LAB-4: 8-Puzzle with A* and IDFS

Observation book:

Date 14/10/2024
Page

IDS:

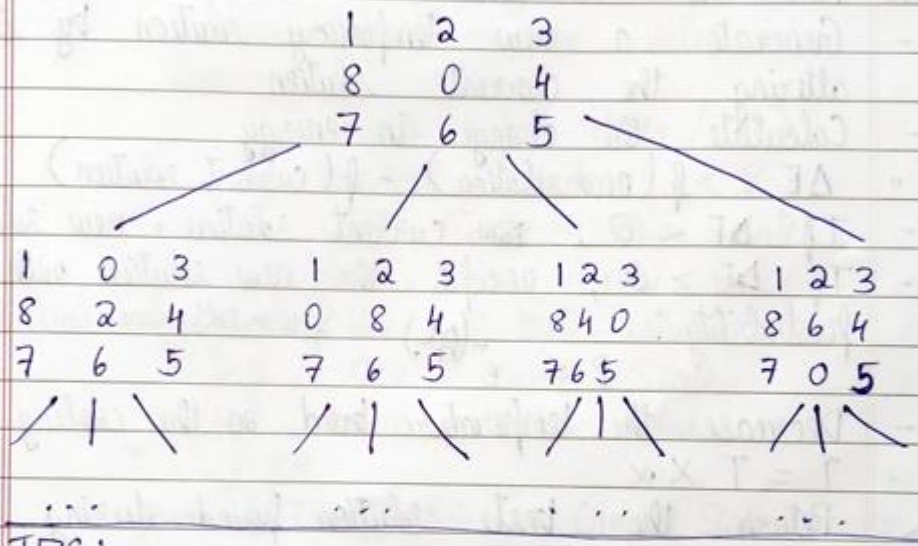
Algorithm [Iterative Deepening Search (IDS)]:

* Define a function to check if the target is reachable from the source within the maximum depth.
  - This function returns true if it is possible
  - False otherwise
* For "depth" iterations, call the check function which:
  - Returns true if source = target
  - false if limit has reached
* Define a boolean function which takes the source state, target state and the maximum depth as parameters.
  - This function iteratively calls the check function for "maximum depth" number of iterations.
  - It returns true if the target is reachable from the source within the depth, else false.
* The check function takes three parameters: source, target and limit.
  - Limit is the depth for that particular iteration.
  - This function returns true if source = target
  - It returns false if the limit goes below zero.
  - It recursively calls itself with the limit decreasing by one for each recursion
  - The check function returns true if the function it recursively calls is true.
  - If none of these conditions are met, it returns false

A* Algorithm for 8-Puzzle :

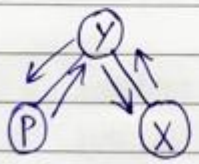
$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix} \quad \{ \text{Initial State} \}$$

$$\begin{bmatrix} 2 & 8 & 1 \\ 0 & 4 & 3 \\ 7 & 6 & 5 \end{bmatrix} \quad \{ \text{Target State} \}$$

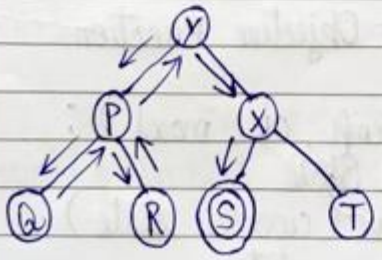


IDS :

depth = 0 — Y
return NULL



depth = 1 — Y → P → X
return NULL



depth = 2 — Y → P → Q → R → X → S
return True

A* algorithm

Code:

```
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

def get_input():
    print("Enter the initial state of the puzzle (3x3 matrix, each row separated by space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i + 1}: ").split()))
        initial_state.append(row)
    return initial_state

initial_state = get_input()
solution = a_star_misplaced_tiles(initial_state)
if solution:
    print("Solution found!")
else:
    print("No solution found.")
print("Nikhilesh 1BM22CS181")
```

Output:

```
Enter the initial state of the puzzle (3x3 matrix, each row separated by space):
Enter row 1: 1 2 0
Enter row 2: 3 4 5
Enter row 3: 6 7 8
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!
Nikhilesh 1BM22CS181
```

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("Nikhilesh 1bm22cs181")
```

Output: