8 puzzle using iddfs

from collections import deque

# Goal state

GOAL = (1, 2, 3,

4, 5, 6,

7, 8, 0)

# Moves: up, down, left, right

MOVES = {

'U': -3,

'D': 3,

'L': -1,

'R': 1

}

def is\_valid\_move(pos, move):

"""Check if blank can move in given direction."""

if move == 'U' and pos < 3: return False

if move == 'D' and pos > 5: return False

if move == 'L' and pos % 3 == 0: return False

if move == 'R' and pos % 3 == 2: return False

return True

def neighbors(state):

"""Generate valid neighbor states from current state."""

new\_states = []

pos = state.index(0) # blank position

for move, offset in MOVES.items():

if is\_valid\_move(pos, move):

new\_pos = pos + offset

new\_state = list(state)

new\_state[pos], new\_state[new\_pos] = new\_state[new\_pos], new\_state[pos]

new\_states.append((tuple(new\_state), move))

return new\_states

def dls(state, depth, visited, path):

"""Depth-limited DFS."""

if state == GOAL:

return path

if depth == 0:

return None

visited.add(state)

for neighbor, move in neighbors(state):

if neighbor not in visited:

result = dls(neighbor, depth - 1, visited, path + [move])

if result is not None:

return result

return None

def iddfs(start, max\_depth=50):

"""Iterative Deepening DFS."""

for depth in range(max\_depth):

visited = set()

path = dls(start, depth, visited, [])

if path is not None:

return path

return None

# --- MAIN PROGRAM ---

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

print("Enter the start state of the 8-puzzle (use 0 for blank):")

nums = []

for i in range(9):

nums.append(int(input(f"Tile {i+1}: ")))

start = tuple(nums)

solution = iddfs(start, max\_depth=30)

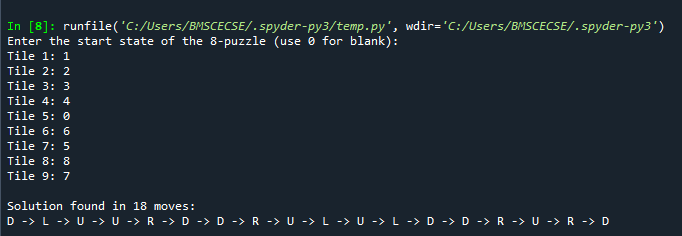
if solution:

print(f"\nSolution found in {len(solution)} moves:")

print(" -> ".join(solution))

else:

print("No solution found within depth limit.")



8 puzzle using A\*

import heapq

# Goal state

GOAL = (1, 2, 3,

4, 5, 6,

7, 8, 0)

# Moves: up, down, left, right

MOVES = {

'U': -3,

'D': 3,

'L': -1,

'R': 1

}

def is\_valid\_move(pos, move):

"""Check if blank can move in given direction."""

if move == 'U' and pos < 3: return False

if move == 'D' and pos > 5: return False

if move == 'L' and pos % 3 == 0: return False

if move == 'R' and pos % 3 == 2: return False

return True

def neighbors(state):

"""Generate valid neighbor states from current state."""

new\_states = []

pos = state.index(0) # blank position

for move, offset in MOVES.items():

if is\_valid\_move(pos, move):

new\_pos = pos + offset

new\_state = list(state)

new\_state[pos], new\_state[new\_pos] = new\_state[new\_pos], new\_state[pos]

new\_states.append((tuple(new\_state), move))

return new\_states

def manhattan(state):

"""Manhattan distance heuristic."""

distance = 0

for idx, value in enumerate(state):

if value == 0:

continue

goal\_idx = GOAL.index(value)

r1, c1 = divmod(idx, 3)

r2, c2 = divmod(goal\_idx, 3)

distance += abs(r1 - r2) + abs(c1 - c2)

return distance

def is\_solvable(state):

"""Check if the 8-puzzle is solvable."""

flat = [x for x in state if x != 0]

inversions = 0

for i in range(len(flat)):

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

if flat[i] > flat[j]:

inversions += 1

return inversions % 2 == 0

def reconstruct\_path(parent, moves, current):

"""Reconstruct path of moves."""

path = []

while current in parent:

path.append(moves[current])

current = parent[current]

return path[::-1]

def a\_star(start):

"""A\* search algorithm for 8-puzzle."""

if not is\_solvable(start):

return None

open\_list = []

g = {start: 0}

parent = {}

moves = {}

heapq.heappush(open\_list, (manhattan(start), start))

while open\_list:

f, current = heapq.heappop(open\_list)

if current == GOAL:

return reconstruct\_path(parent, moves, current)

for neighbor, move in neighbors(current):

tentative\_g = g[current] + 1

if neighbor not in g or tentative\_g < g[neighbor]:

g[neighbor] = tentative\_g

f\_score = tentative\_g + manhattan(neighbor)

heapq.heappush(open\_list, (f\_score, neighbor))

parent[neighbor] = current

moves[neighbor] = move

return None

# --- MAIN PROGRAM ---

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

print("Enter the start state of the 8-puzzle (use 0 for blank):")

nums = []

for i in range(9):

nums.append(int(input(f"Tile {i+1}: ")))

start = tuple(nums)

solution = a\_star(start)

if solution:

print(f"\nSolution found in {len(solution)} moves:")

print(" -> ".join(solution))

else:

print("\nThis puzzle configuration is unsolvable!")

