AI LAB 5

8 PUZZLE PROBLEM BEST FIRST SEARCH AND A* ALGORITHM

1BM21CS203

Best first search:

```
import heapq


class Node:
    def __init__(self, state, level, heuristic):
        self.state = state
        self.level = level
        self.heuristic = heuristic

    def __lt__(self, other):
        return self.heuristic < other.heuristic

def generate_child(node):
    x, y = find_blank(node.state)
    moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
    children = []

    for move in moves:
        child_state = move_blank(node.state, (x, y), move)
        if child_state is not None:
            h = calculate_heuristic(child_state)
            child_node = Node(child_state, node.level + 1, h)
            children.append(child_node)

    return children

def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
```

```
                return i, j

def move_blank(state, src, dest):
    x1, y1 = src
    x2, y2 = dest

    if 0 <= x2 < 3 and 0 <= y2 < 3:
        new_state = [row[:] for row in state]
        new_state[x1][y1], new_state[x2][y2] = new_state[x2][y2], new_state[x1][y1]
        return new_state
    else:
        return None

def calculate_heuristic(state):
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    h = 0

    for i in range(3):
        for j in range(3):
            if state[i][j] != goal_state[i][j] and state[i][j] != 0:
                h += 1

    return h

def best_first_search(initial_state):
    start_node = Node(initial_state, 0, calculate_heuristic(initial_state))
    open_list = [start_node]
    closed_set = set()

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]:
            return current_node
```

```
        closed_set.add(tuple(map(tuple, current_node.state)))

        for child in generate_child(current_node):
            if tuple(map(tuple, child.state)) not in closed_set:
                heapq.heappush(open_list, child)

    return None

# Example initial state
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]

# Run Best-First Search
solution_node = best_first_search(initial_state)

# Display solution
if solution_node:
    print("Solution is given in", solution_node.level, "moves.")
    print("Path:")
    for row in solution_node.state:
        print(row)
else:
    print("No solution found.")
```

```
Solution is given in 3 moves.
Path:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Time complexity of Best First Search:

In the worst case, where all nodes must be expanded before finding the goal state, the time complexity is exponential. Specifically, it can be expressed as O(b^d), where b is the branching factor and d is the depth of the solution.

A* Algorithm:

```
import heapq


class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find(self.data, '_')
        val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
                child_node = Node(child, self.level+1, 0)
                children.append(child_node)
        return children

    def shuffle(self, puz, x1, y1, x2, y2):
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None

    def copy(self, root):
        temp = []
        for i in root:
            t = []
```

```
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self, puz, x):
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if puz[i][j] == x:
                    return i, j



class Puzzle:
    def __init__(self, size):
        self.n = size
        self.open = []
        self.closed = []

    def f(self, start, goal):
        return self.h(start.data, goal) + start.level

    def h(self, start, goal):
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp

    def process(self, start_data, goal_data):
        start = Node(start_data, 0, 0)
        start.fval = self.f(start, goal_data)
        self.open.append(start)
        print("\n\n")
```

```
        while True:
            cur = self.open[0]


            print("\n")
            for i in cur.data:
                for j in i:
                    print(j, end=" ")
                print("")
            if self.h(cur.data, goal_data) == 0:
                break
            for i in cur.generate_child():
                i.fval = self.f(i, goal_data)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]
            self.open.sort(key=lambda x: x.fval, reverse=False)



# Define puzzle states
start_state = [['1', '2', '3'], ['_', '4', '6'], ['7', '5', '8']]
goal_state = [['1', '2', '3'], ['4', '5', '6'], ['7', '8', '_']]

# Create Puzzle object and run the process
puz = Puzzle(3)
puz.process(start_state, goal_state)
```

```
1 2 3
_ 4 6
7 5 8


1 2 3
4 _ 6
7 5 8


1 2 3
4 5 6
7 _ 8


1 2 3
4 5 6
7 8 _
```

Time Complexity of A* Algorithm: The time complexity of A* depends on the heuristic function. In the best case, if the heuristic is perfect, the time complexity is linear (O(d)), where d is the depth of the solution. In the worst case, it can be exponential (O(b^d)).