Testing graph connectivity

Storing graphs

There are two basic ways to store a graph for computational purposes. Let G be an n vertex graph. Let $V(G)$ denote its vertex set and $E(G)$ denote its edge set.

Adjacency matrix: in this method one stores the $n \times n$ matrix $A = A(G)$, where

$$A_{ij} = \begin{cases} 1 & \text{if} \quad \overline{ij} \in E(G) \\ 0 & \text{if} \quad \overline{ij} \notin E(G) \end{cases}$$

A *query* about a graph stored in such manner is a question about the value of a specific entry of the matrix A. For example, you can ask "Are nodes 5 and 17 connected?".

This type of storage always takes up $\sim n^2$ space.

Edge lists: means that one stores for each vertex $v \in V(G)$ a list of the indices of its neighbors. The details can vary. In the simplest implementation you store an array N_i for each $i \in V(G)$. The elements of N_i are the neighbors of i in some order. So if vertex 17 is connected to vertices $1, 5, 16, 20, 25$ then

$$N_{17} = [1, 5, 16, 20, 25]$$

You can store some extra information, for example you can use a separate vector to store the degrees of all vertices.

This type of storage takes up $\sim \max(n, e)$ space where $n = |V(G)|$ and $e = |E(G)|$.

Breadth First Search (BFS)

This is a method that constructs a spanning tree in each component of the graph. Informally you are exploring a dark castle in such a way that in every step you are in a room, you peek through all the doors and switch the light on in each adjacent room. Once you finished with a single room, you move on to another room where the light is already on.

A little bit less informally, this is what happens in one step. You already discovered some of the vertices. For each discovered vertex v, you keep track of how you got there first (i.e. you store the name of the vertex from which you discovered v). Some of the discovered vertices have already been "checked". Others are waiting to be "checked". Let D denote the set of discovered vertices, and W the set of waiting vertices (so $W \subset D$).

Pick a vertex $w \in W$. Look at all its neighbors one-by-one, and if they are not yet in D, then put them into D and also into W. Finally remove w from W. Also record that their "parent" is w. (This way you actually get a spanning tree, not just the component.)

Start the whole algorithm with $D = W = \{x\}$ where x is the vertex whose component you want to find. When W becomes empty, the vertices in D form a connected component.

If the size of the graph is bigger than all the components you found so far together, that indicates that there are more components. Find a vertex that has not been considered yet, and start over from there.

Corollary: If the graph is stored as edge lists, this algorithm tests connectivity (and finds all components) in $O(\max(n, e))$ time.

Connectivity testing with adjacency matrix - the Evil Elf (a.k.a. the Adversary)

Theorem. *If the graph is stored as an adjacency matrix, then there is no algorithm to test connectivity whose running time would be less than $O(n^2)$. In particular, any algorithm has to query every single field of the matrix in the worst case.*

Proof. We introduce the Evil Elf, an answering scheme. For any algorithm $\mathcal{A}$ there is a graph G on which the algorithm asks about every pair of vertices before being able to conclude whether G is connected. The Evil Elf constructs G depending on the queries of the algorithm.

The Evil Elf scheme is as follows:

- For any query, EE checks whether a "No" answer would permanently disconnect the graph and thus allow the algorithm to finish immediately.
- If there is no such danger, EE answers "No".
- Otherwise EE answers "Yes".

We can then make the following observations:

1. The algorithm will never be able to say "disconnected", so the final graph G is going to be connected.
2. The EE scheme will never build a cycle into G: there would be no reason to answer "Yes" for the final edge in a cycle.
3. Hence G will be a tree.

Assume for a contradiction that the algorithm is able to say "connected" before asking every pair of vertices. That means, there is a pair x, y that this pair was never queried, but EE said yes to all the edges in G. There is a unique path between x and y in G (since G is a tree). Let this path be $x = v_0, v_1, \ldots, v_k = y$. Let $v_i v_{i+1}$ be the last edge that the algorithm asked among the edges of this path. So at the moment when the algorithm asks about the $v_i v_{i+1}$ edge, it has already confirmed the existence of all other edges in this path. But not the xy edge.

Claim: at this point there is no reason for EE to answer yes to the $v_i v_{i+1}$ query.

Proof: saying "No" to this query cannot disconnect the graph permanently, since if you swap the xy edge for the $v_i v_{i+1}$ edge in the tree, it remains a tree. So EE could still keep everything connected even if it answered "No" to $v_i v_{i+1}$.

So the assumption that $\mathcal{A}$ never queried the xy edge led to a contradiction. Hence $\mathcal{A}$ indeed had to query every single pair before concluding anything. □