CPSC 320 Sample Solution, Playing with Graphs!

January 23, 2018

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

SOLUTION: All solutions appear just below the graphs.

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).
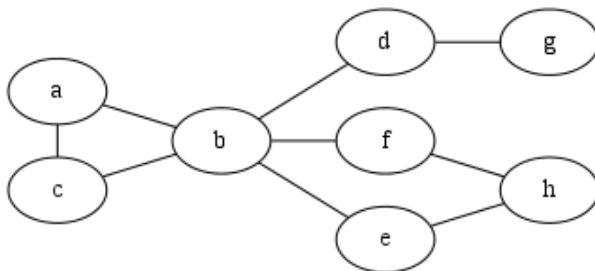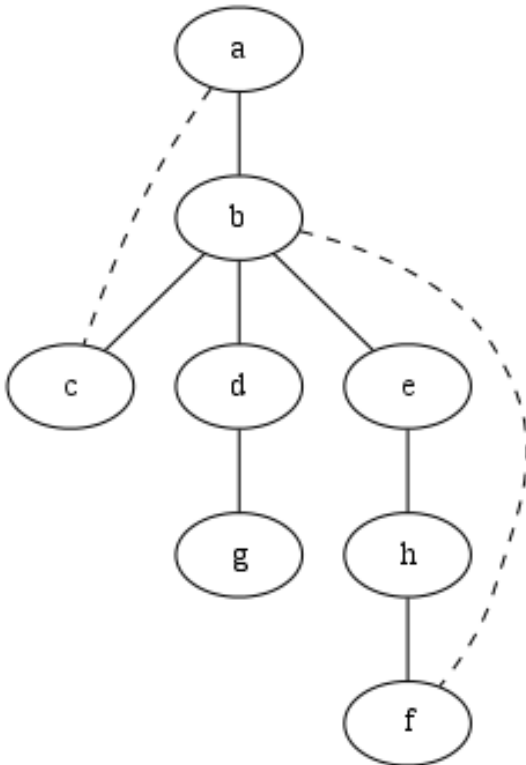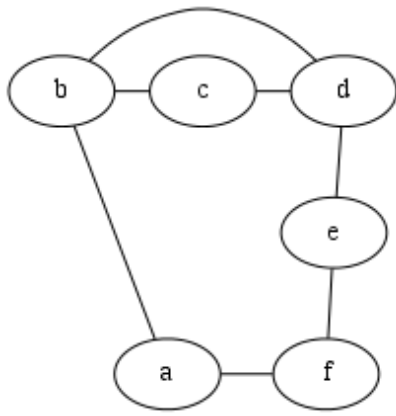

SOLUTION:

1. **b** and **d** are articulation points. Removing **b** (and all edges incident on **b**) disconnects the graph into three components! (The ones containing **a**, **d**, and **f**, each of which also contains at least one other node.) Removing **d** disconnects the graph into two components, of which one just has **g**.

2. The diameter of this graph is 4. The shortest path between nodes **g** and **h** (via **d**, **b**, and one of **e** or **f**) is 4 steps long. This actually only shows a **lower-bound** on the diameter, but if you try all the other pairs of nodes, you'll find the shortest paths between them are all shorter.

3. Here's ours. Yours should look much the same, with possible minor differences in order of nodes at the same level and which edge of (f, h) and (e, h) is dashed:
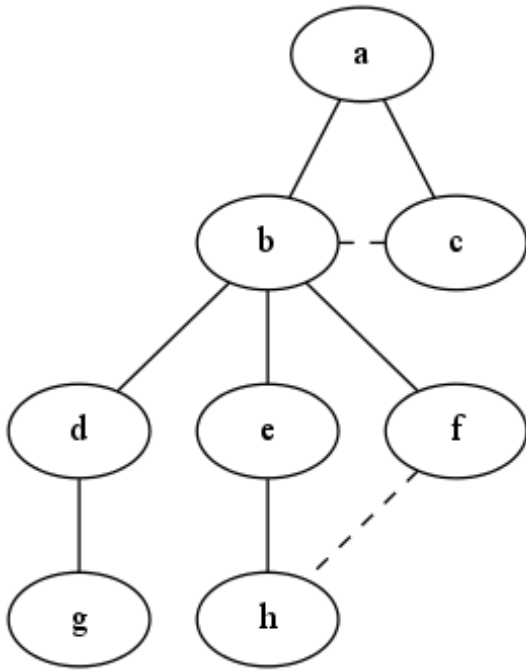

4. Here's ours. Yours may look quite different depending on the order you chose to visit nodes. (DFS generally can have more radically differing shapes depending on order.) We visited alphabetically earlier neighbours first.

SOLUTION:

1. There are no articulation points in this graph. (There are at least two disjoint paths between any pair of nodes; so, removing just one node won't break any other pair's connectivity.)

2. The diameter of this graph is 3, between `c` and `f`. A fun way to double-check this: ignoring `c`, every other pair of nodes is on a single cycle of length 5; so, there will certainly be a 2-step path between them (the shorter "side" of the cycle).

3. Here's ours. As above for BFS, yours should look much the same, with possible minor differences in order of nodes at the same level. (The same edges should be dashed!)
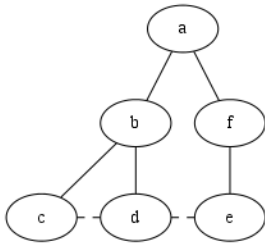
4. Here's ours. Yours should also be a "stick" (i.e., a tree with no branches), but the order of nodes in your stick may differ. We visited alphabetically earlier neighbours first.

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.

   SOLUTION: We can **definitely** argue about this one, but here are some thoughts. An empty graph is either trivial or not allowed. (What's its diameter? 0? ∞? -1 to distinguish it from the one-node graph?) A one-node graph seems more confidently trivial. Its diameter is presumably 0. (Or maybe it's disallowed.) Any graph that's disconnected (i.e., has more than one connected component) is either trivial or not allowed. (I'd call a disconnected graph's diameter ∞, but maybe that's just because I like degenerate cases!) Setting those aside, a two-node **connected** graph is also trivial. Its diameter must be 1.

   Using the notation described below, these are:

   empty graph $G = (\{\}, \{\})$

   one-node graph $G = (\{x\}, \{\})$

   disconnected graph This is hard to specify more succinctly than $G = (\{x, y, \ldots\}, E)$ such that there is no path between x and y.

   two-node connected graph $G = (\{x, y\}, \{(x, y)\})$

   Anything else seems to me to be small, but there are certainly some cases easier than others. (E.g., the diameter of a **tree** can be found using the longest path algorithm for a tree, which runs in linear time in the number of nodes.)

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

   SOLUTION: In my version, any 3-node instance becomes non-trivial. (Your mileage may vary!)

   Here's an amazing ASCII-graphics picture of a 3-node instance: `A -- B -- C`.

   Here's the same instance described using the notation below: $G = (\{A, B, C\}, \{(A, B), (B, C)\})$.

   The solution to this is 2, with A and C bearing witness to that solution, since the shortest path between them is of length 2.

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.

   SOLUTION: Generally, we describe a graph as a tuple $G = (V, E)$. What is that tuple? V is a set of nodes. (As long as we can compare nodes for equality, we don't need to be more specific than that about what a node is, but it's often just a node number or string name.) E is a set of edges. We usually describe an edge as a tuple (v, u), where $v, u \in V$. (That's not the only way or necessarily the best. See below!)

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

   SOLUTION: See above.

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.

   SOLUTION: You might have said that "an edge is a set $\{a, b\}$ of cardinality (size) 2 with $a, b \in V$", in which case you've already disallowed both self-loops (because $|\{a, a\}| = 1$) and repeated edges (because edges are a set and $\{a, b\} = \{b, a\}$). Above, we said an edge is a tuple (v, u), where $v, u \in V$; so, we still need to specify both constraints. (1) No edge can be of the form (v, v), and (2) for a given pair of vertices v and u, only one of (u, v) and (v, u) is allowed in the edge list, where either order represents both orders.

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!)

   SOLUTION: In some sense, the only quantity that matters is a non-negative integer (or if the diameter of a disconnect graph is infinite, a non-negative integer or ∞). However, there's actually more that matter. At least one pair of nodes $u, v \in V$ will define the diameter. The shortest simple path between those will be $(u, w_1, w_2, \ldots, w_k, v)$, where $k \geq 0$, the diameter is $k + 1$, and each neighboring pair is an edge. Note that we may want to relax this a bit so that u can equal v (and thus $k = -1$ in a sense) for one-node graphs to have 0 diameter.

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

   SOLUTION: Interestingly, given a diameter, it's hard to check it. Roughly speaking, it seems no easier than finding the diameter. Given the two nodes that one claims define the diameter, it's somewhat easier to check their shortest path. (We can use BFS from one node, labeling each node with its distance from the root, and produce the label of the other node as the distance. When we add a new node to the tree, its label is one larger than the label of the node from which we reached it.) However, this shortest path doesn't automatically tell us the diameter.

   It **does**, however, give us a lower-bound on the diameter. Using any pair of vertices, we can lower-bound the diameter as the length of the shortest path between them.

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

   SOLUTION: See above, although there was little to add.

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

   SOLUTION: Skipped, since it's a bit painful to draw in LaTeX! But a nice way to do it in the larger example graphs given in the previous problem might be to star the two nodes that define the diameter and shade the edges of the shortest path between them.

3.4 Similar Problems

SOLUTION: We just discussed two related **algorithms**. A natural one to use here is BFS (breadth-first search). That's because BFS gives us the shortest path (in number of edges) from a start point to every other point in the graph.

We haven't discussed a lot of similar problems, but there **are** a variety of them out there. For example, there are several "shortest path" problems, such as "single-source shortest path" and "all-pairs shortest path". In many cases, there are variants of these to handle different conditions like negative edge weights or cycles with negative total edge weight.

There is also a "longest path" problem (related to the Traveling Salesperson Problem or TSP), which is one of a broad class of problems thought to be hard to solve efficiently and exactly.

This also feels similar to various path-like problems such as network flow, minimum spanning tree, and Steiner trees.

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 try to make 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.

   SOLUTION: Bear in mind that we specified a **different** goal for the brute force algorithm than strictly finding the diameter.

   The "form" of a solution to "which two nodes define the diameter by being farthest apart?" is a pair of nodes that **might** define the diameter by being farthest apart. Given a list of the nodes L (i.e., any permutation of V), we might produce all such solutions as:

   ```
   all_solutions(L):
     for i from 1 to |L|:
       for j from i+1 to |L|:
         yield (L[i], L[j])
   ```

   This will yield each pair of vertices in turn without giving both (`L[i]`, `L[j]`) and (`L[j]`, `L[i]`) for any pair of indices `i` and `j`.

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

   SOLUTION: As is usual for graphs (which is discussed in the textbook readings!), we'll probably want to characterize this in terms of the sizes of V and E, which we conventionally write $n = |V|, m = |E|$. These **are** related. In particular, $m \in O(n^2)$ because we can have at most one edge between every pair of vertices. Because we think of disconnected graphs as a special case to be handled separately, we can also assume the graph is connected, in which case, $m \geq n - 1$, since it takes $n - 1$ edges to make a minimally connected graph (a tree).

3. Exactly or asymptotically, how many such "solution forms" are there?

   SOLUTION: In this case, a name isn't really necessary. Out of n nodes, we choose each pair (set of size 2). That's $\binom{n}{2} = \frac{n(n-1)}{2}$.

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?

   SOLUTION: We noted above that the length of the shortest path between any pair of vertices is a lower-bound on the diameter. So, we can track the best pair so far by tracking the length of the shortest path of the best pair. The higher that length, the better the solution is.

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

   SOLUTION: We noted above that BFS is a good tool for determining the shortest path between two nodes. In fact, interestingly, it's overkill. BFS determines the shortest path from a node to **all** other nodes (in an unweighted graph).

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!)

   SOLUTION: Well, a BFS runs in $O(n + m)$ time. We'll do this $\binom{n}{2} \in O(n^2)$ times. That's $O(n^2(n + m))$ runtime. That's not great, but it's also not exponential. It may be good enough for modest-sized graphs. (Certainly, it should do for the two sample graphs above.)

   But... it feels like we could do better!

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.

SOLUTION: You may have come up with a better approach, but we noted above that using BFS on every pair of nodes seems like overkill. After all, BFS already computes the shortest path to **every** node in the graph from a given node. Why not grab the longest of those and use **that** as a lower-bound on diameter? Since this gives us a bound based on a single node (rather than a pair), we can run it once per node to get a $O(n(n + m))$ solution. That's a substantial improvement (a factor of n) and probably fast enough for fairly large graphs.

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:

   We'll "run" our approach on the two sample graphs from the start of this worksheet.

   First graph:

   - From a, we get depths of 1 (b and c), 2 (d, f, and e), and 3 (g and h) for a new lower-bound of 3. (Side note for our implementation: It's safe to start our lower-bound at 0 (perhaps with a special-case for the empty graph if we want it to have diameter -1).
   - From b, we get depths of 1 (a, c, d, f, and e) and 2 (g and h). The maximum of 2 is worse than our current lower-bound of 3. So, no change.
   - From c, we get the same result as a except for a and c themselves (a is at depth 1 rather than c). We get the same 3 as our current lower-bound.
   - From d, we get depths of 1 (b and g), 2 (a, c, f, and e), and 3 (h), which again does not change the lower-bound.
   - From e, we get depths of 1 (b and h), 2 (a, c, d, and e), and 3 (g). No change to our lower-bound.
   - From f, we get depths of 1 (b and h), 2 (a, c, d, and e), and 3 (g). No change to our lower-bound.
   - From g, we get depths of 1 (d), 2 (b), 3 (a, c, f, and e), and 4 (h). The maximum of 4 is better than our best lower-bound so far; so, we update our lower-bound to 4.
   - From h, we get depths of 1 (e and f), 2 (b), 3 (a, c, and d), and 4 (g). The maximum of 4 matches our lower-bound.

   An interesting point you might notice there is that we still seem to be doing some redundant work. After all, we're guaranteed to "go both directions" between any pair of nodes. Perhaps there are tweaks we could make to improve our algorithm, but there's nothing that will make an obvious asymptotic improvement at this point; so, we'll leave it.

   Second graph:

   - From a, we get depths of 1 (b and f) and 2 (c, d, and e), for a new lower-bound of 2.
   - From b, we get depths of 1 (a, c, and d) and 2 (e and f). This doesn't change our lower-bound.
   - From c, we get depths of 1 (b and d), 2 (a and e), and 3 (f). This gives us a new lower-bound of 3.
   - From d, we get depths of 1 (b, c, and e) and 2 (a and f). This doesn't change our lower-bound.
   - From e, we get depths of 1 (d and f) and 2 (a, b, and c). This doesn't change our lower-bound.
   - From f, we get depths of 1 (a and e), 2 (b and d), and 3 (c). This doesn't change our lower-bound.

   Looks like our algorithm is doing well! On a disconnected graph, it will not reach all the nodes when run from any one initial node. So, we can just run an extra BFS at the start from anywhere and check whether we reach all n nodes. If not, we can stop the algorithm and report a result of ∞.

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

   SOLUTION: We're skipping this one, since we know our algorithm is correct. But, you know what would make a good exam and assignment question? To give an **incorrect** algorithm and ask you to give a small input that exercises its flaws. :)