DATE:27/06/2025

EXP:01

IDDFS

1. Which two classical uninformed search strategies does IDDFS combine the advantages of?

Depth-First Search (DFS) and **Breadth-First Search (BFS)**.

- From **DFS**, it takes **low memory usage**.
- From **BFS**, it takes **completeness** and **optimality** (for uniform step costs).

So IDDFS = DFS's space efficiency + BFS's completeness/optimality.

2. Why does IDDFS require only O(b·d) memory?

- At each depth-limited DFS, only the nodes **along the current path** ($\leq d$ deep) and their **siblings** are stored.
- Hence memory is proportional to **branching factor × depth**, i.e. $O(b \cdot d)$.
- Contrast this with BFS, which needs O(b^d) memory.

3. What assumption about step-costs must hold for IDDFS to guarantee an optimal (cheapest) solution?

All step costs must be equal (unit cost, e.g. cost = 1 per move).

- If costs vary, IDDFS may not return the cheapest path.
- For varying costs, we need **Uniform-Cost Search (UCS)** instead.

4. Give the big-O time complexity of IDDFS in terms of branching factor b and optimal solution depth d.

O(b^d)

- At each iteration, IDDFS does DFS up to depth i.
- So nodes near the top are re-generated many times.
- But total overhead is at most a constant factor compared to BFS.
- Therefore, time complexity = O(b^d).

5. Name one drawback of IDDFS compared with Breadth-First Search.

Repetition of nodes.

- IDDFS repeatedly expands the same shallow nodes in every iteration.
- This makes it slower in practice compared to BFS when branching factor b is large.

AIM:

Problem statement
It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state

Goal: Implement IDDFS to reach the goal state.

State Space:
Transition = You can move the tile Up, Down, Left and Right.

ALGORITHM:

Algorithm: Iterative Deepening Depth-First Search (IDDFS)

1. Start with depth limit = 0.
2. Repeat until solution is found or depth exceeds maximum allowed:
   a. Perform a Depth-Limited Search (DLS) from the root node, with the current depth limit.

b. If the goal node is found → return solution.
c. Otherwise, increase depth limit by 1 and repeat.
3. If maximum depth reached without finding the goal → return failure.

Depth-Limited Search (DLS) sub-algorithm

1. If the current node is the goal, return success.
2. If the current depth limit = 0, stop exploring this path.
3. Otherwise:
   a. Expand the current node.
   b. Recursively apply DLS to each child with depth limit − 1.
   c. If any child returns success → propagate success upward.

CODE:

```
from copy import deepcopy


# Define the initial and goal states

initial_state = [[1, 2, 3],

                 [5, 6, 0],   # 0 represents the blank space

                 [7, 8, 4]]


goal_state = [[1, 2, 3],

              [5, 8, 6],
```

```
              [0, 7, 4]]


# Define moves (Up, Down, Left, Right)
moves = {
    'Up': (-1, 0),
    'Down': (1, 0),
    'Left': (0, -1),
    'Right': (0, 1)
}


def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None


def is_goal(state):
    return state == goal_state


def get_possible_moves(state):
    row, col = find_blank(state)
    possible_states = []
```

```
    for move_name, (dr, dc) in moves.items():
        new_r, new_c = row + dr, col + dc
        if 0 <= new_r < 3 and 0 <= new_c < 3:
            new_state = deepcopy(state)
            # Swap blank with the adjacent tile
            new_state[row][col], new_state[new_r][new_c] = new_state[new_r][new_c],
new_state[row][col]
            possible_states.append((new_state, move_name))
    return possible_states


# Depth Limited DFS
def depth_limited_dfs(state, limit, path, visited):
    if is_goal(state):
        return path

    if limit <= 0:
        return None

    visited.append(state)

    for next_state, move in get_possible_moves(state):
        if next_state not in visited:
            result = depth_limited_dfs(next_state, limit - 1, path + [move], visited)
            if result is not None:
```

```
            return result

    return None

# Iterative Deepening Search
def iterative_deepening_dfs(start_state, max_depth):
    for depth in range(max_depth + 1):
        visited = []
        result = depth_limited_dfs(start_state, depth, [], visited)
        if result is not None:
            return result, depth
    return None, max_depth

# Solve using IDS
solution_path, depth = iterative_deepening_dfs(initial_state, 20)

# Output result
if solution_path:
    print(" Goal reached!")
    print("Moves to goal:", ' -> '.join(solution_path))
    print("Total steps:", len(solution_path))
    print("Depth reached:", depth)
else:
```

```
print(" Goal not found within depth limit.")
```

OUTPUT:

```
 Goal reached!
Moves to goal: Left -> Down -> Left
Total steps: 3
Depth reached: 3
```

RESULT:

The programs have been completed and the outputs have been verified.