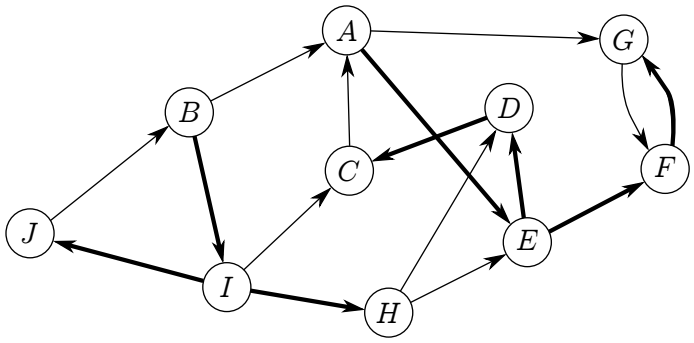
Assignment 6 – Due Thursday, December 1

Objective: The objective of this assignment is for you to combine several algorithms and data structures that we have seen in this class to implement an algorithm for finding strongly connected components in a graph.

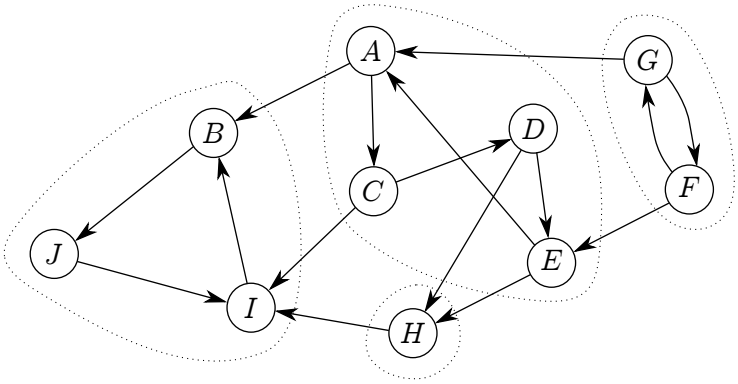
Background: A *strongly connected component* in a directed graph is a maximal subgraph such that there is a directed path between any two vertices in the component (in both directions). In class, we wrote code for depth first search in order to find regular connected components in an undirected graph, and in this assignment you are to use depth first search to find strongly connected components in a directed graph. The algorithm, described below, requires two depth first search operations (slightly modified from what we wrote in class), a stack (you can use the stack provided by `java.util.Stack`), and a graph transpose operation (which you wrote for the previous assignment). To describe the algorithm, consider the following graph, where the dark edges show the edges explored in a depth first search of this graph:



To find strongly connected components, you should use "Kosaraju's algorithm", which works as follows:

1. Use depth first search to explore the graph — in our example, A is visited first, and then we follow the edge to E, and from E we first explore to neighbor D, etc. During the depth first search, push vertices onto a stack when the depth first search has finished exploring all neighbors and the algorithm is about to backtrack. In our example, vertices are pushed on the stack in the following order: $C, D, G, F, E, A, H, J, I, B$ (another way of looking at what this is that we're doing a postorder traversal of the depth first search tree, pushing vertices on the stack during that traversal). *Hint: The stack should be a stack of vertex* names*, not vertex objects, since you'll need to find these vertices in a* different *graph below.*

2. Next, take the transpose of the graph (pictured below for our example — ignore the dashed outlines for now).



3. Finally, do the same depth first search exploration and component printing for the transposed graph that we did in class, *except* instead of a top-level loop that iterates through vertices using the `vertexMap`, you'll iterate through the vertices by popping them off the stack you created in step 1. So in our example, we first pop "B" off the stack, and perform a depth first search to determine the vertices that are reachable from "B" in the transposed graph, giving B, J, I as the first strongly connected component. Next, "I" and "J" are popped off the stack, but since they have been visited we don't do anything with them. Next off the stack is "H" so we perform a depth first search from H and see that H is in a strongly connected component by itself. Continuing in this manner, we explore from A to get component A, C, D, E, and then from F to get component F, G. The dashed outlines on the graph show these components (while they are shown in the transposed graph, these are also strongly connected components in the original graph — do you see why?).

What To Do: Start with the code in Bitbucket, as in previous assignments: fork the "Assign6" repository, rename it to include your username, grant read access to the class administrators, and then use NetBeans on your computer to clone it so you can work with it. This is just a copy of the code we worked with in class, including our class-written depth first search code (slightly cleaned up from class). The repository also contains two data files, "`a6graph1.in`" and "`a6graph2.in`": the first contains the example graph that we used in the discussion above, and the second reflects results from a big chess tournament (we'll say more about this in class). You should implement the algorithm described above, printing out strongly connected components in the same style as we printed regular connected components from an undirected graph.

Submission Instructions: Using NetBeans, commit all changes to your project and do a "push to upstream" to put the most up-to-date files on the Bitbucket server. Remember: Do *not* create a pull request — I will clone your repository (if it exists and you granted me access) at 12:30 on the due date, and will assume that is your submission. If you intend to keep working on your project and submit late, please let me know by email, and I will ignore your repository until the late submission deadline.