Lab 4 : 8 puzzle with A* and IDFS

Observation book:

classmate
Date
Page

10.24

LAB-4

~~Iterating deepening Search Algorithm~~ :

A* Algorithm:

Initial state			Goal state		
[1	2	3]	[2	8	1]
[8	0	4]	[0	4	3]
[7	6	5]	[7	6	5]

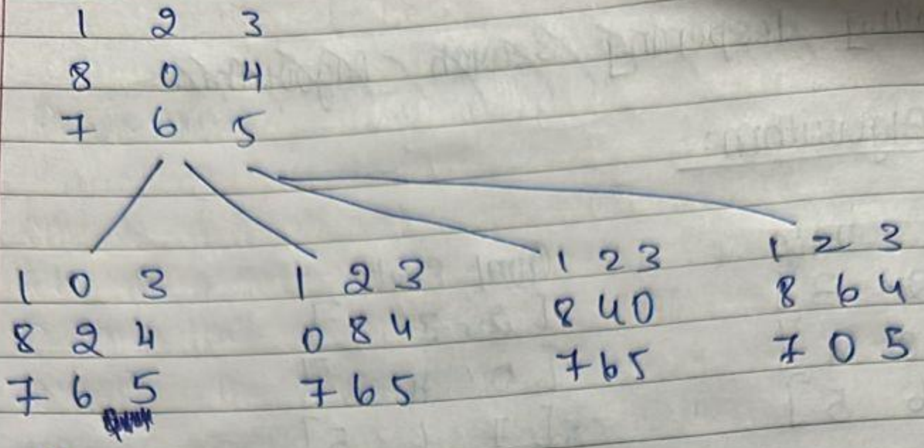
* ~~Open list~~
* Initialize : initial state of the puzzle
  - Set the goal state

* Use a priority queue to store states of the puzzle, prioritized by $f(n) = g(n) + h(n)$;
  - $g(n)$ is the no. of moves
  - $h(n)$ counts how many tiles are not in their position.

* Remove the state with the smallest $f(n)$ from the queue.

* If ~~it~~ the state is the goal state, then stop and return solution.

* For each new state calculate $f(n) = g(n) + h$ and add to priority queue.



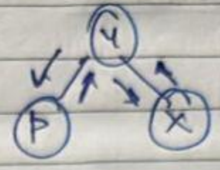
IDFS

function IDDFS (root, goal)
    for d = 0 to INT_MAX
        root = DFS (root, goal, depth)
        if result
            return result
    return NULL

function DLS (root, goal, depth)
if depth = 0:
    if root = goal
            return root
    return NULL
else
    result = DLS (root, goal, depth - 1)
    if result:
            return result
return NULL

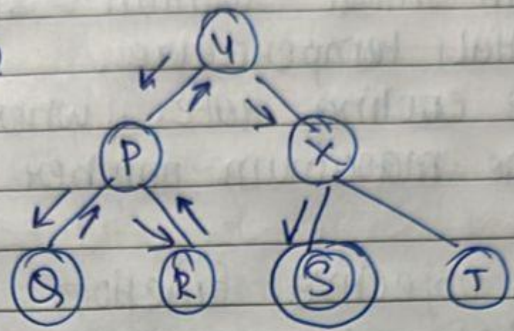
At depth = 0   4   4 return NULL

At depth 1



Y → P → X
return NULL

At depth 2



Y P Q R X S

return F

15/10

Code:

A* algorithm:

```
import heapq
goal_state = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
]
def flatten(puzzle):
    return [item for row in puzzle for item in row]
def find_blank(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == 0:
                return i, j
def misplaced_tiles(puzzle):
    flat_puzzle = flatten(puzzle)
    flat_goal = flatten(goal_state)
    return sum([1 for i in range(9) if flat_puzzle[i] != flat_goal[i] and flat_puzzle[i]
!= 0])
def generate_neighbors(puzzle):
    x, y = find_blank(puzzle)
    neighbors = []
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
```

```
            new_puzzle = [row[:] for row in puzzle]
            new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny],
new_puzzle[x][y]
            neighbors.append(new_puzzle)
    return neighbors


def is_goal(puzzle):
    return puzzle == goal_state


def print_puzzle(puzzle):
    for row in puzzle:
        print(row)
    print()


def a_star_misplaced_tiles(initial_state):
    frontier = []
    heapq.heappush(frontier, (misplaced_tiles(initial_state), 0, initial_state, []))
    visited = set()
    while frontier:
        f, g, current_state, path = heapq.heappop(frontier)
        print("Current State:")
        print_puzzle(current_state)
        h = misplaced_tiles(current_state)
        print(f"g(n) = {g}, h(n) = {h}, f(n) = {g + h}")
        print("-" * 20)

        if is_goal(current_state):
```

```
            print("Goal reached!")
            return path
        visited.add(tuple(flatten(current_state)))
        for neighbor in generate_neighbors(current_state):
            if tuple(flatten(neighbor)) not in visited:
                h = misplaced_tiles(neighbor)
                heapq.heappush(frontier, (g + 1 + h, g + 1, neighbor, path +
[neighbor]))
    return None
initial_state = [
    [1, 2, 0],
    [3, 4, 5],
    [6, 7, 8]
]
solution = a_star_misplaced_tiles(initial_state)
if solution:
    print("Solution found!")
else:
    print("No solution found.")
print("Navya 1bm22cs175")
```

Output:

```
Current State:
[1, 2, 0]
[3, 4, 5]
[6, 7, 8]

g(n) = 0, h(n) = 2, f(n) = 2
--------------------
Current State:
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]

g(n) = 1, h(n) = 1, f(n) = 2
--------------------
Current State:
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]

g(n) = 2, h(n) = 0, f(n) = 2
--------------------
Goal reached!
Solution found!
Navya 1bm22cs175
>>>
```

IDFS:

Code:

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_edge(self, u, v):
        if u not in self.adjacency_list:
            self.adjacency_list[u] = []
        self.adjacency_list[u].append(v)

    def depth_limited_dfs(self, node, goal, limit, visited):
        if limit < 0:
            return False
        if node == goal:
```

```
            return True
        visited.add(node)
        for neighbor in self.adjacency_list.get(node, []):
            if neighbor not in visited:
                if self.depth_limited_dfs(neighbor, goal, limit - 1, visited):
                    return True
        visited.remove(node)  # Allow revisiting for the next iteration
        return False

    def iddfs(self, start, goal, max_depth):
        for depth in range(max_depth + 1):
            visited = set()
            if self.depth_limited_dfs(start, goal, depth, visited):
                return True
        return False


def main():
    graph = Graph()

    # Input number of edges
    num_edges = int(input("Enter the number of edges: "))

    # Input edges
    for _ in range(num_edges):
        edge = input("Enter an edge (format: A B): ").split()
```

```
        graph.add_edge(edge[0], edge[1])

    start_node = input("Enter the start node: ")
    goal_node = input("Enter the goal node: ")
    max_depth = int(input("Enter the maximum depth for IDDFS: "))

    if graph.iddfs(start_node, goal_node, max_depth):
        print(f"Goal node {goal_node} found!")
    else:
        print(f"Goal node {goal_node} not found within depth {max_depth}.")


if __name__ == "__main__":
    main()
print("Navya 1bm22cs175")
```

Output:

```
==================== RESTART: C:\Users\NAVYA\Desktop\c
Enter the number of edges: 5
Enter an edge (format: A B): A B
Enter an edge (format: A B): B C
Enter an edge (format: A B): C D
Enter an edge (format: A B): D E
Enter an edge (format: A B): E F
Enter the start node: A
Enter the goal node: F
Enter the maximum depth for IDDFS: 3
Goal node F not found within depth 3.
Navya 1bm22cs175
```