i in range(N):

if(self.nQueen[i].row == row):

count+=1

ConflictSet.append(self.nQueen[i])

for i in range(N):

if(self.nQueen[i].col == col):

count+=1

ConflictSet.append(self.nQueen[i])

for i in range(N):

if(abs(self.nQueen[i].row - row) ==abs(self.nQueen[i].col - col)):

count+=1

ConflictSet.append(self.nQueen[i])

for obj in ConflictSet:

if(not(obj.row == row and obj.col ==col)):

conflictCount+=1

return conflictCount

def placeQueen(self,row,col):

if(row >= N or col >= N):

return

if(self.nQueen[col].row == row and self.nQueen[col].col == col):

return

self.nQueen[col].row = row

self.nQueen[col].col = col

self.heuristicVal = self.getHeuristicCost()

def printQueen(self):

for i in range(N):#row

for j in range(N):#col

if(self.nQueen[j].row == i):

print("1", end=" ")

else:

print("0", end=" ")

print()

print()

def printBoardQueen(self):

board = self.getMatrix()

for i in range(N):#row

for j in range(N):#col

print(board[i][j], end=" ")

print()

print()

def getMatrix(self):

board = np.zeros((N, N))

board.astype(int)

for j in range(N):#row

for i in range(N):#col

if(self.nQueen[i].row == j):

board[i][j] = 1

else:

board[i][j] = 0

return board

def initQueens(self):

for col in range(N):

row = random.randint(0,N-1)

self.placeQueen(row,col)

self.moves = 0

self.heuristicVal = self.getHeuristicCost()

def getHeuristicCost(self):

count = 0

for i in range(N):

count = count +self.getConflictCount(self.nQueen[i].row,self.nQueen[i].col)

return count

def score(self):

return self.\_h() + self.\_g()

def \_h(self):

return self.heuristicVal

def \_g(self):

return self.moves

def \_\_cmp\_\_(self, other):

if(other == None):

return False

return self.nQueen == other.nQueen

def \_\_eq\_\_(self, other):

return self.\_\_cmp\_\_(other)

def \_\_hash\_\_(self):

return hash(str(self.nQueen))

def \_\_lt\_\_(self, other):

return self.score() < other.score()

def nextAllState(self):

list1 = []

row = self.moves

for i in range(N):

if(not(self.nQueen[i].row == row and self.nQueen[i].col == i)):

nextState = state(self)

nextState.placeQueen(row,i)

list1.append(nextState)

return list1

def solve(initial\_state):

openset = PriorityQueue()

openset.add(initial\_state)

closed = set()

moves = 0

start = time.time()

while openset:

current = openset.poll()

if current.heuristicVal == 0:

end = time.time()

print('True')

current.printQueen()

break

moves += 1

for state in current.nextAllState():

if state not in closed:

openset.add(state)

closed.add(current)

else:

print('False')

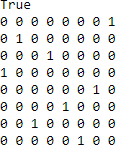
def main():

initial\_state = state(None)

solve(initial\_state)

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Output:**

**Code:** **Recursive Backtracking approach:-**

global N

N = 8

def printSolution(board):

for i in range(N):

for j in range(N):

print (board[i][j],end=" "),

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True:

return True

board[i][col] = 0

return False

def solveNQ():

board = [[0, 0, 0, 0,0, 0, 0, 0],

[0, 0, 0, 0,0, 0, 0, 0],

[0, 0, 0, 0,0, 0, 0, 0],

[0, 0, 0, 0,0, 0, 0, 0],

[0, 0, 0, 0,0, 0, 0, 0],

[0, 0, 0, 0,0, 0, 0, 0],

[0, 0, 0, 0,0, 0, 0, 0],

[0, 0, 0, 0,0, 0, 0, 0]]

if solveNQUtil(board, 0) == False:

print ("Solution does not exist")

return False

printSolution(board)

return True

solveNQ()

**Output:**

