**Experiment:3**

**Aim: Write a Program to Implement** **8-Puzzle problem.**

**Description of 8-Puzzle problem:**

The 8-puzzle problem is a classic problem in artificial intelligence and computer science. It involves a 3x3 grid with 8 numbered tiles and one empty space, arranged in some initial configuration. The goal is to reach a particular goal state by sliding tiles into the empty space, ultimately achieving the desired arrangement of numbers. Here's a step-by-step breakdown of the problem:

**Initial State:** The problem starts with an initial configuration of the 8-puzzle. This configuration can be represented as a 3x3 grid with numbers 1 through 8 and an empty space. For instance:

1 2 3

4 5 6

7 8 –

Here, "-" represents the empty space.

**Goal State:** The goal state is the desired configuration of the puzzle. It's typically defined as follows:

1 2 3

4 7 8

5 6 –

This configuration has all numbers arranged in ascending order from 1 to 8, with the empty space at the bottom right corner.

**Operators:** The legal moves or operators in the 8-puzzle problem involve sliding a tile adjacent to the empty space into the empty space. For example, if the empty space is in the middle of the puzzle, you can slide a neighbouring tile (up, down, left, or right) into the empty space.

**Transition Model:** The transition model defines how the puzzle state changes when a move is made. When a tile is slid into the empty space, the positions of the tile and the empty space are swapped.

**Path Cost:** In most cases, each move in the 8-puzzle problem is considered to have a uniform cost of 1. The total path cost is the number of moves required to reach the goal state from the initial state.

**Heuristic Function:** To guide search algorithms towards finding efficient solutions, a heuristic function is often used. One common heuristic function for the 8-puzzle problem is the Manhattan distance heuristic. It calculates the sum of the distances that each tile is away from its goal position. The heuristic function provides an estimate of how far the current state is from the goal state.

**Search Algorithms:** Various search algorithms such as Breadth-First Search, Depth-First Search, A\* Search, and others can be applied to solve the 8-puzzle problem. A\* Search, in particular, is widely used because it combines the advantages of completeness, optimality, and efficiency by using a heuristic function to guide the search.

**Solution of 8-Puzzle problem**

Solving the 8-puzzle problem involves finding a sequence of moves that transforms the initial configuration of the puzzle into the goal configuration. Here's a high-level overview of how the 8-puzzle problem can be solved:

* **Define the Initial and Goal States:** Start by defining the initial state of the 8-puzzle, where the numbers are arranged in some order with one empty space and define the goal state where the numbers are arranged in ascending order with the empty space in the bottom-right corner.
* **Implement Operators:** Implement the operators that allow you to move tiles in the puzzle. These operators should slide a tile adjacent to the empty space into the empty space.
* **Search Algorithms:** Choose an appropriate search algorithm to find the solution. Common search algorithms used for the 8-puzzle problem include Breadth-First Search, Depth-First Search, and A\* Search.
* **Heuristic Function:** If using A\* Search, implement a heuristic function that estimates the cost of reaching the goal state from a given state. The Manhattan distance heuristic is commonly used for the 8-puzzle problem.
* **Apply the Search Algorithm:** Apply the chosen search algorithm to explore the state space of the puzzle and find the optimal or near-optimal solution. The search algorithm will generate a sequence of moves that transforms the initial state into the goal state.
* **Execute the Solution:** Once the solution sequence is obtained, execute the sequence of moves on the initial state to reach the goal state.

Here's a brief example of how the A\* Search algorithm can be used to solve the 8-puzzle problem:

**Start with the initial state of the puzzle:** Generate possible successor states by applying the valid operators (moving tiles) to the current state.

**Calculate the cost of each successor state using the heuristic function (Manhattan distance):** Expand the state with the lowest estimated cost (f(n) = g(n) + h(n)), where g(n) is the cost of reaching the current state and h(n) is the estimated cost to reach the goal state.

**Repeat the process until the goal state is reached.**

A\* Search ensures that the algorithm explores the most promising paths first based on the heuristic function, leading to an optimal or near-optimal solution.

**Implementation**

Initially, we move the blank space in the initial state in every direction and determine the f-score for each state. Extending the current state is what is meant by this. The current state is expanded and then pushed into the closed list, while the newly created states are pushed into the open list. A state is chosen and expanded once again based on its lowest f-score. Until the goal state becomes the existing state, this process keeps going. In this case, we are essentially giving the algorithm a metric by which to select its actions. After determining which course of action is optimal, the algorithm takes it. By expanding the node with the lowest f-score, the technique addresses the problem of producing redundant child states.

**Implementation of 8-Puzzle problem using A\* Algorithm:**

import heapq

# Define the goal state

goal\_state = [[1, 2, 3],

[4, 5, 6],

[7, 8, 0]] # 0 represents the blank space

# Define the heuristic function (Manhattan distance)

def heuristic(state):

distance = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0:

row, col = divmod(state[i][j] - 1, 3)

distance += abs(row - i) + abs(col - j)

return distance

# Define the Node class to represent states of the puzzle

class Node:

def \_\_init\_\_(self, state, parent=None, move=None, cost=0):

self.state = state

self.parent = parent

self.move = move

self.cost = cost

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

return (self.cost + heuristic(self.state)) < (other.cost + heuristic(other.state))

# Define the A\* search algorithm

def astar(start\_state):

# Initialize the priority queue with the initial state

open\_list = []

heapq.heappush(open\_list, Node(start\_state))

# Initialize the closed list

closed\_list = set()

# Start A\* search

while open\_list:

# Pop the node with the lowest cost from the priority queue

current\_node = heapq.heappop(open\_list)

# Check if the current state is the goal state

if current\_node.state == goal\_state:

# Construct the path from the goal state to the initial state

path = []

while current\_node:

path.append(current\_node.state)

current\_node = current\_node.parent

return path[::-1] # Return the path in reverse order

# Add the current state to the closed list

closed\_list.add(tuple(map(tuple, current\_node.state)))

# Generate successor states

for move in [(0, -1), (0, 1), (-1, 0), (1, 0)]:

new\_state = [row[:] for row in current\_node.state]

row, col = next((i, j) for i, row in enumerate(current\_node.state) for j, val in enumerate(row) if val == 0)

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

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

new\_state[row][col],new\_state[new\_row][new\_col]=new\_state[new\_row][new\_col], new\_state[row][col]

if tuple(map(tuple, new\_state)) not in closed\_list:

heapq.heappush(open\_list, Node(new\_state, current\_node, move, current\_node.cost + 1)))

# Define the initial state of the puzzle

initial\_state = [[1, 2, 3],

[4, 5, 6],

[7, 0, 8]] # 0 represents the blank space

# Solve the 8-Puzzle problem using A\* search

solution = astar(initial\_state)

# Print the solution path

if solution:

print("Solution Path:")

for state in solution:

for row in state:

print(row)

print()

else:

print("No solution exists.")

**Output:**

Solution Path:

[1, 2, 3]

[4, 5, 6]

[7, 0, 8]

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

**Date of experiment performed:**

**Day of experiment performed:**

**Date of experiment submission:**

**Day of experiment Submission:**

Faculty Co-ordinator Signature