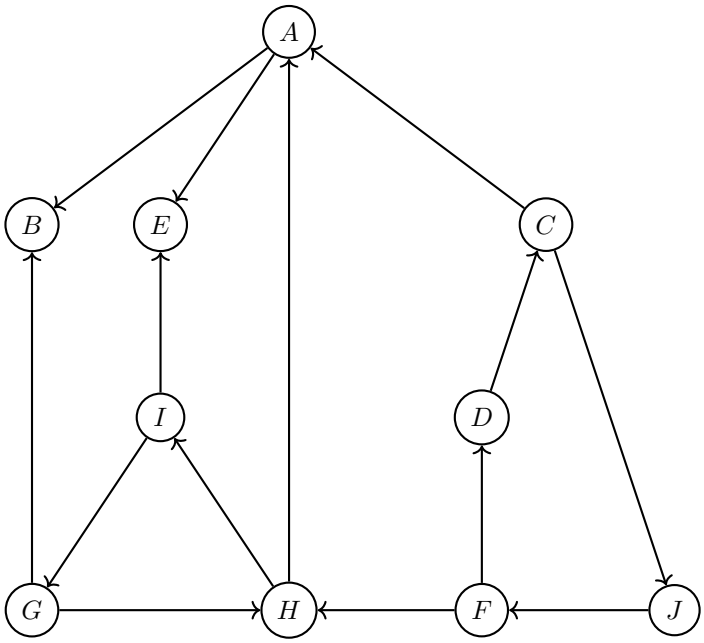
*Note*: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

# 1 Graph Traversal




(a) Recall that given a DFS tree, we can classify edges into one of four types:

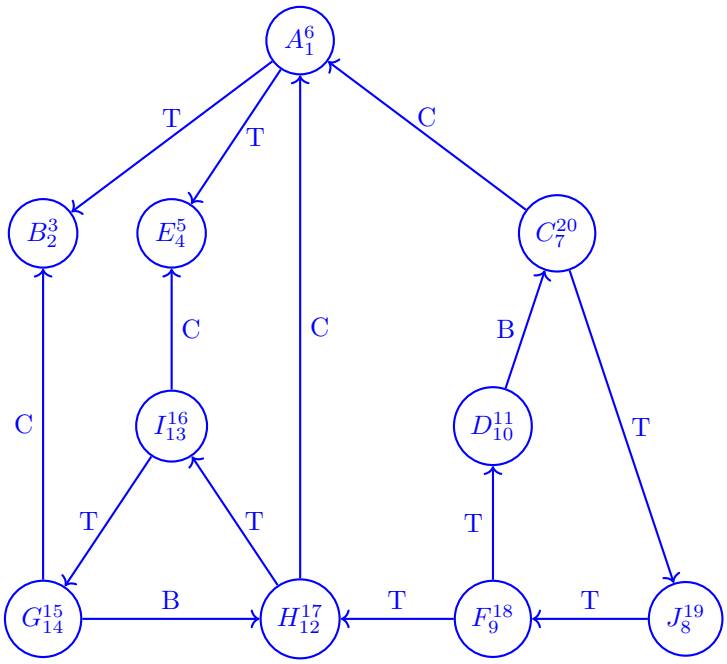
- Tree edges are edges in the DFS tree,
- Back edges are edges $(u, v)$ not in the DFS tree where $v$ is the ancestor of $u$ in the DFS tree
- Forward edges are edges $(u, v)$ not in the DFS tree where $u$ is the ancestor of $v$ in the DFS tree
- Cross edges are edges $(u, v)$ not in the DFS tree where $u$ is not the ancestor of $v$, nor is $v$ the ancestor of $u$.

For the directed graph above, 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?

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

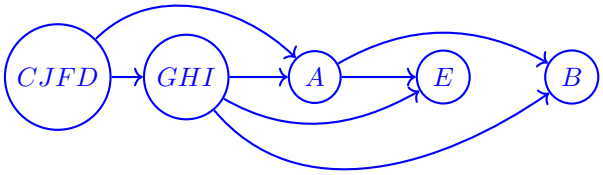
**Solution:**




(a)

(b)

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




(c)

## 2 BFS Intro

In this problem we will consider the shortest path problem: Given a graph $G(V, E)$, find the length of the shortest path from $s$ to every vertex $v$ in $V$. For an unweighted graph, the length of a path is the number of edges in the path. We can do this using the *breadth-first search* (BFS) algorithm, which we will see again in lecture this week.

BFS can be implemented just like the depth-first search (DFS) algorithm, but using a queue instead of a stack. Below is pseudo-code for another implementation of BFS, which computes for each $i \in \{0, 1, \ldots, |V| - 1\}$ the set of vertices distance $i$ from $s$, denoted $L_i$.

```
Input: A graph G(V, E), starting vertex s
for all v ∈ V do
    visited(v) = False
visited(s) = True
L_0 → {s}
for i from 0 to n − 1 do
    L_{i+1} = {}
    for u ∈ L_i do
        for (u, v) ∈ E do
            if visited(v) = False then
                L_{i+1}.add(v)
                visited(v) = True
```

In other words, we start with $L_0 = \{s\}$, and then for each $i$, we set $L_{i+1}$ to be all neighbors of vertices in $L_i$ that we haven't already added to a previous $L_i$.

(a) Prove that BFS computes the correct value of $L_i$ for all $i$ (Hint: Use induction to show that for all $i$, $L_i$ contains all vertices distance $i$ from $s$, and only contains these vertices).

**Solution:** We claim that before we start iteration $i$ of the for loop: (1) all vertices at exactly distance $i$ from $s$ are in $L_i$, and (2) All vertices at distance more than $i$ from $s$ have not been added to any $L_j$, $j \leq i$. We will prove this inductively holds for all $i$, which implies the algorithm is correct.

This holds for $i = 0$. Assume it holds for $i = k$. We will show it holds for $i = k + 1$. (1) holds for $i = k + 1$ because every vertex at distance $k + 1$ is adjacent to some vertex at distance $k$, and thus by inductive hypothesis (1) gets added to $L_{k+1}$ in iteration $k$. (2) holds because no vertex at distance $k + 2$ or more can be adjacent to a vertex at distance $k$ or less, and so the only vertices added to any $L_{k+1}$ in iteration $k$ are those at distance exactly $k + 1$.

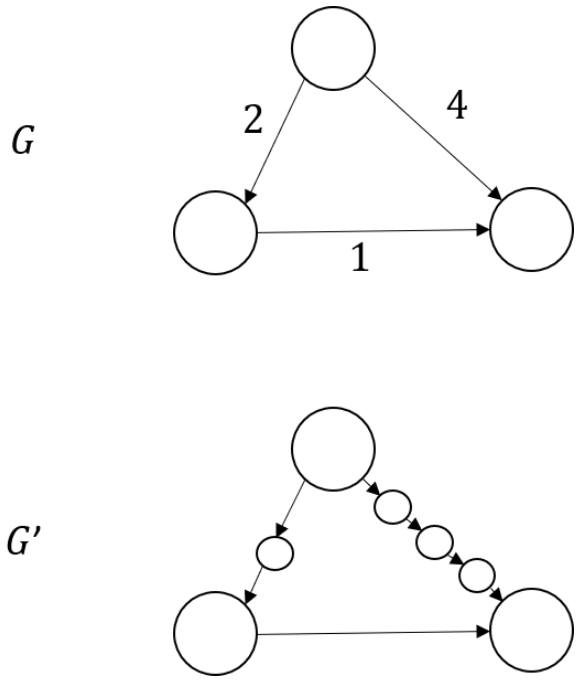
(b) Show that just like DFS, the above algorithm runs in $O(m + n)$ time, where $n$ is the number of nodes and $m$ is the number of edges.

**Solution:** Initializing *visited* takes $O(n)$ time. Iteration $i$ of the for loop takes time $O(\sum_{v \in L_i} \delta(v))$, where $\delta(v)$ is the degree of $v$. Since no $v$ appears in more than one $L_i$, the overall time is $O(n + \sum_{v \in V} \delta(v)) = O(n + m)$.

(c) We might instead want to find the shortest *weighted* path from $s$ to each vertex. That is, each edge has weight $w_e$, and the length of a path is now the sum of weights of edges in the path. The above algorithm works when all $w_e = 1$, but can easily fail if some $w_e \neq 1$.

Fill in the blank to get an algorithm computing the shortest paths when $w_e$ are positive integers: We replace each edge $e$ in $G$ with _____ to get a new graph $G'$, then run BFS on $G'$ starting from $s$. Justify your answer.

**Solution:** A path of $w_e$ unweighted edges. See the below figure for e.g. a directed graph:




(d) What is the runtime of this algorithm as a function of the weights $w_e$? How many bits does it take to write down all $w_e$? Is this algorithm's runtime a polynomial in the input size?

**Solution:** The runtime is $O(\sum_{e \in E} w_e)$, since $G'$ . The number of bits needed is $\Theta(\sum_{e \in E} \log w_e)$. So even though this algorithm's runtime looks like a polynomial, it takes time exponential in the input size when some $w_e$ are large

## 3 Dijkstra's Algorithm Fails on Negative Edges

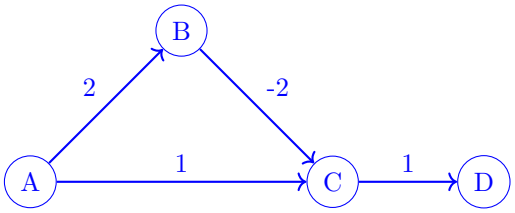
Draw a graph with five vertices or fewer, and indicate the source where Dijkstra's algorithm will be started from.

(a) Draw a graph with no negative cycles for which Dijkstra's algorithm produces the wrong answer.

(b) Draw a graph with at least two negative weight edges for which Dijkstra's algorithm produces the correct answer.

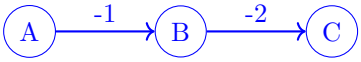
**Solution:**

1. Here's one example:




Dijkstra's algorithm from source $A$ will give the distance to $D$ as 2 rather than 1, because it visits $C$ before $B$.

2. Dijkstra's algorithm always works on directed paths. For example:




## 4 Waypoint

You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights, and there is a special node $v_0 \in V$. Give an efficient algorithm that computes, for all node pairs $s, t$, the length of the shortest path from $s$ to $t$ that passes through $v_0$. Your algorithm should take $O(|V|^2 + |E| \log |V|)$ time.

**Solution:** The length of the shortest path from $s$ to $t$ that passes through $v_0$ is the same as the length of the shortest path from $s$ to $v_0$ plus the length of the shortest path from $v_0$ to $t$.

We compute the shortest path length from $v_0$ to all vertices $t$ using Dijkstra's. Next, we reverse all edges in $G$, to get $G^R$, and then compute the shortest path length from $v_0$ to all vertices in $G^R$. The shortest path length from $v_0$ to $s$ in $G^R$ is the same as the shortest path length from $s$ to $v_0$ in $G$.

These two calls to Dijkstra's take $O((|V| + |E|) \log |V|)$ time. To find the lengths of the shortest paths going through $v_0$ between all pairs, we iterate over the results of the two calls to Dijkstra's, and this takes $O(|V|^2)$ time.

## 5 Dijkstra Tiebreaking

We are given a directed graph $G$ with positive weights on its edges. We wish to find a shortest path from $s$ to $t$, and, among all shortest paths, we want the one in which the longest edge is as short

as possible. How would you modify Dijkstra's algorithm to this end? Just a description of your modification is needed.

(If there are multiple shortest paths where the longest edge is as short as possible, outputting any of them is fine).

**Solution:** Modify Dijkstra's algorithm to keep a map $\ell(v)$ which holds the longest edge weight on the current shortest path to $v$. Initially $\ell(s) := 0$ for source $s$ and $\ell(v) := \infty$ for all $v \in V \setminus \{s\}$. When we consider a vertex $u$ and its neighbour $v$, if $dist(u) + w(u,v) < dist(v)$, then we set $\ell(v) := \max(\ell(u), w(u,v))$ in addition to the standard Dijkstra's steps. If there is a neighbour $v$ of $u$ such that $dist(v) = dist(u) + w(u,v)$, *and* $\ell(v) > \max(\ell(u), w(u,v))$, we set $v$'s predecessor to $u$ and update $\ell(v) := \max(\ell(u), w(u,v))$.

Note: An alternate solution that doesn't get full credit is, letting $W = \max_e w(e)$ and assuming all edge weights are integers, to add e.g. $n^{-1-(W-w(e)}$ to $e$'s edge weight. Using these edge weights, of all the shortest paths, the one with the shortest possible longest edge will have the least weight, and no path can become shorter than a path it was previously longer than. However, writing down the edge weights with $n^{-1-(W-w(e)}$ added to them takes an additional $O(W \log n)$ bits per edge weight. As we saw in the last discussion, algorithms with runtime polynomial in the weights of edges are **not** efficient in general.