CS170 Discussion Section 3: *2/1*

1. *Practice with FT*
   What is the FT of $P(x) = 1 + x^3$? What values of x would we use with the FFT?
   We will find $P(x)$ with x-values $1, i, -1, -i$
   $P(1) = 2$
   $P(i) = 1 - i$
   $P(-1) = 0$
   $P(-i) = 1 + i$

2. *Practice with FFT*

   (a) Run the FFT on $A(x) = 4 + 2x + 3x^2 + x^3$.
   We wish to find $A(1), A(i), A(-1)$, and $A(-i)$.
   We express A in terms of even and odd terms:

   $$A(x) = B(x^2) + xC(x^2)$$

   where $B(x) = 4 + 3x$ and $C(x) = 2 + x$.

   We then recursively evaluate B and C each at 1 and -1. This will give us:

   $$\begin{aligned} B(1) &= 7 \\ B(-1) &= 1 \\ C(1) &= 3 \\ C(-1) &= 1 \end{aligned}$$

   Now we can evaluate A as follows:

   $$\begin{aligned} A(1) &= B(1) + C(1) = 10 \\ A(i) &= B(-1) + iC(-1) = 1 + i \\ A(-1) &= B(1) - C(1) = 4 \\ A(-i) &= B(-1) - iC(-1) = 1 - i \end{aligned}$$

   (b) Is this result enough to compute $A(x)^2$?
   No, $A(x^2)$ will have degree 6 so we need more points to uniquely determine $A(x^2)$.
   We will need to evaluate at the eighth roots of unity instead.

(c) *Extra Practice:* Compute $A(x)^2$ using the FFT.
As stated in (b), we must evaluate A at the eighth roots of unity (four of which we computed in (a)). Let $\omega = e^{\pi i/4}$. We will now evaluate A at the remaining roots. We will write ω^2 instead of i to make the math clearer.

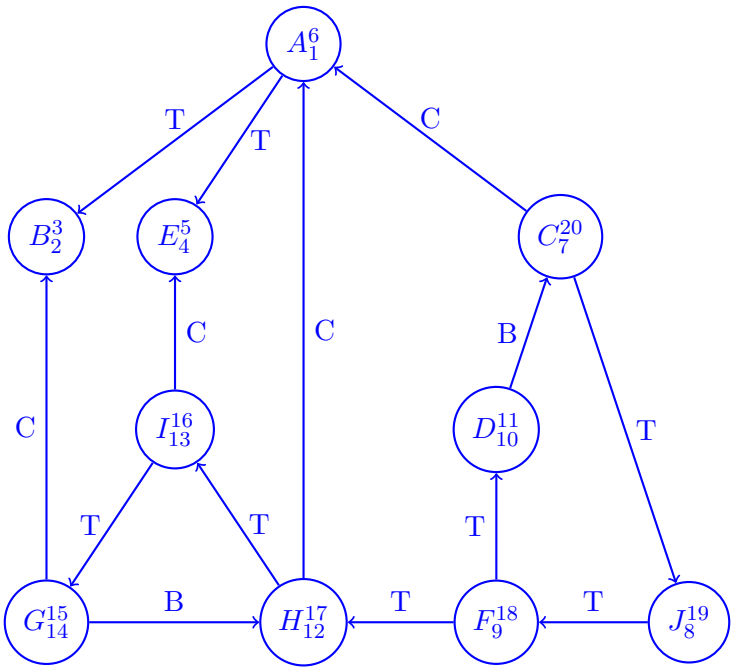
$$B(i) = B(\omega^2) = 4 + 3\omega^2$$
$$B(-i) = B(-\omega^2) = 4 - 3\omega^2$$
$$C(i) = C(\omega^2) = 2 + \omega^2$$
$$C(-i) = C(-\omega^2) = 2 - \omega^2$$

Let $\omega = e^{\pi i/4}$. We will now evaluate A at the remaining roots. Here we will use ω^2 instead of i to make the math clearer.

$$A(\omega) = B(\omega^2) + \omega * C(\omega^2) = 4 + 3\omega^2 + 2\omega + \omega * \omega^2 = 4 + 2\omega + 3\omega^2 + \omega^3$$
$$A(\omega^3) = B(-\omega^2) + \omega^3 C(-\omega^2) =$$
$$A(\omega^5) = B(1) - C(1) =$$
$$A(\omega^7) = B(-1) - iC(-1) =$$

3. *Graph traversal.*

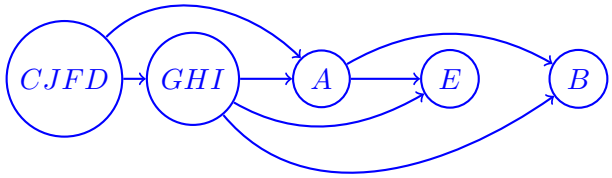
(a) For the directed graph below, perform DFS starting from vertex A, breaking ties alphabetically. As you go, label each node with its pre- and post-number, and mark each edge as **T**ree, **B**ack, **F**orward or **C**ross.



(b) What are the strongly connected components of the above graph?

$$\{A\}, \{B\}, \{E\}, \{G, H, I\}, \{C, J, F, D\}$$

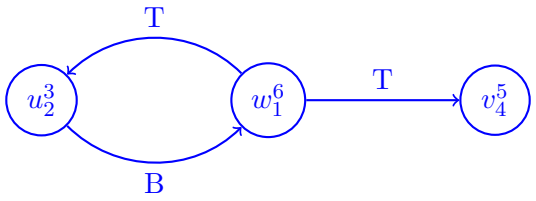
(c) Draw the DAG of the strongly connected components of the graph.



4. *Short answer* For each of the following, either prove the statement is true or give a counterexample to show it is false.

   (a) If (u, v) is an edge in an undirected graph and during DFS, $\text{post}(v) < \text{post}(u)$, then u is an ancestor of v in the DFS tree.
   True. There are two possible cases: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$ or $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$. In the first case, u is an ancestor of v. In the second case, v was popped off the stack without looking at u. However, since there is an edge between them and we look at all neighbors of v, this cannot happen.

   (b) In a directed graph, if there is a path from u to v and $\text{pre}(u) < \text{pre}(v)$ then u is an ancestor of v in the DFS tree.
   False. Consider the following case:



   (c) In any connected undirected graph G there is a vertex whose removal leaves G connected.
   True. Remove a leaf of a DFS tree of the graph.

5. **Extra Practice:** *Compiling on a parallel cluster* We want to compile a large program containing n modules. We are given a dependency graph $G = (V, E)$: G is a directed, acyclic graph with a vertex for each module, and an edge from module v to module w means we must finish compiling v before starting to compile w. Each module takes exactly one minute to compile. We want to compile the program as quickly as possible. We are willing to use Amazon EC2 for this purpose, so we can compile as many modules in parallel as we want, as long as we satisfy the dependencies. Find a linear-time algorithm that computes the minimum time needed to compile the program. Solution: We're looking for the longest path in this DAG, so topologically sort the graph. Then, for each vertex v (in topologically sorted order), we compute the length of the longest path ending at v, in terms of other lengths previously calculated.

   In pseudocode:

   FindCompletionTime(Graph G):
   1. Linearize the graph G, and let $v_1, \ldots, v_n$ denote the vertices of G in topologically sorted order.
   2. For $j := 1, 2, \ldots, n$:
   3. Set $\ell[j] = 1 + \max(0, \max\{\ell[i] : (v_i, v_j) \in E\})$
   4. Return $\max\{\ell[j] : 1 \leq j \leq n\}$.

   This gives us the correct answer because as soon as all predecessors of a module are done compiling we can compile the module, and $\ell[]$ as calculated holds the minimum amount of time needed to compile each module along with all of its dependencies.

6. **Extra Practice:** *True Source* Design an efficient algorithm that given a directed graph G determines whether there is a single vertex v from which every other vertex can be reached. Hint: first solve this for directed acyclic graphs. Note that running DFS from every single vertex is not efficient.

   Solution: In directed acyclic graphs, this is easy to check. We just need to see if the number of source nodes (zero indegree) is 1 or more than 1. Certainly if it is more than 1, there is no true source, because one cannot reach either source from the other. But if there is only 1, that source can reach every other vertex, because if v is any other vertex, if we keep taking one of the incoming edges, starting at v, we have to either reach the source, or see a repeat vertex. But the fact that the graph is acyclic means that we can't see a repeat vertex, so we have to reach the source. This means that the source can reach any vertex in the graph.

   Now for general graphs, we first form the SCCs, and the metagraph. Now if there is only one source SCC, any vertex from it can reach any other vertex in the graph, but if there are more than one source SCCs, there is no single vertex that can reach all vertices.