CPSC 320 Notes, Playing with Graphs!

September 23, 2016

Today we practice reasoning about graphs by playing with two new terms. These terms/concepts are useful in themselves but not tremendously so; they're mainly a tool to spur our graph reasoning.

1 Terms

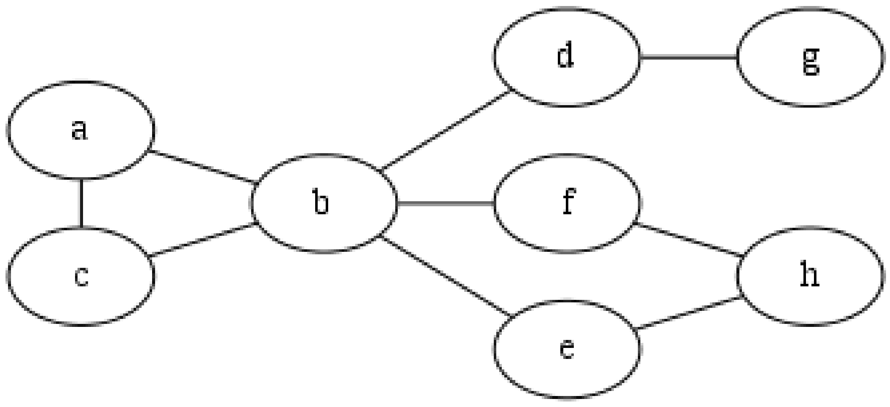
An *articulation point* in an undirected graph is a vertex whose removal increases the number of connected components in the graph.

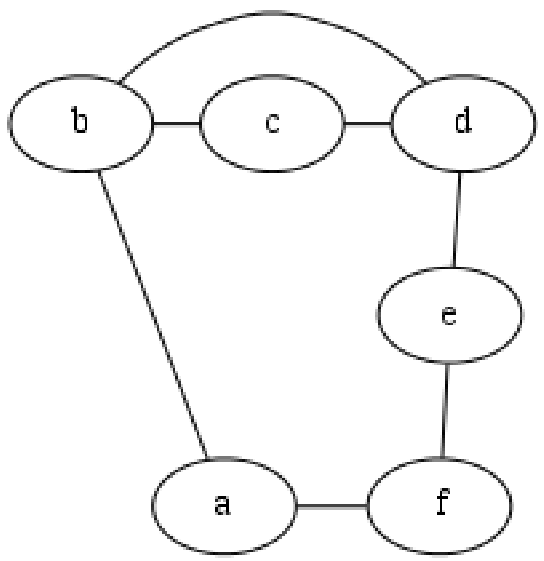
The *diameter* of an undirected, unweighted graph is the largest possible value of the following quantity: the smallest number of edges on any path between two nodes. In other words, it's the largest number of steps **required** to get between two nodes in the graph.

2 Play

For each of the following graphs:

1. Find all the articulation points (if any) in the graph.
2. Give the diameter of the graph.
3. Draw out the rooted tree generated by a breadth-first search of the graph from node a (draw dashed lines for edges that aren't part of the tree).
4. Draw out the rooted tree generated by a depth-first search of the graph from node a (with the same use for dashed lines).





3 Diameter Algorithm

Design an algorithm to find the **diameter** of an unweighted, undirected graph. For short, we'll call this problem DIAM.

3.1 Trivial and Small Instances

1. Write down all the **trivial** instances of DIAM.

2. Write down one more **small** instance of DIAM. (Smaller than the ones above but non-trivial.)

3. Hold this space for another instance, in case we need more.

3.2 Represent the Problem

1. The input to this problem is an unweighted, undirected graph. Use names to express what such a graph looks like as input.

2. Go back up and rewrite one trivial and one small instance using these names.

3. Our sketched representation of graphs is great! Always keep thinking about others as you solve a problem. (E.g., for n vertices, you might draw the adjacency matrix as an $n \times n$ grid with X's where there is an edge.)

4. Our input graph has some constraints. If needed, use the names above to express that: a node may not have an edge to itself, and an edge between two nodes may only appear once.

3.3 Represent the Solution

1. What are the quantities that matter in the solution to the problem? Give them short, usable names. (You may find yourself returning to this step later!)

2. Describe using these quantities makes a solution **valid** and **good**:

3. Go back up to your new trivial and small instances and write out one or more solutions to each using these names.

4. Go back up to your drawn representations of instances and draw at least one more solution.

3.4 Similar Problems

We've already suggested some related **algorithms** (breadth-first and depth-first search). Spend 3 minutes trying to brainstorm at least one more similar problem. It's OK if you cannot think of one!

3.5 Brute Force?

In this case, the solution (the diameter of the graph, an integer) doesn't have a very interesting form. But a closely related question ("which two nodes define the diameter by being farthest apart?") **does** have an interesting form. It's common that we'll have to make this kind of shift in focus when we try to make a brute force algorithm. (E.g., if I pose the problem "does a graph have a cycle?", the solution is either "yes" or "no", but in our brute force, we might consider the related problem "find a cycle in a graph" or in more detail "find a list of nodes in the graph that forms a cycle".)

For this part, consider the problem "which two nodes define the diameter by being farthest apart?"

1. Sketch an algorithm to produce everything with the "form" of a solution. (It will help to **give a name** to your algorithm and its parameters, *especially* if your algorithm is recursive.)

2. Choose an appropriate variable (or variables!) to represent the "size" of an instance.

3. Exactly or asymptotically, how many such "solution forms" are there? (It will help to **give a name** to the number of solutions as a function of instance size.)

4. In this problem, you'll likely keep track of the best candidate solution you've found so far as you work through brute force. What will characterize how good a possible solution is?

5. Given a possible solution, how can you determine how good it is?

6. Will brute force be sufficient for this problem for the domains we're interested in? (Since I didn't give you a domain, pick one!)

3.6 Lower-Bound

In terms of instance size, exactly or asymptotically, how "big" is an instance? (That is, how long will it take for an algorithm just to read the input to the problem?)

(We won't do more for now to lower-bound the problem.)

3.7 Promising Approach

You can do better than the naïve brute force approach we came up with by tweaking that brute force approach. Describe—in as much detail as you can—an approach that looks promising to adjust brute force and improve its asymptotic performance.

3.8 Challenge Your Approach

1. **Carefully** run your algorithm on your instances above. (Don't skip steps or make assumptions; you're debugging!) Analyse its correctness and performance on these instances:

2. Design an instance that specifically challenges the correctness (or performance) of your algorithm:

3.9 Repeat!

Hopefully, we've already bounced back and forth between these steps in today's worksheet! You usually *will* have to. Especially repeat the steps where you generate instances and challenge your approach(es).

4 Challenge Problem

1. Design an algorithm to find **articulation points** in an undirected graph. Hint: think about how the dashed edges in a DFS relate to articulation points. It may be easiest to start with the root, which is a simpler special case than the other nodes.

2. The book described the "Independent Set" problem as one where we do not know a polynomial-time solution. If we had a linear time algorithm for finding articulation points, describe how it could help (and how **much** it could help) for **some** graphs.