1. 8 PUZZLE USING ITERATIVE DEEPENING DEPTH FIRST SEARCH ALGORITHM 1

**Code:**

class PuzzleState:

def finite(self, boa rd, empty\_tile\_pos, depth=0, path=[]):

self.board = board

self.empty\_tile\_pos = empty\_tile\_pos # (row, col) self.depth = depth

self.path = path # Keep track of the path taken to reach this state

def is\_goal(self, goal):

return self.board == goal

def generate\_moves(self):

row, col = self.empty\_tile\_pos moves = []

directions = [(-1, 0, ’Up'), (1, 0, 'Down’), (0, -1, 'Left'), (0, 1, ’Right')] # up, down, left, right for dr, dc, move\_name in directions:

new\_row, new\_col = row + dr, col + dc

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

new\_board = self.board[:]

new\_board[row \* 3 + col], new\_board[new\_row \* 3 + new\_col] = new\_board[new\_row \* 3 + new\_col], new\_board[row \* 3 + col]

new\_path = self.path + [move\_name] # Update the path with the new move

moves.append(PuzzleState(new\_board, (new\_row, new\_col), self.depth + 1, new\_path))

return moves

def display(self):

# Display the board in a matrix form

for i in range(0, 9, 3): print(self. board[i:i + 3])

print(f"Moves: (self.path}") # Display the moves taken to reach this state

print() # Newline for better readability

def iddfs(initial\_state, goal, max\_depth): for depth in range(max\_depth + 1):

print(f"Searching at depth: {depth}")

found = dls(initial\_state, goal, depth) if found:

print(f"Goal found at depth: (found.depth}") found.display()

return found

print("Goal not found within max depth.") return None

def dls(state, goal, depth):

if state.is\_goal(goal):

return state if depth <= 0: return None

for move in state.generate\_moves():

print("Current state:")

move.display() # Display the current state

result = dls(move, goal, depth - 1)

if result is not None:

return result

return None

def main():

# User input for initial state, goal state, and maximum depth

initial\_state\_input = input("Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): ")

goal\_state\_input = input("Enter goal state (0 for empty tile, space-separated, e.g. ’1 2 3 4

5 6 7 8 0’): ")

max\_depth = int(input("Enter maximum depth: ")) initial\_board = list(map(int, initial\_state\_input.split())) goal\_board = list(map(int, goal\_state\_input.split()))

empty\_tile\_pos = initial\_board.index(0) // 3, initial\_board.index(0) % 3 # Calculate the

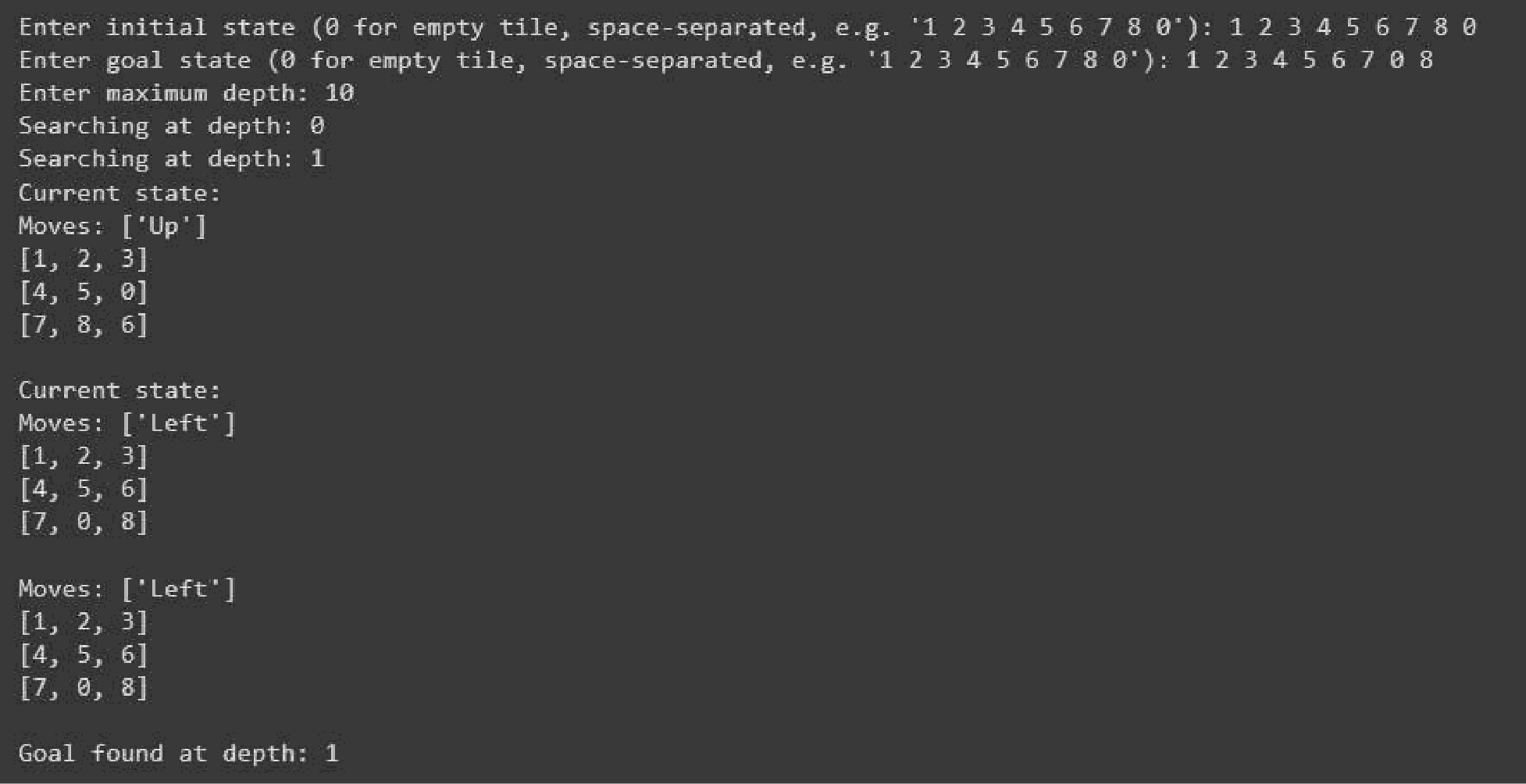
**position of** the **empty tile**

initial\_state= PuzzleState(initial\_board, empty\_tile\_pos) solution = iddfs(initial\_state, goal\_board, max\_depth)

if enamel == " main ":

main()

OUTPUT :



1. N QUEENS PROBLEM USING HILL CLIMBING METHOD

Code:

import random

def calculate\_cost(board):

n = len(board) attacks = 0

for i in range(n):

forj in range(i + 1, n):

if board[i] == board[j]: # Same column attacks += 1

if abs(board[i] - board[j]) == abs(i -j): # Same diagonal attacks += 1

return attacks

def get\_neighbors(board):

neighbors = [] n = len(board)

for col in range(n):

for row in range(n):

if row != board[col]: # Only change the row of the queen

new\_board = board[:] new\_board[col] = row neighbors.append(new\_board)

return neighbors

**def** hill\_climb(board, **max\_resta** rts=100):

current\_cost = **calculate\_cost(board)** print("Initialboard configuration:") print\_board(board, current\_cost) iteration = 0

restarts = 0

while restarts < max\_restarts: # Add limit to the number of restarts while current\_cost != 0: # Continue until cost is zero

neighbors = get\_neighbors(board) best\_neighbor = None

best\_cost = current\_cost

for neighbor in neighbors:

cost= calculate\_cost(neighbor)

if cost < best\_cost: # Looking for a lower cost best\_cost = cost

best\_neighbor = neighbor

if best\_neighbor is None: # No better neighbor found

break # Break the loop if we are stuck at a local minimum

board = best\_neighbor current\_cost = best\_cost iteration += 1 print(f"Iteration(iteration}:")

print\_boa rd(board, current\_cost)

if current\_cost == 0:

break # We found the solution, no need for further restarts else:

# Restart with a new random configuration

board = [random.randint(0, len(board)-1) for\_ in range(len(board))]

current\_cost = calculate\_cost(board)

resta rts += 1

print(f"Resta rt(resta rts}:")

print\_boa rd(board, current\_cost)

return board, current\_cost

def print\_board(boa rd, cost):

n = len(board)

display\_board = [[’.’] \* n for\_ in range(n)] # Create an empty board

for col in range(n):

display\_boa rd[board[col]][col] = 'Q' # Place queens on the boa rd

for row in range(n):

print(' ’.join(display\_board[row])) # Print the board

print(f"Cost: (cost}\n")

if enamel == " main ":

n = int(input("Enter the number of queens (N): ")) # User input for N

initial\_state = list(map(int, input(f"Enter the initial state (row numbers for each column, space-sepa rated): ").split()))

if len(initial\_state) != n or any(r < 0 or r >= n for r in initial\_state):

print("Invalid initial state. Please ensure it has N elements with values from 0 to N -1.") else:

solution, cost = hill\_climb(initial\_state) if cost == 0:

print(f"Solution found with no conflicts:") else:

print(f"No solution found within the restart limit:")

print\_boa rd(solution, cost)

Output:

