CS 3510: Design & Analysis of Algorithms (Section C)

Exam 2 (Graph Algorithms) Study Guide

Professor Abrahim Ladha

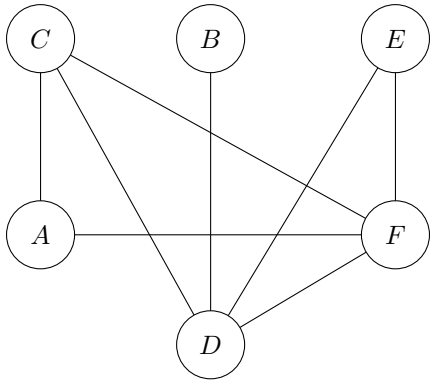
1 Recommended Problems:

- (DFS, SCC) DPV 3.1-3.5, 3.7, 3.8, 3.11, 3.15, 3.16, 3.22, 3.24
- (BFS, Shortest Paths) DPV 4.1-4.3, 4.8, 4.11, 4.12-4.14, 4.20, 4.21
- (MSTs) DPV 5.1, 5.2, 5.5, 5.6, 5.7, 5.9, 5.20, 5.22, 5.23
- (Max-Flow Min-Cut) DPV 7.10, 7.17

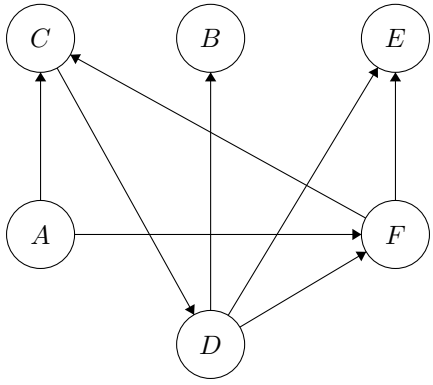
2 Topic List:

2.1 Introduction to Graphs

Let's briefly review graph concepts. A **graph** is a structure comprised of **vertices** and **edges** connecting them. We represent a graph as $G(V, E)$, where V is the set of vertices and E is the set of edges. In this class, we only look at *simple* graphs, meaning that they do not contain double edges between two vertices or self-loops. Below is an example of a simple, **undirected** graph:



Here, we can traverse from vertex D to C and from C to D using the same edge - this is why we refer to these edges as undirected. Consider a similar graph, but with one-way edges instead of two-way:



This is known as a **directed** graph. In this case, the edge CD can only be used to travel from edge C to edge D, and not the other way around.

Another important observation about graphs are **cycles**. Consider the edges DF, FC, CD. These form a triangle of edges which we can use to start at vertex D and end up back at D after traversing the edges. A **path** is any traversal of the graph without repetition of edges or vertices, while a **walk** is a traversal of the grpah with repetition allowed. Therefore a cycle is a special case of a path where the start and end vertices of the path are the same.

Graph Representation
While this isn't something we've focused much on this unit, let's also review canonical graph representation you would use if doing a coding problem with graphs. There are two main methods of graph representation:

Adjacency Matrix. Define a matrix $A[|V|, |V|]$ where entry $A[i, j]$ determines whether there is an edge from vertex v_i to v_j. If $A[i, j] = 1$ there is an edge, and the entry is 0 if not. Note that in an undirected graph, $A[i, j] = A[j, i]$ for all vertices v_i and $_j$.

Adjacency List. Define a list $A[|V|]$ where entry $A[i]$ stores a list of vertices to which vertex v_i has edges. For example, of v_1 has edges to v_2, v_4, v_{10}, then $A[1] = [2, 4, 10]$.

Note that graphs may also be **disconnected**, in that there could be two subgraphs that have no edges to the other.

2.2 Graph Traversal

2.2.1 Depth First Search

Suppose we want to find all vertices reachable from some vertex v in a graph G. Let's define a method called `Explore` that achieves this:

```
Explore(G, v):
    visited[v] = true
    previsit(v)
    for each edge v → u from v:
        if not visited[u]:
            Explore(G, u)
    postvisit(v)
```

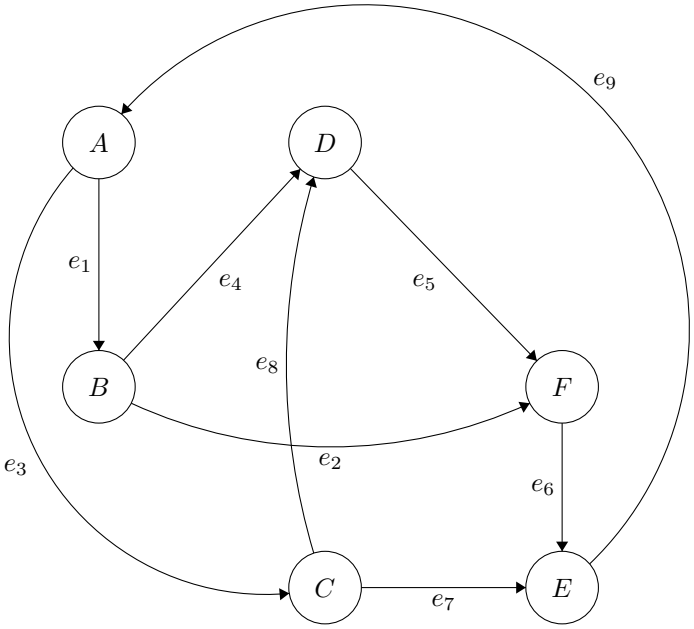
At the end of `Explore`, the visited set tells us exactly which nodes are reachable from the initial node v. *previsit* and *postvisit* are two small functions which store the *pre* and *post* numbers of each vertex that's looked at. We utilize a global counter `clock` that is incremented each time one of these methods is called and then store it into the node's pre/post number:

```
previsit(v):
    pre[v] = clock
    clock++

postvisit(v):
    post[v] = clock
    clock++
```

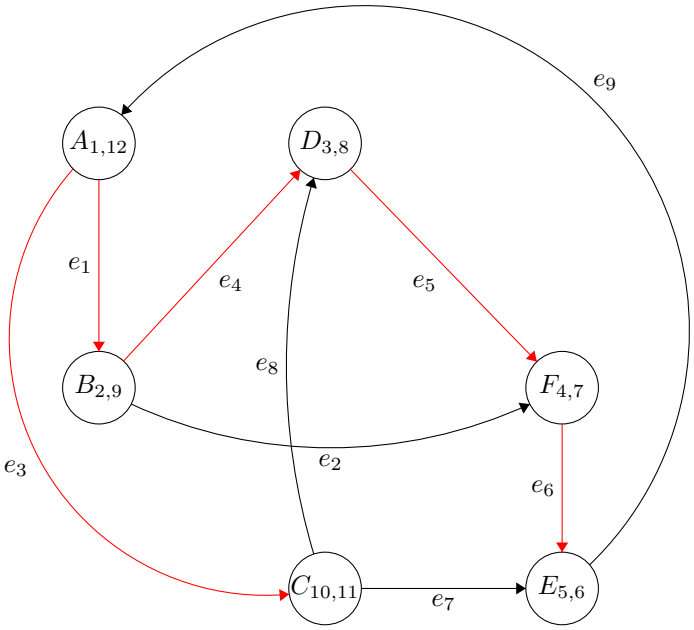
In an undirected connected graph, we can run `Explore` from any vertex and be able to reach all vertices. However, in the case of directed graphs (and definitely disconnected graphs), this may not happen. Therefore we define our full DFS algorithm as follows:

```
Depth First Search(G):
    for each vertex v in V
        visited[v] = false

    for each vertex v in V:
        if not visited[v]:
            Explore(G, v)
```

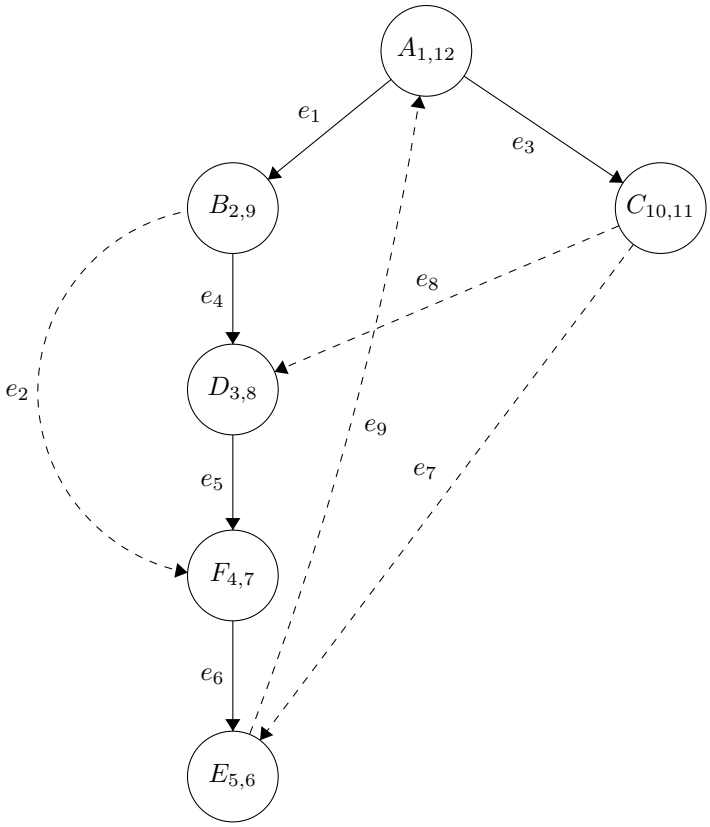
Let's visualize DFS on the following directed graph. When choosing between edges, we can specify that our algorithm will choose vertices based on alphabetical order. Therefore, in this case, we would start from vertex A.



We would traverse edges in the following order: e_1, e_4, e_5, e_6, e_3. After edge e_6 we must backtrack, as edge e_9 points to A which has already been visited, and edge e_2 points to F which has already been visited. We can now calculate the pre and post numbers of each vertex. Here, $v_{a,b}$ denotes that a is the pre-number and b is the post-number of v. We can also highlight the edges we actually explored:



With those edges, we can construct a *DFS Tree*:



The edges e_1, e_3, e_4, e_5, e_6 are classified as **tree** edges, since they are part of our DFS tree as they were explored. The other edges are not part of the tree, so they are dashed. We can classify these edges as a **forward** edge, **cross** edge, or **back** edge. To classify an edge $u \to v$, we can use the following definitions:

a) Tree edge: explored in DFS

b) Back edge: points from a vertex to an ancestor in the DFS tree. `pre[v] < pre[u] < post[u] < post[v]`

c) Forward edge: points from vertex to descendant in DFS tree. `pre[u] < pre[v] < post[v] < post[u]`

d) Cross edge: points from vertex to another branch in DFS tree. `pre[v] < post[v] < pre[u] < post[u]`

Based on these definitions, e_2 is a forward edge, e_7 is a cross edge, e_8 is a cross edge, and e_9 is a back edge.

> Edge classification with DFS is also really useful for cycle detection algorithms! When we see a back edge such as e_9 in the above graph, we can immediately tell there is a cycle $ABDFEA$. This is an example of why keeping track of the pre and post numbers in DFS is really helpful.

2.2.2 Breadth First Search

In DFS, we utilized a recursive stack to push and pop vertices to explore. Essentially, we prioritized going down a path in the graph. What if we wanted to search the graph level-by-level rather than backtracking? To achieve this, we can utilize a queue instead of a stack. We could perform DFS iteratively by creating a stack ourselves and then manually pushing and popping elements, but recursion handles that for us. We can define a Breadth First Search algorithm as follows:

```
Breadth First Search(G, start):
    for each vertex v in V:
        dist[v] = ∞
    dist[start] = 0
    queue = start
    while queue is not empty:
        u = queue.dequeue()
        for all edges (u, v) from u:
            if dist[v] == ∞:
                queue.enqueue(v)
                dist[v] = dist[u] + 1
```

Instead of just using a visited set, BFS allows us to track the distance between two vertices (the number of edges between them). BFS is guaranteed to find us the path with the shortest amount of edges from one vertex to another, since we expand our layer of vertices out one edge at a time.

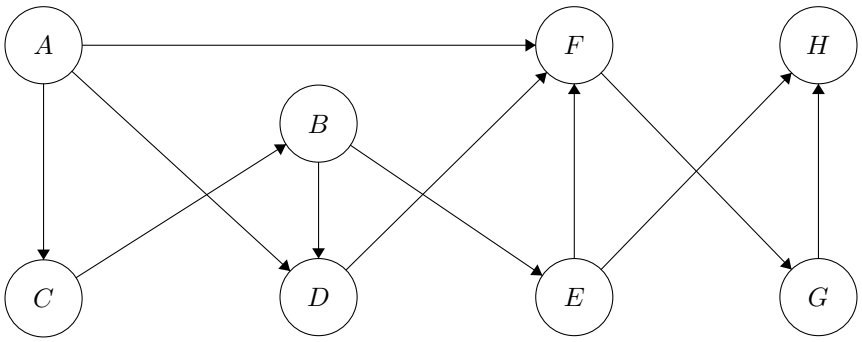
The efficiency of both DFS and BFS is $\mathcal{O}(|V| + |E|)$ since we traverse all edges and vertices.

2.3 Topological Sort

Let's define an important term: a **directed acyclic graph**, or DAG, is a directed graph that does not contain any cycles. What if we wanted to sort the vertices in such a way that if a vertex v is a descendant of another vertex u, v appears after u in the sort? This is known as topological sorting.

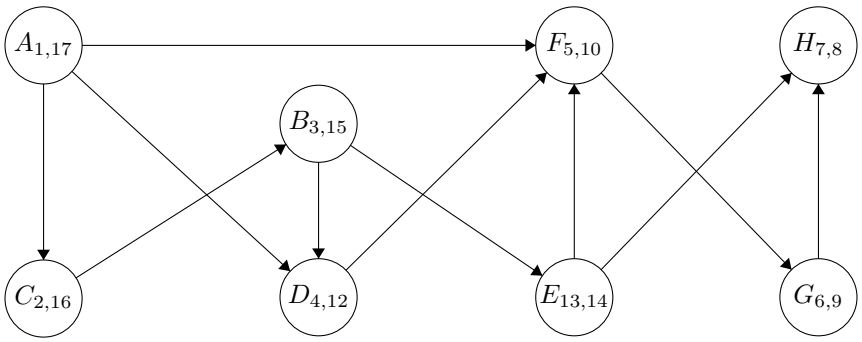
Formally, a **topological sort** of vertices V in a graph is some ordering $[v_1, v_2, \ldots, v_k]$ such that for all edges $(v_i, v_j) \in E$, $i < j$ in the ordering.

Let's look at an example. Given the following DAG:



A valid topological ordering would be A, C, B, D, E, F, G, H. We can see this is true by inspection, as all edges satisfy the conditions.

What if we wanted to do this algorithmically? As it turns out, we can utilize the pre/post numbers from DFS. If we perform DFS on the above DAG, we get the following pre/post numbers:

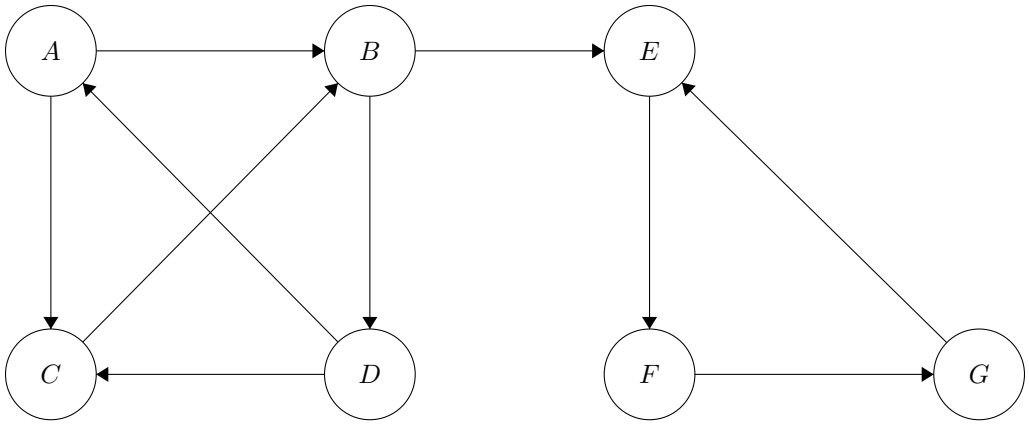


Now, to obtain a valid topological ordering, we can sort the vertices based on their post numbers in *descending* order: A, C, B, E, D, F, G, H. Notice that we found a different ordering from the one above - vertices D and E have switched orders.

We can define a **source** vertex as a vertex which has only outward edges, and a **sink** vertex as a vertex which has only inward edges. In the above example, A is a source vertex, while H is a sink. There may be multiple sources and sinks.

2.4 Strongly Connected Components

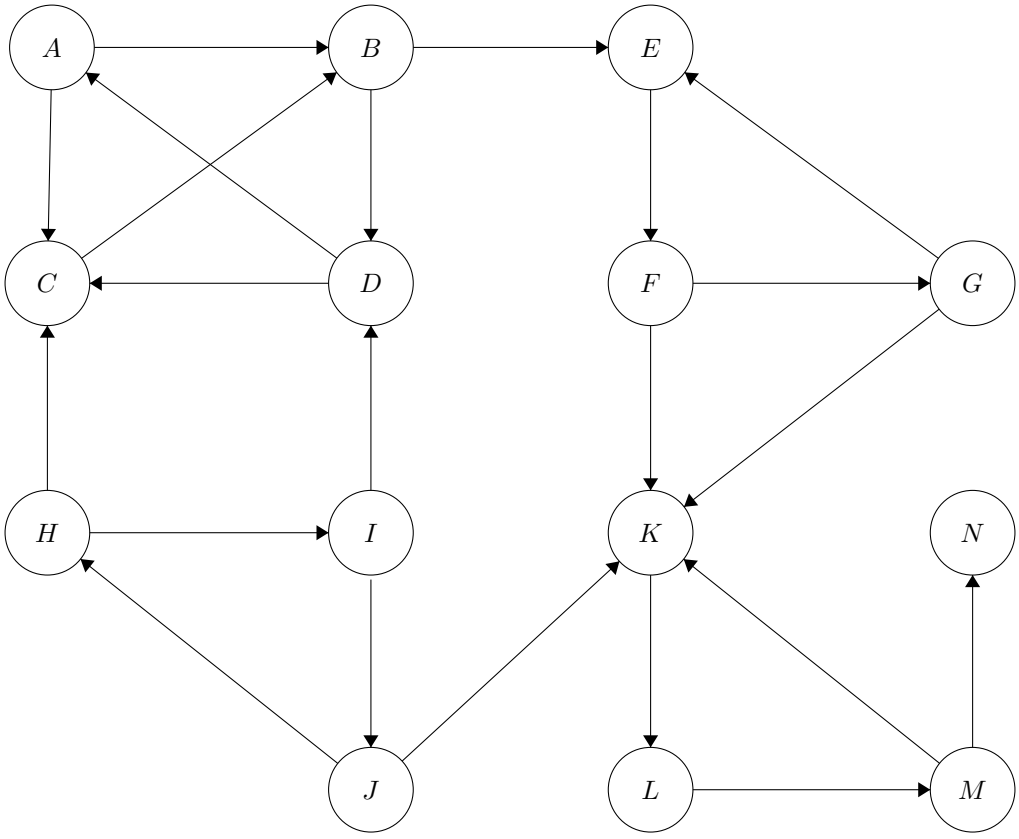
A **strongly connected component**, or SCC, of a directed graph G is a subgraph $G'(V', E')$ in which all vertices in V' can reach each other using only edges in E'. For example, in the following graph:



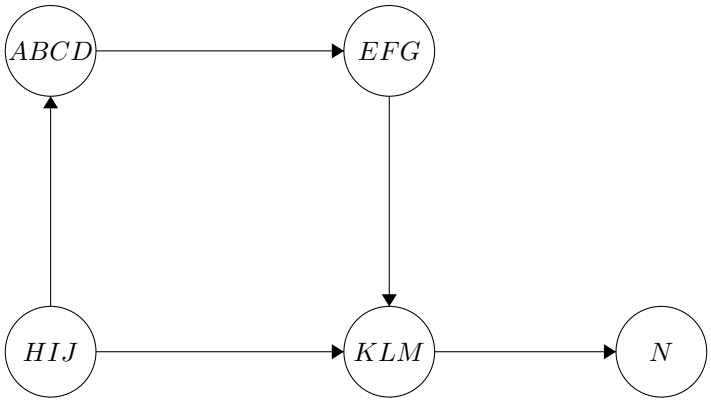
We have two SCCS: $ABCD$, and EFG. Vertices A, B, C, D can all reach each other, and E, F, G can all reach other. While all vertices in $ABCD$ can reach vertices in EFG, vertices in EFG cannot reach $ABCD$ because of the directed edge from B to E.

What if we want to find all of the SCCs in any given directed graph?

To get some intuition for this, let's look at a larger example:



Suppose we abstract out the SCCs in this graph and represent them as their own "vertices". We obtain the following meta-graph:



This is actually a DAG! We can prove by contradiction that this holds true for all SCC meta-graphs. Assume that there exists a cycle in the metagraph. That implies that the vertices in the cycle can access all other vertices, which means that all the original vertices inside the meta-vertex could also access the meta-vertices. This would mean that the SCC would encompass all meta-vertices in the cycle, which is a contradiction.
Therefore, one way we could approach this problem is to identify sink SCCs and remove them from the metagraph one at a time. In this case, we would remove N, then KLM, EFG, and then ABCD.
With that in mind, let's present the following algorithm. Given as input a graph $G(V, E)$, we want to find the SCCs of G.

1. Generate graph G^R, which is the reverse of the graph G. All edges in the graph are reversed.
2. Run DFS on G^R and identify the vertex v with the highest post order. This vertex must be part of the source SCC in G^R, so it must be part of the sink SCC in G!
3. Run `Explore` on G starting from the vertex v and remove all those vertices from G and G^R. Store these vertices as an SCC. Set v to the next highest post order vertex in G^R which has not been removed.
4. Repeat 3 until all vertices have been removed.

In pseudocode form:

```
Strongly Connected Components (G):
    G_R = reverse(G)
    DFS(G_R)
    for each vertex v in V in decreasing order of post numbers from G_R:
        if not visited[v]:
            Explore(G, v)
            mark new visited nodes as new SCC
```

This is actually just a linear time ($\mathcal{O}(|V| + |E|)$) algorithm! This is because steps 1 and 2 takes linear time, and in steps 3-4 we iterate through all vertices and edges once *in total*, since we remove vertices and edges after we mark them as an SCC.

2.5 Shortest Paths

The next two sections involve the concept of edge *weights*, which is a number associated to each edge $e \in E$ which denotes the cost of traversing that edge.

Given a graph $G(V, E)$ and edge weights W, can we find the minimum cost path distance from some vertex s to all other vertices in the graph? We want to minimize the cost of traversing edges from s to v for all vertices $v \in V$. Specifically, given as input $G = (V, E)$, weights W, and a start vertex $s \in V$, we want to output a list of shortest distances $dist(s, v)$ for all $v \in V$.

2.5.1 Dijkstra's Algorithm

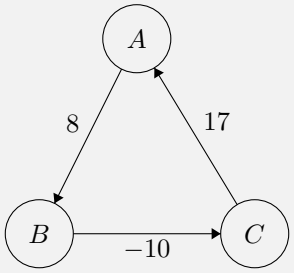
Dijkstra's Algorithm finds the shortest distance between a given node (which is called the "source node") and all other nodes in a graph. Dijkstra's greedily selects nodes with the shortest edge distances and finalizes node distances after traversing the minimum weight edges.

1. Create a list of distances for all nodes initialized to infinity, except for the entry from the source vertex to itself (which is initialized to 0). Add this node to the priority queue with its distance.
2. Once the algorithm pops the minimum distance node, it is marked as visited and its distance is finalized. As it examines the neighbors of the current, the algorithm maintains the track of the currently recognized shortest distance from it to the source node and updates these values if it identifies another shortest path. It then adds all neighbors to the priority queue based on their new distances.
3. In a variant of this algorithm, we can also keep track of a prev list which allows us to tell which nodes are in the shortest path to a node. We can find the shortest path by backtracking through this data structure.
4. This process continues till all the nodes in the graph have been added to the path.
5. We return the shortest distances to all nodes.
6. In the variant of this algorithm that returns a path, a path gets created that connects the source node to all the other nodes following the plausible shortest path to reach each node.

While it seems odd to mark a node as completed once we pop it off the queue, this works because we only look at the minimum weight edges *first* in the queue. As a result, if, for example there was an edge uv with weight 5 and two edges uw and wv with weights 1 and 3, we would explore uw and wv first, before selecting node v from u. This property ensures that we get the shortest path for each node.

The pseudocode of Dijkstra's algorithm is shown below.

Negative Weights
Consider the graph:

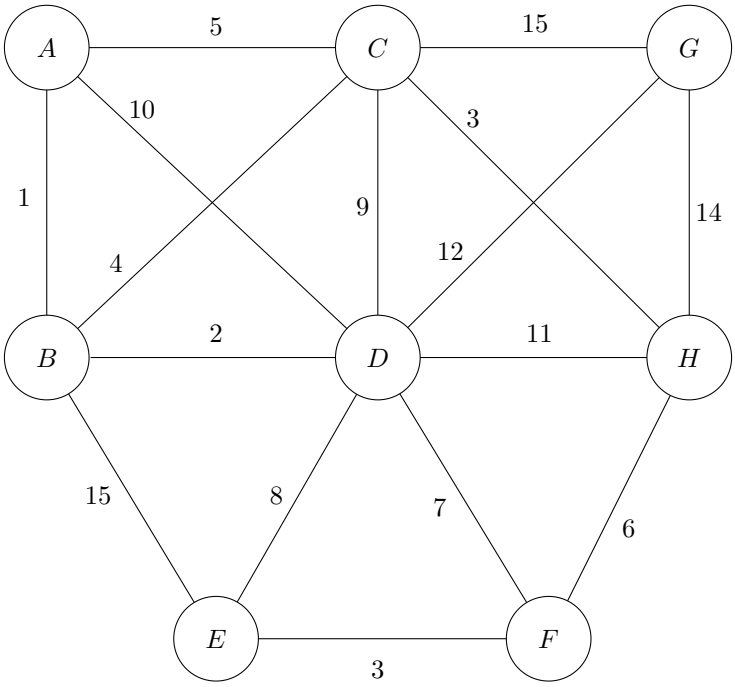


If we run Dijkstra's from A, we would find a distance of 8 to B and 17 to C in the first iteration and then immediately mark both as visited. This is because we assume other weights must be positive, as it wouldn't make sense for there to be a faster way to reach node B if we take a larger weight to C in the beginning. Therefore Dijkstra's will return 8 and 17, which is incorrect. As a result, Dijkstra's algorithm is not guaranteed to work with negative weight edges!

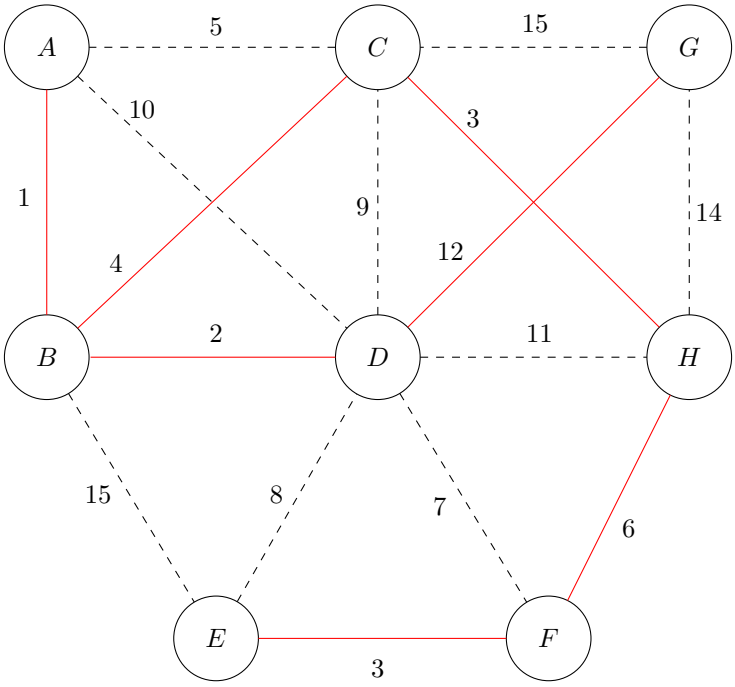
```
function Dijkstra(G, s):
    PQ = PriorityQueue()
    visited = {}
    for each vertex v in V:
        dist[v] = ∞
        prev[v] = null
    dist[s] = 0
    PQ.enqueueWithDist(s, dist[s])
    while PQ is not empty:
        u = PQ.deleteMinFromDist()
        visited[u] = true
        for each neighbor v of u:
            if v not in visited:
                alt = dist[u] + W(u, v)
                if alt < dist[v]
                    dist[v] = alt
                    prev[v] = u
                PQ.enqueueWithDist(v, dist[v])
    return dist, prev
```

2.6 Minimum Spanning Tree

A spanning tree of an undirected graph G is defined as a subset of the edges E such that all vertices V are connected and that there are no cycles. A **minimum spanning tree**, or MST, is a spanning tree in which we aim to minimize the total cost of the edges in the tree. Let's take a look at the following graph:



For the graph above, the minimum spanning tree would be the following:



The algorithm we've covered in class to find MST is Kruskal's Algorithm. Let's review it here:

2.6.1 Kruskal's Algorithm

Kruskal's algorithm works as follows:

1. Sort edges $e_1 < e_2 < e_3 < \cdots < e_m$ by their weights in ascending order.
2. Set minimum spanning tree $T = \{\}$
3. For i in $1 \ldots m$, if $T \cup \{e_i\}$ is acyclic, $T = T \cup \{e_i\}$
4. Return T

Essentially, Kruskal's greedily selects the best edge and checks if it could be added to the MST. Remember that the two requirements for spanning trees is that they cover all vertices in the graph, and that they are acyclic. The important step in this algorithm is checking if the edge e_i causes the MST to contain a cycle. For that, we should introduce the **Union-Find** approach at a high level:

1. Create clusters for each vertex v in G
2. When we want to merge two vertices i and j, add i to j's cluster.
3. To check if an edge (i, j) causes a cycle, check the root of i's cluster and the root of j's cluster. If they are the same, then we cannot add that edge.

Let's view Kruskal's in action. In the example graph above, we would first add edge AB, then edge BD, CH, EF, and BC. All of these edges can be added safely, as they do not cause a cycle. However, the next edge to check would be AC. A and C are already in the same cluster due to the edge BC. Therefore, we do not add the edge AC and continue by adding FH. All of the rest of the edges up until DG cause cycles. Once we add DG, we can stop since they have added 7 edges to our MST, which would have to traverse 8 vertices. There are 8 vertices in our graph, so we are finished.

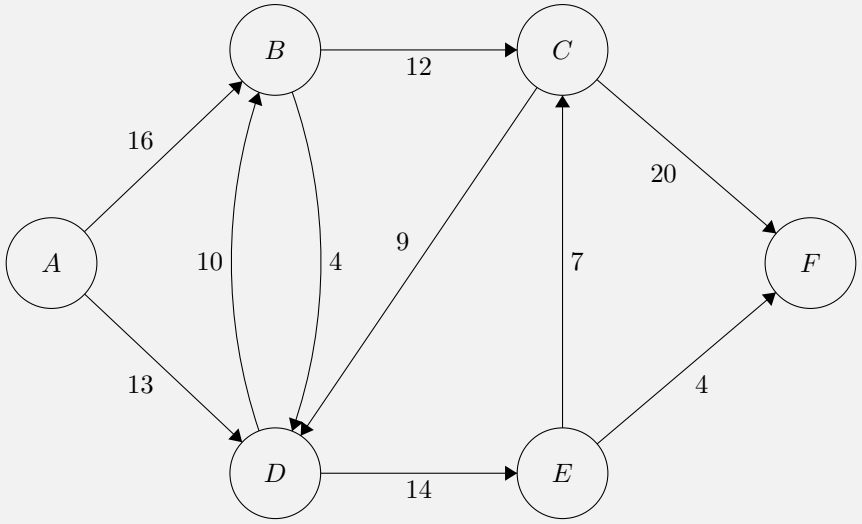
The efficiency of Kruskal's algorithm is $\mathcal{O}(|E| \log |V|)$ or $\mathcal{O}(|E| \log |E|)$. This is because the sorting would take $\mathcal{O}(|E| \log |E|)$, and the union-find approach takes $\mathcal{O}(\log V)$.

Cut Lemma
Why do Prim's and Kruskal's algorithms work? They are greedy algorithms, so they always pick the cheapest edge whenever possible. The reason for this is known as the Cut Lemma, or Cut Property. Given any two subgraphs R and S of G, consider E' to be the set of edges that directly connects nodes in R to nodes in S. If we choose the cheapest edge in E', it is actually guaranteed to be the MST of G.

2.7 Max Flow Min Cut Theorem

The max-flow min-cut theorem is a network flow theorem. It states that the maximum flow through any network from a given source to a given sink is exactly the sum of the edge weights that would totally disconnect the source from the sink upon removal.

Max Flow
Let's remind ourselves of the what the Max-Flow problem represents. Consider the following graph:



Let's say that each vertex represents a distribution center for some product, and the edge weights represent the maximum amount of product that can be sent from one distribution center to another. The problem is this: what is the maximum number of units that be delivered from distribution center A (the source vertex), and F (the sink vertex).

It might be tempting to say the answer is 24 and be done, since we see two inward edges into F that sum up to 24. However, this is not the case, since if we look at vertex C, the maximum flow into C is only 19, since there are two inward edges BC and CE whose weights sum to 19. With that in mind, let's look at the strategy to transport product from A to C.

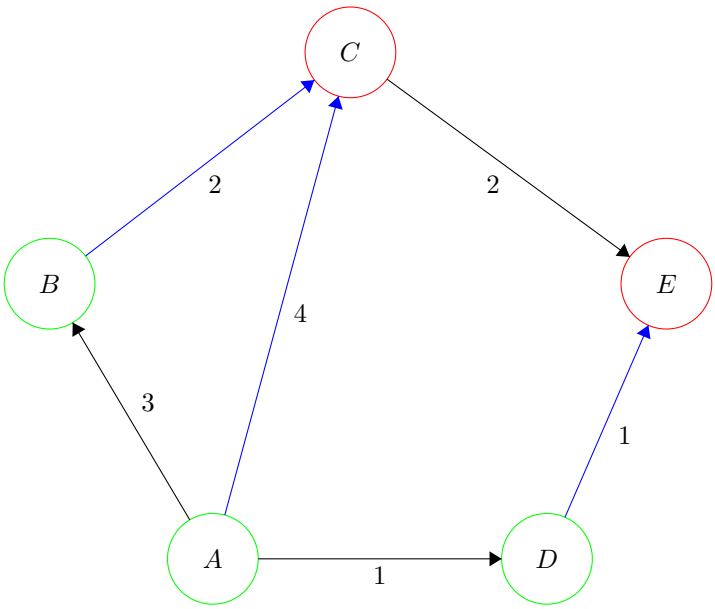
1. We have an infinite supply of product at A, so lets send 16 of those to B and 13 to D.
2. B can send 12 of its inventory to C, and D can send all 13 products in its inventory to E.
3. E sends 7 products up to C, making C's inventory 19, and sends 4 of the remaining products to F.
4. C can send all 19 products to F, so the inventory of F is 23.

As it turns out the maximum flow is in fact 23, since any path we take here will be bottlenecked by the fact that C can receive at most 19 packages, and E can send at most 4 packages. In more complicated graphs it is more difficult to find the maximum flow, and we'll review the algorithm discussed in lecture to solve it.

Suppose we select a source vertex s and a sink vertex t. Then, the Max-Flow Min-Cut Theorem tells is that the **max-flow from s to t is equal to the min-cut necessary to separate s from t.**

Things to keep in mind:

1. The network is a directed, weighted graph.
2. The source is where all of the flow is coming from. All edges that touch the source must be leaving the source. And, there is the sink, the vertex where all of the flow is going. Similarly, all edges touching the sink must be going into the sink.
3. A cut is a partitioning of the network, G, into two disjoint sets of vertices. These sets are called L and R. L is the set that includes the source, and R is the set that includes the sink. The only rule is that the source and the sink cannot be in the same set.
4. A cut has two important properties. The first is the cut-set, which is the set of edges that start in L and end in R. The second is the capacity, which is the sum of the weights of the edges in the cut-set. Look at the following graphic for a visual depiction of these properties.



Our cut partitions L and R are $\{A, B, D\}$ and $\{C, E\}$ respectively. The cut edges are BC, AC, DE whose weights add up to to 7. Therefore, the cut capacity is 7. Is the minimum cut to separate A and E 7, or can we do better?

The **Ford-Fulkerson Algorithm** can be used to find the maximum flow in a graph. Here's the pseudocode:

```
function Ford-Fulkerson Algorithm(Graph G, source s, sink t)
    flow = 0
    for each edge (u, v) in G:
        flow(u, v) = 0
    while there is a path, p, from s to t in residual network Gf:
        residualCapacity(p) = min(residualCapacity(u, v) : for (u, v) in p)
        flow = flow + residualCapacity(p)
        for each edge (u, v) in p:
            if (u, v) is a forward edge:
                flow(u, v) = flow(u, v) + residualCapacity(p)
            else:
                flow(u, v) = flow(u, v) - residualCapacity(p)
    return flow
```

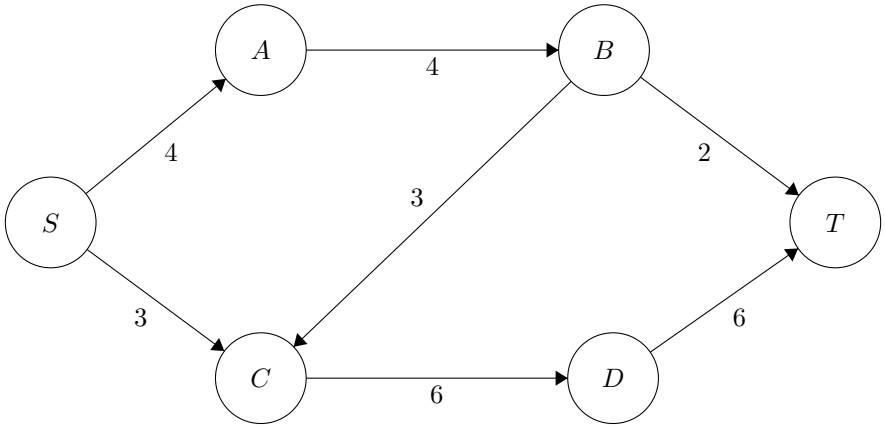
Time complexity: The analysis of Ford-Fulkerson depends heavily on how the augmenting paths are found. The typical method is to use breadth-first search to find the path. If this method is used, Ford-Fulkerson runs in polynomial time.

If all flows are integers, then the while loop of Ford-Fulkerson is run at most $|f^*|$ times, where f^* is the maximum flow. This is because the flow is increased, at worst, by 1 in each iteration.

Finding the augmenting path inside the while loop takes $\mathrm{O}(V + E')$ where E' is the set of edges in the residual graph. This can be simplified to $O(E)$. So, the runtime of Ford-Fulkerson is $O(E \times f*)$. Once again, you'll have to do some analysis based on the specific implementation of this algorithm which you're using to decide its time complexity.

Residual Graphs
Before we look at an example of Ford-Fulkerson on a graph, let's review what residual graphs are. For any edge (u, v), we define the **residual capacity** $c_f(u, v) = c(u, v) - f(u, v)$ where $c(u, v)$ is the weight of the edge (u, v) and $f(u, v)$ is the maximum flow through the edge. We also need to calculate a residual capacity for the reverse edge (v, u). The max-flow min-cut theorem states that flow must be preserved in a network, so we have that $f(v, u) = -f(u, v)$. We use these residual capacities to build a residual graph from the original graph.

Let's take a look at an example case of this algorithm on the following graph:



The first path we find from S to T (which is arbitrarily first) can be SA, AB, BT. We can push at most 2 units from S to T using in this path. We have the new residual capacities:

$$c_f(SA) = c(SA) - f(SA) = 4 - 2 = 2$$

$$c_f(AB) = c(AB) - f(AB) = 4 - 2 = 2$$

$$c_f(BT) = c(BT) - f(BT) = 2 - 2 = 0$$

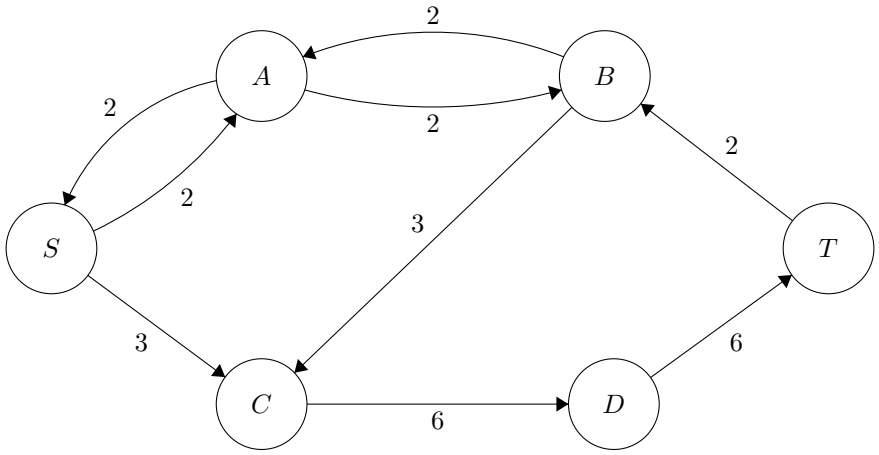
We now calculate the residual capacities of the reverse edges. Since the reverse edges do not exist yet, their capacity c is equal to 0. We have:

$$c_f(AS) = c(AS) + f(SA) = 0 + 2 = 2$$

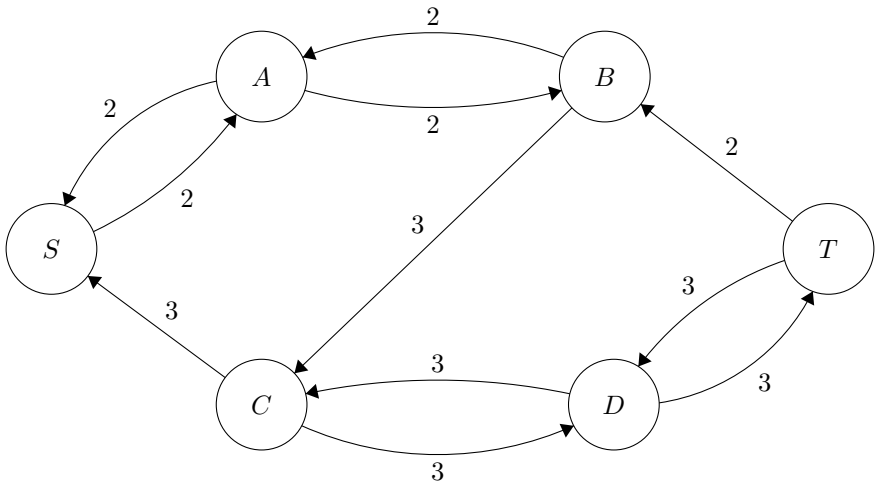
$$c_f(BA) = c(BA) + f(AB) = 0 + 2 = 2$$

$$c_f(TB) = c(TB) + f(BT) = 0 + 2 = 2$$

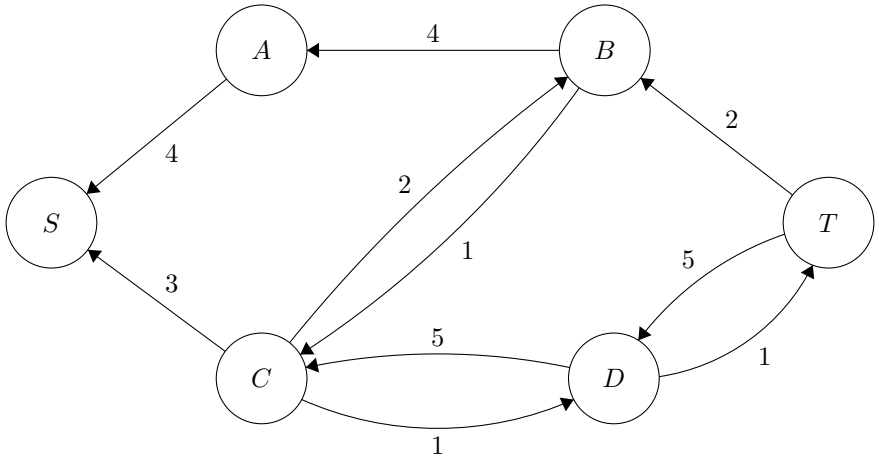
We update the residual graph with all edges with nonzero residual capacities:



Now, we repeat the algorithm and look for a path from S and T. Because ST was erased, the path must end with edge DT. We now have the path SC, CD, DT with a maximum flow of 3. If we update the graph using the same mechanism, we have the new residual graph as follows. As practice, see if you can calculate the new edge weights.



We now have another pass we can traverse: SA, AB, BC, CD, DT with maximum flow of 2. The new graph is as follows:



We now have no paths from S to T, as S is now a sink in the residual graph. We are now done with the algorithm, and the maximum flow is the sum of the weights of the inward edges to S, which is $4 + 3 = \boxed{7}$.

So that's the content you need to know! To best prepare for the exam, review the lectures and ensure you have a good understanding of all the concepts and algorithms we've covered in class. Going through the practice quiz and textbook questions will really help. Feel free to attempt graph textbook problems we haven't listed in the recommended problems section; other problems will definitely be more involved and/or harder, but will really help you be prepared for the quiz. Please ask any questions you have on Piazza or in TA office hours, and we'd be glad to help. Best of luck on the exam :-)!