LAB-4 - 8 Puzzle with A\* and IDDFS on a Graph

Code: (8 Puzzle with A\*)

import heapq

# Goal state where blank (0) is the first tile

goal\_state = [

[0, 1, 2],

[3, 4, 5],

[6, 7, 8]

]

# Helper functions

def flatten(puzzle):

return [item for row in puzzle for item in row]

def find\_blank(puzzle):

for i in range(3):

for j in range(3):

if puzzle[i][j] == 0:

return i, j

def misplaced\_tiles(puzzle):

flat\_puzzle = flatten(puzzle)

flat\_goal = flatten(goal\_state)

return sum([1 for i in range(9) if flat\_puzzle[i] != flat\_goal[i] and flat\_puzzle[i] != 0])

def generate\_neighbors(puzzle):

x, y = find\_blank(puzzle)

neighbors = []

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

for dx, dy in moves:

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

new\_puzzle = [row[:] for row in puzzle]

new\_puzzle[x][y], new\_puzzle[nx][ny] = new\_puzzle[nx][ny], new\_puzzle[x][y]

neighbors.append(new\_puzzle)

return neighbors

def is\_goal(puzzle):

return puzzle == goal\_state

def print\_puzzle(puzzle):

for row in puzzle:

print(row)

print()

def a\_star\_misplaced\_tiles(initial\_state):

# Priority queue (min-heap) and visited states

frontier = []

heapq.heappush(frontier, (misplaced\_tiles(initial\_state), 0, initial\_state, []))

visited = set()

while frontier:

f, g, current\_state, path = heapq.heappop(frontier)

# Print the current state

print("Current State:")

print\_puzzle(current\_state)

h = misplaced\_tiles(current\_state)

print(f"g(n) = {g}, h(n) = {h}, f(n) = {g + h}")

print("-" \* 20)

if is\_goal(current\_state):

print("Goal reached!")

return path

visited.add(tuple(flatten(current\_state)))

for neighbor in generate\_neighbors(current\_state):

if tuple(flatten(neighbor)) not in visited:

h = misplaced\_tiles(neighbor)

heapq.heappush(frontier, (g + 1 + h, g + 1, neighbor, path + [neighbor]))

return None # No solution found

# Initial puzzle state

initial\_state = [

[1, 2, 0],

[3, 4, 5],

[6, 7, 8]

]

solution = a\_star\_misplaced\_tiles(initial\_state)

if solution:

print("Solution found!")

else:

print("No solution found.")  
Ouput:-





Code: (IDDFS on a Graph )

class Graph:

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

self.adjacency\_list = {}

def add\_edge(self, u, v):

if u not in self.adjacency\_list:

self.adjacency\_list[u] = []

self.adjacency\_list[u].append(v)

def depth\_limited\_dfs(self, node, goal, limit, visited):

if limit < 0:

return False

if node == goal:

return True

visited.add(node)

for neighbor in self.adjacency\_list.get(node, []):

if neighbor not in visited:

if self.depth\_limited\_dfs(neighbor, goal, limit - 1, visited):

return True

visited.remove(node) # Allow revisiting for the next iteration

return False

def iddfs(self, start, goal, max\_depth):

for depth in range(max\_depth + 1):

visited = set()

if self.depth\_limited\_dfs(start, goal, depth, visited):

return True

return False

def main():

graph = Graph()

# Input number of edges

num\_edges = int(input("Enter the number of edges: "))

# Input edges

for \_ in range(num\_edges):

edge = input("Enter an edge (format: A B): ").split()

graph.add\_edge(edge[0], edge[1])

start\_node = input("Enter the start node: ")

goal\_node = input("Enter the goal node: ")

max\_depth = int(input("Enter the maximum depth for IDDFS: "))

if graph.iddfs(start\_node, goal\_node, max\_depth):

print(f"Goal node {goal\_node} found!")

else:

print(f"Goal node {goal\_node} not found within depth {max\_depth}.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:



Obersvation :-





