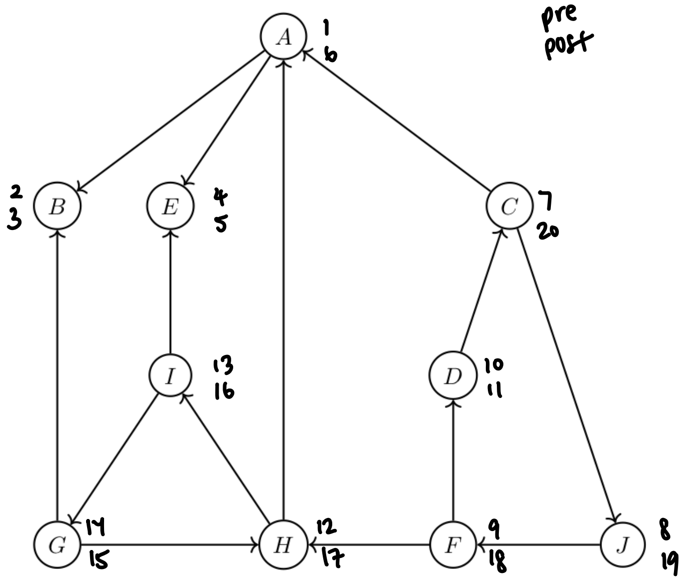
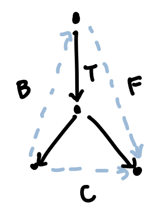
1 Graph Traversal



pre
post

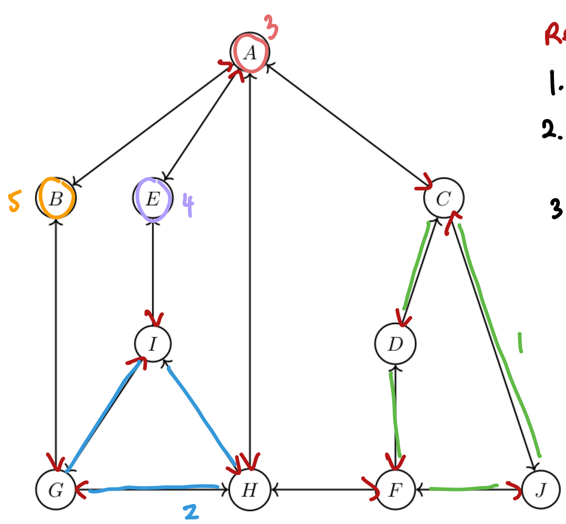


(a) Recall that given a DFS tree, we can classify edges into one of four types:

- 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?



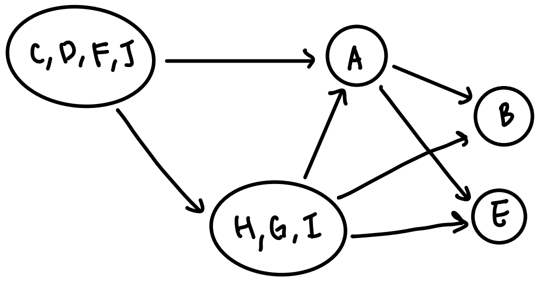
Red arrows indicate reversed edges

1. Run DFS on G (part a)
2. Run DFS starting from vertex with highest post number
3. Each time you have to restart represents a connected component

* Highest post number of DFS must lie in a source SCC in G (think how the only way to reach a vertex in a source SCC is to start at the source)

* A source SCC in G is a sink SCC in G^R

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



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 $v \in V$ **do**
  $visited(v) = False$
$visited(s) = True$
$L_0 \to \{s\}$
for i from 0 to $n - 1$ **do**
  $L_{i+1} = \{\}$
  for $u \in L_i$ **do**
    for $(u, v) \in E$ **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).

Base case: $i = 0$

True that $L_0 = \{s\}$, since we start at s, s is zero distance away from s.

Induction case:

Assume this holds for $i = k$. Will prove for $i = k+1$.

prove no vertex left out

L_k contains all vertices distance k from s.

Every vertex at distance $k+1$ is adjacent to a vertex in L_k.

No vertex at distance $k+2$ or more can be adjacent to a vertex in L_k.

prove there is no unwanted vertex

L_{k+1} set to all neighbors of vertices in L_k so L_{k+1} contains all vertices distance $k+1$ from s.

(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.

Init visited : $O(n)$

Each iteration : $O\left(\sum_{v \in L_i} \deg(v)\right)$

↳ eventually visit each edge twice (once per adjacent vertex)

Overall : $O(n+m)$

(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 integers: We replace each edge e in G with w_e to get a new graph G', then run BFS on G' starting from s. Justify your answer.

$A \xrightarrow{3} B$ becomes $A \xrightarrow[1]{} \circ \xrightarrow[1]{} \circ \xrightarrow[1]{} B$

- now each edge has weight 1
- only one neighbor to visit after each dummy node, so path is still correct

(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?

# of edges: $\sum_{e \in |E|} w_e$

Runtime : $O\left(\sum_{e \in |E|} w_e + n\right) = O\left(\sum_{e \in |E|} w_e\right)$

bits to represent the number N: $\log N$

bits to write down all w_e : $\sum_{e \in |E|} \log w_e$

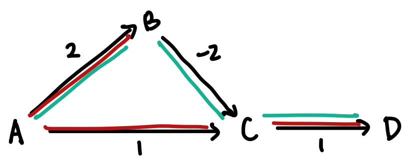
Not polynomial, runtime is exponential in input size.

For runtime to be polynomial it should run in $O\left(\left(\sum_{e \in |E|} \log w_e\right)^c\right)$ for some constant $c > 0$.

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.

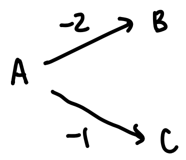
(a) Draw a graph with no negative cycles for which Dijkstra's algorithm produces the wrong answer.



Dijkstra's from A

Correct from A

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

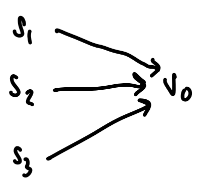


4 Waypoint

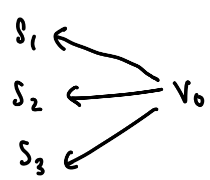
You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights, and there is a special node $v_0 \in V$. Give an efficient algorithm that computes, for all node pairs s, t, the length of the shortest path from s to t that passes through v_0. Your algorithm should take $O(|V|^2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

Observation 1: shortest path $s \to v_0 \to t$ is broken down into two shortest paths $s \to v_0$ and $v_0 \to t$

Observation 2:



Each of these paths can be found with one iteration of Dijkstra's from v_0 on the graph:



Algorithm:

1. Compute shortest paths $v_0 \to t$ by running Dijkstra's on v_0 in G
2. " " " $s \to v_0$ " " " " v_0 in G^R

$O(|E| \log |V|)$

3. Iterate over results of 1 and 2 to piece together all shortest paths between all pairs

$O(|V|^2)$

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.

tie break based on longest edge

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

ex: $A \xrightarrow{1} B \xrightarrow{1} C \xrightarrow{1} D$ ✱ we choose this as THE shortest path

$A \xrightarrow[1]{} E \xrightarrow[2]{} D$

Modify Dijkstra's to keep a map $\ell(v) \rightarrow$ longest edge in shortest path to v

Init $\ell(s) = 0$ and $\ell(v) = \infty$ for all $v \in V / \{s\}$

When considering the edge $u \rightarrow v$

```
if dist(u) + w(u,v) < dist(v):
    edgeTo(v) = u
    ℓ(v) = max(ℓ(u), w(u,v))
if dist(u) + w(u,v) = dist(v) and ℓ(v) > max(ℓ(u), w(u,v)):
    edgeTo(v) = u
    ℓ(v) = max(ℓ(u), w(u,v))
```