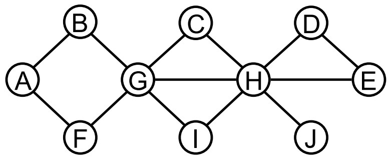
CSE 417
Algorithms & Computational Complexity
Assignment #3 (rev. b)
Due: Friday, 1/28/22

This assignment is part written, part programming. It all focuses on the articulation points algorithm presented in class, and the related problem of decomposing a graph into its biconnected components (defined below). I ***strongly*** recommend that you do problems 1–3 ***before*** you start the programming portion, and do them soon so you have time to ask questions before you get immersed in coding.

Recall that a vertex in a connected undirected graph is an *articulation point* if removing it (and all edges incident to it, i.e., touching it) results in a non-connected graph. (More generally, in a not-necessarily-connected graph, if it increases the number of connected components). As important special cases, a graph having a single vertex has no articulation points, nor does the graph having just two vertices and one edge.

1. Simulate (i.e., by hand) the algorithm presented in class for finding articulation points on the graph shown below.



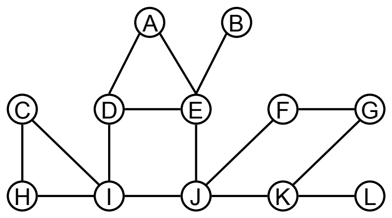
   Redraw the graph clearly showing (a) tree edges, (b) back edges, and, in a tidy table similar to ones shown in the slides, list (c) the DFS number and (d) LOW value assigned to each vertex, and (e) identify the articulation points. In addition, (f) list the edges in the order that they are explored (i.e., traversed for the 1st time) by the algorithm. For definiteness, start your DFS **at vertex C** and whenever the algorithm has a choice of which edge to follow next, pick the edge to the alphabetically first vertex among the choices. (Note that this is *not* a requirement of the algorithm; it's just to simplify grading. The algorithm would correctly identify all articulation points no matter the order in which edges are traversed.)

2. A connected graph is *biconnected* if it has no articulation points. A *biconnected component* of an undirected graph $G = (V, E)$ is a maximal subset B of the edges with the property that the graph $G_B = (V_B, B)$ is biconnected, where V_B is the subset of vertices incident to (touched by) edges in B. (I.e., G_B is B's *edge-induced subgraph*. *Maximal* means you can't enlarge B without destroying its biconnected property.)

   Note that a graph consisting of single edge is biconnected. Consequently, every edge in G is part of *some* biconnected component. In fact, every edge is part of *exactly one* biconnected components. (This really needs a proof, which you don't need to give, but basically it's true because if some edge were in two components, their union would also be biconnected, contradicting the "maximality" condition.) So, the biconnected components partition the edges. To reiterate a point that many people overlook on first reading: a biconnected component is a set of *edges*, not a set if vertices. Each component's edge-induced subgraph of course defines a set of vertices, but these vertex sets do *not* partition the vertices: they overlap. (Where?) Another fact, just to help your intuition: two distinct edges lie on a common simple cycle if and only if they are in the same biconnected component. (This motivates the term "biconnected"—there are always two independent paths between places, excluding the degenerate case where the component is just a single edge. Again, these statements need careful proof, but the idea is simple: if there weren't two paths, you could disconnect the graph by removing some vertex on the one path.)

   For example, the biconnected components and articulation points in the graph below are the following:

   `Component 1:` $\{\{A,D\},\{A,E\},\{D,E\},\{D,I\},\{E,J\},\{I,J\}\}$
   `Component 2:` $\{\{B,E\}\}$
   `Component 3:` $\{\{C,H\},\{C,I\},\{H,I\}\}$
   `Component 4:` $\{\{F,G\},\{F,J\},\{G,K\},\{J,K\}\}$
   `Component 5:` $\{\{K,L\}\}$
   `Articulations:` $\{E,I,J,K\}$
   `Summary:  Example2, 12, 15, 4, 5, 99.9`



Find and list the biconnected components of the graph in problem 1.

3. There is a very close relationship between biconnected components and articulation points. Namely, the articulation points are exactly the vertices at which two (or more) biconnected components are connected. Given this fact, it shouldn't be a big surprise that there is a linear time algorithm that identifies the biconnected components of a graph, and in fact this algorithm is quite similar to the articulation point algorithm.

   Give a modification of the articulation-points algorithm that finds biconnected components in linear time. Describe the algorithm (in English; it should only take a few sentences to describe the necessary modifications.) Simulate it on the example in problem 1, showing enough of a trace of the algorithm's execution to demonstrate that it finds exactly the components you found above. [Hints: look carefully at the order in which the articulation points algorithm explores edges and discovers articulation points, and relate this to which edges are part of which biconnected components. Among other things, you might want to circle or otherwise mark the biconnected components on the edge list you produced as part of your solution to problem 1 (step (f)). Initially, focus on the first biconnected component to be completely explored by the depth-first search.] As always, you should discuss both correctness and complexity of your algorithm, but for this problem, only a short paragraph is required for each, not full proofs.

   To outline what you're expected to give in support if your "simulation", the examples on slides 72-103 and 125-6 may help, but being clear is the first priority – your hard-working TA's need to read and understand it!

4. Implement, test, and time the algorithm you found in problem 3. It is sufficient if your algorithm only handles connected graphs.

   Language: You may use either Java or Python; ask if you have a reason to prefer another language.

   Execution: Name your program "hw3.java" or "hw3.py". During grading, we will run your program from the *command line*, giving it, as command line arguments, the name(s) of one or more input files, each describing one graph, as described below. Process each file, in the order given.

   Input format: The input will consist of an odd number of whitespace-separated integers. ("Whitespace" = one or more space, tab, newline, and/or return characters.) The first must be a positive integer "N"; this is followed by some number of pairs of integers in the range 0 to $N-1$. "N" represents the number of vertices in the graph, and each pair u, v represents an edge between vertices u and v. Although a good program really should check for the following malformed inputs, you may simply assume that you never get: numbers outside the range 0 to $N-1$, an even number of integers in total (i.e., the last pair is unfinished), or a pair u, v where $u = v$, or where either (u, v) or (v, u) duplicates a previous pair in the list. The listed edges may appear in any order.

   Output Format: For graphs having 20 or fewer nodes, print a list of the edges in each biconnected component, followed by a list of the articulation points. Following that, and for all graphs with more than 20 nodes, print a summary that gives (a) the name of the input file, (b) the number of nodes, (c) number of edges, (d) number of articulation points, (e) number of biconnected components, and (f) the algorithm's run time in milliseconds (excluding input/output; see below). Output from your program on the command-line-specified input files should all go to a single .txt file named `hw3out.txt`.

   Because we will be using Gradescope's "autograder" we need to be somewhat rigid about the formats of these outputs. The edges in the nnn-th biconnected component should be listed on a single line, preceeded by "`Component nnn:`"; edges and edge lists should use set notation, i.e., comma-separated values enclosed in "{}"; white-space around the punctuation is allowed. Similarly, the articulation points should be listed, again in set notation, on one line beginning with "`Articulations:`". The summary line should begin with "`Summary:`", and the 6 requested fields should be separated by commas. Insert one blank line following the summary line; otherwise, don't print any blank lines. The example included in problem 2 illustrates the desired format, (except that vertex "names" in the defined input format are always integers, not alphabetic as in that example). Also, the vertex "names" appearing in your output should be the ones from the input, *not* the DFS numbers assigned by your program, and note that it may list the edges, components and articulation points in a different order than shown.

   Implementation: You should use some flavor of edge-list representation so that your algorithm is faster on sparse graphs, but I recommend keeping this as simple as possible. In particular, you may use built-in or standard library packages for hash tables, lists, dynamic arrays, etc., for the edge lists so that you don't need to spend a lot of time duplicating all that fun linked list code you wrote in your data structures course. (However, depending on the language you chose, you may need to be careful to avoid hidden $\Omega(n)$ operations such as copying large data structures, which may increase your algorithm's asymptotic run time.) Since the time spent reading the graph is excluded from your timing study (below), it's OK if the data structures are somewhat slow to construct, just so you can efficiently traverse all the edges as needed by the main algorithm. Also, you do *not* have to obey anything like the "order edges alphabetically" convention I use on by-hand examples—just traverse them in whatever order comes naturally for your data structure.

A recursive implementation of the core DFS algorithm is recommended, and (esp. for purposes of the timing study requested below) should be able to handle graphs with a few thousand nodes and a few tens of thousands of edges. However, I believe that both Java and Python, by default, put somewhat stringent limits on recursion depth and on the space consumed by local variables allocated in recursive functions, so it is recommended that storage for the graph itself be dynamically allocated, (e.g., using "new" in Java, or using libraries that do so). Talk to your TAs if you think you are running into errors due to these limitations.

Test Case: Run it on the specific sample graph ~~that I will provide later~~ in the .zip archive linked here and **turn in the output of your program** when run on this test case. (You should of course do more extensive testing as well, both much simpler and much more complex examples, but you only need to turn in this one case.)

Timing: Also run it on a variety of graphs of different sizes and different densities (average number of edges per vertex) and measure its run time on each.

To simplify your job doing the timing measurements, this zip archive (0.5 Mb) contains a number of random graphs of various sizes in the specified input format (in the "tests" folder). Each should be connected and have one or more biconnected components. If you are interested in viewing them, the smaller of those graphs are also provided as corresponding ".dot" files (in the "dots" folder) for the wonderful, free Graphviz program, or its companion web viewer www.webgraphviz.com. If you want to run some larger tests, this zip archive (89 Mb) contains additional examples. If you want to generate more test cases, or are curious, the "generator" folder contains the C and R code I used to generate these examples. (For the curious, a file named "n64d3s3.txt" has 64 nodes with average degree around 3; "s" = 3 is the seed for the random number generator, irrelevant except for reproducibility.) You are not *required* to use any of this.

Two hints on timing measurements: Record separately or exclude from your timing measurements the time spent reading inputs, since this may dominate the interesting part. Similarly, except for the summary parameters giving numbers of vertices, edges and components, you may want to disable output formatting and printing during your timing runs. See also the FAQ page for other timing tips.

Results: Write a brief report (1–2 pages, say) summarizing your measurements. Include two graphs (old fashioned scatter plots, e.g., using Excel or R) of run time versus problem size, one with "size" defined by the number of vertices, the other with "size" defined by number of edges. Which do you find most informative, and why? Fit linear "trend lines" to the data. [TIP: if you extract just the "`Summary:`" lines from your output file(s), and put them into a file named `something.csv`, then you can directly import them into Excel or R or other plotting programs as "comma separated values".] Compare to the theoretical big-O bounds for your sample graphs. Are your observations in line with the dogma we spout about the utility of big-O analysis? Are there discrepancies? Can you explain them? Does number of edges versus number of vertices have any bearing on performance? Also, if possible, give us a quick summary of the kind of processor/memory system on which you ran your timing tests. E.g., "1.2 Ghz Intelerola Kryptonite dodeca-core processor with 64 kb cache @ 5Ghz and 96 Gb DDR7 DRAM @ 2Ghz."

Turn In: As usual, upload your answers to Gradescope. For problem 4, you will need to upload 3 files: your program's output on the required test case (`hw3out.txt`), your report (`hw3report.pdf`), your code (`hw3.java` or `hw3.py`), all to the respective hw3.4 Java/Python dropbox.

Revision History:
Rev a: Tightened output format requirements — 1/24/22.
Rev b: Added link to required test case — 1/27/22.