Graphs Cheatsheet

Topic Overview

Graphs in Java represent relationships as nodes and edges, used for network and path problems. This cheatsheet covers graph-based techniques.

Prerequisites

Trees

List of Subtopics

- Adjacency Matrix
- Adjacency List
- BFS (Breadth-First Search)
- DFS (Depth-First Search)
- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- Kruskal's Algorithm
- Prim's Algorithm
- Topological Sort
- Detect Cycle

Key Concepts Explained

- **Adjacency Matrix**: 2D array representing edges, space-intensive.
- **Adjacency List**: Array of lists, space-efficient for sparse graphs.
- **Dijkstra's Algorithm**: Finds shortest paths in weighted graphs.

Approaches to Solve Problems with Step-by-Step Algorithms

- **Adjacency Matrix**:
  - **Algorithm**:
    1. Create a VxV matrix, initialize with infinity.
    2. Set 1 or weight for edges, 0 for no edge.
    3. Access edge with matrix[u][v].
  - **Context**: O(Vš) space, O(1) edge check.
- **Adjacency List**:
  - **Algorithm**:
    1. Use array of lists, each index for a vertex.
    2. Add edges to the list of the source vertex.
    3. Traverse list for neighbors.
  - **Context**: O(V+E) space, O(degree) edge check.
- **BFS (Breadth-First Search)**:
  - **Algorithm**:
    1. Use a queue, enqueue start node.
    2. Dequeue, process, enqueue unvisited neighbors.
    3. Mark visited, repeat until queue is empty.
  - **Context**: O(V+E) time, O(V) space.
- **DFS (Depth-First Search)**:
  - **Algorithm**:
    1. Use a stack or recursion, start with root.
    2. Visit node, recurse on unvisited neighbors.
    3. Backtrack when no unvisited neighbors.
  - **Context**: O(V+E) time, O(V) space.
- **Dijkstra's Algorithm**:
  - **Algorithm**:
    1. Use min heap for distances, initialize to infinity.
    2. Start from source, update distances to neighbors.
    3. Repeat until all nodes processed.
  - **Context**: O((V+E) log V) time, O(V) space.
- **Bellman-Ford Algorithm**:
  - **Algorithm**:

    1. Initialize distances to infinity, source to 0.
    2. Relax all edges V-1 times.
    3. Check for negative cycles with extra relaxation.
  - **Context**: O(VE) time, O(V) space.

- **Kruskal's Algorithm**:
  - **Algorithm**:
    1. Sort edges by weight.
    2. Use union-find to add edges without cycles.
  - **Context**: O(E log E) time, O(V) space.

- **Prim's Algorithm**:
  - **Algorithm**:
    1. Use min heap for edges, start with any node.
    2. Add minimum edge to unvisited node, update heap.
  - **Context**: O((V+E) log V) time, O(V) space.

- **Topological Sort**:
  - **Algorithm**:
    1. Use DFS, store finish times in a stack.
    2. Pop stack to get topological order.
  - **Context**: O(V+E) time, O(V) space.

- **Detect Cycle**:
  - **Algorithm**:
    1. Use DFS with visited and recursion stack.
    2. If node in recursion stack, cycle exists.
  - **Context**: O(V+E) time, O(V) space.

Common LeetCode Problems with Approaches

- **Clone Graph (133)**: Use DFS or BFS with a map for cloning.
- **Network Delay Time (743)**: Use Dijkstra's for shortest paths.
- **Number of Islands (200)**: Use DFS or BFS for counting.
-
- **Course Schedule (207)**: Use topological sort for prerequisites.

Time & Space Complexities

- BFS/DFS: O(V+E)
- Dijkstra's: O((V+E) log V)
- Space: O(V) or O(V+E)

Important Tips & Tricks

- Use adjacency lists for sparse graphs.
- Optimize Dijkstra's with binary heap.
- Handle disconnected components in DFS/BFS.
- Use union-find for cycle detection in MST.
- Preprocess weights for efficiency.