***8-Puzzle Solver***

**Name**: Ashish Yadav

Roll No: 202401100300076

**Date**: 11 March 2025

**Institution**: KIET Group of Institution

**1. Introduction**

The 8-puzzle is a classical problem in the field of artificial intelligence and computer science. It consists of a 3x3 grid with 8 numbered tiles and one empty space. The goal is to slide the tiles around using the empty space to arrange them in a predefined goal configuration. The problem is often used as a benchmark for search algorithms due to its simplicity and computational tractability.

In this report, we present a solution to the 8-puzzle problem using the *A search algorithm*\*, a well-known informed search method. A\* uses both the **g-cost** (the number of steps taken to reach a state) and a **heuristic** (an estimate of the remaining steps) to find the optimal solution efficiently. Specifically, we use the **Manhattan distance** heuristic to guide the search.

**Methodology**

**3.1. Problem Representation**

Each state of the 8-puzzle is represented as a 3x3 grid or a tuple of tuples. The positions of the tiles are encoded as numbers from 0 to 8, where 0 represents the empty space.

The puzzle’s state space is large, and the number of potential configurations is 9!9!9! (362,880). We need an algorithm that efficiently explores these states and finds the solution.

**3.2. Search Algorithm: A\***

The A\* search algorithm was chosen for this task due to its efficiency in solving problems like the 8-puzzle. It combines **Breadth-First Search** (BFS) for exploration and **Greedy Search** for optimality.

A\* operates with a cost function f(n)f(n)f(n), where:

f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)

* **g(n)** is the actual cost to reach the current state from the start state (i.e., the number of moves).
* **h(n)** is the heuristic estimate of the cost from the current state to the goal state. We use **Manhattan Distance** for this heuristic.

At each step, A\* selects the state with the lowest f(n)f(n)f(n) and expands it until the goal state is reached.

**3.3. Heuristic Function**

In this implementation, we use the **Manhattan distance** as the heuristic function, which calculates the sum of the absolute differences between the current position and the goal position of each tile. This is a well-suited heuristic for the 8-puzzle as it reflects the actual minimal moves required to position a tile at its goal.

For example, if a tile is located at position (i, j) and its goal position is (x, y), the Manhattan distance is:

h=∣i−x∣+∣j−y∣h = |i - x| + |j - y|h=∣i−x∣+∣j−y∣

**3.4. User Input and Output**

The program accepts the initial state of the puzzle as input from the user, where the user enters the configuration row by row, with each row consisting of 3 space-separated values. The input is validated to ensure that the state contains all numbers from 0 to 8 exactly once. Once the input is validated, the algorithm finds the optimal solution path using A\* and prints the sequence of moves to solve the puzzle.

Code Typed

import heapq

# Goal state of the 8 puzzle

goal\_state = (

(1, 2, 3),

(4, 5, 6),

(7, 8, 0)

)

# Directions for sliding tiles: Up, Down, Left, Right

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)] # (row change, column change)

def manhattan\_distance(state):

"""Calculate the Manhattan distance of the current state from the goal state."""

distance = 0

for r in range(3):

for c in range(3):

value = state[r][c]

if value != 0:

goal\_r, goal\_c = (value - 1) // 3, (value - 1) % 3

distance += abs(goal\_r - r) + abs(goal\_c - c)

return distance

def get\_neighbors(state):

"""Generate the possible states by moving the empty space (0)."""

neighbors = []

zero\_r, zero\_c = next((r, c) for r in range(3) for c in range(3) if state[r][c] == 0)

for dr, dc in DIRECTIONS:

new\_r, new\_c = zero\_r + dr, zero\_c + dc

if 0 <= new\_r < 3 and 0 <= new\_c < 3:

new\_state = list(list(row) for row in state)

new\_state[zero\_r][zero\_c], new\_state[new\_r][new\_c] = new\_state[new\_r][new\_c], new\_state[zero\_r][zero\_c]

neighbors.append(tuple(tuple(row) for row in new\_state))

return neighbors

def a\_star\_search(start\_state):

"""Solve the 8-puzzle using A\* search algorithm."""

# Priority queue: stores (f\_cost, g\_cost, current\_state, path)

open\_list = []

heapq.heappush(open\_list, (manhattan\_distance(start\_state), 0, start\_state, []))

closed\_set = set()

closed\_set.add(start\_state)

while open\_list:

f\_cost, g\_cost, current\_state, path = heapq.heappop(open\_list)

if current\_state == goal\_state:

return path # Solution found

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

if neighbor not in closed\_set:

closed\_set.add(neighbor)

h\_cost = manhattan\_distance(neighbor)

f\_cost = g\_cost + 1 + h\_cost

heapq.heappush(open\_list, (f\_cost, g\_cost + 1, neighbor, path + [neighbor]))

return None # No solution

def print\_state(state):

"""Helper function to print the state in a readable format."""

for row in state:

print(" ".join(str(x) for x in row))

print()

def get\_user\_input():

"""Helper function to get the initial state from the user."""

print("Enter the 8-puzzle initial state (use 0 for the empty space):")

initial\_state = []

for i in range(3):

while True:

try:

row = input(f"Enter row {i + 1} (space-separated values): ").strip().split()

row = [int(x) for x in row]

if len(row) != 3 or any(x < 0 or x > 8 for x in row):

raise ValueError("Each row must contain exactly three numbers between 0 and 8.")

initial\_state.append(tuple(row))

break

except ValueError as e:

print(f"Invalid input: {e}. Please try again.")

return tuple(initial\_state)

# Main execution

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

start\_state = get\_user\_input()

print("\nStart State:")

print\_state(start\_state)

solution = a\_star\_search(start\_state)

if solution is not None:

print("Solution path:")

for step in solution:

print\_state(step)

else:

print("No solution found.")

OUTPUT



