Scaler Companion beta

Software Engineering - DSA and Graphs Revision Notes

Introduction

This class covers Dynamic Programming (DP) and Graph theory, focusing on practical approaches and optimization techniques. The lecture pivoted around the use of efficient data structures and algorithms to handle common problems in these domains.

Dynamic Programming (DP)

Dynamic Programming Concepts

- **Idea**: Dynamic Programming is an optimization technique used to solve problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid duplicate computations .
- **Recursive DP (Top-Down Approach)**
  - **Method**: Solve larger problems by tackling smaller subproblems and storing results using a memoization table.
  - **Example**: Using a DP table (`dp`) to store interim results, reducing the need to recompute them .
- **Iterative DP (Bottom-Up Approach)**
  - **Method**: Fill up a DP table iteratively, starting from known values (base cases) and building up to the target problem.
  - **Space Optimization**: Instead of using a large DP array, space can be optimized by keeping track of only a limited number of state transitions at a time .

Example Problem - Minimum Cost Path

- **Recursive Approach:** Solve from the top with memoization to store results .
- **Iterative Approach:** Create a DP table and compute results from base cases upwards (bottom-up).
- The example discussed involved calculating minimal costs using an optimized iterative approach that reduces space usage by storing just the last two computed values .

Graph Theory

Graph Representations

- **Adjacency Matrix**
  - **Description:** A 2-D array representation where `matrix[i][j]` indicates the presence or weight of an edge from node `i` to `j` .
  - **Complexity:** Space complexity of O(N^2)【7:7†typed.md】.
- **Adjacency List**
  - **Description:** An array where each entry stores a list of nodes that are adjacent to it.
  - **Complexity:** Typically offers better space efficiency compared to an adjacency matrix, especially for sparse graphs【7:7†typed.md】.

Graph Traversal Algorithms

- **Depth First Search (DFS)**
  - **Pseudo Code:**

```
visited[N] = {False}

void dfs(source) {
    visited[source] = True;
    for (all u connected to source) {
        if (visited[u] == false) {
            dfs(u);
        }
```

  - **Use Case:** Suitable for exploring as far as possible along each branch before backtracking【7:7†typed.md】.
- **Breadth First Search (BFS)**
  - **Pseudo Code:**

```
void bfs(source) {
    Queue<int> q;
    q.enqueue(source);
    visited[source] = True;
    while (!q.isEmpty()) {
        int u = q.dequeue();
        for (all v connected to u) {
            if (!visited[v]) {
                q.enqueue(v);
                visited[v] = True;
            }
        }
    }
}
```

  - **Use Case:** Ideal for finding the shortest path in unweighted graphs 【7:7†typed.md】.

Example Problem - Graph Inputs

- When processing graph inputs:
  - **Input Format**: The number of nodes (n) and edges (m) is usually followed by a list of edges. Each edge specifies a vertex pair indicating a connection .

Conclusion

This session emphasized utilizing space and computational resources efficiently through dynamic programming and graph theory techniques. Practice assignments following the class aim to reinforce these concepts through implementation challenges .