CPSC 320 Notes, Playing with Graphs

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

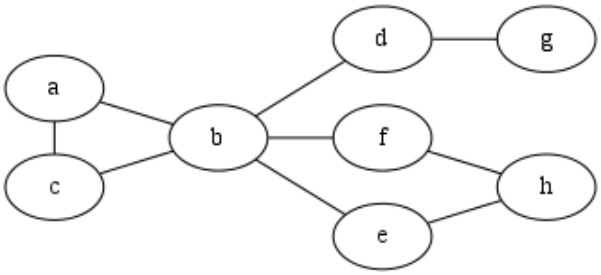
An *articulation point* in an undirected graph is a vertex whose removal increases the number of connected components in the graph.

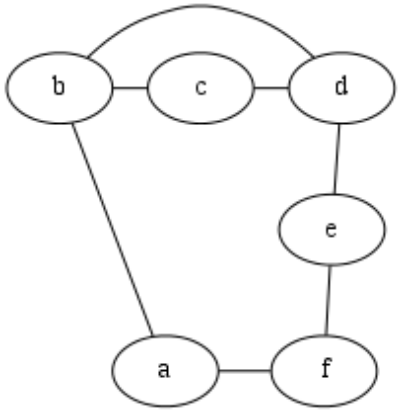
The *diameter* of an undirected, unweighted graph is the largest possible value of the following quantity: the smallest number of edges on any path between two nodes. In other words, it's the largest number of steps **required** to get between two nodes in the graph.

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, **connected** 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 in Section 2 but non-trivial.)

3.2 Represent the Problem

1. An instance of 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 visually easy to interpret. But always keep thinking about other representations 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. Using the names above, express the constraints that (i) a node may not have an edge to itself, and (ii) 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 valid solutions to each using these names.

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

3.4 Similar Problems

Think of related algorithms and/or problems.

3.5 Brute Force?

The solution to the problem is an integer, the diameter of the graph. That's not very useful for planning a brute force approach. A closely related quantity is "the two nodes that define the diameter by being farthest apart". (We often make this kind of shift in focus when we design a brute force algorithm.)

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.

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?

4. You will want to 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 we didn't give you a domain, pick one!)

3.6 Promising Approach

You can "tweak" the naive brute force approach to make it better. Describe—in as much detail as you can—a promising "tweak" to the approach that improves its asymptotic performance.

3.7 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:

3.8 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).