Unweighted Graph Algorithms: DFS, BFS and Topological Sort

Definition
A graph is a collection of dots, called vertices, and connections between those dots, called edges. There are two categories of adjectives to describe different types of graphs:

unweighted vs. weighted
undirected vs. directed

In a weighted graph, each connection between vertices has an associated number, called an "edge weight". In an undirected graph, no such number is associated and by default, we typically assign 1.

In a directed graph, the order of the two vertices in a connection matters. Thus, in a directed graph, an edge from vertex a to vertex b does not imply an edge from vertex b to vertex a. In an undirected graph, no order is given to the two vertices that are connected, so if vertex a and b are connected via an edge, one can go from a to b, OR b to a.

A multi-graph allows for more than one edge between the same two vertices. These are relatively rare in contests, however, when they appear, they are rather tricky, because many algorithms that assume only one connection between any pair of vertices tends to fail when this assumption isn't true.

How to Store a Graph
The easiest way to store a graph is a two dimensional integer array of size n x n, where n is the number of vertices in the graph:

```
int[][] adjmat = new int[n][n];
```

Typically, adjmat[i][j] would store the edge weight for the edge from vertex i to vertex j. If it's an unweighted graph, we store 1 if the edge exists. If no such edge exists, we can either store a large integer or 0 and code accordingly. Alternatively, we can store null and make it an array of type Integer (in Java).

This storage method is great for when you are first learning about graphs. It's often inefficient though, especially for sparse graphs, where a vast majority of possible edges don't exist. (Imagine a graph with 10^5 vertices and $3x10^5$ edges!!!)

The best way to store a graph for contests is an array of lists:

```
ArrayList[] adjList = new ArrayList[n];
for (int i=0; i<n; i++)
    adjList[i] = new ArrayList<Integer>();
```

adjList[i] would be a list storing all vertices, vertex i is connected to, filled in later.

Graph Traversal - Depth First Search
The goal of a graph traversal is simply to mark all vertices that can be visited, following edges from a particular vertex.

The general "rule" used in searching a graph using a depth first search is to search down a path from a particular source vertex as far as you can go. When you can go to farther, "backtrack" to the last vertex from which a different path could have been taken. Continue in this fashion, attempting to go as deep as possible down each path until each node has been visited. Here is some code for DFS assuming the more efficient graph storage - it just marks

```
public static void dfs(ArrayList[] graph, boolean[] visited,
int v) {

    visited[v] = true;
    for (Integer next : ((ArrayList<Integer>)graph)[v])
        if (!visited[next])
            dfs(graph, visited, next);
}
```

The running time of DFS is O(V+E). To see this, note that each edge and vertex is visited at most twice. In order to get this efficiency, an adjacency list must be used. (An adjacency matrix can not be used to complete this algorithm that quickly.)

Graph Traversal - Breadth First Search
The idea in a breadth first search is opposite to a depth first search. Instead of searching down a single path until you can go no longer, you search all paths at a uniform depth from the source before moving onto deeper paths. Once again, we'll need to mark both edges and vertices based on what has been visited.

In essence, we only want to explore one "unit" away from a searched node before we move to a different node to search from. All in all, we will be adding nodes to the back of a queue to be ones to searched from in the future. Thus, we start with our source vertex in the queue and then whenever we dequeue an item, we enqueue all of its "new" neighbors who are all one unit away, so the queue stores all items of distance 1 from the source before all items who are distance 2 from the source, and so forth.

The code on the following page runs a bfs from vertex v, marking the distance to all vertices from v (on an unweighted graph). It returns an array with these distances and a -1 to indicate unreachable vertices.

```
public static int[] bfs(ArrayList[] graph, int v) {

    int n = graph.length;
    int[] distance = new distance[n];
    Arrays.fill(distance, -1);
    visited[v] = true;
    ArrayDeque<Integer> q = new ArrayDeque<Integer>();
    q.offer(v);

    while (q.size() > 0) {
        int cur = q.poll();
        for (Integer next : ((ArrayList<Integer>)graph)[cur]) {
            if (distance[next] == -1) {
                distance[next] = distance[cur]+1;
                q.offer(next);
            }
        }
    }

    return distance;
}
```

Basically, we need two data structures: an array that keeps track of where we've been (and how far away those vertices are) AND the queue to keep track of the locations from which we still need to explore. When we dequeue, we basically just add all relevant (previously unvisited vertices) vertices to our queue. Note that as soon as we do this, we MUST mark these new vertices as visited. We can't wait until we dequeue them to do so, can you see why?

Topological Sort
The goal of a topological sort is given a list of items with dependencies, (ie. item 5 must be completed before item 3, etc.) to produce an ordering of the items that satisfies the given constraints. In order for the problem to be solvable, there can not be a cyclic set of constraints. (We can't have that item 5 must be completed before item 3, item 3 must be completed before item 7, and item 7 must be completed before item 5, since that would be an impossible set of constraints to satisfy.)

We can model a situation like this using a directed acyclic graph. Given a set of items and constraints, we create the corresponding graph as follows:

1) Each item corresponds to a vertex in the graph.
2) For each constraint where item a must finish before item b, place a directed edge in the graph starting from the vertex for item a to the vertex for item b.

This graph is directed because each edge specifically starts from one vertex and goes to another. Given the fact that the constraints must be acyclic, the resulting graph will be as well.

Here is a simple situation:

A → B (Imagine A standing for waking up,
| | B standing for taking a shower,
V   V   C standing for eating breakfast, and
C → D   D leaving for work.)

Here a topological sort would label A with 1, B and C with 2 and 3, and D with 4.

Let's consider the following subset of CS classes and a list of prerequisites:

CS classes: COP 3223, COP 3502, COP 3330, COT 3100, COP 3503, CDA 3103, COT 3960 (Foundation Exam), COP 3402, and COT 4210.

Here are a set of prerequisites:

COP 3223 must be taken before COP 3330 and COP 3502.
COP 3330 must be taken before COP 3503.
COP 3502 must be taken before COT 3960, COP 3503, CDA 3103.
COT 3100 must be taken before COT 3960.
COT 3960 must be taken before COP 3402 and COT 4210.
COP 3503 must be taken before COT 4210.

A goal of a topological sort then is to find an ordering of these classes that you can take.

Topological Sort – Iterative Version

Just as there is always a vertex in a directed acyclic graph (DAG) that has no outgoing edges, there must ALSO be a vertex in a DAG that has no incoming edges. This vertex corresponds to one that is safe to put in the *front* of the topological sort, since it has no prerequisites.

Thus, the algorithm is as follows for a graph, G, with n vertices:

1. Initialize TOP to be an empty list

2. Using the graph edges, for each vertex calculate its in-degree and store these in an array indegree.

3. Create a list of next items. Add to next each vertex with in degree 0.

4. While TOP has fewer than n items:

   a. Take the next item, v, in the list next. If next is empty, there is no top sort.
   b. Add v to TOP.
   c. Remove all edges (v to w) in G.
      i. subtract 1 from the indegree of w.
     ii. if the indegree[w] == 0, then add w to the list next.

The run time of this algorithm is O(V+E), since each vertex and edge are processed a constant number of times.

Let's apply this algorithm to our class constraints:

COP 3223 must be taken before COP 3330 and COP 3502.
COP 3330 must be taken before COP 3503.
COP 3502 must be taken before COT 3960, COP 3503, CDA 3103.
COT 3100 must be taken before COT 3960.
COT 3960 must be taken before COP 3402 and COT 4210.
COP 3503 must be taken before COT 4210.

COP 3223 goes first, since it has no pre-requisites.
COP 3502 goes next, since it has no pre-requisites left.
COP 3330 goes next, since it has no pre-requisites left.
COT 3100 goes next, since it has no pre-requisites left.
COP 3503 goes next, since it has no pre-requisites left.
COT 3960 goes next, since it has no prerequisites left.
COP 3402 goes next, followed by
COT 4210.