CSCI 3104 Algorithms- Lecture Notes

Michael Levet

June 14, 2021

Contents

1 Proof by Induction

In this section, we recall the proof by induction technique. Induction is particularly useful in proving theorems, where properties of smaller or earlier cases imply that similar properties hold for subsequent cases. In order for the theorem to be true, we need to show that there exist initial or minimal objects that satisfy the desired properties. Having initial objects that satisfy the desired properties serves as the starting point for our chain of implications. Showing that this chain of implications exists is effectively what an inductive proof does.

Intuitively, we view the statements as a sequence of dominos. Proving the necessary base cases knocks (i.e., proves true) the subsequent dominos (statements). It is inescapable that all the statements are knocked down; thus, the theorem is proven true.

Any inductive proof has three components: the base case(s), the inductive hypothesis, and the inductive step.

- **Base Case(s):** We first establish that the minimal case(s) satisfy our desired properties.
- **Inductive Hypothesis:** Recall that our goal is to show that if earlier cases satisfy our desired properties, then so do subsequent cases. That is, we have an *if ..., then ...* statement. The inductive hypothesis is the *if* part. Here, we assume that all smaller cases hold.
- **Inductive Step:** The inductive step is where we use the inductive hypothesis (the *if* part of our *if ..., then ...* statement) to show that a subsequent case also holds. That is, we prove that the *then* part of our *if ..., then ...* statement holds.

We illustrate this proof technique with the following example.

Proposition 1. For all $n \in \mathbb{N}$, we have:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

Proof. We prove this theorem by induction on $n \in \mathbb{N}$.

- **Base Case.** Our first step is to verify the base case: $n = 0$. In this case, we have $\sum_{i=0}^{n} i = 0$. Note as well that $\frac{0 \cdot 1}{2} = 0$. Thus, the proposition holds when $n = 0$.
- **Inductive Hypothesis.** Fix $k \geq 0$, and suppose that:

$$\sum_{i=0}^{k} i = \frac{k(k+1)}{2}.$$

- **Inductive Step.** We will use the assumption that:

$$\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$$

  to show that:

$$\sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}.$$

  We have the following:

$$\begin{aligned}
\sum_{i=0}^{k+1} i &= (k+1) + \sum_{i=0}^{k} i && (1)\\
&= (k+1) + \frac{k(k+1)}{2} && (2)\\
&= \frac{2(k+1) + k(k+1)}{2} && (3)\\
&= \frac{(k+1)(k+2)}{2}. && (4)
\end{aligned}$$

  Here, we applied the inductive hypothesis to $\sum_{i=0}^{k} i$, which yielded the expression on line (2).

The result follows by induction. $\square$

Remark 2. We also stress the verbiage for the inductive hypothesis. Notice in the proof of Proposition 1, that our verbiage was:

> Fix $k \geq 0$, and suppose that:
> $$\sum_{i=0}^{k} i = \frac{k(k+1)}{2}.$$

This is analogous to writing a method or function, which accepts a parameter `int k`. For our inductive step, we argue about this parameter `k`. As `k` was arbitrary, the argument applies to any specific value of `k` we want. This is analogous to writing code that works for any value of `k` we provide when invoking our method.

In contrast, a common mistake when writing an inductive hypothesis is to assume the entire theorem is true. This is called *begging the question*. Here is an example of such an **incorrect** inductive hypothesis:

> Suppose that for all $k \geq 0$:
> $$\sum_{i=0}^{k} i = \frac{k(k+1)}{2}.$$

Note that we are trying to prove the desired equation holds for all $k \geq 0$. So we cannot assume this holds for all k. Instead, we fix $k \geq$ our largest base case and assume the proposition holds for our fixed k. In the context of Proposition 1, our largest base case is 0. So we fix $k \geq 0$ in the inductive hypothesis.

Remark 3. Observe that in the inductive step for the proof of Proposition 1, we started with

$$\sum_{i=0}^{k+1} i$$

and manipulated this expression to obtain $(k+1)(k+2)/2$. We did **not** manipulate both sides of the equation at the same time. When trying to establish numerical equalities or inequalities, start with one side of the equation or inequality. Then work to manipulate that one side to obtain the expression on the other side. It is poor practice to manipulate both sides of the inequality at the same time, as it often leads to making the subtle yet fatal mistake of begging the question.

We now examine a second example of proof by induction.

Proposition 4. Fix $c > -1$ to be a constant. For each $n \in \mathbb{N}$, we have that $(1+c)^n \geq 1 + nc$.

Proof. The proof is by induction on $n \in \mathbb{N}$.

- **Base Case.** Consider the base case of $n = 0$. So we have $(1+c)^n = 1 \geq 1 + 0c = 1$. So the proposition holds at $n = 0$.
- **Inductive Hypothesis.** Fix $k \geq 0$, and suppose that $(1+c)^k \geq 1 + kc$.
- **Inductive Step.** We will use the assumption that $(1+c)^k \geq 1 + kc$ in order to show that $(1+c)^{k+1} \geq 1 + (k+1)c$. We have that:

$$\begin{aligned}
(1+c)^{k+1} &= (1+c)^k(1+c) && (5)\\
&\geq (1+kc)(1+c) && (6)\\
&= 1 + (k+1)c + kc^2 && (7)\\
&\geq 1 + (k+1)c. && (8)
\end{aligned}$$

  Here, line (6) follows by applying the inductive hypothesis to $(1+c)^k$. Now we obtain the expression on line (7) by expanding the expression on line (6). Finally, we note that as $c > -1$, $c^2 \geq 0$. So $kc^2 \geq 0$. This yields the inequality on line (8).

The result follows by induction. $\square$

In the proofs of Proposition 1 and Proposition 4, we have only had a single base case. Additionally, the kth case implied the $(k+1)$st case for both propositions. This is known as *weak induction*. In order to prove certain theorems, we may need to verify multiple base cases and use multiple prior cases to prove a subsequent case. This is known as *strong induction*. Surprisingly, weak and strong induction are equally powerful. In practice, it may be easier to use strong induction, while weak induction may be clunky to use.

We next look at an example of where strong induction is helpful.

Proposition 5. Let $f_0 = 0, f_1 = 1$; and for each natural number $n \geq 2$, let $f_n = f_{n-1} + f_{n-2}$. We have $f_n \leq 2^n$ for all $n \in \mathbb{N}$.

Proof. The proof is by induction on $n \in \mathbb{N}$.

- **Base Cases:** We have two base cases in our recurrence: $n = 0$, and $n = 1$. So we verify that $f_0 \leq 2^0$ and $f_1 \leq 2^1$. Observe that $f_0 = 0 \leq 2^0 = 1$. Similarly, $f_1 = 1 \leq 2^1 = 2$. So our base cases of $n = 0, 1$ hold.
- **Inductive Hypothesis:** Fix $k \geq 1$; and suppose that for all $i \in \{0, \ldots, k\}$, $f_i \leq 2^i$.
- **Inductive Step:** Using the assumption that $f_i \leq 2^i$ for all $i \in \{0, \ldots, k\}$, we will show that $f_{k+1} \leq 2^{k+1}$. We have that:

$$\begin{aligned} f_{k+1} &= f_k + f_{k-1} && (9) \\ &\leq 2^k + 2^{k-1} && (10) \\ &= 2^k \left(1 + \frac{1}{2}\right) && (11) \\ &\leq 2^k \cdot 2 = 2^{k+1}. && (12) \end{aligned}$$

  Here, line (11) follows by applying the inductive hypothesis to obtain that $f_k \leq 2^k$ and $f_{k-1} \leq 2^{k-1}$.

The result follows by induction. $\square$

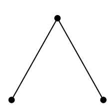
Up to this point, we have only proven theorems regarding numerical equalities or inequalities. From the perspective of Algorithms, we will want to use induction to prove that our Algorithms are correct or to prove theorems about underlying structures. We illustrate this technique of inductive proofs on structures (also known as *structural induction*) by proving a result about rooted binary trees. To this end, we begin with the following definition.

Definition 6. Let $d \in \mathbb{N}$. The *complete, balanced binary tree* of depth d, denoted $\mathcal{T}(d)$, is defined as follows.

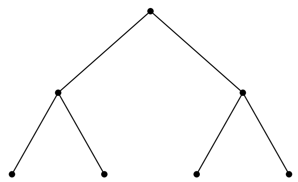
- $\mathcal{T}(0)$ is a single vertex.
- For $d > 0$, $\mathcal{T}(d)$ is obtained by starting with a single vertex and setting both of its children to be copies of $\mathcal{T}(d-1)$.

Example 7. We provide illustrations for $\mathcal{T}(0), \mathcal{T}(1)$, and $\mathcal{T}(2)$ below.

$\mathcal{T}(0)$.



$\mathcal{T}(1)$.



$\mathcal{T}(2)$.

Proposition 8. The tree $\mathcal{T}(d)$ has $2^d - 1$ non-leaf nodes.

Proof. The proof is by induction on $d \in \mathbb{N}$, the depth of the tree.

- **Base Case:** We consider the base case of $d = 0$. Note that $\mathcal{T}(0)$ consists precisely of a single vertex, which is a leaf node. So $\mathcal{T}(0)$ has $2^0 = 1$ leaf node. Thus, $\mathcal{T}(0)$ has $0 = 2^0 - 1$ non-leaf nodes, as desired.
- **Inductive Hypothesis:** Fix $d \geq 0$, and suppose that $\mathcal{T}(d)$ has $2^d - 1$ non-leaf nodes.
- **Inductive Step:** We will use the assumption that $\mathcal{T}(d)$ has $2^d - 1$ non-leaf nodes to show that $\mathcal{T}(d+1)$ has $2^{d+1} - 1$ non-leaf nodes. By construction, $\mathcal{T}(d+1)$ consists of a single root node v, where each of v's two children are copies of $\mathcal{T}(d)$. By the inductive hypothesis, both the left and right copies of $\mathcal{T}(d)$ have $2^d - 1$ non-leaf nodes. This accounts for $2(2^d - 1)$ non-leaf nodes in $\mathcal{T}(d+1)$. Note that v is also a non-leaf node. So $\mathcal{T}(d+1)$ has:

$$\begin{aligned} 2(2^d - 1) + 1 &= 2^{d+1} - 2 + 1 \\ &= 2^{d+1} - 1 \end{aligned}$$

  non-leaf nodes, as desired.

The result follows by induction. □

Remark 9. In the inductive step of Proposition 8, observe that we used the structure of $\mathcal{T}(d+1)$ to obtain the number of non-leaf nodes. Precisely, we used the fact that $\mathcal{T}(d+1)$ is constructed by taking a root node v and setting each of its children to be copies of $\mathcal{T}(d)$. We then used the inductive hypothesis to obtain the number of non-leaf nodes in the copies of $\mathcal{T}(d)$. That is, we had to use the construction of $\mathcal{T}(d+1)$ to obtain the initial count of $2(2^d - 1) + 1$ non-leaf nodes.

A common and significant mistake would be to simply use algebraic manipulations to show that $2(2^d - 1) + 1 = 2^{d+1} - 1$, without first showing that $\mathcal{T}(d+1)$ has $2(2^d - 1) + 1$ non-leaf nodes.

1.1 Supplemental Reading

For more resources on proof by induction, we recommend Richard Hammack's *Book of Proof* (Chapter 10) [Ham20, Ch. 10], as well as Joe Fields' *A Gentle Introduction to the Art of Mathematics* (Chapter 5) [Fie15, Ch. 5].

2 Graph Traversals

In this section, we examine graph traversal algorithms. Intuitively, a graph traversal takes as input a specified source vertex, which we call s, and attempts to visit the remaining vertices in the graph. As a graph may have cycles, we need to take care not to revisit nodes, so as to avoid looping indefinitely. To this end, we mark vertices as visited once they have been evaluated. A graph traversal then only examines the unvisited neighbors of the current vertex being considered.

We begin with the depth-first and breadth-first traversal algorithms. The depth-first traversal algorithm has a myriad of applications, such as finding connected components on a graph, testing whether a graph is planar, topological sorting, and exploring mazes. We may discuss some of these applications, such as topological sorting, later in the course. Other applications, such as generating and solving mazes, appear in subsequent courses such as Artificial Intelligence.

The breadth-first search algorithm similarly has a number of applications. For our purposes, the key application of the breadth-first traversal is that it correctly finds shortest paths in unweighted graphs. For weighted graphs, the breadth-first traversal fails. This motivates Dijkstra's algorithm.

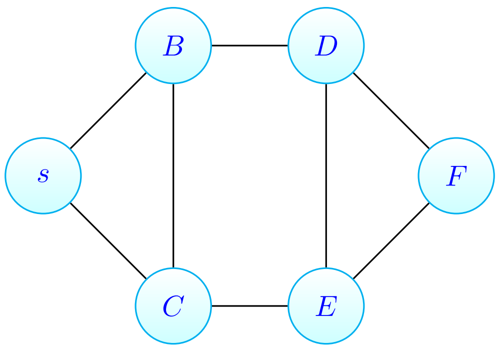
2.1 Depth-First Traversal

The Depth-First Traversal algorithm takes as input a graph G, together with a source vertex $s \in V(G)$. We recursively traverse the unvisited neighbors of s. Effectively, DFS places the edges of s on to a stack. It then pops off the first edge sv and recursively examines the neighbor v. The algorithm terminates when there are no more unvisited neighbors of s. We include pseudo-code below.

Algorithm 1 Depth-First Traversal

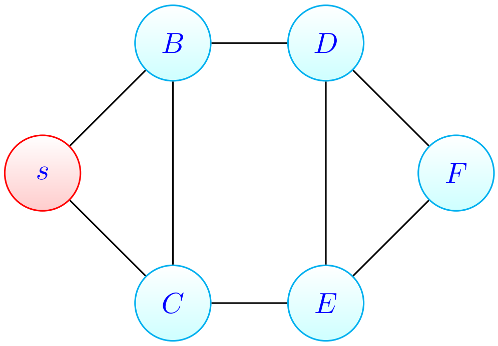
```
procedure DFS(Graph G, Vertex s)
    s.visited ← true
    for each unvisited neighbor v of s do
        DFS(G, v)
```

Example 10. We consider an example of the Depth-First Traversal procedure on the following graph G.

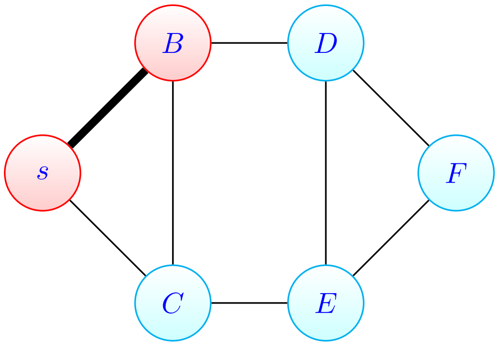


Suppose we invoke $\texttt{DFS}(G, s)$. For the purpose of this example, we examine the unvisited neighbors of the current vertex. Note that one could alternatively choose to order the vertices in a different way, which would result in visiting the vertices in a different order. We also use red nodes to denote visited vertices. The algorithm does the following.

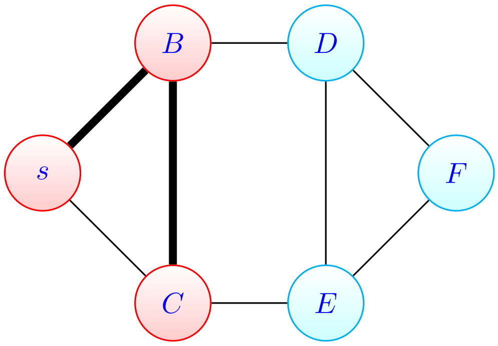
1. We first set s.visited := $\texttt{true}$.



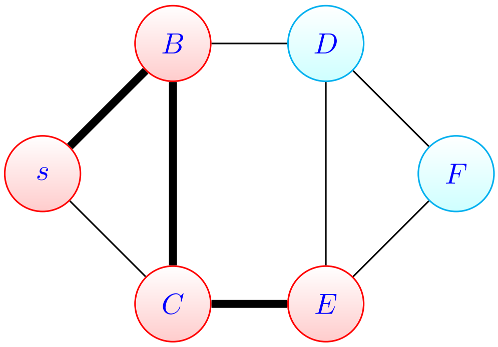
2. Both neighbors of s, B and C, are unvisited. So we recurse and visit B next, invoking $\texttt{DFS}(G, B)$.



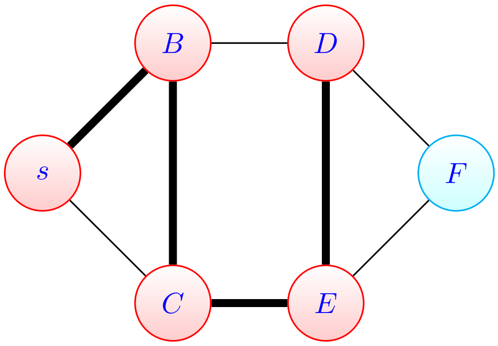
3. Now the two unvisited neighbors of B are C and D. We visit C next, as we are choosing to order the neighbors alphabetically. We invoke $\texttt{DFS}(G, C)$.



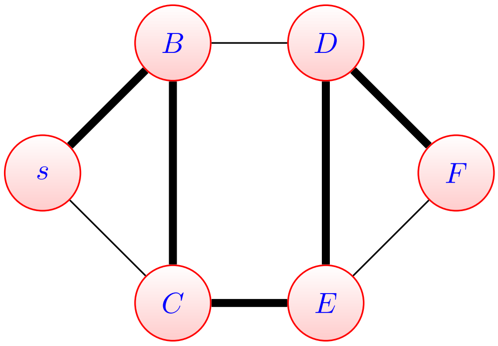
4. The only unvisited neighbor of C is E, so we next invoke $\texttt{DFS}(G, E)$.



5. The unvisited neighbors of E are D and F. We visit D next, as D comes before F alphabetically. We invoke $\texttt{DFS}(G, D)$.



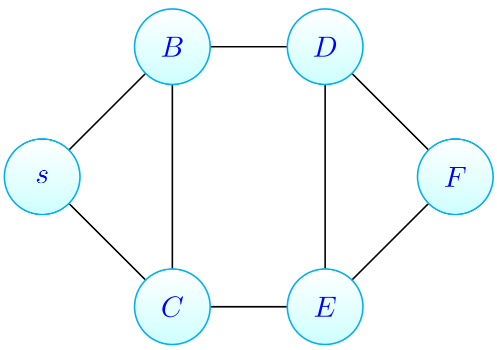
6. The only unvisited neighbor of D is F. So we invoke $\texttt{BFS}(G, D)$.



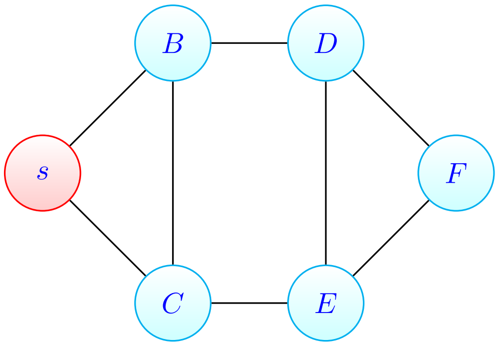
7. Now F has no unvisited neighbors. So $\texttt{DFS}(G, F)$ returns control to the invoking call $\texttt{DFS}(G, D)$.
8. Similarly, as D has no unvisited neighbors, $\texttt{DFS}(G, D)$ returns control to the invoking call $\texttt{DFS}(G, E)$.
9. Now E has no unvisited neighbors. So $\texttt{DFS}(G, E)$ returns control to the invoking call $\texttt{DFS}(G, C)$.
10. As C has no unvisited neighbors, $\texttt{DFS}(G, C)$ returns control to the invoking call $\texttt{DFS}(G, B)$.
11. Now B has no unvisited neighbors, so $\texttt{DFS}(G, B)$ returns control to the invoking call $\texttt{DFS}(G, s)$.
12. As s has no unvisited neighbors, $\texttt{DFS}(G, s)$ returns control to the line where it was invoked.

Remark 11. Using a different ordering would have resulted in traversing the vertices in a different order. We illustrate this with the next example.

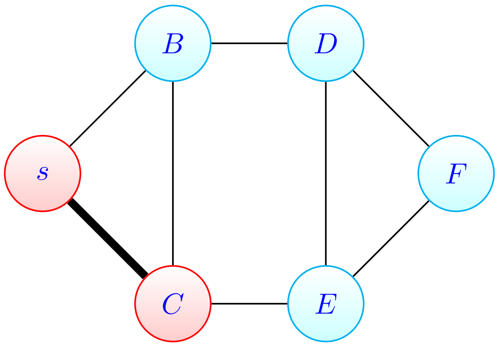
Example 12. In Example 10, we considered the result of calling $\texttt{DFS}(G, s)$ on the following graph G, where the unvisited neighbors were examined in alphabetical order. We choose to select neighbors arbitrarily; that is, not according to a prescribed rule. This yields a different order in which the vertices are visited. There are alternative choices one may make, which will again yield different orderings in which the vertices are visited.



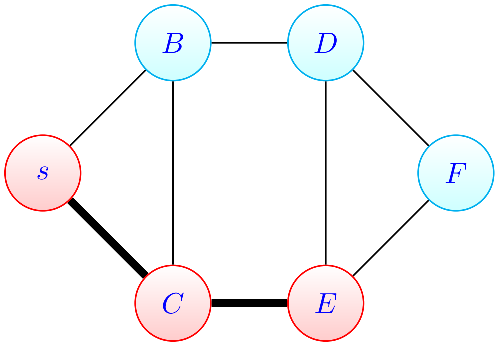
1. As before, we start by setting $s.\text{visited} := \texttt{true}$.



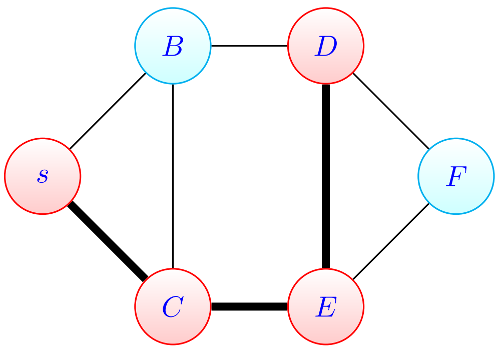
2. The unvisited neighbors of s are B and C. We choose to visit C next. We invoke `DFS`(G, C).



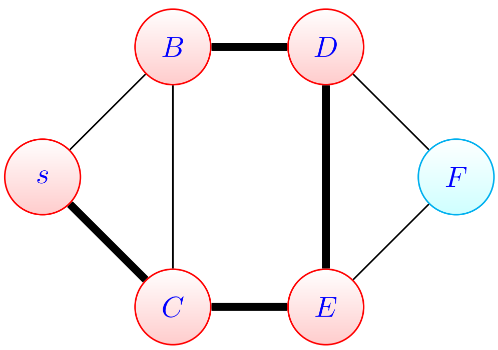
3. The unvisited neighbors of C are B and E. We choose to visit E next, invoking `DFS`(G, E).



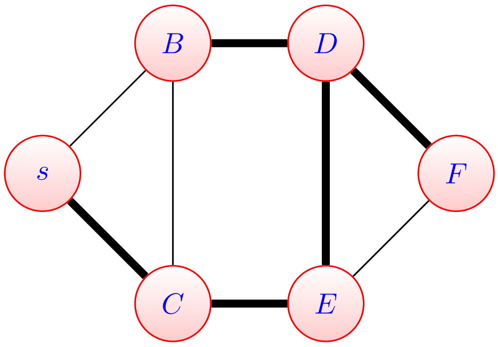
4. The unvisited neighbors of E are D and F. We choose to visit D next, invoking `DFS`(G, D).



5. The unvisited neighbors of D are B and F. We choose to visit B next, invoking `DFS`(G, B).



6. Now B has no unvisited neighbors, so `DFS`(G, B) returns control to `DFS`(G, D).
7. The only unvisited neighbor of D is F. So we invoke `DFS`(G, F).



8. Now F has no unvisited neighbors, so `DFS`(G, F) returns control to `DFS`(G, E).
9. As E has no unvisited neighbors, `DFS`(G, E) returns control to `DFS`(G, C).
10. Now C has no unvisited neighbors, so `DFS`(G, C) returns control to `DFS`(G, s).
11. Finally, `DFS`(G, s) returns control to the line where it was invoked.

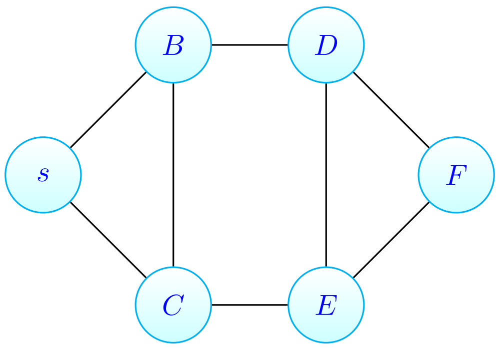
2.2 Breadth-First Traversal

The Breadth-First Traversal algorithm works similarly to the Depth-First Traversal, except that the Breadth-First Traversal examines all of the unvisited neighbors of the current vertex before examning vertices further away. In order to accomplish this, we place the neighbors of the current vertex in a queue and process subsequent vertices based on the ordering enforced by the queue.

Algorithm 2 Breadth-First Traversal

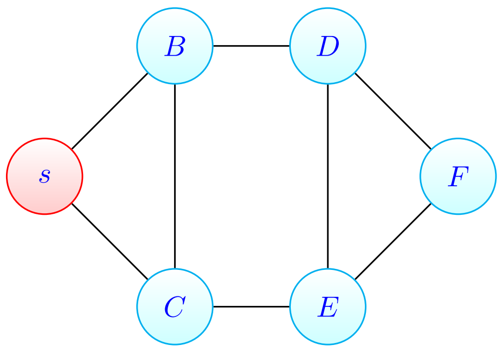
procedure BFS(Graph G, Vertex s)
    s.visited $\leftarrow$ `true`
    Queue $Q \leftarrow [s]$
    while $Q \neq []$ **do**
        current $\leftarrow Q$.poll()
        for each unvisited neighbor v of current **do**
            v.visited $\leftarrow$ `true`
            Q.push(v)

Example 13. We consider an example of the Depth-First Traversal procedure on the following graph G. As in Example 10, we examine the neighbors of the current vertex in alphabetical order.



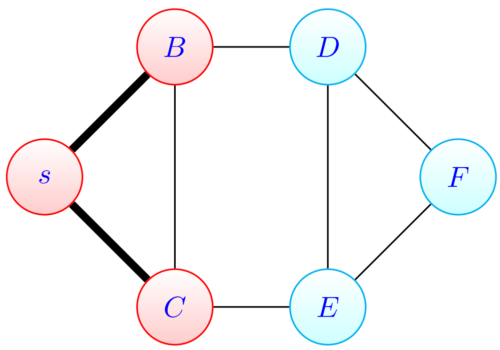
We invoke `BFS`(G, s).

1. We set s.visited := `true`, then push s into the queue Q. So $Q = [s]$.

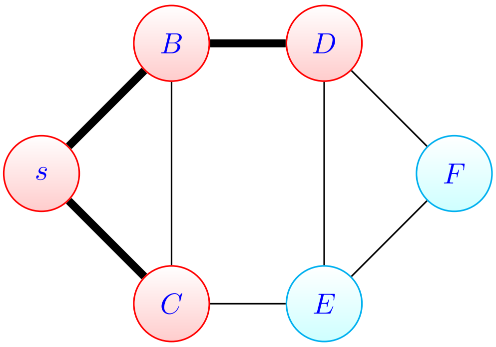


2. We set current := Q.poll(). So current $= s$.

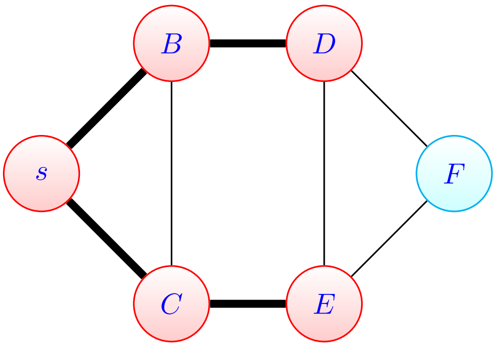
3. We now set the neighbors of s as visited and then push them into Q. So B.visited := `true`, C.visited := `true`, and $Q = [B, C]$.



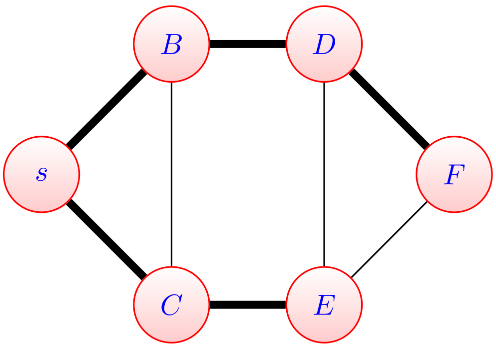
4. We now set current := Q.poll(). So current $= B$.
5. We now mark the unvisited neighbors of B as visited and push them into the queue. So D.visited := `true`, and $Q = [C, D]$.



6. We now set current := Q.poll(). So current $= C$.
7. We now mark the unvisited neighbors of C as visited and push them into the queue. So E.visited := `true`, and $Q = [D, E]$.



8. We now set current := Q.poll(). So current $= D$.
9. We now mark the unvisited neighbors of D as visited and push them into the queue. So F.visited := `true`, and $Q = [E, F]$.



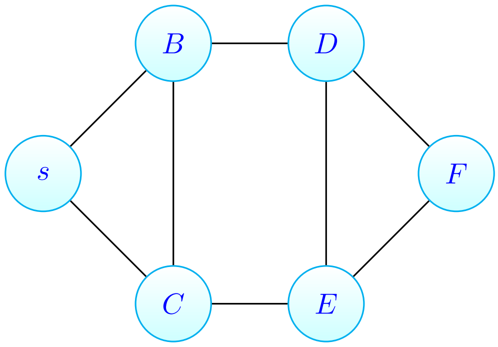
10. We now set current := Q.poll(). So current $= E$. Now E has no unvisited neighbors, so we do not add any vertices to the queue. So $Q = [F]$.
11. We now set current := Q.poll(). So current $= F$. Now F has no unvisited neighbors, so we do not add any vertices to the queue. Now $Q = []$. As $Q = []$, the algorithm terminates.

Remark 14. Just as with the Depth-First Traversal, we need not examine the neighbors of the current vertex alphabetically.

2.3 Shortest-Path Problem: Unweighted Graphs

A problem of key interest in computer science is finding shortest paths between two vertices in a graph. We consider this problem for undirected, unweighted graphs. We note that in an unweighted graph, the length of a path is the number of edges.

Example 15. As an example, we consider the graph below.



Consider the following paths.

- The path $D - E - F$ has length 2, as there are two edges. Note that $D - E - F$ is not a shortest path from D to F, as the path $D - F$ has length 1.
- In general, shortest paths are not unique. Observe that $s - B - D - F$ and $s - C - E - F$ are both shortest paths from s to F.

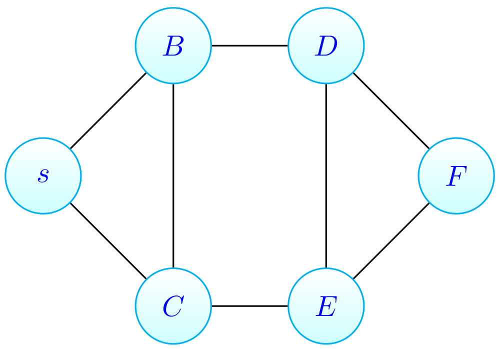
Definition 16 (Unweighted Shortest Path Problem)**.** The Unweighted Shortest Path problem is defined as follows.

- Instance: We take as input an undirected, unweighted graph $G(V, E)$, together with prescribed vertices $s, t \in V(G)$.
- Solution: The length of a shortest path from s to t.

It turns out that the Breadth-First Traversal algorithm can solve this problem. In fact, it does much more. Fix a starting vertex s in our graph G. Applying $\texttt{BFS}(G, s)$ allows us to construct a tree T rooted at s, with the property that for any vertex $v \in V(G)$, the s to v path in T is a shortest s to v path in G. We call such trees *single-source shortest path trees*, or SSPTs. I will often refer to these as *shortest-path trees*.

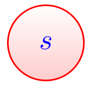
We may modify the Breadth-First Traversal algorithm to construct a SSPT in the following manner. Suppose we are currently examining the vertex u. As we examine u's unvisited neighbor v, we add the edge $\{u, v\}$ to our tree. We consider the following example.

Example 17. Recall Example 13, where we added the unvisited neighbors to the queue in alphabetical order. We again use the same graph G, below.

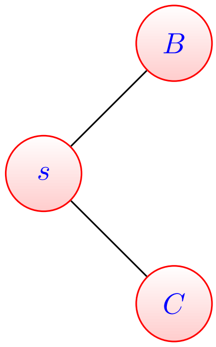


We seek to find the shortest s to v path, for every vertex $v \in V(G)$.

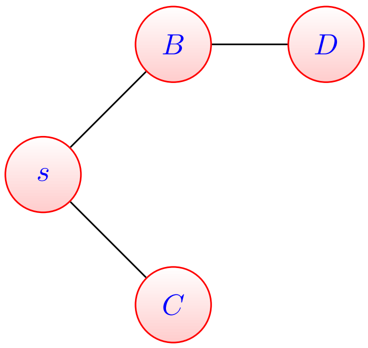
1. We begin by marking s as visited and pushing s into the queue. So $Q = [s]$, and our intermediary tree is as follows:



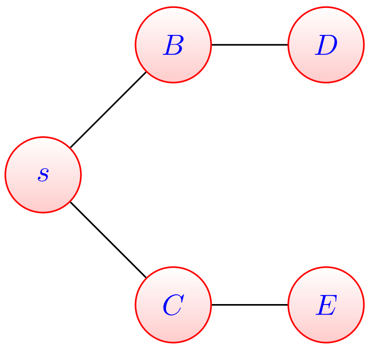
2. We next poll s from Q. We now mark both neighbors of s, B and C, as visited and place them into the queue. So B.visited $=$ `true`, C.visited $=$ `true`, and $Q = [B, C]$. We now add the edges $\{s, B\}$ and $\{s, C\}$ to the tree. So our intermediary tree is as follows.



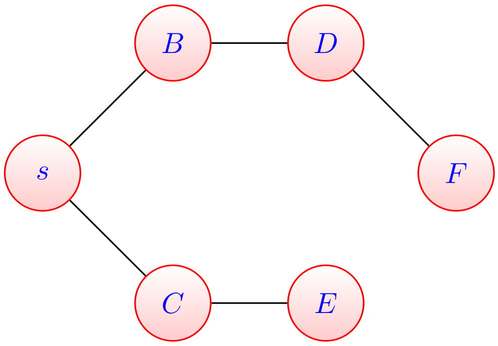
3. We next poll B from Q. We now mark the unvisited neighbor of B, which is D, as visited and place D into the queue. So D.visited $=$ `true`, and $Q = [C, D]$. We now add the edge $\{B, D\}$ to the tree. So our intermediary tree is as follows.



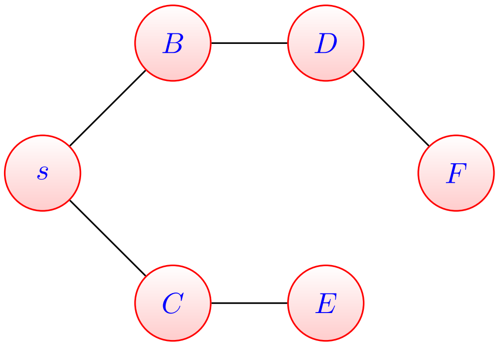
4. We next poll C from Q. We now mark the unvisited neighbor of C, which is E, as visited and place E into the queue. So E.visited $=$ `true`, and $Q = [D, E]$. We now add the edge $\{C, E\}$ to the tree. So our intermediary tree is as follows.



5. We next poll D from Q. We now mark the unvisited neighbor of D, which is F, as visited and place F into the queue. So F.visited $=$ `true`, and $Q = [E, F]$. We now add the edge $\{E, F\}$ to the tree. So our intermediary tree is as follows.



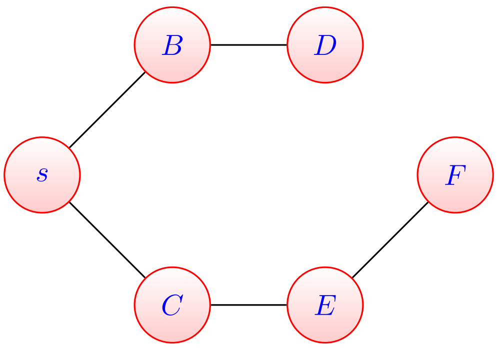
6. Note that the remaining elements of $Q = [E, F]$ have no unvisited neighbors. So we poll E, then poll F. Now as $Q = []$, the algorithm terminates. Our SSPT is:



We leave it to the reader to verify that for each vertex v, the s to v path in the tree above is a shortest s to v path in our original graph G.

Remark 18. Recall that a tree on n vertices has $n - 1$ edges. So once we have placed five edges into the tree for Example 17, we may terminate the algorithm rather than processing the rest of the queue.

Remark 19. In Step 2 of Example 17, we placed B into the queue before C. Had we instead processed C first, we would have obtained the following SSPT.



We record the fact that the Breadth-First Traversal solves the Unweighted Shotest Path problem with the following theorem.

Theorem 20. Let G be an unweighted, undirected graph. Fix a vertex $s \in V(G)$, and fix an ordering[1] $\prec$ in which we place vertices into the queue when running the Breadth-First Traversal algorithm. Let T be the tree obtained by running $\texttt{BFS}(G, s)$, using the ordering $\prec$. Then T is a single-source shortest path tree, with root node s.

Remark 21. While every tree produced by the Breadth-First Traveral algorithm is a SSPT, the converse is not true. That is, there are SSPTs which cannot be obtained by BFS. We leave it to the reader to construct an example.

[1]For instance, we may place the neighbors of a given vertex v into the queue in alphabetical order, as in previous examples. However, different orderings may give rise to different SSPTs, as we saw with Remark 19.

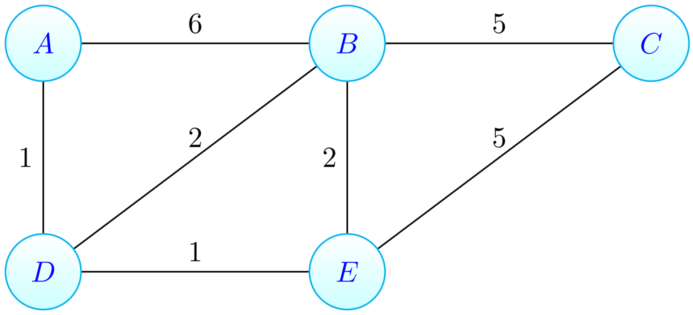
2.4 Shortest-Path Problem: Weighted Graphs and Dijkstra's Algorithm

In this section, we consider the Weighted Shortest Path Problem. We begin by introducing the notion of a weighted graph.

Definition 22. A *weighted graph* $G(V, E, w)$ is a graph, together with a weight function $w : E \to \mathbb{R}$. That is, we label each edge with a real number. For a path P, the weight of P is the sum of the edge weights in P.

Example 23. Consider the following graph. Here, the number on the edge is the weight. So $w(\{B, C\}) = 5$, and $w(\{A, D\}) = 1$. The weight of the path $P := A - B - E$ is:

$$w(P) = w(\{A, B\}) + w(\{A, E\}) = 6 + 2 = 8.$$



We now define the Weighted Shortest Path problem.

Definition 24. The Weighted Shortest Path problem defines as follows.

- Instance: Let $G(V, E, w)$ be a weighted graph, and let $u, v \in V(G)$.
- Solution: Determine the length of a minimum-weight (shortest) path from u to v.

Remark 25. For this section, we restrict attention to when the weights are all non-negative. That is, for any edge $\{x, y\}$, we have that $w(\{x, y\}) \geq 0$. Dijkstra's algorithm may fail to correctly solve the Weighted Shortest Path problem in the presence of negative edge weights. In general, when we allow for both positive and negative edge weights, the Weighted Shortest Path problem is quite hard. There is compelling evidence that there is no efficient (polynomial-time) algorithm to solve the general Weighted Shortest Path problem.[2]

A natural first step is to consider whether the Breadth-First Traversal algorithm correctly solves the Weighted Shortest Path problem. It turns out that the Breadth-First Traversal fails to do so; we leave constructing an example as a homework exercise. Instead, we modify the Breadth-First Traversal to use a priority queue, rather than a queue, to provide the next vertex to consider. Precisely, we start by marking each vertex as un-processed. A vertex is only marked as processed once it is polled from the priority queue. Suppose we polled the vertex x from the priority queue. We examine each un-processed neighbor y of x. If the path from $s \to x \to y$ is shorter than the current known distance from s to y, then we do the following:

(a) We set $\text{dist}(s, y) := \text{dist}(s, x) + \text{w}(\{x, y\})$. If y is already in the priority queue, then we update its position in line. Otherwise, we push y on to the priority queue.

(b) We set y.predecessor $:= x$. By storing the predecessors, we may recover our desired shortest paths.

Note that by construction, no processed vertex can be pushed back on to the priority queue. This ensures that the algorithm terminates. Furthermore, as each vertex stores its predecessor, Dijkstra's algorithm effectively constructs a SSPT.

The algorithm we just described is known as Dijkstra's algorithm. We include the formal algorithm below.

[2]Precisely, the Weighted Shortest Path problem is NP-hard. Algorithms that solve the general Weighted Shortest Path problem can be adapted to detect the presence of Hamiltonian paths in unweighted graphs by setting each edge to have weight -1. So Hamiltonian paths correspond precisely to paths of weight $-(n-1)$ (where n is the number of vertices), and no path can have weight smaller than $-n$. Determining whether a graph has a Hamiltonian path is a well-known NP-hard problem. We will formalize these notions of hardness later in the course, during our discussions of the P vs. NP problem.

Algorithm 3 Dijkstra's Algorithm

Require: The graph G is connected and has no negative-weight edges.

procedure Dijkstra(WeightedGraph $G(V, E, w)$, Vertex s)
    PriorityQueue $Q \leftarrow []$
    for each $v \in V(G)$ **do**
        v.dist $\leftarrow \infty$
        v.predecessor = `NULL`
        Q.push(v)
    s.dist $\leftarrow 0$
    Q.updatePosition(s)

    while $Q \neq []$ **do**
        current $\leftarrow Q$.poll()
        current.processed = `true`

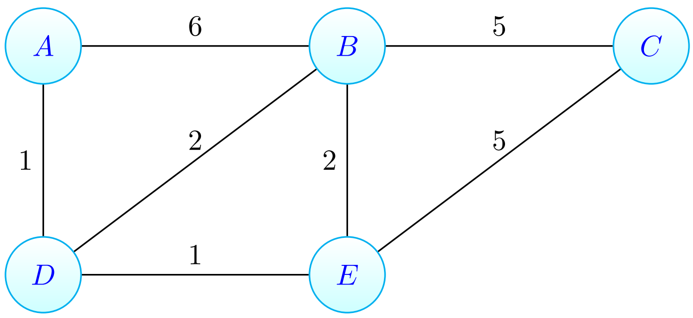
        for each unprocessed neighbor y of current **do**

            if current.dist $+ w(\{\text{current}, y\}) < y$.dist **then**
                y.dist $\leftarrow$ current.dist $+ w(\{\text{current}, y\})$
                y.predecessor $\leftarrow$ current
                Q.updatePosition(y)
    return G

2.4.1 Dijkstra's Algorithm: Example 1

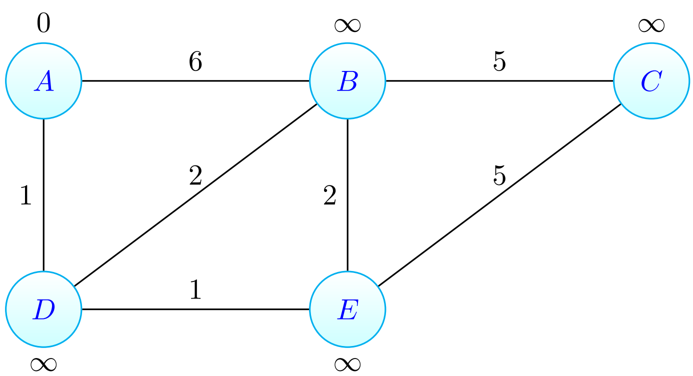
We now consider an example.

Example 26. Consider the graph G below. Suppose we invoke Dijkstra(G, A).



We do the following. We will mark processed vertices in red and use thick edges to denote predecessors.

1. We set the distance attributes for all vertices other than A to ∞. The distance attribute of A is set to 0, and then we push A into the priority queue Q. So $Q = [(A, 0)]$, and the graph with the distance markers is pictured below.



2. We now poll A from the priority queue and set A.processed = `true`. Now the unprocessed neighbors of A are B and D. Observe that:

- $w(\{A, B\}) = 6 < \infty$. So we set:

$$\begin{aligned} B.\text{dist} &= 6, \\ B.\text{predecessor} &= A, \end{aligned}$$

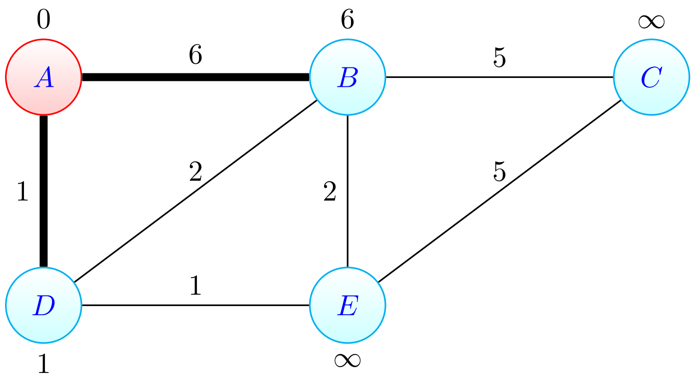
  and then push B into the priority queue.

- Similarly, $w(\{A, D\}) = 1 < \infty$. So we set:

$$\begin{aligned} D.\text{dist} &= 1, \\ D.\text{predecessor} &= A, \end{aligned}$$

  and then push D into the priority queue.

So the priority queue is $Q = [(D, 1), (B, 6)]$. The updated graph is below.



3. We now poll D from the priority queue and mark D as processed. The unprocessed neighbors of D are B and E. Observe that:

   - $\text{dist}(A, D) + w(\{D, B\}) = 1 + 2 < 6$. So we set:

$$\begin{aligned} B.\text{dist} &= 3, \text{ and} \\ B.\text{predecessor} &= D. \end{aligned}$$

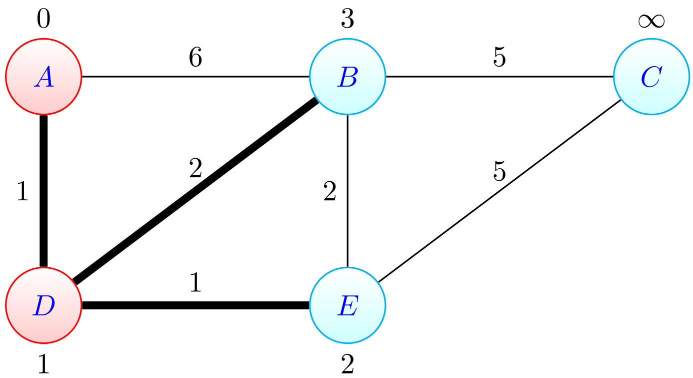
     As B is in the priority queue, we update its position. [**Note:** As B is the only element in the priority queue, the update position call will simply return control without making changes.]

   - As $\text{dist}(A, D) + w(\{D, E\}) = 1 + 1 < \infty$, we set:

$$\begin{aligned} E.\text{dist} &= 2, \\ E.\text{predecessor} &= D, \end{aligned}$$

     and then push E into the priority queue.

   So the priority queue is $Q = [(E, 2), (B, 3)]$. The updated graph is below.



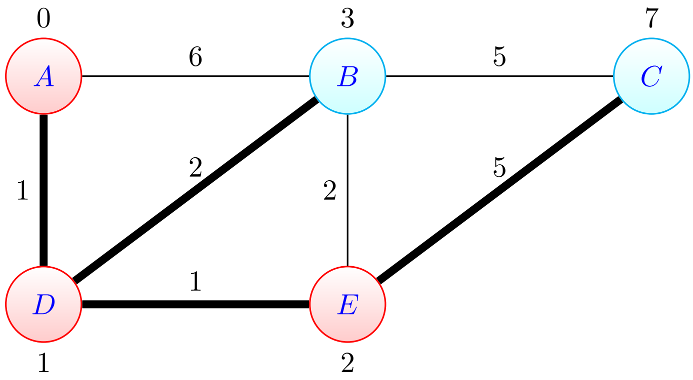
4. We now poll E from the priority queue and mark E as processed. The two unprocessed neighbors of E are B and C. Observe that:

   - $\text{dist}(A, E) + w(\{E, B\}) = 2 + 2 \not< 3$. So we make no further changes to B.
   - As $\text{dist}(A, E) + w(\{E, C\}) = 2 + 5 < \infty$, we set:

   $$C.\text{dist} = 7,$$
   $$C.\text{predecessor} = E,$$

   and then push C into the priority queue. So

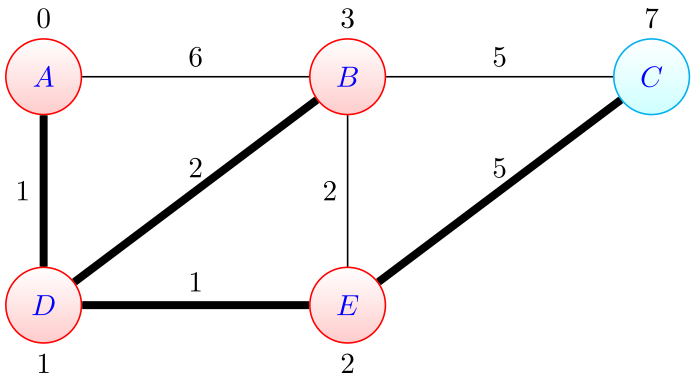
   So the priority queue is $Q = [(B, 3), (C, 7)]$. The updated graph is below.



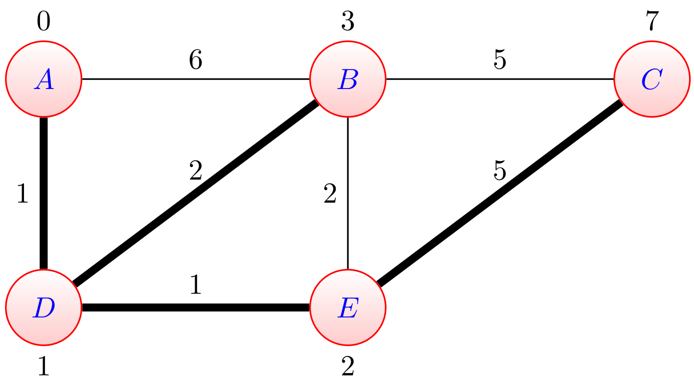
5. We poll B from the priority queue and mark B as processed. The only unprocessed neighbor of B is C. As:

   $$\text{dist}(A, B) + \text{w}(\{B, C\}) = 3 + 5 \not< 7,$$

   we make no changes to C. So the priority queue is $Q = [(C, 7)]$, and the updated graph is below.



6. We poll C from the priority queue and mark C as visited. As C has no unprocessed neighbors and Q is empty, the algorithm terminates. The final graph is below.

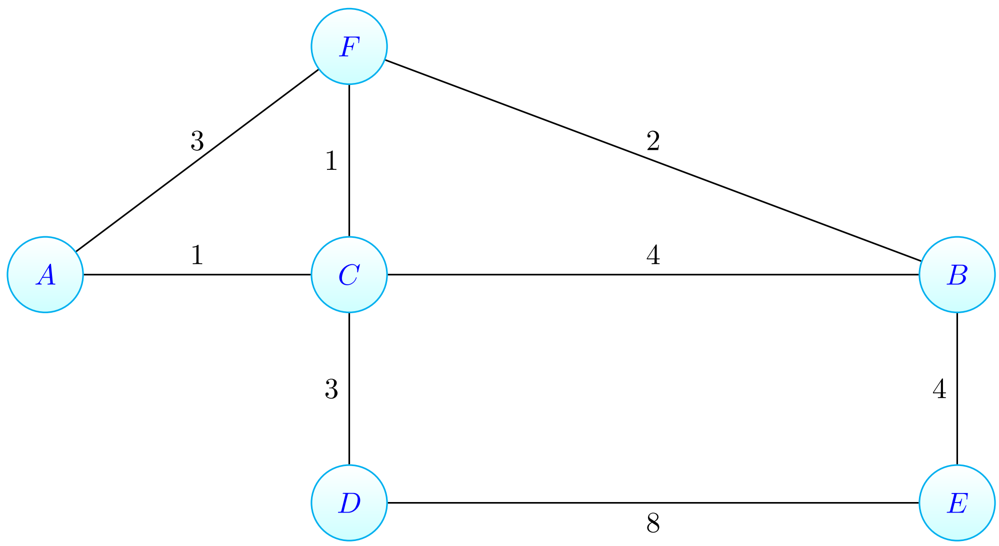


So Dijkstra's algorithm found the following shortest paths from A.

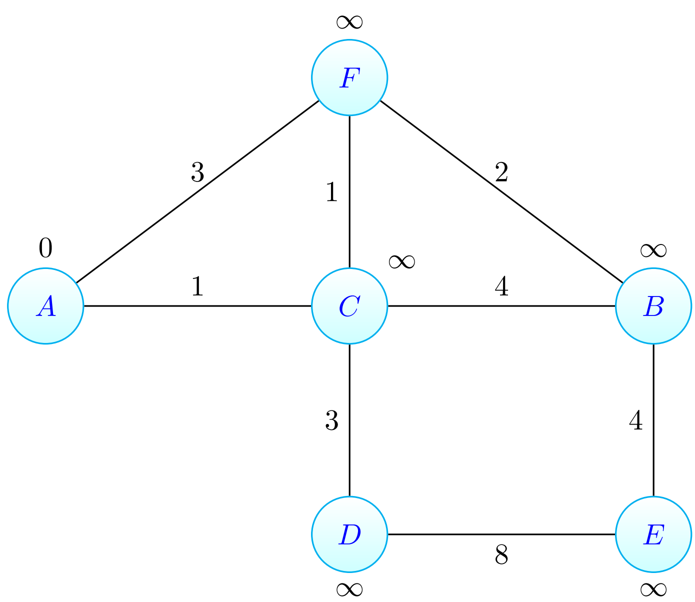
- A to B: The shortest path found was $A - D - B$, which has weight 3.
- A to C: The shortest path found was $A - D - E - C$, which has weight 7.
- A to D: The shortest path found was $A - D$, which has weight 1.
- A to E: The shortest path found was $A - D - E$, which has weight 2.

2.4.2 Dijkstra's Algorithm: Example 2

Example 27. We consider a second example on the graph below. Suppose we invoke `Dijkstra`(G, A).



1. We set the distance attributes for all vertices other than A to ∞. The distance attribute of A is set to 0, and then we push A into the priority queue Q. So $Q = [(A, 0)]$, and the graph with the distance markers is pictured below.



2. We now poll A from the priority queue and set $A.\text{processed} = \texttt{true}$. Now the unprocessed neighbors of A are C and F. Observe that:

   - $w(\{A, C\}) = 1 < \infty$. So we set:

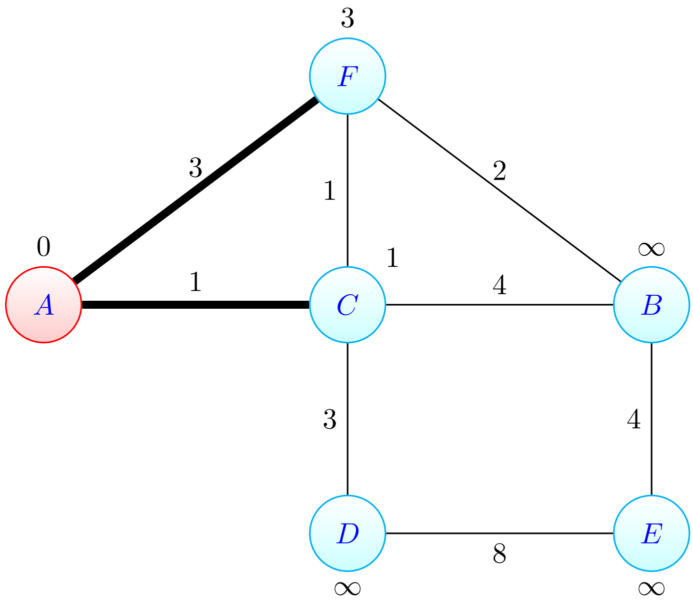
     $$\begin{aligned} &C.\text{dist} = 1, \\ &C.\text{predecessor} = A, \end{aligned}$$

     and then push C into the priority queue.
   - Similarly, $w(\{A, F\}) = 3 < \infty$. So we set:

     $$\begin{aligned} &F.\text{dist} = 3, \\ &F.\text{predecessor} = A, \end{aligned}$$

     and then push F into the priority queue.

   So the priority queue is $Q = [(C, 1), (F, 3)]$. The updated graph is below.



3. We now poll C from the priority queue and set C.processed $=$ `true`. Now the unprocessed neighbors of C are B, D, and F. Observe that:

   - $\text{dist}(A, C) + \text{w}(\{C, B\}) = 1 + 4 < \infty$. So we set:

     $$B.\text{dist} = 5,$$
     $$B.\text{predecessor} = C,$$

     and then push B into the priority queue.
   - Similarly, $\text{dist}(A, C) + \text{w}(\{C, D\}) = 1 + 3 < \infty$. So we set:

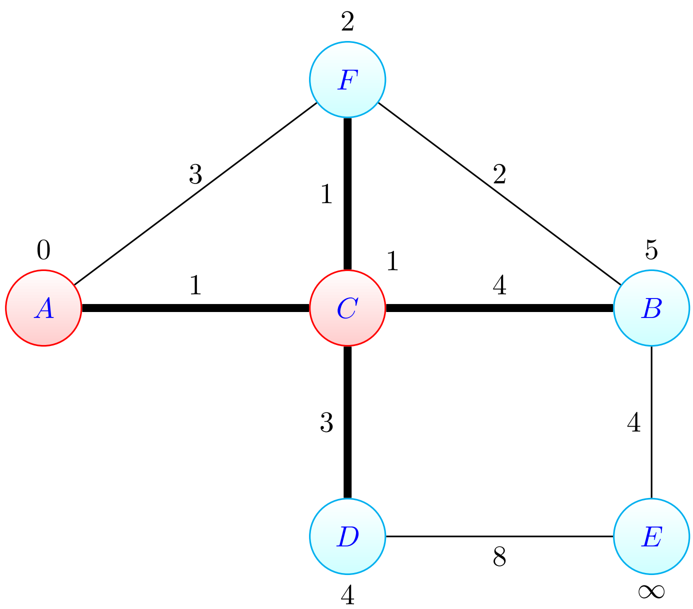
     $$D.\text{dist} = 4,$$
     $$D.\text{predecessor} = C,$$

     and then push D into the priority queue.
   - We have that $\text{dist}(A, C) + w(\{C, F\}) = 1 + 1 < 3$. So we set:

     $$F.\text{dist} = 2,$$
     $$F.\text{predecessor} = C,$$

     and then update F's position in the priority queue.

   So the priority queue is $Q = [(F, 2), (D, 4), (B, 5)]$. The updated graph is below.



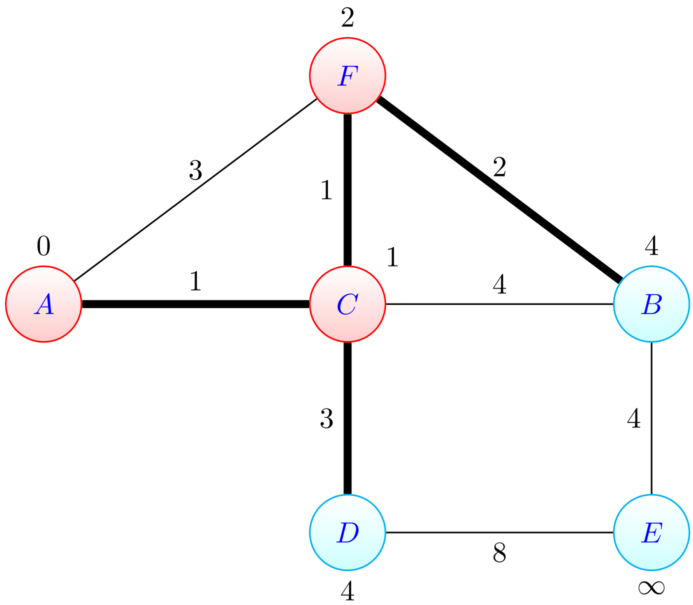
4. We now poll F from the priority queue and set F.processed = `true`. Now the unprocessed neighbor of F is B. Observe that:

   - $\text{dist}(A, F) + \text{w}(\{B, F\}) = 2 + 2 < 5$. So we set:

     $$\begin{aligned} B.\text{dist} &= 4, \\ B.\text{predecessor} &= F, \end{aligned}$$

     and we update B's position in the priority queue. [**Note:** You may choose to either move B ahead of D or keep it after D.]

   So the priority queue is $Q = [(D, 4), (B, 4)]$. The updated graph is below.



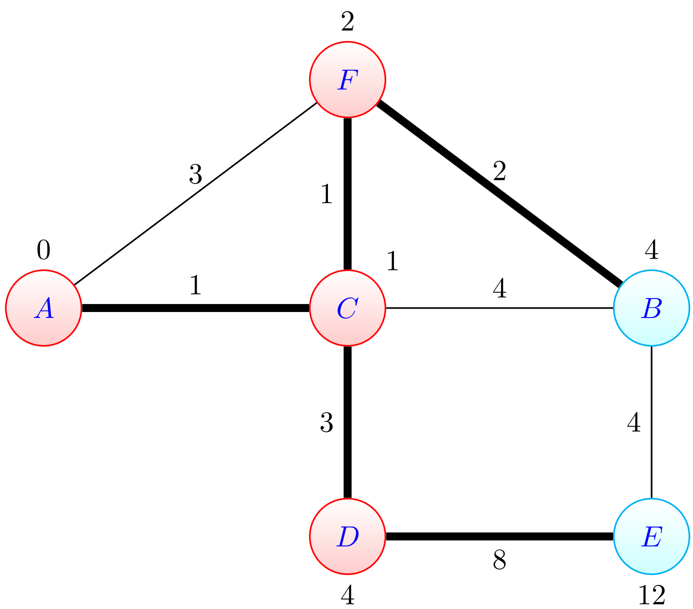
5. We now poll D from the priority queue and set D.processed = `true`. Now the unprocessed neighbor of D is E. Observe that:

   - $\text{dist}(A, D) + \text{w}(\{D, E\}) = 4 + 8 < \infty$. So we set:

     $$\begin{aligned} E.\text{dist} &= 12, \\ E.\text{predecessor} &= D, \end{aligned}$$

     and we push E into the priority queue.

   So the priority queue is $Q = [(B, 4), (E, 12)]$. The updated graph is below.



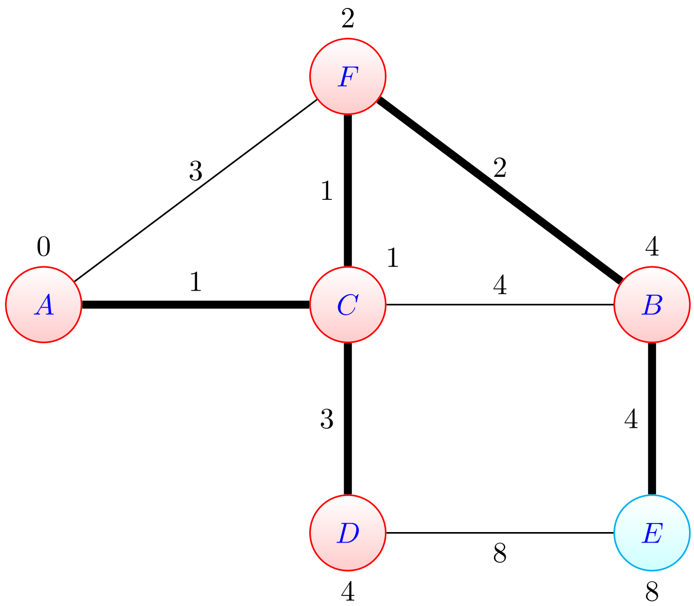
6. We now poll B from the priority queue and set B.processed = `true`. Now the unprocessed neighbor of B is E. Observe that:

   - $\text{dist}(A, B) + \text{w}(\{B, E\}) = 4 + 4 < 12$. So we set:

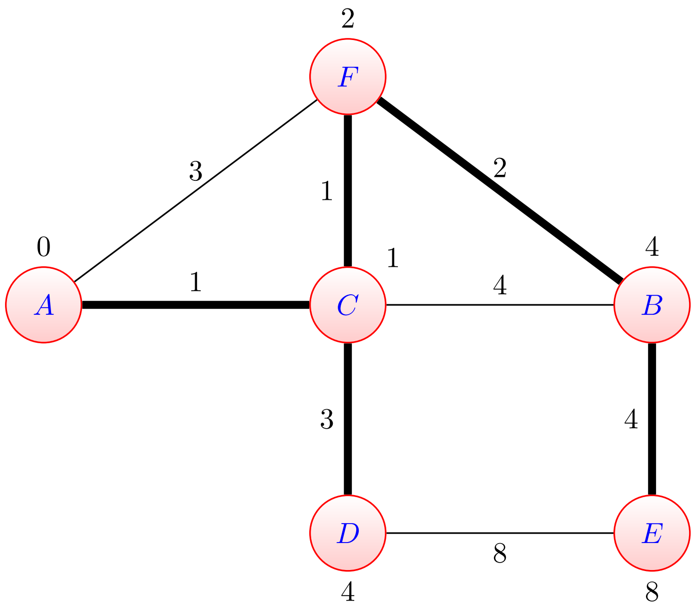
   $$\begin{aligned} &E.\text{dist} = 8, \\ &E.\text{predecessor} = B, \end{aligned}$$

     and we update E's position in the priority queue.

   So the priority queue is $Q = [(E, 12)]$. The updated graph is below.



7. We poll E from the priority queue. As E has no unprocessed neighbors and Q is empty, the algorithm terminates. The final graph is below.



So Dijkstra's algorithm found the following shortest paths from A.

- A to B: The shortest path found was $A - C - F - B$, which has weight 4.
- A to C: The shortest path found was $A - C$, which has weight 1.

- A to D: The shortest path found was $A - C - D$, which has weight 4.
- A to E: The shortest path found was $A - C - F - B - E$, which has weight 8.
- A to F: The shortest path found was $A - C - F$, which has a weight of 2.

2.5 Dijkstra's Algorithm: Proof of Correctness

In this section, we establish the correctness of Dijkstra's algorithm, as well as analyze its runtime complexity. We begin by showing that Dijkstra's algorithm terminates.

Proposition 28. Let $G(V, E, w)$ be a finite, simple, connected, and weighted graph with no negative weight edges. Let $s \in V(G)$ be the selected source vertex. Dijkstra's algorithm terminates.

Proof. Suppose to the contrary that Dijkstra's algorithm does not terminate. So the priority queue Q is never empty. As we poll one vertex from Q at each iteration, it follows that at least one vertex is pushed on to the queue at each iteration. We note that vertices that have been removed from Q are marked as processed and never placed back into Q. It follows that G must be infinite, a contradiction. $\square$

We now seek to show that Dijkstra's algorithm correctly computes a single-source shortest path tree. To this end, we first introduce the following lemma, which establishes that the restriction of a shortest u to v path to any pair of vertices x, y along this path yields a shortest x to y path.

Lemma 29. Let G be a connected, undirected graph, and let $u, v \in V(G)$ be vertices. Let P be a shortest path from u to v. If x, y are vertices in P, then the sub-graph P_{xy} of P with endpoints x and y is a shortest x to y path in P.

Proof. Suppose to the contrary that P_{xy} is not a shortest path from x to y. Let Q be a shortest x to y path in G. We have two cases.

- **Case 1:** Suppose that the only vertices Q shares with P are x and y, then we may shorten P by replacing P_{xy}. That is, we use the path $P' := P_{ux} \cdot Q \cdot P_{yv}$. By construction, $w(P') < w(P)$, contradicting the assumption that P is a shortest path from u to v.
- **Case 2:** Suppose that Q contains a vertex $z \in P_{yv}$. Then we may shorten P by using the path $P' := P_{ux} \cdot Q_{xz} \cdot P_{zv}$. By construction, $w(P') < w(P)$, contradicting the assumption that P is a shortest path from u to v.
- **Case 3:** Suppose that Q contains a vertex $z \in P_{xv}$. We apply the same argument as in Case 2, reversing the roles of x and y.

The result follows. $\square$

We now show that Dijkstra's algorithm correctly computes the shortest s to v path, for every vertex v.

Theorem 30. Let $G(V, E, w)$ be a finite, simple, connected, and weighted graph such that all edge weights are non-negative. Suppose that we run Dijkstra's algorithm on G, using the source vertex s. Let $\text{dist}(s, v)$ denote the distance from s to v computed by Dijkstra's algorithm, and let $d^*(s, v)$ be the length of a shortest path from s to v. After Dijkstra's algorithm terminates, we have for all $v \in V(G)$, that $\text{dist}(s, v) = d^*(s, v)$.

Proof. The proof is by induction on k, the number of vertices polled from the priority queue.

- **Base Case:** When $k = 0$, we have polled no vertices from the priority queue. By construction $\text{dist}(s, s) = d^*(s, s) = 0$, as desired.
- **Inductive Hypothesis:** Fix $\ell \geq 0$. Suppose that for each of the ℓ vertices $s = v_1, \ldots, v_\ell$ polled from the priority queue that $\text{dist}(s, v_i) = d^*(s, v_i)$ for all $i \in \{1, \ldots, \ell\}$.
- **Inductive Step:** Let $v_{\ell+1}$ be the $(\ell+1)$st vertex polled from the priority queue. Suppose to the contrary that the $\text{dist}(s, v_{\ell+1}) > d^*(s, v_{\ell+1})$. We note that at the point where $v_{\ell+1}$ was polled from the priority queue, that $\text{dist}(s, v_{\ell+1})$ is realized via a path containing only vertices from $\{v_1, \ldots, v_\ell\}$. So for any shortest path P from s to $v_{\ell+1}$, there exists an unprocessed vertex w in P. Fix such a shortest s to $v_{\ell+1}$ path P, and let w be the unprocessed vertex in P that is closest to s. By Lemma 29, the sub-path of P with endpoints s and w, which we denote P_{sw}, is a shortest s to w path. So $d^*(s, w) \leq d^*(s, v_{\ell+1})$.

  Now as w is the first unprocessed vertex in P_{sw}, the predecessor of w in P_{sw} is v_i for some $i \in \{v_1, \ldots, v_\ell\}$. Now Dijkstra's algorithm would have examined the $\{v_i, w\}$ edge after polling v_i from the priority queue; at which point, w would have been placed into the priority queue. Thus, the distance computed by Dijkstra's algorithm $\text{dist}(s, w) < \text{dist}(s, v_{\ell+1})$. It follows that w would have been polled before $v_{\ell+1}$, contradicting the fact that w was unprocessed. It follows that $\text{dist}(s, v_{\ell+1}) = d^*(s, v_{\ell+1})$.

$\square$

We now turn towards analyzing the runtime complexity of Dijkstra's algorithm. We first need to understand the complexity of implementing the priority queue. Suppose we implement the priority queue using a standard binary heap. Recall that the binary heap operations have the following runtime complexities:

- Insertion: $O(\log(n))$.
- Removing the first element from the priority queue: $O(\log(n))$.
- Updating an element's position: $O(\log(n))$.
- Searching: $O(n)$

We note that the loop at line 3 of Dijkstra's Algorithm (see, 3) examines each vertex once, taking $O(\log(n))$ steps at each iteration. Here, the $O(\log(n))$ complexity comes from pushing each vertex into the priority queue. So the complexity of lines 2-8 is $O(|V|\log(|V|))$, where $|V|$ is the number of vertices in the graph.

Now lines 9-16 examine each edge of G exactly once. At line 10, we poll a single vertex from the priority queue. As G is connected, we poll each vertex from the queue exactly once. As polling takes time $O(\log(n))$, this adds complexity $O(|V| \cdot \log(|V|))$. Now when we evaluate each edge, we at most update the position of a vertex in the priority queue. This accounts for time complexity $O(|E| \cdot \log(|V|))$, where $|E|$ is the number of edges in the graph. Thus, the time complexity of Dijkstra's algorithm is $O(|V|\log(|V|) + |E|\log(|V|))$, when using a binary heap as our priority queue. We record this with the following theorem.

Theorem 31. The time complexity of Dijkstra's algorithm is $O(|V|\log(|V|) + |E|\log(|V|))$, when using a binary heap as our priority queue.

2.6 Supplemental Reading

For more on the Breadth and Depth First Traversals, we defer to Errickson [Err, Chapters 4-6], CLRS [CLRS09, Chapter 22], Kleinberg & Tardos [KT05, Chapter 2], and OpenDSA [Tea21, Chapter 19.3] (use the Canvas version of All Current OpenDSA Content).

For supplemental reading on Dijkstra's algorithm, we defer to Errickson [Err, Chapter 8], CLRS [CLRS09, Chapter 24], Kleinberg & Tardos [KT05, Chapter 3.3], and OpenDSA [Tea21, Chapter 19.5] (use the Canvas version of All Current OpenDSA Content).

3 Greedy Algorithm Principles

3.1 Exchange Arguments

In this section, we explore a key proof technique used in establishing the correctness of greedy algorithms; namely, the notion of an exchange argument. The key idea is to start with a solution (multi)set S and show that we may swap out or *exchange* elements of S in such a way that improves the solution. Understanding which elements to exchange often provides key insights into designing effective greedy algorithms. Such provable observations imply the correctness of our greedy algorithms.

Example 32. Recall the Making Change problem, where we have an infinite supply of pennies (worth 1 cent), nickels (worth 5 cents), dimes (worth 10 cents), and quarters (worth 25 cents). We take as input an integer $n \geq 0$. The goal is to make change for n using the fewest number of coins possible. The greedy algorithm chooses as many quarters as possible, followed by as many dimes as possible, then as many nickels as possible. Finally, the greedy algorithm uses pennies to finish making change.

Why is the greedy algorithm correct? Why does it select dimes before nickels? Exchange arguments allow us to answer this question. Consider the following lemma.

Lemma 33. Let $n \in \mathbb{N}$ be the amount for which we wish to make change. In an optimal solution, we have at most one nickel.

Proof. Let S be the multiset of coins used to make change for n. Suppose that S contains $k > 1$ nickels. The key idea is that we may exchange each pair of nickels for a single dime. We formalize this as follows.

By the Division Algorithm, we may write $k = 2j + r$, where $j \in \mathbb{N}$ and $r \in \{0, 1\}$. As $k > 1$, we have that $j \geq 1$. So we exchange $2j$ nickels for j dimes to obtain a new solution set S'. Observe that: $|S'| = |S| - j < |S|$. As we may construct a solution using fewer coins, it follows that any optimal solution uses at most one nickel. $\square$

While we will not go through a full proof of correctness for the greedy algorithm to make change, similar lemmas regarding dimes and pennies serve as key steps in establishing the correctness of this algorithm. In fact, Lemma 33 provides the key insight that we should select dimes before nickels; as otherwise, we may need to swap out the nickels for fewer dimes.

In the next section, we will examine an application of the exchange argument to reason about the Interval Scheduling problem.

3.1.1 Supplemental Reading

We refer to Errickson [Err, Chapter 4] and Kleinberg & Tardos [KT05, Chapter 3.2] for supplemental reading on exchange arguments.

3.2 Interval Scheduling

In this section, we consider the Interval Scheduling problem. Intuitively, we have a single classroom. The goal is to assign the maximum number of courses to the classroom, such that no two classes are scheduled for our room at the same time. We now turn to formalizing the Interval Scheduling problme. Here, we think of intervals as line segments on the real line. We specify each interval by a pair s_i and f_i, where $s_i < f_i$. An *interval* with starting point s_i and ending point s_i is the set:

$$[s_i, f_i] = \{x \in \mathbb{R} : s_i \leq x \leq f_i\}.$$

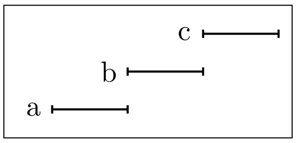
As an example, $[0, 1]$ is the set of real numbers between 0 and 1, including the endpoints 0 and 1. Intuitively, the Interval Scheduling problem takes as input $\mathcal{I}$, a set of intervals. The goal is to find the maximum number of intervals we can select, such that no two intervals overlap.

Definition 34. The Interval Scheduling problem is defined as follows.

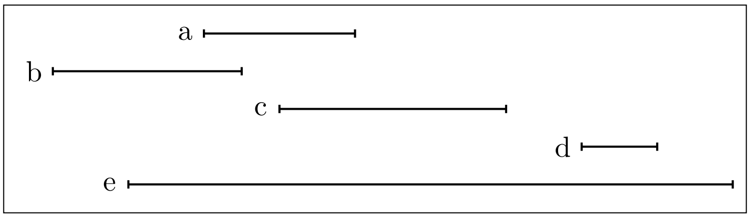
- Instance: Let $\mathcal{I} = \{[s_1, f_1], \ldots, [s_k, f_k]\}$ be our set of intervals.
- Solution: A set $S \subseteq \mathcal{I}$ such that no two intervals in S overlap, where $|S|$ is as large as possible.

We consider some examples.

Example 35. Let $\mathcal{I} = \{[0, 1], [1, 2], [2, 3]\}$. Note that $[0, 1]$ and $[1, 2]$ overlap in the point 1. Similarly, $[1, 2]$ and $[2, 3]$ overlap in the point 2. So any maximum sized set of pairwise disjoint intervals can only contain at most two intervals. Our unique maximum solution set is $S = \{[0, 1], [2, 3]\}$. As $[1, 2]$ overlaps with both $[0, 1]$ and $[2, 3]$, we cannot add $[1, 2]$ to S.
In order to help visualize the problem, the intervals are pictured below.



Example 36. Let $\mathcal{I}$ be the set of intervals pictured below. Observe that the maximum set of pairwise non-overlapping intervals is $S = \{b, c, d\}$.



We now turn towards designing a greedy algorithm for the Interval Scheduling problem. The most natural approach is to place the intervals into a priority queue, polling the intervals one at a time. We store a set S of intervals. As we poll an interval $[s_i, f_i]$ from the priority queue, we place it in S precisely if $[s_i, f_i]$ does not overlap with any of the intervals stored in S. The key issue is to determine how order the intervals within the priority queue. There are several natural orderings, including:

(a) Sorting the intervals from earliest start time to latest start time.

(b) Sorting the intervals by their length. Note that the length of an interval $[s, f]$ is $f - s$.

(c) Sorting the intervals from earliest end time to latest end time.

The following lemma (Lemma 37) provides the key insight that sorting the intervals from earliest end time to latest end time yields a greedy algorithm that solves the Interval Scheduling problem. Note that Lemma 37 does not suggest that all such optimal solutions are of this form, or that there is a unique optimal solution. Rather, Lemma 37 only provides that there exists an optimal solution which is obtained by selecting the intervals from earliest end time to latest end time. The proof of Lemma 37 is adapted from [Mou17].

Lemma 37. Let $\mathcal{I}$ be a set of intervals, and let $S = \{[s_1, f_1], \ldots, [s_m, f_m]\}$ be a set of pairwise non-overlapping intervals. Without loss of generality, suppose that $f_1 < f_2 < \ldots < f_m$. Suppose that there is an interval $[s, f] \in \mathcal{I}$ and an index $i \in \{1, \ldots, m\}$ such:

$$f_{i-1} < s < f < f_i.$$

So $[s, f]$ overlaps with at most one interval: $[s, f]$. Then:

$$S' = \{[s_1, f_1], \ldots, [s_{i-1}, f_{i-1}], [s, f], [s_{i+1}, f_{i+1}], \ldots, [s_m, f_m]\}$$

is a set of pairwise non-overlapping intervals of size $|S|$.

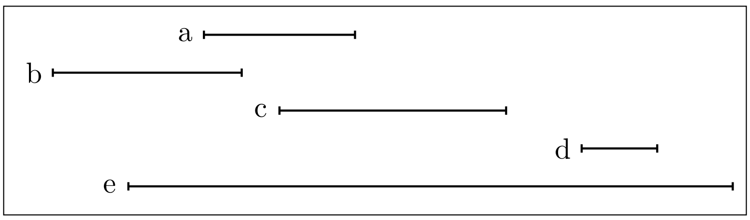
Proof. As $f_{i-1} < s$, we have that $[s, f]$ does not overlap with $[s_1, f_1], \ldots, [s_{i-1}, f_{i-1}]$. Similarly, as $f < f_i < s_{i+1}$, it follows that $[s, f]$ does not overlap with $[s_{i+1}, f_{i+1}], \ldots, [s_m, f_m]$. So S' is a set of pairwise non-overlapping intervals. The result follows. □

In light of Lemma 37, we propose the following greedy algorithm for the Interval Scheduling problem. We use the ordering $\preceq_{\text{end}}$, where $[s_i, f_i] \preceq_{\text{end}} [s_j, f_j]$ precisely if $f_i \leq f_j$.

Algorithm 4 Interval Scheduling

procedure GreedyIntervalScheduling(IntervalSet $\mathcal{I}$, Ordering $\preceq$)
  PriorityQueue $Q \leftarrow []$
  Q.addAll($\mathcal{I}, \preceq$) ▷ Sort the elements of $\mathcal{I}$ according to the ordering $\preceq$
  $S \leftarrow \emptyset$
  while $Q \neq []$ **do**
    $I \leftarrow Q$.poll()
    if I does not overlap with any interval in S **then**
      S.add(I)

Example 38. We apply our algorithm to the set of intervals $\mathcal{I} = \{a, b, c, d, e\}$, where the intervals are pictured below.



The algorithm proceeds as follows.

1. We start by placing the intervals into the priority queue Q, ordered from earliest finish time to latest finish time. So $Q = [b, a, c, d, e]$. Our solution set S is initialized to $S := \emptyset$.
2. We first poll b from the priority queue. As $S = \emptyset$, b does not overlap with any interval in S. So we set $S := S \cup \{b\} = \{b\}$.
3. We next poll a from the priority queue. As a overlaps with b, we discard a.
4. We next poll c from the priority queue. As c does not overlap with b, we set $S := S \cup \{c\}$. So $S = \{b, c\}$.
5. We next poll d from the priority queue. As d does not overlap with either b or c, we set $S := S \cup \{d\}$. So $S = \{b, c, d\}$.
6. We finally poll e from the priority queue. As e overlaps with at least one interval in S, we discard e.
7. The algorithm returns the solution set $S = \{b, c, d\}$.

We now turn to proving that our algorithm for Interval Scheduling (Algorithm 4) yields an optimal solution. There are two things we have to show.

(a) We first show that Algorithm 4 yields a set of pairwise non-overlapping intervals. Note that this holds regardless of the ordering we choose to use.

(b) Now suppose that we order the intervals from earliest end time to latest end time. We show that for any interval set $\mathcal{I}$, Algorithm 4 returns a maximum-sized set of pairwise non-overlapping intervals when using the ordering $\preceq_{\text{end}}$.

Lemma 39. For any ordering $\preceq$, Algorithm 4 yields a set of pairwise non-overlapping intervals.

Proof. Algorithm 4 only adds an interval $[s, f]$ to the solution set S if $[s, f]$ does not overlap with any interval already in S. The result follows. $\square$

We now show that when ordering the intervals from earliest end time to latest end time, that Algorithm 4 yields a maximum-sized set of pairwise non-overlapping intervals. This effectively follows by applying Lemma 37 inductively. Our proof of Theorem 40 is adapted from [Mou16].

Theorem 40. Recall the ordering $\preceq_{\text{end}}$, where $[s_i, f_i] \preceq_{\text{end}} [s_j, f_j]$ precisely if $f_i \leq f_j$. Algorithm 4, using this ordering $\preceq_{\text{end}}$, returns a maximum sized set of pairwise disjoint intervals.

Proof. Let S_i denote the solution set stored at the start of iteration i. We first show by induction on i that there exists an maximum-sized set of pairwise disjoint intervals $\mathcal{O}_i$ containing S_i.

- **Base Case:** We first consider the case when $i = 0$. So before any element is polled from the priority queue, $S_0 = \emptyset$. As every set contains the emptyset, we have that some optimal solution $\mathcal{O}_0$ contains S_0.
- **Inductive Hypothesis:** Fix $k \geq 0$, and let S_k be the solution set at the start of iteration k. Suppose that there exists an maximum-sized solution set $\mathcal{O}_k$ that contains S_k.
- **Inductive Step:** Let S_{k+1} be the solution set at the start of iteration $k + 1$, and let I be the interval polled at iteration k. We have two cases.
  - **Case 1:** If I overlaps with an element of S_k, then I was not added to k. In this case, $S_{k+1} = S_k$. By the inductive hypothesis, there exists a maximum-sized solution set $\mathcal{O}_k$ that contains S_k. So $\mathcal{O}_k$ contains S_{k+1} as well.
  - **Case 2:** Suppose that I does not overlap with any interval in S_k. So $S_{k+1} = S_k \cup \{I\}$. By the inductive hypothesis, there exists a maximum-sized solution set $\mathcal{O}_k$ that contains S_k. As $\mathcal{O}_k$ is a maximum-sized solution set for the Interval Scheduling problem and S_{k+1} is a set of pairwise non-overlapping intervals such that $|S_{k+1}| = |S_k| + 1$, it follows that $|\mathcal{O}_k| \geq |S_{k+1}|$.

    Suppose that the elements of $\mathcal{O}_k$ are ordered from earliest finish time to latest finish time. Let J be the $(k + 1)$st interval in $\mathcal{O}_k$ under this ordering. If $I = J$, then $\mathcal{O}_k$ contains S_{k+1}, and we are done. Suppose instead that $I \neq J$. As S_k is contained in both S_{k+1} and $\mathcal{O}_k$, we have that the first k intervals (under the ordering of intervals from earliest finish time to latest finish time) of S_{k+1} and $\mathcal{O}_k$ agree. Now we write $I = [s, f]$ and $J = [x, y]$. As Algorithm 4 selects intervals based on the earliest end time, it follows that Algorithm 4 considered I before J. So $f \leq y$. Thus, by Lemma 37, $\mathcal{O}_{k+1} := (\mathcal{O}_k \setminus \{J\}) \cup \{I\}$ is also a maximum-sized solution. Additionally, $\mathcal{O}_{k+1}$ contains S_{k+1}, as desired.

  So by induction, we have that the solution set S^* returned by the algorithm is contained in some optimal solution $\mathcal{O}$.

We now claim that S^* is an optimal solution. Suppose to the contrary that $S^* \subsetneq \mathcal{O}$. Then there exists an interval I in $\mathcal{O}$ that is not contained in S^*. As I does not overlap with any interval in $\mathcal{O}$, we have that I does not overlap with any interval of S^*. So the greedy algorithm would have placed I in S^*, contradicting the assumption that $S^* \subsetneq \mathcal{O}$. Thus, S^* is indeed an optimal solution, as desired. $\square$

3.2.1 Supplemental Reading

The course notes by Mount [Mou17] and Moutadid [Mou16] serve as good references for the Interval Scheduling problem. Errickson [Err, Chapter 4.2], CLRS [CLRS09, Chapter 16], and Kleinberg & Tardos [KT05, Chapter 3.1] also serve as standard references on the Interval Scheduling Problem.

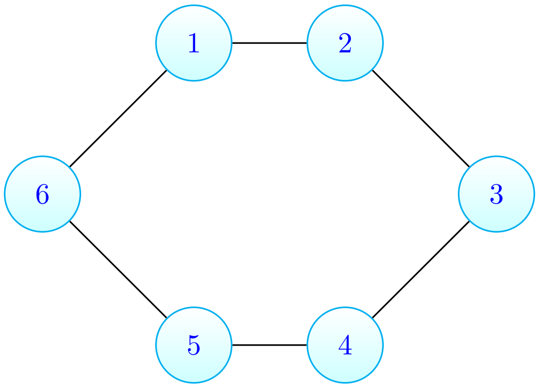
3.3 Example Where the Greedy Algorithm Yields Sub-Optimal Solutions

The greedy technique is quite powerful. However, not all problems are amenable to greedy solutions. In this section, we examine one such problem: finding maximum-sized matchings in arbitrary graphs. We begin by introducing the notion of a matching.

Definition 41. Let $G(V, E)$ be a graph. A *matching* $\mathcal{M}$ is a set of edges such that no two edges in $\mathcal{M}$ share a common vertex. That is, if $\{i, j\}, \{u, v\} \in \mathcal{M}$, then $i \neq u, i \neq v$ and $j \neq u, j \neq v$.

We now consider some examples.

Example 42. Consider the cycle graph C_6 pictured below.



There are several matchings on C_6. We consider some of them below.

(a) First, $\mathcal{M} = \emptyset$ is a matching of C_6. As there are no edges in $\mathcal{M}$, the condition that every pair of distinct edges from $\mathcal{M}$ are disjoint is vacuously satisfied.

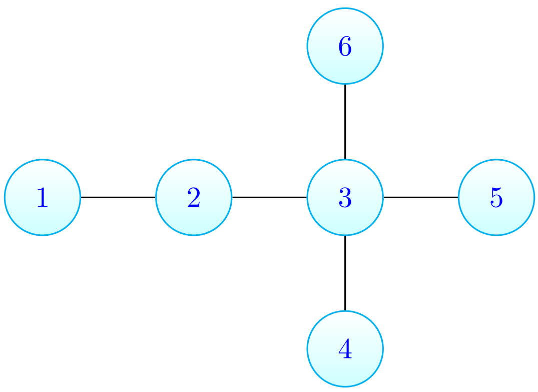
(b) Let $\{i, j\}$ be an arbitrary edge of C_6. Then $\mathcal{M} = \{\{i, j\}\}$ is a matching of C_6. As there is only one edge in $\mathcal{M}$, no two distinct edges of $\mathcal{M}$ share any endpoints.

(c) Consider the set $S = \{\{1, 2\}, \{2, 3\}\}$. As $\{1, 2\}$ and $\{2, 3\}$ share a common endpoint (namely, the vertex 2), the set S is **not** a matching.

(d) Consider the set $\mathcal{M} = \{\{1, 2\}, \{3, 4\}\}$. As $\{1, 2\}$ and $\{3, 4\}$ do not share any endpoints in common, $\mathcal{M}$ is a matching.

(e) The set $\mathcal{M} = \{\{1, 2\}, \{3, 4\}, \{5, 6\}\}$ is a matching, as no two edges share any endpoints in common. Observe that any remaining edge of C_6 shares an endpoint with exactly two edges of $\mathcal{M}$. For instance, $\{2, 3\}$ shares the vertex 2 in common with $\{1, 2\}$ and the vertex 3 in common with $\{3, 4\}$. As $\{2, 3\}$ shares an endpoint in common with at least one edge of $\mathcal{M}$, the set $\mathcal{M} \cup \{\{2, 3\}\}$ is **not** a matching. By similar argument, $\mathcal{M} \cup \{\{4, 5\}\}$ and $\mathcal{M} \cup \{\{1, 6\}\}$ are also **not** matchings of C_6.

Example 43. Consider the following graph G, pictured below.



We consider some examples of matchings on G.

(a) Let $\mathcal{M}_1 = \{\{2,3\}\}$. Observe that every other edge of G shares an endpoint with $\{2,3\}$. Therefore, $\mathcal{M}_1$ does not sit inside a larger matching.

(b) Let $\mathcal{M}_2 = \{\{1,2\},\{3,5\}\}$. Observe that every other edge of G shares an endpoint with either $\{1,2\}$ or $\{3,5\}$. Therefore, $\mathcal{M}_2$ does not sit inside a larger matching.

We note that $\mathcal{M}_1$ and $\mathcal{M}_2$ are both *maximal*, in the sense that neither $\mathcal{M}_1$ nor $\mathcal{M}_2$ sit inside a larger matching. However, $\mathcal{M}_1$ has a single edge, while $\mathcal{M}_2$ has two edges. Now the maximum number of edges a matching of our graph G can have is 2. Therefore, $\mathcal{M}_2$ is a maximum-sized matching, while $\mathcal{M}_1$ is maximal but not maximum-sized.

We now formalize the notions of maximal and maximum matchings.

Definition 44. Let $G(V,E)$ be a graph, and let $\mathcal{M}$ be a matching of G. We say that $\mathcal{M}$ is *maximal* if there is no other matching $\mathcal{M}'$ such that $\mathcal{M} \subsetneq \mathcal{M}'$. That is, $\mathcal{M}$ is maximal if no other matching of G strictly contains $\mathcal{M}$.

We say that $\mathcal{M}$ is a *maximum-cardinality matching* if $\mathcal{M}$ has the maximum number of possible edges. The *matching number* $\nu(G)$ is the size of a maximum-cardinality matching of G.

We now consider the Maximum-Cardinality Matching problem.

Definition 45. The Maximum-Cardinality Matching problem is defined as follows.

- Instance: Let $G(V,E)$ be a graph.
- Solution: A matching $\mathcal{M}$ of G that has size $|\mathcal{M}| = \nu(G)$.

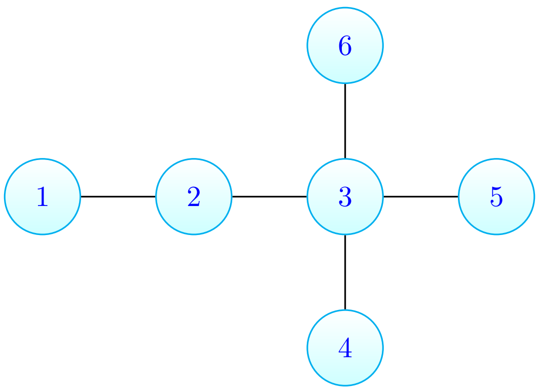
We consider the following greedy algorithm, in an attempt to solve the Maximum-Cardinality Matching problem.

Algorithm 5 GreedyMatching

procedure GreedyMatching(Graph G)
    Queue $Q \leftarrow []$
    Q.addAll($E(G)$)
    $\mathcal{M} \leftarrow \emptyset$
    while $Q \neq []$ **do**
        $e \leftarrow Q$.poll()
        if e does not share an endpoint with any edge in $\mathcal{M}$ **then**
            $\mathcal{M}$.add(e)

Algorithm 5 clearly produces a matching, as it only adds an edge e to $\mathcal{M}$ if e does not share an endpoint with any edge already in $\mathcal{M}$. Furthermore, as the algorithm examines every edge, the matching $\mathcal{M}$ is maximal. However, the matching returned depends on the order in which the edges are added to the queue. We consider an example.

Example 46. Again consider the following graph G, pictured below.



We have the following.

- Suppose that Algorithm 5 adds the edges to the queue in the following order:

$$Q = [\{2,3\}, \{3,6\}, \{3,4\}, \{3,5\}, \{1,2\}].$$

  The first edge considered is $\{2,3\}$. As all other edges of G share an endpoint with $\{2,3\}$, we have that the matching constructed by Algorithm 5 is $\mathcal{M} = \{\{2,3\}\}$.

- Suppose that Algorithm 5 adds the edges to the queue in the following order:

$$Q = [\{1,2\}, \{2,3\}, \{3,6\}, \{3,4\}, \{3,5\}].$$

  The first edge considered is $\{1,2\}$, which is added to our matching. Now as $\{2,3\}$ shares an endpoint with $\{1,2\}$, the algorithm discards $\{2,3\}$. We next consider $\{3,6\}$, which we add to the matching. The remaining edges in Q, $\{3,4\}$ and $\{3,5\}$, each share an endpoint with either $\{1,2\}$ or $\{3,6\}$. So the matching constructed by Algorithm 5 is $\mathcal{M} = \{\{1,2\}, \{3,6\}\}$.

So Algorithm 5 is not guaranteed to return a maximum-cardinality matching.

4 Spanning Trees

In this section, we introduce the Minimum Spanning Tree problem. We begin with a motivating example. Suppose a town experiences a snowstorm. It is imperative that that the residents are able to travel between any two destinations. However, given the volume of snow, plowing all of the roads will take time. Longer roads will also take more time to clear. Therefore, we seek to find the shortest roads to clear, such that clearing those specific roads will allow for travel between any two destinations.

We may formalize this problem using the language of graph theory. Here, the destinations are the vertices of our graph. There is an edge $\{u, v\}$ in our graph precisely if there is a road connected u and v. The weight of the edge $w(\{u, v\})$ is the length of the road. Determining which roads to plow is equivalent to finding a minimum-weight spanning tree of our graph. We formalize this problem as follows.

Definition 47. The Minimum Spanning Tree problem is defined as follows.

- Instance: An undirected, connected, and weighted graph $G(V, E, w)$, where $w : E(G) \to \mathbb{R}$ is the function assigning a real-valued weight to each edge.
- Solution: A spanning tree T of G such that $w(T)$ is minimized. Recall that:

$$w(T) := \sum_{e \in E(T)} w(e).$$

4.1 Preliminaries: Trees

In order to design efficient algorithms to construct minimum-weight spanning trees, we seek to answer two main questions:

(a) How many edges belong to a spanning tree?

(b) Which edges of the input graph should we include in the spanning tree?

Understanding the theory of trees allows us to answer these questions. We begin by recalling the definitions of both a tree and a cut edge.

Definition 48. A *tree* is a connected, acyclic graph.

Definition 49. Let $G(V, E)$ be a graph. An edge $e \in E(G)$ is said to be a *cut edge* if $G - e$ is not connected.

There are three key properties of interest when discussing trees: connectivity, acyclicity, and the number of edges. We will show that if a graph has any two properties drawn from (i) being connected, (ii) having no cycles, and (iii) having $n - 1$ edges (where n is the number of vertices); then the graph necessarily has all three properties. As a result, we note that any spanning tree has $n - 1$ vertices. Additionally, these properties suggest that we retain edges from the input graph that connect the graph, but do not create cycles.

Theorem 50. Let $T(V, E)$ be a graph on n vertices. The following are equivalent.

(a) T is a tree. (That is, T is a connected, acyclic graph.)

(b) T is acyclic and has $n - 1$ edges.

(c) T is connected and has $n - 1$ edges.

Proof. We have the following.

- (a) $\implies$ (b) and (c): As a tree is connected and acyclic, it suffices to show that T has $n - 1$ edges. We do so by induction on n, the number of vertices. When $n = 1$, T consists of a single vertex and has no edges. Now fix $k \geq 1$ and suppose that any tree on at most k vertices has $k - 1$ edges. Let T be a tree on $k + 1$ vertices. As $k + 1 \geq 2$, T has a leaf vertex, which we call v. Note that $T - v$ is a tree with k vertices. So by the IH, $T - v$ has $k - 1$ edges. As v is a leaf in T, $\deg(v) = 1$. So $T - v$ has one fewer edge than T. Thus, T has k edges. The result follows by induction.

- (b) $\implies$ (a) and (c): Let T be an acyclic graph with $n-1$ edges. Let $X_1, \ldots, X_k$ be the connected components of T. As T is acyclic, $X_1, \ldots, X_k$ are trees. So by the proof that (a) $\implies$ (b) and (c), we have that X_i has $|X_i| - 1$ edges. As each vertex appears in exactly one component of T, we have that:

$$\sum_{i=1}^{k} (|X_i| - 1) = n - k.$$

  As we have $n-1$ edges by assumption, $k = 1$. So T is connected.

- (c) $\implies$ (a) and (b): Suppose T is a connected graph on $n-1$ edges. We first recall that if C is a cycle in T and e is an edge on C, then $T - e$ remains connected. So while T has a cycle, we remove an edge on said cycle. As T is finite, this procedure will terminate. We label the updated tree as T'. Let k denote the number of edges removed from T to obtain T'. We note that T' is connected (as none of the edges removed were cut edges) and acyclic. So T' is a tree, with the same vertex set as T. By the (a) $\implies$ (b) and (c) paragraph, we note that T' has $n-1$ edges. By assumption, T also has $n-1$ edges, which implies that no edges were removed from T to obtain T'. So $T = T'$. Thus, T is acyclic.

□

Theorem 50 suggests a couple of algorithms for constructing spanning trees.

- The first approach is to first sort the edges of the graph from lowest weight to highest weight. We then construct a tree by adding edges one at a time, so long as they do not create a cycle. This algorithm is known as Kruskal's Algorithm.
- The second approach is to first sort the edges of the graph from highest weight to lowest weight. We then construct a tree by removing edges from the graph one at a time, so long as removing a given edge does not disconnect the remaining graph. This second algorithm is known as the Reverse-Delete algorithm. We will not pursue the Reverse-Delete algorithm further in this class.

While both algorithms will construct spanning trees, it is less clear that these algorithms (or even, other algorithms) return minimum-weight spanning trees. As all spanning trees on n vertices have $n-1$ edges, it seems plausible to use an exchange argument that exchanges one edge for exactly one other edge, in order to prove that our algorithms return minimum-weight spanning trees. Theorem 50 does not provide sufficiently precise insights as to the exchange. To this end, we introduce additional characterizations of trees. The proofs of these characterizations provide insights on how to exchange edges. We will apply these proof techniques in the next section to begin reasoning as to which edges are safe to include in a minimum-weight spanning tree.

Theorem 51. Let $T(V, E)$ be a graph on n vertices. The following are equivalent.

(a) T is a tree.

(b) T is connected; and for every edge $e \in E(T)$, $T - e$ is not connected. [That is, T is minimally connected.]

(c) For every pair of vertices $u, v \in V(T)$, there exists a unique path from u to v in T.

(d) T contains no cycles; and for any vertices $u, v \in V(T)$, adding the edge $\{u, v\}$ creates a cycle in T. [That is, T is maximally acyclic.]

Proof. We have the following.

- (a) $\implies$ (b): Let T be a tree, and let $e = \{u, v\} \in E(T)$. Suppose to the contrary that $T - e$ is connected. So there exists a $u - v$ P in T, where $P \neq \{u, v\}$. So $P \cup \{u, v\}$ is a cycle, which implies that T has a cycle. This contradicts the assumption that T has no cycles. So $T - e$ is not connected.
- (b) $\implies$ (c): The proof is by contrapositive. Suppose there exist vertices $u, v \in V(T)$, such that there exist two $u - v$ paths in T. Label these paths P_1, P_2, where we provide P_1 and P_2 as sequences of vertices: $P_1 = (x_1, \ldots, x_k)$ and $P_2 = (y_1, \ldots, y_j)$. As $P_1 \neq P_2$, there exist subpaths $P_1' = (x_i, \ldots, x_h)$ and $P_2' = (y_\ell, \ldots, y_m)$ and P_1' and P_2' agree only on the endpoints. It follows that $P_1' \cup P_2'$ is a cycle. So not every edge of T is a cut edge.

- (c) $\implies$ (a): As there exists a unique $u-v$ path in T for every pair of vertices $u, v \in T$, we have that T is connected. From the proof in the (b) $\implies$ (c) direction, it follows that T is acyclic (for if T had multiple $u-v$ paths for some vertices u, v, then T would have a cycle). Thus, T is a tree.

- (a) $\implies$ (d): As T is a tree, T contains no cycles. Let $u, v \in V(T)$ be non-adjacent vertices. So $\{u, v\}$ is not a $u-v$ path in T. As T is connected, there exists a $u-v$ path $P \neq uv$ in T. So $P \cup \{u, v\}$ forms a cycle.

- (d) $\implies$ (a): As T contains no cycles, it remains to show that T is connected. Suppose to the contrary that T is not connected. Let X_1, X_2 be two connected components of T. Let $u \in V(X_1)$ and $v \in V(X_2)$. So $\{u, v\}$ is a cut edge of $T \cup \{u, v\}$, which implies that $T \cup \{u, v\}$ does not contain a cycle, a contradiction.

$\square$

Remark 52. Theorem 51 suggests two exchange arguments. The first natural approach is to add an edge to the tree, which creates a cycle, and then removing another edge from said cycle. The second approach is to remove an edge, which disconnects the tree; we then seek to the removed edge with another edge that connects the two components. We will apply these exchange techniques in the subsequent sections.

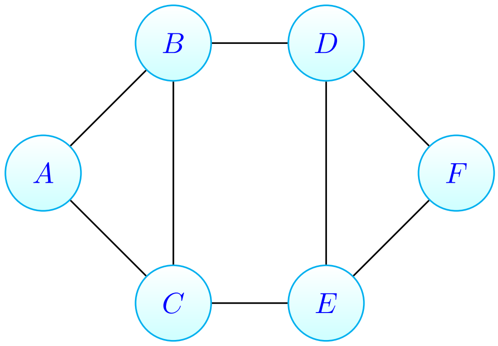
4.2 Safe and Useless Edges

Theorem 50 and Theorem 51 suggest that we should retain edges in the graph that connect the graph without creating cycles. We note that these theorems only deal with unweighted graphs. As we are interested in finding minimum-weight spanning trees on weighted graphs, we need to modify the techniques from Section 4.1 to handle edge weights. To this end, we introduce the notions of safe and useless edges. We follow closely the exposition from Carlson & Davies [CD20].

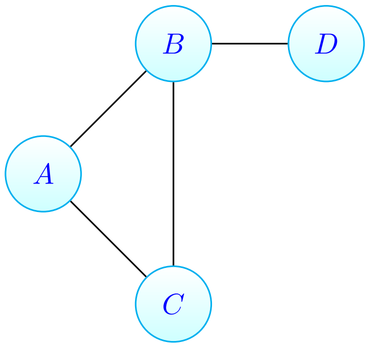
The key approach we will use in designing algorithms to find minimum-weight spanning trees is to manage an *intermediate spanning forest* and add edges until our forest becomes a tree. We wish to add edges in such a way that the resulting spanning tree has minimum weight. We begin with the notions of an induced subgraph and intermediate spanning forest.

Definition 53. Let $G(V, E)$ be a graph, and let $S \subseteq V(G)$ be a set of vertices. The graph *induced* by S is the graph H, where (i) the vertex set of H is S, and (ii) $\{u, v\} \in E(H)$ precisely if $\{u, v\} \in E(G)$. That is, we start with the vertices of S and add all available edges from G where both endpoints are in S.

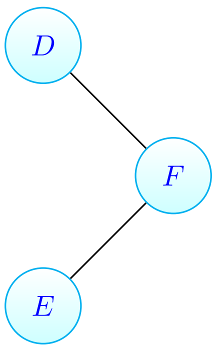
Example 54. Consider the following graph G.



Let $S = \{A, B, C, D\}$. The subgraph H induced by S is pictured below. Note that only the edges of G where both endpoints belong to S are included.



The following graph K pictured below is **not** an induced subgraph. Here, our vertex set is $S = \{D, E, F\}$. Note that both the edges $\{D, F\}$ and $\{E, F\}$ are in G, so these edges do not cause our graph not to be induced. However, the edge $\{D, E\}$ is in G, but not in K. Therefore, the absence of the edge $\{D, E\}$ from K is why K is not an induced subgraph.

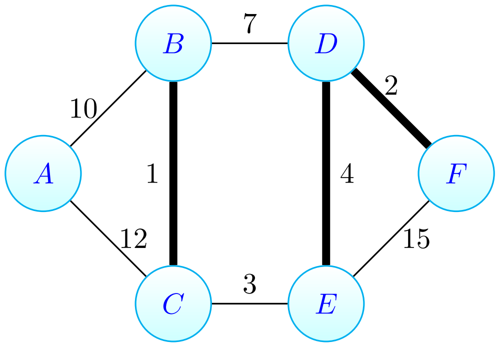


We now turn to formalizing the notion of an *intermediate spanning forest*.

Definition 55. A *forest* $\mathcal{F}$ is a collection of disjoint trees $T_1, \ldots, T_k$. We say that $\mathcal{F}$ is a *spanning forest* of the graph $G(V, E)$ if every vertex of $V(G)$ is contained in $\mathcal{F}$. Note that as the trees of $\mathcal{F}$ are disjoint, each vertex $v \in V(G)$ will be contained in exactly one tree of $\mathcal{F}$.

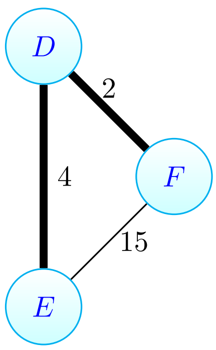
Now we say that $\mathcal{F}$ is an *intermediate spanning forest* if (i) $\mathcal{F}$ is a spanning forest, and (ii) each tree T_j is a minimum-weight spanning tree on the graph induced by $V(T_j)$.

Example 56. Consider the following weighted graph $G(V, E, w)$. The edges in our intermediate spanning forest $\mathcal{F}$ are indicated by thick edges.



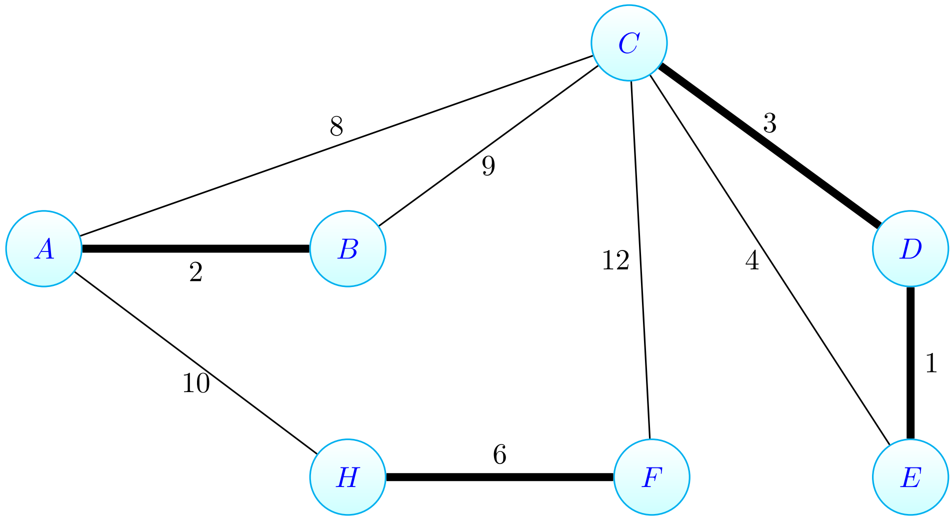
In order to be explicit, we identify the individual trees in $\mathcal{F}$:

- $T_1 = \{A\}$. This tree consists solely of the isolated vertex A, with no additional edges. As there are no edges, T_1 is a (and in fact, the only) minimum-weight spanning tree on the component corresponding to the vertex set $\{A\}$.
- $T_2 = \{B, C\}$. This tree consists of the vertices B and C, together with the edge $\{B, C\}$. As there are only two vertices, this component has at most one edge from G. Therefore, T_2 is a minimum-weight spanning tree on the component corresponding to the vertex set $\{B, C\}$.
- $T_3 = \{D, E, F\}$. This tree consists of the vertices $\{D, E, F\}$, together with the edges $\{D, E\}$, and $\{D, F\}$. Note that the component induced by $\{D, E, F\}$ is the cycle C_3 consisting of the edges $\{D, E\}, \{D, F\}, \{E, F\}$ (pictured below). The unique minimum-weight spanning tree on this component consists precisely of the edges $\{D, E\}$ and $\{D, F\}$.



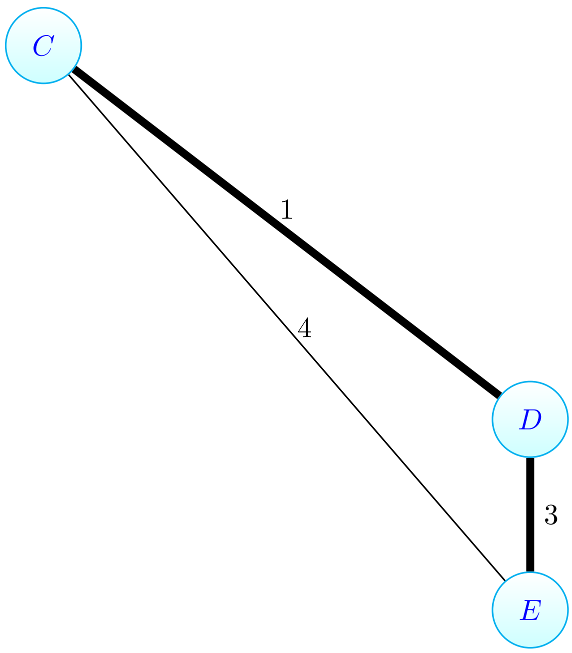
We now consider a second example of an intermediate spanning forest.

Example 57. Consider the following weighted graph $G(V, E, w)$. The edges in our intermediate spanning forest $\mathcal{F}$ are indicated by thick edges.



In order to be explicit, we identify the individual trees in $\mathcal{F}$:

- $T_1 = \{A, B\}$. That is, this tree consists of the vertices A and B, together with the edge $\{A, B\}$. As there is only one edge from G that can be included, T_1 is a minimum-weight spanning tree on the component induced by the vertex set $\{A, B\}$.
- $T_2 = \{H, F\}$. That is, this tree consists of the vertices H and F, together with the edge $\{H, F\}$. By the same reasoning as for T_1, we have that T_2 is a minimum-weight spanning tree on the component induced by the verted set $\{H, F\}$.
- $T_3 = \{C, D, E\}$. That is, this tree consists of the vertices C, D, E, together with the edges $\{C, D\}$ and $\{D, E\}$. Note that the component induced by $\{C, D, E\}$ is the cycle C_3 consisting of the edges $\{C, D\}, \{D, E\}, \{C, E\}$ (pictured below). The unique minimum-weight spanning tree on this component consists precisely of the edges $\{C, D\}$ and $\{D, E\}$.



We now introduce the notions of safe, light, and useless edges. Intuitively, a safe edge is an edge that can be added to an intermediate spanning forest in such a way that we can still find a minimum-weight spanning tree of G. While precise, this definition of safe edge does not provide a useful way to efficiently identify which edges to add. To this end, we have the notion of a light edge, which is a minimum-weight edge that crosses a partition of the vertices. We will show later that light edges are safe. In particular, any light edge that connects two components in an intermediate spanning forest is safe. An edge is useless if it will create a cycle. As trees do not contain cycles, we do not include useless edges in our minimum-weight spanning trees. We formalize these notions below.

Definition 58. Let $G(V, E, w)$ be a weighted graph, and let $\mathcal{F}$ be an intermediate spanning forest of G. Let $e = \{u, v\}$ be an edge of G.

(a) We say that e is *safe* with respect to $\mathcal{F}$ if $\mathcal{F} \cup e$ is a subset of some minimum-weight spanning tree of G.

(b) Let $S \subseteq V(G)$ be a set of vertices such that every edge $\{x, y\}$ in $\mathcal{F}$ has either $x, y \in S$ or $x, y \in V(G) \setminus S$. We say that e is a *light* edge e is a minimum weight edge with one endpoint in S and the other endpoint in $V(G) \setminus S$.

(c) We say that e is *useless* with respect to $\mathcal{F}$ if both u and v lie on the same tree in $\mathcal{F}$. Note that in this case, adding e to $\mathcal{F}$ creates a cycle.

(d) We say that e is *undecided* with respect to $\mathcal{F}$ if e is neither safe nor useless.

We now turn to showing that light edges are safe. The key proof technique is to start with a minimum-weight spanning tree T and exchange an edge that crosses the cut $(S, V(G) \setminus S)$ (as in the definition of a light edge) for our light edge. As removing any edge of T disconnects T, we are effectively exchanging a cut edge of T for another cut edge. This exchange is very similar as in the proof of Theorem 51. Now as our light edge is a minimum-weight edge that crosses the cut, replacing an appropriate edge of T with our light edge does not increase the weight of our modified tree. As T was assumed to be a minimum-weight spanning tree, we have that our modified tree is also a minimum-weight spanning tree.

Theorem 59. Let $G(V, E, w)$ be a weighted graph, and let $A \subseteq E(G)$ be a subset of edges that belong to some minimum-weight spanning tree of G. Let $S \subseteq V(G)$ such that every edge $e = \{x, y\} \in A$ has either $x, y \in S$ or $x, y \in V(G) \setminus S$ (that is, no edge of S crosses the cut $(S, V(G) \setminus S)$). If $\{u, v\}$ is an edge with one endpoint in S and the other endpoint in $V(G) \setminus S$, then $\{u, v\}$ is a safe edge with respect to A.

Proof. Let T be a minimum-weight spanning tree of G that contains the edges of A. If T contains $\{u, v\}$, then we are done. So suppose that T does not contain $\{u, v\}$. As T is a spanning tree of G, there exists an edge $e = \{a, b\}$ where one endpoint of e belongs to S, and the other endpoint belongs to $V(G) \setminus S$ (that is, e crosses the cut $(S, V(G) \setminus S)$). As T is a tree, every edge of T is a cut edge. So $T \setminus e$ contains two components, T_1 and T_2. As $\{u, v\}$ crosses the cut $(S, V(G) \setminus S)$, it follows (without loss of generality) that $u \in V(T_1)$ and $v \in V(T_2)$. So $T' := (T \setminus e) \cup \{u, v\}$ is a spanning tree of G. As $\{u, v\}$ is a light edge, $w(\{u, v\}) \leq w(e)$. So $w(T') \leq w(T)$. As T is a minimum-weight spanning tree, it follows that $w(T') = w(T)$. So T' is a minimum-weight spanning tree. As $A \cup \{\{u, v\}\} \subseteq E(T')$, we have that $\{u, v\}$ is safe, as desired. □

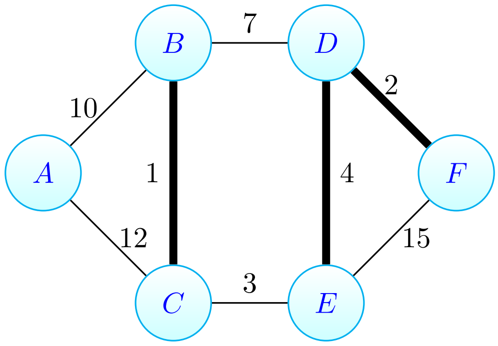
Remark 60. As a corollary, we obtain that a light edge connecting two components in an intermediate spanning forest $\mathcal{F}$ is safe with respect to $\mathcal{F}$. This allows us to easily identify safe edges. In addition, this corollary will be key in establishing the correctness of our minimum-weight spanning tree algorithms.

Corollary 61. Let $G(V, E, w)$ be a weighted graph, and let $\mathcal{F}$ be an intermediary spanning forest. Fix a tree T_i, and tet $e \in E(G)$ be a light edge with exactly one endpoint in T_i. Then e is safe.

Proof. Let A be the set of edges in $\mathcal{F}$. Let $S = V(T_i)$. We apply Theorem 59 with the edge set A and the vertex set S to obtain that e is safe with respect to $\mathcal{F}$. □

We now apply Corollary 61 to help find safe edges.

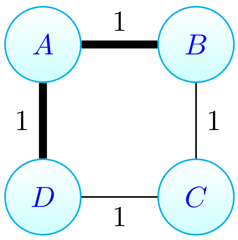
Example 62. Consider again the following weighted graph $G(V, E, w)$. The edges in our intermediate spanning forest $\mathcal{F}$ are indicated by thick edges.



We have the following.

- $\{A, B\}$ is the minimum-weight edge incident to $\{A\}$. Therefore, as $\{A, B\}$ is a light edge with exactly one endpoint belonging to $\{A\}$, we have by Corollary 61 that $\{A, B\}$ is **safe** with respect to $\mathcal{F}$.
- $\{C, E\}$ is the minimum-weight edge with exactly one endpoint in the component $\{B, C\}$ (as well as the minimum-weight edge with exactly one endpoint in the component $\{D, E, F\}$). Therefore, we have by Corollary 61 that $\{C, E\}$ is **safe** with respect to $\mathcal{F}$.
- While the edge $\{A, C\}$ connects the components $\{A\}$ and $\{B, C\}$, $\{A, C\}$ is not a minimum-weight edge doing so. Therefore, $\{A, C\}$ is **undecided** with respect to $\mathcal{F}$.
- While the edge $\{B, D\}$ connects the components $\{B, C\}$ and $\{D, E, F\}$, $\{B, D\}$ is not a minimum-weight edge doing so. Therefore, $\{B, D\}$ is **undecided** with respect to $\mathcal{F}$.
- The edge $\{E, F\}$ creates has both endpoints in the component $\{D, E, F\}$. So $\{E, F\}$ is **useless** with respect to $\mathcal{F}$.

Example 63. We note that if the edge weights are not distinct, then there may be multiple safe edges for an intermediate spanning forest $\mathcal{F}$. However, there are examples where we may only select one of the safe edges. Consider the cycle graph on four vertices, C_4, pictured below. The edges of our intermediate spanning forest $\mathcal{F}$ are indicated by thick edges. While both $\{B, C\}$ and $\{C, D\}$ are safe with respect to $\mathcal{F}$, only one of these edges may be added to $\mathcal{F}$. Adding both $\{B, C\}$ and $\{C, D\}$ would create a cycle.



4.3 Kruskal's Algorithm

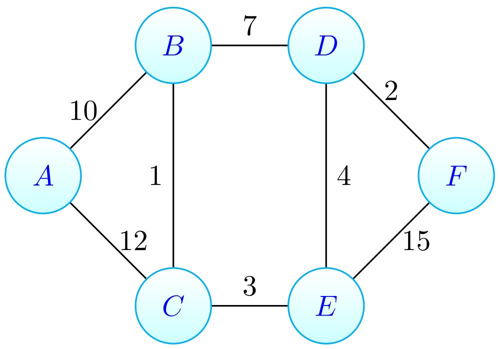
We briefly introduced Kruskal's algorithm in Section 4.1. In this section, we will examine Kruskal's algorithm in more detail. Recall that Kruskal's algorithm places the edges of the input graph into a priority queue. It then polls the edges one at a time, adding the edge e currently being considered to the intermediate spanning forest precisely if e connects two disjoint components. As the edges are sorted from lowest weight to highest weight, it follows that e is added precisely if there exists a component T where e is a light edge with exactly one endpoint in T. So by Corollary 61, e is added precisely if e is safe.

We formalize Kruskal's algorithm below.

Algorithm 6 Kruskal's Algorithm

```
procedure KRUSKAL(ConnectedWeightedGraph G(V, E, w))
    𝓕 ← (V(G), ∅)                 ▷ Initialize the Intermediate Spanning Forest to contain no edges.
    PriorityQueue Q ← []
    Q.addAll(E(G))
    while 𝓕.numEdges() < |V(G)| − 1 do
        {u, v} ← Q.poll()          ▷ Poll an edge and call the endpoints u and v
        if u and v are on different components of 𝓕 then
            𝓕.addEdge({u, v})
    return 𝓕
```

Example 64. We now work through Kruskal's algorithm on the following graph.

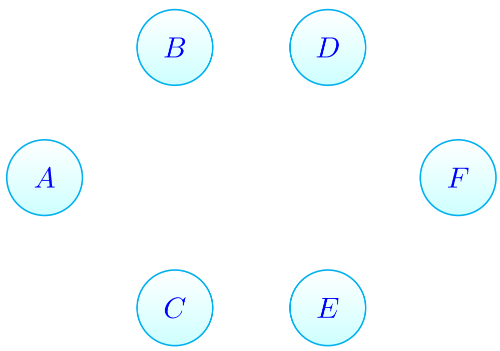


We proceed as follows.

1. We initialize the intermediate spanning forest $\mathcal{F}$ to be the empty graph (the graph on no edges). We also place the edges of G into a priority queue, which we call Q. So:

   $Q = [(\{B, C\}, 1), (\{D, F\}, 2), (\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)]$.

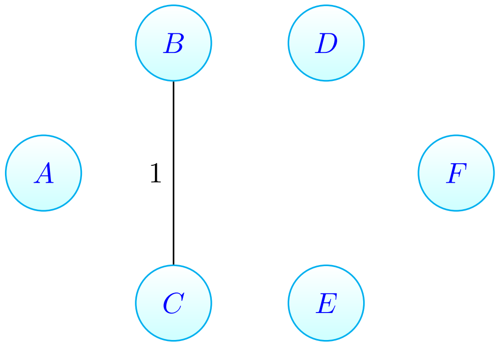
   Here, $(\{B, C\}, 1)$ indicates the edge $\{B, C\}$ has weight 1. The intermediate spanning forest $\mathcal{F}$ is pictured below.



2. We poll from Q, which returns the edge $\{B, C\}$. Note that $w(\{B, C\}) = 1$. As B and C are on different components of $\mathcal{F}$, we add the edge $\{B, C\}$ to $\mathcal{F}$. So:

   $$Q = [(\{D, F\}, 2), (\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

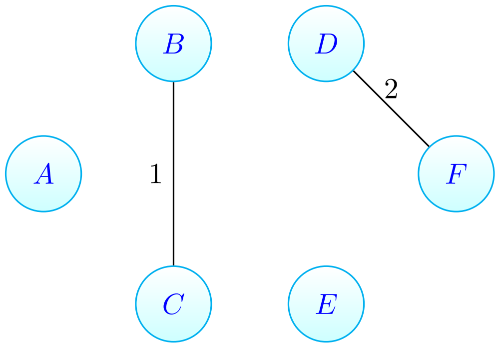
   and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



3. We poll from Q, which returns the edge $\{D, F\}$. Note that $w(\{D, F\}) = 2$. As B and C are on different components of $\mathcal{F}$, we add the edge $\{D, F\}$ to $\mathcal{F}$. So:

$$Q = [(\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

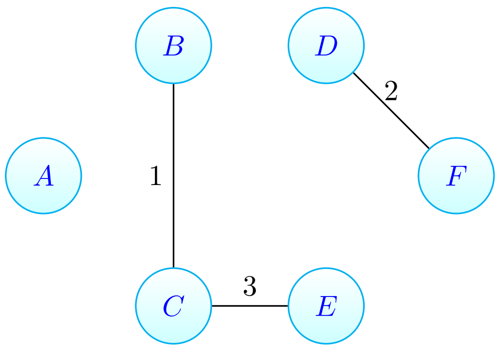
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



4. We poll from Q, which returns the edge $\{C, E\}$. Note that $w(\{C, E\}) = 3$. As C and E are on different components of $\mathcal{F}$, we add the edge $\{C, E\}$ to $\mathcal{F}$. So:

$$Q = [(\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

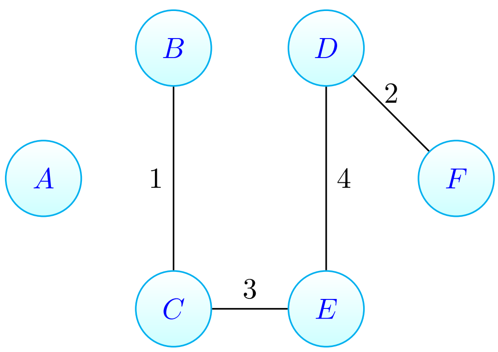
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



5. We poll from Q, which returns the edge $\{D, E\}$. Note that $w(\{D, E\}) = 4$. As D and E are on different components of $\mathcal{F}$, we add the edge $\{D, E\}$ to $\mathcal{F}$. So:

$$Q = [(\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



6. We poll from Q, which returns the edge $\{B, D\}$. Note that $w(\{B, D\}) = 7$. As B and D are on the same component of $\mathcal{F}$, we do **not** add the edge $\{B, D\}$ to $\mathcal{F}$. So:

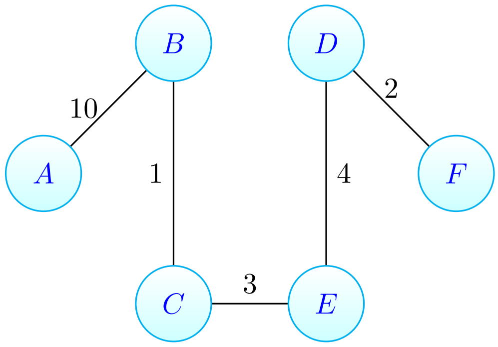
$$Q = [(\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

   and the intermediate spanning forest $\mathcal{F}$ remains unchanged from the previous iteration.

7. We poll from Q, which returns the edge $\{B, A\}$. Note that $w(\{B, A\}) = 10$. As B and A are on different components of $\mathcal{F}$, we add the edge $\{B, A\}$ to $\mathcal{F}$. So:

$$Q = [(\{A, C\}, 12), (\{E, F\}, 15)],$$

   and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



   As there are 6 vertices and $\mathcal{F}$ has 5 edges, Kruskal's algorithm terminates and returns $\mathcal{F}$, which is our minimum-weight spanning tree.

Remark 65. Now that we have worked through an example of Kruskal's algorithm, we wish to comment a bit about the algorithm provided (Algorithm 6). On line 7 of Algorithm 6, there is an `if` statement checking whether two vertices belong to the same connected component. For the purposes of this class, we will not examine the details associated with implementing this functionality. In practice, a Union-Find data structure is used to manage the intermediate spanning forest. We direct the reader to [CLRS09, Chapter 21] for details regarding the Union-Find data structure.

4.3.1 Kruskal's Algorithm: Proof of Correctness

We now turn to proving that Kruskal's algorithm returns a minimum-weight spanning tree. Precisely, we need to show the following.

(a) Kruskal's algorithm terminates.

(b) Kruskal's algorithm returns a spanning tree.

(c) The spanning tree that Kruskal's algorithm returns is of minimum weight.

We begin by showing that Kruskal's algorithm terminates.

Proposition 66. Let $G(V, E, w)$ be a finite, connected, weighted graph. Kruskal's algorithm terminates, when applied to G.

Proof. Kruskal's algorithm examines each edge of G at most once. As G has finitely many edges, the algorithm terminates. □

We next show that Kruskal's algorithm returns a spanning tree. We do this in two parts. First, we show that Kruskal's algorithm returns a spanning forest. Next, we show that the final spanning forest is indeed connected.

Proposition 67. Let $G(V, E, w)$ be a finite, connected, weighted graph. Kruskal's algorithm returns a spanning forest of G.

Proof. Kruskal's algorithm begins with a spanning forest $\mathcal{F}$ of G that has no edges (see line 2 of Algorithm 6). At each iteration of Kruskal's algorithm, at most one edge is added to $\mathcal{F}$. As a necessary condition for a given edge e to added to $\mathcal{F}$, it must be the case that e does not create a cycle in $\mathcal{F}$. So the graph that Kruskal's algorithm returns indeed spans G and is acyclic. Thus, Kruskal's algorithm returns a spanning forest of G, as desired. □

Proposition 67 provides that Kruskal's algorithm indeed returns a spanning forest of G. We show that this spanning forest is indeed a tree.

Proposition 68. Let $G(V, E, w)$ be a finite, connected, weighted graph. Kruskal's algorithm returns a spanning tree of G.

Proof. Let $\mathcal{F}$ be the spanning forest of G returned by Kruskal's algorithm. Suppose to the contrary that $\mathcal{F}$ is not connected (and therefore, not a tree). Let $T_1, \ldots, T_k$ be the connected components of $\mathcal{F}$. As G is connected, there exists an edge $e = \{u, v\}$ of G that does not belong to $\mathcal{F}$ and has endpoints in two distinct components of $\mathcal{F}$. We take $e = \{u, v\}$ to be such a minimum-weight edge. Let T_i be the component of $\mathcal{F}$ containing u, and let T_j be the component of $\mathcal{F}$ containing v. As u and v belong to different components of $\mathcal{F}$, adding e to $\mathcal{F}$ would not create a cycle. In particular, as e is a minimum-weight edge with endpoints in different components of $\mathcal{F}$, Kruskal's algorithm would have placed e into $\mathcal{F}$ (see lines 7-8 of Algorithm 6), contradicting the assumption that Kruskal's algorithm returned a spanning forest that was not a tree. The result follows. □

It remains to show that the spanning tree returned by Kruskal's algorithm is a minimum-weight spanning tree. Recall that Corollary 61 states that if there is a component T such that the edge e is a light edge with exactly one endpoint in T, then e is safe. We apply Corollary 61 and induction to show that Kruskal's algorithm returns a minimum-weight spanning tree.

Theorem 69. Let $G(V, E, w)$ be a finite, connected, weighted graph. Kruskal's algorithm returns a minimum-weight spanning tree of G.

Proof. Let $\mathcal{F}$ be the spanning forest that Kruskal's algorithm maintains. We show by induction on the number of iterations of Kruskal's algorithm that $\mathcal{F}$ is an intermediate spanning forest. Note that if $\mathcal{F}$ has $|V(G)| - 1$ edges, then $\mathcal{F}$ is a tree. So if $\mathcal{F}$ is an intermediate spanning forest with $|V(G)| - 1$ edges, then $\mathcal{F}$ is a minimum-weight spanning tree.

- **Base Case:** Prior to the first iteration of Kruskal's algorithm, $\mathcal{F}$ has no edges. So $\mathcal{F}$ is contained in some (and in fact, every) minimum-weight spanning tree of G. Thus, $\mathcal{F}$ is an intermediate spanning forest of G.

- **Inductive Hypothesis:** Suppose that at the start of iteration $m \geq 0$, $\mathcal{F}$ is an intermediate spanning forest of G.
- **Inductive Step:** Let $e = \{u, v\}$ be the edge polled from the priority queue at iteration m of Kruskal's algorithm, and let $\mathcal{F}$ be the intermediate spanning forest at the start of iteration m. We note that Kruskal's algorithm adds e to the intermediate spanning if and only if u and v belong to different components T_i and T_j of $\mathcal{F}$. So suppose Kruskal's algorithm adds e to $\mathcal{F}$. As the priority queue stores edges in order from smallest weight to largest weight, it follows that (without loss of generality) e is a minimum-weight edge with exactly one endpoint in T_i. So e is a light edge with exactly one endpoint in T_i. Thus, by Corollary 61, e is a safe edge and $\mathcal{F} \cup e$ is an intermediate spanning forest.

The result follows by induction. $\square$

4.3.2 Kruskal's Algorithm: Runtime Complexity

In this section, we turn to analyzing the runtime complexity of Kruskal's algorithm. We begin by sorting the edges of G from smallest weight to largest weight. Sorting the edges takes time $O(|E| \cdot \log(|E|))$, where $|E| := |E(G)|$. Now we note that there are at most $\binom{|V|}{2}$ edges in our input graph, where $|V| := |V(G)|$. So:

$$\begin{aligned}\log(|E|) &\leq \log\left(\binom{|V|}{2}\right) \\ &\leq \log(|V|^2) \\ &= 2\log(|V|).\end{aligned}$$

Thus, sorting the edges takes time $O(|E| \cdot \log(|V|))$. Next, we initialize a Union-Find data structure [CLRS09, Chapter 21] with each vertex of G as an isolated component. Here, we use the Union-Find to maintain the intermediate spanning forest. The Union-Find data structure has two keep operations, `find` and `union`, each of which runs in time $O(\alpha(n))$ (where n is the number of elements in the Union-Find). Here, $\alpha(n)$ is the *inverse Ackermann function*, which grows much more slowly than $O(\log(n))$. In the context of Kruskal's algorithm, n is the number of vertices in the graph. So both the `find` and `union` operations run in time $O(\alpha(|V|))$.

- The `find` operation takes as input a vertex v and returns a distinguished vertex x that lies on the same component as v. We note that if u and v belong to the same component, then $\texttt{find}(u) = \texttt{find}(v)$. Conversely, if $\texttt{find}(u) = \texttt{find}(v)$, then u and v lie on the same component.
- The `union` operation combines two disjoint components into one component.

For each edge of G, Kruskal's algorithm utilizes two `find` calls to determine if the endpoints of the edge belong to the same component. If the edges belong to different components, then Kruskal's algorithm makes one `union` call to merge the components. So the total complexity of processing the edges after they are sorted is $O(|E| \cdot \alpha(|V|))$. Thus, the total runtime complexity of Kruskal's algorithm is:

$$O(|E| \cdot (\log(V) + \alpha(|V|))) = O(|E| \cdot \log(|V|).$$

We record the complexity with the following theorem.

Theorem 70. Kruskal's algorithm runs in time $O(|E| \cdot \log(|V|)$.

4.4 Prim's Algorithm

In this section, we examine a second technique to construct minimum-weight spanning trees; namely, Prim's algorithm. We again start with the intermediate spanning forest $\mathcal{F}$ that contains all the vertices of our input graph $G(V, E, w)$, but none of the edges. While Kruskal's algorithm determines which edges to add to $\mathcal{F}$ by examining the entire graph, Prim's algorithm takes a more local perspective. We provide as input a specified source vertex $s \in V(G)$. Let T^* be the component of $\mathcal{F}$ that contains s. Prim's algorithm examines the edges of G that have exactly one endpoint in T^* and select a light edge e from these to add to $\mathcal{F}$. As e has exactly one endpoint in T^*, e connects two distinct components of $\mathcal{F}$. So by Corollary 61, e is a safe edge with respect to $\mathcal{F}$. This is the key observation in establishing that Prim's algorithm returns a minimum-weight spanning tree.

We now turn to formalizing Prim's algorithm.

Algorithm 7 Prim's Algorithm

```
 1: procedure Prim(ConnectedWeightedGraph G(V, E, w), Vertex source)
 2:     F ← (V(G), ∅)                  ▷ Initialize the Intermediate Spanning Forest to contain no edges.
 3:     PriorityQueue Q ← []

 4:     for each edge e ∈ E(G) do
 5:         e.processed ← false

 6:     for each edge e incident to source do
 7:         Q.add(e)
 8:         e.processed ← true

 9:     while F.numEdges() < |V(G)| − 1 do
10:         {u, v} ← Q.poll()                    ▷ Poll an edge and call the endpoints u and v
11:         T_u ← F.componentContaining(u)
12:         T_v ← F.componentContaining(v)

13:         if T_u ≠ T_v then                    ▷ Check that u and v belong to different components
14:             F.addEdge({u, v})

15:             if source ∈ T_u then             ▷ If v was added to the component containing source
16:                 for each unprocessed edge e incident to v do
17:                     Q.add(e)                 ▷ Then add to Q each unprocessed edge incident to v

18:             else                             ▷ If u was added to the component containing source
19:                 for each unprocessed edge e incident to u do
20:                     Q.add(e)                 ▷ Then add to Q each unprocessed edge incident to u

        return F
```

We associate to each edge an attribute `processed` to indicate whether that edge has been placed into the priority queue. This ensures that each edge is considered at most once, which helps ensure that the algorithm will terminate.

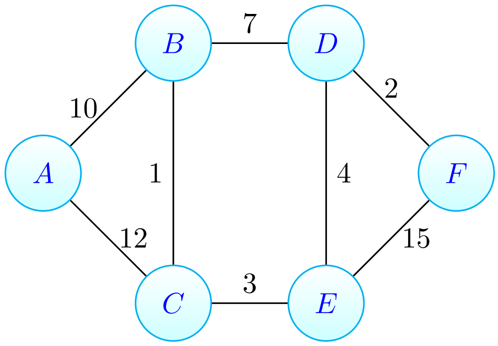
Now at lines 6-8, we initialize the priority queue to contain only edges that are incident to the source vertex. This ensures that the first edge placed into the intermediate spanning forest is incident to the source vertex. Now by adding an edge to $\mathcal{F}$, we introduce a new vertex v to the component containing our source vertex. Prim's algorithm then adds to the prioritiy queue the edges incident to v, provided such edges have not already been polled from the queue. So the `while` loop at line 9 preserves the invariant that every edge in the priority queue has at least one endpoint in the component containing our source vertex.

Prim's algorithm only adds an edge if it connects two components. Such an edge e is polled from the priority queue, and so (i) has an endpoint in the component containing the source vertex, and (ii) is a minimum-weight edge connecting two distinct components. Therefore, e is a safe edge.

4.4.1 Prim's Algorithm: Example 1

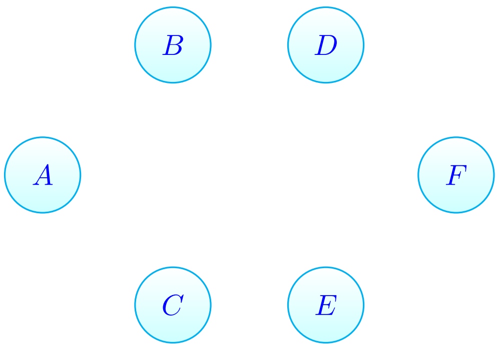
We now work through an example of Prim's algorithm.

Example 71. Consider the following graph $G(V, E, w)$ pictured below. Suppose we select the source vertex A.

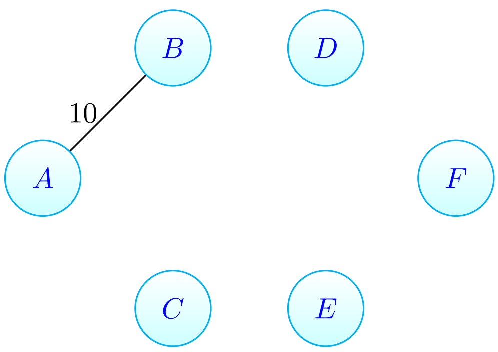


Prim's algorithm proceeds as follows.

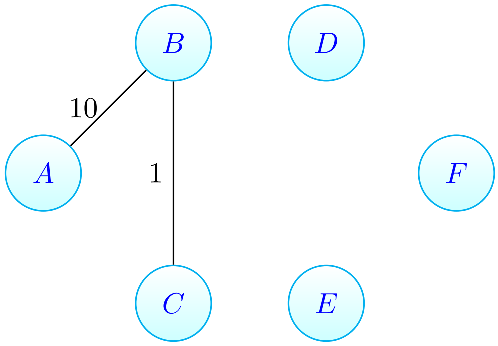
1. We initialize the intermediate spanning forest to contain all the vertices of G, but no edges. We then initialize the priority queue to contain the edges incident to our source vertex A. So:
$$Q = [(\{A, B\}, 10), (\{A, C\}, 12)],$$
and our intermediate spanning forest $\mathcal{F}$ is pictured below.



2. We poll the edge $\{A, B\}$ from the queue and mark $\{A, B\}$ as processed. Note that $w(\{A, B\}) = 10$. As $\{A, B\}$ has exactly one endpoint on the component containing A (which is the isolated vertex A), we add $\{A, B\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to B. So:
$$Q = [(\{B, C\}, 1), (\{B, D\}, 7), (\{A, C\}, 12)],$$
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



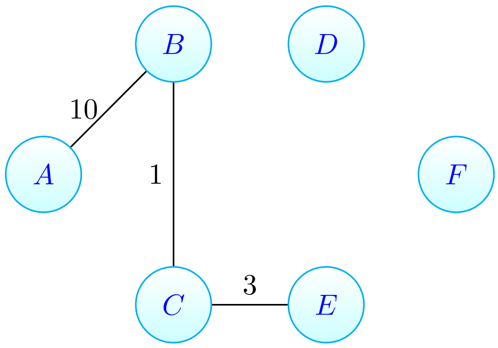
3. We poll the edge $\{B, C\}$ from the queue and mark $\{B, C\}$ as processed. Note that $w(\{B, C\}) = 1$. As $\{B, C\}$ has exactly one endpoint on the component containing A (which is the isolated vertex $\{A, B\}$), we add $\{B, C\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to C (provided said edges are not already in the priority queue). So:
$$Q = [(\{C, E\}, 3), (\{B, D\}, 7), (\{A, C\}, 12)],$$
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



4. We poll the edge $\{C, E\}$ from the queue and mark $\{C, E\}$ as processed. Note that $w(\{C, E\}) = 3$. As $\{C, E\}$ has exactly one endpoint on the component containing A, we add $\{C, E\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to E (provided said edges are not already in the priority queue). So:

$$Q = [(\{E, D\}, 4), (\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

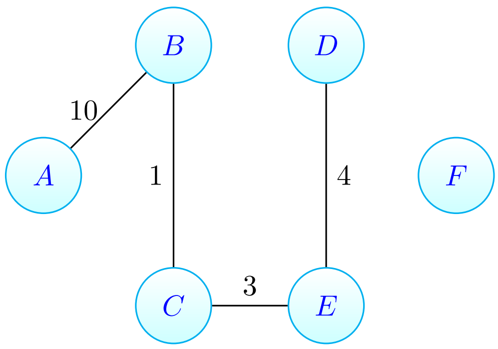
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



5. We poll the edge $\{E, D\}$ from the queue and mark $\{E, D\}$ as processed. Note that $w(\{E, D\}) = 4$. As $\{E, D\}$ has exactly one endpoint on the component containing A, we add $\{E, D\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to D (provided said edges are not already in the priority queue). So:

$$Q = [(\{D, F\}, 2), (\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

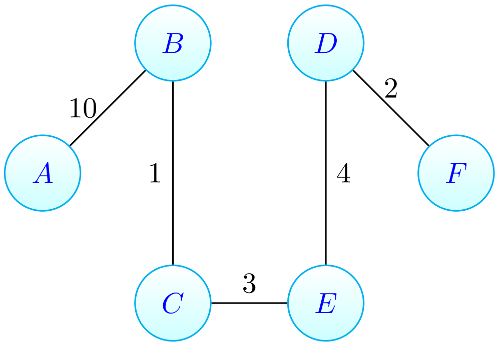
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



6. We poll the edge $\{D, F\}$ from the queue and mark $\{D, F\}$ as processed. Note that $w(\{D, F\}) = 2$. As $\{D, F\}$ has exactly one endpoint on the component containing A, we add $\{D, F\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to D (provided said edges are not already in the priority queue). So:

$$Q = [(\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



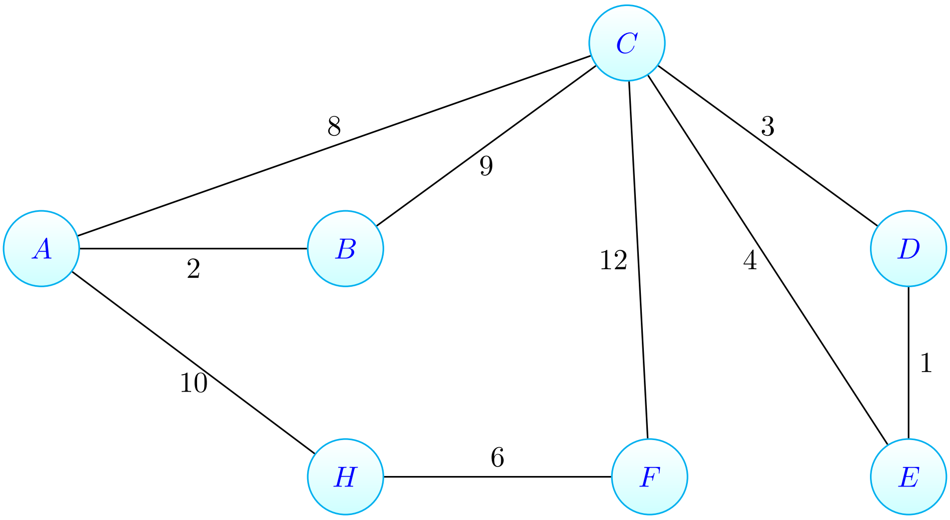
7. As $\mathcal{F}$ has $|V(G)| - 1 = 6 - 1 = 5$ edges, the algorithm terminates and returns $\mathcal{F}$, pictured in Step 6 immediately above.

Remark 72. We note that the minimum-weight spanning tree constructed by Kruskal's algorithm in Example 64 is the same tree that Prim's algorithm constructed in Example 71. For this input graph, the edge weights were distinct. Therefore, the graph had only one minimum-weight spanning tree. In general, Prim's algorithm and Kruskal's do not construct the same minimum-weight spanning tree.

4.4.2 Prim's Algorithm: Example 2

We consider a second example of Prim's algorithm.

Example 73. Consider the following graph $G(V, E, w)$. We execute Prim's algorithm, using the source vertex D.

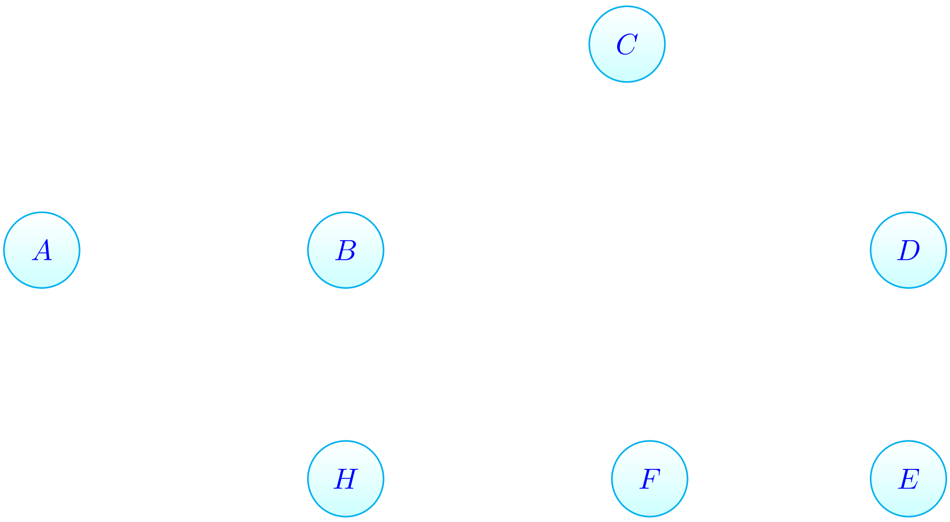


Prim's algorithm proceeds as follows.

1. We initialize the intermediate spanning forest to contain all the vertices of G, but no edges. We then initialize the priority queue to contain the edges incident to our source vertex A. So:

$$Q = [(\{D, E\}, 1), (\{D, C\}, 3)],$$

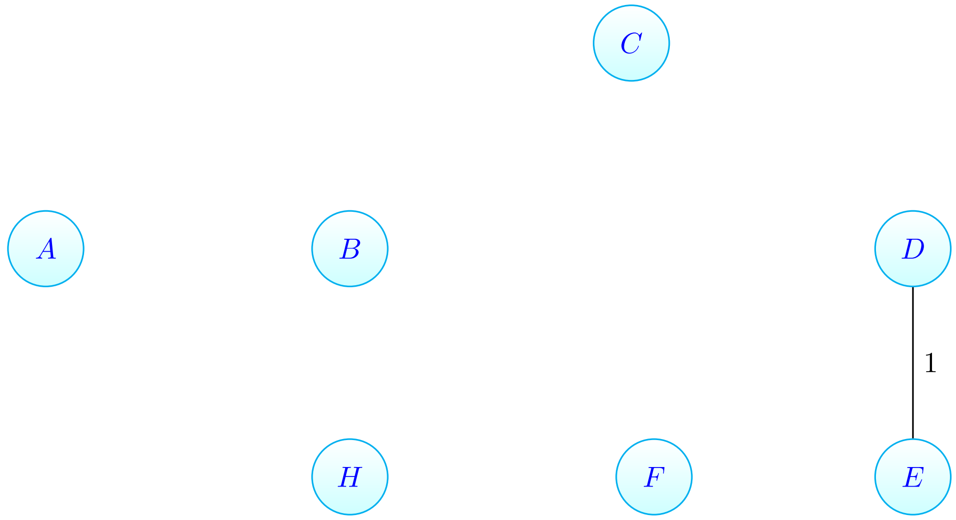
and our intermediate spanning forest $\mathcal{F}$ is pictured below.



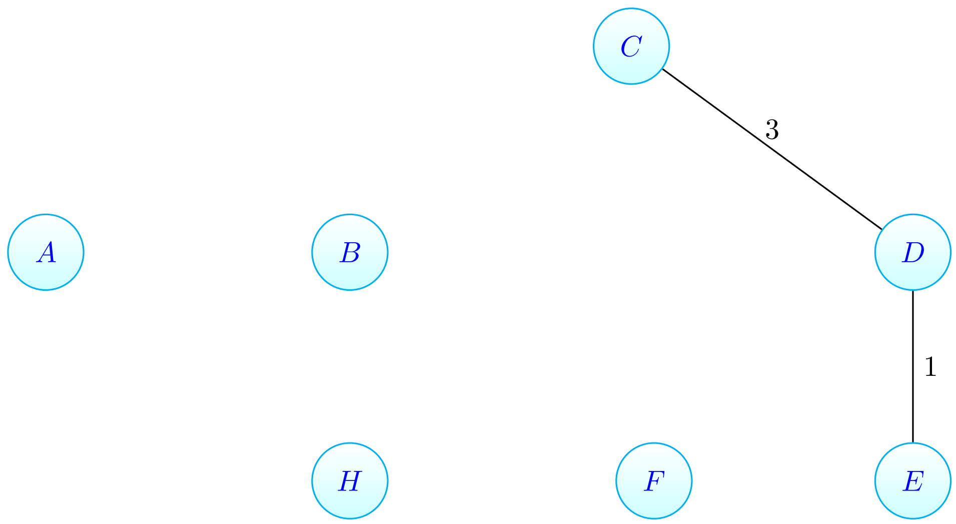
2. We poll the edge $\{D, E\}$ from the queue and mark $\{D, E\}$ as processed. Note that $w(\{C, E\}) = 1$. As $\{D, E\}$ has exactly one endpoint on the component containing D, we add $\{D, E\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to E (provided said edges are not already in the priority queue). So:

$$Q = [(\{D, C\}, 3), (\{E, C\}, 4)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.

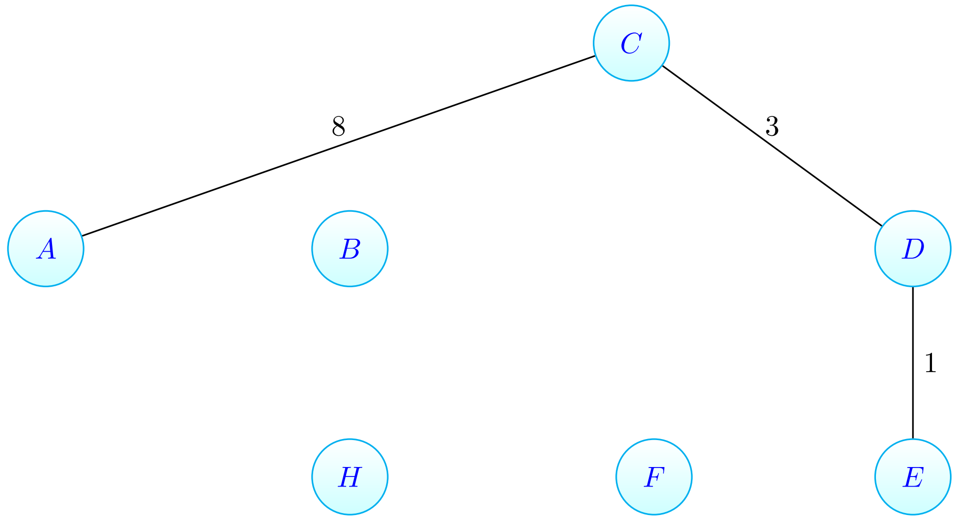


3. We poll the edge $\{D, C\}$ from the queue and mark $\{D, C\}$ as processed. Note that $w(\{D, C\}) = 3$. As $\{D, C\}$ has exactly one endpoint on the component containing D, we add $\{D, C\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to C (provided said edges are not already in the priority queue). So:
$$Q = [(\{E, C\}, 4), (\{C, A\}, 8), (\{C, B\}, 9), (\{C, F\}, 12)],$$
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.

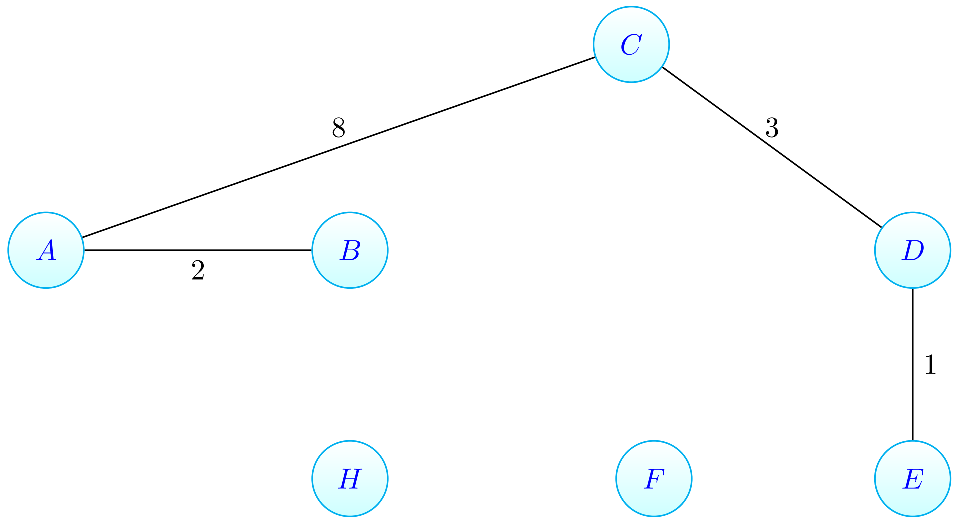


4. We poll the edge $\{E, C\}$ from the queue and mark $\{E, C\}$ as processed. As both E and C belong to the component containing D, we do **not** add $\{E, C\}$ to $\mathcal{F}$. The updated priority queue is below, and the intermediate spanning forest $\mathcal{F}$ does not change from the previous iteration.
$$Q = [(\{C, A\}, 8), (\{C, B\}, 9), (\{C, F\}, 12)].$$

5. We poll the edge $\{C, A\}$ from the queue and mark $\{C, A\}$ as processed. Note that $w(\{C, A\}) = 8$. As $\{C, A\}$ has exactly one endpoint on the component containing D, we add $\{C, A\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to A (provided said edges are not already in the priority queue). So:
$$Q = [(\{A, B\}, 2), (\{C, B\}, 9), (\{A, H\}, 10), (\{C, F\}, 12)],$$
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.

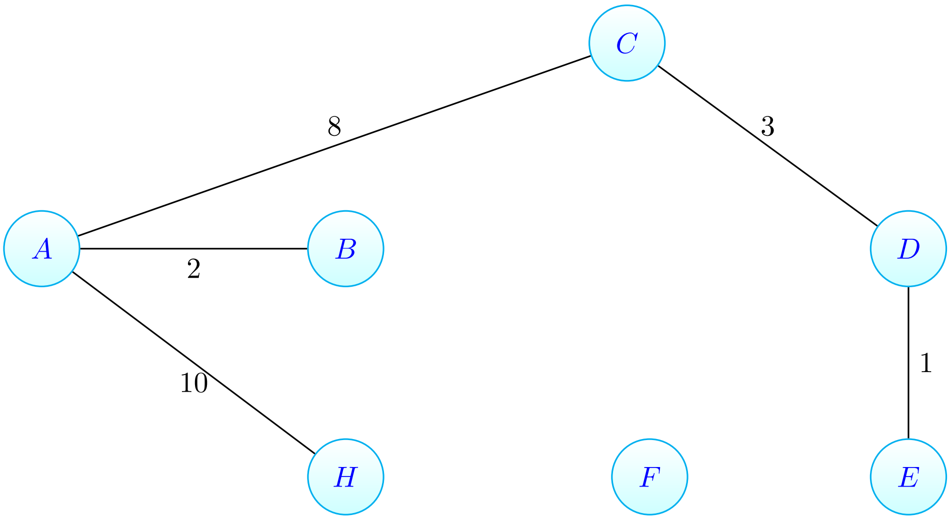


6. We poll the edge $\{A, B\}$ from the queue and mark $\{A, B\}$ as processed. Note that $w(\{A, B\}) = 2$. As $\{A, B\}$ has exactly one endpoint on the component containing D, we add $\{A, B\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to B (provided said edges are not already in the priority queue). So:
$$Q = [(\{C, B\}, 9), (\{A, H\}, 10), (\{C, F\}, 12)],$$
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



7. We poll the edge $\{C, B\}$ from the queue and mark $\{C, B\}$ as processed. As both C and B belong to the component containing D, we do **not** add $\{C, B\}$ to $\mathcal{F}$. The updated priority queue is below, and the intermediate spanning forest $\mathcal{F}$ does not change from the previous iteration.
$$Q = [(\{A, H\}, 10), (\{C, F\}, 12)],$$

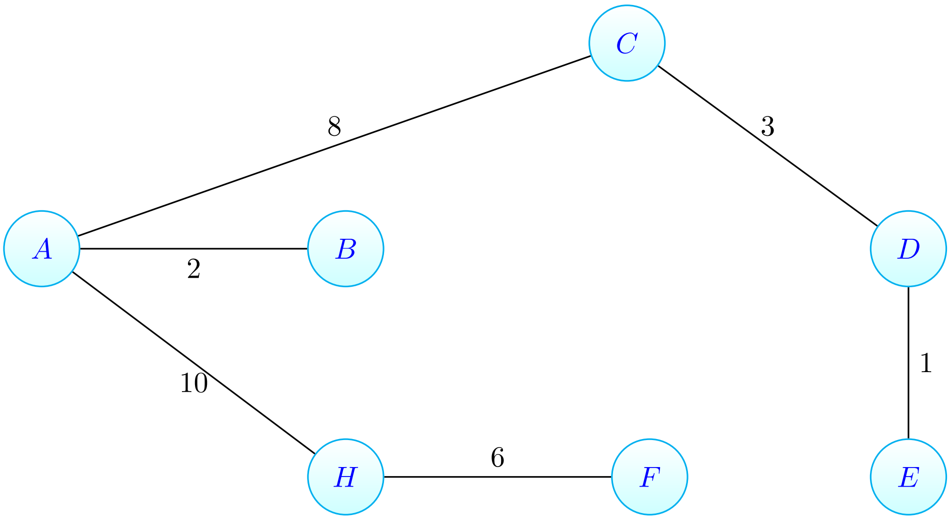
8. We poll the edge $\{A, H\}$ from the queue and mark $\{A, H\}$ as processed. Note that $w(\{A, H\}) = 10$. As $\{A, H\}$ has exactly one endpoint on the component containing D, we add $\{A, H\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to H (provided said edges are not already in the priority queue). So:
$$Q = [(\{H, F\}, 6), (\{C, F\}, 12)],$$
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



9. We poll the edge $\{H, F\}$ from the queue and mark $\{H, F\}$ as processed. Note that $w(\{H, F\}) = 6$. As $\{H, F\}$ has exactly one endpoint on the component containing D, we add $\{A, H\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to H (provided said edges are not already in the priority queue). So:

$$Q = [(\{C, F\}, 12)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



As $\mathcal{F}$ has $|V(G)| - 1 = 7 - 1 = 6$ edges, the algorithm terminates and returns $\mathcal{F}$, pictured in Step 8 immediately above.

4.4.3 Algorithm 7 Correctly Implements Prim's Algorithm

Prim's algorithm works by, at a given iteration, adding a light edge with exactly one endpoint in the component containing the source vertex. We proposed an implementation of Prim's algorithm with Algorithm 7. In this section, we show that Algorithm 7 adds at each iteration a light edge with exactly one endpoint in the component containing the source vertex. That is, we show that Algorithm 7 correctly implements Prim's algorithm. We are **not** showing in this section that Prim's algorithm returns a minimum-weight spanning tree; those details will be covered in the next section.

We first show that Algorithm 7 only adds edges to the priority queue where at least one endpoint is contained in same component as the source vertex.

Lemma 74. Prior to the start of iteration $i \geq 0$ of the `while` loop on line 9 of Algorithm 7, the priority queue Q only contains edges where at least one endpoint is in the same connected component as the source vertex.

Proof. The proof is by induction on the number of iterations of the `while` loop on line 9.

- **Base Case:** Prior to the start of the `while` loop, the edges incident to the source vertex are placed in the priority queue (see lines 6-7). So the priority queue contains edges where at exactly one endpoint is in the same connected component as the source vertex.
- **Inductive Hypothesis:** Fix $k \geq 0$. Suppose that prior to the start of iteration k of the `while` loop on line 9, that the priority queue Q contains only edges where at least one endpoint of each edge belongs to the same connected component as the source vertex.
- **Inductive Step:** Let $e = \{u, v\}$ be the edge polled at iteration k of the `while` loop. By the inductive hypothesis, at least one endpoint of e belongs to the same connected component as the source vertex. We have two cases.
  - **Case 1:** Suppose both u and v belong to the same connected component as the source vertex. In this case, the condition of the `if` statement on line 13 is not satisfied. So e is not added to $\mathcal{F}$ and no new edges are added to the priority queue Q. By the inductive hypothesis, Q contains only edges where at least one endpoint of each edge belongs to the same connected component as the source vertex, which remains true prior to the start of iteration $k + 1$ of the `while` loop.
  - **Case 2:** Suppose exactly one endpoint of e belongs to the same connected component as the source vertex. Let T_u and T_v be the components of $\mathcal{F}$ containing u and v respectively. As exactly one endpoint of e belongs to the same connected component as the source vertex, we have that $T_u \neq T_v$. So the condition of the `if` statement on line 13 is satisfied. Thus, Algorithm 7 adds e to $\mathcal{F}$. Now if source is on the same component as u (that is, source $\in T_u$), then the algorithm adds the unprocessed edges incident to v to the priority queue (provided said edges are not already in the priority queue). By similar argument, if instead source $\in T_v$, then the algorithm adds the unprocessed edges incident to u to the priority queue (provided said edges are not already in the priority queue). In either case, Q contains only edges where at least one endpoint of each edge belongs to the same connected component as the source vertex, which remains true prior to the start of iteration $k+1$ of the `while` loop.

The result follows by induction. □

Remark 75. In the proof of Lemma 74, we showed that Algorithm 7 adds the edge e being considered at the given iteration to $\mathcal{F}$ precisely if e has exactly one endpoint in the connected component containing the source vertex. As the edges are sorted in from smallest weight to largest weight, it follows that Algorithm 7 adds e to $\mathcal{F}$ precisely if e is a light edge that has exactly one endpoint in the connected component containing the source vertex. This observation, together with Lemma 74 yields the following.

Theorem 76. Algorithm 7 correctly implements Prim's algorithm.

4.4.4 Prim's Algorithm: Proof of Correctness

In this section, we establish the correctness of Prim's algorithm. That is, we show that Prim's algorithm returns a minimum-weight spanning tree. Just as with Kruskal's algorithm, we need to show the following.

(a) Prim's algorithm terminates.

(b) Prim's algorithm returns a spanning tree.

(c) The tree that Prim's algorithm returns is of minimum-weight.

We begin by showing that Prim's algorithm terminates.

Proposition 77. Let $G(V, E, w)$ be a connected, weighted graph. Prim's algorithm terminates, when applied to G.

Proof. Prim's algorithm examines each edge of G at most once. As G has finitely many edges, the algorithm terminates. $\square$

We next show that Prim's algorithm returns a spanning tree. We do this in two parts. First, we show that Prim's algorithm returns a spanning forest. Second, we show that the spanning forest is connected.

Proposition 78. Let $G(V, E, w)$ be a connected, weighted graph. Prim's algorithm returns a spanning forest of G.

Proof. Prim's algorithm initializes a spanning forest $\mathcal{F}$ of G that has no edges (see line 2 of Algorithm 7). At each iteration of Prim's algorithm, at most one edge is added to $\mathcal{F}$. As a necessary condition for Prim's algorithm to add a given edge e, it must be the case that the endpoints of e belong to different components of $\mathcal{F}$. Thus, if an edge e is added to $\mathcal{F}$, then $\mathcal{F} \cup e$ does not contain a cycle. So the graph that Prim's algorithm returns indeed spans G and is acyclic. Thus, Prim's algorithm returns a spanning forest of G, as desired. $\square$

We now show that the spanning forest $\mathcal{F}$ that Prim's algorithm returns is indeed a tree.

Proposition 79. Let $G(V, E, w)$ be a connected, weighted graph. Prim's algorithm returns a spanning tree of G.

Proof. Let $\mathcal{F}$ be the spanning forest of G that Prim's algorithm returns. Suppose to the contrary that $\mathcal{F}$ is not connected. Let T_{source} be the component of $\mathcal{F}$ associated with the source vertex, and let $T_1, \ldots, T_k$ be the remaining components of $\mathcal{F}$. As G is connected, there exists an edge $e \in E(G)$ such that one endpoint is in T_{source} and the other endpoint belongs to T_i for some $i \in \{1, \ldots, k\}$. In particular, there exists an edge e of minimum weight edge with one endpoint in T_{source} and the other endpoint in T_i for some $i \in \{1, \ldots, k\}$. However, Prim's algorithm would have added e to $\mathcal{F}$, contradicting the assumption that T_{source} and T_i were not connected. The result follows. $\square$

It remains to show that the tree returned by Prim's algorithm is a minimum-weight spanning tree. Recall that Corollary 61 states that if for a component T, e is a light edge with exactly one endpoint in T, then e is a safe edge with respect to the given intermediate spanning forest. We apply Corollary 61 and induction to show that Prim's algorithm returns a minimum-weight spanning tree.

Theorem 80. Let $G(V, E, w)$ be a connected, weighted graph. Prim's algorithm returns a minimum-weight spanning tree of G.

Proof. Let $\mathcal{F}$ be the spanning forest that Prim's algorithm maintains. We show by induction on the number of iterations of Prim's algorithm that $\mathcal{F}$ is an intermediate spanning forest. Note that if $\mathcal{F}$ has $|V(G)| - 1$ edges, then $\mathcal{F}$ is a tree. So if $\mathcal{F}$ is an intermediate spanning forest with $|V(G)| - 1$ edges, then $\mathcal{F}$ is a minimum-weight spanning tree.

- **Base Case:** Prior to the start of the first iteration, $\mathcal{F}$ has no edges. So $\mathcal{F}$ is contained in some (and in fact, every) minimum-weight spanning tree of G. Thus, $\mathcal{F}$ is an intermediate spanning forest of G.
- **Inductive Hypothesis:** Suppose that at the start of iteration $m \geq 0$, $\mathcal{F}$ is an intermediate spanning forest of G.

- **Inductive Step:** Let $e = \{u, v\}$ be the edge polled from the priority queue at iteration m of Prim's algorithm, and let $\mathcal{F}$ be the intermediate spanning forest at the start of iteration m. We note that Prim's algorithm adds e to $\mathcal{F}$ if and only if exactly one endpoint of e belongs to the same component as the source vertex. By Lemma 74, at least one endpoint of e belongs to the same component as the source vertex. So we have two cases.
  - **Case 1:** Suppose that both endpoints of e belong to the same component as the source vertex. In this case, e is not added to $\mathcal{F}$. So no changes to $\mathcal{F}$ are made, and we retain $\mathcal{F}$ as our intermediate spanning forest at the start of iteration $m + 1$.
  - **Case 2:** Suppose that exactly one endpoint of e belongs to the same component as the source vertex. As the priority queue stores edges in order from smallest weight to largest weight, it follows that e is a minimum-weight edge with exactly one endpoint in the component containing the source vertex. So by Corollary 61, e is a safe edge and $\mathcal{F} \cup e$ is an intermediate spanning forest.

The result follows by induction. $\square$

4.4.5 Prim's Algorithm: Runtime Complexity

We now turn to analyzing the runtime complexity of our implementation (Algorithm 7) of Prim's algorithm. We make several assumptions. First, we assume that the graph is provided as an adjacency list. Second, we use a standard binary heap to implement the priority queue. Both the `insert` and `poll` operations take time $O(\log(m))$, where m is the number of elements in the priority queue. Now the worst case complexity of the binary heap's `lookup` operation is $O(m)$. We circumvent this by using a `processed` attribute for each edge, to indicate whether that edge was placed into the priority queue. Algorithm 7 does not place processed edges back into the priority queue, which ensures that each edge is examined at most once. We stress that as the priority queue is storing edges, $m := |E|$.

Now as with Kruskal's algorithm, we use the Union-Find data structure [CLRS09, Chapter 21] to implement the intermediate spanning forest. Recall that both the `union` and `find` operations take $O(n \cdot \alpha(n))$ time, where $\alpha(n)$ is the *inverse Ackermann function*, which grows more slowly than $O(\log(n))$. We stress that as the Union-Find data structure is storing vertices, that $n := |V|$.

With our assumptions in tow, we begin analyzing the runtime complexity of Algorithm 7. Note that in the worst case for a given edge $e = \{u, v\}$, we do the following:

- Place e into the priority queue,
- Remove e from the priority queue,
- Perform two `find` calls on the Union-Find, using the endpoints u and v (see lines 11-12).
- Perform a `union` operation if u and v belong to different components (see line 14).

Thus, for a given edge, the worst case runtime complexity is $O(\log(|E|) + \alpha(|V|))$. An edge $e = \{u, v\}$ is examined twice. The first time occurs when we examine the edges incident to u, and the second time is when we examine the edges incident to v. Note that we are using the adjacency list structure at lines 15-20, in that we can examine each edge incident to a given vertex without traversing through the entire vertex set (which would be necessary if we were using an adjacency matrix representation of hte graph). Thus, in the worst case, Algorithm 7 takes time:

$$\begin{aligned} O(2|E| \cdot (\log(|E|) + \alpha(|V|))) &= O(|E| \cdot (\log(|E|) + \alpha(|V|))) \quad (13) \\ &= O(|E| \cdot \log(|E|)) \quad (14) \\ &= O(|E| \cdot \log(|V|)). \quad (15) \end{aligned}$$

Here, line (14) follows from the fact that $\alpha(|V|) \in O(\log(|V|)) \subseteq O(\log(|E|))$. Line (15) follows from the fact that $|E| \leq \binom{|V|}{2}$; so $\log(|E|) \leq 2\log(|V|) \in O(\log(|V|))$. Thus, we have the following.

Theorem 81. Prim's algorithm has the worst case time complexity $O(|E| \cdot \log(|V|))$ when using an adjacency list for the graph, a binary heap to support the priority queue, and a Union-Find data structure to implement the intermediate spanning forest.

5 Network Flows

In this chapter, we examine the notion of flows on networks. Intuitively, network flows are used to model settings such as routing oil from drilling platforms to shore, sending packets of information over the internet, routing electricity throughout a city, and scheduling. Here, our networks are directed graphs. Each edge (u, v) of the directed graph has a capacity $c((u, v))$, which is the amount of flow that can be pushed through the edge from u to v. Note that flow is directional. Intuitively, flow can only move in one direction, such as pushing oil through a pipe. It is not possible to push oil both forwards and backwards along the same pipe.

5.1 Framework

We first formalize the network flows framework.

Definition 82. Let $G(V, E)$ be a connected, directed graph. A *source* vertex s is a vertex with in-degree 0; that is, s has no incoming edges. A *sink* vertex t is a vertex with out-degree 0; that is, t has no outgoing edges.

Remark 83. Informally, a source vertex is where flow originates, and a sink vertex is a destination point. The goal of the flow problem is to push as much flow as possible from the source vertices to the sink vertices.

Definition 84. A *flow network* $\mathcal{N}(G, c, S, T)$ consists of a directed graph $G(V, E)$, together with a set S of source vertices and a set T of sink vertices. We note that S and T are disjoint sets. $\mathcal{N}$ also has a *capacity function* $c : V(G) \times V(G) \to [0, \infty)$, such that if the ordered pair $(u, v) \in V(G) \times V(G)$ is not an edge, then $c((u, v)) = 0$. This condition ensures that we can only push flow along edges.

Remark 85. For undirected graphs, we denote edges as sets of the form $\{u, v\}$. Recall that a set is a collection of distinct, unordered elements. For undirected graphs, it is not the case that there are designated starting points and ending points along edges. So $\{u, v\} = \{v, u\}$. In the case of directed graphs, each edge has a clear starting and ending point. For this reason, we denote edges as *ordered pairs*. The edge (u, v) indicates that u is the starting vertex and v is the ending vertex. Similarly, (v, u) has v as the starting point and u as the ending vertex. So we note that $(u, v) \neq (v, u)$.

Our goal is to push flow across a flow network. To this end, we first formalize the notion of flow.

Definition 86. Let $\mathcal{N}(G, c, S, T)$ be a flow network. A *flow* is a function $f : V(G) \times V(G) \to [0, \infty)$ satisfying the following.

- For any ordered pair $(u, v) \times V(G) \times V(G)$, $f((u, v)) \leq c((u, v))$. That is, we cannot push more flow across a given edge than the capacity permits. Similarly, if the pair $(u, v) \not\in E(G)$, then we cannot push flow across (u, v). That is, we can only push flow along edges.
- For any pair (u, v), we have that $f((u, v)) = -f((v, u))$. Informally, this condition ensures that if flow is being pushed from u to v, that we can reroute the flow at u to avoid pushing flow to v.
- We have a conservation of flow condition. For every vertex v that is neither a source vertex nor a sink vertex, we have that the amount of flow coming into v is the amount of flow leaving v.

$$\sum_{(u,v) \in E(G)} f((u, v)) = \sum_{(v,w) \in E(G)} f((v, w)).$$

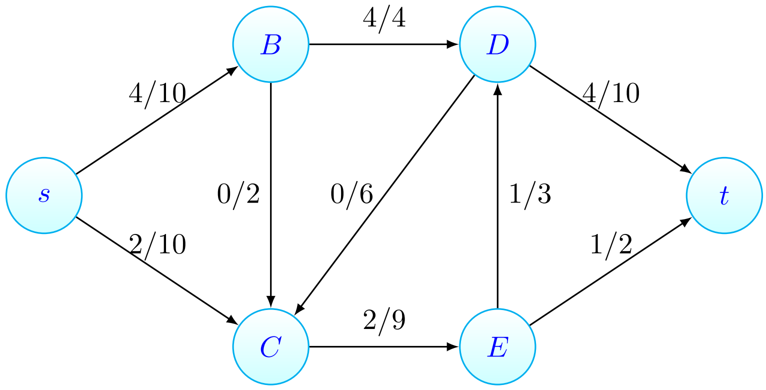
The *value* of f, denoted $\text{val}(f)$ or $|f|$, is the amount of flow that is moved from the source nodes to the sink nodes. By the conservation of flow condition, $|f|$ is precisely the amount of flow leaving the source nodes.

Remark 87. The condition that for any pair (u, v), we have that $f((u, v)) = -f((v, u))$ will become important later when we discuss algorithmic approaches to pushing as much flow as possible from the source to the sink vertices. This condition is not explicitly represented in how we will depict flow networks, so we will not discuss this condition further in this section.

We now turn to considering examples of flow networks and flows.

Example 88. Consider the following flow network $\mathcal{N}(G, c, S, T)$. We indicate both the flow and the capacity of a given edge using the labeling convention `flow/capacity`. For instance:

- The edge (s, A) has capacity 10, and there are 4 units of flow being pushed from $s \to A$.
- The edge (A, B) has capacity 4, and there are 4 units of flow being pushed from $A \to B$.
- The edge (D, t) has capacity 10, and there are 4 units of flow being pushed from $D \to t$.
- The edge (C, E) has capacity 9, and there is 1 unit of flow being pushed from $C \to E$.

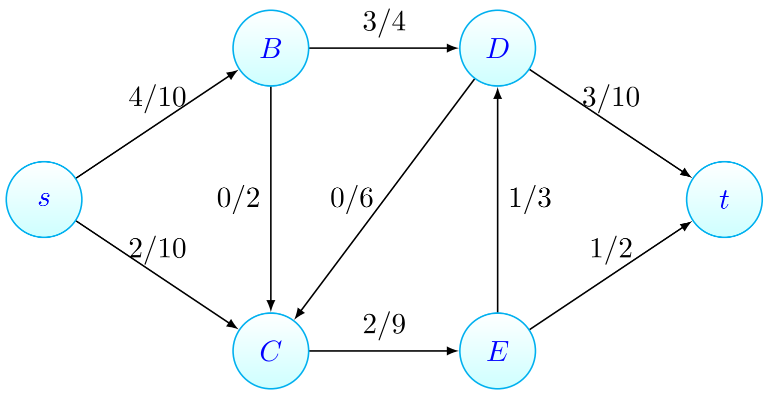


Additionally, observe that for each vertex v other than the source s and the sink t, the flow coming into v is equal to the flow leaving v. We check this for the vertices other than s and t.

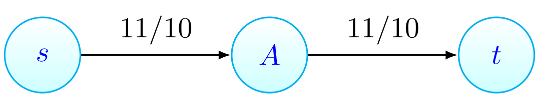
- Observe that there are 4 units of flow coming into vertex B via the edge (s, B). These 4 units of flow leave B along the edge (B, D).
- Observe that there are 4 units of flow coming into D via the edge (B, D). These 4 units of flow leave D along the edge (D, t).
- Observe that there are 2 units of flow coming into C via the edge (s, C). These 2 units of flow leave C along the edge (C, E).
- Observe that there are 2 units of flow coming into E via the edge (C, E). Precisely 1 unit leaves E along the edge (E, D), and the other 1 unit of flow leaves E along the edge (E, T).

We now consider examples of configurations that fail to be valid flows.

Example 89. Consider the following flow network. The flow configuration below is **invalid**, as flow is not conserved at vertex B. Here, 4 units of flow enter B through the edge (s, B). However, only 3 units of flow leave B (see the edge (B, D)).



Example 90. Consider the following flow network. The flow configuration is **invalid**, as the edges (s, A) and (A, t) each have capacity 10, but there are 11 units of flow being pushed through each edge.



5.2 Flow Augmenting Paths

The goal of the max-flow problem is to move as much flow as possible from the source nodes to the sink nodes. We formalize this as follows.

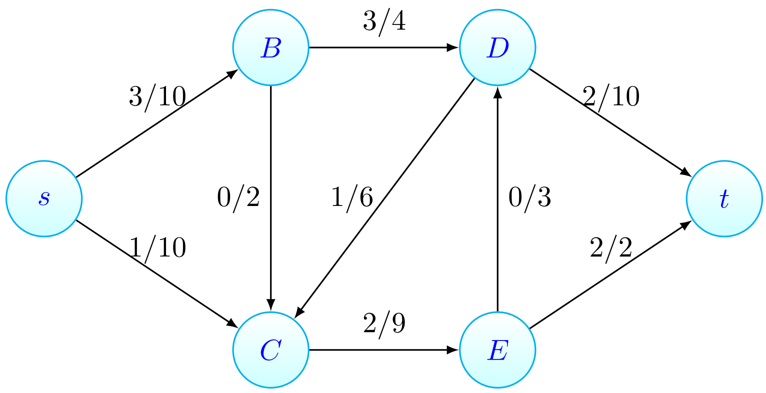
Definition 91. The Maximum Flow problem is defined as follows.

- Instance: We take as input a flow network $\mathcal{N}(G, c, S, T)$, with no flow along any directed edge.
- Solution: A flow function $f^* : V(G) \times V(G) \to [0, \infty)$ such that $|f^*|$ (the amount of flow leaving the source nodes) is maximized.

Algorithmically, our general approach is to find paths along which we can move positive flow from a source to a sink. We call such paths *flow-augmenting*. While we can always push flow in the forward direction along an edge, we can also re-route existing flow back along an edge. Precisely, let (u, v) be an edge, and let $f((u, v))$ be the current flow that is being pushed from $u \to v$. We may push at most $c((u, v)) - f((u, v))$ additional units of flow from $u \to v$. Atlernatively, we may push $f((u, v))$ units of flow back from $v \to u$. We then re-route this flow at v.

Example 92. Consider the following flow network. We examine certain directed edges to determine the amount of flow that can be pushed in the forward and backwards directions.

- Consider the edge (D, C). There is currently 1 unit of flow being pushed from $D \to C$, and (D, C) has a capacity of 6 units of flow. So we can push up to an additional 5 units of flow from $D \to C$. Alternatively, we can take up to the 1 unit of flow currently being routed from $D \to C$ and push that flow backwards from $C \to D$.
- Consider the edge (B, C). We may push 2 units of flow from $B \to C$. As there is no current flow being pushed from $B \to C$, we may **not** push flow backwards from $C \to B$.
- Consider the edge (B, D). There are currently 3 units of flow being pushed from $B \to D$. We may push at most 1 additional unit of flow from $B \to D$. Alternatively, we can take up to the 3 units of flow currently being routed from $D \to C$ and push that flow back from $D \to B$.



We now turn to examining flow-augmenting paths for the above flow network.

- We have that $s \to B \to D \to t$ is a flow-augmenting path. While we can push 7 units of flow from $s \to B$, we can only push 1 unit of flow from $B \to D$. Thus, we can only push 1 unit of flow along the path $s \to B \to D \to t$.
- We have that $s \to B \to C \to D \to t$ is a flow-augmenting path. While we can push 7 units of flow from $s \to B$, we can only push 2 units of flow from $B \to C$. Now as there is 1 unit of flow coming in from $C \to D$, we may push that 1 unit of flow backwards from $C \to D$. So only 1 of the 2 units of flow that make it to C end up at D. We then push that 1 unit of flow from $D \to t$. So we may push 1 unit of flow along the path $s \to B \to C \to D$.
- We have that $s \to C \to E \to D \to t$ is a flow-augmenting path. While we can push 9 units of flow from $s \to C$, we can only push back 7 units of flow from $C \to E$. Of those 7 units of flow, we may only push 3 units from $E \to D$, as (E, D) has a capacity of 3. Now we push 3 units of flow from $D \to t$. So along the flow augmenting path $s \to C \to E \to D \to t$, we may push at most 3 units of flow.

- We have that $s \to C \to D \to t$ is a flow-augmenting path. While we can push 9 units of flow from $s \to C$, we can only push 1 unit of flow from $C \to D$. We may then push this 1 unit of flow from $D \to t$. So we may push 1 unit of flow along the path $s \to C \to D \to t$.

5.3 Ford-Fulkerson Algorithm

In this section, we introduce the Ford-Fulkerson algorithm to solve the Maximum Flow problem. The key idea behind the Ford-Fulkerson procedure is to iteratively select a flow-augmenting path and then push as much flow as possible along said path.

Algorithm 8 Ford-Fulkerson

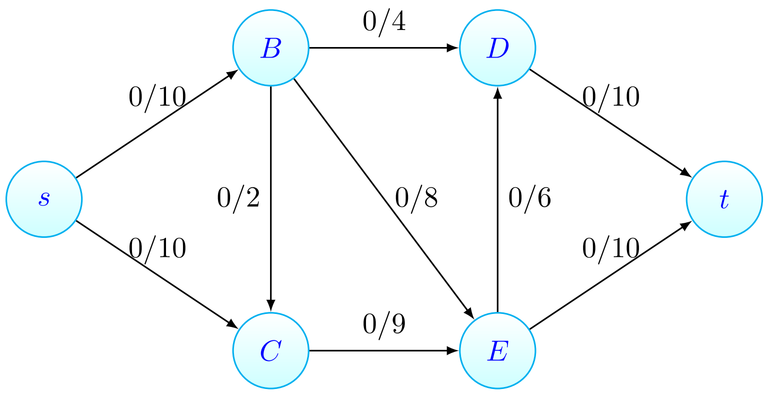
```
1: procedure FordFulkerson(FlowNetwork N)
2:     while N has a flow-augmenting path do
3:         Let P be a flow-augmenting path of N
4:         Push as much flow as possible along P
```

Remark 93. Algorithm 8 does not specify how to select flow-augmenting paths in an optimal manner. We will not delve into these details in this class; rather, we defer them for an Advanced Algorithms course. For the purposes of this class, it suffices to select your favorite flow-augmenting path at a given iteration.

We now work through some examples of the Ford-Fulkerson algorithm.

5.3.1 Ford-Fulkerson: Example 1

The example we consider in this section is based on the lecture slides of Wayne [Way05]. Consider the following flow network.



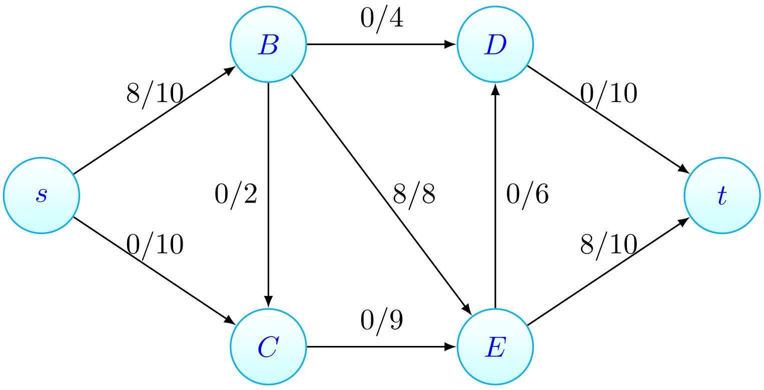
We proceed as follows.

1. There are several flow-augmenting paths we could select at the first iteration, including:

   - $s \to B \to D \to t$
   - $s \to B \to E \to t$
   - $s \to B \to E \to D \to t$
   - $s \to B \to C \to E \to t$
   - $s \to C \to E \to t$
   - $s \to C \to E \to D \to t$.

   We make the choice to select the flow-augmenting path $s \to B \to E \to t$, though we stress that any of the flow-augmenting paths could be selected at the initial round. Again, we made a choice to select $s \to B \to E \to t$, rather than following a prescribed rule.

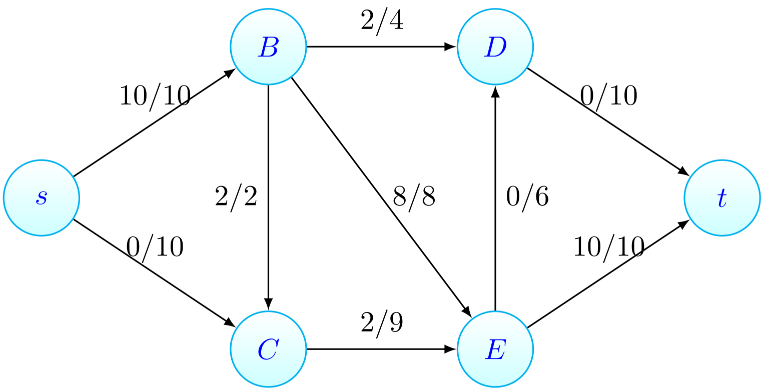
   We push the full 8 units of flow from $s \to B \to E \to t$. The updated flow network is below.



2. There are several flow-augmenting paths we could select at the second iteration, including:
   - $s \to B \to D \to t$
   - $s \to B \to C \to E \to t$
   - $s \to C \to E \to t$
   - $s \to C \to E \to D \to t$.

   We make the choice to select the flow-augmenting path $s \to B \to C \to E \to t$. Again, we stress that any of the above flow-augmented paths could be selected instead at this round. The selection of $s \to B \to C \to E \to t$ was arbitrary, rather than due to following a prescribed rule.

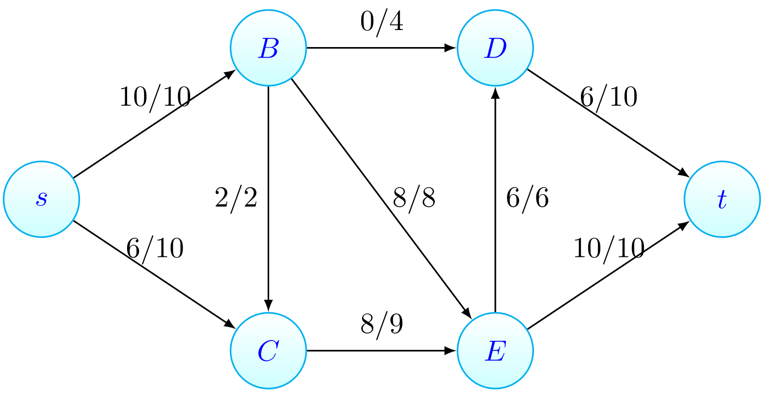
   We push the full 2 units of flow from $s \to B \to C \to E \to t$. The updated flow network is below.



3. There are several flow-augmenting paths we could select at the third iteration, including:
   - $s \to C \to E \to D \to t$
   - $s \to C \to E \to B \to D \to t$
   - $s \to C \to B \to D \to t$.

   We make the choice to select the flow-augmenting path $s \to C \to E \to D \to t$. Again, we stress that any of the above flow-augmented paths could be selected instead at this round. The selection of $s \to C \to E \to D \to t$ was arbitrary, rather than due to following a prescribed rule.

   We push the full 6 units of flow from $s \to C \to E \to D \to t$. The updated flow network is below.

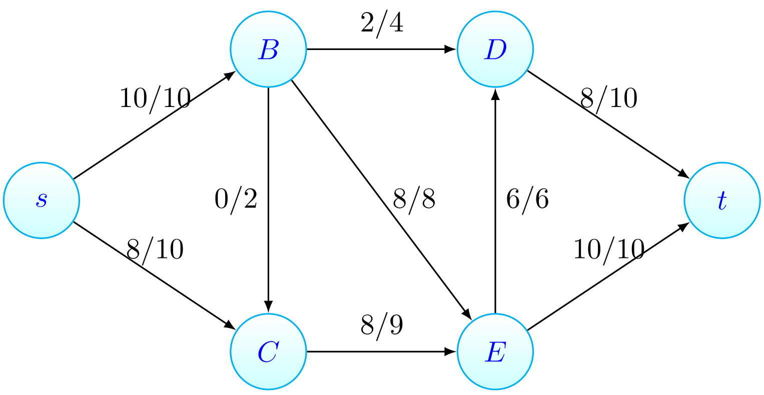


4. There are two flow-augmenting paths we could select at the fourth iteration, including:

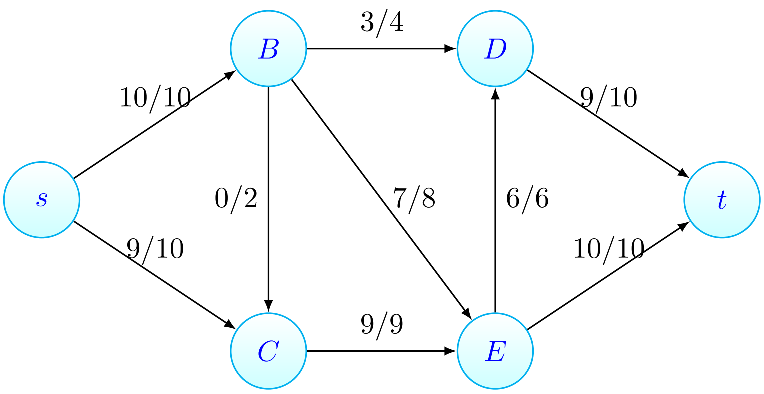
   - $s \to C \to B \to D \to t$
   - $s \to C \to E \to B \to D \to t$.

   We make the choice to select the flow-augmenting path $s \to C \to B \to D \to t$. Again, we stress that any of the above flow-augmented paths could be selected instead at this round. The selection of $s \to C \to B \to D \to t$ was arbitrary, rather than due to following a prescribed rule.

   We push the full 2 units of flow from $s \to C \to B \to D \to t$. The updated flow network is below.



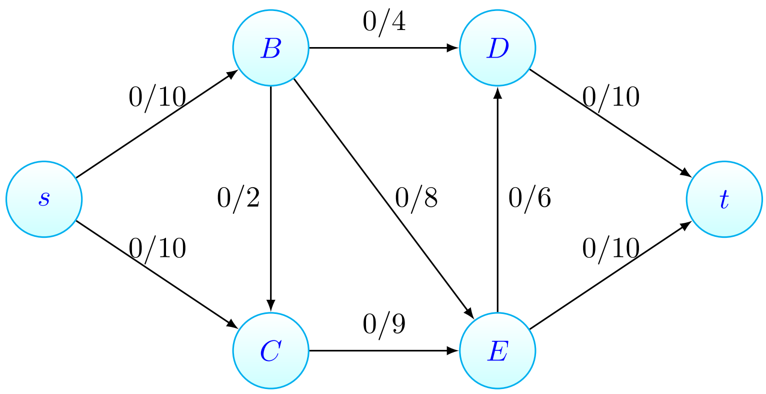
5. There is one flow-augmenting path at the fifth iteration: $s \to C \to E \to B \to D \to t$. We push the full 1 unit of flow across this path. The updated and final flow network is below.



   There are no more flow-augmenting paths, so the algorithm terminates. The total flow pushed from $s \to t$ is 19 units of flow.

5.3.2 Ford-Fulkerson: Example 2

We consider the same flow network as in the previous example. Here, we select alternative flow-augmenting paths to illustrate that there is choice in selecting said paths.



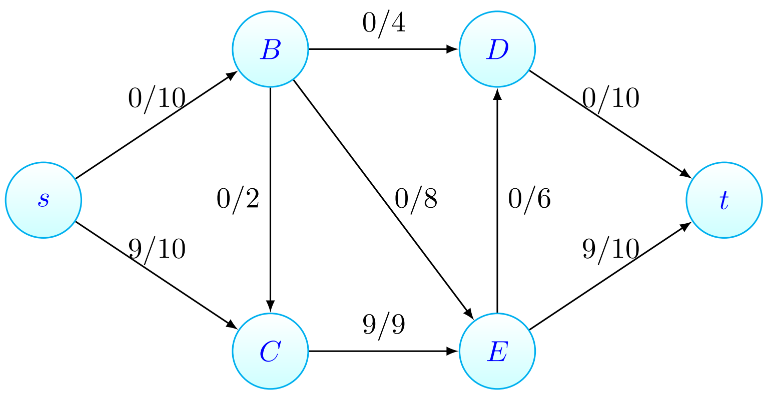
We proceed as follows.

1. There are several flow-augmenting paths we could select at the first iteration, including:

   - $s \to B \to D \to t$
   - $s \to B \to E \to t$
   - $s \to B \to E \to D \to t$
   - $s \to B \to C \to E \to t$
   - $s \to C \to E \to t$
   - $s \to C \to E \to D \to t$.

   We make the choice to select $s \to C \to E \to t$. Again, our choice is arbitrary, and not according to a prescribed rule.

   We push the full 9 units of flow from $s \to C \to E \to t$. The updated flow network is below.

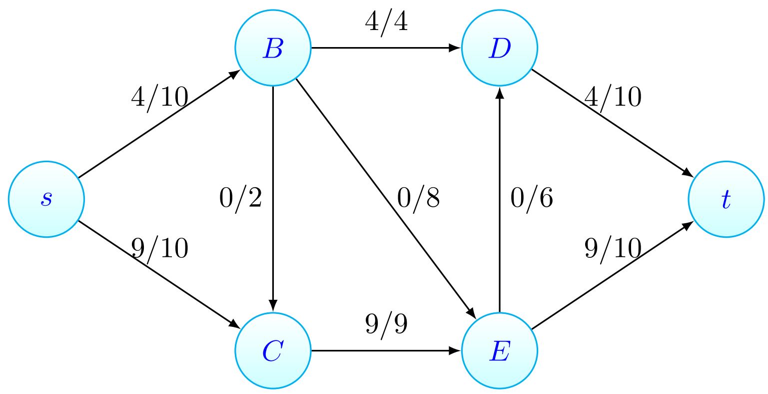


2. There are several flow-augmenting paths we could select at the second iteration, including:

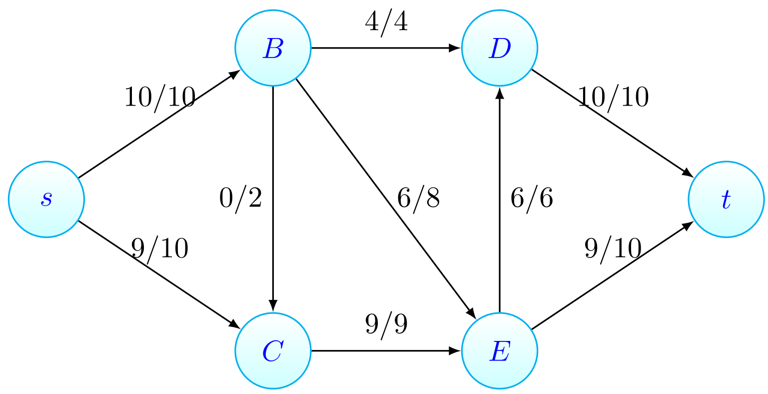
   - $s \to B \to D \to t$
   - $s \to B \to E \to D \to t$
   - $s \to B \to E \to t$.

   We make the choice to select $s \to B \to D \to t$. Again, our choice is arbitrary, and not according to a prescribed rule.

   We push the full 4 units of flow from $s \to B \to D \to t$. The updated flow network is below.



3. There is only one available flow-augmenting path at the third iteration: $s \to B \to E \to D \to t$. We push the full 6 units of flow along this path. The updated flow network is below.



There are no more flow-augmenting paths, so the algorithm terminates. Observe that 19 units of flow are being pushed from $s \to t$.

5.4 Minimum-Capacity Cuts

We may naturally think of the Maximum Flow problem as an attempt to maximize the number of resources that reach a collection of destinations. In this vein, there is a natural dual problem: what are the minimum number of disruptions needed to prevent *any* resources from reaching any of the destinations? The motivation for these problems arose during the Cold War. Here, the United States was interested in the Soviet Union Railway System that connected Eastern Europe- particularly, East Germany- and the western region of the Soviet Union. In particualr, the United States wanted to identify the minimum number of points to bomb in order to disrupt the flow of resources along this system [Ano, Sch02]. We will see later that the Maximum Flow problem is equivalent to finding the minimum number of disruptions. This is the celebrated Max-Flow Min-Cut Theorem.

We now turn to formalizing the Minimum Cut problem.

Definition 94. Let $\mathcal{N}(G, c, S, T)$ be a flow network. A *cut* of $\mathcal{N}$ is a partition of the vertices (X, Y), where $S \subseteq X$ (that is, X contains the source vertices), and $T \subseteq Y$ (that is, Y contains the sink vertices). Note that as (X, Y) is a partition, we have that X and Y are disjoint.

The *capacity* of the partition (X, Y) is the sum of the edge capacities with the initial endpoint in X and the destination vertex in Y. That is:

$$c(X, Y) := \sum_{x \in X} \sum_{y \in Y} c((x, y)).$$

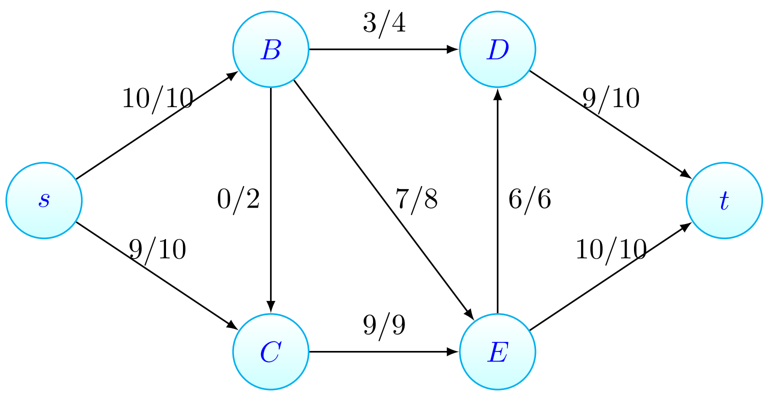
Recall that if (x, y) is not an edge of the flow network, then $c((x, y)) = 0$.

Definition 95. The Minimum Cut problem is defined as follows.

- Instance: Let $\mathcal{N}(G, c, S, T)$ be a flow network.
- Solution: A cut (X, Y) such that $c(X, Y)$ is minimized.

We may readily compute a minimum-capacity cut from a maximum flow f^* in the following manner. The vertices of X are precisely the source vertices, together with the vertices to which we can push positive flow from a source. The vertices of Y are the remaining vertices; that is, $Y := V(G) \setminus X$.

Example 96. Recall the maximum flow f^* from Example 5.3.1, pictured below.



We compute the minimum-capacity cut corresponding to f^*. We begin by computing X.

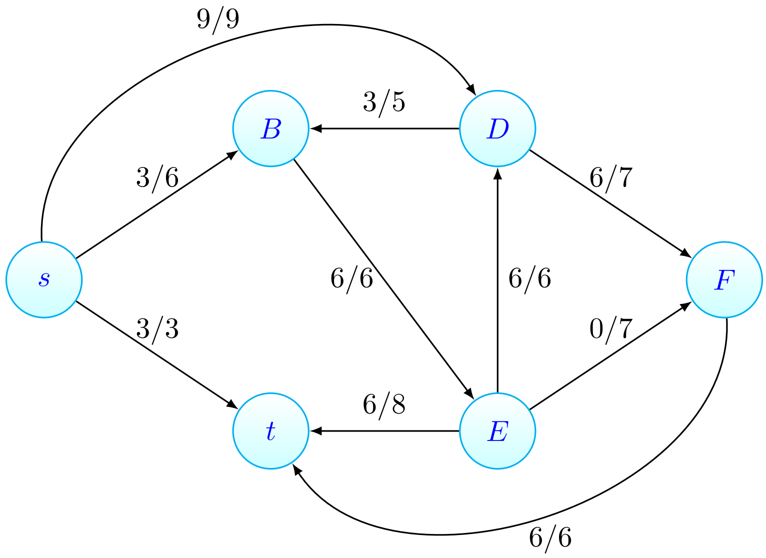
- The source vertex s belongs to X.
- We may push 1 unit of flow from $s \to C$. So C belongs to X.

There are no other vertices to which we can push positive flow from s. So $X = \{s, C\}$ and $Y = \{B, D, E, t\}$. Now the capacity of the cut $c(X, Y)$ is the sum of the capacities of the edges (x, y), where $x \in X$ and $y \in Y$. The only such edges are (s, B) and (C, E). So:

$$\begin{aligned} c(X, Y) &= c((s, B)) + c((C, E)) \\ &= 10 + 9 \\ &= 19. \end{aligned}$$

Observe that the capacity of the cut (X, Y) is precisely the value $|f^*| = 19$, the amount of flow we can push from $s \to t$. This is no coincidence; in fact, this is the precise statement of the Max-Flow Min-Cut Theorem: the capacity of a minimum-capacity cut is the same as the value of a maximum flow. We will prove the Max-Flow Min-Cut Theorem later.

Example 97. Consider the maximum flow f^* for the following flow network, below.



We compute the minimum-capacity cut corresponding to f^*. We begin by computing X.

- The source vertex s is in X.
- We may push up to 5 units of flow from $s \to B$. So B is in X.
- We may push up to 3 units of flow from $s \to D$ along the path $s \to B \to D$. So D belongs to S.
- We may push up to 1 unit of flow from $s \to F$ along the path $s \to B \to D \to F$. So F belongs to X.

There are no other vertices to which we can push positive flow from S. So $X = \{s, B, D, F\}$ and $Y = \{E, t\}$. Now the capacity of the cut $c(X, Y)$ is the sum of the capacities of the edges (x, y), where $x \in X$ and $y \in Y$. The edges crossing the cut are (s, t), (F, t), and (B, E). So:

$$\begin{aligned} c(X, Y) &= c((s, t)) + c((F, t)) + c((B, E)) \\ &= 3 + 6 + 6 \\ &= 15. \end{aligned}$$

Observe that the capacity of the cut (X, Y) is precisely the value $|f^*| = 15$, the amount of flow we can push from $s \to t$.

A Notation

A.1 Collections

Definition 98. A *set* S is a collection of distinct, unordered elements. That means that S does not have any repeated elements, nor does the order in which the elements are listed matter. We denote sets using **curly braces** (and **not** square brackets or parentheses).

Example 99. The collection $S = \{1, 2, 3\}$ is a set. Note that $\{1, 2, 3\} = \{2, 1, 3\} = \{3, 2, 1\}$ are all the same set, as the order in which elements are listed does not matter. Notice as well that we wrote $S = \{1, 2, 3\}$, and **not** $S = [1, 2, 3]$ or $S = (1, 2, 3)$.

We recall several families of sets:

- The natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, which are the non-negative integers. Note that $0 \in \mathbb{N}$.
- The integers $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.
- The positive integers $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$.

Definition 100. Let S be a set. If an element x belongs to S, we denote it as $x \in S$. If x does not belong to x, we denote this as $x \notin S$.

Example 101. Let $S = \{1, 2, 3\}$.

- As 3 is in S, we may write $3 \in S$.
- As 4 is not in S, we may write $4 \notin S$.

A.2 Series

Definition 102. Let $a_1, \ldots, a_k$ be numbers. The *summation series* is defined as:

$$\sum_{i=1}^{k} a_i := a_1 + a_2 \cdots + a_k.$$

Similarly, the *product series* is defined as:

$$\prod_{i=1}^{k} a_i := a_1 \cdot a_2 \cdots a_k.$$

Example 103. We have that:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n,$$

while:

$$\prod_{i=1}^{n} = 1 \cdot 2 \cdot 3 \cdots n = n!$$

B Graph Theory

To quote Bud Brown, "Graph theory is a subject whose deceptive simplicity masks its vast applicability." Graph theory provides simple mathematical structures known as graphs to model the relations of various objects. The applications are numerous, including efficient storage of chemicals (graph coloring), optimal assignments (matchings), distribution networks (flows), efficient storage of data (tree-based data structures), and machine learning. In automata theory, we use directed graphs to provide a visual representation of our machines. Many elementary notions from graph theory, such as path-finding and walks, come up as a result. In complexity theory, many combinatorial optimization problems of interest are graph theoretic in nature. Therefore, it is important to discuss basic notions from graph theory. We begin with the basic definition of a graph.

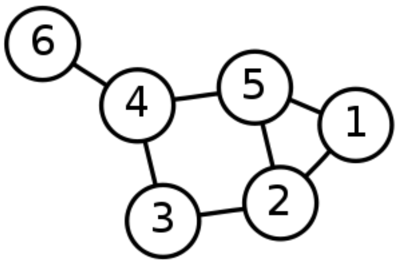
The exposition in this section has benefitted from [Die10, Lev20, Wes00].

B.1 Introduction to Graphs

Definition 104 (Simple Graph)**.** A simple graph is a two-tuple $G(V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$.

By convention, a simple graph is referred to as a *graph*, and an edge $\{i, j\}$ is written as ij. In simple graphs, $ij = ji$. Two vertices i, j are said to be *adjacent* if $ij \in E(G)$. Now let's consider an example of a graph.

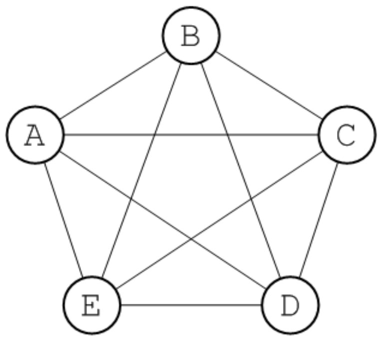
Example 105. Let $G(V, E)$ be the graph where $V = \{1, 2, \dots, 6\}$ and $E = \{12, 15, 23, 25, 34, 45, 46\}$. This graph is pictured below.



We now introduce several common classes of graphs.

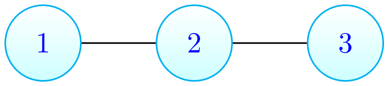
Definition 106 (Complete Graph)**.** The complete graph, denoted K_n, has the vertex set $V = \{1, 2, \dots, n\}$ and edge set E which consists of **all** two-element subsets of V. That is, K_n has all possible edges between vertices.

Example 107. The complete graph on five vertices K_5 is pictured below.



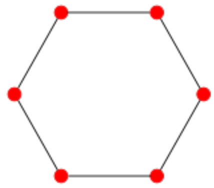
Definition 108 (Path Graph)**.** The path graph, dentoed P_n, has vertex set $V = \{1, 2, \dots, n\}$ and the edge set $E = \{\{i, i + 1\} : 1 \leq i \leq n - 1\}$.

Example 109. The path on three vertices P_3 is shown below.



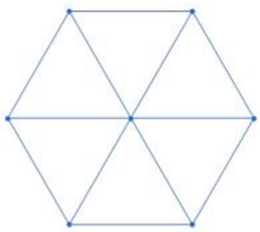
Definition 110 (Cycle Graph)**.** Let $n \geq 3$. The cycle graph, denoted C_n, has the vertex set $V = \{1, 2, \dots, n\}$ and the edge set $E = \{\{i, i+1\} : 1 \leq i \leq n-1\} \cup \{\{1, n\}\}$.

Example 111. Intuitively, C_n can be thought of as the regular n-gon. So C_3 is a triangle, C_4 is a quadrilateral, and C_5 is a pentagon. The graph C_6 is pictured below.



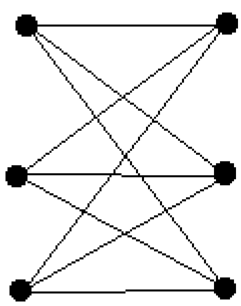
Definition 112 (Wheel Graph)**.** Let $n \geq 4$. The wheel graph, denoted W_n, is constructed by joining a vertex n to each vertex of C_{n-1}. So we take $C_{n-1} \dot{\cup} n$ and add the edges vn for each $v \in [n-1]$.

Example 113. The wheel graph on seven vertices W_7 is pictured below.



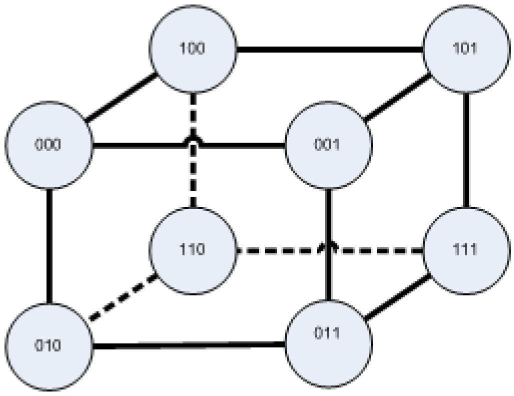
Definition 114 (Bipartite Graph)**.** A bipartite graph $G(V, E)$ has a vertex set $V = X \dot{\cup} Y$, with edge set $E \subseteq \{xy : x \in X, y \in Y\}$. That is, no two vertices in the same part of V are adjacent. So no two vertices in X are adjacent, and no two vertices in Y are adjacent.

Example 115. A common class of bipartite graphs include even-cycles C_{2n}. The complete bipartite graph is another common example. We denote the complete bipartite graph as $K_{m,n}$ which has vertex partitions $X \dot{\cup} Y$ where $|X| = m$ and $|Y| = n$. The edge set $E(K_{m,n}) = \{xy : x \in X, y \in Y\}$. The graph $K_{3,3}$ is pictured below.



Definition 116 (Hypercube)**.** The hypercube, denoted Q_n, has vertex set $V = \{0, 1\}^n$. Two vertices are adjacent if the binary strings differ in precisely one component.

Example 117. The hypercube Q_2 is isomorphic to C_4 (isomorphism roughly means that two graphs are the same, which we will formally define later). The hypercube Q_3 is pictured below.



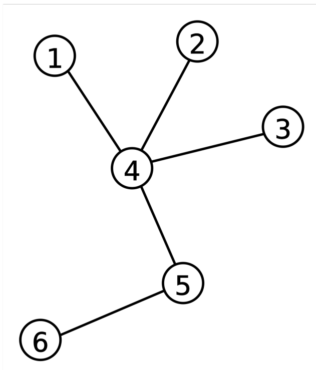
Definition 118 (Connected Graph)**.** A graph $G(V, E)$ is said to be connected if for every $u, v \in V(G)$, there exists a $u-v$ path in G. A graph is said to be *disconnected* if it is not connected; and each connected subgraph is known as a *component*.

Example 119. So far, every graph presented has been connected. If we take two disjoint copies of any of the above graphs, their union forms a disconnected graph.

Definition 120 (Tree)**.** A Tree is a connected, acyclic graph.

Example 121. A path is an example of a tree. Additional examples include the binary search tree, the binary heap, and spanning trees of graphs.

Example 122. The following is an example of a tree.



Definition 123 (Degree)**.** Let $G(V, E)$ be a graph and let $v \in V(G)$. The degree of v, dentoed $\deg(v)$ is the number of edges containing v. That is, $\deg(v) = |\{vx : vx \in E(G)\}|$.

Example 124. Each vertex in the Cycle graph C_n has degree 2. In Example 105, $\deg(6) = 1$ and $\deg(5) = 3$.

Theorem 125 (Handshake Lemma)**.** Let $G(V, E)$ be a graph. We have:

$$\sum_{v \in V(G)} \deg(v) = 2|E(G)|.$$

Proof. The proof is by double counting. The term $\deg(v)$ counts the number of edges incident to v. Each edge has two endpoints v and x, for some other $x \in V(G)$. So the edge vx is double counted in both $\deg(v)$ and $\deg(x)$. Thus,

$$\sum_{v \in V(G)} \deg(v) = 2|E(G)|.$$

□

Remark: The Handshake Lemma is a *necessary condition* for a graph to exist. That is, all graphs satisfy the Handshake Lemma. Consider the following: does there exist a graph on 11 vertices each having degree

5? By the Handshake Lemma, $11 \cdot 5 = 2|E(G)|$. However, 55 is not even, so no such graph exists. Note that the Handshake Lemma is not a *sufficient condition*. That is, there exist degree sequences such as $(3, 3, 1, 1)$ satisfying the Handshake Lemma which are not realizable by any graph. Theorems such as Havel-Hakimi and Erdős-Gallai provide conditions that are both sufficient and necessary for a degree sequence to be realizable by some graph.

Next, the notion of a walk will be introduced.

Definition 126 (Walk)**.** Let $G(V, E)$ be a graph. A walk of length n is a sequence $(v_i)_{i=0}^{n}$ such that $v_i v_{i+1} \in E(G)$ for all $i \in \{0, \ldots, n-1\}$. If $v_0 = v_n$, the walk is said to be *closed.*

Let us develop some intuition for a walk. We start a given vertex v_0. Then we visit one of v_0's neighbors, which we call v_1. Next, we visit one of v_1's neighbors, which we call v_2. We continue this construction for the desired length of the walk. The key difference between a walk and a path is that a walk can repeat vertices, while all vertices in a path are distinct.

Example 127. Consider a walk on the hypercube Q_3. The sequence of vertices $(000, 100, 110, 111, 101)$ forms a walk, while $(000, 100, 110, 111, 101, 001, 000)$ is a closed walk. The sequence $(000, 111)$ is not a walk because 000 and 111 are not adjacent in Q_3.

We now define the adjacency matrix, which is useful for enumerating walks of a given length.

Definition 128 (Adjacency Matrix)**.** Let $G(V, E)$ be a graph. The adjacency matrix A is an $n \times n$ matrix where:

$$A_{ij} = \begin{cases} 1 & : ij \in E(G) \\ 0 & : ij \not\in E(G) \end{cases} \tag{16}$$

Example 129. Consider the adjacency matrix for the graph K_5:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{17}$$

Theorem 130. Let $G(V, E)$ be a graph, and let A be its adjacency matrix. For each $n \in \mathbb{Z}^+$, A^n_{ij} counts the number of walks of length n starting at vertex i and ending at vertex j.

Proof. The proof is by induction on n. When $n = 1$, we have A. By definition $A_{ij} = 1$ iff $ij \in E(G)$. All walks of length 1 correspond to the edges incident to i, so the theorem holds true when $n = 1$. Now fix $k \geq 1$ and suppose that for each $m \in [k]$ that A^m_{ij} counts the number of $i - j$ walks of length m. The $k+1$ case will now be shown.

Consider $A^{k+1} = A^k \cdot A$ by associativity. By the inductive hypothesis, A^k_{ij} and A_{ij} count the number of $i - j$ walks of length k and 1 respectively. Observe that:

$$A^{k+1}_{ij} = \sum_{x=1}^{n} A^k_{ix} A_{xj}$$

So A^k_{ix} counts the number of ix walks of length k, and $A_{xj} = 1$ iff $xj \in E(G)$. Adding the edge xj to an $i - x$ walk of length k forms an $i - j$ walk of length $k + 1$. The result follows by induction. $\square$

We will prove one more theorem before concluding with the graph theory section. In order to prove this theorem, the following lemma (or helper theorem) is needed.

Lemma 131. Let $G(V, E)$ be a graph. Every closed walk of odd length at least 3 in G contains an odd-cycle.

Proof. The proof is by induction on the length of the walk. Note that a closed walk of length 3 forms a K_3. Now fix $k \geq 1$ and suppose the that any closed walk of odd length up to $2k+1$ has an odd-cycle. We prove true for walks of length $2k+3$. Let $(v_i)_{i=0}^{2k+3}$ be a walk closed of odd length. If $v_0 = v_{2k+3}$ are the only repeated vertices, then the walk itself is an odd cycle and we are done. Otherwise, suppose $v_i = v_j$ for some $0 \leq i < j \leq 2k+3$. If the walk $(v_t)_{t=i}^{k}$ is odd, then there exists an odd cycle by the inductive hypothesis. Otherwise, the walk $W = (v_0, \ldots, v_i, v_{j+1}, \ldots, v_{2k+3})$ is of odd length at most $2k+1$. So by the inductive hypothesis, W has an odd cycle. So the lemma holds by induction. □

We now characterize bipartite graphs.

Theorem 132. A graph $G(V, E)$ is bipartite if and only if it contains no cycles of odd length.

Proof. Suppose first that G is bipartite with parts X and Y. Now consider a walk of length n. As no vertices in a fixed part are adjacent, only walks of even lengths can end back in the same part as the staring vertex. A cycle is a walk where all vertices are distinct, save for v_0 and v_n which are the same. Therefore, no cycle of odd length exists in G.

Conversely, suppose G has no cycles of odd length. We construct a bipartition of $V(G)$. Without loss of generality, suppose G is connected. For if G is not connected, we apply the same construction to each connected component. Fix the vertex v. Let $X = \{u \in V(G) : d(u, v) \text{ is even }\}$, where $d(u, v)$ denotes the distance or length of the shortest uv path. Let $Y = \{u \in V(G) : d(u, v) \text{ is odd }\}$. Clearly, $X \cap Y = \emptyset$. So it suffices to show no vertices within X are adjacent, and no vertices within Y are adjacent. Fix $v \in X$ and suppose to the contrary that two vertices in $y_1, y_2 \in Y$ are adjacent. Then there exists a closed walk of odd length $(v, \ldots, y_1, y_2, \ldots v)$. By Lemma 131, G must contain an odd-cycle, a contradiction. By similar argument, no vertices in X can be adjacent. So G is bipartite with bipartition $X \dot{\cup} Y$. □

References

[Ano] Anonymous, *The origin of the study of network flow.*

[CD20] Charles Carlson and Ewan Davies, *Csci 3104: Introduction to algorithms*, 2020.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms, third edition*, 3rd ed., The MIT Press, 2009.

[Die10] Reinhard Diestel, *Graph theory*, 2010.

[Err] Jeff Errickson, *Algorithms homepage*, Available at `https://jeffe.cs.illinois.edu/teaching/algorithms/`.

[Fie15] Joe Fields, *A gentle introduction to the art of mathematics*, Joe Fields, 2015, https://raw.githubusercontent.com/osj1961/giam/master/GIAM.pdf.

[Ham20] Richard Hammack, *Book of proof*, Richard Hammack, 07 2020, https://www.people.vcu.edu/ rhammack/BookOfProof/.

[KT05] Jon Kleinberg and Eva Tardos, *Algorithm design*, Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[Lev20] Michael Levet, *Theory of computation- lecture notes.*

[Mou16] Lalla Mouatadid, *Greedy algorithms: Interval scheduling*, 2016, http://www.cs.toronto.edu/ lalla/373s16/notes/ISP.pdf.

[Mou17] Dave Mount, *Cmsc 451: Lecture 7 greedy algorithms for scheduling*, https://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect07-greedy-sched.pdf.

[Sch02] Alexander Schrijver, *On the history of the transportation and maximum flow problems*, Mathematical Programming (2002), 437–445.

[Tea21] OpenDSA Team, *All current opendsa content (canvas version)*, 2021.

[Way05] Kevin Wayne, 2005.

[Wes00] Douglas B. West, *Introduction to graph theory*, 2 ed., Prentice Hall, September 2000.