Experiment No: 8 Date:

8-Puzzle Problem Using Best First Search

Aim: To Implement the 8-Puzzle Problem using Best First Search

Theory:

Introduction:
The 8-puzzle problem is a classic problem in the field of artificial intelligence and computer science. It involves a 3x3 grid with eight numbered tiles and one empty space(0), with the goal being to rearrange the tiles from a given initial state to a goal state by sliding tiles into the empty space.

Best First Search is an informed search algorithm that explores a graph by expanding the most promising node chosen based on a specified heuristic function. In the context of the 8-puzzle problem, the heuristic function typically evaluates each possible board configuration based on the number of misplaced tiles, which counts the number of tiles that are not in their goal positions.

Approach:

- State Representation: Represent each state as a 3x3 grid, where each cell contains a number from 1 to 8 and one empty space.
- Initial State: Start with an initial configuration of the puzzle.
- Goal State: A state is a goal state if the puzzle is arranged in a specific configuration, usually in ascending order from left to right, top to bottom, with the empty space in the bottom-right corner.
- Successor Function: Generate successor states by moving tiles into the empty space. Each successor state is generated by moving a tile adjacent to the empty space into the empty space.
- Heuristic Function: Evaluate each state based on the number of misplaced tiles heuristic, which counts the number of tiles that are not in their goal positions. The goal is to minimize this value.
- This heuristic estimates the distance from the current state to the goal state by counting the number of tiles that are not in their correct positions. The lower the number of misplaced tiles, the closer the current state is to the goal state.

Advantages of Best First Search:

- Efficiency: Best First Search tends to explore promising areas of the search space first, potentially leading to faster convergence to the goal state.
- Domain-specific Knowledge: By incorporating domain-specific heuristic knowledge, Best First Search can guide the search towards more promising solutions, especially in large or complex problem spaces.

Limitations of Best First Search:

- Completeness: Best First Search is not guaranteed to find a solution if one exists. It may get stuck in local optima or loops.

- Heuristic Accuracy: The effectiveness of Best First Search heavily relies on the accuracy of the heuristic function. A poorly chosen heuristic may lead to suboptimal or incorrect solutions.

Algorithm:

```
BestFirstSearch()

open ← ((start NIL))
closed ← ()
while not Null(open)
  do nodePair ← Head(open)
     node ← Head(nodePair)
     if GoalTest(node) = TRUE
       then return ReconstructPath(nodePair, closed)
       else closed ← Cons(nodePair, closed)
            children ← MoveGen(node)
            noLoops ← RemoveSeen(children, open, closed)
            new ← MakePairs(noLoops, node)
            OPEN ← sort_h (append (NEW, tail(OPEN)))
return "No solution found"
```

Example:

initial state:

2 8 3

1 6 4

7 0 5

Heuristic = 5

goal state:

1 2 3

8 0 4

7 6 5

- Generate possible moves from the initial state.
- Choose the best move based on the misplaced tiles heuristic.
- Repeat until the goal state is reached.

steps to reach the goal state

```
 2 8 3      2 0 3     0 2 3     1 2 3
 1 0 4  => 1 8 4  => 1 8 4  => 8 0 4
 7 6 5      7 6 5     7 6 5     7 6 5
```

Program:

```
from queue import PriorityQueue
#1 2 3 7 8 4 6 0 5
def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i + 3])

def heuristic(state, goal_state):
    return sum(1 for i in range(9) if state[i] != goal_state[i])

def move(state, direction):
    blank_index = state.index(0)
    new_state = list(state)
    if direction == 'up' and blank_index > 2:
        new_state[blank_index], new_state[blank_index - 3] = new_state[blank_index - 3],
new_state[blank_index]
    elif direction == 'down' and blank_index < 6:
        new_state[blank_index], new_state[blank_index + 3] = new_state[blank_index + 3],
new_state[blank_index]
    elif direction == 'left' and blank_index % 3 != 0:
        new_state[blank_index], new_state[blank_index - 1] = new_state[blank_index - 1],
new_state[blank_index]
    elif direction == 'right' and blank_index % 3 != 2:
        new_state[blank_index], new_state[blank_index + 1] = new_state[blank_index + 1],
new_state[blank_index]
    else:
        return None
    return tuple(new_state)

def generate_neighbors(state):
    neighbors = []
    directions = ['up', 'down', 'left', 'right']
    for direction in directions:
        neighbor = move(state, direction)
        if neighbor is not None:
            neighbors.append(neighbor)
    return neighbors

def solve_puzzle(initial_state, goal_state, max_steps=10):
    current_state = initial_state
    current_cost = heuristic(initial_state, goal_state)
    steps = 0
    path = [current_state]
    print_state(current_state)
    print("Current heuristic : ",current_cost)
    print()
    priority_queue = PriorityQueue()
    priority_queue.put((current_cost, current_state))

    while not priority_queue.empty() and steps < max_steps:
        current_cost, current_state = priority_queue.get()
```

```
        if current_cost == 0:
            print_state(current_state)
            print("Current heuristic : ",current_cost)
            print()
            print("Goal state reached!")
            break

        neighbors = generate_neighbors(current_state)
        for neighbor in neighbors:
            neighbor_cost = heuristic(neighbor, goal_state)
            priority_queue.put((neighbor_cost, neighbor))

        if path[-1] != current_state:  # Avoid duplicate states
            path.append(current_state)
            print_state(current_state)
            print("Current heuristic : ",current_cost)
            print()
        steps += 1
    if current_cost > 0:
        print("No solution found")
initial_state = tuple(map(int, input("Enter initial state : ").split()))
goal_state = (1, 2, 3, 8, 0, 4, 7, 6, 5)
solve_puzzle(initial_state, goal_state)
```

Output

```
Enter initial state : 1 2 3 7 8 4 6 0 5
(1, 2, 3)
(7, 8, 4)
(6, 0, 5)
Current heuristic :  4

(1, 2, 3)
(7, 0, 4)
(6, 8, 5)
Current heuristic :  3

(1, 2, 3)
(7, 8, 4)
(0, 6, 5)
Current heuristic :  3

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)
Current heuristic :  2

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
Current heuristic :  0

Goal state reached!
```

Conclusion: Solved 8-puzzle problem using Best First Search with successful execution of programs.