EXPERIMENT – 02

AIM:

To Implement an 8 Puzzle Problem using Heuristic Search Technique.

DESCRIPTION:

The 8-puzzle, often regarded as a small, solvable piece of a larger puzzle, holds a central place in AI because of its relevance in understanding and developing algorithms for more complex problems.

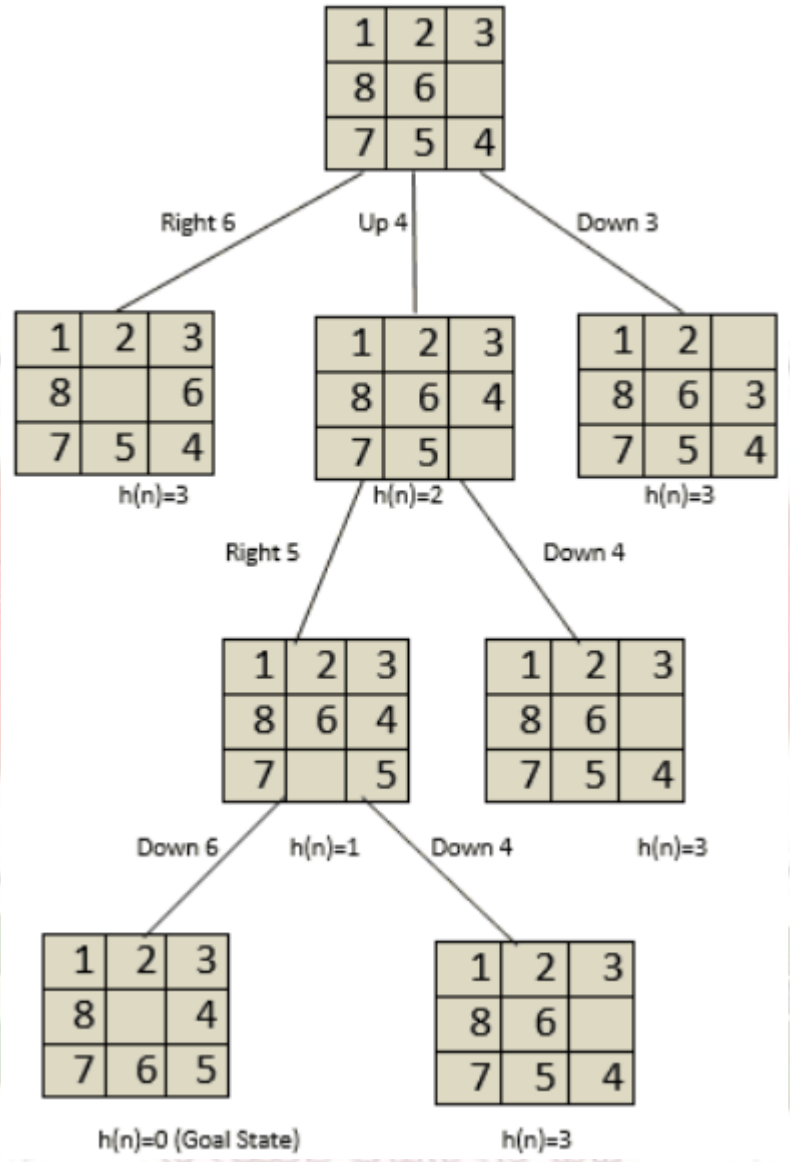
To tackle the 8-puzzle, it's crucial to comprehend its rules and constraints:

- The 8-puzzle is typically played on a 3x3 grid, which provides a 3x3 square arrangement for tiles. This grid structure is fundamental to the problem's organization.
- The puzzle comprises 8 numbered tiles (usually from 1 to 8) and one blank tile. These numbered tiles can be slid into adjacent positions (horizontally or vertically) when there's an available space, which is occupied by the blank tile.
- The objective of the 8-puzzle is to transform an initial state, defined by the arrangement of the tiles on the grid, into a specified goal state. The goal state is often a predefined configuration, such as having the tiles arranged in ascending order from left to right and top to bottom, with the blank tile in the bottom-right corner.

The state of the 8-puzzle is represented using a 3x3 grid, where each cell can hold one of the numbered tiles or remain empty (occupied by the blank tile).

- In a 3x3 grid, each cell can contain one of the following elements:
  - Numbered tiles, typically from 1 to 8.
  - A blank tile, represented as an empty cell.
- The arrangement of these elements in the grid defines the state of the puzzle. The state represents the current position of the tiles within the grid, which can vary as the puzzle is manipulated.

EXAMPLE:



ALGORITHM:

1. Initialize the open list
2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)
3. while the open list is not empty
   a) find the node with the least f on the open list, call it "q"
   b) pop q off the open list

c) generate q's 8 successors and set their parents to q
d) for each successor
i) if successor is the goal, stop search
ii) else, compute both g and h for successor
successor.g = q.g + distance between successor and q
successor.h = distance from goal to successor
(This can be done using many ways, we will discuss three heuristics-
Manhattan, Diagonal and Euclidean Heuristics)
successor.f = successor.g + successor.h
iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
iv) if a node with the same position as is in the CLOSED list which has a lower f than successor, skip this successor, add the node to the open list

end (for loop)

e) push q on the closed list
end (while loop)

CODE:

```
import heapq

def manhattan_distance(state, goal):
    distance = 0
    for i in range(1, 9):
        current_index = state.index(i)
        goal_index = goal.index(i)
        current_row, current_col = divmod(current_index, 3)
        goal_row, goal_col = divmod(goal_index, 3)
        distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

class Node:
    def __init__(self, state, parent=None, move=None, depth=0, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = cost

    def __lt__(self, other):
```

```
        return (self.cost + self.depth) < (other.cost + other.depth)

def get_neighbors(state):
    neighbors = []
    index = state.index(0)
    row, col = divmod(index, 3)

    if row > 0:
        new_state = state[:]
        new_state[index], new_state[index - 3] = new_state[index - 3], new_state[index]
        neighbors.append((new_state, "Up"))

    if row < 2:
        new_state = state[:]
        new_state[index], new_state[index + 3] = new_state[index + 3], new_state[index]
        neighbors.append((new_state, "Down"))

    if col > 0:
        new_state = state[:]
        new_state[index], new_state[index - 1] = new_state[index - 1], new_state[index]
        neighbors.append((new_state, "Left"))
    if col < 2:
        new_state = state[:]
        new_state[index], new_state[index + 1] = new_state[index + 1], new_state[index]
        neighbors.append((new_state, "Right"))

    return neighbors

def a_star(start, goal):
    open_list = []
    closed_list = set()

    start_node = Node(start, None, None, 0, manhattan_distance(start, goal))
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.state == goal:
            moves = []
            while current_node.parent:
                moves.append(current_node.move)
                current_node = current_node.parent
```

```
            return moves[::-1]

        closed_list.add(tuple(current_node.state))

        for neighbor, move in get_neighbors(current_node.state):
            if tuple(neighbor) in closed_list:
                continue

            neighbor_node = Node(
                neighbor,
                current_node,
                move,
                current_node.depth + 1,
                manhattan_distance(neighbor, goal)
            )

            heapq.heappush(open_list, neighbor_node)

    return None

def get_puzzle_input():
    print("Enter the 8-puzzle start state (9 numbers, 0 for the empty space):")
    start = list(map(int, input().split()))
    if len(start) != 9 or any(x < 0 or x > 8 for x in start):
        print("Invalid input. Please enter exactly 9 numbers between 0 and 8.")
        return None, None
    print("Enter the 8-puzzle goal state (9 numbers, 0 for the empty space):")
    goal = list(map(int, input().split()))
    if len(goal) != 9 or any(x < 0 or x > 8 for x in goal):
        print("Invalid input. Please enter exactly 9 numbers between 0 and 8.")
        return None, None

    return start, goal

start, goal = get_puzzle_input()
if start and goal:
    solution = a_star(start, goal)
    if solution:
        print("Moves to solve the puzzle:", solution)
    else:
        print("No solution found.")
```

OUTPUT:

Enter the 8-puzzle start state (9 numbers, 0 for the empty space):
1 2 3 8 6 0 7 5 4
Enter the 8-puzzle goal state (9 numbers, 0 for the empty space):
1 2 3 8 0 4 7 6 5
Moves to solve the puzzle: ['Down', 'Left', 'Up']

OUTPUT ANALYSIS:

The 8-puzzle problem consists of 9 tiles (8 numbered tiles and one blank space), which can be arranged in 9! (factorial of 9) possible ways. However, only half of these configurations are solvable, so the number of valid states in the state space is 9!/2=181,440.

Time Complexity

The time complexity of solving the 8-puzzle problem using an AI algorithm like A* depends on several factors:

Branching Factor (b): In the worst case, each state can generate up to 4 new states (by moving the blank tile up, down, left, or right). However, on average, the effective branching factor is usually around 2.5 to 3.

Depth of Solution (d): This is the length of the shortest path from the start state to the goal state in the state space.

Heuristic Quality: The efficiency of A* depends heavily on the heuristic used. A good heuristic, like Manhattan distance, will prune many non-optimal paths, reducing the time complexity.

In the worst case, the time complexity of A* search is: O(b^d)

For the 8-puzzle problem:

i. b≈3 (average branching factor)
ii. d varies depending on the specific instance of the puzzle.

Thus, the time complexity can be approximately O(3^d)

Space Complexity

The space complexity is also a crucial consideration, as A* search needs to store all generated nodes in memory.

The space complexity of A* is generally:O(b^d)

This is because A* stores all nodes in the open and closed lists, which can be as large as the number of nodes generated up to the solution depth.

CHAITANYA BHARATHI
INSTITUTE OF TECHNOLOGY
స్వయం తేజస్వినా భవ
1979