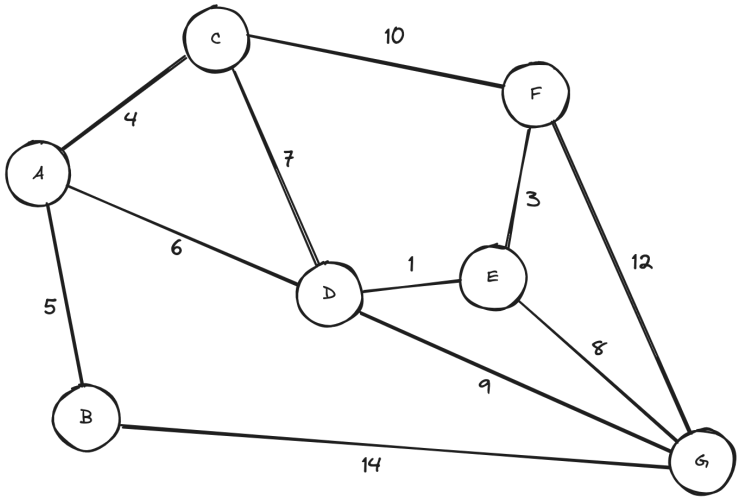
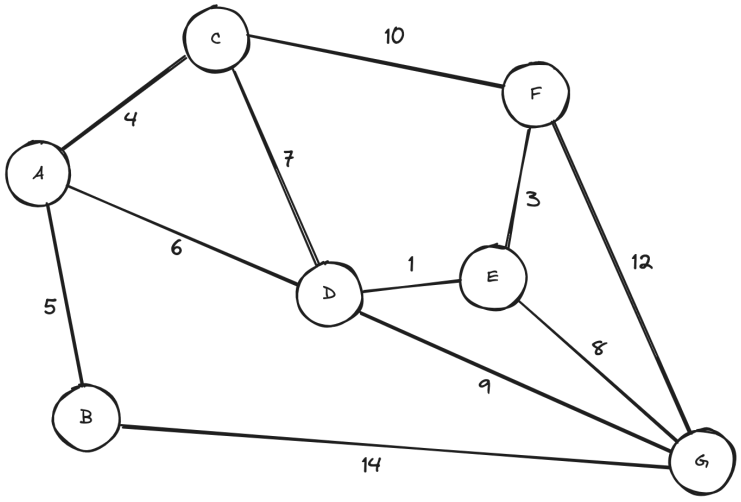
❖ Graph Algorithms

8.1 Minimum Spanning Tree

Suppose we are designing a network to connect n different houses. We can create a graph representing all possible ways to connect the houses. Ultimately, it is sufficient to use $n-1$ of the edges to make sure that there is a path from one house to any other house. We want to find the way to do this using the minimum amount of wire possible. We call a subset of edges that connect all the vertices a **spanning tree**. The weight of a spanning tree is the sum of the weights that form the spanning tree.

Question 121. For the following graph, find a spanning tree in the first image, then find a spanning tree that has a better weight than your first one.





Problem (MST)**.**

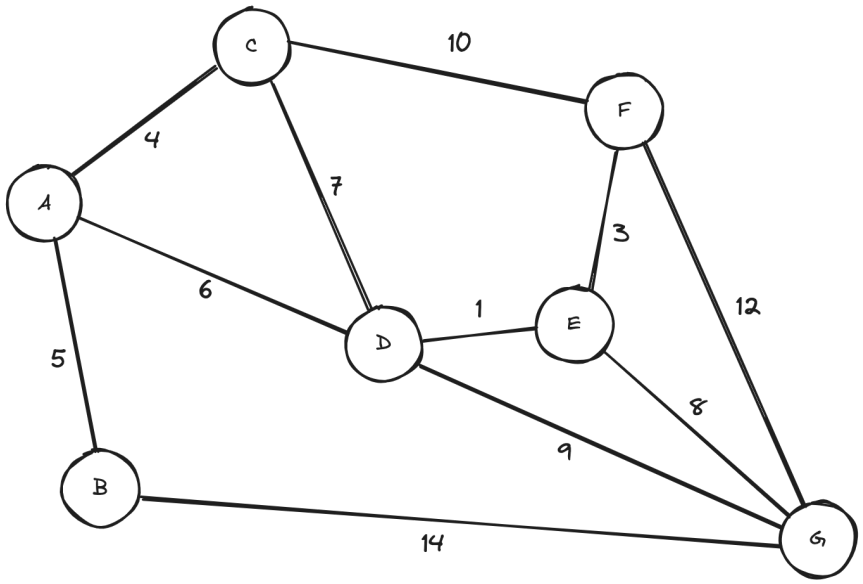
INPUT: A connected, weighted, undirected graph $G = (V, E)$.

OUTPUT: The spanning tree of G with the minimum weight.

Question 122. What are some greedy choices that you might want to try to solve this problem?

It turns out that this problem can be solved by two different very simple greedy algorithms! We will spend more time examining why we can use these algorithms and confidently expect them to output the correct answer.

Question 123. Divide the vertices of the graph into two clusters. What can be said about the smallest weight edge that connect the two clusters?



Theorem 8.1. Let $G = (V, E)$ be a connected, weighted, undirected graph, and let V_1 and V_2 be a partition of the vertices of G into two disjoint nonempty sets. Furthermore, let e be an edge in G with minimum weight from among those with one endpoint in V_1 and the other in V_2. There is a minimum spanning tree T that has e as one of its edges.

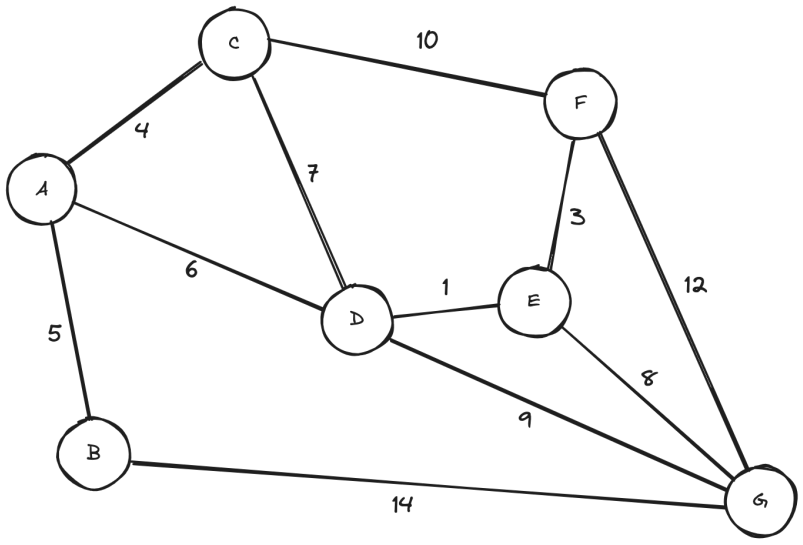
8.1.1 Kruskal's Algorithm

One greedy way to try to build an MST is by continually selecting the lowest weight edge and adding it to our running MST, so long as we don't generate any cycles. This is the idea behind Kruskal's algorithm, and all that remains is to discuss how we may implement this in practice efficiently.

To make use of Theorem 8.1, we will maintain different sets of "clusters" that we will gradually combine until the full MST is formed. To better communicate the idea of the algorithm, let's look at it in pseudocode form.

1. Put each vertex v of G in its own cluster $C(v)$.
2. Create a priority queue Q that stores edges in G, with the edge weights as keys.
3. Initialize a data structure T to store edges in our MST.
4. While T has fewer than $n-1$ edges,
   (a) $(u, v) =$ `Q.remove_min()`
   (b) Let $C(u), C(v)$ be the clusters containing u and v respectively.
   (c) If $C(u) \neq C(v)$, add edge (u, v) to T, and merge the two clusters.
5. Return T

Question 124. Write down the order that the edges get added to the MST if we use Kruskal's algorithm on the following graph.



There are a few implementation decisions that need to be made, which can affect the final running time as well. For the following discussion, let $n = |V|$ and $m = |E|$.

Question 125. What is the runtime of the Kruskal's algorithm?

The simplest way to implement the clusters is to store them as a linked list, and also store a reference to its cluster in each vertex. To merge two clusters, we will opt to move the elements of the smaller cluster into the larger cluster, meaning that the runtime of merging will always be proportional to the size of the smaller cluster. A single vertex will only move to a new cluster at most $\log n$ times. Since there are n vertices, the operations to merge clusters takes at most $O(n \log n)$ time.

Combining the above two, we get the final running time of Kruskal's algorithm to be $O((n + m) \log n)$, which again can be simplified to $O(m \log n)$ because the graph is simple and connected.

8.1.2 Prim-Dijkstra Algorithm

The second MST algorithm goes by a few names, including Prim's algorithm, Prim-Jarnik algorithm, or Prim-Dijkstra. I chose to introduce it as Prim-Dijkstra, since the core idea of this algorithm is the same as what we will encounter when studying Dijkstra's algorithm.

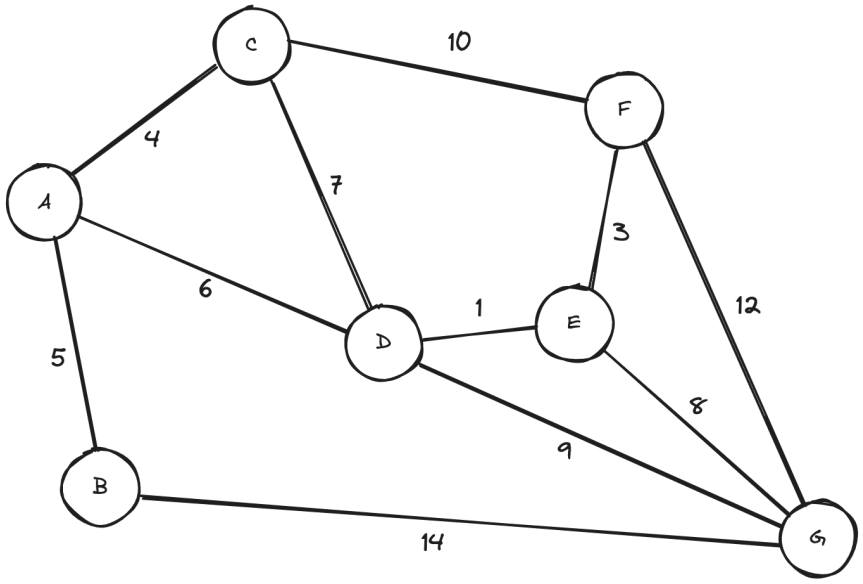
In the Prim-Dijkstra algorithm, we arbitrarily choose a starting vertex, and then from there greedily choose the neighboring edge with the smallest weight to add to our MST. At each step, we will be maintaining a subtree of the final MST, and choosing the smallest weight neighbor that 1. is connected to some vertex in our subtree, and 2. doesn't form a cycle. We will refer to this intermediate solution as a "cloud" C.

It can be unnecessarily costly if we do this check for each potential edge we want to add to our MST, so we will do the following instead. For each vertex u, we maintain a label D_u that stores the weight of the best current edge for joining u to the cloud C. Initially, $D_u = \infty$ for all non starting vertices. Furthermore, each vertex has a u.pred attribute storing its contribution to the MST, and initially set to null.

Here is some pseudocode for how this algorithm works.

1. Pick a random vertex v of G, and set $D_v = 0$.
2. Create a priority queue using the edge as the element and D_u as the key.
3. While the priority queue is not empty,
   (a) Pop the minimum D_u element, and add the edge stored with it to the tree T.
   (b) For each neighbor z of u, update D_z with the edge weight from u to z, and store the edge (u, z).
4. Return the tree T.

Question 126. Demonstrate an example run of Prim-Dijkstra on the below graph, with node A as the source.



v	A	B	C	D	E	F	G
D_v	∞	∞	∞	∞	∞	∞	∞
`v.pred`	null	null	null	null	null	null	null

Question 127. What is the runtime of Prim-Dijkstra?

8.2 Single-source Shortest Path

Often times we want to find a shortest path between two nodes in a graph, which can represent locations on a map or a network. We will use what we learned so far in this class to study some algorithms that are able to accomplish this.

Problem (SSSP)**.**

INPUT: A directed, weighted graph $G = (V, E)$, and a source vertex $s \in V$.

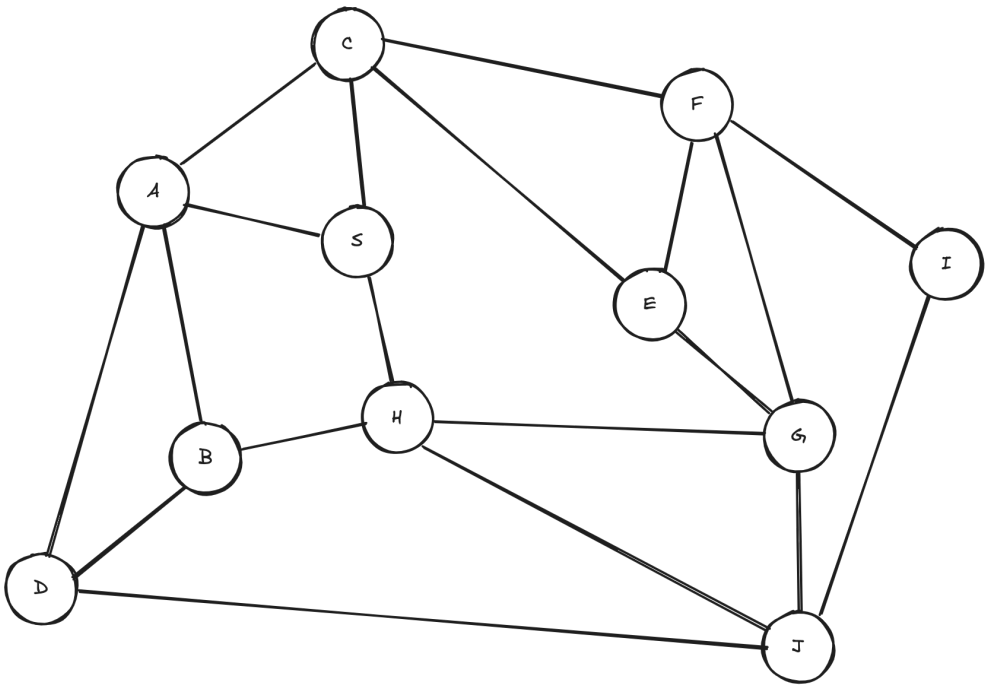
OUTPUT: For each vertex $v \in V$, the shortest path between s and v.

Note that we are trying to solve this for all destinations. It turns out that focusing on a single source, destination pair and studying all destinations from a single source have the same **worst case running time**, so we will opt to study this more general problem. We saw in an earlier class that the shortest path problem exhibits optimal substructure. Many shortest path algorithms exploit this fact, and therefore use dynamic programming or greedy approaches to solve the problem.

As a starting point, let's take a look at what we can do if we have a graph that is not weighted. In this case, a breadth-first search algorithm is sufficient to solve the problem. In breadth-first search, we use a queue to keep track of which vertices we need to visit next.

```
def BFS(G, s):
    # Each vertex v has
    #   - v.dist = distance to s (initially infty)
    #   - v.pred = predecessor of v on shortest path to s (initially null)
    s.dist = 0
    Q = [empty queue]
    Q.enqueue(s)
    while not Q.empty():
        u = Q.dequeue()
        for each neighbor v of u:
            if v.dist = infty:
                v.dist = u.dist + 1
                v.pred = u
                Q.enqueue(v)
```

Question 128. Demonstrate a run of BFS on the following graph.



	S	A	B	C	D	E	F	G	H	I	J
`v.dist`	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
`v.pred`	null	null	null	null	null	null	null	null	null	null	null

Question 129. What is the runtime of BFS?

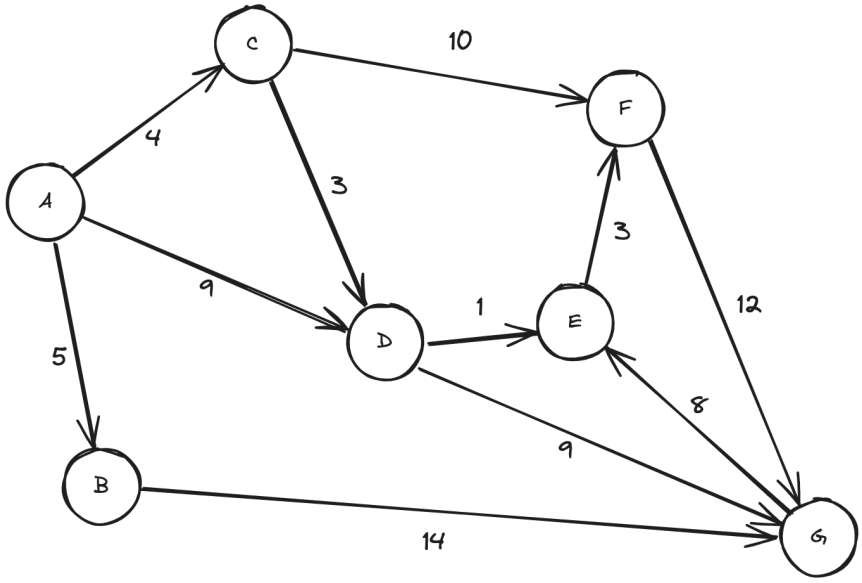
8.2.1 Dijkstra's algorithm

Dijkstra's algorithm allows us to handle graphs with weights, allowing us to find shortest paths in many more realistic scenarios. It is still slightly constrained though in requiring that all the weights be nonnegative. The algorithm works almost in the exact same way as Prim-Dijkstra, with the only difference being that each vertex stores the nearest distance to the *starting point*, rather than to the MST.

Formally, we will apply Dijkstra's algorithm in the case where the input graph is directed and weighted with nonnegative weights.

```
def dijkstra(G, s):
    # Each vertex v has
    #   - v.dist = distance to s (initially infty)
    #   - v.pred = predecessor of v on shortest path to s (initially null)
    s.dist = 0
    Q = [new priority queue with all vertices, sorted by v.dist]
    while not Q.empty():
        u = Q.dequeue()
        for each edge E from u to v:
            dist_through_u = u.dist + E.weight
            if dist_through_u < v.dist:
                v.dist = dist_through_u
                v.pred = u
```

Question 130. Demonstrate an example run of Dijkstra's algorithm on the below graph, with node A as the source.



v	A	B	C	D	E	F	G
D_v	∞	∞	∞	∞	∞	∞	∞
`v.pred`	null	null	null	null	null	null	null

Question 131. What is the runtime of Dijkstra's algorithm?

8.2.2 Bellman Ford

The situation for the SSSP problem becomes considerably trickier when we allow negative weight edges in the graph.

Question 132. Draw a weighted, directed graph that has negative weight edges and state an issue you may run into in this scenario.

Bellman Ford is a dynamic programming algorithm that can solve the problem even for this instance. Importantly, if we run into the above scenario, the algorithm returns False stating that there is no way to have a shortest path between two edges.

Question 133. The subproblem solved in Bellman-Ford is the following: "What is the length of the shortest path from s to u that uses at most i edges?". If no path exists to do this, we will answer ∞. State the following pieces of the dynamic programming algorithm.

Subproblem domain:

Memo table definition:

Goal:

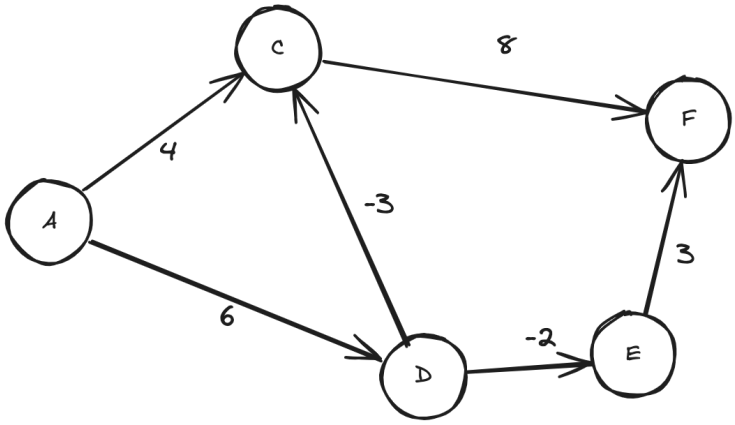
Initial values:

Recurrence:

Here is some python-pseudocode that demonstrates a bottom-up implementation of Bellman-Ford.

```
d = [][]  # Initialize a 2D array to store subproblem solutions
def bellman_ford(G, s):
    d[v][0] = infty  for all v
    d[s][i] = 0  for all i
    for i in range(1, n):
        for each v != s:
            d[v][i] = min((v,x): len(v,x) + d[x][i - 1])
```

Question 134. Demonstrate a run of Bellman-Ford on the below graph.



	A	B	C	D	E
0					
1					
2					
3					
4					

The algorithm we wrote works perfectly for a graph with no negative cycles, but we would like a way to detect when that is the case, rather than hope that we never have to deal with it. We never need to iterate more than $|V| - 1$ times because a path of length any more than that would have to have a cycle. This means the algorithm will terminate with no problem, but we need to make sure that our instance was a valid one.

We can add the following check after line 7 of the pseudocode in the previous page.

```
1     for each edge (u, v):
2         if d[u][n - 1] > d[v][n - 1] + weight(u, v):
3             return False
4     return True
```

A quick lemma about shortest paths:

Lemma 8.2 (Triangle Inequality)**.** Let $G = (V, E)$ be a weighted, directed graph and source vertex s. Then for all edges $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + w(u, v) \tag{25}$$

where $\delta(u, v)$ is the total weight of the shortest path from u to v.

Question 135. Use the above lemma to show that our condition for checking for loops works correctly.

```
1  d = [][]  # Initialize a 2D array to store subproblem solutions
2  def bellman_ford(G, s):
3      d[v][0] = infty  for all v
4      d[s][i] = 0  for all i
5      for i in range(1, n):
6          for each v != s:
7              d[v][i] = min((v,x): len(v,x) + d[x][i - 1])
8      for each edge (u, v):
9          if d[u][n - 1] > d[v][n - 1] + weight(u, v):
10             return False
11     return True
```

Question 136. What is the runtime of the inner loop in lines 6-7?

Question 137. What is the total runtime of Bellman-Ford?

8.3 All Pairs Shortest Path: Floyd-Warshall

Given some network, we define the **diameter** of the network to be the longest of all shortest paths. If we are thinking of a communication network, this effectively characterizes the longest possible transit time of a message in the network. To determine something like this, we need to find the shortest path between **all** pairs of nodes in the network.

Problem (APSP)**.**

INPUT: A weighted, directed graph $G = (V, E)$.

OUTPUT: A table of size $V \times V$ that in index (u, v) stores the weight of the shortest path from u to v.

A simple way to start solving this problem is to simply apply the single-source shortest path algorithm starting from each vertex, effectively running it $|V|$ times.

Question 138. Suppose we had a graph with no negative weights. What's the total runtime if we used Dijkstra's algorithm? What if the graph did have negative weights, so we used Bellman-Ford instead?

Recall that shortest paths have the optimal substructure property, meaning that if I had a shortest path going from i to j through some other vertex k, the same path from k to j is the shortest path between those two vertices. This means that if we were to run something like Dijkstra's or Bellman-Ford n different times, we would be recomputing a lot of overlapping shortest paths! Dynamic programming is the strategy we know that shines in situations with many overlapping subproblems like this, and that is exactly what the Floyd-Warshall algorithm is.

Let's take the path decomposition discussion from above, and try to exploit it for speeding up our computation. Floyd-Warshall is often discussed as a bottom-up dynamic programming approach, and we will do the same. What this means is that we will be computing the smallest subproblems that exist in our graph first, and then combine these results to solve larger problems.

Pick some numbering of your vertices $\{1, \ldots, n\}$, and take a subset of the first k of them $\{1, 2, \ldots, k\}$. Now for each pair of vertices $i, j \in V$, we will consider the shortest path p from i to j that only uses vertices from $\{1, 2, \ldots, k\}$. Let's focus on vertex k from this set. There are two cases that we are interested in:

- If k **is not** a part the path p from i to j, this means that the shortest path between these two using vertices $\{1, 2, \ldots, k\}$ actually doesn't need vertex k. In other words, the solution in this case is the same as what we would have if we only used $\{1, 2, \ldots, k-1\}$.

- If k **is** a part the path p from i to j, this means we can decompose p into two paths p_1 and p_2, from i to k and from k to j respectively that only use vertices in $\{1, 2, \ldots, k\}$. Since both of these paths include k, this means that they actually only use vertices in $\{1, 2, \ldots, k-1\}$.

In both cases, we can build up the solution by using solutions that only involve vertices in $\{1, 2, \ldots, k-1\}$. This will justify Floyd-Warshall's dynamic programming approach that starts from $k = 1$, and builds all the way up to $k = n$.

Question 139. The subproblem solved in Floyd-Warshall is the following: "What is the length of the shortest path from i to j that only uses vertices $\{1, 2, \ldots, k\}$?". If no path exists to do this, we will answer ∞. State the following pieces of the dynamic programming algorithm.

Subproblem domain:

Memo table definition:

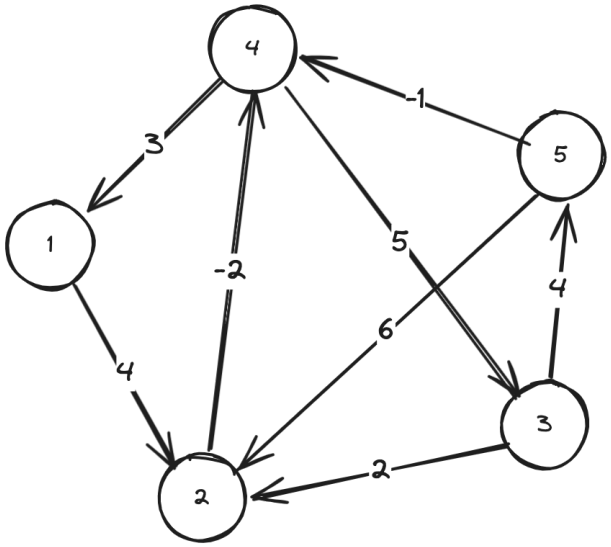
Goal:

Initial values:

Recurrence:

As is the case in most dynamic programming algorithms, once the problem solving part is done they can be implemented quite simply. It is standard in this algorithm for the input graph to be represented by what is called an adjacency matrix, instead of a set of vertices and edges. The matrix stores at index i, j, the weight of the edge from i to j if it exists, and ∞ if there is no such edge.

Question 140. Fill out the adjacency matrix for the following graph. We will let an empty cell represent ∞.



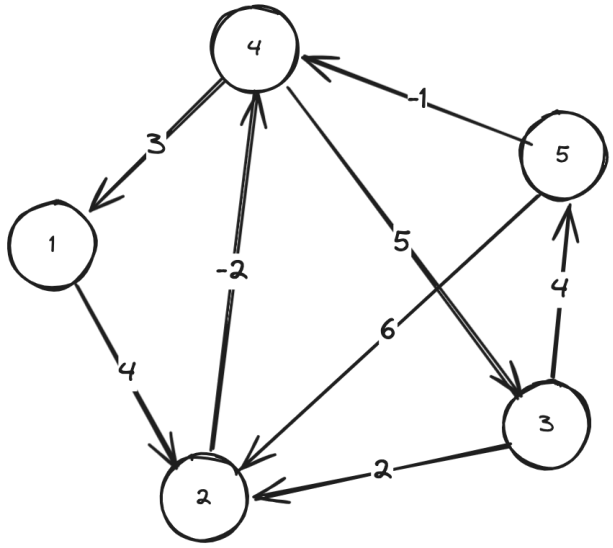
D[2]	1	2	3	4	5
1					
2					
3					
4					
5					

The following is some python-pseudocode for the Floyd-Warshall algorithm.

```
def floyd-warshall(W, n):
    # W is the adjacency matrix representing G
    D = []  # Length n + 1 array, each index storing an n by n matrix
    D[0] = W
    for k in range(1, n + 1):
        D[k] = [new n by n matrix]
        for i in range(1, n + 1):
            for j in range(1, n + 1):
                D[k][i][j] = min(
                    D[k - 1][i][j],  # Using k doesn't improve the shortest path.
                    D[k - 1][i][k] + D[k - 1][k][j] # Glue together solutions.
                )
    return D[n]
```

Question 141. Simulate a run of the algorithm for $k = 3$. On the left is the matrix D[2], fill out the empty cells in the matrix D[3]. For reference, the recurrence is:

$$D[k][i][j] = \min\{D[k-1][i][j], D[k-1][i][k] + D[k-1][k][j]\} \tag{26}$$



D[2]	1	2	3	4	5
1	0	4	∞	2	∞
2	∞	0	∞	-2	∞
3	∞	2	0	0	4
4	3	7	5	0	∞
5	∞	6	∞	-1	0

D[3]	1	2	3	4	5
1	0	4	∞		
2		0	∞	-2	
3	∞	2	0		4
4	3	7	5	0	
5		6	∞	-1	0

Question 142. What is the runtime of Floyd-Warshall?

8.4 Maximum-Flow

Imagine a material coursing through a system from a **source** where the material is produced, to a **sink** where it is consumed. Intuitively, we will use the term "flow" to capture the rate at which this material moves. This is a useful abstraction to model many problems including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks.

We will use vertices to represent junctions in the network, and edges to represent some capacity at which the material can travel between two junctions. The maximum-flow problem is to compute the greatest rate for moving material from the source to the sink without violating any capacity constraints.

More formally, a **flow network** is a directed, weighted graph $G = (V, E)$ with non-negative weights (representing capacities). Every flow network has two key vertices, the source s and the sink t. We will assume that there is always a path from s to t, and every vertex in our graph lies along some path between the two vertices.

Given a flow network, a **flow** is defined as a function from each edge to a real value satisfying the following two constraints:

- **Capacity constraint**: The flow along an edge cannot exceed the capacity:

$$0 \leq f(u, v) \leq c(u, v) \tag{27}$$

- **Flow conservation**: For all non sink or source vertices, the total flow **into** a vertex must equal the total flow **out of** that vertex.

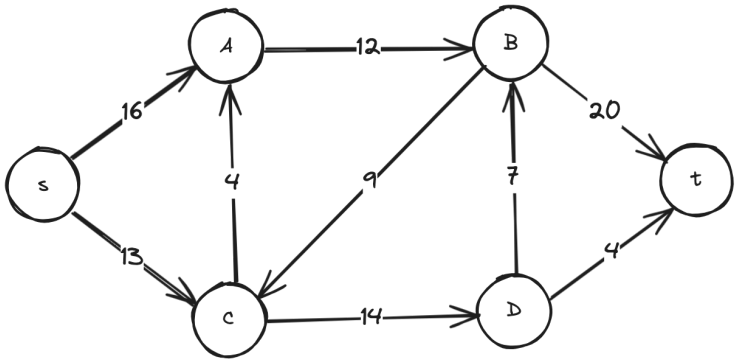
$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v). \tag{28}$$

Finally, the **value** $|f|$ of a flow f is defined as

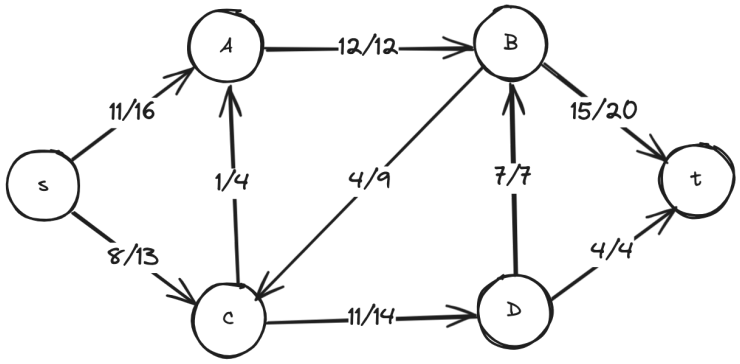
$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s). \tag{29}$$

That is, the total flow *leaving* the source minus the total flow *entering* the source.

The discussion so far has been quite abstract. Let's see all of it in action through an example. We will focus on the following graph:



To the above graph, I claim that the following is a **flow**. The value to the left of the slash is the flow we are assigning to the edge.



Question 143. Verify that the flow f in the above figure satisfy the capacity constraint and flow conservation.

Question 144. What is the value of the flow f from above?

8.4.1 The Ford-Fulkerson Method

The Ford-Fulkerson method is more often referred to as a method rather than an algorithm, since it abstracts away many design choices that can improve the runtime for different problem settings. Intuitively, it starts from a starting flow of 0 throughout the network, then uses the existing constraints to iteratively increase the flow until it cannot be increased anymore. Here is some pseudocode capturing the idea behind the method.

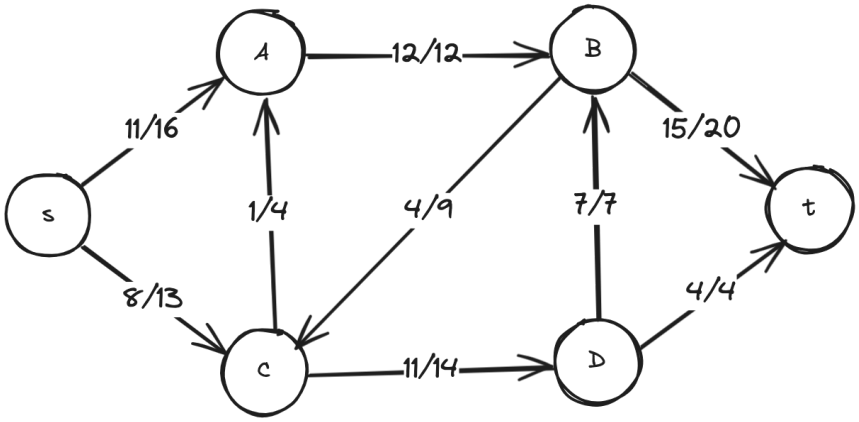
1. Initialize flow f to 0
2. **while** there exists an augmenting path p in the residual network G_f:
   (a) augment flow f along p
3. **return** f

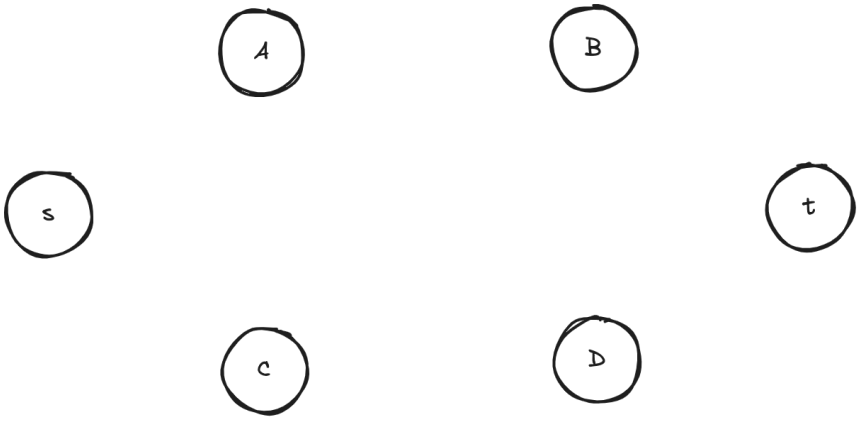
There are some unfamiliar words that showed up here, so we'll spend the remainder of the time defining what they are. Once we understand these, the algorithm is quite simple. As you will see, this is another greedy algorithm that simply makes the greedy choice at each iteration of the loop until the algorithm is done.

Residual Network

A residual network is a graph we can define over our flow f that essentially keeps track of the changes we are allowed to make in our flow network. For each edge in our flow f, we create two directed edges. One edge in the same direction storing the amount of flow that we can increase, and another in the opposite direction storing the amount of flow that we could retract. We will call the graph that we make using these changes the **residual network** G_f.

Question 145. Draw the residual network G_f for the flow we created earlier, drawn here again for convenience.

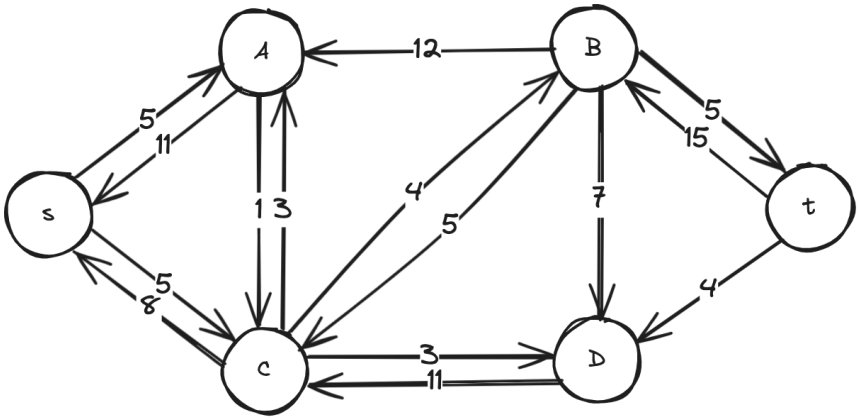


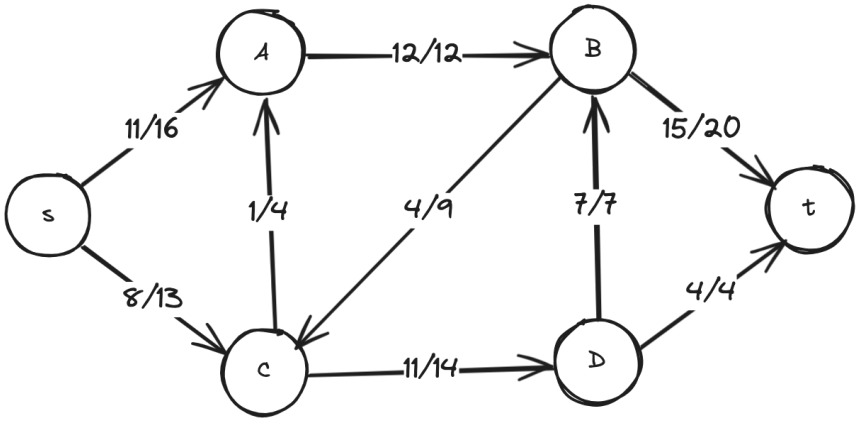


Augmenting Paths

Given a flow network $G = (V, E)$ and a flow f, an **augmenting path** p is a simple path from s to t in the **residual network** G_f. If an augmenting path can be found, we simply have to find the smallest capacity edge on this path, and adjust our flow so that we maximize the capacity usage on that edge.

Question 146. Find an augmenting path in the following residual network. Use that augmenting path to increase the flow in our flow network.





Let's take a look at a basic implementation of Ford-Fulkerson to analyze the running time.

```
def ford_fulkerson(G, s, t):
    f[][] is the table that will store the flow
    for each edge (u, v) in G.E:
        f[u][v] = 0
    while there exists a path p from s to t in Gf:
        cap = minimum capacity along p in Gf
        for each edge (u, v) on the path p:
            if (u, v) in G.E:
                f[u][v] = f[u][v] + cap
            else:
                f[v][u] = f[v][u] - cap
    return f
```

Question 147. What is the runtime of the above implementation of Ford-Fulkerson?