Graph Problems: Shortcut Tricks and Notes

# Common Graph Representations

1. Adjacency Matrix: A 2D array where matrix[i][j] is 1 (or the weight) if there is an edge from vertex i to vertex j, otherwise 0.

2. Adjacency List: An array of lists. Each index i contains a list of vertices adjacent to vertex i.

# Basic Graph Traversal

1. Depth-First Search (DFS): Explores as far as possible along each branch before backtracking.

- Use for detecting cycles, topological sorting, finding connected components.

2. Breadth-First Search (BFS): Explores all neighbors at the present depth before moving to the next depth level.

- Use for finding the shortest path in unweighted graphs, level-order traversal.

# Common Graph Algorithms

1. Dijkstra’s Algorithm: Finds the shortest path from a source to all other vertices in a weighted graph with non-negative weights.

2. Bellman-Ford Algorithm: Finds the shortest path from a source to all other vertices in a weighted graph, including those with negative weights. Can detect negative cycles.

3. Floyd-Warshall Algorithm: Finds the shortest paths between all pairs of vertices. Useful for dense graphs.

4. Kruskal’s Algorithm: Finds the Minimum Spanning Tree (MST) using a greedy approach.

5. Prim’s Algorithm: Another algorithm to find the MST, also using a greedy approach.

# Important Patterns and Tricks

1. Cycle Detection:

- Undirected Graph: Use DFS or Union-Find.

- Directed Graph: Use DFS with recursion stack or Kahn’s algorithm for topological sorting.

2. Topological Sorting:

- Applicable only to Directed Acyclic Graphs (DAGs). Use DFS or Kahn’s Algorithm.

3. Connected Components:

- Use DFS or BFS to find all components in an undirected graph.

- For strongly connected components in a directed graph, use Kosaraju’s or Tarjan’s algorithm.

4. Bipartite Check:

- Use BFS or DFS to try to color the graph with two colors. If you can, it’s bipartite.

5. Shortest Path in Unweighted Graph:

- Use BFS starting from the source vertex.

6. Graph Coloring:

- Use backtracking or greedy algorithms depending on constraints.

# Advanced Techniques

1. Dynamic Programming on Graphs:

- Use for problems involving paths, like the longest path in a DAG.

2. Graph Matching:

- Use algorithms like Hopcroft-Karp for finding maximum matching in bipartite graphs.

3. Network Flow:

- Use the Ford-Fulkerson method or Edmonds-Karp algorithm to solve maximum flow problems.

4. Tree Algorithms:

- LCA (Lowest Common Ancestor): Binary lifting, RMQ (Range Minimum Query).

- Diameter of a Tree: Two-pass BFS or DFS.

# Problem-Solving Patterns

1. Reduction to Known Problems: Reduce the problem to a well-known graph problem (e.g., finding the shortest path, MST, etc.).

2. Graph Transformations: Transform the graph (e.g., creating an auxiliary graph) to simplify the problem.

3. Divide and Conquer: Use this approach for tree problems where you can solve subproblems independently.

4. Iterative Deepening: Combine depth-first search and breadth-first search, often useful in pathfinding problems.

# Practice and Familiarity

1. Familiarize yourself with common graph problems and their solutions.

2. Practice problems on platforms like LeetCode, Codeforces, and GeeksforGeeks.

3. Review graph theory concepts regularly to keep them fresh in your mind.