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.

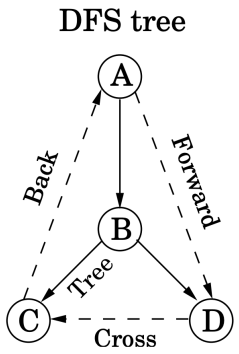
Graphs Cheatsheet

Depth First Search (DFS)

```
def dfs(G, s):
    def explore_recursive(G, v):
        visited(v) = true
        previsit(v) # set the pre-order of v
        for each edge(v, u) in E:
            if not visited(u):
                explore_recursive(G, u)
        postvisit(v) # set the post-order of v

    def explore_iterative(G, v):
        st = stack()
        st.push(v)

        while st is not empty:
            u = st.pop()
            visited(u) = true

            for each edge(u, w) in E:
                if not visited(w):
                    st.push(w)

    # depending on how you want to DFS, you can use
    # either explore_recursive or explore_iterative below
    explore(G, s)
    for all v in V:
        if not visited(v):
            explore(G, v)
```

↪ **Runtime of DFS:** $O(|V| + |E|)$

DFS Tree/Forest: the tree/forest produced by the edges traversed during a given DFS



Edge Types:

- *Tree Edge:* leads to child; part of the DFS Tree/Forest
- *Forward Edge:* leads to a non-child descendant
- *Back Edge:* leads to an ancestor
- *Cross Edge:* leads to a node that's neither a descendant nor an ancestor

Edge Type based on Pre/Post-orders: an edge $(u, v) \in E$ is a:

- *Tree or Forward Edge* if $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
- *Back Edge* if $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$
- *Cross Edge* if $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$

Breadth First Search (BFS)

```
def bfs(G, s):
    q = queue()
    q.push(v)

    while q is not empty:
        v = q.pop()
        visited(v) = true

        for each edge(v, u) in E:
            if not visited(u):
                q.push(u)
```

↪ **Runtime of BFS:** $O(|V| + |E|)$

Strongly Connected Components

A **strongly connected component** of G is a subset of vertices in which there is a path from every vertex to every other vertex.

Kosaraju's Algorithm

Given a graph $G = (V, E)$, we can find all the SCCs as follows:

1. Run DFS on G^{rev} to get the post-order values of all vertices $v \in V$; i.e. we compute $\text{post}^{\text{rev}}(v)$ for all $v \in V$.
2. Run DFS on G starting at the vertex with the highest post-order in G^{rev} (that is un-visited), which must belong in the sink SCC of G. Throughout this DFS, we label each traversed vertex as part of the current SCC.
3. Repeat steps 2-3 until we've labeled all SCCs.

↪ **Runtime of Kosaraju's:** $O(|V| + |E|)$

Dijkstra's Algorithm

```
def dijkstra(G, s):
    for all v in V:
        dist(v) = infinity # distances
        par(v) = none # parents in shortest paths tree

    dist(s) = 0
    h = min_heap() # priority according to distance
```

```
    h.insert((s, 0))

    while h is not empty:
        v = h.delete_min()
        for each edge(v, u) in E:
            if dist(u) > dist(v) + w(v, u):
                dist(u) = dist(v) + w(v, u)
                par(u) = v
                h.decrease_key(u) # sets priority of u to be the updated dist(u)

    return dist, par
```

↪ **Runtime of Dijkstra's:**

- $O((|E| + |V|) \log |V|)$ using a binary min-heap
- $O(|E| + |V| \log |V|)$ using a fibonacci min-heap

1 Graph Short Answer

For each of the following, either prove the statement is true or give a counterexample to show it is false. Note that pre(v) and post(v) denote that pre-order and post-order values of v.

(a) If (u, v) is an edge in an undirected graph and during DFS, post(v) < post(u), then u is an ancestor of v in the DFS tree.

(b) In a directed graph, if there is a path from u to v and pre(u) < pre(v) then u is an ancestor of v in the DFS tree.

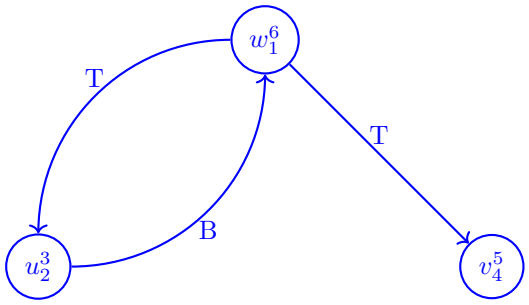
(c) We can modify the SCC algorithm from lecture, so that, the DFS on G is done in decreasing pre-order of G instead of decreasing post-order of G^R. This modified algorithm is also correct.

(d) We can modify the SCC algorithm from lecture so that the first DFS is run on G and the second DFS is run on G^R. This modified algorithm is also correct.

Solution:

(a) 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) False. Consider the following case where we DFS starting from w:



(c) False. Consider a 2-vertex graph with a single directed edge, i.e. $G = (V = \{v_1, v_2\}, E = \{(v_1, v_2)\})$. Then if we start DFS from v_2, the vertex with the largest pre-order in G would be v_1. However, we know that v_1 is not the sink SCC and so this modified algorithm would return the wrong answer.

(d) True. The SCCs of G are the same as the SCCs of G^R.

2 Biconnected Components

Consider any undirected connected graph $G = (V, E)$. We say that an edge $(u, v) \in E$ is *critical* if removing it disconnects the graph. In other words, the graph $(V, E \setminus (u, v))$ is no longer connected. Similarly, we call a vertex $v \in V$ critical if removing v (and all its incident edges) leaves the graph disconnected.

(a) Can you always find a vertex $v \in V$ that is **not** critical? What about an edge that is not critical?

(b) Give a linear time algorithm to find all the critical edges of G.

(c) Modify your algorithm above to find all the critical vertices of G.

(d) A biconnected component of G is a connected subgraph that does not contain critical vertices; in other words, if you remove a vertex from the component, then it will remain connected. If we collapse all the biconnected components of G into a meta-node (similar to what we did with SCCs), what does the resulting graph look like?

Solution:

(a) Consider running DFS from any vertex on the graph, and any leaf in the resulting DFS tree. The leaf can be removed without disconnecting the graph, since the remaining vertices are connected using the DFS tree edges. But we can't always find such an edge. For example, every edge in a tree is critical.

(b) Perform DFS on G while keeping track of $pre[v]$ for each vertex. We also maintain a low value for each vertex, where $low[v]$ denotes the smallest $pre[u]$ such that there is a back edge to u from the subtree of v. The only potential critical edges are the tree edges in the DFS. An edge between v and its parent p is critical if and only if $low[v] > pre[p]$ (i.e. there does not exist a back edge from v to p or any of its ancestors).

(c) The root of the DFS tree is critical iff it has more than one child. A non-root vertex v is critical iff for at least one of its children c, $low[c] > pre[v]$.

(d) The resulting graph is a tree.

To prove this, show that the if there is a cycle in the resulting graph then all the meta nodes in the cycle will belong to the same biconnected component. This gives a contradiction.

3 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.

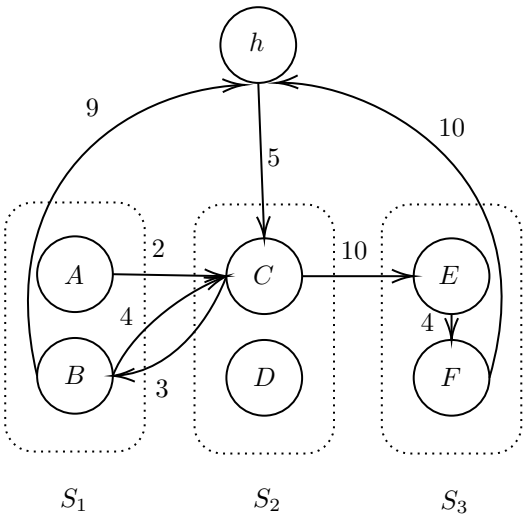
These two calls to Dijkstra's take $O((|V|+|E|)\log|V|)$ time. To find the lengths of the shortest paths going through v_0 between all pairs, we iterate over the results of the two calls to Dijkstra's, and this takes $O(|V|^2)$ time.

4 Running Errands

You need to run a set of k errands in Berkeley. Berkeley is represented as a directed weighted graph G, where each vertex v is a location in Berkeley, and there is an edge (u, v) with weight w_{uv} if it takes w_{uv} minutes to go from u to v. The errands must be completed in order, we'll assume the ith errand can be completed immediately upon visiting any vertex in the set S_i (for example, if you need to buy snacks, you could do it at any grocery store). Your home in Berkeley is the vertex h.

Given G, h, and all $(S_i)_{i=1}^k$ as input, give an efficient algorithm that computes the least amount of time (in minutes) required to complete all the errands starting at h. That is, find the shortest path in G that starts at h and passes through a vertex in S_1, then a vertex in S_2, then in S_3, etc.

For instance in the graph below, the shortest such path is $h \to C \to B \to C \to E$ and the time needed is $5 + 3 + 4 + 10 = 22$.

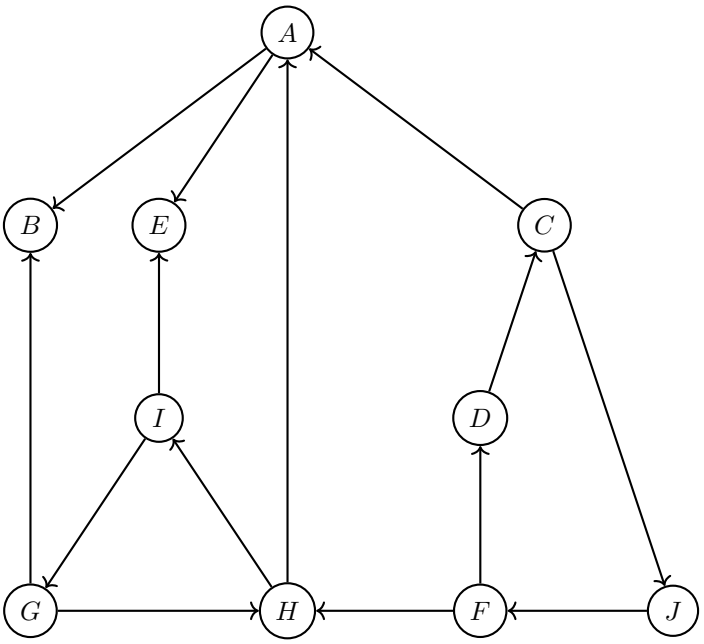


Solution: Main idea Create $k+1$ copies of G, called $G_0, G_1, ...G_k$, to form G'. Let the copy of v in G_i be v_i. For every v in S_i, we add an edge from v_{i-1} to v_i with weight 0. We run Dijkstra's starting from h_0 in G', and output the shortest path length to any vertex in G_k.

Correctness Any path in G' from h_0 to a vertex G_k can be mapped to a path in G of the same length passing through vertices, by taking each edge (u_i, v_i) and replacing it with the edge (u, v) in G, ignoring edges of the form (v_{i-1}, v_i). These paths must also complete the errands in order, since they must contain edges of the forms $(v_0, v_1), (v_1, v_2), \ldots$ in that order.

Runtime analysis This takes time $O(k(|V| + |E|) \log k|V|)$ since the new graph is k times the size of the original graph.

5 Graph Traversal (Optional)



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

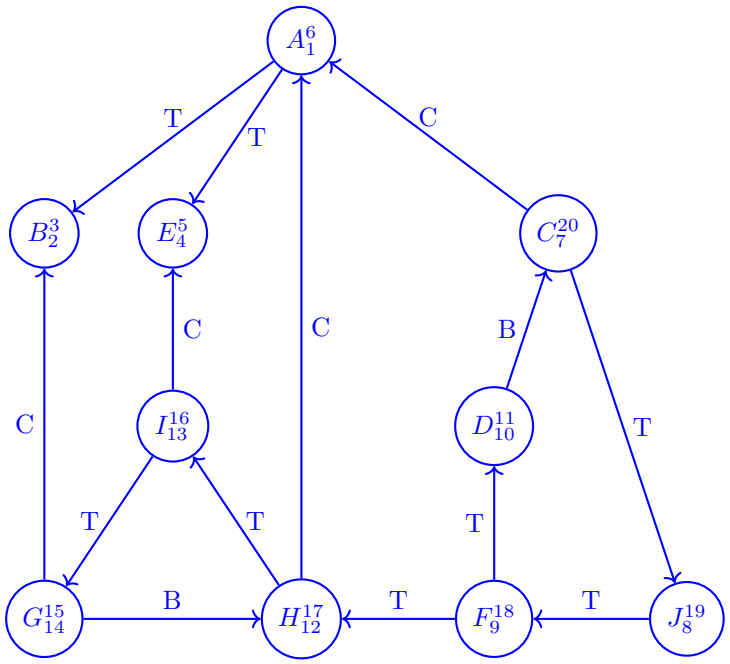
- Tree edges are edges in the DFS tree,
- Back edges are edges (u, v) not in the DFS tree where v is the ancestor of u in the DFS tree
- Forward edges are edges (u, v) not in the DFS tree where u is the ancestor of v in the DFS tree
- Cross edges are edges (u, v) not in the DFS tree where u is not the 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) A strongly connected component (SCC) is defined as a subset of vertices in which there exists a path from each vertex to another vertex. What are the SCCs of the above graph?

(c) Collapse each SCC you found in part (b) into a meta-node, so that you end up with a graph of the SCC meta-nodes. Draw this graph below, and describe its structure.

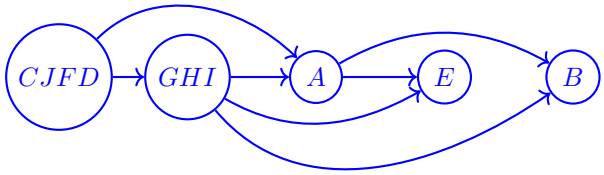
Solution:



(a)

(b)

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



(c)

6 Dijkstra's Algorithm Fails on Negative Edges (Optional)

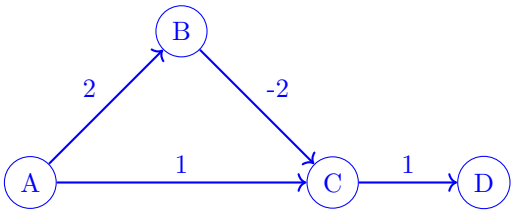
Draw a graph with five vertices or fewer, and indicate the source where Dijkstra's algorithm will be started from.

(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.

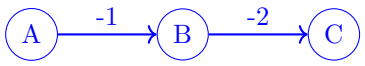
Solution:

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:



7 BFS Intro (Optional)

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.

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.

Algorithm 1 Vertex Distances via BFS

```
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
```

(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 $k + 1$.

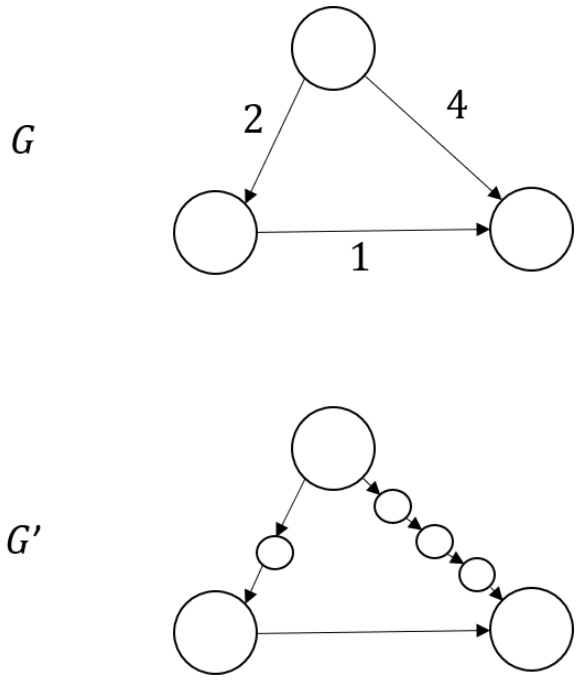
(b) Show that just like DFS, the above algorithm runs in $O(m+n)$ time, where n is the number of nodes and m is the number of edges.

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' has $\sum_{e \in E} w_e$ edges. 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