Assignment: Solving the 15-Puzzle Using Backtracking and Dynamic Programming

Course Code: CSE221

1. Introduction

The **15-Puzzle** (also called the sliding puzzle) is a classic problem in algorithms and artificial intelligence. It consists of a 4×4 grid containing tiles numbered from 1 to 15 and one empty space. A move consists of sliding a tile into the empty space.

Goal State:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & \_ \end{bmatrix}$$

The objective of this assignment is to:

- Implement a naive solver using **Backtracking (DFS)**.
- Show its limitations on large state spaces.
- Implement a more efficient solver using **Dynamic Programming (Memoization)**.
- (Optional) Extend to BFS or A* for shortest path computation.

2. The Puzzle as a State-Space Problem

- **State:** A configuration of the 16 tiles.
- **Initial State:** Given as input.
- **Goal State:** Ordered tiles with the blank at the bottom-right.
- **Operators:** Move the blank tile {Up, Down, Left, Right}.
- **Search Tree:** Each node is a state; children are states reachable in one move.

3. Part A – Backtracking Approach

3.1 Idea

We use Depth-First Search (DFS) with backtracking:

1. Start from the initial configuration.
2. Explore moves recursively.
3. If a state repeats, backtrack.

Problem: The state space has size $16! \approx 2 \times 10^{13}$, making naive DFS infeasible.

3.2 Pseudocode

```
function BACKTRACK(state, visited):
    if state == GOAL:
        return True
    for move in possible_moves(state):
        new_state = apply_move(state, move)
        if new_state not in visited:
            add new_state to visited
            if BACKTRACK(new_state, visited):
                return True
            remove new_state from visited
    return False
```

4. Part B – Dynamic Programming Approach

4.1 Motivation

Backtracking recomputes the same states multiple times. Dynamic Programming (DP) with **memoization** stores results for already-solved states.

4.2 Pseudocode

```
function DP_SOLVE(state):
    if state == GOAL:
        return 0
    if state in memo:
        return memo[state]

    min_steps = infinity
    for move in possible_moves(state):
        new_state = apply_move(state, move)
        steps = DP_SOLVE(new_state)
        if steps != -1:
            min_steps = min(min_steps, 1 + steps)

    memo[state] = -1 if min_steps == infinity else min_steps
    return memo[state]
```

5. Python Implementation (Skeleton)

```
N = 4
GOAL = tuple(range(1, N*N)) + (0,)

def get_moves(pos):
    x, y = divmod(pos, N)
    moves = []
    if x > 0: moves.append(pos - N)
    if x < N-1: moves.append(pos + N)
    if y > 0: moves.append(pos - 1)
    if y < N-1: moves.append(pos + 1)
    return moves

def apply_move(state, blank, new_pos):
    state = list(state)
    state[blank], state[new_pos] = state[new_pos], state[blank]
    return tuple(state), new_pos
```

6. Assignment Tasks

Part A – Backtracking

(a) Implement the backtracking algorithm.

(b) Print whether the puzzle is solvable and the path taken.

(c) Analyze the branching factor and complexity.

Part B – Dynamic Programming

(a) Implement DP with memoization.

(b) Compare runtime with pure backtracking.

(c) Print the minimum number of steps to solve.

Part C – Extensions (Optional)

- Reconstruct the solution path.
- Implement BFS for shortest path.
- Implement A* with Manhattan distance heuristic.
- Compare all methods experimentally.

7. Deliverables

- **Code:** Python/Java/C++ implementation.
- **Report:** 2–3 pages explaining algorithms, complexity, results.
- **Experimental comparison:** Show runtimes on different scrambles.

8. Grading Rubric

Task	Marks
Backtracking implementation	20
DP implementation with memoization	20
Complexity analysis	10
Experimental comparison	20
Code readability & documentation	10
Extension (BFS / A*)	20 (bonus)