📘 Comprehensive Summary: Graph Problems and Solutions - 3

---

1. Cycle Detection in Directed Graph

Approach 1: DFS (using recursion stack)

- Maintain two arrays: `visited[]` and `pathVisited[]`.
- Recurse through DFS; if a node is visited and also in the recursion stack (`pathVisited`), then a cycle exists.

Approach 2: Kahn's Algorithm (Topological Sort)

- Count in-degrees.
- Push nodes with in-degree = 0 into the queue.
- If the resulting topological sort has fewer nodes than `V`, a cycle exists.

```
class Solution {
public:
    bool dfs(int node, vector<vector<int>>& adj, vector<bool>& visited, vector<bool
        visited[node] = true;
        pathVisited[node] = true;

        for (int ngbr : adj[node]) {
            if (!visited[ngbr]) {
                if (dfs(ngbr, adj, visited, pathVisited)) return true;
            } else if (pathVisited[ngbr]) {
                return true;
            }
        }

        pathVisited[node] = false;
        return false;
    }

    bool isCyclic(int V, vector<vector<int>>& adj) {
        vector<bool> visited(V, false), pathVisited(V, false);
        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                if (dfs(i, adj, visited, pathVisited)) return true;
            }
        }
        return false;
    }
};
```

2. Topological Sort

DFS-based:

- Perform DFS and push nodes onto a stack after exploring all neighbours.
- Reverse stack for topological order.

```
void dfs(int node, vector<int> adj[], vector<bool>& visited, stack<int>& st){
    visited[node] = true;
    for(int ngbr : adj[node]){
        if(!visited[ngbr]){
            dfs(ngbr, adj, visited, st);
        }
    }
    st.push(node);
}
```

Kahn's Algorithm (BFS-based):

```
vector<int> topoSort(int V, vector<vector<int>>& edges) {
    vector<int> adj[V], indegree(V, 0);
    for(auto edge : edges){
        adj[edge[0]].push_back(edge[1]);
        indegree[edge[1]]++;
    }

    queue<int> q;
    for(int i = 0; i < V; i++) if(indegree[i] == 0) q.push(i);

    vector<int> result;
    while(!q.empty()){
        int node = q.front(); q.pop();
        result.push_back(node);
        for(int ngbr : adj[node]){
            if(--indegree[ngbr] == 0) q.push(ngbr);
        }
    }
    return result;
}
```

3. Course Schedule I & II

canFinish:

```
bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
    vector<int> adj[numCourses], indegree(numCourses, 0);
    for(auto pre : prerequisites){
        adj[pre[1]].push_back(pre[0]);
        indegree[pre[0]]++;
    }
    queue<int> q;
    for(int i = 0; i < numCourses; i++) if(indegree[i] == 0) q.push(i);

    int count = 0;
    while(!q.empty()){
        int node = q.front(); q.pop(); count++;
        for(int ngbr : adj[node]){
            if(--indegree[ngbr] == 0) q.push(ngbr);
        }
    }
    return count == numCourses;
}
```

findOrder:

- Same as Kahn's algorithm but store nodes in result.
- If cycle, return empty vector.

---

4. Alien Dictionary

Build char graph, apply topological sort

```
string findOrder(vector<string> &words) {
    vector<vector<int>> adj(26);
    vector<int> indegree(26, 0);
    vector<bool> used(26, false);

    for(int i = 0; i < words.size(); i++)
        for(char c : words[i]) used[c - 'a'] = true;

    for(int i = 0; i < words.size()-1; i++) {
        string s1 = words[i], s2 = words[i+1];
        for(int j = 0; j < min(s1.length(), s2.length()); j++) {
            if(s1[j] != s2[j]) {
                adj[s1[j]-'a'].push_back(s2[j]-'a');
                indegree[s2[j]-'a']++;
                break;
            }
        }
    }

    queue<int> q;
    for(int i = 0; i < 26; i++) if(used[i] && indegree[i] == 0) q.push(i);
```

```
    for(int i = 0; i < 26; i++) if(used[i] && indegree[i] == 0) q.push(i);

    string result;
    while(!q.empty()) {
        int node = q.front(); q.pop();
        result += (char)(node + 'a');
        for(int ngbr : adj[node]) {
            if(--indegree[ngbr] == 0) q.push(ngbr);
        }
    }
    return result.size() == count(used.begin(), used.end(), true) ? result : "";
}
```

---

5. **Shortest Path in DAG**

- Do a topological sort.
- Then relax edges in topological order.

```
void dfs(int node, vector<pair<int,int>> adj[], vector<bool>& visited, stack<int>&
    visited[node] = true;
    for(auto [v, wt] : adj[node]){
        if(!visited[v]) dfs(v, adj, visited, st);
    }
    st.push(node);
}

vector<int> shortestPath(int V, vector<vector<int>>& edges) {
    vector<pair<int,int>> adj[V];
    for(auto e : edges) adj[e[0]].push_back({e[1], e[2]});

    vector<bool> visited(V, false);
    stack<int> st;
    for(int i = 0; i < V; i++) if(!visited[i]) dfs(i, adj, visited, st);

    vector<int> dist(V, 1e9); dist[0] = 0;
    while(!st.empty()) {
        int u = st.top(); st.pop();
        for(auto [v, w] : adj[u])
            if(dist[u] + w < dist[v]) dist[v] = dist[u] + w;
    }
    return dist;
}
```

---

6. **Shortest Path in Unweighted Graph** (BFS)

```
vector<int> shortestPath(vector<vector<int>>& adj, int src) {
    int V = adj.size();
    vector<int> dist(V, INT_MAX);
    queue<int> q;
    dist[src] = 0;
    q.push(src);

    while(!q.empty()) {
        int u = q.front(); q.pop();
        for(int v : adj[u]) {
            if(dist[v] > dist[u] + 1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
    return dist;
}
```

---

7. **Dijkstra's Algorithm** (Weighted Undirected Graph)

```
vector<int> dijkstra(int V, vector<vector<pair<int,int>>>& adj, int src) {
    vector<int> dist(V, INT_MAX);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> pq;
    dist[src] = 0;
    pq.push({0, src});

    while(!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        for(auto [v, wt] : adj[u]) {
            if(dist[v] > d + wt) {
                dist[v] = d + wt;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

---

8. **Check if Graph is Bipartite**

- Color nodes with two colors using BFS/DFS.
- If a neighbor has the same color, it's not bipartite.

```
class Solution {
```

```
class Solution {
public:
    bool isBipartite(vector<vector<int>>& graph) {
        int n = graph.size();
        vector<int> color(n, -1);

        for (int i = 0; i < n; i++) {
            if (color[i] == -1) {
                queue<int> q;
                q.push(i);
                color[i] = 0;

                while (!q.empty()) {
                    int node = q.front(); q.pop();
                    for (int ngbr : graph[node]) {
                        if (color[ngbr] == -1) {
                            color[ngbr] = 1 - color[node];
                            q.push(ngbr);
                        } else if (color[ngbr] == color[node]) {
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    }
};
```

---

9. Eventual Safe Nodes

- Reverse the graph.
- Run topological sort using Kahn's algorithm.
- Nodes with 0 in-degree in reverse graph are safe.

```
class Solution {
    bool dfs(int node, vector<vector<int>>& adj, vector<bool>& visited,
             vector<bool>& pathvis, vector<bool>& check) {
        visited[node] = true;
        pathvis[node] = true;

        for (int ngbr : adj[node]) {
            if (!visited[ngbr]) {
                if (dfs(ngbr, adj, visited, pathvis, check)) {
                    check[node] = false;
                    return true;
                }
            } else if (pathvis[ngbr]) {
                check[node] = false;
```

```
            return true;
        }
    }

    check[node] = true;
    pathvis[node] = false;
    return false;
}
public:
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
        int n = graph.size();
        vector<bool> visited(n, false), pathvis(n, false), check(n, false);
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                dfs(i, graph, visited, pathvis, check);
            }
        }
        vector<int> safe;
        for (int i = 0; i < n; i++) {
            if (check[i]) safe.push_back(i);
        }
        return safe;
    }
};
```

- Reverse graph + Kahn's Algorithm

```
class Solution {
public:
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
        int V = graph.size();
        vector<vector<int>> revGraph(V);
        vector<int> indegree(V, 0);

        // Reverse the graph and compute indegrees
        for(int u = 0; u < V; u++){
            for(int v : graph[u]){
                revGraph[v].push_back(u); // reverse edge
                indegree[u]++; // original outdegree becomes indegree
            }
        }

        queue<int> q;
        // All terminal nodes (original indegree 0)
        for(int i = 0; i < V; i++){
            if(indegree[i] == 0){
                q.push(i);
            }
        }
```

```cpp
        vector<bool> safe(V, false);
        while(!q.empty()){
            int node = q.front(); q.pop();
            safe[node] = true;
            for(int prev : revGraph[node]){
                indegree[prev]--;
                if(indegree[prev] == 0){
                    q.push(prev);
                }
            }
        }

        vector<int> result;
        for(int i = 0; i < V; i++){
            if(safe[i]) result.push_back(i);
        }

        return result;
    }
};
```

Let me know if you want visual diagrams or dry runs for any of the above.