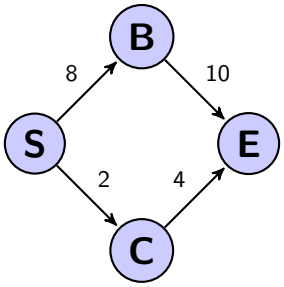
1 Depth First Search and Connected Components

Runtime: $O(|V| + |E|)$
DFS uses the explore function to assign post and pre numbers to nodes. V can reach U if post(V) > post(U) in a Directed Acyclic Graph.

pre(V) < pre(U) does not prove anything. Consider the case where we ran DFS and obtained first path SCE. B would have a larger pre than E



Can V reach U if pre(V) is greater than pre(U)? run DFS starting at V to find out.
Types of edges: Tree edges lead to a child in the BFS/DFS tree. Forward edges lead from a child to a non-child descendant in the BFS/DFS tree. Back edges lead to an ancestor in the BFS/DFS tree. Cross edges lead to neither - they lead to a node that has already been postvisited. For edge (u, v): Tree and Forward edges = uvvu. Back edge = vuuv. Cross edges = vvuu.

DAGs: Every edge leads to a vertex with a lower post number. A DAG has a cycle iff a DFS reveals a back edge.

Topological Sort: A topological sort is always possible for any DAG and is always impossible for non-DAG. If we sort by decreasing post number in a DAG, we get a topological sort.

SCC: V_1, .., V_K is an SCC if each V_i is connected to V_1, .., V_K. If we run explore anywhere in a sink component, we will find all of that sink. To find a meta graph, run DFS on G^R (source become sink, sink become source) and vertex with highest post number is in a sink of G. Run DFS on G starting at vertex with highest post number to find the sink. Continue using next highest post number not in sink to find the meta graph.

Claim: If two SCC A and B, and A points to B, then the highest post number in A > highest post number in B.
Proof:
1) Visit A before B: Because a DFS only returns when all reachable nodes are visited the Post NO in A > Post in B.
2) Visit B before A: A is not reachable from B. A will not be visited when B is fully explored. Clearly Post A > Post B

2 Breadth First Search and Dijkstra's Algorithm

BFS Runtime: $O(|V| + |E|)$
BFS uses a queue to branch out and find a goal, with each nodes length being cost(parent) + 1. Prove BFS has the exact path cost using induction by showing it overestimates cost (either ∞ or 0 at start) and underestimates (V_i is at most i and $V_{i+1} \leq$ i + 1)

```
procedure bfs(G, s)
dist(u) = ∞ for all vertices
dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
    u = eject(Q)
    for all edges (u, v) ∈ E:
        if dist(v) = ∞:
            inject(Q, v)
            dist(v) = dist(u) + 1
```

Dijkstra's Algorithm: an algorithm created to find the shortest path from node S to node T in any directed, positively weighted graph. In Dijkstra's Algorithm, every edge is updated once in a particular order. Dijsktra's uses priority queue to

decide the order of updates.

First, set weight for all vertices to ∞. Set weight for starting vertex to 0 and put it into a Priority Queue, q. While q is not empty, pop out a vertex, v. For all edges (v, w), if $w.dist > v.dist + weight(v, w)$, then $w.dist = v.dist + weight(v, w)$ and insert w to q.

At each step of algorithm, the poped vertex is set correctly to the true shortest path distance. Could be proved by induction

```
procedure dijkstra(G, l, s)
    for all u ∈ V :
        dist(u) = ∞
        prev(u) = nil
    dist(s) = 0
    H = makequeue (V) (using dist-values as keys)
    while H is not empty:
        u = deletemin(H)
        for all edges (u, v) ∈ E:
          if dist(v) > dist(u) + l(u, v):
              dist(v) = dist(u) + l(u, v)
              prev(v) = u
              decreasekey(H, v)
```

Dijkstra's Runtime: $|V| * deletemin * (|V| + |E|) * insert$. With priority queue implemented as an array: $O(|V|^2)$: $O(|V|)$ for deletemin, $O(1)$ for insert/decreasekey, . With priority queue implemented as a heap(Min Heap): $O(log|V| * (|V| + |E|))$. $O(log|V|)$ for deletemin, $O(log|V|)$ for insert/decreasekey. Use a heap if $|E| << |V|^2$. Does not work for negative costs.

With negative edges: update all edges a lot, for $|V| - 1$ steps. At the first inner loop, all edges are updated. The first edge of any shortest path is updated. Bellman-Ford $O(|V| * |E|)$. To optimize we can terminate after a round with no distance updates.

```
procedure bellmanFord(G, l, s)
    for all u ∈ V :
        dist(u) = ∞
        prev(u) = nil
    dist(s) = 0

    repeat |V| - 1 times:
        for all edges (u, v) ∈ E:
          dist(v) = min( dist(v), dist(u) + l(u, v))
```

Determine negative cycles by calling update $|V| - 1$ times. Then call it again. If any node's distance decreases, we have a negative cycle.

Shortest Path in DAG: lineraize DAG by DFS, then visit edges in order and update weights

```
procedure DAGShortest(G, l, s)
    for all u ∈ V :
        dist(u) = ∞
        prev(u) = nil
    dist(s) = 0
    Linearize G
    for all edges (u, v) ∈ E:
     dist(v) = min( dist(v), dist(u) + l(u, v))
```

3 Minimum Spanning Tree

Properties of Trees: A tree on n nodes has n-1 edges. Any connected, undirected graph with $|E| = |V| - 1$ is a tree. An undirected graph is a tree iff there is a unique path between any pair of nodes.

Prim's Algorithm
At each step in the algorithm, attach one vertex adjacent to the already attached vertices by finding the edge with the

smallest weight (not cost of vertex) that connects an unattached vertex to an attached vertex. This is basically Dijkstra's with the priority queue ordered by edge weight. Runtime is the same as Dijkstra's algorithm. Relies on Cut Property. $O((|V| + |E|) * log|V|)$

Kruskal's Algorithm
More simple to implement than Prim's algorithm since it is a greedy algorithm. First, sort the edges by weight, smallest to largest. Then, iterate through the edges. If adding an edge e does not create a cycle, then add e to the MST. Takes $O(|E| * log|V|)$ to sort the edges, and another $O(|E| * log|V|)$ for the union and find operations

```
procedure kruskal(G, w)
    for all u ∈ V :
        makeset(u)
    X = {}
    Sort the edges E by weight
    for all edges {u, v} ∈ E, in increasing order of weight:
        if find(u) != find(v):
            add edge {u, v} to X
            union(u, v)
```

Union-Find Data Structure
Finds an element and returns the set containing it with Find(u). Union(u,v) unions the sets containing u and v. Attaches root with shorter depth to root with longer one. Find(u) can be optimized be attaching all nodes on the path to u to the root.

4 Max Flow / Min Cut

Cut Property: Given any subset of vertices, $S \subset V, S \neq \emptyset$, then the cut $(S, \bar{S})$ is defined to be the set of edges with one end in S and the other in $\bar{S}$. For any cut $(S, \bar{S})$, the unique smallest edge in the cut must be a part of all MSTs of G. For any cut $(S, \bar{S})$, any minimal edge must be part of some MST of G.

Ford Fulkerson: The Ford-Fulkerson algorithm was created to find the max flow in a graph. First, create a "residual" graph that is a copy of G, and a "solution" graph that is a copy of G, with all edges having a weight of 0. Then, as long as the start node S and the end node T are connected in the residual graph, find a path from S to T. Find the minimum weighted edge e along this path (call this value w). w represents the maximum flow that can go through this particular path. So in the solution graph, add w to every edge in this path. In the residual graph, subtract w from every edge in the path, and add w/create an edge of weight w going in the opposite direction of e. Once there are no more paths for S to T, you have found the max flow of the graph. The final max flow should have the same value as the minimum cut of the graph.

Runtime: We argued that it took at most F iterations, where F is the value of the max flow. Each iteration could be done with a DFS in linear time, so this is a total runtime of $\theta(F * (|V| + |E|))$. A different book might write it as $\theta(F * |E|)$., since we are assuming that the graph is connected, so $|V| + |E| = O(E)$.
Later, it was proven that if you always use shortest paths, (often called the Edmond's Karp algorithm) then you will always terminate in $\theta(|V| * |E|)$ iterations. Each iteration can be done with a BFS in linear time, for a total runtime of $\theta(|V| * |E| * (|V| + |E|))$, which could also be written $\theta(|V| * |E|^2)$.

Max flow - Min cut theorem: In any weighted directed graph the Max s-t flow = Min s-t cut.
For any s-t cut, the sum of the edges in the cut are an upper bound on the flow.

To check that a flow is legal: For each edge, check that the flow value is not greater than the capacity. For each vertex, check that the sum of the flow on its incoming edges is equal to outgoing edges.

To check that it is a max flow: Construct a residual graph using the flow and run DFS from s-t. If there is a path found, the flow is not a max flow.

Max Flow is also useful for Bipartite matching. Recall the jobs/workers example form class where we wanted to maximize the number of completed jobs.