***TIC TAC TOE GAME :***

import os

import time

board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']

player = 1

Win = 1

Draw = -1

Running = 0

Stop = 1

Game = Running

Mark = 'X'

def DrawBoard():

print(" %c | %c | %c " % (board[1], board[2], board[3]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[4], board[5], board[6]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[7], board[8], board[9]))

print(" | | ")

def CheckPosition(x):

if board[x] == ' ':

return True

else:

return False

def CheckWin():

global Game

if board[1] == board[2] and board[2] == board[3] and board[1] != ' ':

Game = Win

elif board[4] == board[5] and board[5] == board[6] and board[4] != ' ':

Game = Win

elif board[7] == board[8] and board[8] == board[9] and board[7] != ' ':

Game = Win

elif board[1] == board[4] and board[4] == board[7] and board[1] != ' ':

Game = Win

elif board[2] == board[5] and board[5] == board[8] and board[2] != ' ':

Game = Win

elif board[3] == board[6] and board[6] == board[9] and board[3] != ' ':

Game = Win

elif board[1] == board[5] and board[5] == board[9] and board[5] != ' ':

Game = Win

elif board[3] == board[5] and board[5] == board[7] and board[5] != ' ':

Game = Win

elif board[1] != ' ' and board[2] != ' ' and board[3] != ' ' and \

board[4] != ' ' and board[5] != ' ' and board[6] != ' ' and \

board[7] != ' ' and board[8] != ' ' and board[9] != ' ':

Game = Draw

else:

Game = Running

print("Tic-Tac-Toe Game Designed By Sourabh Somani")

print("Player 1 [X] --- Player 2 [O]\n")

print()

print()

print("Please Wait...")

time.sleep(3)

while Game == Running:

os.system('cls')

DrawBoard()

if player % 2 != 0:

print("Player 1's chance")

Mark = 'X'

else:

print("Player 2's chance")

Mark = 'O'

choice = int(input("Enter the position between [1-9] where you want to mark: "))

if CheckPosition(choice):

board[choice] = Mark

player + = 1

CheckWin()

os.system('cls')

DrawBoard()

if Game == Draw:

print("Game Draw")

elif Game == Win:

player - = 1

if player % 2 != 0:

print("Player 1 Won")

else:

print("Player 2 Won")

**OUTPUT :**

Player 1 [X] --- Player 2 [O]

Player 1's chance

Enter the position between [1-9] where you want to mark: 1

X | |

| |

| |

Player 2's chance

Enter the position between [1-9] where you want to mark: 5

X | |

| O |

| |

Player 1's chance

Enter the position between [1-9] where you want to mark: 3

X | | X

| O |

| |

Player 2's chance

Enter the position between [1-9] where you want to mark: 2

X | O | X

| O |

| |

Player 1's chance

Enter the position between [1-9] where you want to mark: 8

X | O | X

| O |

| X |

Player 2's chance

Enter the position between [1-9] where you want to mark: 4

X | O | X

O | O |

| X |

Player 1's chance

Enter the position between [1-9] where you want to mark: 6

X | O | X

O | O | X

| X |

Player 2's chance

Enter the position between [1-9] where you want to mark: 7

X | O | X

O | O | X

O | X |

Player 1's chance

Enter the position between [1-9] where you want to mark: 9

X | O | X

O | O | X

O | X | X

Player 1 Won

***BFS :***

from collections import deque

# Define the goal state and initial state

goal\_state = (1, 2, 3, 8, 0, 4, 7, 6, 5)

initial\_state = (2, 8, 3, 1, 6, 4, 7, 0, 5)

# Define the possible moves

moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]

def print\_board(board\_state):

for i in range(0, 9, 3):

print(board\_state[i:i+3])

def get\_next\_states(current\_state):

next\_states = []

zero\_index = current\_state.index(0)

zero\_row, zero\_col = zero\_index // 3, zero\_index % 3

for move in moves:

new\_row, new\_col = zero\_row + move[0], zero\_col + move[1]

if 0 <= new\_row < 3 and 0 <= new\_col < 3:

new\_state = list(current\_state)

new\_index = new\_row \* 3 + new\_col

new\_state[zero\_index], new\_state[new\_index] = new\_state[new\_index], new\_state[zero\_index]

next\_states.append(tuple(new\_state))

return next\_states

def bfs(initial\_state, goal\_state):

visited = set()

queue = deque([(initial\_state, [])])

while queue:

current\_state, path = queue.popleft()

visited.add(current\_state)

if current\_state == goal\_state:

print("Goal state found!")

print\_board(current\_state)

print("Path to goal state:", path)

return

for next\_state in get\_next\_states(current\_state):

if next\_state not in visited:

queue.append((next\_state, path + [next\_state]))

print("Goal state not reachable.")

bfs(initial\_state, goal\_state)

**OUTPUT :**

Goal state found!

(1, 2, 3)

(8, 0, 4)

(7, 6, 5)

Path to goal state: [(2, 8, 3, 1, 0, 4, 7, 6, 5), (2, 0, 3, 1, 8, 4, 7, 6, 5), (0, 2, 3, 1, 8, 4, 7, 6, 5), (1, 2, 3, 0, 8, 4, 7, 6, 5), (1, 2, 3, 8, 0, 4, 7, 6, 5)]

***DFS:***

import copy

from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]

cols = [ 0, -1, 0, 1 ]

class priorityQueue:

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

self.heap = []

def push(self, key):

heappush(self.heap, key)

def pop(self):

return heappop(self.heap)

def empty(self):

if not self.heap:

return True

else:

return False

class nodes:

def \_\_init\_\_(self, parent, mats, empty\_tile\_posi,

costs, levels):

self.parent = parent

self.mats = mats

self.empty\_tile\_posi = empty\_tile\_posi

self.costs = costs

self.levels = levels

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

return self.costs < nxt.costs

def calculateCosts(mats, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mats[i][j]) and

(mats[i][j] != final[i][j])):

count += 1

return count

def newNodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi,

levels, parent, final) -> nodes:

new\_mats = copy.deepcopy(mats)

x1 = empty\_tile\_posi[0]

y1 = empty\_tile\_posi[1]

x2 = new\_empty\_tile\_posi[0]

y2 = new\_empty\_tile\_posi[1]

new\_mats[x1][y1], new\_mats[x2][y2] = new\_mats[x2][y2], new\_mats[x1][y1]

costs = calculateCosts(new\_mats, final)

new\_nodes = nodes(parent, new\_mats, new\_empty\_tile\_posi,

costs, levels)

return new\_nodes

def printMatsrix(mats):

for i in range(n):

for j in range(n):

print("%d " % (mats[i][j]), end = " ")

print()

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatsrix(root.mats)

print()

def solve(initial, empty\_tile\_posi, final):

pq = priorityQueue()

costs = calculateCosts(initial, final)

root = nodes(None, initial,

empty\_tile\_posi, costs, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.costs == 0:

printPath(minimum)

return

for i in range(n):

new\_tile\_posi = [

minimum.empty\_tile\_posi[0] + rows[i],

minimum.empty\_tile\_posi[1] + cols[i], ]

if isSafe(new\_tile\_posi[0], new\_tile\_posi[1]):

child = newNodes(minimum.mats,

minimum.empty\_tile\_posi,

new\_tile\_posi,

minimum.levels + 1,

minimum, final,)

pq.push(child)

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

empty\_tile\_posi = [ 1, 2 ]

solve(initial, empty\_tile\_posi, final)

**OUTPUT :**

1 2 3

5 6 0

7 8 4

1 2 3

5 0 6

7 8 4

1 2 3

5 8 6

7 0 4

1 2 3

5 8 6

0 7 4

***A\* ALGORITHM***

from copy import deepcopy

import numpy as np

# Function to find the best solution path

def best\_solution(state):

best\_sol = np.array([], int).reshape(-1, 9)

count = len(state) - 1

while count != -1:

best\_sol = np.insert(best\_sol, 0, state[count]['puzzle'], 0)

count = state[count]['parent']

return best\_sol.reshape(-1, 3, 3)

# Function to check if a state is unique

def is\_unique(check\_array, all\_states):

for state in all\_states:

if np.array\_equal(check\_array, state['puzzle']):

return False

return True

# Calculate the number of misplaced tiles in a state compared to the goal state

def misplaced\_tiles(puzzle, goal):

ms\_cost = np.sum(puzzle != goal) - 1

return ms\_cost if ms\_cost > 0 else 0

# Identify the coordinates of each value in the goal or initial state

def coordinates(puzzle):

pos = np.array(range(9))

for p, q in enumerate(puzzle):

pos[q] = p

return pos

def evaluate\_misplaced(puzzle, goal):

steps = np.array([('up', [0, 1, 2], -3), ('down', [6, 7, 8], 3), ('left', [0, 3, 6], -1), ('right', [2, 5, 8], 1)],

dtype=[('move', str, 1), ('position', list), ('head', int)])

dt\_state = [('puzzle', list), ('parent', int), ('gn', int), ('hn', int)]

cost\_g = coordinates(goal)

parent = -1

gn = 0

hn = misplaced\_tiles(coordinates(puzzle), cost\_g)

state = np.array([(puzzle, parent, gn, hn)], dt\_state)

dt\_priority = [('position', int), ('fn', int)]

priority = np.array([(0, hn)], dt\_priority)

while True:

priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

position, fn = priority[0]

priority = np.delete(priority, 0, 0)

puzzle, parent, gn, hn = state[position]

puzzle = np.array(puzzle)

blank = int(np.where(puzzle == 0)[0])

gn = gn + 1

for s in steps:

if blank not in s['position']:

open\_states = deepcopy(puzzle)

open\_states[blank], open\_states[blank + s['head']] = open\_states[blank + s['head']], open\_states[blank]

if is\_unique(open\_states, state):

hn = misplaced\_tiles(coordinates(open\_states), cost\_g)

q = np.array([(open\_states, position, gn, hn)], dt\_state)

state = np.append(state, q, 0)

fn = gn + hn

q = np.array([(len(state) - 1, fn)], dt\_priority)

priority = np.append(priority, q, 0)

if np.array\_equal(open\_states, goal):

print('The 8 puzzle is solvable!')

return state

return state

puzzle = []

print("Input values from 0-8 for the start state:")

for i in range(0, 9):

x = int(input("Enter value: "))

puzzle.

goal = []

print("Input values from 0-8 for the goal state:")

for i in range(0, 9):

x = int(input("Enter value: "))

goal.append(x)

state = evaluate\_misplaced(puzzle, goal)

best\_path = best\_solution(state)

print("Best path to the goal state:")

print(str(best\_path).replace('[', ' ').replace(']', ''))

total\_moves = len(best\_path) - 1

print('Steps to reach the goal:', total\_moves)

print('Total generated states:', len(state))

**OUTPUT :**

Input values from 0-8 for the start state:

Enter value: 0

Enter value: 1

Enter value: 2

Enter value: 3

Enter value: 4

Enter value: 5

Enter value: 6

Enter value: 7

Enter value: 8

Input values from 0-8 for the goal state:

Enter value: 1

Enter value: 2

Enter value: 3

Enter value: 4

Enter value: 0

Enter value: 5

Enter value: 6

Enter value: 7

Enter value: 8

Best path to the goal state:

0 1 2 1 0 2 1 2 0 1 2 5

3 4 5 3 4 5 3 4 5 3 4 0

6 7 8 6 7 8 6 7 8 6 7 8

1 2 5 1 2 5 0 2 5 2 0 5

3 0 4 0 3 4 1 3 4 1 3 4

6 7 8 6 7 8 6 7 8 6 7 8

2 3 5 2 3 5 2 3 0 2 0 3

1 0 4 1 4 0 1 4 5 1 4 5

6 7 8 6 7 8 6 7 8 6 7 8

0 2 3 1 2 3 1 2 3

1 4 5 0 4 5 4 0 5

6 7 8 6 7 8 6 7 8

Steps to reach the goal: 14

Total generated states: 534

***HILL CLIMBING ALGORITHM :***

import random

import numpy as np

import networkx as nx

coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7], [15,20], [10,9], [12,12]])

def generate\_matrix(coordinate):

matrix = []

for i in range(len(coordinate)):

for j in range(len(coordinate)) :

p = np.linalg.norm(coordinate[i] - coordinate[j])

matrix.append(p)

matrix = np.reshape(matrix, (len(coordinate),len(coordinate)))

return matrix

def solution(matrix):

points = list(range(0, len(matrix)))

solution = []

for i in range(0, len(matrix)):

random\_point = points[random.randint(0, len(points) - 1)]

solution.append(random\_point)

points.remove(random\_point)

return solution

def path\_length(matrix, solution):

cycle\_length = 0

for i in range(0, len(solution)):

cycle\_length += matrix[solution[i]][solution[i - 1]]

return cycle\_length

def neighbors(matrix, solution):

neighbors = []

for i in range(len(solution)):

for j in range(i + 1, len(solution)):

neighbor = solution.copy()

neighbor[i] = solution[j]

neighbor[j] = solution[i]

neighbors.append(neighbor)

best\_neighbor = neighbors[0]

best\_path = path\_length(matrix, best\_neighbor)

for neighbor in neighbors:

current\_path = path\_length(matrix, neighbor)

if current\_path < best\_path:

best\_path = current\_path

best\_neighbor = neighbor

return best\_neighbor, best\_path

def hill\_climbing(coordinate):

matrix = generate\_matrix(coordinate)

current\_solution = solution(matrix)

current\_path = path\_length(matrix, current\_solution)

neighbor = neighbors(matrix,current\_solution)[0]

best\_neighbor, best\_neighbor\_path = neighbors(matrix, neighbor)

while best\_neighbor\_path < current\_path:

current\_solution = best\_neighbor

current\_path = best\_neighbor\_path

neighbor = neighbors(matrix, current\_solution)[0]

best\_neighbor, best\_neighbor\_path = neighbors(matrix, neighbor)

return current\_path, current\_solution

final\_solution = hill\_climbing(coordinate)

print("The solution is \n", final\_solution[1])

**OUTPUT :**

[2, 4, 8, 3, 7, 5, 0, 6, 9, 10, 1]

***N QUEEN’s (CSP) :***

def is\_safe(board, row, col):

    for i in range(row):

        if board[i][col] == 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, -1, -1), range(col, len(board))):

        if board[i][j] == 1:

            return False

    return True

def solve\_nqueens(N):

    board = [[0 for \_ in range(N)] for \_ in range(N)]

    if solve\_nqueens\_util(board, 0):

        print\_solution(board)

    else:

        print("No solution exists")

def solve\_nqueens\_util(board, row):

    if row == len(board):

        return True

    for col in range(len(board)):

        if is\_safe(board, row, col):

            board[row][col] = 1

            if solve\_nqueens\_util(board, row + 1):

                return True

            board[row][col] = 0

    return False

def print\_solution(board):

    for row in board:

        print(" ".join(["Q" if cell == 1 else "." for cell in row]))

N=4

solve\_nqueens(N)

**OUTPUT :**

Q . . . . . . .

. . . . Q . . .

. . . . . . . Q

. . . . . Q . .

. . Q . . . . .

. . . . . . Q .

. Q . . . . . .

. . . Q . . . .