CSCI 3104 Algorithms- Lecture Notes

Michael Levet

October 4, 2023

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. □

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. □

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 $f_{k-1} \leq 2^{k-1}$.

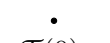
The result follows by induction. □

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.

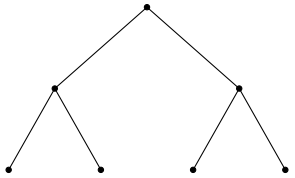
- $\mathcal{T}(0)$ is a single vertex.
- For $d > 0$, $\mathcal{T}(d)$ is obtained by starting with a single vertex and setting both of its children to be copies of $\mathcal{T}(d-1)$.

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.

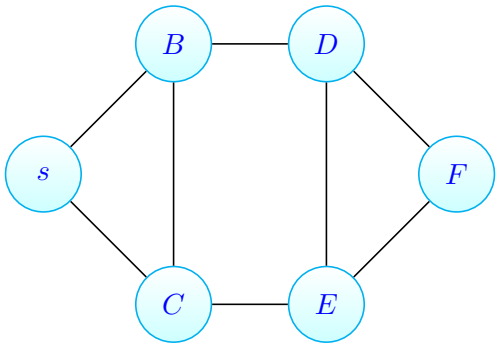
2.1 Depth-First Traversal

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

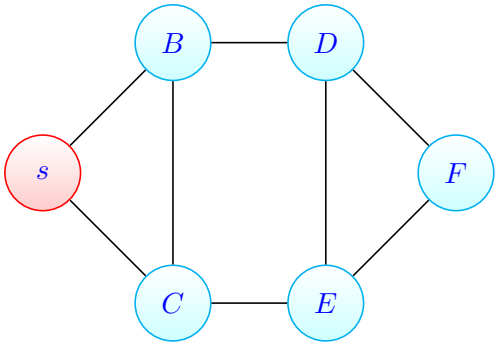
```
1: procedure DFS(Graph G, Vertex s)
2:     s.visited ← true
3:     for each unvisited neighbor v of s do
4:         DFS(G, v)
```

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

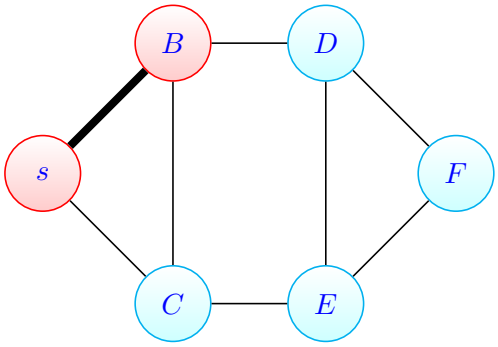


Suppose we invoke `DFS`(G, s). For the purpose of this example, we examine the unvisited neighbors of the current vertex. Note that one could alternatively choose to order the vertices in a different way, which would result in visiting the vertices in a different order. We also use red nodes to denote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

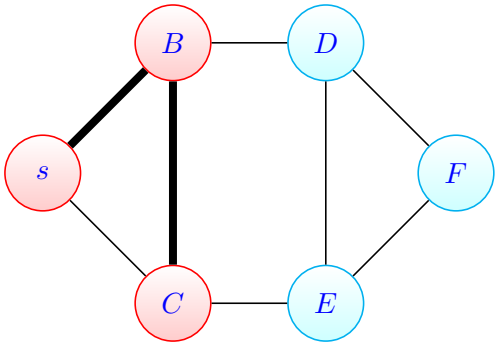
1. We first set s.visited := `true`.



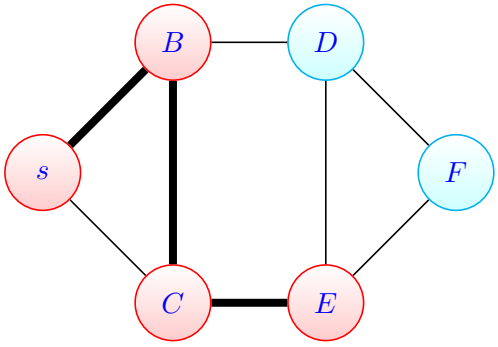
2. Both neighbors of s, B and C, are unvisited. So we recurse and visit B next, invoking $\texttt{DFS}(G, B)$.



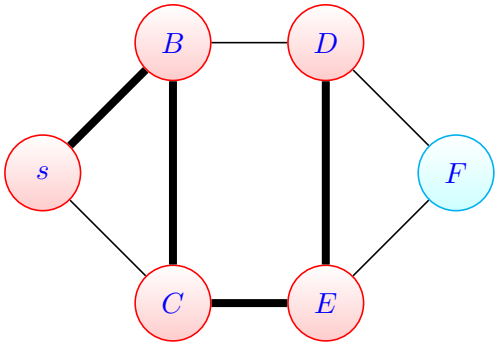
3. Now the two unvisited neighbors of B are C and D. We visit C next, as we are choosing to order the neighbors alphabetically. We invoke $\texttt{DFS}(G, C)$.



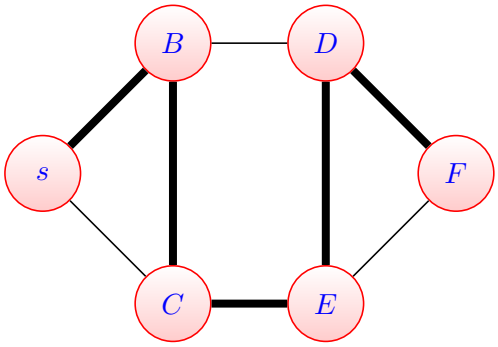
4. The only unvisited neighbor of C is E, so we next invoke $\texttt{DFS}(G, E)$.



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



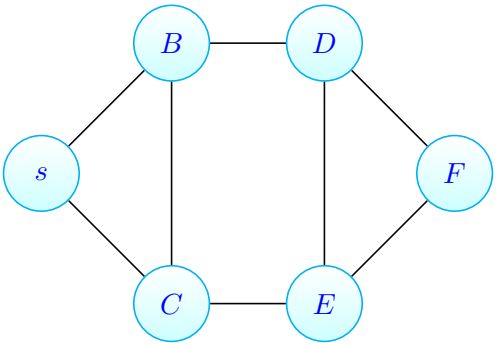
6. The only unvisited neighbor of D is F. So we invoke $\texttt{BFS}(G, D)$.



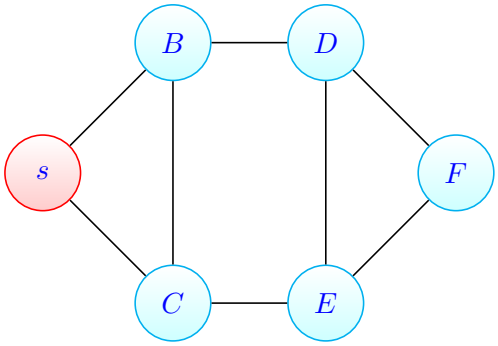
7. Now F has no unvisited neighbors. So $\texttt{DFS}(G, F)$ returns control to the invoking call $\texttt{DFS}(G, D)$.
8. Similarly, as D has no unvisited neighbors, $\texttt{DFS}(G, D)$ returns control to the invoking call $\texttt{DFS}(G, E)$.
9. Now E has no unvisited neighbors. So $\texttt{DFS}(G, E)$ 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.

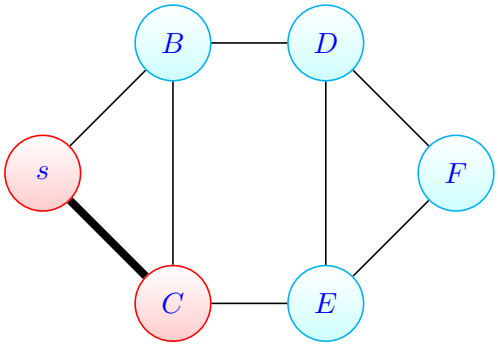
Example 12. In Example 10, we considered the result of calling $\texttt{DFS}(G, s)$ on the following graph G, where the unvisited neighbors were examined in alphabetical order. We choose to select neighbors arbitrarily; that is, not according to a prescribed rule. This yields a different order in which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



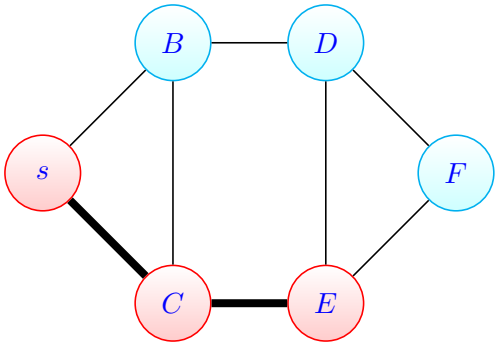
1. As before, we start by setting $s.\text{visited} := \texttt{true}$.



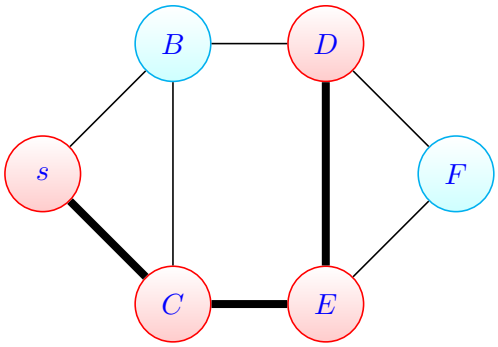
2. The unvisited neighbors of s are B and C. We choose to visit C next. We invoke $\texttt{DFS}(G, C)$.



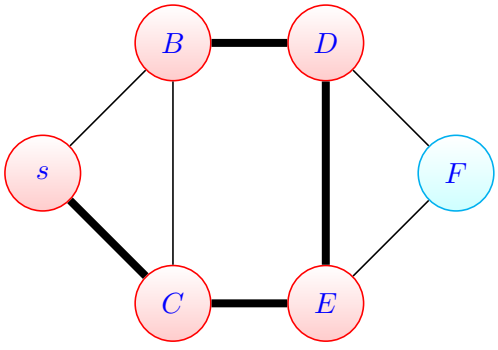
3. The unvisited neighbors of C are B and E. We choose to visit E next, invoking $\texttt{DFS}(G, E)$.



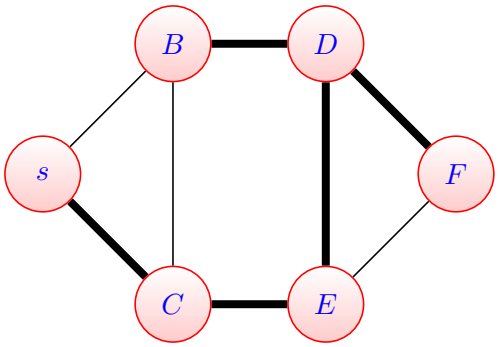
4. The unvisited neighbors of E are D and F. We choose to visit D next, invoking $\texttt{DFS}(G, D)$.



5. The unvisited neighbors of D are B and F. We choose to visit B next, invoking $\texttt{DFS}(G, B)$.



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



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

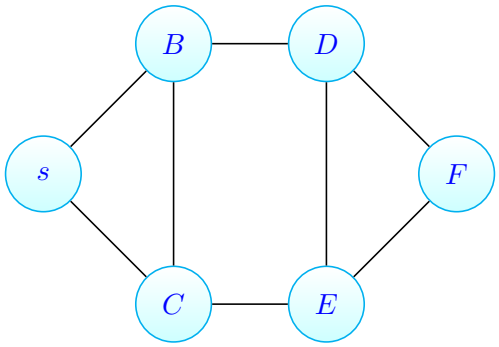
2.2 Breadth-First Traversal

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

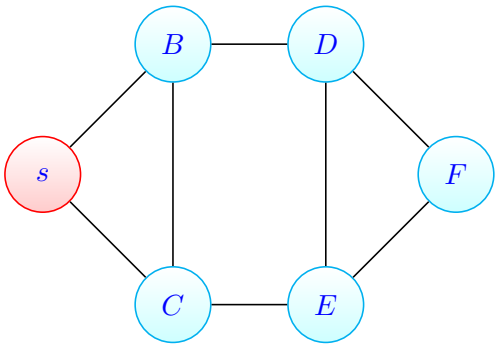
```
1: procedure BFS(Graph G, Vertex s)
2:     s.visited ← true
3:     Queue Q ← [s]
4:     while Q ≠ [] do
5:         current ← Q.poll()
6:         for each unvisited neighbor v of current do
7:             v.visited ← true
8:             Q.push(v)
```

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.

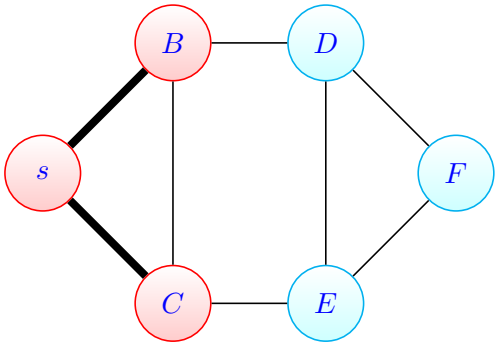


We invoke `BFS`(G, s).

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

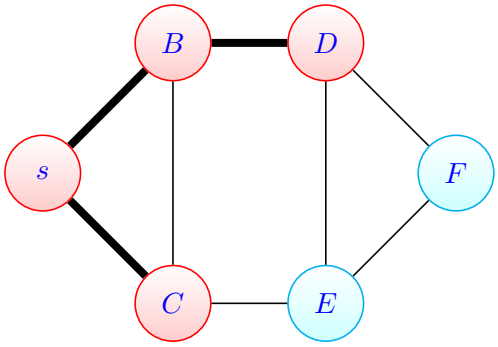


2. We set current := Q.poll(). So current = s.
3. We now set the neighbors of s as visited and then push them into Q. So B.visited := `true`, C.visited := `true`, and $Q = [B, C]$.



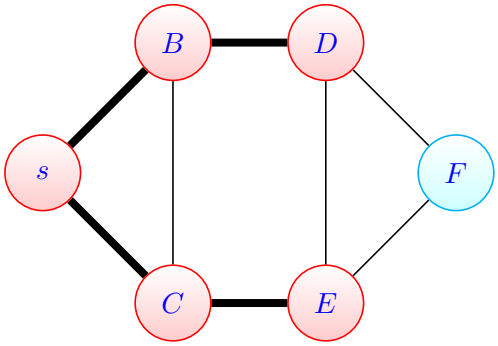
4. We now set current := Q.poll(). So current = B.

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



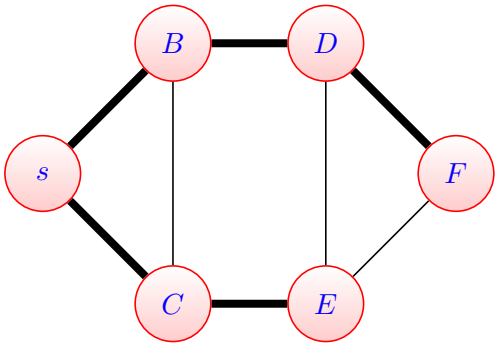
6. We now set current := Q.poll(). So current = C.

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 $Q = [F]$.

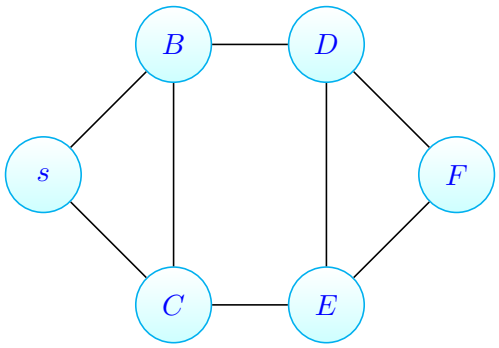
11. We now set current := Q.poll(). So current = F. Now F has no unvisited neighbors, so we do not add any vertices to the queue. Now $Q = []$. As $Q = []$, the algorithm terminates.

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.

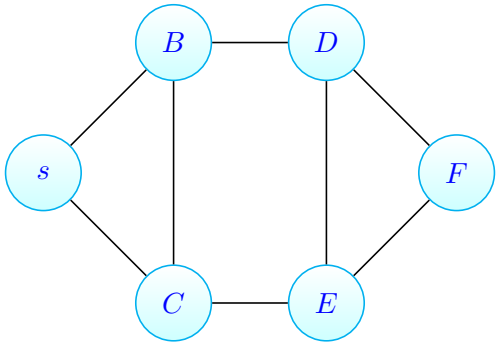
Definition 16 (Unweighted Shortest Path Problem)**.** The Unweighted Shortest Path problem is defined as follows.

- 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.

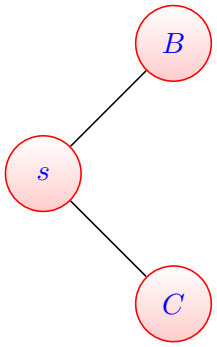
Example 17. Recall Example 13, where we added the unvisited neighbors to the queue in alphabetical order. We again use the same graph G, below.



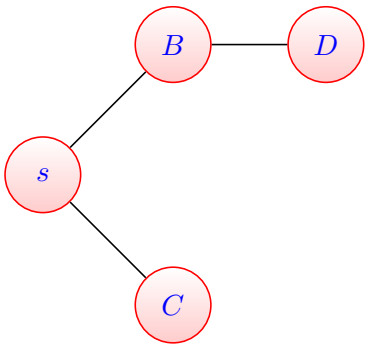
We seek to find the shortest s to v path, for every vertex $v \in V(G)$.

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

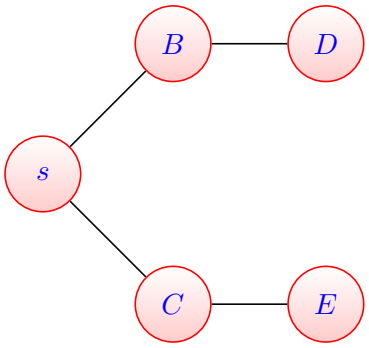
2. We next poll s from Q. We now mark both neighbors of s, B and C, as visited and place them into the queue. So B.visited = `true`, C.visited = `true`, and $Q = [B, C]$. We now add the edges $\{s, B\}$ and $\{s, C\}$ to the tree. So our intermediary tree is as follows.



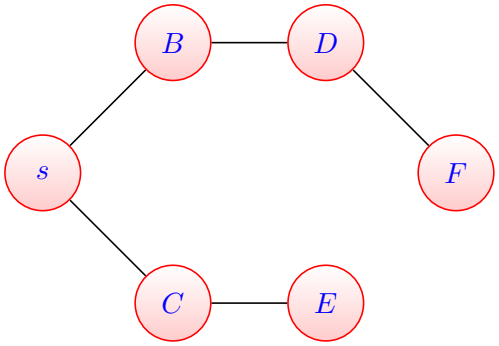
3. We next poll B from Q. We now mark the unvisited neighbor of B, which is D, as visited and place D into the queue. So D.visited = `true`, and $Q = [C, D]$. We now add the edge $\{B, D\}$ to the tree. So our intermediary tree is as follows.



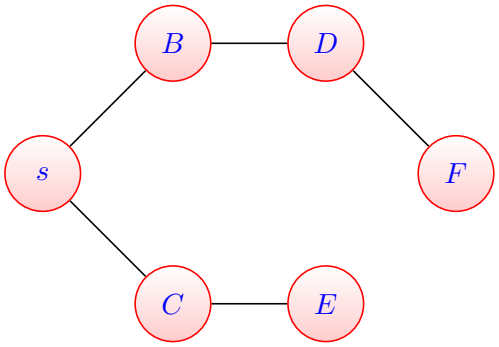
4. We next poll C from Q. We now mark the unvisited neighbor of C, which is E, as visited and place E into the queue. So E.visited = `true`, and $Q = [D, E]$. We now add the edge $\{C, E\}$ to the tree. So our intermediary tree is as follows.



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.



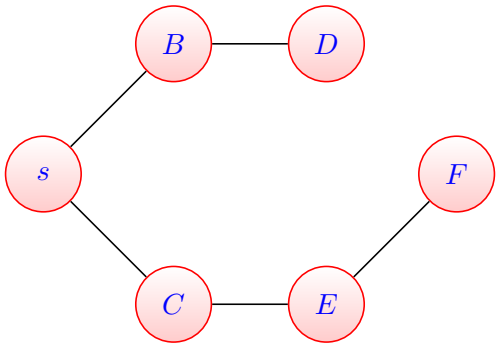
6. Note that the remaining elements of $Q = [E, F]$ have no unvisited neighbors. So we poll E, then poll F. Now as $Q = []$, the algorithm terminates. Our SSPT is:



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.

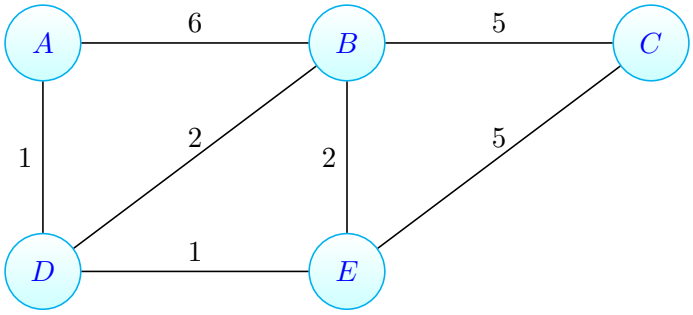
2.4 Shortest-Path Problem: Weighted Graphs and Dijkstra's Algorithm

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 $y.\text{predecessor} := x$. 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

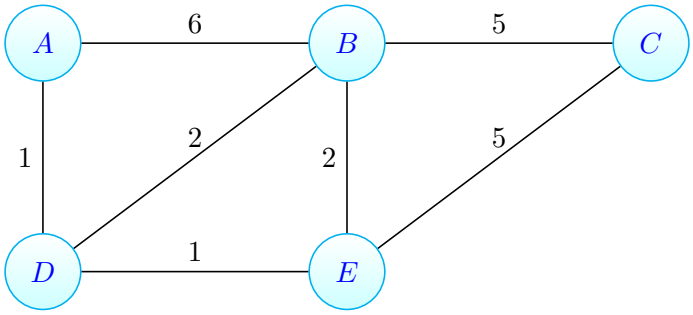
Require: The graph G is connected and has no negative-weight edges.

```
1:  procedure DIJKSTRA(WeightedGraph G(V, E, w), Vertex s)
2:      PriorityQueue Q ← []
3:      for each v ∈ V(G) do
4:          v.dist ← ∞
5:          v.predecessor = NULL
6:          Q.push(v)
7:      s.dist ← 0
8:      Q.updatePosition(s)

9:      while Q ≠ [] do
10:         current ← Q.poll()
11:         current.processed = true

12:         for each unprocessed neighbor y of current do

13:             if current.dist + w({current, y}) < y.dist then
14:                 y.dist ← current.dist + w({current, y})
15:                 y.predecessor ← current
16:                 Q.updatePosition(y)
        return G
```

2.4.1 Dijkstra's Algorithm: Example 1

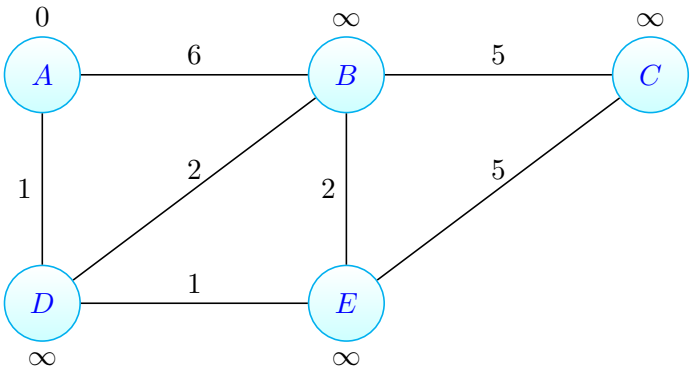
We now consider an example.

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:

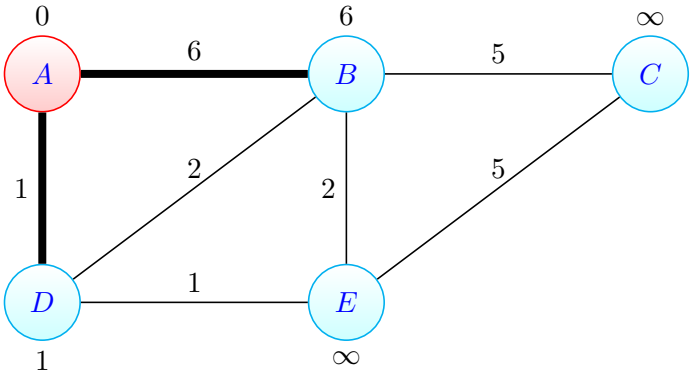
$$B.\text{dist} = 6,$$
$$B.\text{predecessor} = A,$$

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

$$D.\text{dist} = 1,$$
$$D.\text{predecessor} = A,$$

  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:

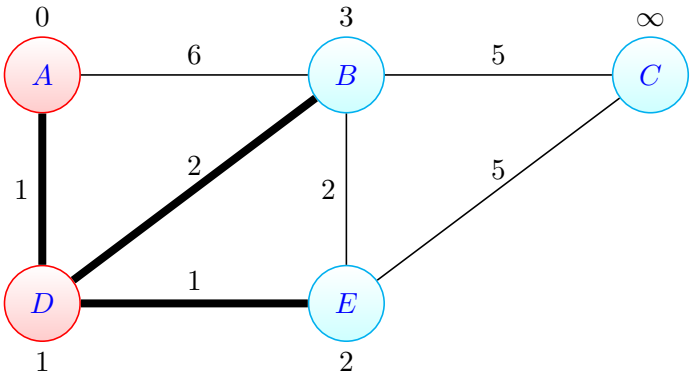
$$B.\text{dist} = 3, \text{ and}$$
$$B.\text{predecessor} = D.$$

     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:

$$E.\text{dist} = 2,$$
$$E.\text{predecessor} = D,$$

     and then push E into the priority queue.

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



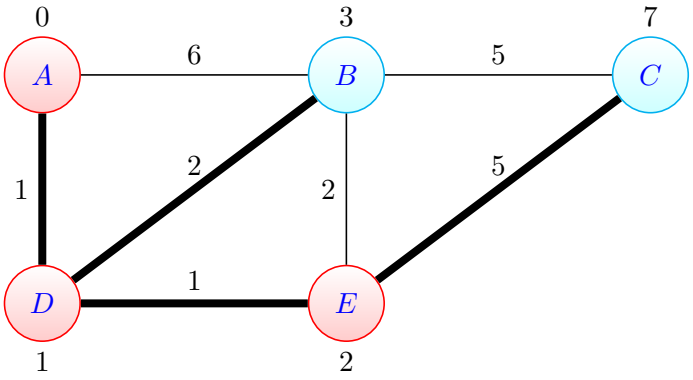
4. We now poll E from the priority queue and mark E as processed. The two unprocessed neighbors of E are B and C. Observe that:

   - $\text{dist}(A, E) + w(\{E, B\}) = 2 + 2 \not< 3$. 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

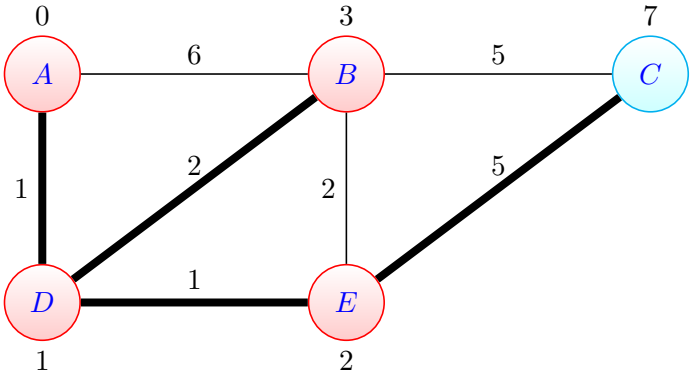
   So the priority queue is $Q = [(B, 3), (C, 7)]$. The updated graph is below.



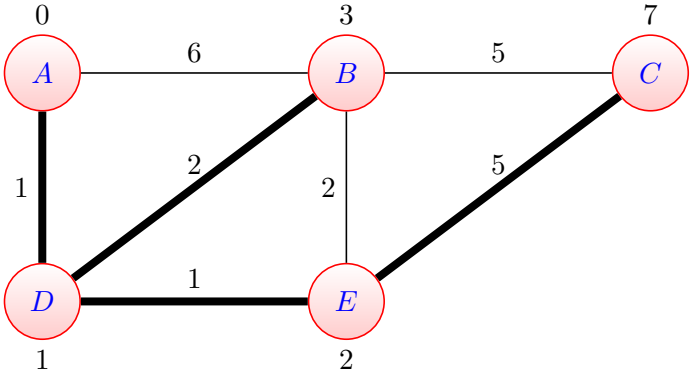
5. We poll B from the priority queue and mark B as processed. The only unprocessed neighbor of B is C. As:

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

   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.

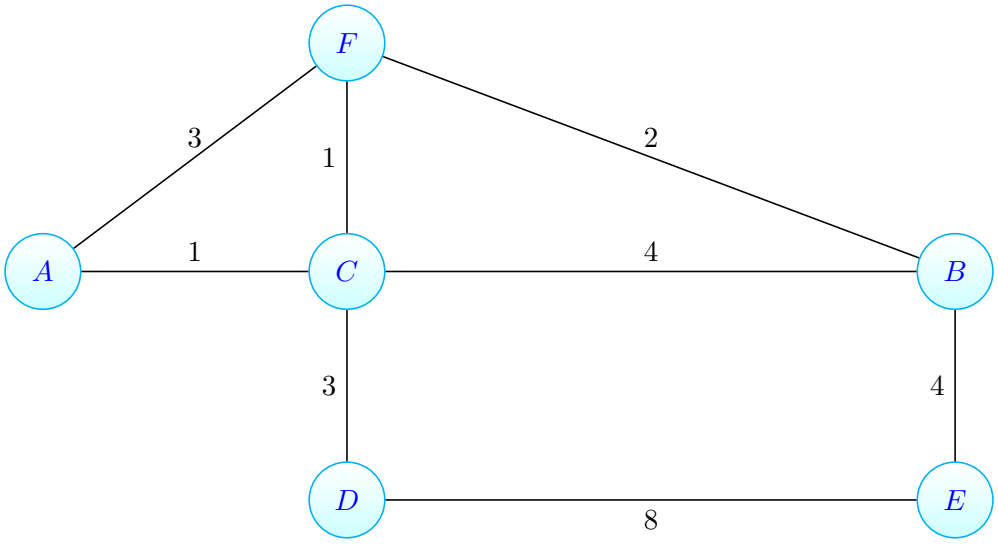


So Dijkstra's algorithm found the following shortest paths from A.

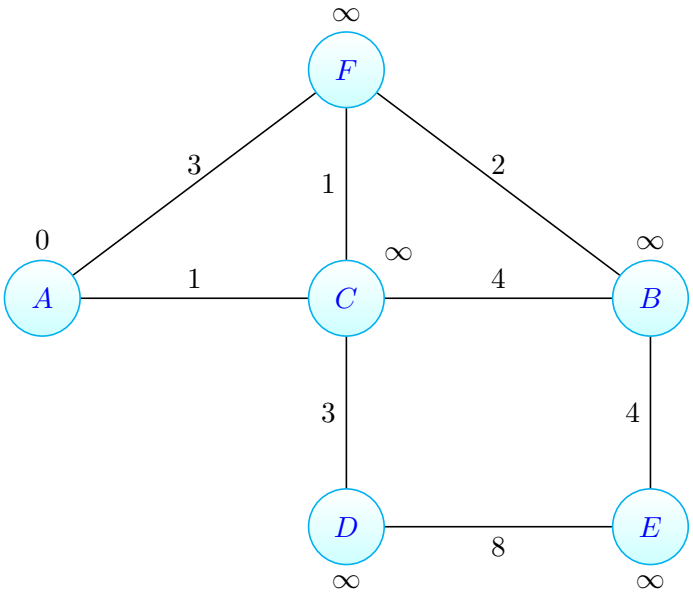
- A to B: The shortest path found was $A - D - B$, which has weight 3.
- A to C: The shortest path found was $A - D - E - C$, which has weight 7.
- A to D: The shortest path found was $A - D$, which has weight 1.
- A to E: The shortest path found was $A - D - E$, which has weight 2.

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.processed = `true`. Now the unprocessed neighbors of A are C and F. Observe that:

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

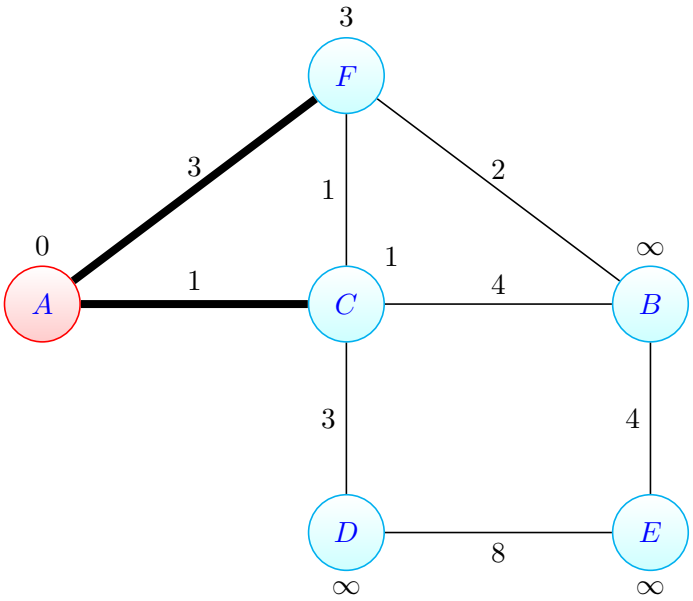
     $$C.\text{dist} = 1,$$
     $$C.\text{predecessor} = A,$$

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

     $$F.\text{dist} = 3,$$
     $$F.\text{predecessor} = A,$$

     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:

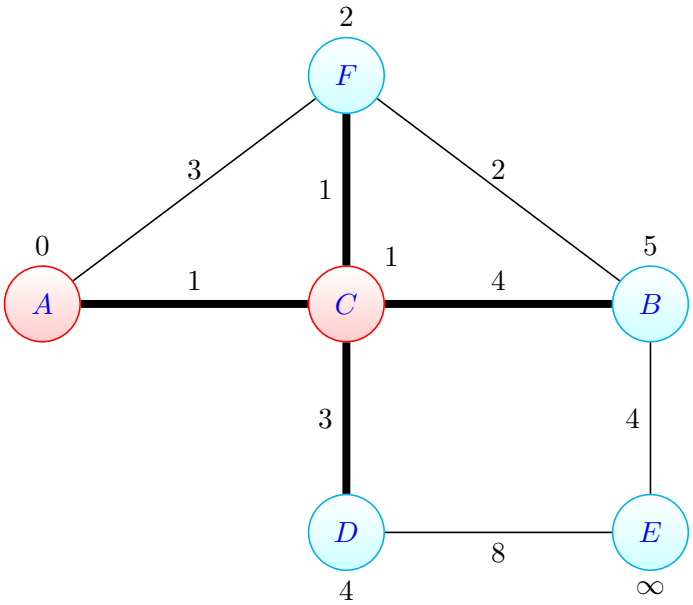
     $$D.\text{dist} = 4,$$
     $$D.\text{predecessor} = C,$$

     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.



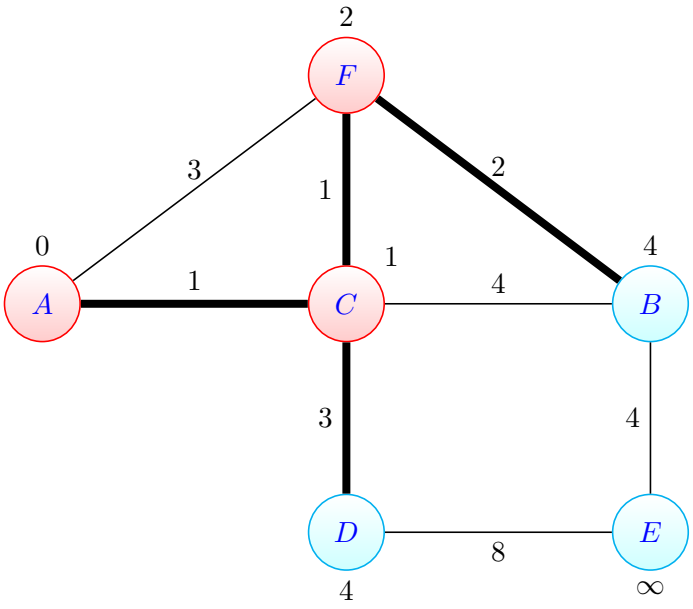
4. We now poll F from the priority queue and set F.processed = `true`. Now the unprocessed neighbor of F is B. Observe that:

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

     $$B.\text{dist} = 4,$$
     $$B.\text{predecessor} = F,$$

     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.



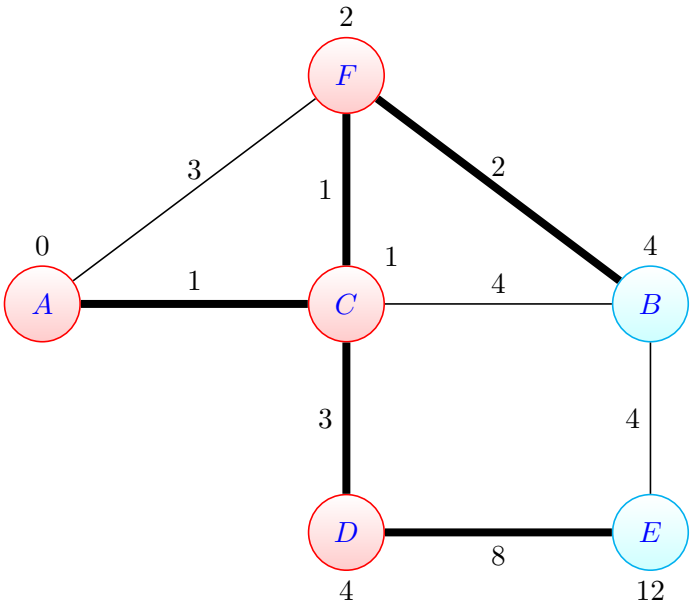
5. We now poll D from the priority queue and set D.processed = `true`. Now the unprocessed neighbor of D is E. Observe that:

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

     $$E.\text{dist} = 12,$$
     $$E.\text{predecessor} = D,$$

     and we push E into the priority queue.

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



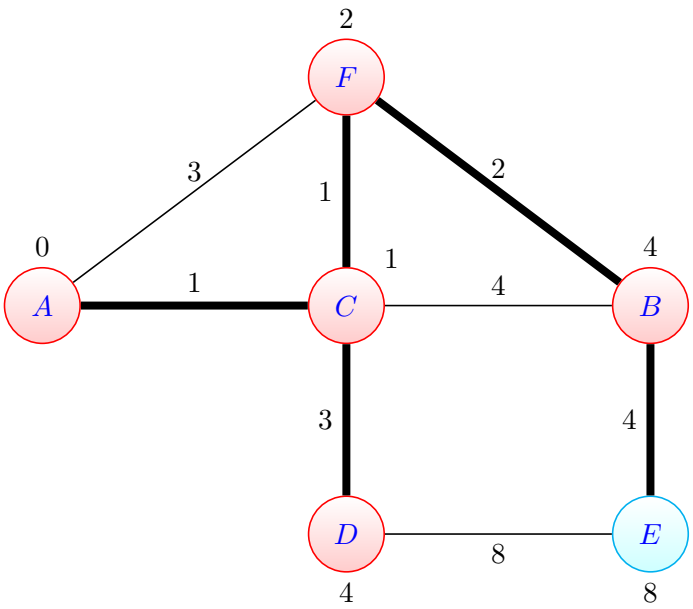
6. We now poll B from the priority queue and set B.processed = `true`. Now the unprocessed neighbor of B is E. Observe that:

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

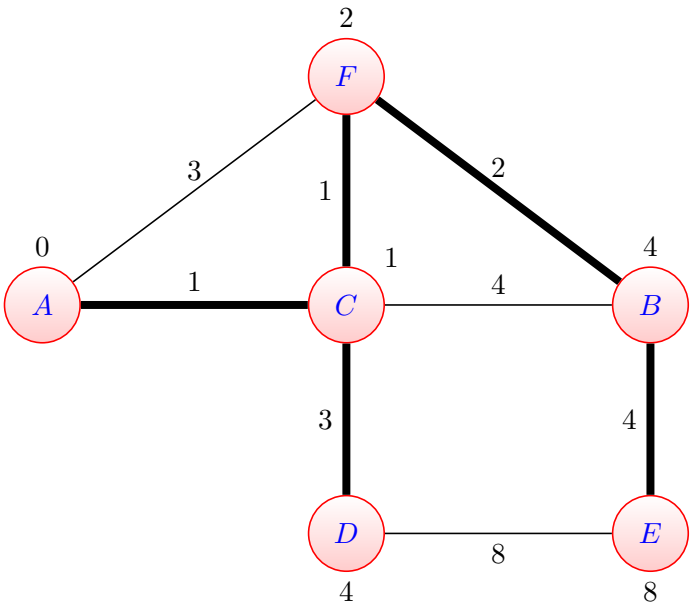
     $$E.\text{dist} = 8,$$
     $$E.\text{predecessor} = B,$$

     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. □

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. □

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 $\text{dist}(s, v_{\ell+1}) = d^*(s, v_{\ell+1})$.

□

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. □

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:

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

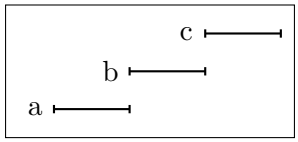
As an example, $[0, 1]$ is the set of real numbers between 0 and 1, including the endpoints 0 and 1. Intuitively, the Interval Scheduling problem takes as input $\mathcal{I}$, a set of intervals. The goal is to find the maximum number of intervals we can select, such that no two intervals overlap.

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

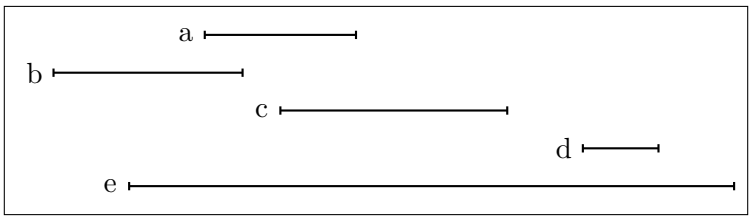
- Instance: Let $\mathcal{I} = \{[s_1, f_1], \ldots, [s_k, f_k]\}$ be our set of intervals.
- Solution: A set $S \subseteq \mathcal{I}$ such that no two intervals in S overlap, where $|S|$ is as large as possible.

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 $|S|$.

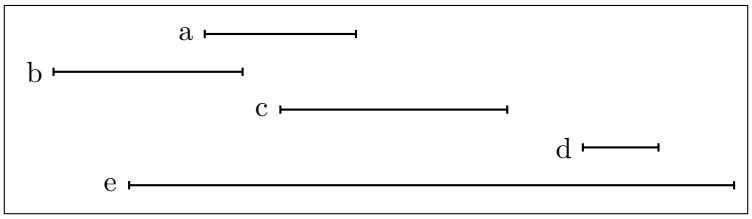
Proof. As $f_{i-1} < s$, we have that $[s, f]$ does not overlap with $[s_1, f_1], \ldots, [s_{i-1}, f_{i-1}]$. Similarly, as $f < f_i < s_{i+1}$, it follows that $[s, f]$ does not overlap with $[s_{i+1}, f_{i+1}], \ldots, [s_m, f_m]$. So S' is a set of pairwise non-overlapping intervals. The result follows.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

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

```
1: procedure GreedyIntervalScheduling(IntervalSet I, Ordering ⪯)
2:     PriorityQueue Q ← []
3:     Q.addAll(I, ⪯)                    ▷ Sort the elements of I according to the ordering ⪯
4:     S ← ∅
5:     while Q ≠ [] do
6:         I ← Q.poll()
7:         if I does not overlap with any interval in S then
8:             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. □

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 [Mou16a].

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. □

3.2.1 Supplemental Reading

The course notes by Mount [Mou17] and Moutadid [Mou16a] 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.

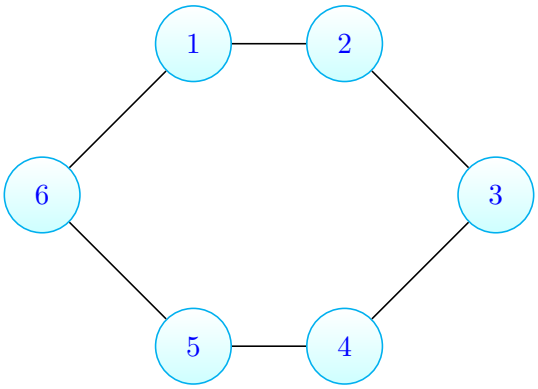
3.3 Example Where the Greedy Algorithm Yields Sub-Optimal Solutions

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.

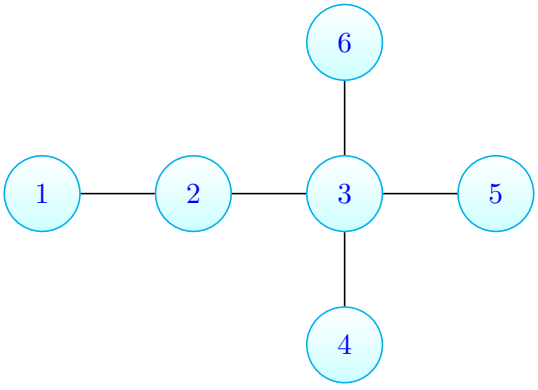
(b) Let $\{i, j\}$ be an arbitrary edge of C_6. Then $\mathcal{M} = \{\{i, j\}\}$ is a matching of C_6. As there is only one edge in $\mathcal{M}$, no two distinct edges of $\mathcal{M}$ share any endpoints.

(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 $|\mathcal{M}| = \nu(G)$.

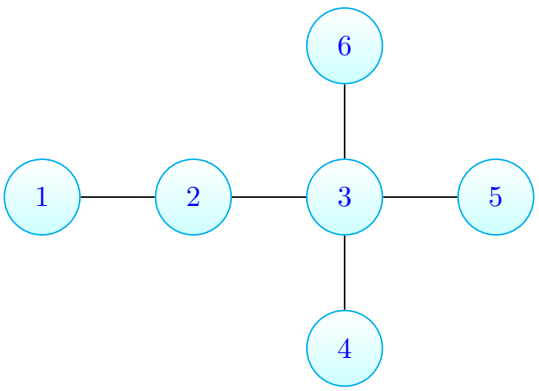
We consider the following greedy algorithm, in an attempt to solve the Maximum-Cardinality Matching problem.

Algorithm 5 GreedyMatching

```
1: procedure GreedyMatching(Graph G)
2:     Queue Q ← []
3:     Q.addAll(E(G))
4:     M ← ∅
5:     while Q ≠ [] do
6:         e ← Q.poll()
7:         if e does not share an endpoint with any edge in M then
8:             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 $u, v \in V(T)$ such that $\{u, v\} \notin E(T)$, 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 $u - v$ path 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.

□

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.

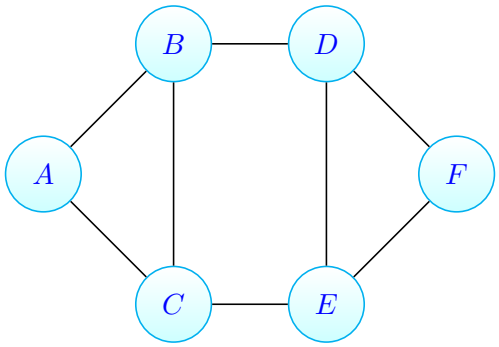
4.2 Safe and Useless Edges

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].

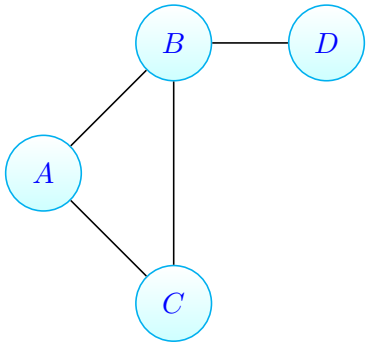
The key approach we will use in designing algorithms to find minimum-weight spanning trees is to manage an *intermediate spanning forest* and add edges until our forest becomes a tree. We wish to add edges in such a way that the resulting spanning tree has minimum weight. We begin with the notions of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

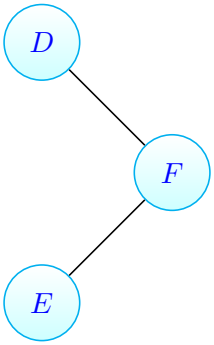
Example 54. Consider the following graph G.



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.

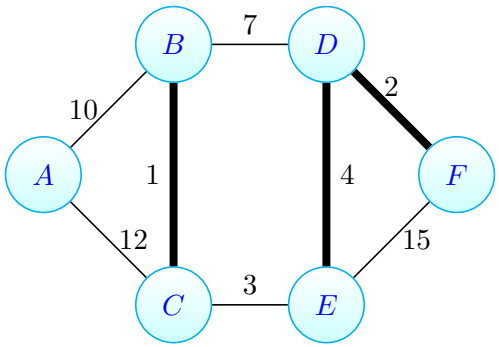


We now turn to formalizing the notion of an *intermediate spanning forest*.

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 $\mathcal{F}$.

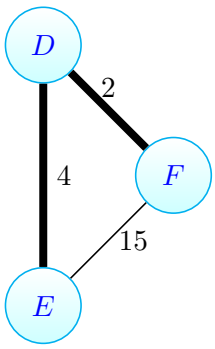
Now we say that $\mathcal{F}$ is an *intermediate spanning forest* if (i) $\mathcal{F}$ is a spanning forest, and (ii) each tree T_j is a minimum-weight spanning tree on the graph induced by $V(T_j)$.

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.



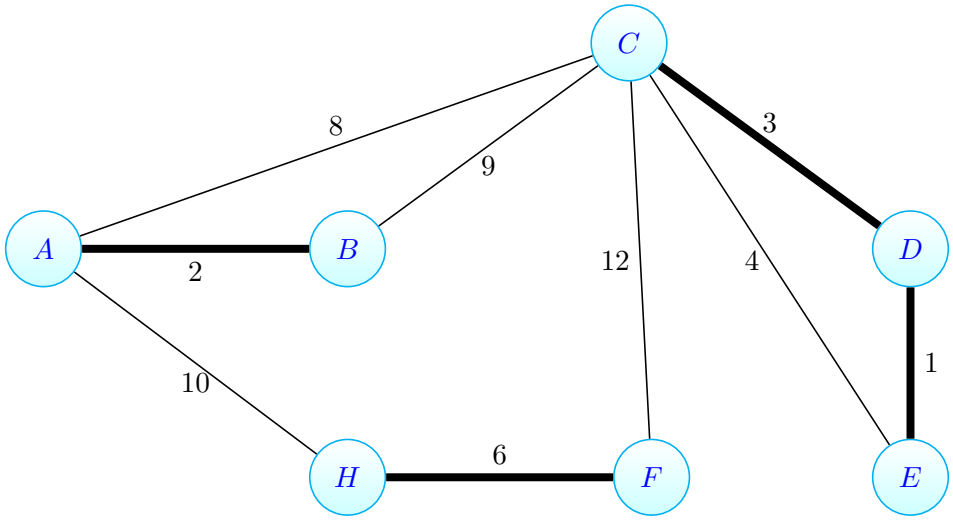
In order to be explicit, we identify the individual trees in $\mathcal{F}$:

- $T_1 = \{A\}$. 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 $\{D, F\}$.



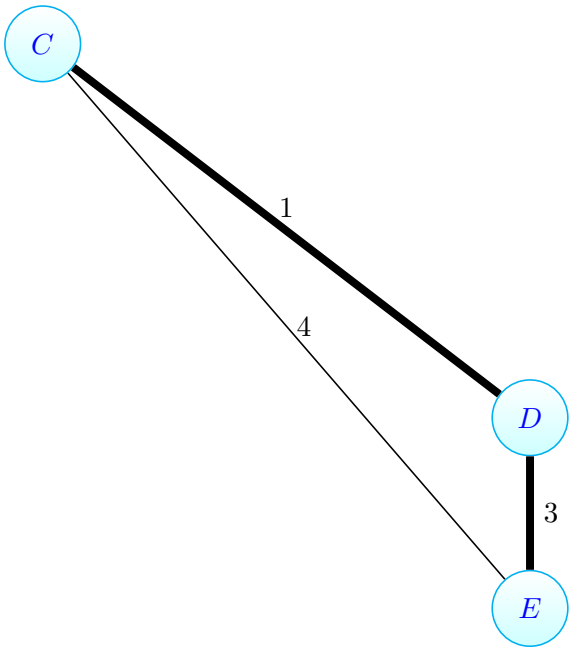
We now consider a second example of an intermediate spanning forest.

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 such that $e \notin \mathcal{F}$.

(a) We say that e is *safe* with respect to $\mathcal{F}$ if $\mathcal{F} \cup e$ is a subgraph 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. □

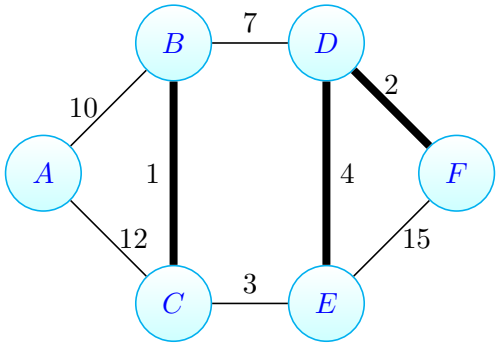
Remark 60. As a corollary, we obtain that a light edge connecting two components in an intermediate spanning forest $\mathcal{F}$ is safe with respect to $\mathcal{F}$. This allows us to easily identify safe edges. In addition, this corollary will be key in establishing the correctness of our minimum-weight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

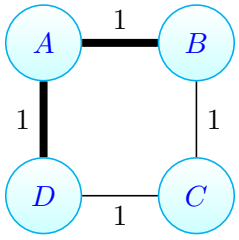
Example 62. Consider again the following weighted graph $G(V, E, w)$. The edges in our intermediate spanning forest $\mathcal{F}$ are indicated by thick edges.



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

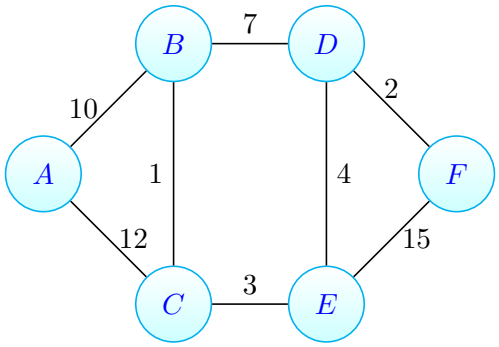
We briefly introduced Kruskal's algorithm in Section 4.1. In this section, we will examine Kruskal's algorithm in more detail. Recall that Kruskal's algorithm places the edges of the input graph into a priority queue. It then polls the edges one at a time, adding the edge e currently being considered 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

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

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

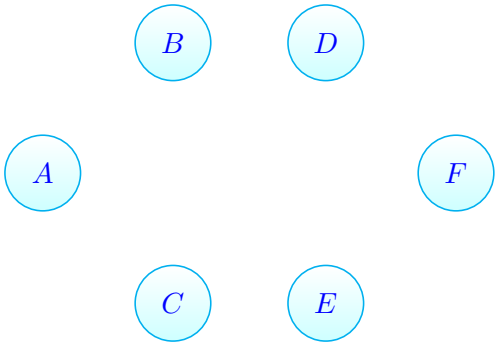


We proceed as follows.

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:

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

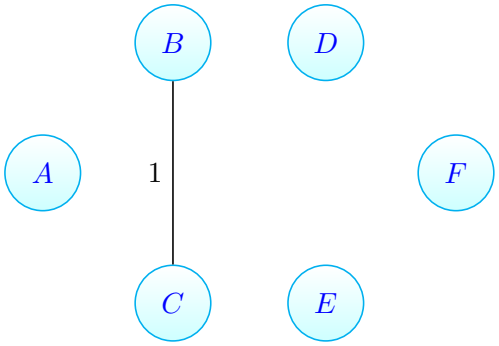
   Here, $(\{B, C\}, 1)$ indicates the edge $\{B, C\}$ has weight 1. The intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

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

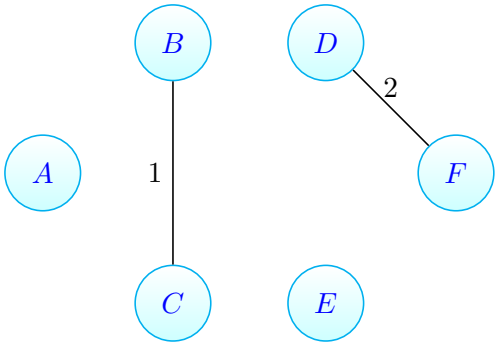
   and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

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

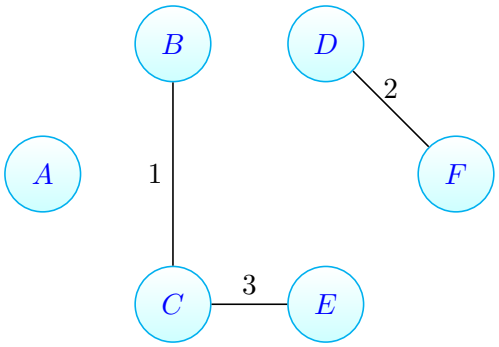
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

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

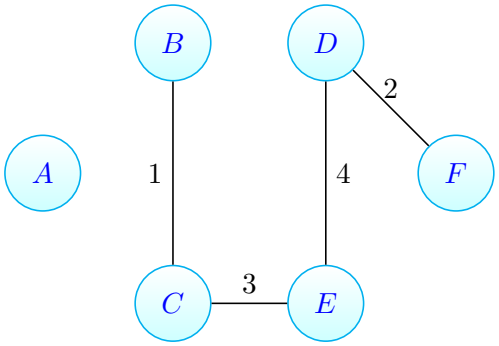
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

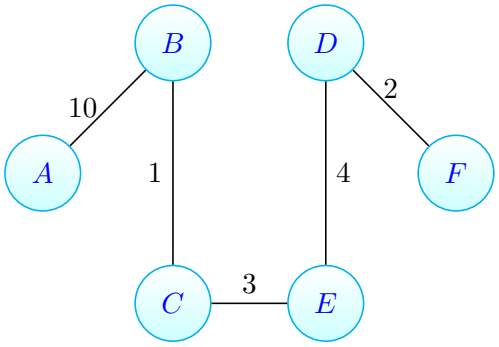
$$Q = [(\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

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. □

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 `find`$(u) =$ `find`(v). Conversely, if `find`$(u) =$ `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.

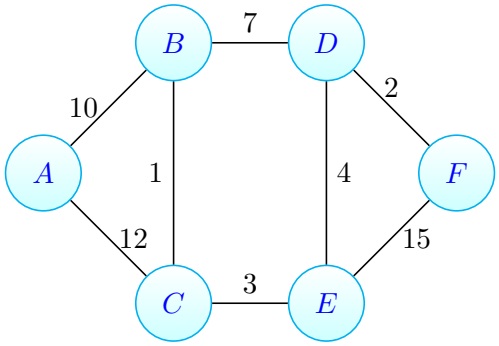
Now at lines 6-8, we initialize the priority queue to contain only edges that are incident to the source vertex. This ensures that the first edge placed into the intermediate spanning forest is incident to the source vertex. Now by adding an edge to $\mathcal{F}$, we introduce a new vertex v to the component 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.

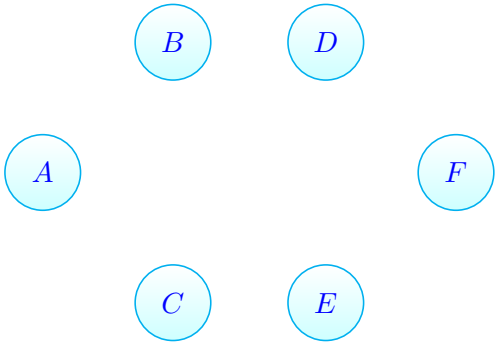


Prim's algorithm proceeds as follows.

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:

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

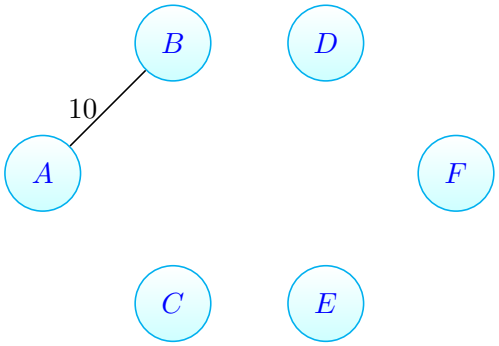
and our intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

$$Q = [(\{B, C\}, 1), (\{B, D\}, 7), (\{A, C\}, 12)],$$

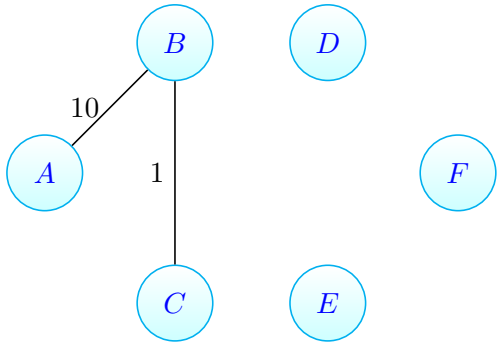
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

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

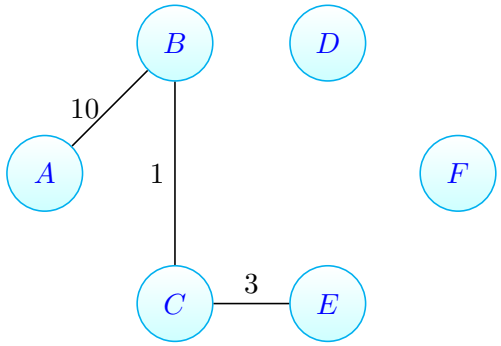
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

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

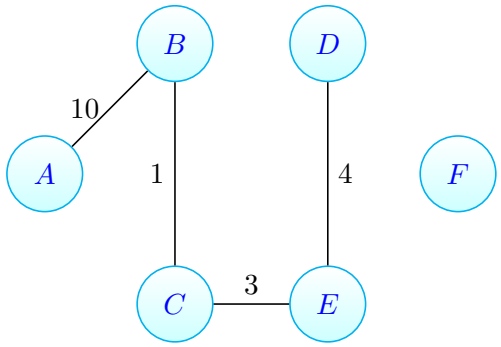
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

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

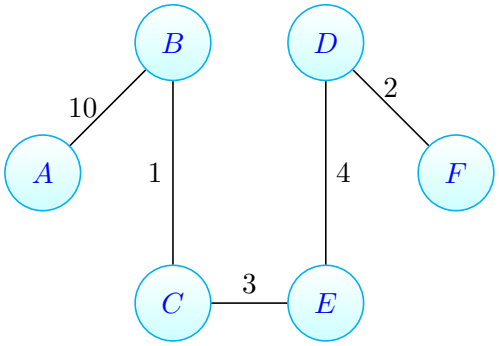
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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.



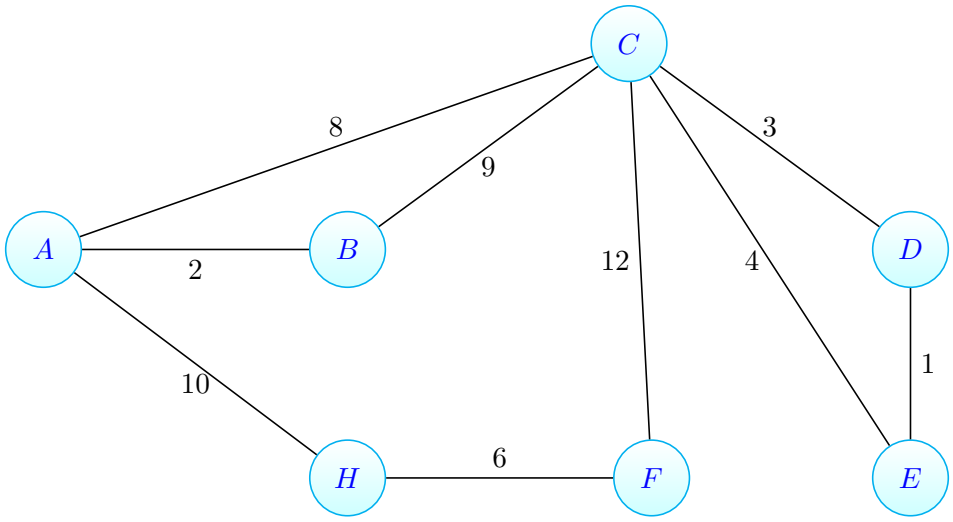
7. As $\mathcal{F}$ has $|V(G)| - 1 = 6 - 1 = 5$ edges, the algorithm terminates and returns $\mathcal{F}$, pictured in Step 6 immediately above.

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.

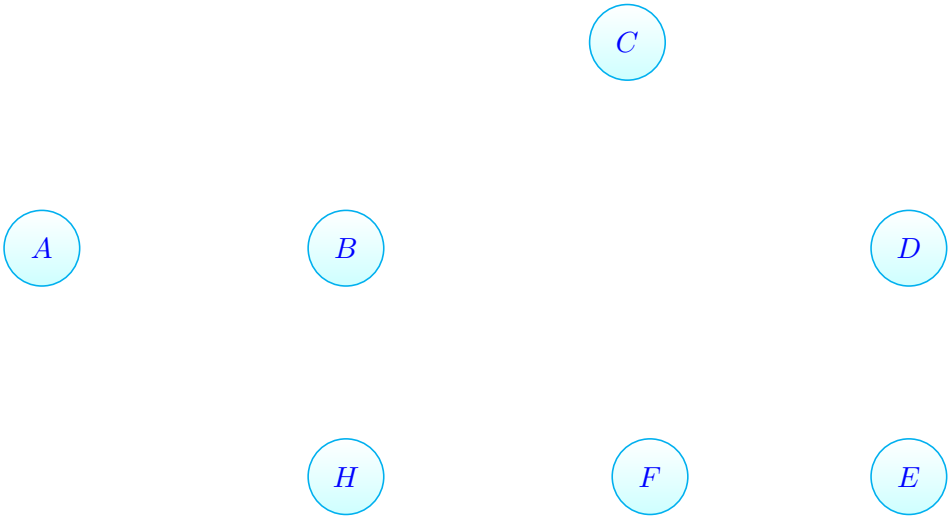


Prim's algorithm proceeds as follows.

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:

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

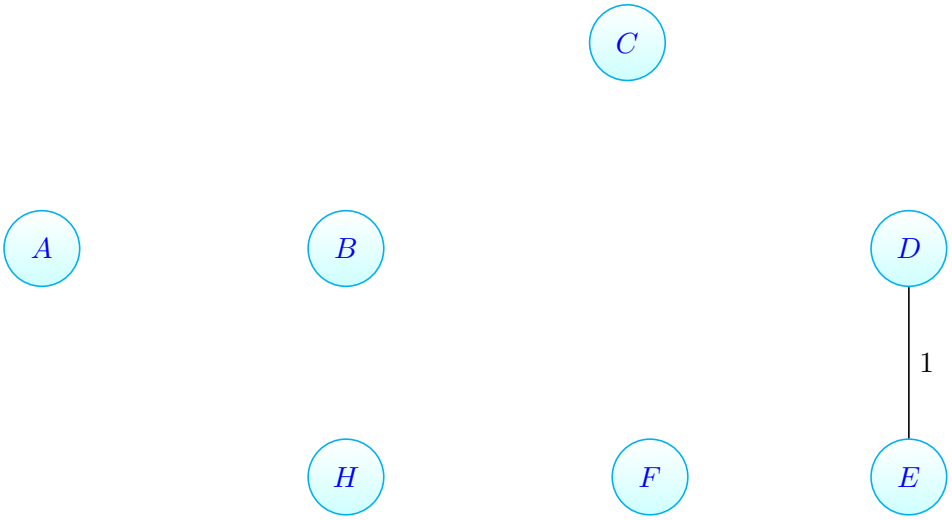
   and our intermediate spanning forest $\mathcal{F}$ is pictured below.



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:

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

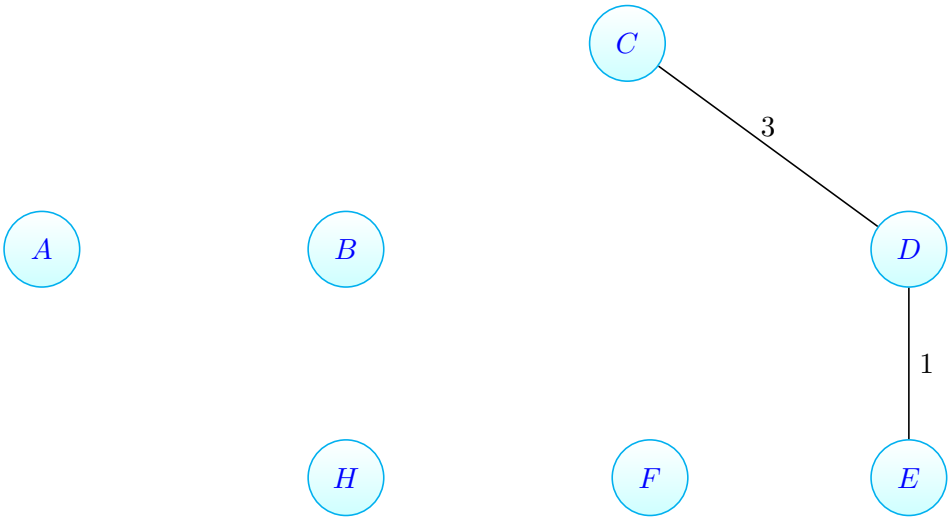
   and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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.



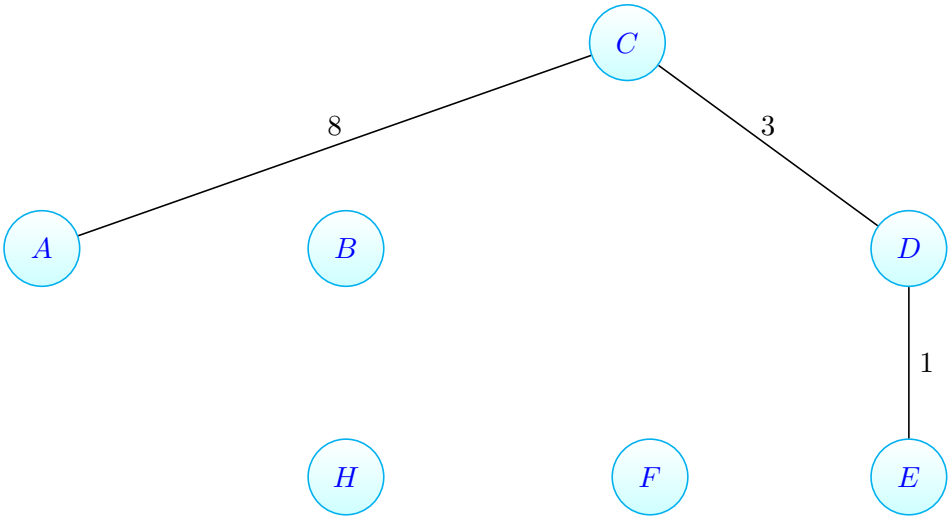
4. We poll the edge $\{E, C\}$ from the queue and mark $\{E, C\}$ as processed. As both E and C belong to the component containing D, we do **not** add $\{E, C\}$ to $\mathcal{F}$. The updated priority queue is below, and the intermediate spanning forest $\mathcal{F}$ does not change from the previous 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:

$$Q = [(\{A, B\}, 2), (\{C, B\}, 9), (\{A, H\}, 10), (\{C, F\}, 12)],$$

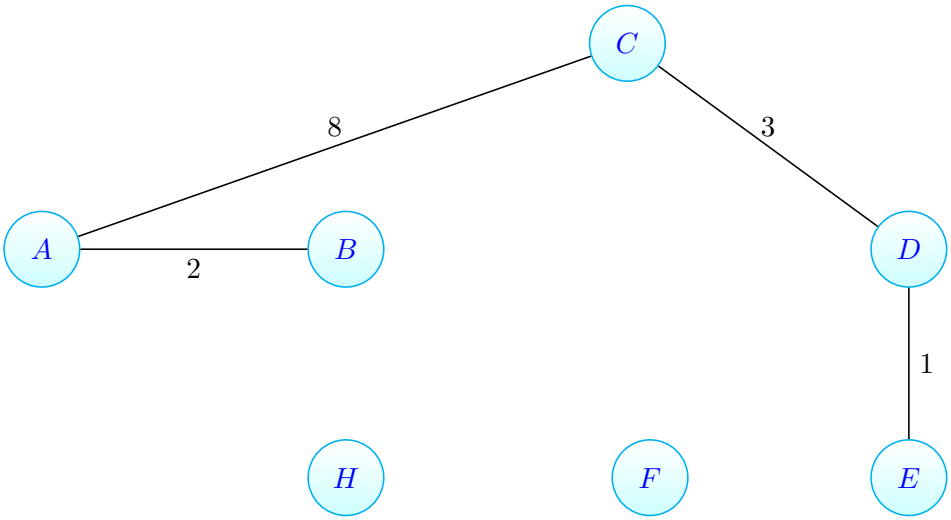
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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.



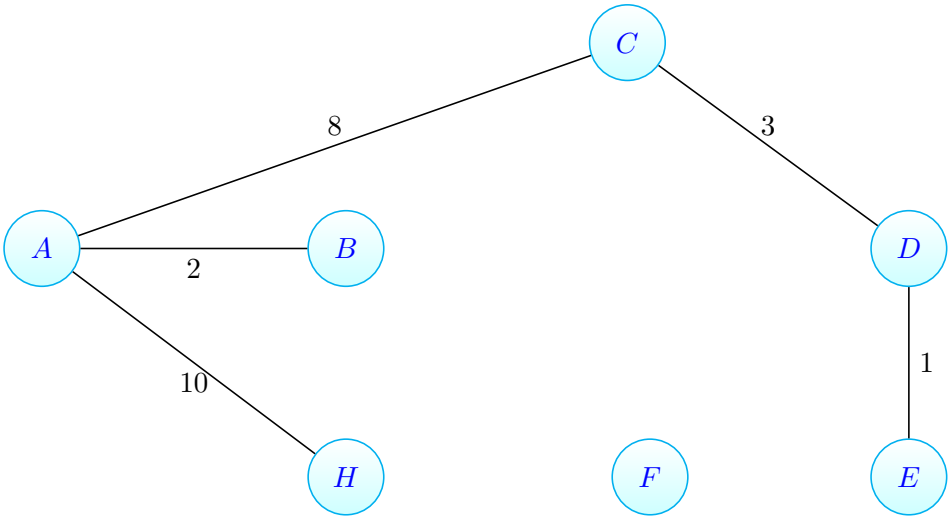
7. We poll the edge $\{C, B\}$ from the queue and mark $\{C, B\}$ as processed. As both C and B belong to the component containing D, we do **not** add $\{C, B\}$ to $\mathcal{F}$. The updated priority queue is below, and the intermediate spanning forest $\mathcal{F}$ does not change from the previous 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:

$$Q = [(\{H, F\}, 6), (\{C, F\}, 12)],$$

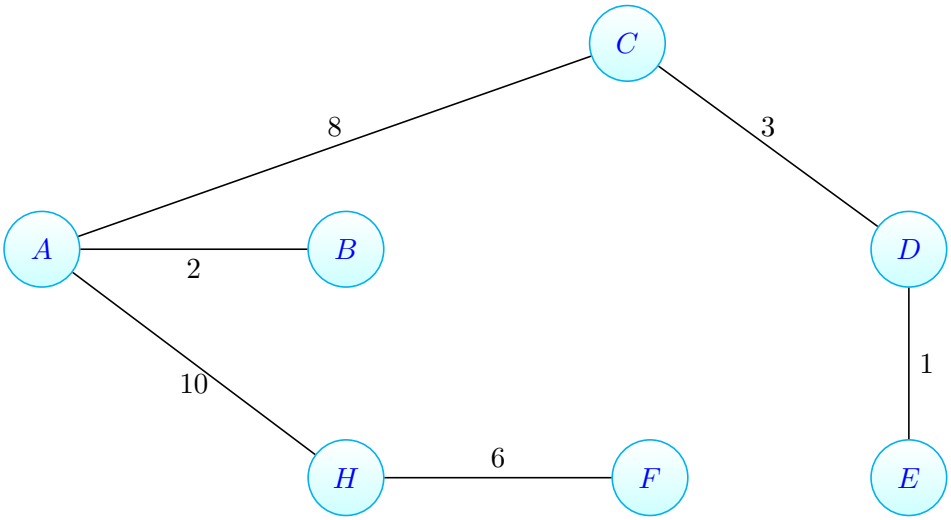
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



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. □

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. □

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. □

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. □

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:

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

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 $(u, v) \in V(G) \times V(G)$, $f(u, v) \leq c(u, v)$. That is, we cannot push more flow across a given edge than the capacity permits. Similarly, if the pair $(u, v) \notin E(G)$, then we cannot push flow across (u, v). That is, we can only push flow along edges.
- 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.

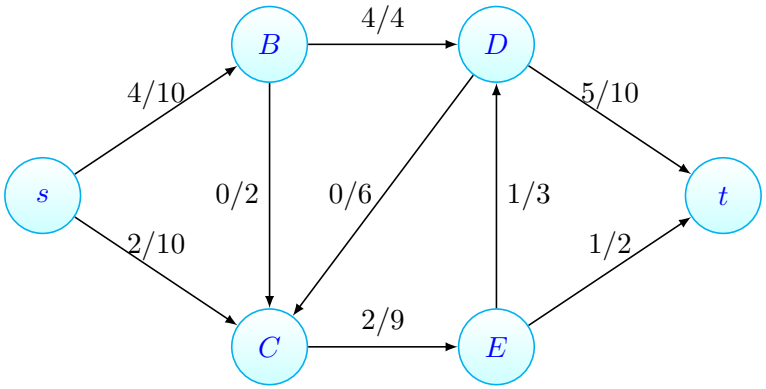
$$\sum_{(u,v)\in E(G)} f(u, v) = \sum_{(v,w)\in E(G)} f(v, w).$$

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.

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

Example 87. 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, B) has capacity 10, and there are 4 units of flow being pushed from $s \to B$.
- The edge (B, D) has capacity 4, and there are 4 units of flow being pushed from $B \to D$.
- The edge (D, t) has capacity 10, and there are 5 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 $C \to E$.

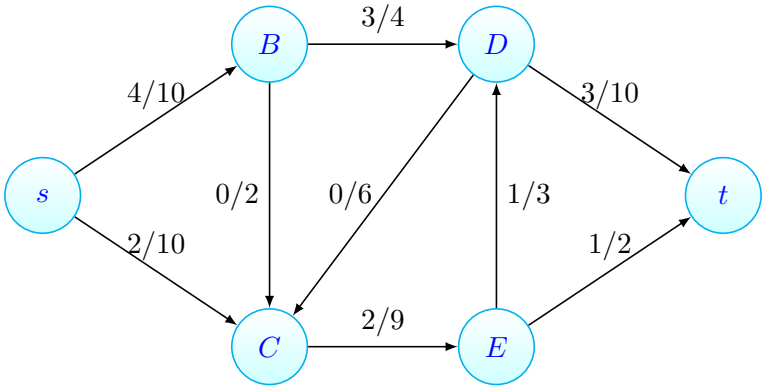


Additionally, observe that for each vertex v other than the source s and the sink t, the flow coming into v is equal to the flow leaving v. We check this for the vertices other than s and t.

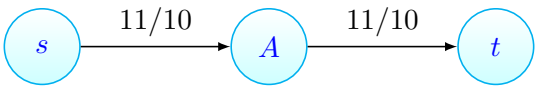
- Observe that there are 4 units of flow coming into vertex B via the edge (s, B). These 4 units of flow leave B along the edge (B, D).
- Observe that there are 4 units of flow coming into D via the edge (B, D). These 4 units of flow leave D along the edge (D, t).
- Observe that there are 2 units of 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 88. 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 89. 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.

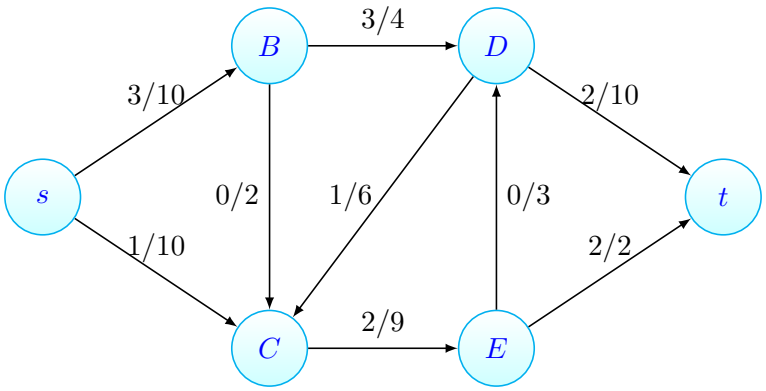
Definition 90. The Maximum Flow problem is defined as follows.

- 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 $u \to v$. Alternatively, we may push $f(u, v)$ units of flow back from $v \to u$. We then re-route this flow at v.

Example 91. 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

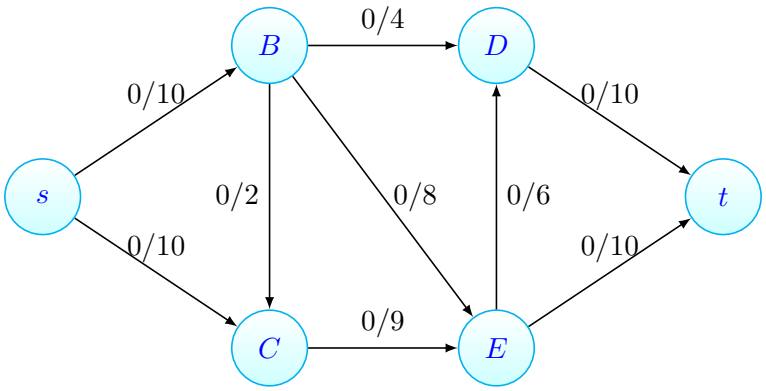
```
1: procedure FordFulkerson(FlowNetwork 𝒩)
2:     while 𝒩 has a flow-augmenting path do
3:         Let P be a flow-augmenting path of 𝒩
4:         Push as much flow as possible along P
```

Remark 92. 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.



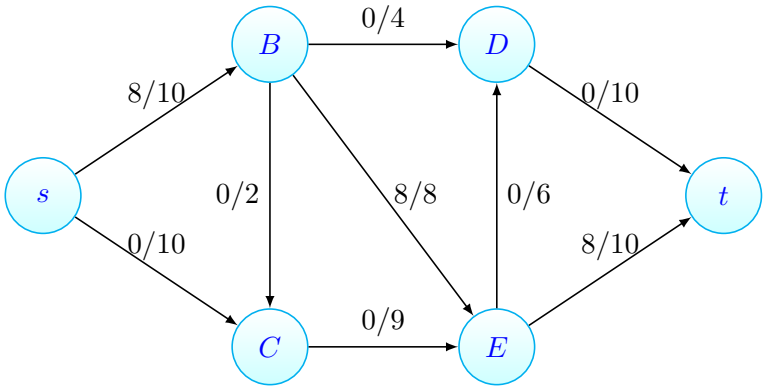
We proceed as follows.

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.

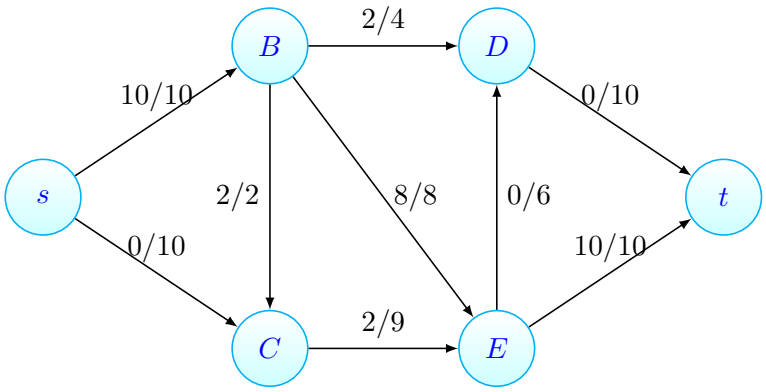


2. There are several flow-augmenting paths we could select at the second iteration, including:

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

   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.

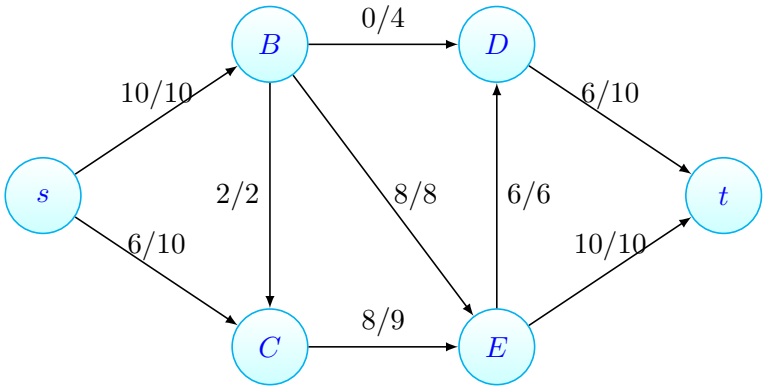


3. There are several flow-augmenting paths we could select at the third iteration, including:

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

   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.

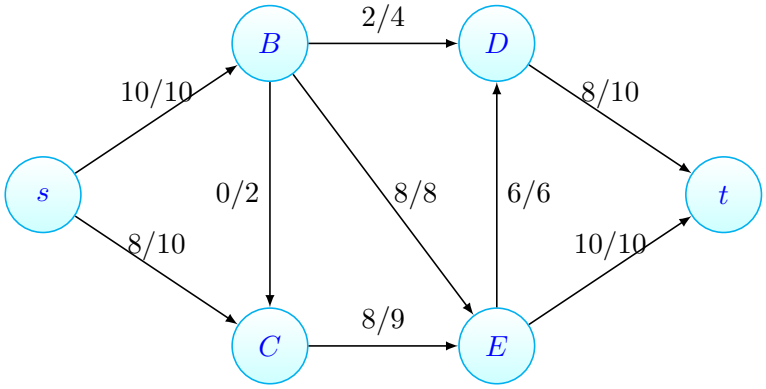


4. There are two flow-augmenting paths we could select at the fourth iteration, including:

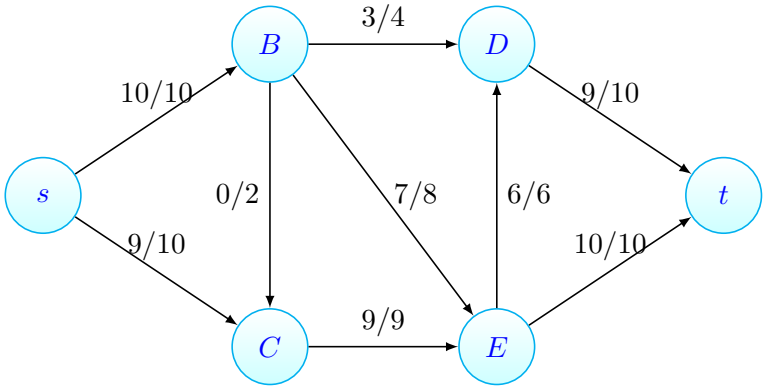
   - $s \to C \to B \to D \to t$
   - $s \to C \to E \to B \to D \to t$.

   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.



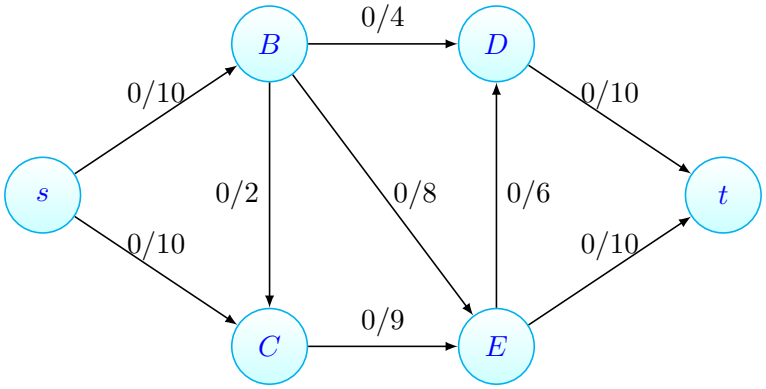
5. There is one flow-augmenting path at the fifth iteration: $s \to C \to E \to B \to D \to t$. We push the full 1 unit of flow across this path. The updated and final flow network is below.



   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.



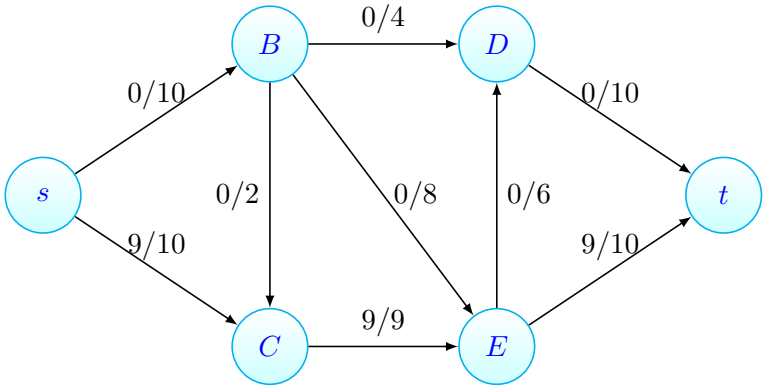
We proceed as follows.

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.

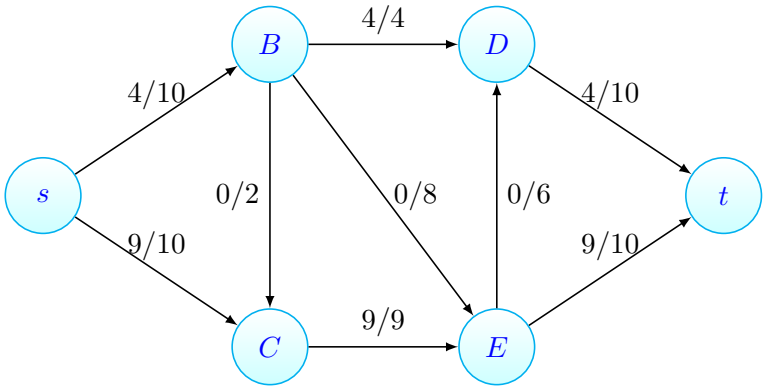


2. There are several flow-augmenting paths we could select at the second iteration, including:

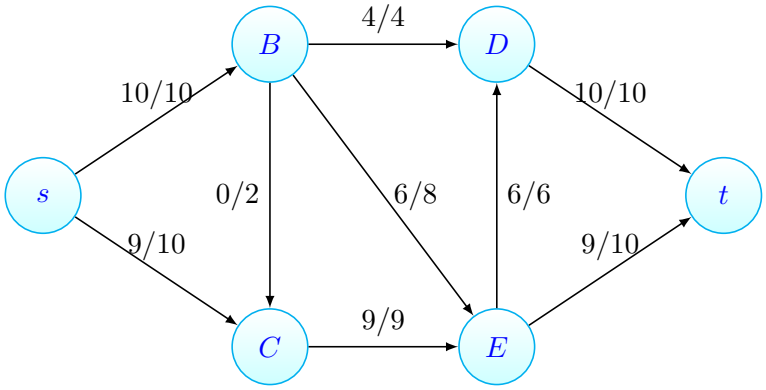
   - $s \to B \to D \to t$
   - $s \to B \to E \to D \to t$
   - $s \to B \to E \to t$.

   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 are two available flow-augmenting paths at the third iteration: $s \to B \to E \to D \to t$ and $s \to B \to E \to t$. We select the path $s \to B \to E \to D \to t$ and 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 93. 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:

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

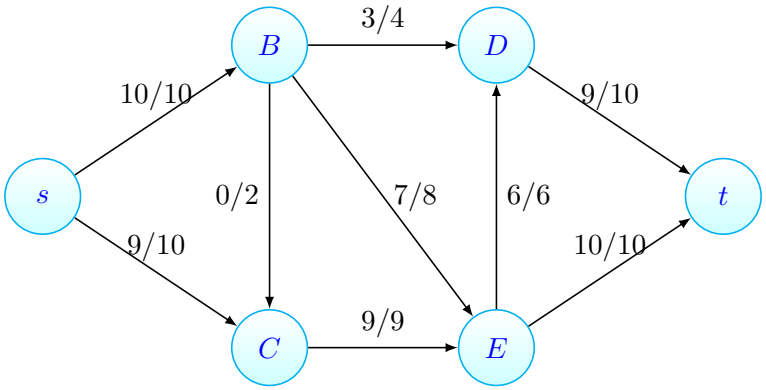
Recall that if (x, y) is not an edge of the flow network, then $c(x, y) = 0$.

Definition 94. 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 95. 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.

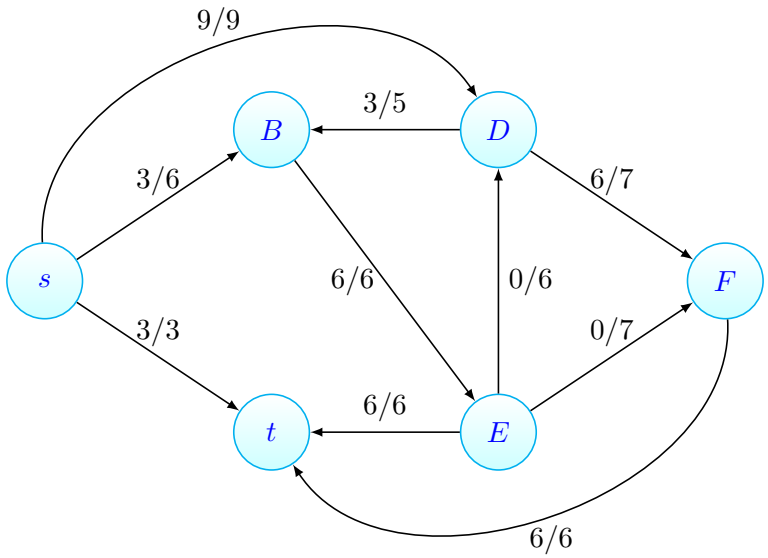
- The source vertex s belongs to X.
- We may push 1 unit of flow from $s \to C$. So C belongs to X.

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 96. 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 $s \to t$.

5.5 Max-Flow Min-Cut Theorem

In this section, we prove the Max-Flow Min-Cut Theorem. Our proof is based on [Mou16b].

Theorem 97 (Max-Flow Min-Cut Theorem (Ford-Fulkerson, 1956).)**.** Let $\mathcal{N}(G, c, S, T)$ be a flow network. Let f^* be a maximum-valued flow, and let (X, Y) be a minimum-capacity cut. We have that $\text{val}(f^*) = c(X, Y)$.

We prove Theorem 97. We first show that the value of a maximum flow is no bigger than the capacity of a minimum cut. We then show that there exists a cut whose capacity is no bigger than the value of a maximum flow. It follows from this second claim that the capacity of a minimum cut is no bigger than the value of a maximum flow.

We begin by showing that the value of a maximum flow is no bigger than the capacity of a minimum cut. To this end, we introduce the following lemma, which intuitively states that the amount of flow that we can push from the source nodes to the sink nodes cannot exceed the capacity of a cut.

Lemma 98. Let $\mathcal{N}(G, c, S, T)$ be a flow network, and let f be a flow. Let (X, Y) be a cut. We have that $\text{val}(f) \leq c(X, Y)$.

Proof. As (X, Y) is a cut, we have by the conservation of flow the total flow that makes it from the source vertices to the sink vertices is the amount of flow leaving X, minus the amount of flow returning to X from Y. This is precisely:

$$\text{val}(f) = \sum_{u \in X} \sum_{v \in Y} f(u, v) - \sum_{u \in Y} \sum_{v \in X} f(u, v).$$

By ignoring the flow coming back into X, we have that:

$$\begin{aligned}
\text{val}(f) &= \sum_{u \in X} \sum_{v \in Y} f(u, v) - \sum_{u \in Y} \sum_{v \in X} f(u, v) && (16)\\
&\leq \sum_{u \in X} \sum_{v \in Y} f(u, v) && (17)\\
&\leq \sum_{u \in X} \sum_{v \in Y} c(u, v) && (18)\\
&= c(X, Y). && (19)
\end{aligned}$$

Here, line (19) follows from the fact that $f(u, v) \leq c(u, v)$ for all $u, v \in V(G)$. The result follows. □

Remark 99. Lemma provides that if f is a flow and (X, Y) is a cut, then $\text{val}(f) \leq c(X, Y)$. Maximizing over the choice of flows, we obtain:

$$\max_{f \text{ is a flow}} \text{val}(f) \leq c(X, Y).$$

Minimizing over the cuts, we obtain the desired result:

$$\max_{f \text{ is a flow}} \text{val}(f) \leq \min_{(X,Y) \text{ is a cut}} c(X, Y).$$

Now to establish that:

$$\min_{(X,Y) \text{ is a cut}} c(X, Y) \leq \max_{f \text{ is a flow}} \text{val}(f),$$

we first prove a more general theorem.

Theorem 100. Let $\mathcal{N}(G, c, S, T)$ be a flow network, and let f be a flow on $\mathcal{N}$. The following are equivalent.

(a) f is a maximum-valued flow.

(b) There are no flow-augmenting paths.

(c) There exists a cut (X, Y) corresponding to f such that $c(X, Y) = \text{val}(f)$.

Proof. We have the following.

- $(a) \implies (b)$: Suppose that f is a maximum-valued flow, and suppose to the contrary that there is a flow-augmenting path P. Let f' be the flow obtained by starting with f and pushing as much flow as possible through P. So $\text{val}(f') > \text{val}(f)$, contradicting the assumption that f was maximal.

- $(b) \implies (c)$: Suppose there exist no flow-augmenting paths. We construct the set X such that (i) X contains the source nodes (that is, $S \subseteq X$), and (ii) X contains a vertex v precisely if we can push positive flow to v from a given source node. Now let $Y := V(G) \setminus X$. As there are no flow-augmenting paths, no sink node belongs to X. Thus, (X, Y) is a cut.

  We now claim that $c(X, Y) = \text{val}(f)$. Let (u, v) be an edge that crosses the cut; that is, $u \in X$ and $v \in Y$. As $v \notin X$, we cannot push positive flow from a source node to v. So $f(u, v) = c(u, v)$. It follows that $\text{val}(f)$ is precisely the flow that is pushed from X to Y, which we denote $f(X, Y)$. In particular, we have that:

$$\begin{aligned} \text{val}(f) &= f(X, Y) && (20)\\ &= \sum_{u \in X} \sum_{v \in Y} f(u, v) - \sum_{a \in Y} \sum_{b \in X} f(x, y) && (21)\\ &= \sum_{u \in X} \sum_{v \in Y} c(u, v) && (22)\\ &= c(X, Y), && (23) \end{aligned}$$

  as desired.

- $(c) \implies (a)$: Suppose there exists a cut (X, Y) corresponding to f such that $c(X, Y) = \text{val}(f)$. Let f' be a maximum-valued flow. So we have that:

$$\begin{aligned} c(X, Y) &= \text{val}(f)\\ &\leq \text{val}(f')\\ &\leq c(X, Y). \end{aligned}$$

  The last inequality follows from Lemma 99.

□

Theorem 100 immediately implies Theorem 97.

5.6 Bipartite Matching

In this section, we show that the Maximum Flow problem can be used to find maximum-cardinality matchings in bipartite graphs. The key technique is the reduction. Informally, suppose we have two computational problems $\mathcal{A}$ and $\mathcal{B}$. The idea behind a reduction is as follows:

- Let $A \in \mathcal{A}$ be an instance of A. We apply a transformation T to obtain $T(A) \in \mathcal{B}$.
- Apply our algorithm to solve B to obtain a solution for $T(A)$.
- Use the solution for $T(A)$ to recover a solution for A.

If each of these steps are efficiently computable, then our reduction provides an efficient algorithm to solve $\mathcal{A}$. Reductions also provide an ordering amongst computational problems. Namely, if $\mathcal{A}$ reduces to $\mathcal{B}$, then $\mathcal{A}$ is no harder than $\mathcal{B}$. This ordering allows us to formalize the notion of what constitutes a hard computational problem, which is a fundamental idea in Computational Complexity. We will revisit the technicalities of reductions and their theoretical properties later during our discussions of the P vs. NP problem.

For now, we focus on the idea of reducing to the Maximum Flow problem. We begin by introducing the Maximum Bipartite Matching problem.

Definition 101. The Maximum Bipartite Matching problem is defined as follows.

- Instance: Let $G(L \dot{\cup} R, E)$ be a bipartite graph.
- Solution: A maximum-cardinality matching $\mathcal{M}$ of G.

The notion of a matching was introduced in Section 3.3. We defer the reader there for examples. We now turn to proving our main theorem.

Theorem 102. The Maximum Bipartite Matching problem reduces to the Maximum Flow problem.

Proof. Let $G(L \dot{\cup} R, E)$ be a bipartite graph. We construct a corresponding flow network $\mathcal{N}(H, c, S, T)$ from G as follows.

- H contains the vertices of G, together with a source node s and a sink node t.
- Let $u \in L$ and $v \in R$. If $\{u, v\} \in E(G)$, then we include a directed edge $(u, v) \in E(H)$.
- For each $u \in L$, we include a directed edge (s, u) in $E(H)$.
- For each $v \in R$, we include a directed edge (v, t) in $E(H)$.
- Each edge in H has capacity 1.

We now claim that G has a matching of size k if and only if $\mathcal{N}$ admits a flow f where (i) $\text{val}(f) = k$ and (ii) for each edge $(u, v) \in E(H)$, $f(u, v) = 0$ or $f(u, v) = 1$. That is, condition (ii) effectively states that there is no fractional flow in G.

Proof of Claim. Suppose first that G has a matching $\mathcal{M} = \{\{u_1, v_1\}, \ldots, \{u_k, v_k\}\}$ of size k. We construct a flow f of value k on $\mathcal{N}$ as follows. For each edge $\{u_i, v_i\} \in \mathcal{M}$, we push 1 unit of flow along the path $s \to u_i \to v_i \to t$. As no two edges in $\mathcal{M}$ share a common endpoint, f is a feasible flow. Furthermore, $\text{val}(f) = k$ where each edge has either 0 or 1 unit of flow.

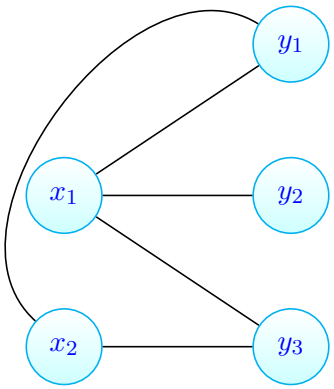
Conversely, suppose that $\mathcal{N}$ has a feasible flow f' of value k where each edge has either 0 or 1 unit of flow. Let S be the set of edges of the form $(u, v) \in E(H)$ where $f'(u, v) = 1$, $u \neq s$, and $v \neq t$. That is, (u, v) corresponds to an edge $\{u, v\}$ in G, and (u, v) is currently receiving flow. We claim that the set:

$$\mathcal{M} = \{\{u, v\} \in E(G) : (u, v) \in S\}$$

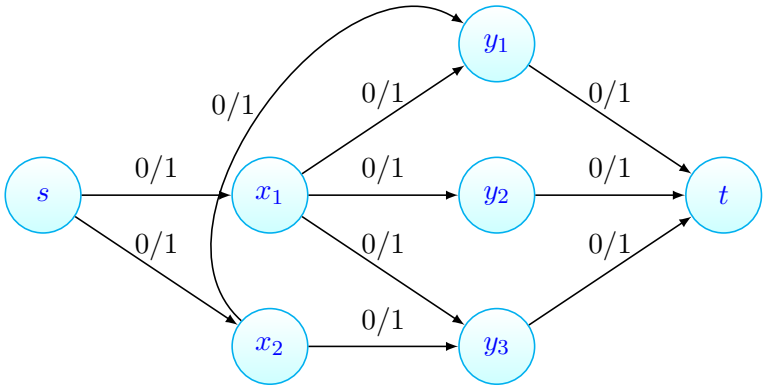
is a matching of size k. As f' is a feasible flow, we have by the conservation of flow condition that if $(u, v) \in S$, then (u, v) is the only edge incident to u that has positive flow, and (u, v) is the only edge incident to v with positive flow. It follows that no two edges in S share a common endpoint. Thus, no two edges in $\mathcal{M}$ share a common endpoint. So $\mathcal{M}$ is a matching. As $\text{val}(f') = k$ and each edge has flow value 0 or 1 under f', it follows that $|S| = k$. As $|\mathcal{M}| = |S|$, we have that $|\mathcal{M}| = k$. This completes the proof of the claim. □

□

Example 103. We now work through an example of this construction. Let G be the graph pictured below.

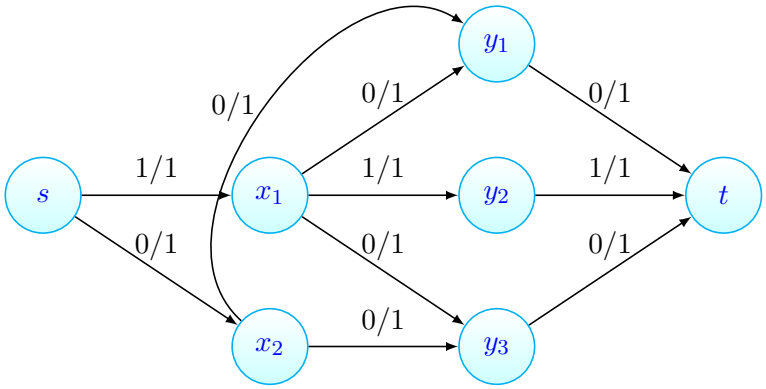


- **Step 1:** We begin by constructing the corresponding flow network $\mathcal{N}$:

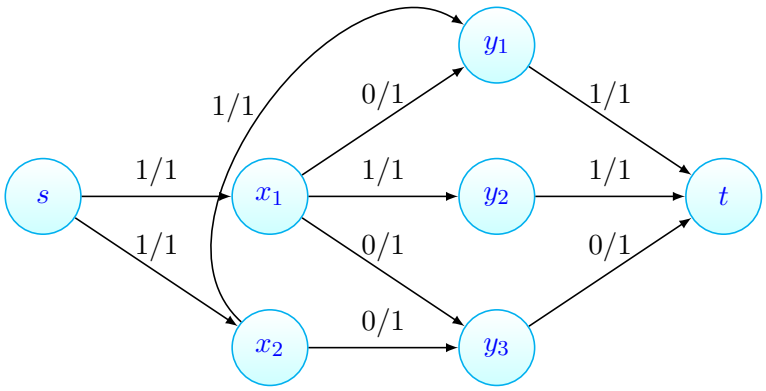


- **Step 2:** We now find a maximum flow. **We stress that the selection of our flow-augmenting paths below is a choice. There are other valid choices; that is, our selection is in a sense arbitrary.**

  – Observe that we have a flow-augmenting path $s \to x_1 \to y_2 \to t$, allowing us to push 1 unit of flow from $s \to t$. This yields the updated flow network:



  – Observe that we have a flow-augmenting path $s \to x_2 \to y_1 \to t$, allowing us to push 1 unit of flow from $s \to t$. This yields the updated flow network:



  There are no more flow-augmenting paths. So we now have a maximum flow on $\mathcal{N}$.

- **Step 3:** We now construct the matching on G based on our maximum flow on $\mathcal{N}$. Since the edges (x_1, y_2) and (x_1, y_2) are saturated on $\mathcal{N}$, our corresponding maximum matching $\mathcal{M} = \{\{x_1, y_2\}, \{x_2, y_1\}\}$.

Remark 104. We note that other maximum flows on $\mathcal{N}$ would yield different maximum matchings. We leave it as an exercise for the reader to find the other maximum matchings of G.

6 Asymptotic Analysis

Our goal in this section is to develop tools to rigorously analyze the runtime complexity of our algorithms. We will begin by formalizing our measures of complexity, including Big-O, Big-Omega, Big-Theta, little-o, and little-omega. Next, we turn to analyzing iterative and recursive algorithms. For our discussions of recursive algorithms, we will introduce the unrolling and tree methods. Such techniques will be key when analyzing divide and conquer algorithms, as well as dynamic programming algorithms.

6.1 Asymptotics

We begin by recalling the notions of Big-O, Big-Omega, and Big-Theta from CSCI 2270 and CSCI 2824. As we recall the definitions, we will also highlight examples for the purpose of developing intuition as to what the asymptotic expressions represent. However, we will not initially prove the claims made in the examples. Rather, after ensuring that the meaning behind the asymptotic expressions is clear, we will then recall techniques from Calculus that will allow us to formally prove the claims from our examples.

Definition 105. Let k be an integer. Denote $\mathbb{Z}_{\geq k} = \{n \in \mathbb{Z} : n \geq k\}$ to be the set of integers that are greater than or equal to k.

Definition 106 (Big-O)**.** Let k be an integer. Let $f, g : \mathbb{Z}_{\geq k} \to \mathbb{R}$ be functions. We say that $f(n) \in O(g(n))$ if there exist constants $c, h \in \mathbb{Z}^+$ such that $|f(n)| \leq c \cdot |g(n)|$ for all $n \geq h$.

Example 107. You may recall from previous courses that $n^2 \in O(n^3)$, for example. Similarly, $2^n \in O(n!)$. However, $n! \notin O(2^n)$. While a simple induction proof is sufficient to establish that $2^n \in O(n!)$, this technique does not provide that $n! \notin O(2^n)$. To this end, we will require Calculus techniques, which will be introduced later.

In some sense, we think of Big-O as a *asymptotic weak upper bound*. That is, if $f(n) \in O(g(n))$, we say that $g(n)$ grows at least as quickly as $f(n)$ in the long run (that is, for all $n \geq k$). In contrast, Big-Omega is our *asymptotic weak lower bound*, and it is defined analogously as Big-O.

Definition 108 (Big-Omega)**.** Let k be an integer. Let $f, g : \mathbb{Z}_{\geq k} \to \mathbb{R}$ be functions. We say that $f(n) \in \Omega(g(n))$ if there exist constants $c, h \in \mathbb{Z}^+$ such that $|f(n)| \geq c \cdot |g(n)|$ for all $n \geq h$.

Remark 109. We note that if $f(n) \in O(g(n))$, then $g(n) \in \Omega(f(n))$. In particular, $n^3 \in \Omega(n^2)$ and $n! \in \Omega(2^n)$. However, $2^n \notin \Omega(n!)$.

We next discuss Big-Theta. Informally, $f(n) \in \Theta(g(n))$ provided that $f(n)$ and $g(n)$ grow at the same asymptotic rate. This is formalized as follows.

Definition 110. Let k be an integer. Let $f, g : \mathbb{Z}_{\geq k} \to \mathbb{R}$ be functions. We say that $f(n) \in \Theta(g(n))$ provided that $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Example 111. We have that the function $f(n) = 3n^2 + 5n + 7 \in \Theta(n^2)$. Similarly, $n! \in \Theta(2 \cdot n!)$.

Example 112. Recall that $n^2 \in O(n^3)$. However, $n^3 \notin O(n^2)$. So $n^2 \notin \Theta(n^3)$.

Remark 113. As $n^2 \in O(n^3)$, but $n^2 \notin \Omega(n^3)$, we have that in some sense, n^3 is an asymptotic *strict* upper bound for n^2. We formalize this notion as follows.

Definition 114. Let k be an integer. Let $f, g : \mathbb{Z}_{\geq k} \to \mathbb{R}$ be functions. We say that $f(n) \in o(g(n))$ (here, $o(g(n))$ is little-o, which is different than Big-O) if for every $\epsilon > 0$, there exists $h \in \mathbb{N}$ such that $|f(n)/g(n)| < \epsilon$ for all $n \geq h$. Equivocally, $f(n) \in o(g(n))$ precisely if:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Example 115. We claim that $n^2 \in o(n^3)$. To see this, we compute:

$$\lim_{n \to \infty} \frac{n^2}{n^3} = \lim_{n \to \infty} \frac{1}{n} = 0.$$

We may similarly define a strict asymptotic lower bound: little-omega.

Definition 116. Let k be an integer. Let $f, g : \mathbb{Z}_{\geq k} \to \mathbb{R}$ be functions. We say that $f(n) \in \omega(g(n))$ (here, $\omega(g(n))$ is little-omega, which is different than Big-Omega Ω) if for every $\epsilon > 0$, there exists $h \in \mathbb{N}$ such that $|f(n)/g(n)| > \epsilon$ for all $n \geq h$. Equivocally, $f(n) \in \omega(g(n))$ precisely if:

$$\lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = \infty.$$

We note that the limit may be either ∞ or $-\infty$. Taking the absolute value of the limit will be useful to do later on.

We note that little-o is a stronger relation than Big-O. Similarly, little-omega is a stronger relation than Big-Omega. We capture this with the following theorem, which intuitively states that a strict asymptotic bound implies the corresponding weak asymptotic bound.

Theorem 117. Let k be an integer. Let $f, g : \mathbb{Z}_{\geq k} \to \mathbb{R}$ be functions.

(a) If $f(n) \in o(g(n))$, then $f(n) \in O(g(n))$.

(b) If $f(n) \in \omega(g(n))$, then $f(n) \in \Omega(g(n))$.

We now turn to our key tool in comparing asymptotic expressions: the limit comparison test.

Theorem 118 (Limit Comparison Test)**.** Let k be an integer. Let $f, g : \mathbb{Z}_{\geq k} \to \mathbb{R}$ be functions. Suppose that the following limit exists:

$$L := \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right|. \tag{24}$$

- If $0 < L < \infty$, then $f(n) \in \Theta(g(n))$.
- If $L = 0$, then $f(n) \in O(g(n))$. However, $f(n) \notin \Theta(g(n))$. (That is, $f(n) \in o(g(n))$).
- If $L = \infty$, then $f(n) \in \Omega(g(n))$. However, $f(n) \notin \Theta(g(n))$. (That is, $f(n) \in \omega(g(n))$).

We illustrate using the Limit Comparison Test with the following example.

Example 119. Take $f(n) = 3n^2 + 5n + 7$, and take $g(n) = n^2$. We have that:

$$\begin{aligned}
&\lim_{n\to\infty} \frac{3n^2 + 5n + 7}{n^2} \\
&= \lim_{n\to\infty} \frac{3n^2}{n^2} + \lim_{n\to\infty} \frac{5n}{n^2} + \lim_{n\to\infty} \frac{7}{n^2} \\
&= \lim_{n\to\infty} 3 + \lim_{n\to\infty} \frac{5}{n} + \lim_{n\to\infty} \frac{7}{n^2} \\
&= 3 + 0 + 0 \\
&= 3.
\end{aligned}$$

As $0 < 3 < \infty$, $3n^2 + 5n + 7 \in \Theta(n^2)$.

In practice, $f(n)$ and $g(n)$ are often not (both) polynomials, and so the limit computations are not always as straight-forward. Other techniques are needed to evaluate (24). To this end, we recall L'Hopital's Rule and some useful convergence tests from the first year calculus sequence.
We conclude this section by summarizing some useful useful properties of our asymptotic expressions.

Theorem 120. Let k be an integer. Let $f, g, h : \mathbb{Z}_{\geq k} \to \mathbb{R}$ be functions. We have the following.

(a) $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$.

(b) $f(n) \in o(g(n))$ if and only if $g(n) \in \omega(f(n))$.

(c) Transitivity:

- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$.
- If $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$, then $f(n) \in \Omega(h(n))$.
- If $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$, then $f(n) \in \Theta(h(n))$.
- If $f(n) \in o(g(n))$ and $g(n) \in o(h(n))$, then $f(n) \in o(h(n))$.
- If $f(n) \in \omega(g(n))$ and $g(n) \in \omega(h(n))$, then $f(n) \in \omega(h(n))$.

6.1.1 L'Hopital's Rule

Frequently, when computing (24), we will obtain the indeterminate forms $\pm\frac{\infty}{\infty}$ or $\frac{0}{0}$. L'Hopital's Rule provides a means to evaluate (24), operating under certain modest assumptions.

Theorem 121 (L'Hopital's Rule)**.** Suppose $f(x)$ and $g(x)$ are differentiable functions, where either:

(a) $\lim_{x\to\infty} f(x) = 0$ and $\lim_{x\to\infty} g(x) = 0$; or

(b) $\lim_{x\to\infty} f(x) = \pm\infty$ and $\lim_{x\to\infty} g(x) = \pm\infty$.

If $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ exists, then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}.$$

We now consider an example of applying L'Hopital's Rule.

Example 122. Let $f(n) = n$, and let $g(n) = 2^n$. Consider:

$$\lim_{n\to\infty} \frac{n}{2^n},$$

which has the indeterminate form $\frac{\infty}{\infty}$. Note that both $f(n) = n$ and $g(n) = 2^n$ are differentiable. So we can apply L'Hopital's rule. We have that:

$$\lim_{n\to\infty} \frac{n}{2^n} = \lim_{n\to\infty} \frac{1}{\ln(2)\cdot 2^n} = 0.$$

As $\lim_{n\to\infty} \frac{n}{2^n} = 0$, the Limit Comparison Test provides that $f(n) = n \in O(2^n)$.

Remark 123. In certain instances, L'Hopital's Rule may need to be applied multiple times. We provide an example to illustrate.

Example 124. Let $f(n) = n^2 + 3n + 5$, and let $g(n) = 3^n$. The limit

$$\lim_{n\to\infty} \frac{n^2 + 3n + 5}{3^n}$$

has the indeterminate form $\frac{\infty}{\infty}$. Furthermore, $f(n) = n^2 + 3n + 5$ and $g(n) = 3^n$ are both differentiable. In order to apply L'Hopital's rule, we must first establish that the following limit exists:

$$\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \lim_{n\to\infty} \frac{2n + 3}{\ln(3)\cdot 3^n}. \tag{25}$$

To do this, we seek to apply L'Hopital's Rule to (25). Note that (2) has the indeterminate form $\frac{\infty}{\infty}$. Furthermore, note that $f'(n) = 2n + 3$ and $g'(n) = \ln(3)\cdot 3^n$ are differentiable functions. Now:

$$\lim_{n\to\infty} \frac{f''(n)}{g''(n)} = \lim_{n\to\infty} \frac{2}{(\ln(3))^2\cdot 3^n} = 0.$$

So by L'Hopital's Rule, we have that:

$$\lim_{n\to\infty} \frac{2n + 3}{\ln(3)\cdot 3^n} = \lim_{n\to\infty} \frac{2}{(\ln(3))^2\cdot 3^n} = 0.$$

Thus, we can apply L'Hopital's Rule to the original limit $\lim_{n\to\infty} \frac{n^2 + 3n + 5}{3^n}$, to deduce that:

$$\begin{aligned}
&\lim_{n\to\infty} \frac{n^2 + 3n + 5}{3^n} \\
&= \lim_{n\to\infty} \frac{2n + 3}{\ln(3)\cdot 3^n} \\
&= \lim_{n\to\infty} \frac{2}{(\ln(3))^2\cdot 3^n} = 0.
\end{aligned}$$

So by the Limit Comparison Test, $n^2 + 3n + 5 \in O(3^n)$.

L'Hopital's Rule is useful in situations where both $f(n)$ and $g(n)$ are differentiable, and when differentiating $f(n)$ and $g(n)$ simplifies the limit. This is often not the case. For instance, suppose that one of our functions is $n!$, which is not differentiable or even continuous. Thus, L'Hopital's Rule cannot be applied to compute (24).

Example 125. We consider another example of when L'Hopital's rule is unhelpful in applying the Limit Comparison Test. Let $f(n) = 2^n$ and $g(n) = n^n$. We consider:

$$\lim_{n\to\infty} \frac{2^n}{n^n}, . \tag{26}$$

which has the indeterminate form $\frac{\infty}{\infty}$. Now $f(n) = 2^n$ and $g(n) = n^n$ are both differentiable, with derivatives $f'(n) = \ln(2) \cdot 2^n$ and $g'(n) = (\ln(n) + 1) \cdot n^n$.[3] Now in order to apply L'Hopital's Rule, it remains to show that the following limit exists.

$$\lim_{n\to\infty} \frac{f'(n)}{g'(n)} \tag{27}$$

$$= \lim_{n\to\infty} \frac{\ln(2) \cdot 2^n}{(\ln(n) + 1) \cdot n^n} \tag{28}$$

$$= \ln(2) \cdot \lim_{n\to\infty} \frac{2^n}{(\ln(n) + 1) \cdot n^n}. \tag{29}$$

Notice that the limit at (29) is not simpler to evaluate than the original limit at (26). For this reason, L'Hopital's Rule is ineffectual in helping us to apply the Limit Comparison Test.

In order to apply the Limit Comparison Test in cases such as in Example 125 or when one of our functions is not continuous, we use series convergence tests from calculus. The two most commonly used convergence tests in an algorithms course are the Ratio Test and the Root Test, which test whether a series of the form $\sum_{n=0}^{\infty} a_n$ converges. Note that if $\sum_{n=0}^{\infty} a_n$ converges, then $\lim_{n\to\infty} a_n = 0$. This is applied in asymptotic analysis in the following manner. Let $f(n)$ and $g(n)$ be functions. Let $k \in \mathbb{N}$. If the following series

$$\sum_{n=k}^{\infty} \frac{f(n)}{g(n)}$$

converges, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

By the Limit Comparison Test, we have that $f(n) \in O(g(n))$.

[3]Recall that we use *logarithmic differentiation* to compute the derivative of n^n, and **not** the power or exponential rules.

6.1.2 Ratio Test

Theorem 126 (Ratio Test)**.** Let $k \in \mathbb{N}$. Suppose that we have the series $\sum_{n=k}^{\infty} a_n$. Suppose that the following limit exists.

$$L := \lim_{n\to\infty} \left| \frac{a_{n+1}}{a_n} \right|.$$

(a) If $L < 1$, then the series is convergent. (In particular, the series converges absolutely).

(b) If $L > 1$, then the series diverges. In particular, if $L > 1$, then the sequence $(a_n)_{n\in\mathbb{N}}$ diverges to either ∞ or $-\infty$ as well. That is, $\lim_{n\to\infty} |a_n| = \infty$.

(c) If $L = 1$, the Ratio Test is inconclusive and another convergence test is needed.

The Ratio Test works particularly well when functions have terms dealing with factorials or exponentials. In contrast, the Ratio Test is less helpful when dealing with polynomials. We provide some examples illustrating the Ratio Test.

Example 127. Let $f(n) = 2^n$ and $g(n) = n!$. Recall that $g(n) = n!$ is not even continuous, let alone differentiable. So we cannot use L'Hopital's Rule when applying the Limit Comparison Test. Rather, we appeal to the Ratio Test. Consider the following series:

$$\sum_{n=0}^{\infty} \frac{2^n}{n!}.$$

Here, $a_n = \dfrac{2^n}{n!}$. Now consider:

$$\begin{aligned} L &:= \lim_{n\to\infty} \frac{a_{n+1}}{a_n} \\ &= \lim_{n\to\infty} \frac{2^{n+1} n!}{2^n (n+1)!} \\ &= \lim_{n\to\infty} \frac{2}{n+1} = 0. \end{aligned}$$

As $L = 0$, the Ratio Test tells us that our series converges. Thus,

$$\lim_{n\to\infty} \frac{2^n}{n!} = 0.$$

So $2^n \in O(n!)$, as desired.

Example 128. Let $f(n) = n!$ and $g(n) = 5^n$. We apply the Ratio Test. Consider the following series:

$$\sum_{n=0}^{\infty} \frac{n!}{5^n}.$$

Here, $a_n = \dfrac{n!}{5^n}$. Now consider:

$$\begin{aligned} L &:= \lim_{n\to\infty} \frac{a_{n+1}}{a_n} \\ &= \lim_{n\to\infty} \frac{(n+1)! \cdot 5^n}{n! \cdot 5^{n+1}} \\ &= \lim_{n\to\infty} \frac{n+1}{5} = \infty. \end{aligned}$$

So as $L = \infty$, the Ratio Test tells us that not only does our series diverge, but that:

$$\lim_{n\to\infty} \frac{n!}{5^n} = \infty,$$

as well. So by the Limit Comparison Test, we have that $n! \in \Omega(5^n)$.

Remark 129. Let $f(n), g(n)$ be functions, and let $a_n = f(n)/g(n)$. Note that the ratio limit

$$L_1 := \lim_{n\to\infty} \left|\frac{a_{n+1}}{a_n}\right| = \lim_{n\to\infty} \left|\frac{f(n+1)}{g(n+1)} \cdot \frac{g(n)}{f(n)}\right|$$

is in general **not** the same as:

$$L_2 := \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right|.$$

The Ratio Test tells us that if $0 < L_1 < 1$, then $L_2 = 0$. To see that L_1 need to be equal to L_2, take $f(n) = 2^n$ and $g(n) = 3^n$. So $a_n = 2^n/3^n$. Observe that:

$$L_1 = \lim_{n\to\infty} \frac{2^{n+1}}{3^{n+1}} \cdot \frac{3^n}{2^n} = \lim_{n\to\infty} \frac{2}{3} = \frac{2}{3}.$$

On the other hand,

$$L_2 = \lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0.$$

6.1.3 Root Test

We conclude with a discussion of the Root Test.

Theorem 130 (Root Test)**.** Let $k \in \mathbb{N}$. Suppose that we have the series $\sum_{n=k}^{\infty} a_n$. Suppose that the following limit exists.

$$L := \lim_{n\to\infty} |a_n|^{1/n}.$$

(a) If $L < 1$, then the series is convergent. (In particular, the series converges absolutely).

(b) If $L > 1$, then the series diverges. In particular, if $L > 1$, then the sequence $(a_n)_{n\in\mathbb{N}}$ diverges to either ∞ or $-\infty$ as well. That is, $\lim_{n\to\infty} |a_n| = \infty$.

(c) If $L = 1$, the Root Test is inconclusive and another convergence test is needed.

Example 131. Take $f(n) = 2^n$ and $g(n) = n^n$. Recall from Example 125 that L'Hopital's Rule is insufficient to help us in applying the Limit Comparison Test. Rather, we appeal to the Root Test. Consider the following series:

$$\sum_{n=1}^{\infty} \frac{2^n}{n^n}.$$

So $a_n = \dfrac{2^n}{n^n}$. Now consider the following limit.

$$\begin{aligned} L &:= \lim_{n\to\infty} |a_n|^{1/n} \\ &= \lim_{n\to\infty} \left(\frac{2^n}{n^n}\right)^{1/n} \\ &= \lim_{n\to\infty} \frac{2}{n} = 0. \end{aligned}$$

As $L = 0$, our series converges. Thus, $\lim_{n\to\infty} \dfrac{2^n}{n^n} = 0$. So by the Limit Comparison Test, $2^n \in O(n^n)$.

Remark 132. Let $f(n), g(n)$ be functions, and let $a_n = f(n)/g(n)$. Note that the root limit

$$L_1 := \lim_{n\to\infty} |\sqrt[n]{a_n}|$$

is in general **not** the same as:

$$L_2 := \lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

The Root Test tells us that if $0 < L_1 < 1$, then $L_2 = 0$. To see that L_1 need to be equal to L_2, take $f(n) = 2^n$ and $g(n) = 3^n$. So $a_n = 2^n/3^n$. Observe that:

$$L_1 = \lim_{n\to\infty} \left|\left(\frac{2^n}{3^n}\right)^{1/n}\right| = \lim_{n\to\infty} \frac{2}{3} = \frac{2}{3}.$$

On the other hand,

$$L_2 = \lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0.$$

6.2 Analyzing Iterative Code

In this section, we look at analyzing the runtime complexity of iterative algorithms. Our exposition is adapted from [McQ09].

Our goal in analyzing an algorithm A is to construct a function $T(n)$, such that if A is given an input of size n, then A takes at most $T(n)$ steps. While it is easy to construct a function $T(n)$ that (grossly) overestimates the actual runtime of A, our goal is that $T(n)$ should represent the actual runtime of A as accurately as possible. Precisely, if the actual runtime of A is $\Theta(f(n))$, then we want to construct $T(n)$ such that $T(n) \in \Theta(f(n))$.

In order to rigorously and carefully analyze our algorithms, it is necessary to make underlying assumptions about the model of computation. We assume an arbitrary time unit, which is the cost of a single computation step.

(a) The following operations take 1 time step:

- Single arithmetic, Boolean, and assignment operations.
- Single numeric or Boolean comparisons.
- Single Input and output statements.
- A `return` statement in a function.
- Array index operations.

(b) Selection statements like `if...else` and `switch` blocks take the sum of the following steps.

- The time to evaluate the Boolean condition.
- The maximum running time of each case.

(c) Loop execution time is computed as follows.

- Time for the loop initialization, plus
- The sum over each iteration of the total time it takes for each of the following: the condition check, the update operations, and the body of the loop.

We always assume that the loop executes the maximum number of times possible.

(d) The execution time for a function is computed as follows.

- 1 unit of time for invoking the function, plus
- The cost of any parameter computations, plus
- The time for the execution of the body of the function.

We now turn to consider an example.

6.2.1 Analyzing Code: Example 1

Example 133. Consider the Linear Search algorithm. Note that $A[1, \ldots, n]$ indicates an array of n elements, where the indexing scheme starts at 1 (rather than at 0).

Algorithm 9 Linear Search

```
1: procedure LinearSearch(Array A[1, ..., n], key)
2:     found ← false
3:     for i ← 1; i ≤ len(A); i ← i + 1 do
4:         if A[i] == key then
5:             found ← true
       return found
```

We now construct the runtime function $T(n)$ to model the Linear Search algorithm.

- At line 2, we have a single assignment statement. So line 2 takes 1 unit of time.
- Initializing the loop at line 3 takes a single assignment. So the loop initialization takes 1 unit of time.
- The loop itself takes n iterations. At each iteration we have:
  - The comparison that $i \leq \text{len}(A)$, which takes 1 step.
  - The variable update $i \leftarrow i + 1$, which takes 2 steps: one step for the $i + 1$ computation and one step for the assignment.
  - In the body of the loop, we have the comparison at line 4, which takes 2 steps: one for the array access and one for the comparison.
  - At line 5, we have a single assignment, which takes 1 step.

  Thus, the loop at line 3 has time complexity:

$$\sum_{i=1}^{n}(1 + 2 + 2 + 1) = \sum_{i=1}^{n} 6 = 6n.$$

- Finally, we have the return statement after line 5, which takes 1 step.

As we have an iterative algorithm, we add up the costs to obtain:

$$\begin{aligned} T(n) &= 1 + 1 + 6n + 1 \\ &= 6n + 3. \end{aligned}$$

Thus, $T(n) \in \Theta(n)$.

6.2.2 Analyzing Code: Example 2

Example 134. We now consider an example of analyzing nested loops where the loop variables do **not** depend on each other.

Algorithm 10 Nested Independent Loops

```
1: procedure Foo1(Integer n)
2:     for i ← 1; i ≤ n; i ← i + 1 do
3:         for j ← 1; j ≤ n; j ← j + 2 do
4:             print "Foo"
```

When analyzing nested loops, it is often easier to start by analyzing the inner-most loop first and work outwards. To this end, we begin with analyzing the j-loop.

- At line 3, the j-loop takes 1 step to initialize $j \leftarrow 1$.
- The loop terminates when $1 + 2k > n$. Solving for k, we obtain that: $k > \lceil (n-1)/2 \rceil$. As the loop takes at least one iteration to compare i to n, we have that the loop takes $\lceil (n-1)/2 \rceil + 1$ iterations.
- At each iteration, the loop does the following:
  - The comparison $j \leq n$ takes 1 step.
  - The update $j \leftarrow j + 2$ takes 2 steps: one step to evaluate $j + 2$ and one step for the assignment.
  - The body of the loop consists of a single **print** statement, which takes 1 step.

  So the runtime complexity of the j-loop is:

$$\begin{aligned} 1 + \sum_{j=1}^{\lceil (n-1)/2 \rceil + 1} (1 + 2 + 1) &= 1 + \sum_{j=1}^{\lceil (n-1)/2 \rceil + 1} 4 \\ &= 1 + 4\left(\lceil (n-1)/2 \rceil + 1\right) \\ &= 5 + 4\left(\lceil (n-1)/2 \rceil\right). \end{aligned}$$

We now analyze the outer loop.

- Initializing the outer loop takes 1 step.
- The outer loop runs for n iterations. At each iteration, the loop does the following.
  - The comparison $i \leq n$ takes 1 step.
  - The update $i \leftarrow i + 1$ takes 2 steps: one step to evaluate $i + 1$ and one step for the assignment.
  - The body the i-loop consists solely of the j-loop.

  So the runtime complexity function is:

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n} \left(1 + 2 + 4 + 4\left(\lceil (n-1)/2 \rceil\right)\right) \\ &= 1 + \sum_{i=1}^{n} \left(8 + 4\left(\lceil (n-1)/2 \rceil\right)\right) \\ &= 1 + \sum_{i=1}^{n} 8 + \sum_{i=1}^{n} 4\left(\lceil (n-1)/2 \rceil\right) \\ &= 1 + 8n + 4n \cdot \left(\lceil (n-1)/2 \rceil\right). \end{aligned}$$

  So $T(n) \in \Theta(n^2)$.

6.2.3 Analyzing Code: Example 3

Example 135. We consider an example of analyzing nested loops where the loop variables depend on each other.

Algorithm 11 Nested Dependent Loops

```
1: procedure Foo2(Integer n)
2:     for i ← 1; i ≤ n; i ← i + 1 do
3:         for j ← 1; j ≤ i; j ← j + 1 do
4:             print "Foo"
```

We again start by analyzing the inner loop. For the purposes of analyzing the j-loop, we think of i as being fixed. The j-loop does the following.

- The initialization of the loop takes 1 step.
- Observe that the j-loop takes i iterations. At each iteration, the loop does the following.
  - The comparison $j \leq i$ takes 1 step.
  - The update $j \leftarrow j + 1$ takes 2 steps: one step to evaluate $j + 1$ and one step for the assignment.
  - The body of the loop consists of a single **print** statement, which takes 1 step.

  So the runtime complexity of the j-loop is:

$$\begin{aligned} 1 + \sum_{j=1}^{i} (1 + 2 + 1) &= 1 + \sum_{j=1}^{i} 4 \\ &= 1 + 4i. \end{aligned}$$

We now turn to analyzing the outer i-loop.

- Initializing the loop takes 1 step.
- Observe that the i loop takes n iterations. At each iteration, the loop does the following.
  - The comparison $i \leq n$ takes 1 step.
  - The update $i \leftarrow i + 1$ takes 2 steps: one step to evaluate $i + 1$ and one step for the assignment.
  - The body the i-loop consists solely of the j-loop.

  So the runtime complexity function $T(n)$ is:

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n} (1 + 2 + (1 + 4i)) \\ &= 1 + \sum_{i=1}^{n} (4 + 4i) \\ &= 1 + \sum_{i=1}^{n} 4 + 4 \cdot \sum_{i=1}^{n} i \\ &= 1 + 4n + 4 \cdot \frac{n(n+1)}{2}. \end{aligned}$$

  Thus, $T(n) \in \Theta(n^2)$.

6.3 Analyzing Recursive Code: Mergesort

Mergesort is a recursive algorithm, which partitions the array in half. The algorithm then sorts each half, after which it merges the sorted arrays together. Our goals for this section are to (i) understand how Mergesort works, and (ii) to write down a recursive expression for the runtime complexity function $T(n)$ for Mergesort. Our techniques for analyzing iterative code are not sufficient for analyzing recursive algorithms, which motivates the need for the unrolling and tree methods to solve recurrences. We will examine these new methods later.

We begin by introducing Megresort. We decompose the algorithm to identify the base cases and the recursive calls.

- **Base Cases:** There are two base cases.
  - **Case 1:** Suppose that the input array has at most 1 element. In this case, the array is already sorted. So there is nothing more to be done, and the algorithm returns.
  - **Case 2:** Suppose the input array has at 2 elements. If the array is sorted, then there is nothing more to be done. So the algorithm returns. If instead the elements are out of order, then the algorithm swaps them before returning.
- **Recursive Case:** The algorithm first creates two new arrays, named `left` and `right`. Denote n as the length of the array. The `left` array stores the first $n/2$ elements, while the `right` array stores the last $n/2$ elements. The algorithm recursively sorts `left` first. Afterwards, the algorithm then recursively sorts `right`. After `left` and `right` are sorted, then the algorithm merges them back into the original array. In the merge step, care is taken to ensure the elements are inserted in order, so that the array is sorted when the method terminates.

We provide the following pseudocode.

Algorithm 12 Mergesort

```
1:  procedure MERGESORT(Array A[1, ..., n])
2:      if len(A) ≤ 1 then return A
3:      Left ← []
4:      Right ← []
5:
6:      for i ← 1; i ≤ len(A)/2; i ← i + 1 do
7:          Left[i] ← A[i]
8:
9:      for i ← len(A)/2 + 1, j ← 1; i ≤ len(A); i ← i + 1, j ← j + 1 do
10:         Right[j] ← A[i]
11:
12:     Mergesort(Left)
13:     Mergesort(Right)
14:     Merge(Left, Right, A)
```

For completeness, we also provide an implementation of the `Merge` algorithm, which takes as input three arrays: L, R, and A. Provided L and R are sorted, then the `Merge` algorithm successfully merges L and R into A, such that A is sorted.

Algorithm 13 Merge

```
1: procedure MERGE(Array L[1,...,k], Array R[1,...,ℓ], Array A[1,...,n])
2:     LIndex ← 1
3:     RIndex ← 1
4:     AIndex ← 1

5:     while LIndex ≤ k AND RIndex ≤ ℓ do
6:         if L[LIndex] ≤ R[RIndex] then
7:             A[AIndex] ← L[LIndex]
8:             LIndex ← LIndex + 1
9:         else
10:            A[AIndex] ← R[RIndex]
11:            RIndex ← RIndex + 1
12:        AIndex ← AIndex + 1
13:
14:    while LIndex ≤ k do
15:        A[AIndex] ← L[LIndex]
16:        LIndex ← LIndex + 1
17:        AIndex ← AIndex + 1
18:
19:    while RIndex ≤ ℓ do
20:        A[AIndex] ← R[RIndex]
21:        RIndex ← RIndex + 1
22:        AIndex ← AIndex + 1
```

6.3.1 Mergesort: Example

Example 136. We work through an example illustrating mergesort. Suppose we wish to sort `[2,3,1,5,7,6,4]` using mergesort.

- The algorithm begins by partitioning the array into two arrays, `left = [2, 3, 1, 5]` and `right = [7, 6, 4]`. Next, the algorithm recursively sorts the `left` array.
  - **Recursive Call:** `Mergesort(left)`. The algorithm partitions `[2, 3, 1, 5]` into two subarrays: `[2, 3]` and `[1, 5]`. The algorithm then recursively sorts `[2, 3]` and `[1, 5]`.
    - **Recursive Call:** `Mergesort([2, 3])`. As `[2, 3]` has length 2 and is sorted, the algorithm returns.
    - **Recursive Call:** `Mergesort([1, 5])`. As `[1, 5]` has length 2 and is sorted, the algorithm returns.

    Next, the algorithm merges `[2, 3]` and `[1, 5]`, leaving the array `[1, 2, 3, 5]`.
  - **Recursive Call:** `Mergesort(right)`. The algorithm partitions `[7, 6, 4]` into two subarrays: `[7, 6]` and `[4]`.
    - **Recursive Call:** `Mergesort([7, 6])`. As `[7, 6]` has length 2 and is not sorted, the algorithm swaps the elements, leaving: `[6, 7]`.
    - **Recursive Call:** `Mergesort([4])`. As `[4]` has length 1, the algorithm returns.

    The algorithm merges `[6, 7]` and `[4]`, leaving the array `[4, 6, 7]`.

  Finally, the algorithm merges `[1, 2, 3, 5]` and `[4, 6, 7]`, leaving `[1, 2 , 3, 4, 5, 6, 7]` as the final sorted array. The algorithm now returns.

6.3.2 Mergesort: Runtime Complexity Function

We now turn to constructing the runtime complexity function $T(n)$ for Mergesort. For the purposes of this section, we will not carefully analyze the Merge function. Instead, we will simply use the fact that the Merge function runs in time $\Theta(n)$.

The complexity of Mergesort is as follows.

- Line 3 and Line 4 **each** take 1 step, for the assignments.
- The loop at line 6 takes 1 step for the initialization.
- The loop at line 6 runs for $\lceil n/2 \rceil$ iterations. Now the body of the loop proceeds as follows.
  - The comparison $i \leq \text{len}(A)/2$ takes 2 steps: one step for computing $\text{len}(A)/2$ and one step for the comparison $i \leq \text{len}(A)/2$.
  - The update statement takes 2 steps: one step for computing $i + 1$ and one statement for the assignment.
  - The body consists of two array accesses and one assignment, for a total of 3 steps.

  Thus, the loop at Line 6 has time complexity:

$$\begin{aligned} 1 + \sum_{i=1}^{\lceil n/2 \rceil} (2 + 2 + 3) &= 1 + \sum_{i=1}^{\lceil n/2 \rceil} 7 \\ &= 1 + 7 \cdot \lceil n/2 \rceil. \end{aligned}$$

- We now consider the loop at Line 9. We look first at the initialization, which takes 4 steps in total.
  - The command $i \leftarrow \text{len}(A)/2+1$ takes 3 steps: one to compute $\text{len}(A)/2$, one to compute $\text{len}(A)/2+1$, and a third for the assignment.
  - The command $j \leftarrow 1$ takes a single step.
- The loop at Line 9 takes $\lfloor n/2 \rfloor$ steps. At each iteration, the loop proceeds as follows.
  - The comparison $i \leq \text{len}(A)$ takes 1 step.
  - The update statement takes 4 steps: computing $i + 1$ and $j + 1$ each take one step, and then we have two assignments.
  - The body of the loop consists of two array accesses and one assignment, for a total of 3 steps.

  Thus, the loop at Line 9 has runtime complexity:

$$\begin{aligned} 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} (1 + 4 + 3) &= 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} 8 \\ &= 1 + 8 \cdot \lfloor n/2 \rfloor. \end{aligned}$$

- We have our two recursive calls to Mergesort, which take $T(n/2)$ steps each.
- The call to Merge takes time $\Theta(n)$.

Thus, for $n > 1$, the complexity of Mergesort is:

$$\begin{aligned} T(n) &= 1 + 1 + 1 + (1 + 7 \cdot \lceil n/2 \rceil) + (1 + 8 \cdot \lfloor n/2 \rfloor) + 2T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n). \end{aligned}$$

Now if $n \leq 1$, then $T(n) = \Theta(1)$. So the runtime complexity function is:

$$T(n) = \begin{cases} \Theta(1) & : n \leq 1, \\ 2T(n/2) + \Theta(n) & : n > 1. \end{cases}$$

We do not know how to find a closed-form asymptotic expression for $T(n)$. This motivates the need for new techniques, such as the unrolling and tree methods, which we will examine later.

6.4 Analyzing Recursive Code: More Examples

In this section, we examine additional examples of determining the runtime complexity functions for recursive algorithms. With Mergesort, the key approach was as follows: (i) determine the cost of the non-recursive work, (ii) determine the number of recursive calls, and (iii) determine the size of the input for each recursive call. We will apply this technique below.

Example 137. Consider the following algorithm. There are three recursive calls, each of which has size $n/4$. The non-recursive part of the function takes time $\Theta(n)$. The runtime complexity function is thus:

$$T(n) = \begin{cases} \Theta(1) : n \leq 0, \\ 3T(n/4) + \Theta(n) : n > 0. \end{cases}$$

Algorithm 14 Recursive Example 1

```
1: procedure Foo3(Integer n)
2:     if n ≤ 0 then return
3:     Foo3(n/4)
4:     Foo3(n/4)
5:     Foo3(n/4)
6:     for i ← 1; i ≤ n; i ← i + 1 do
7:         print "Foo"
```

Example 138. We consider a second example. Here, there are two recursive calls, each of which takes input $n - 3$. The non-recursive part of the code takes time $\Theta(n)$. The runtime complexity function is thus:

$$T(n) = \begin{cases} \Theta(1) : n \leq 0, \\ 2T(n-3) + \Theta(n) : n > 0. \end{cases}$$

Algorithm 15 Recursive Example 2

```
1: procedure Foo4(Integer n)
2:     if n ≤ 0 then return
3:     Foo4(n − 3)
4:     Foo4(n − 3)
5:     for i ← 1; i ≤ n; i ← i + 1 do
6:         print "Foo"
```

6.5 Unrolling Method

The Unrolling Method is particularly useful for finding closed-form asymptotic expressions for recurrences of the form:

$$T(n) = \begin{cases} c & : n \leq k, \\ aT(n-b) + f(n) & : n > k. \end{cases}$$

That is, on recursive calls, the size of the input decreases additively rather than multiplicatively. The key idea is that we apply the recurrence iteratively until we hit a base case, and add up the non-recursive work as we go.

Example 139. Consider the recrurrence:

$$T(n) = \begin{cases} 1 & : n \leq 1, \\ T(n-2) + \Theta(n) & : n > 1. \end{cases}$$

Whenever we have an asymptotic expression for the non-recursive work, such as $\Theta(n)$, it is best to replace it with the corresponding term cn, where c is an arbitrary constant. This serves to reduce careless mistakes when working problems.

Now the key idea behind unrolling is that we wish to compute:

$$\begin{aligned} T(n) &= T(n-2) + cn \\ &= T(n-2) + c(n-2) + cn \\ &= T(n-4) + c(n-4) + c(n-2) + cn \\ &\vdots \\ &= T(n - 2 \cdot \lceil (n-1)/2 \rceil) + 2c + 4c + \cdots + c(n-4) + c(n-2) + cn. \end{aligned}$$

Thus:

$$\begin{aligned} T(n) &= 1 + c \cdot \sum_{i=0}^{\lceil (n-1)/2 \rceil} (n - 2i) \\ &= 1 + c \cdot \sum_{i=0}^{\lceil (n-1)/2 \rceil} n - 2c \cdot \sum_{i=0}^{\lceil (n-1)/2 \rceil} i \\ &= 1 + nc \cdot \lceil (n-1)/2 \rceil - 2c \cdot \frac{\lceil (n-1)/2 \rceil \cdot (\lceil (n-1)/2 \rceil + 1)}{2} \\ &= 1 + nc \cdot \lceil (n-1)/2 \rceil - c \cdot [\lceil (n-1)/2 \rceil \cdot (\lceil (n-1)/2 \rceil + 1)]. \end{aligned}$$

Thus, $T(n) \in \Theta(n^2)$.

Remark 140. For some, eyeballing the expression below and coming up with a series may be very natural. However, this approach can also be error-prone.

$$T(n - 2 \cdot \lceil (n-1)/2 \rceil) + 2c + 4c + \cdots + c(n-4) + c(n-2) + cn$$

We propose the following strategy:

- Identify the number of times we need to unroll.
- Identify the non-recursive work each time we unroll.
- Add up the non-recursive work using the previous two steps to construct a series.

So in Example 139, we could have more systematically approached things as follows.

1. We subtract 2 from the size of the input each time we unroll. So we hit a base case when:

$$n - 2k \leq 1.$$

Solving for k, we obtain:

$$k \geq (n-1)/2.$$

So $k = \lceil (n-1)/2 \rceil$.

2. When we substitute $T(n-2i)$ using the recurrence, we obtain the term $T(n-2(i+1)) + c(n-2i)$. Therefore, the non-recursive work is $c(n-2i)$ (where $i \geq 0$ is an integer).

3. It follows that:

$$\begin{aligned} T(n) &= (\text{cost of base case}) \cdot (\text{number of times the base case is reached}) + c \cdot \sum_{i=0}^{\lceil (n-1)/2 \rceil} (n-2i) \\ &= 1 \cdot 1 + c \cdot \sum_{i=0}^{\lceil (n-1)/2 \rceil} n - 2c \cdot \sum_{i=0}^{\lceil (n-1)/2 \rceil} i \\ &= 1 + nc \cdot (\lceil (n-1)/2 \rceil + 1) - 2c \cdot \frac{\lceil (n-1)/2 \rceil \cdot (\lceil (n-1)/2 \rceil + 1)}{2} \\ &= 1 + nc \cdot (\lceil (n-1)/2 \rceil + 1) - c \cdot [\lceil (n-1)/2 \rceil \cdot (\lceil (n-1)/2 \rceil + 1)]. \end{aligned}$$

Thus, we have that $T(n) \in \Theta(n^2)$.

Example 141. We consider a second example. Let:

$$T(n) = \begin{cases} 1 & : n < 2, \\ 2T(n-3) + 5 & : n \geq 2. \end{cases}$$

We proceed as follows.

1. We first determine the number of times that we need to unroll. We hit a base case when $n - 3k < 2$. Solving for k, we obtain that:

$$k > (n-2)/3.$$

So we have to unroll until $k = \lceil (n-2)/3 \rceil$.

2. When we unroll $T(n-3i)$, we obtain $2T(n-3(i+1)) + 5$. Note that when we unroll $2T(n-3(i+1))$, we obtain $2 \cdot [2T(n-3(i+2))) + 5]$. So:

$$\begin{aligned} T(n-3i) &= 2 \cdot [2T(n-3(i+2))) + 5] + 5 \\ &= 2^2 T(n-3(i+2))) + 2 \cdot 5 + 2^0 \cdot 5. \end{aligned}$$

We intentionally left the coefficients as 2^2, 2, and 2^0 respectively, to emphasize a pattern.

3. Thus:

$$\begin{aligned} T(n) &= (\text{cost of base case}) \cdot (\text{number of times the base case is reached}) + \sum_{i=0}^{\lceil (n-2)/3 \rceil} 2^i \cdot 5 \\ &= 1 \cdot 2^{\lceil (n-2)/3 \rceil} + 5 \cdot \sum_{i=0}^{\lceil (n-2)/3 \rceil} 2^i \\ &= 2^{\lceil (n-2)/3 \rceil} + 5 \cdot \frac{2^{\lceil (n-2)/3 \rceil + 1} - 1}{2 - 1} \\ &= 2^{\lceil (n-2)/3 \rceil} + 5 \cdot (2^{\lceil (n-2)/3 \rceil + 1}). \end{aligned}$$

Taking the high-order term, we obtain that $T(n) \in \Theta(2^{\lceil (n-2)/3 \rceil + 1}) = \Theta(2^{n/3})$.

6.6 Tree Method

The tree method is particularly useful for analyzing recurrences of the form $T(n) = aT(n/b)+f(n)$. Intuitively, we may use a tree structure to model the recursive calls of the method. A recurrence of the form $T(n) = aT(n/b) + f(n)$ indicates that a function makes a recursive calls, where each such call has input of size n/b. The non-recursive work at the initial call is $f(n)$. The key idea is to identify the non-recursive work at each level of the tree, and then add up said work.

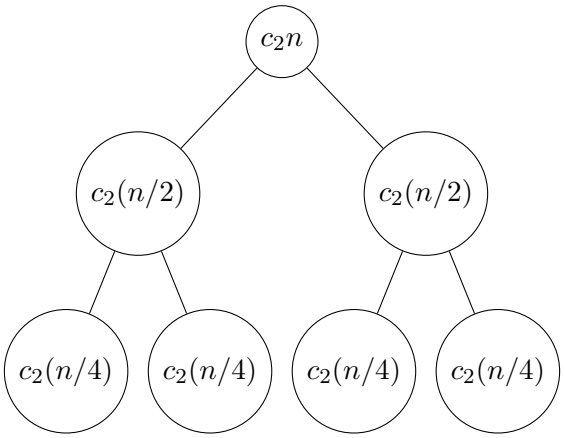
Example 142. Recall the recurrence for Mergesort:

$$T(n) = \begin{cases} \Theta(1) & : n \leq 10, \\ 2T(n/2) + \Theta(n) & : n > 1. \end{cases}$$

The first thing we do is replace $\Theta(1)$ with a constant c_1 and $\Theta(n)$ with $c_2 n$. We do this for clarity, as well as to remove a common place for errors and misconceptions to occur. So effectively, we now have the recurrence:

$$T(n) = \begin{cases} c_1 & : n \leq 1, \\ 2T(n/2) + c_2 n & : n > 1. \end{cases}$$

We use a tree diagram to help us spot the pattern. Note that the tree extends much further than the initial few levels, but the initial few levels are usually enough to helps us spot a pattern.



Observe that at level i of the tree, the non-recursive work associated with the given method call (node on the tree) is $c_2 n/2^i$. Furthermore, there are 2^i nodes at level i. So the total non-recursive work at level i is:

$$c_2(n/2^i) \cdot 2^i = c_2 n.$$

We now determine the number of levels of the tree. Let k denote the number of levels of the tree. We will solve for k. The leaf nodes correspond to base cases. So we hit a base case when: $n/2^k \leq 1$; or equivocally, when $n \leq 2^k$. Thus, $k \leq \log_2(n)$. As the number of levels of the tree is an integer, we take:

$$k = \lceil \log_2(n) \rceil.$$

Thus, the total work performed by Mergesort is:

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{\lceil \log_2(n) \rceil} c_2 n \\ &= c_2 n \cdot \lceil \log_2(n) \rceil. \end{aligned}$$

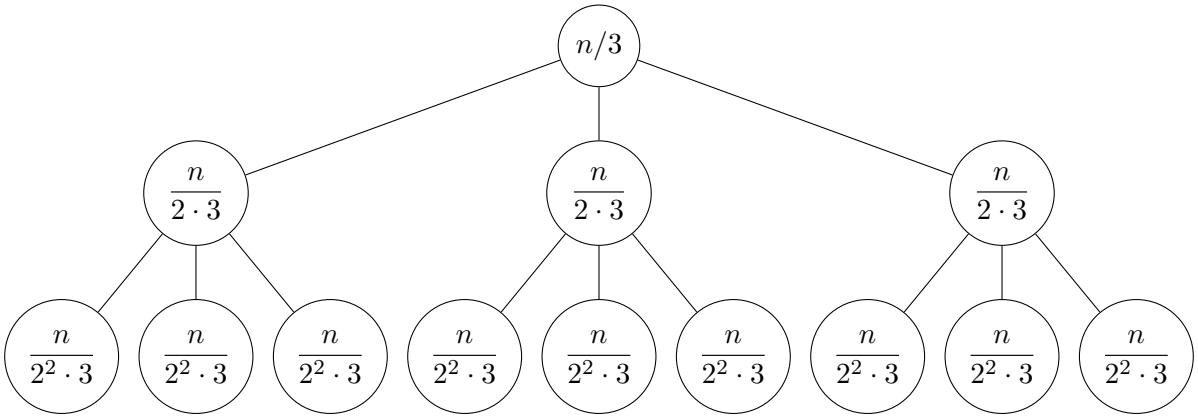
Thus, $T(n) \in O(n \log(n))$. We will show on homework that this bound is actually tight, and so $T(n) \in \Theta(n \log(n))$.

Remark 143. The tree diagram may be useful in spotting patterns, but it is not required on homework. The key pieces of information include the cost of the non-recursive work at each call (i.e., at each node), as well as the number of levels in the tree. Using these pieces of information, we can determine the asymptotic runtime complexity.

Example 144. Consider the recurrence:

$$T(n) = \begin{cases} 1 & : n < 2, \\ 3T(n/2) + n/3 & : n \geq 2. \end{cases}$$

We wish to determine a tight asymptotic bound on $T(n)$. To this end, we employ the tree method. We identify the non-recursive work at each level, as well as the number of levels. To help us identify the non-recursive work at each level, we use a tree diagram. Of course, the tree proceeds for many more levels, but we draw only the first few to help us spot the pattern.



Observe that at level i, the non-recursive work identified in a given node is $n/(2^i \cdot 3)$. We intentionally leave $2^i \cdot 3$ factored, as this will be useful for evaluating a geometric series that we will see later. To be more suggestive that $n/3$ is the constant for our geometric series, we write this as:

$$\frac{n}{3} \cdot \frac{1}{2^i}.$$

Now at level i, there are 3^i nodes. So the total non-recursive work done at level i of the tree is:

$$\frac{n}{3} \cdot \frac{3^i}{2^i}.$$

We now determine the number of levels of the tree. We hit a base case when $n/2^k < 2$; or equivocally, when $n < 2^{k+1}$. So $\log_2(n) < k + 1$, or $k > \log_2(n) - 1$. Thus, the tree has $k = \lceil \log_2(n) - 1 \rceil$ levels.

Thus:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lceil \log_2(n)-1 \rceil} \frac{n}{3} \cdot \left(\frac{3}{2}\right)^i \\ &= \frac{n}{3} \sum_{i=0}^{\lceil \log_2(n)-1 \rceil} \cdot \left(\frac{3}{2}\right)^i \\ &= \frac{n}{3} \left[\frac{1 - (3/2)^{\lceil \log_2(n)-1 \rceil + 1}}{1 - (3/2)} \right] \\ &= \frac{n}{3} \cdot 2 \cdot \left[(3/2)^{\lceil \log_2(n)-1 \rceil + 1} \right] \\ &= \frac{2n}{3} \cdot \left[(3/2)^{\lceil \log_2(n) \rceil} \right] \\ &= \frac{2n}{3} \cdot \left[(3/2)^{\lceil \log_{3/2}(n)/\log_{3/2}(2) \rceil} \right] \\ &= \frac{2n}{3} \cdot \left[n^{\lceil 1/\log_{3/2}(2) \rceil} \right] \end{aligned}$$

So $T(n) \in \Theta\left(n^{1+\lceil 1/\log_{3/2}(2) \rceil}\right)$.

7 Divide and Conquer

In this section, we introduce the Divide and Conquer strategy. The key approach is to divide a problem up into smaller, disjoint instances of the same problem. We then solve the smaller problems and use those solutions to obtain a solution for our original instance. As a result, Divide and Conquer approaches are often amenable to parallelization. In particular, the Divide and Conquer strategy arises naturally in designing (Boolean) circuits, which constitute the standard model for parallel computation. Both parallel and distributed computing, as well as circuit complexity, extend beyond the scope of the course, and we will not discuss them further here.

We have already seen an example of the Divide and Conquer paradigm with Mergesort. Here, the key strategy was to divide the array into two sub-arrays. We then sorted the sub-arrays and merged those into a single, sorted array. We now consider a second divide and conquer sorting algorithm: Quicksort. While Quicksort does not achieve the same worst-case runtime complexity of $\Theta(n\log(n))$ that Mergesort achieves, Quicksort is much faster in practice than Mergesort. Additionally, Quicksort has an average runtime of $\Theta(n\log(n))$.

7.1 Deterministic Quicksort

The key idea behind Quicksort is to select an element x of our array and rearrange the elements so that the elements smaller than x are to the left of x, and the elements that are larger than x are to the right of x. We then recursively sort the left and right sub-arrays. We begin by introducing the partition subroutine.

Algorithm 16 Partition

```
1: procedure PARITION(Array A[1,...,n], Integer start, Integer end)
2:     i ← start
3:     for j ← start; j ≤ end − 1; j ← j + 1 do
4:         if A[j] ≤ A[end] then
5:             Swap(A, i, j)
6:             i ← i + 1
7:     Swap(A, i, e) return i
```

Example 145. Suppose we wish to `Partition`$([2,6,5,1,4,3],1,6)$. So here, `start` $= 1$ and `end` $= 6$. We proceed as follows.

(a) We set $i \leftarrow 1$ and $j \leftarrow 1$. Here, $A[j] = 2$. As $A[j] \leq A[\text{end}] = 3$, we swap $A[i]$ and $A[j]$. As $i = j = 1$, the state of the array is not modified. We now update $i \leftarrow i + 1$, as per line 6 of the algorithm. We also update $j \leftarrow j + 1$ at the end of the first iteration of the loop.

(b) Now here, we have $i = j = 2$ and $A[j] = 6$. As $A[j] \not\leq A[\text{end}] = 3$, the if statement on line 4 is not executed. So we set $j \leftarrow j + 1$ and proceed to the next iteration of the loop.

(c) Now here, we have $i = 2, j = 3$, and $A[j] = 5$. As $A[j] \not\leq A[\text{end}] = 3$, the if statement on line 4 is not executed. So we set $j \leftarrow j + 1$ and proceed to the next iteration of the loop.

(d) Now here, we have $i = 2, j = 4$, and $A[j] = 1$. As $A[j] \leq A[\text{end}] = 3$, we swap $A[i]$ and $A[j]$. So $A = [2,1,5,6,4,3]$. We now increment $i \leftarrow i+1$, as per line 6 of the algorithm. We also update $j \leftarrow j+1$ at the end of the first iteration of the loop.

(e) Now here, we have $i = 3, j = 5$, and $A[j] = 4$. As $A[j] \not\leq A[\text{end}] = 3$, the if statement on line 4 is not executed. So we set $j \leftarrow j + 1$. Now $j = 6$, so the loop terminates.

(f) At our last step, we have we have $i = 3, j = 6$, and $A[j] = 3$. As $A[j] = A[\text{end}]$, we swap $A[i]$ and $A[j]$. So $A = [2,1,3,6,4,5]$. The algorithm terminates, returning $i = 3$.

Example 146. Suppose we wish to `Partition`$([1,4,3,6,5,2],1,6)$. So here, `start` $= 1$ and `end` $= 6$. We proceed as follows.

- We set $i \leftarrow 1$ and $j \leftarrow 1$. Here, $A[j] = 1$. As $A[j] \leq A[\text{end}] = 2$, we swap $A[i]$ and $A[j]$. As $i = j = 1$, the state of the array is not modified. We now update $i \leftarrow i + 1$, as per line 6 of the algorithm. We also update $j \leftarrow j + 1$ at the end of the first iteration of the loop.

- Now here, we have $i = j = 2$ and $A[j] = 4$. As $A[j] \not\leq A[\text{end}] = 2$, the if statement on line 4 is not executed. So we set $j \leftarrow j + 1$ and proceed to the next iteration of the loop.
- Now here, we have $i = 2, j = 3$, and $A[j] = 3$. As $A[j] \not\leq A[\text{end}] = 2$, the if statement on line 4 is not executed. So we set $j \leftarrow j + 1$ and proceed to the next iteration of the loop.
- Now here, we have $i = 2, j = 4$, and $A[j] = 6$. As $A[j] \not\leq A[\text{end}] = 2$, the if statement on line 4 is not executed. So we set $j \leftarrow j + 1$ and proceed to the next iteration of the loop.
- Now here, we have $i = 2, j = 5$, and $A[j] = 5$. As $A[j] \not\leq A[\text{end}] = 2$, the if statement on line 4 is not executed. So we set $j \leftarrow j + 1$. Now $j = 6$, so the loop terminates.
- At our last step, we have $i = 2, j = 6$, and $A[j] = 2$. As $A[j] = A[\text{end}] = 2$, we swap $A[i]$ and $A[j]$. So $A = [1, 2, 3, 6, 5, 4]$. The algorithm terminates, returning $i = 2$.

We now turn to formalizing Quicksort.

Algorithm 17 Quicksort

```
1: procedure QUICKSORT(Array A[1, ..., n], Integer start, Integer end)
2:     if start < end then
3:         q ← Partition(A, start, end)
4:         Quicksort(A, 1, q − 1)
5:         Quicksort(A, q + 1, end)
```

Example 147. We wish to use Quicksort to sort $A = [2, 6, 5, 1, 4, 3]$. Here, we invoke `Quicksort`$(A, 1, 6)$. So `start` $= 1$ and `end` $= 6$. We proceed as follows. Note that for the `Partition` calls, ony the result will be shown.

(a) We partition the array, invoking `Partition`$(A, 1, 6)$. We note that the call to `Partition` updates $A = [2, 1, 3, 6, 4, 5]$ and returns $q = 3$. So we recursively invoke `Quicksort`$(A, 1, 2)$ and `Quicksort`$(A, 3, 6)$.

(b) We consider `Quicksort`$(A, 1, 2)$. Here, we wish to sort $A[1, 2] = [2, 1]$. Now `Partition`$(A, 1, 2)$ selects **1** as the pivot and returns the index $q = 1$, with $A = [1, 2, 3, 6, 4, 5]$. So now we have start $= 1$, end $= 2$, and $q = 1$. So the subsequent recursive calls `Quicksort`$(A, 1, 1)$ and `Quicksort`$(A, 2, 2)$ are base cases. So no further modifications to A are made here.

(c) We now consider `Quicksort`$(A, 4, 6)$. So we wish to sort the subarray $A[4, \ldots, 6] = [6, 4, 5]$. The call to `Partition`$(A, 4, 6)$ selects **5** as the pivot and returns $q = 5$, with $A = [1, 2, 3, 4, 5, 6]$. So start $= 4$, end $= 6$, and $q = 5$. So the subsequent recursive calls `Quicksort`$(A, 4, 4)$ and `Quicksort`$(A, 6, 6)$ are base cases. So no further modifications to A are made here.

(d) The algorithm terminates, and the array is sorted: $A = [1, 2, 3, 4, 5, 6]$.

We now examine the worst-case complexity of Quicksort. In order for Quicksort to be effective, we need to make progress on sorting two sub-arrays of comparable size. In the extreme case, the pivot element has nothing to its right (or nothing to its left) after the partition step. This yields the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & : n \leq 1, \\ T(n-1) + \Theta(n) & : n > 1. \end{cases}$$

Here, the $\Theta(n)$ term for the non-recursive work arises from the complexity of the Partition function. So unrolling this recurrence, we obtain that $T(n) \in \Theta(n^2)$.

7.2 Randomized Quicksort

In this section, we consider a randomized implementation of Quicksort. The key idea is that we select a random element of the array as our pivot. We then swap this random element so that it is at the end of the array, and then proceed with the partitioning procedure. The expected runtime of this procedure is $O(n\log(n))$. Our proof is adapted from [IS10].

Theorem 148. Let $A[1,\ldots,n]$ be an array of length n, and suppose we select a pivot uniformly at random. Then the expected runtime of the randomized Quicksort procedure is $O(n\log(n))$.

Proof. As we only care about the relative order of the elements, we may assume that A contains precisely the elements of $\{1,\ldots,n\}$ in some order. For each $i,j\in\{1,\ldots,n\}$, let X_{ij} denote the event that Quicksort compares i and j. Let X denote the random variable denoting the number comparisons that Quicksort makes. As no pair of elements are compared twice, we have that:

$$X=\sum_{i=1}^{n}\sum_{j=i+1}^{n}X_{ij}.$$

Our goal is to determine the expected number of comparisons, $\mathbb{E}[X]$. By the linearity of expectation, we have that:

$$\begin{aligned}\mathbb{E}[X]&=\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}]\\&=\sum_{i=1}^{n}\sum_{j=i+1}^{n}\Pr[\text{Quicksort compares } i,j].\end{aligned}$$

We now determine $\Pr[\text{Quicksort compares } i,j]$. Without loss of generality, suppose $i<j$. We note that we only compare elements of the array to the pivot. Now if we select an element $x\in\{i+1,\ldots,j-1\}$ as a pivot before i or j, then i and j belong to separate sub-arrays and are not compared on recursive calls to Quicksort. Hence, i and j are compared if and only if either i or j is selected as the pivot before any element of $\{i+1,\ldots,j-1\}$. As the pivots are selected uniformly at random, we have that the probability that i or j is selected as a pivot before any element of $\{i+1,\ldots,j-1\}$ is:

$$\Pr[\text{Quicksort compares } i,j]=\frac{2}{|\{i,\ldots,j\}|}=\frac{2}{j-i+1}.$$

It follows that:

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\Pr[\text{Quicksort compares } i,j]&=\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{2}{j-i+1}\\&=2\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{j-i+1}\\&\leq 2\sum_{i=1}^{n}\sum_{j=i}^{n}\frac{1}{j}\\&=2n\sum_{j=1}^{n}\frac{1}{j}\\&\leq 2n\cdot\int_{1}^{n}\frac{1}{x}\,dx\\&=2n\ln(n).\end{aligned}$$

So the randomized Quicksort procedure makes at most $2n\ln(n)$ comparisons. Hence, randomized Quicksort has expected runtime $O(n\log(n))$, as desired. □

8 Dynamic Programming

Dynamic programming is a powerful technique to solve computational problems, which have a recursive substructure and recurring subproblems. The idea is to solve these recursive subcases and store these solutions in a lookup table. When a solved recursive subcase is encountered, the existing solution is accessed using only a constant number of steps. A solution to the initial instance is constructed from the solutions to the subcases, typically in a bottom-up manner. Frequently, the computational problems of interest are optimization problems. We begin by introducing some examples amenable to the dynamic programming technique.

8.1 Rod-Cutting Problem

In this section, we examine the Rod-Cutting Problem. Let us consider a motivating example. Suppose we have a rod of length 5, which can be cut into smaller pieces of integer lengths $1, 2, 3$, or 4. These smaller rods can then be further cut into smaller pieces. We stress that the rod pieces can only be cut into pieces of integer length; so pieces of size $1/2$ or π are not considered in this problem. Now suppose that we can sell rods of length 1 for \$1, which we denote $p_1 = 1$. Similarly, suppose that the prices for rods of length $2, \ldots, 5$ are given by $p_2 = 4, p_3 = 7, p_4 = 8$, and $p_5 = 9$ respectively. We make two key assumptions: we will sell all the smaller rods, regardless of the cuts; and that each cut is free. Under these assumptions, how should the rod be cut to maximize the profit? We note the following cuts and the corresponding profits.

- If the rod is cut into five pieces of length 1, we stand to make $5 \cdot p_1 = \$5$.
- If the rod is cut into one piece of length 2 and one piece of length 3, we stand to make $p_2 + p_3 = 4 + 7 = \$11$.
- If the rod is cut into two pieces of length 2 and one piece of length 1, we stand to make $2 \cdot p_2 + p_1 = 8 + 1 = \$9$.

Out of the above options, cutting the rod into one piece of length 2 and one rod of length 3 is the most profitable. Of course, there are other possible cuts not listed above, such as cutting the rod into one piece of length 1 and one piece of length 4. The goal is to determine the most profitable cut. The Rod-Cutting Problem is formalized as follows.

Definition 149 (Rod-Cutting Problem)**.**

- `Instance:` Let $n \in \mathbb{N}$ be the length of the rod, and let $p_1, p_2, \ldots, p_n$ be non-negative real numbers. Here, p_i is the price of a length i rod.
- `Solution:` The maximum revenue, which we denote r_n, obtained by cutting the rod into smaller pieces of integer lengths and selling the smaller rods.

Intuitively, the maximum revenue is determined by examining the revenues for the subdivisions and taking the largest. Mathematically, this amounts to the following expression:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1). \tag{30}$$

We begin by working through an example of utilizing dynamic programming to determine the maximum profit.

Example 150. Suppose we have a rod of length 5, with prices $p_1 = 1, p_2 = 4, p_3 = 7, p_4 = 8, p_5 = 9$. We proceed as follows.

(a) Initialize a lookup table $T[1, \ldots, 5]$. Now for a rod of length 1, there is only one price: p_1. So we set $T[1] := p_1$.

(b) Now consider a rod of length 2. There are two options: either don't cut the rod, or cut the rod into two smaller pieces of length 1. Here, $p_2 = 4$ represents the case in which no cuts to a rod of length 2. Now suppose instead we cut the rod up into two smaller pieces each of length 1. We know that the maximum profit for a rod of length 1 is $r_1 = T[1] = 1$. So the profit of cutting the rod into two smaller pieces each of length 1 is $2r_1 = 2$. Now $r_2 = \max(p_2, 2r_1) = 4$, so we set $T[2] = 4$.

(c) Now consider a rod of length 3. Here, we have more options: we can leave the rod untouched, we can divide the rod into smaller pieces of length 1 or length 2; or we can divide the rod into three pieces of length 1. If we do not divide the rod into smaller pieces, the profit is $p_3 = 7$. Now suppose we divide

the rod up into smaller pieces of length 2 and length 1. We may now keep this configuration, or further divide the rod of length 2 into two rods each of length 1, as discussed in the previous bullet point. Rather than re-solving this problem, we can simply look up the maximum profit for a rod of length 2 in the lookup table. Recall that $r_2 = T[2] = 4$ and $r_1 = 1$. So the profit from cutting the rod into smaller pieces of length 2 and length 1 is $r_2 + r_1 = 5$. Now $r_3 = \max(p_3, r_2 + r_1) = 7$, so we set $T[3] = 7$.

(d) Now consider a rod of length 4. We have the following options for the first cut: leave the rod uncut, in which we stand to make profit $p_4 = 8$; cut the rod into smaller pieces of length 1 and length 3; or cut the rod into two smaller rods, each of length 2. Consider the case in which we cut the rod up into smaller pieces of length 1 and length 3. The natural, though inefficient, approach here is to consider all the ways in which we could cut up the rod of legnth 3. It turns out that we don't need to do this, as the maximum revenue attainable from a rod of length 3 was found in the previous bullet point. This is the power of dynamic programming: once a solution to a smaller problem is found, we simply look it up rather than re-solving the smaller problem. Similarly, we can look up the maximum profit for a rod of length 2. So given our cases, we have the following possible profits:

- The uncut rod of length 4 will result in profit $p_4 = 8$.
- The rod cut into pieces of length 3 and length 1 will result in profit $r_3 + r_1 = T[3] + T[1] = 7 + 1 = 8$.
- The rod cut into two pieces, each of length 2, will result in profit $2r_2 = 2 \cdot T[2] = 2 \cdot 4 = 8$.

So $T[4] = \max(8, 8, 8) = 8$.

(e) Finally, consider our original rod of length 5. We have the following possible initial cuts:

- We can leave the rod uncut, in which case we will make profit $p_5 = 9$.
- We can cut the rod into one piece of length 4 and one piece of length 1. The maximum revenue attainable by cutting up a rod of length 4 was determined already. So we can simply look up this solution in $T[4]$. Thus, the profit in this case is $r_4 + r_1 = T[4] + T[1] = 8 + 1 = 9$.
- We can cut up the rod into one piece of length 3 and one piece of length 2. By similar argument as above, we may simply look up the maximum revenues attainable from a rod of length 3 and a rod of length 2. So our profit is $r_3 + r_2 = T[3] + T[2] = 7 + 4 = 11$.

So $r_5 = \max(9, 9, 11) = 11$. Thus, we set $T[5] = 11$.

We conclude that we stand to make \$11 from a rod of length 5.

While the expresion (30) may not seem insightful, it in fact provides an algoritheorem to compute r_n. Example 150 provides a tangible example of this algoritheorem. The goal now is to generalize the algoritheorem from Example 150 to work for any rod of positive integer length any list of prices. We proceed as follows.

(a) Initialize the lookup table $T[1, \ldots, n]$, and set $T[1] := p_1$.

(b) We set $T[2] := \max(p_2, 2r_1)$. Here, p_2 represents the case in which no cuts to a rod of length 2, and $2r_1$ represents the case in which a rod of length 2 is cut into two rods each of length 1. We note that $r_1 = T[1] = p_1$.

(c) We set $T[3] := \max(p_3, r_1 + r_2)$. Now $r_1 = T[1]$, and $r_2 = T[2]$. We have already solved the rod cutting problem for a length 2 rod, so we simply look up r_2 in the table T rather than re-solving the problem.

(d) $T[4] := \max(p_4, r_1 + r_3, 2r_2)$. As we have already computed r_1, r_2, r_3, we may look up their respective values in T rather than re-computing these values.

Continuing in this manner, we compute r_n, which is the value in $T[n]$ after the algoritheorem terminates.

Remark 151. This algorithm only provides the maximum revenue. It does not tell us how to achieve that result. As an exercise, modify the algoritheorem to produce an optimal set of rod cuts.

8.2 Recurrence Relations

The key aspect in designing a dynamic programming algorithm is to identify the recursive structure. In particular, finding an explicit recurrence informs us as to the lookup table, as well as the natural strategy to fill in the lookup table. In this section, we discuss strategies for identifying recurrence relations. We will use combinatorics to explore these ideas. Our exposition is adopted from [Loe18].

Suppose we are interested in counting sets of combinatorial objects. And suppose we have several related families of objects, which we label $T_0, T_1, \ldots, T_n, \ldots$. Here, we think of the index n as measuring the size of the objects in T_n (e.g., the set of rooted binary trees on n vertices). It is sometimes possible to explicitly describe the objects in T_n, such as with the sum and product rules, to arrive at a closed-form expression for $|T_n|$, the number of elements in T_n. Other times, it is more natural to describe a construction for the objects in T_n using smaller objects from the sets $T_0, \ldots, T_{n-1}$. Such arguments are inherent in the dynamic programming paradigm, and they lead to natural recursive formulas which describe the number of viable solutions (or if there even is a viable solution).

We illustrate this technique below.

Example 152. Let W_n be the set of all words $\omega \in \{0,1\}^n$ such that there is no substring of the form 00 in ω. Let $f_n := |W_n|$. Our first step is to describe the initial conditions. Observe that $W_0 = \{\varepsilon\}$, where ε is the string of length 0, and $W_1 = \{0,1\}$. So $f_0 = 0$ and $f_1 = 2$.

Now consider a string $\omega \in W_n$ of length $n \geq 2$. We note that the first character $\omega_0 \in \{0,1\}$. Consider the following cases.

- **Case 1:** If $\omega_1 = 0$, then necessarily $\omega_2 = 1$; otherwise, we would have a 00 substring. So $\omega = 01\tau$, where $\tau \in W_{n-2}$. There are $f_{n-2} = |W_{n-2}|$ ways to select τ. So there are f_{n-2} such strings ω of the form 01τ in W_n.
- **Case 2:** If $\omega_1 = 1$, then the second character may be either 0 or 1. In this case, $\omega = 0\tau$, where $\tau \in W_{n-1}$. So we have f_{n-1} such ways to select ω in this case.

As Cases 1 and 2 are disjoint, we have that $f_n = f_{n-1} + f_{n-2}$, with the initial conditions that $f_0 = 1$ and $f_1 = 2$.

Example 153. Recall from Discrete Math that $\binom{n}{k}$ counts the number of k-element subsets of $\{1, \ldots, n\}$. We derive a recurrence for $\binom{n}{k}$ by counting the number of ways to construct a k-element set S. We first ascertain the base cases. We set $\binom{n}{k} := 0$ whenever $k < 0$ or $k > n$.

- **Case 1:** Suppose that $1 \in S$. In this case, we must select the remaining $n-1$ elements of S from $\{2, \ldots, n\}$. There are $n-1$ such elements from which to choose, and we want a $k-1$ element set $S' \subseteq \{2, \ldots, n\}$ such that $S = \{1\} \cup S'$. There are $\binom{n-1}{k-1}$ such sets.
- **Case 2:** Suppose that $1 \notin S$. So we select $S \subseteq \{2, \ldots, n\}$. There are $n-1$ such elements from which to choose, and we would like a k-element set. So there are $\binom{n-1}{k-1}$ such ways to select S.

So the total number of ways to select a k-element set from $\{1, \ldots, n\}$ is $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

Example 154. Consider an $n \times m$ rectangle, where $n, m \geq 0$. So we have n rows and m columns. We start at the origin $(0,0)$, with the goal of reaching (m,n). We may only move one cell to the left or one cell up at a time. That is, we may move from (x,y) to $(x+1,y)$ or $(x,y+1)$. We now count the number of ways to reach (m,n). Denote this recurrence $L(m,n)$. Observe that when either $n = 0$ or $m = 0$, there is only one such path. So $L(m,0) = L(0,n) = 1$.

Now consider $L(m,n)$. Here, if we move one cell up, then we have an $(n-1) \times m$ rectangle, which yields $L(m,n-1)$ such paths. If we instead move one cell to the left, then we have an $n \times (m-1)$ rectangle, which yields $L(m-1,n)$ such paths. These choices are disjoint, so we have the recurrence:

$$L(n,m) = \begin{cases} 1 & : n = 0 \text{ or } m = 0, \\ L(n-1,m) + L(n,m-1) & : n, m > 1. \end{cases}$$

8.3 Longest Common Subsequence

In this section, we introduce the Longest Common Subsequence Problem, which is also amenable to the dynamic programming technique. Unlike other dynamic programming problems, the Longest Common Subsequence Problem utilizes a two-dimensional table rather than a one-dimensional array. The purpose of this section is to illustrate the usage of multidimensional lookup tables in dynamic programming problems. To this end, the Longest Common Subsequence Problem serves as a tangible example. We begin by formalizing the Longest Common Subsequence Problem.

Definition 155 (Subsequence)**.** Let Σ be a finite set, which we refer to as an *alphabet*. Let $\omega \in \Sigma^n$ be a string of length n (so each letter of ω belongs to Σ). We say that $\psi \in \Sigma^m$ is a subsequence of ω if there exists a strictly increasing sequence of indices $(i_1, i_2, \ldots, i_m)$ such that $\omega_{i_k} = \psi_k$ for all $k \in [m]$.

Example 156. Let $\omega = (A, B, C, B, D, A, B)$, and let $\psi = (A, C, D, B)$. We indicate how ψ is realized as a subsequence of ω by bolding the appropriate letters:

$$(\mathbf{A}, B, \mathbf{C}, B, \mathbf{D}, A, \mathbf{B}).$$

Note that the letters of ψ need to appear in the same order in ω; that is, the selected A must appear before the selected C, which then appears before the selected D, and so on. However, the letters of ψ need not be consecutive or contiguous in ω. Consider the sequence of indices $(1, 3, 5, 7)$. So $\psi_1 = \omega_1$, $\psi_2 = \omega_3, \psi_3 = \omega_5$, and $\psi_4 = \omega_7$. Thus, ψ is a subsequence of ω.

Definition 157 (Common Subsequence)**.** Let Σ be an alphabet. Let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be sequences. We say that $\psi \in \Sigma^\ell$ is a *common subsequence* of ω and τ if: ψ is a subsequence of ω, and ψ is a subsequence of τ. Note that ψ does not have to appear as a subsequence in the same position in both ω and τ.

Example 158. Let $\omega = (0, 2, 1, 2, 3, 0, 1)$ and $\tau = (2, 3, 1, 0, 2, 0)$. The sequence $(2, 1, 0)$ is a subsequence of both ω and τ. Here, $(2, 1, 0)$ appears in ω at the indices $(2, 3, 6)$. That is, $(2, 1, 0)$ appears in ω as follows:

$$(0, \mathbf{2}, \mathbf{1}, 2, 3, \mathbf{0}, 1).$$

Similarly, $(2, 1, 0)$ appears in τ at the indices $(1, 3, 4)$. That is, $(2, 1, 0)$ is realized as a subsequence of τ as follows:

$$(\mathbf{2}, 3, \mathbf{1}, \mathbf{0}, 2, 0)$$

Definition 159 (Longest Common Subsequence Problem (LCS))**.**

- `Instance:` Let Σ be an alphabet, and let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be sequences of length n and m respectively.
- `Solution:` A sequence ψ that is common to both ω and τ; and for any other common subsequence σ of ω and τ, $|\sigma| \leq |\psi|$.

The naïve approach to solving LCS is enumerating all the possible subsequences of σ and τ, and recording the longest. Without loss of generality, suppose that $m \leq n$. So there are 2^m possible index sequences to check, which correspond bijectively to subsequences of τ. So for large sequences, the brute force and ignorance solution is not a practical solution. The dynamic programming approach provides a linear time algoritheorem instead.

Dynamic programming works best when optimal solutions to subproblems can be used to construct an optimal solution to the original instance. We first show that LCS exhibits this property.

Theorem 160. Let Σ be an alphabet, and let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be sequences. Let $\psi \in \Sigma^k$ be a longest common subsequence of ω and τ. The following hold:

(a) If $\omega_n = \tau_m$, then $\psi_k = \omega_n = \tau_n$ and $\psi[1, \ldots, k-1]$ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$.

(b) If $\omega_n \neq \tau_m$ and $\psi_k \neq \omega_n$, then ψ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and τ. Similarly, if $\omega_n \neq \tau_m$ and $\psi_k \neq \tau_m$, then ψ is a longest common subsequence of ω and $\tau[1, \ldots, m-1]$.

Proof.

(a) Let σ be a common subsequence of ω and τ whose last digit does not correspond to the last instance of the character $\omega_n = \tau_m$ in ω and τ. Then σ can be augmented by appending the character $\omega_n = \tau_m$. So every longest common subsequence of ω and τ has last character $\omega_n = \tau_m$.

We now show that $\psi[1, \ldots, k-1]$ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$. Observe that $\psi[1, \ldots, k-1]$ is a common subsequence of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$. Suppose to the contrary that there exists a longest common subsequence σ of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$, with $|\sigma| > k$. Then σ can be augmented with $\omega_n = \tau_m$ to obtain a common subsequence of ω and τ. This contradicts the assumption that any longest common subsequence of ω and τ has length k. So $\psi[1, \ldots, k-1]$ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$.

(b) Suppose that $\omega_n \neq \tau_m$. Now suppose that $\psi_k \neq \omega_n$. We show that ψ is a longest common subsequence of $\omega[1, \ldots, n-1]$ and τ, by contradiction. Let σ be a longest common subsequence of $\omega[1, \ldots, n-1]$ and τ of length $|\sigma| > k$. Clearly, σ is a common subsequence of ω and τ. Now $|\sigma| > |\psi| = k$, contradicting the assumption that ψ was a longest common subsequence of ω and τ. So ψ is a longest common subsequence of ω and τ. Interchanging the roles of ω and τ, we obtain that: if $\omega_n \neq \tau_m$ and $\psi_k \neq \tau_m$, then ψ is a longest common subsequence of ω and $\tau[1, \ldots, m-1]$.

□

Theorem 160 provides the insights necessary for designing a dynamic programming algoritheorem to solve LCS. Let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be sequences. If $\omega_n = \tau_m$, we record the last character and examine the smaller LCS instance with $\omega[1, \ldots, n-1]$ and $\tau[1, \ldots, m-1]$. If $\omega_n \neq \tau_m$. Otherwise, we need to find the longest common subsequences of ω and $\tau[1, \ldots, m-1]$; and $\omega[1, \ldots, n]$ and τ. These observations yield a natural recurrence to compute the length of the longest common subsequence for a pair of strings:

$$\ell[i,j] = \begin{cases} 0: & i = 0 \text{ or } j = 0, \\ \ell[i-1, j-1] + 1: & i, j > 0 \text{ and } x_i = y_j, \\ \max(\ell[i-1, j], \ell[i, j-1]): & i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Using the recurrence $\ell[i, j]$ as a template, we design an explicit dynamic programming algoritheorem. We proceed as follows.

(a) Let $\omega \in \Sigma^n, \tau \in \Sigma^m$ be our input sequences. We initialize a lookup table $T[0, \ldots, n][0, \ldots, m]$ to be a two-dimensional array, where each cell stores:

- A natural number corresponding to the length of a longest common subsequence; and
- A pointer to another cell in the lookup table, which corresponds to the optimal subproblem as specified in Theorem 160.

Now recall that if either of the input sequences have length 0, the length of the longest common subsequence is 0. Therefore, we set $T[i][0] = 0$ and $T[0][j] = 0$ for all $i \in [n]$ and all $j \in [m]$. While our original input sequences may not have length 0, sequences we encounter in subproblems may indeed have length 0.

(b) We now proceed to fill in the remaining cells in a bottom up manner, row-by-row. Each row is filled left-to-right. The cells $T[i][j]$ are filled as follows.

- **Case 1:** Suppose $\omega_i = \tau_j$. By Theorem 160, any longest common subsequence ψ of $\omega[1, \ldots, i]$ and $\tau[1, \ldots, j]$ ends with $\omega_i = \tau_j$. Furthermore, $\psi[1, \ldots, |\psi| - 1]$ is a longest common subsequence of $\omega[1, \ldots, i-1]$ and $\tau[1, \ldots, j-1]$. So we take the following actions:
  - Set $T[i][j]$.length $= T[i-1][j-1]$.length $+ 1$; and
  - Set $T[i][j]$.subproblem $= T[i-1][j-1]$.
- **Case 2:** Suppose $\omega_i \neq \tau_j$. Theorem 160 tells us that we need to consider the two subproblems, whose solutions (or at least, their optimal lengths) are stored in: $T[i-1][j]$ and $T[i][j-1]$, respectively. If $T[i-1][j]$.length $\geq T[i][j-1]$.length, we set:
  - $T[i][j]$.length $= T[i-1][j]$.length
  - $T[i][j]$.subproblem $= T[i-1][j]$.

  Otherwise, we set:

- $T[i][j]$.length $= T[i][j-1]$.length
- $T[i][j]$.subproblem $= T[i][j-1]$.

In order to construct a longest common subsequence from the lookup table T, we start at $T[n][m]$ and follow the pointers to the subproblem. Each time some $T[i][j]$ points to $T[i-1][j-1]$ as a subproblem, we prepend the character $\omega_i = \tau_j$ to the front of the longest common subsequence. We stop once the currently visited cell has no pointer to a subproblem.

Example 161. Let $\omega = (A, B, C)$ and $\tau = (B, A, C, B, D)$. By inspection, it is easy to see that any longest common subsequence of ω and τ has length 2. In particular, $(A, B), (B, C)$, and (A, C) are all longest common subsequences of ω and τ. We work through the dynamic programming algoritheorem to explicitly find a longest common subsequence.

(a) We begin by initializing a 4×6 lookup table $T[0, \ldots, 3][0, \ldots, 5]$, and filling the first row and column with 0's. So we have:

		B	**A**	**C**	**B**	**D**
	0	0	0	0	0	0
A	0					
B	0					
C	0					

(b) We now fill Row 1.

- Consider $T[1][1]$. Observe that $\omega_1 = A$ and $\tau_1 = A$ are different. So $T[1][1]$.length is the maximum of $T[1][0]$.length $= 0$ and $T[0][1]$.length $= 0$. Thus, $T[1][1]$.length $= 0$. By Case 2 of the algoritheorem, $T[1][1]$.subproblem points to $T[1][0]$.
- Consider $T[1][2]$. Observe that $\omega_1 = \tau_2$. So $T[1][2]$.length $= T[0][1]$.length $+ 1 = 1$, and $T[1][2]$.subproblem points to $T[0][1]$.
- Consider $T[1][3]$. Observe that $\omega_1 = A$ and $\tau_3 = C$ are different. So $T[1][3]$.length is the maximum of $T[0][3]$.length $= 0$ and $T[1][2]$.length $= 1$. So $T[1][3]$.length $= 1$, and $T[1][3]$.subproblem points to $T[1][2]$.
- Consider $T[1][4]$. Observe that $\omega_1 = A$ and $\omega_4 = B$ are different. By similar argument as for $T[1][1]$ and $T[1][3]$, we set $T[1][4]$.length $= 1$ and $T[1][4]$.subproblem to point to $T[1][3]$.
- Consider $T[1][5]$. By similar argument as for $T[1][4]$, $T[1][5]$.length $= 1$ and $T[1][5]$.subproblem points to $T[1][4]$.

The updated lookup table is as follows:

		B	**A**	**C**	**B**	**D**
	0	0	0	0	0	0
A	0	$\leftarrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\leftarrow 1$
B	0					
C	0					

As an exercise, try filling out the rest of the lookup table. A completed lookup table is on the next page, but the details of how to fill in the specific cells are omitted. Please stop by office hours to with any questions on how to complete the lookup table.

(c) We next fill Rows 2-3, omitting the detailed explanation associated with filling Row 1. The completed lookup table is as follows.

		B	**A**	**C**	**B**	**D**
	0	0	0	0	0	0
A	0	$\leftarrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\leftarrow 1$
B	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\nwarrow 2$	$\leftarrow 2$
C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\leftarrow 2$

(d) Finally, we construct a longest common subsequence of ω and τ from the lookup table. We start at $T[3][5]$ and follow the arrows, prepending the character at the given index every time we see $\nwarrow$. So we have the sequence:

$$\begin{aligned} T[3][5] &\rightarrow T[3][4] \rightarrow T[3][3] \text{ (Record C)} \\ &\rightarrow T[2][2] \rightarrow T[2][1] \text{ (Record B)} \\ &\rightarrow T[1][0]. \end{aligned}$$

After which, we stop, as $T[1][0]$ does not reference any subproblems. So our longest common subsequence is (B, C), which we identified at the start of this example.

8.4 Edit Distance

In this section, we consider the edit distance problem. Our goal in this section is to practice filling in a lookup table. The exposition is adapted from [CG18].

Fix an alphabet Σ, and let $x, y \in \Sigma^*$. Our goal is to convert x into y using the minimum number of operations. That is, we wish to find the minimum distance between $x \in \Sigma^n$ and $y \in \Sigma^m$. This problem is motivated by applications, such as comparing two strands of DNA in bioinformatics. In addition, computing the edit distance is a key dynamic programming algorithm in natural language processing.

In order to compute the edit distance of two strings, we need to introduce the allowed operations, as well the costs of each operation. Our operations are as follows.

- **Substitution:** We may replace character i in x with a new character. For example, let $x = \texttt{do}$ and $y = \texttt{so}$. We may replace $x[1] = \texttt{s}$ obtain y.
- **Indel:** An *indel* operation stands for an insertion/deletion operation. We may insert a new character into x, shifting the subsequent characters. Equivocally, we may delete a character from y, shifting the subsequent characters forward. As an example, $x = \texttt{grand}$ and $y = \texttt{grande}$ differ by a single indel operation.
- **Swap:** We may swap two consecutive elements in x. So if $x = (x_1, \ldots, x_i, x_{i+1}, \ldots, x_n)$ and we swap x_i, x_{i+1}, then our new string is $x = (x_1, \ldots, x_{i+1}, x_i, \ldots, x_n)$. As an example, $x = \texttt{thier}$ and $y = \texttt{their}$ differ by a single swap operation; namely, swapping the pair $\texttt{ie}$ in x.

For our purposes, we take the costs of each operation to be 1. So the distance between two strings under this cost scheme is known as the *Damerau–Levenshtein* (DL) distance.

We now turn to identifying a recurrence. Given strings $x[1, \ldots, n]$ and $y[1, \ldots, m]$, denote $c[i, j]$ to be the minimum DL distance between $x[1, \ldots, i]$ and $y[1, \ldots, j]$. We note that if $i = j = 0$, then $x = y = \varepsilon$ (that is, x and y are both the empty string). In this case, $c[i, j] = 0$. We now turn to our recursive cases.

- Suppose that $x_i = y_j$. In this case, we wish to find the DL distance of $x[1, \ldots, i-1]$ and $y[1, \ldots, j-1]$. This is precisely $c[i-1, j-1]$.
- If $x[1, \ldots, i]$ is not empty (that is, if $i > 0$), we may try to align $x[1, \ldots, i]$ to $y[1, \ldots, j]$ by simply deleting the last character of $x[1, \ldots, i]$. In this case, we have cost $1 + c[i-1, j]$.
- Similarly, if $j > 0$, then we may simply delete the last character of $y[1, \ldots, j]$, and try to align $x[1, \ldots, i]$ and $y[1, \ldots, j-1]$. The cost here is $1 + c[i, j-1]$.
- If $i, j > 0$ and $x_i \neq x_j$, we may substitute x_i for y_j, and then find the DL distance of $x[1, \ldots, i-1]$ and $y[1, \ldots, j-1]$. This is precisely $1 + c[i-1, j-1]$.
- If $i, j > 1$ and $x_{i-1}x_i = y_j y_{j-1}$, we may swap x_{i-1} and x_j. We then find the DL distance of $x[1, \ldots, i-2]$ and $y[1, \ldots, j-2]$. This is precisely $2 + c[i-2, j-2]$.

The case analysis is involved, and the mathematical recursive formula for $c[i, j]$ is quite dense. We omit the formula, as the case analysis suffices to fill in the lookup table. We now turn to consider an example.

Example 162. Suppose we wish to align $x = (S, T, E, P)$ and $y = (A, P, E)$. As $c[i, j]$ has two variables, we have a two-dimensional lookup table.

	ε	**A**	**P**	**E**
ε				
S				
T				
E				
P				

We now turn to filling in the lookup table row-by-row.

- **Row ε.**
  - We consider $c[0,0]$. In this case, both strings are ε. So $c[0,0] = 0$.
  - We consider $c[0,1]$, where we seek to align ε and A. This can be accomplished with a single indel operation. So we set $c[0,1] = 1$ and have $c[0,1]$ point to $c[0,0]$ with a label for indel.
  - We consider $c[0,2]$, where we seek to align ε and (A,P). This can be accomplished by inserting P and aligning P and (A,P). So $c[0,2] = 1 + c[0,1] = 2$. Furthermore, we have $c[0,2]$ point to $c[0,1]$ with a label for indel.
  - We consider $c[0,3]$, where we seek to align ε and (A,P,E). This can be accomplished by inserting E and aligning E and (A,P,E). So $c[0,3] = 1 + c[0,2] = 3$. Furthermore, we have $c[0,2]$ point to $c[0,1]$ with a label for indel.

  The updated table is below.

	ε	**A**	**P**	**E**
ε	0	$\xleftarrow{\text{indel}}$ 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
S				
T				
E				
P				

- **Row S:**
  - We consider $c[1,0]$. Here, we are aligning S and ε, which can be accomplished with a single indel operation. So we set $c[1,0] = 1$ and point upwards.
  - We consider $c[1,1]$. Here, we are aligning S and A, which can be accomplished with a single substitution operation. So we set $c[1,1] = 1 + c[0,0] = 1$. Now $c[1,1]$ points diagonally to $c[0,0]$, and we label this with substitution.
  - We consider $c[1,2]$. Here, we seek to align S and (A,P). As $S \neq P$, we may either delete S (which means we consider $c[0,2]$) or insert P (which means we consider $c[1,1]$). As $1 = c[1,1] < c[0,2] = 2$, we insert P. So $c[1,2] = 2$, and we set $c[1,2]$ to point to $c[1,1]$ with label indel.
  - We consider $c[1,3]$. Here, we seek to align S and (A,P,E). As $S \neq E$, we may either delete S (which means we consider $c[0,3]$) or insert E (which means we consider $c[1,2]$). As $2 = c[1,2] < c[0,3] = 3$, we choose to insert E. So $c[1,3] = 1 + c[1,2] = 3$, and we set $c[1,3]$ to point to $c[1,2]$ with label indel. The updated table is below.

	ε	**A**	**P**	**E**
ε	0	$\xleftarrow{\text{indel}}$ 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
S	$\uparrow$ indel 1	$\nwarrow$ sub 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
T				
E				
P				

Remark 163. By similar arguments as above, the remaining empty cells in columns ε and **A** correspond to indel operations. we fill them in below.

	ε	**A**	**P**	**E**
ε	0	$\xleftarrow{\text{indel}}$ 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
S	$\uparrow$ indel 1	$\nwarrow$ sub 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
T	$\uparrow$ indel 2	$\uparrow$ indel 2		
E	$\uparrow$ indel 3	$\uparrow$ indel 3		
P	$\uparrow$ indel 4	$\uparrow$ indel 4		

- **Row T:**
  - We consider the cell $c[2,2]$. Here, we seek to align (S,T) and (A,P). As $T \neq P$, we may either use an indel operation such as to delete T (in which case we consider $c[1,2] = 2$) or P (in which case, we consider $c[2,1] = 2$), or a substitution operation to change T to P (in which case, we consider $c[1,1] = 1$). We choose the substitution operation. So we set $c[2,2] = 1 + c[1,1]$. Now $c[2,2]$ points to $c[1,1]$ with label sub.
  - We consider the cell $c[2,3]$. Here, we seek to align (S,T) with (A,P,E). As $T \neq E$, we may either use an indel operation such as to delete T (in which case, we consider $c[1,3] = 3$) or E (in which case, we consider $c[2,2] = 2$), or we may substitute T for E (in which case, we consider $c[1,2] = 2$). As $c[1,2] = c[2,2]$, we may choose either indel or sub. We opt for deleting E. So we set $c[2,3] = 1 + c[2,2] = 3$, and $c[2,3]$ points to $c[2,2]$ with label for index.

	ε	**A**	**P**	**E**
ε	0	$\xleftarrow{\text{indel}}$ 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
S	$\uparrow$ indel 1	$\nwarrow$ sub 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
T	$\uparrow$ indel 2	$\uparrow$ indel 2	$\nwarrow$ sub 2	$\xleftarrow{\text{indel}}$ 3
E	$\uparrow$ indel 3	$\uparrow$ indel 3		
P	$\uparrow$ indel 4	$\uparrow$ indel 4		

- **Row E:**
  - Consider $c[3,2]$. Here, we seek to align (S,T,E) and (A,P). As $E \neq P$, we may either use an indel operation such as to delete E (in which case, we consider $c[2,2] = 2$) or to delete P (in which case, we consider $c[3,1] = 3$), or we may use a substitution operation to change E to P (in which case, we consider $c[2,1] = 2$). We opt for deleting E. So we set $c[3,2] = 1 + c[2,2] = 3$, and we set $c[3,2]$ to point to $c[2,2]$ with label indel.
  - Consider $c[3,3]$. Here, we seek to align (S,T,E) and (A,P,E). As the end characters agree, we do not modify them. Instead, we consider $c[2,2]$. So we simply set $c[3,3] = c[2,2]$ and set $c[3,3]$ to point to $c[2,2]$.

  The updated lookup table is below.

	ε	**A**	**P**	**E**
ε	0	$\xleftarrow{\text{indel}}$ 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
S	$\uparrow$ indel 1	$\nwarrow$ sub 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
T	$\uparrow$ indel 2	$\uparrow$ indel 2	$\nwarrow$ sub 2	$\xleftarrow{\text{indel}}$ 3
E	$\uparrow$ indel 3	$\uparrow$ indel 3	$\uparrow$ indel 3	$\nwarrow$ 2
P	$\uparrow$ indel 4	$\uparrow$ indel 4		

- **Row P:**
  - We consider $c[4,2]$. Here, we seek to align (S,T,E,P) with (A,P). As the end characters agree, we set $c[4,2] = c[3,1]$ and set $c[4,2]$ to point to $c[3,1]$.
  - We consider $c[4,3]$. Here, we seek to align (S,T,E,P) and (A,P,E). We have several options. First, we may swap (E,P) in (S,T,E,P) to obtain (S,T,P,E); in this case, we conisder $c[2,1] = 2$. Alternatively, as $P \neq E$, we may perform an indel operation either deleting P (in which case, we consider $c[3,3] = 2$). We choose the swap operation. So we set $c[4,3] = c[2,1] + 1 = 3$ and point to $c[2,1]$.

	ε	**A**	**P**	**E**
ε	0	$\xleftarrow{\text{indel}}$ 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
S	$\uparrow$ indel 1	$\nwarrow$ sub 1	$\xleftarrow{\text{indel}}$ 2	$\xleftarrow{\text{indel}}$ 3
T	$\uparrow$ indel 2	$\uparrow$ indel 2	$\nwarrow$ sub 2	$\xleftarrow{\text{indel}}$ 3
E	$\uparrow$ indel 3	$\uparrow$ indel 3	$\uparrow$ indel 3	$\nwarrow$ 2
P	$\uparrow$ indel 4	$\uparrow$ indel 4	$\nwarrow$ 3	3 swap- point to c[2, 1]

Now to recover an optimal sequence of operations, we can simply follow the arrows. Namely, we do the following:

- Substitue $S \mapsto A$. So we now have $(S, T, E, P) \mapsto (A, T, E, P)$.
- Delete the T in (A, T, E, P) to obtain (A, E, P).
- Swap the (E, P) substring to obtain (A, P, E).

9 Computational Complexity Theory

9.1 Decision Problems

The goal of Computational Complexity Theory is to classify computational problems according to the amount of resources required to solve them. Space and time are the two most common measures of complexity. Time complexity measures how many computations are required for a computer to solve decide an instance of the problem, with respect to the instance's size. Space complexity is analogously defined for the amount of extra space a computer needs to decide an instance of the problem, with respect to the instance's size.

So far, in Algorithms, we have only studied problems which have efficient (polynomial-time) solutions. However, there are numerous practical problems of interest for which it is unkown if efficient solutions exist. Some of these problems include the Traveling Salesman Problem, Protein Folding, Scheduling, and Optimization problems. For other problems, such as generalizations of Chess, Checkers, and Go, it has been proven that no efficient algorithms exist. Computational complexity provides a framework to help us understand which problems have efficient solutions and which don't. For problems that have resisted or don't have efficient solutions, there are algorithmic techniques to construct approximate solutions in polynomial time. Such algorithms are known as *approximation algorithms*.

In this section, we seek to understand the complexity classes P and NP, from the perspective of algorithms. Informally, P is the class of decision problems that can be efficiently solved. The class NP contains precisely the decision problems, where a correct solution is easy to verify. We see that $\mathsf{P} \subseteq \mathsf{NP}$, as we can solve a problem in P and compare the solutions when verifying. It is unknown whether $\mathsf{P} = \mathsf{NP}$, though it is generally believed that they are different.

Whether $\mathsf{P} = \mathsf{NP}$ is of great practical interest to everyone- not just those in computing. We consider one example. The RSA Cryptosystem, which secures our online transactions, is based on factoring large integers. It is well-known that factoring integers belongs to NP; however, it is not known if this problem is NP-complete (one of the hardest problems in NP). If it turns out $\mathsf{P} = \mathsf{NP}$, then there exists an efficient algorithm to factor integers. Such an algorithm can be used to break RSA, and therefore compromise the security of our online transactions.

We proceed to introduce the formal framework. Attention will be restricted to the computational complexity of decision problems. Informally, a decision problem provides some input. The goal is to determine whether the input satisfies some given property. We seek to formalize the notion of a decision problem in terms of *formal languages*. We proceed as follows.

Definition 164. Let Σ be a finite set, which we refer to as an *alphabet*. The set of all possible finite strings over Σ is the set Σ^*.

Example 165. Let $\Sigma = \{0, 1\}$ be a binary alphabet. We note that $010 \in \Sigma^*$, as 010 is a finite string where each letter is drawn from Σ. However, $012 \notin \Sigma^*$ as $2 \notin \Sigma$.

Remark 166. We stress that Σ^* contains only **finite** length strings. Recall that any input to a computer must be finite in length.

We next introduce the notion of a language.

Definition 167. Let Σ be an alphabet. A *language* over Σ is a set $L \subseteq \Sigma^*$.

Example 168. Take $\Sigma = \{0, 1, 2\}$. An example of a language L over Σ is the set of strings that start with 2. That is, $L = \{\omega \in \Sigma^* : \omega_1 = 2\}$.

With the notion of a language in hand, we can now define the notion of a decision problem. Formally, a decision problem is simply a language.

Definition 169. Let Σ be an alphabet. A *decision problem* is a language $L \subseteq \Sigma^*$.

Remark 170. Before considering examples of decision problems, we seek to unpack the definition first. Any computer program receives as input textual data. That is, any input can be represented as a string ω of text for the purposes of the computer. We then seek to answer the following question: is the string ω in the given language L?

Motivation: Defining a decision problem as a formal language provides a tangible, combinatorial object with which to work. Having this combinatorial object is useful when proving theorems. One key goal of computational complexity is to determine whether two related complexity classes are the same or different (e.g., Heirarchy Theorems, Ladner's Theorem, Oracle constructions). A common strategy to show that two complexity classes are different is to construct a language L that belongs to one complexity class, but not the other. The construction of such a language L is usually quite technical, and it is important to be able to precisely identify which strings of Σ^* we include in L.

We won't reach a point where we are proving such theorems in this class. However, you should be comfortable formalizing the notion of a decision problem as a language. To this end, we consider some examples.

Example 171. Consider the following decision problem.

- Instance: Let $m, n \in \mathbb{N}$.
- Decision: Is $\gcd(m, n) = 1$?

We can formalize this decision problem as a language as follows. Consider the following:

$$L = \{\langle m, n\rangle : \gcd(m, n) = 1\}.$$

Here, $\langle m, n\rangle$ denotes some *encoding* of m and n, for the computer. If it is helpful, you can think about m and n being represented as binary strings. However, we don't concern ourselves with the technicalities of how to encode these objects; only that some encoding exists.

We consider the following instances.

- Suppose $m = 3$ and $n = 5$. The decision problem asks us whether $\gcd(3, 5) = 1$. In other words, is $\langle 3, 5\rangle \in L$? Here, the answer is Yes.
- Suppose $m = 2$ and $n = 4$. Is $\langle 2, 4\rangle \in L$? Here, the answer is No, as $\gcd(2, 4) = 2$.

Example 172. Consider the following decision problem. The Path problem is defined as follows.

- Instance: Let G be a simple, unweighted graph, and let $u, v \in V(G)$. Let $k \in \mathbb{N}$.
- Decision: Does there exist a path from u to v in G, using at most k edges?

We formalize this decision problem as a language, as follows.

$$L_{\mathsf{Path}} = \{\langle G, u, v\rangle : G \text{ is a graph; } u, v \in V(G), \text{ and } G \text{ has a path from } u \text{ to } v\}.$$

9.2 P and NP

In this section, we introduce the complexity classes P and NP. Informally, P is the set of decision problems (languages) that can be solved efficiently, and NP is the set of decision problems where a correct solution accompanied by supporting work (which we call a *certificate*) can be efficiently verified. By efficient, we mean polynomial time in the length of the input.

Intuitively, we may think of P as consisting of decision problems that are easy to solve; while NP is the set of decision problems that are easy to grade, provided sufficient work is shown. With this in mind, it is straight-forward to intuit why $\mathsf{P} \subseteq \mathsf{NP}$. In order to check one's work, the grader can simply solve the problem themselves, ignoring the student's work.

It is unknown as to whether $\mathsf{NP} \subseteq \mathsf{P}$, though it is widely believed that $\mathsf{P} \neq \mathsf{NP}$. Resolving this is known as the P vs. NP problem, which is the biggest open problem in computer science and one of the biggest open problems in mathematics.

In this section, we seek to clearly define P and NP, as well as consider examples of problems belonging to each of these complexity classes. In particular, we emphasize the details required to show (prove) a given decision problem belongs to each of these complexity classes. In the next section, we will discuss the P vs. NP problem in more detail. We begin with the definition of P and NP.

Definition 173. Let $\Sigma = \{0, 1\}$. We say that a language $L \in \mathsf{P}$ if there exists a polynomial $p(n)$ (depending only on L) and an algorithm A such that for all $x \in \Sigma^*$:

- $A(x) = 1$ if $x \in L$, and
- $A(x) = 0$ if $x \notin L$.

Here, $A(x)$ denotes A run on the input x. In all cases, A takes at most $p(|x|)$ steps.

Remark 174. More succinctly, P is the set of decision problems that are solvable in polynomial time. Here, we stress that the polynomial may (and often does) vary amongst decision problems. For example, deciding if an element belongs to a sorted array takes $O(\log(n))$ time using binary search, while deciding if there exists an element in an unsorted array takes $O(n)$ time using linear search. Note that searching a sorted array belongs to P, as does searching an unsorted array.

In order to show a decision problem is in P, it suffices to exhibit a polynomial time algorithm to solve the problem. More generally, from the perspective of Computational Complexity, algorithms are useful in helping us to establish whether a given problem belongs to a certain complexity class. This motivates the importance of algorithm design techniques, as well as our tools in analyzing the runtime complexity of an algorithm.

We consider some examples of problems in P.

Example 175. Recall the following decision problem from Example 171, which we refer to as RelPrime.

- Instance: Let $m, n \in \mathbb{N}$.
- Decision: Is $\gcd(m, n) = 1$?

In order to show that $\mathsf{RelPrime} \in \mathsf{P}$, we exhibit a polynomial time algorithm to solve the problem. Recall that we can compute $\gcd(m, n)$ in $O(\log(n))$ time using the Euclidean Algorithm. So we simply compute $\gcd(m, n)$ using the Euclidean algorithm, and check if $\gcd(m, n) = 1$. It follows that $\mathsf{RelPrime} \in \mathsf{P}$.

Example 176. Recall that Path problem from Example 172.

- Instance: Let G be a simple, unweighted graph, and let $u, v \in V(G)$. Let $k \in \mathbb{N}$.
- Decision: Does there exist a path from u to v in G using at most k edges?

We can apply Dijkstra's algorithm to G, starting at vertex u. We return an answer of Yes if the path to v has weight at most k. Otherwise, we return an answer of No. We note that Dijkstra's algorithm runs in time $O(|E| + |V| \log(|V|))$, which is polynomial. Thus, $\mathsf{Path} \in \mathsf{P}$.

We next introduce the complexity class NP.

Definition 177. Let $\Sigma = \{0, 1\}$. A language $L \in \mathsf{NP}$ if there exists a polynomial $p(n)$ (depending only on L) and an algorithm A, such that for every $\omega \in L$, there exists $C \in \Sigma^*$ such that:

- $A(\omega, C) = 1$, and
- A takes at most $p(|\omega|)$ steps.

Here, C is our *certificate*, and A is our *verifier*.

Remark 178. We note that NP stands for **non-deterministic polynomial time**, as NP was originally defined in terms of non-deterministic Turing Machines (which you may talk about in CSCI 3434 Theory of Computation), rather than verifiers. The verifier definition, which we provided, is equivalent to the non-deterministic Turing Machine definition. We note that the verifier definition is the more modern and practical definition for working with NP. For one, the verifier definition provides a very clear outline to show that a language L belongs to NP. Additionally, the verifier definition is useful in other areas of theoretical computer science, such as Interactive Proofs, Probabilistic Checkable Proofs, and Communication Complexity. (If these areas sound interesting to you, then you should take CSCI 3434 Theory of Computation or CSCI 6114 Computational Complexity in the Fall!)

We now seek to unpack the definition of NP. Informally, a decision problem (language) is said to be in NP if we can verify correct answers with some additional supporting work. The certificate C is the additional supporting work. The algorithm A checks or verifies that the input string ω actually belongs to L.

In order to show a problem is in NP, we describe exhibit the certificate and describe a polynomial time verification algorithm. It is helpful to think of the certificate as the object for which we are looking. The verifier then checks if the certificate is indeed valid. Note that we are only concerned with strings belonging to the language (i.e., Yes instances).

Example 179. The Independent Set problem is defined as follows.

- Instance: Let G be a graph, and let $k \in \mathbb{N}$.
- Decision: Does there exist a set of vertices $S \subseteq V(G)$, where $|S| = k$, such that no two vertices in S are adjacent? That is, for all distinct $i, j \in S$, $ij \notin E(G)$.

Suppose G is a graph with an independent set S of k vertices. Here, S serves as our certificate. We provide a polynomial time algorithm to verify that S is an independent set. For each pair of distinct vertices $i, j \in S$, we check that ij is not an edge of G. There $\binom{k}{2} = k(k-1)/2$ such pairs. Note that $k \leq n$ (where $n := |V|$), so there are at most $n(n-1)/2$ pairs to check. Thus, the algorithm takes $O(n^2)$ time to verify that S is an independent set of k vertices.

Example 180. We next consider the Hamiltonian Path problem. Formally, we have:

- Instance: A graph G.
- Decision: Does G have a path that visits every vertex? We refer to such a path as a *Hamiltonian path.*

Suppose G is a graph with a Hamiltonian path. A viable certificate is a sequence of vertices that forms a Hamiltonian path in G. We then check that the consecutive vertices in the sequence are adjacent, and that each vertex in the graph is included precisely once in the sequence. This algorithm takes $O(n)$ time (where $n := |V|$) to verify the certificate, so Hamiltonian Path $\in$ NP.

We now arrive at the P = NP problem. Intuitively, the P = NP problem asks if every decision problem that can be easily verified can also be easily solved. It is straight-forward to show that P $\subseteq$ NP. It remains open as to whether NP $\subseteq$ P.

Proposition 181. P $\subseteq$ NP.

Proof. Let $L \in$ P and let A be a polynomial time algorithm to solve L. We show that $L \in$ NP by constructing a polynomial time verifier A' for L as follows. Let $\omega \in L$, and let 0 be our certificate. On input $\langle \omega, 0 \rangle$, A' runs A on ω ignoring the certificate. A' accepts $\langle \omega, 0 \rangle$ if and only if $A(\omega) = 1$. Since A solves L in polynomial time, A' is thus a polynomial time verifier for L. So $L \in$ NP. □

Remark 182. It is unknown as to whether NP $\subseteq$ P, though it is widely believed that P and NP are different. The hardest problems in NP are largely believed to be computationally intractible; that is, it is largely believed that no efficient algorithms exist to solve these problems. However, it is unknown if this is actually the case. Resolving the P = NP problem is one of the Millenium problems. A correct solution is worth $1 Million.

9.3 NP-completeness

The P = NP problem has been introduced at a superficial level- are problems whose solutions can be verified easily also easy to solve? In some cases, the answer is yes- for the problems in P. In general, this is unknown. However, it is widely believed that P $\neq$ NP. In this section, the notion of NP-completeness will be introduced. NP-complete problems are the hardest problems in NP and are widely believed to be intractible. We begin with the notion of a reduction.

Definition 183 (Polynomial Time Computable Function)**.** A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial time computable function* if some polynomial time TM M exists that halts with just $f(w)$ on the tape when started on w.

Definition 184 (Polynomial Time Reducible)**.** Let $A, B \subseteq \Sigma^*$. We say that A is *polynomial time reducible* to B, which is denoted $A \leq_p B$ if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that $\omega \in A$ if and only if $f(\omega) \in B$.

The notion of reducibility provides an order on computational problems with respect to hardness. That is, suppose A and B are problems such that $A \leq_p B$. Then an algorithm to solve B can be used to solve A. Suppose we have the corresponding polynomial time reduction $f : \Sigma^* \to \Sigma^*$ to reduce A to B. Formally, we take an input $\omega \in \Sigma^*$ and transform it into $f(\omega)$. We use a decider for B to decide if $f(\omega) \in B$. As $\omega \in A$ if and only if $f(\omega) \in B$, we have an algorithm to decide if $\omega \in A$. This brings us to the definition of NP-hard.

Definition 185 (NP-hard)**.** A problem A is NP-hard if for every $L \in \mathsf{NP}$, $L \leq_p A$.

Definition 186 (NP-complete)**.** A language L is NP-complete if $L \in \mathsf{NP}$ and L is NP-hard.

Remark 187. Obseve that every NP-complete problem is a decision problem. In general, NP-hard problems need not be decision problems. Optimization and enumeration problems are common examples of NP-hard problems. Note as well that any two NP-complete languages are polynomial time reducible to each other, and so are equally hard. That is, a solution to one NP-complete language is a solution to all NP-complete languages. This leads to the following result.

Theorem 188. Let L be an NP-complete language. If $L \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.

Proof. Proposition 181 already provides that $\mathsf{P} \subseteq \mathsf{NP}$. So it suffices to show that $\mathsf{NP} \subseteq \mathsf{P}$. Let $L \in \mathsf{NP}$ and let K be an NP-complete language that is also in P. Let $f : \Sigma^* \to \Sigma^*$ be a polynomial time reduction from L to K, and let M be a polynomial time decider for K. Let $\omega \in L$. We transform ω into $f(\omega)$ and run M on $f(\omega)$. From the reduction, we have $\omega \in L$ if and only if $f(w) \in K$ accepts $f(\omega)$. Since M is a decider, we have a polynomial time decider for L. Thus, $L \in \mathsf{P}$ and we have $\mathsf{P} = \mathsf{NP}$. □

In order to show that a language L is NP-complete, it must be shown that $L \in \mathsf{NP}$ and for every language $K \in \mathsf{NP}$, $K \leq L$. Constructing a polynomial-time reductions from each language in NP to the target language L is not easy. However, if we already have an NP-complete problem J, it suffices to show $J \leq_p L$, which shows L is NP-hard. Of course, in order to use this technique, it is necessary to have an NP-complete language with which to begin. The Cook-Levin Theorem provides a nice starting point with the Boolean Satisfiability problem, better known as SAT. There are several variations on the Cook-Levin Theorem. One variation restricts to CNF-SAT, in which the Boolean formulas are in *Conjunctive Normal Form*. We begin with a some definitions.

Definition 189 (Boolean Satisfiability Problem (SAT))**.**

- Instance: Let $\phi : \{0,1\}^n \to \{0,1\}$ be a Boolean function, restricted to the operations of AND, OR, and NOT.
- Decision: Does there exist an input vector $x \in \{0,1\}^n$ such that $\phi(x) = 1$?

Example 190. The Boolean function $\phi(x_1, x_2, x_3) = x_1 \vee x_2 \wedge \overline{x_3}$ is an instance of SAT.

Definition 191 (Clause)**.** A *Clause* is a Boolean function $\phi : \{0,1\}^n \to \{0,1\}$ where ϕ consists of variables or their negations, all added together (where addition is the OR operation).

Definition 192 (Conjunctive Normal Form)**.** A Boolean function $\phi : \{0,1\}^n \to \{0,1\}$ is in *Conjunctive Normal Form* if $\phi = C_1 \wedge C_2 \wedge \ldots C_k$, where each C_i is a clause.

Example 193. The Boolean function $\phi(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3})$ is in Conjunctive Normal Form. Note that $(x_1 \vee x_2)$ is a clause, as is $(x_2 \vee \overline{x_3})$.

Definition 194 (CNF-SAT)**.**

- Instance: A Boolean function $\phi : \{0,1\}^n \to \{0,1\}$ in Conjunctive Normal Form.
- Decision: Does there exist an input vector $x \in \{0,1\}^n$ such that $\phi(x) = 1$?

Theorem 195 (Cook-Levin)**.** CNF-SAT is NP-complete.

Remark 196. The proof of the Cook-Levin theorem is quite involved. We omit the proof here.

We now reduce CNF-SAT to the general SAT problem to show SAT is NP-complete.

Theorem 197. SAT is NP-complete.

Proof.

- **Claim 1:** SAT is in NP.

  Proof. In order to show SAT is in NP, it suffices to exhibit a polynomial time verification algorithm. Let $\phi : \{0,1\}^n \to \{0,1\}$ be a Boolean function with k literals (either a variable or its negation), which we note is the length of ϕ. Let $x \in \{0,1\}^n$ such that $\phi(x) = 1$. We simply evaluate $\phi(x)$, which takes $O(k)$ time. So SAT is in NP. □

- **Claim 2:** SAT is NP-hard.

  Proof. As CNF-SAT is a subset of SAT, the inclusion map $f : \text{CNF-SAT} \to \text{SAT}$ sending $f(\phi) = \phi$ is a polynomial time reduction from CNF-SAT to SAT. So SAT is NP-hard. Thus, SAT is NP-complete. □

□

From CNF-SAT, there is an easy reduction to the Clique problem.

Definition 198 (Clique)**.**

- Instance: Let G be a graph and $k \in \mathbb{N}$.
- Decision: Does G contain a complete subgraph with k vertices?

Theorem 199. Clique is NP-complete.

Proof. We show that Clique is in NP and that Clique is NP-hard.

- **Claim 1:** Clique is in NP.

  Proof. Let (G, k) be an instance of Clique. Let $S \subseteq V(G)$ be a set of vertices that induce a complete subgraph on k vertices. We check that all $\binom{k}{2}$ edges are present in $G[S]$, the subgraph of G induced by S. This takes $O(n^2)$ time, which is polynomial. So Clique is in NP. □

- **Claim 2:** Clique is NP-hard.

  Proof. It suffices to show CNF-SAT $\leq_p$ Clique. Let ϕ be an instance of CNF-SAT with k-clauses. We construct an instance of Clique as follows. Let G be the graph we construct. Each occurrence of a variable in ϕ corresponds to a vertex in G. We add all possible edges except if: (1) two vertices belong to the same clause; or (2) if two vertices are contradictory. If the length of ϕ is n, then this construction takes $O(n^2)$ time which is polynomial time. It suffices to show that ϕ is satisfiable if and only if there exists a k-clique in G.

  Suppose first ϕ is satisfiable. Let x be a satisfying configuration for ϕ. As ϕ is in Conjunctive Normal Form, there exists a literal in each clause that evaluates to 1. We select one such literal from each clause. As none of these literals are contradictory, the corresponding vertices in G induce a k-Clique.

  Conversely, suppose G has a k-clique. Let $S \subseteq V(G)$ be a set of vertices that induce a k-Clique in G. If $v \in S$ corresponds to a variable x_i, then we set $x_i = 1$. Otherwise, v corresponds to a variable's negation and we set $x_i = 0$. Any variable not corresponding to a vertex in the set is set to 0. Recall that each vertex in S corresponds to a literal from each clause and the literals are not contradictory. As ϕ is in Conjunctive Normal Form, we have a satisfying configuration for ϕ. We conclude that Clique is NP-hard. □

□

The Clique problem gives us two additional NP-complete problems. The first problem is the Independent Set problem, and the second is the Subgraph Isomorphism problem. The Subgraph Isomorphism problem is formally:

Definition 200 (Subgraph Isomorphism)**.**

$$L_{SI} = \{\langle G, H\rangle : G, H \text{ are graphs, and } H \subseteq G\}.$$

The inclusion map from Clique to Subgraph Isomorphism provides that Subgraph Isomorphism is NP-hard. It is quite easy to verify that H is a subgraph of G given an isomorphism.

An independent set is the complement of a Clique. Formally, we have the following.

Definition 201 (Independent Set)**.** Let G be a graph. An independent set is a set $S \subseteq V(G)$ such that for any $i, j \in S$, $ij \notin E(G)$. That is, all vertices of S are pairwise non-adjacent in G.

This leads to the Independent Set problem.

Definition 202 (Independent Set (Problem))**.**

- Instance: Let G be a graph and $k \in \mathbb{N}$.
- Decision: Does G have an independent set with k vertices?

Theorem 203. Independent Set is NP-complete.

Proof. We show that Independent Set is in NP, and that Independent Set is NP-hard.

- **Claim 1:** Independent Set is in NP.

  Proof. Let $\langle G, k\rangle$ be an instance of Independent Set and let $S \subseteq V(G)$ be an independent set of order k. S serves as our certificate. We check that for each distinct $i, j \in S$, $ij \notin E(G)$. This check takes $\binom{k}{2}$ steps, which is $O(|V|^2)$ time. So Independent Set is in NP. □

- **Claim 2:** Independent Set is NP-hard.

  Proof. We show Clique $\leq_p$ Independent Set. Let $\langle G, k\rangle$ be an instance of Clique. Let $\overline{G}$ be the complement of G, in which $V(\overline{G}) = V(G)$ and $E(\overline{G}) = \{ij : i, j \in V(G), ij \notin E(G)\}$. This construction takes $O(|V|^2)$ time. So this construction is in polynomial time. As an independent set is the complement of a Clique, G has a k-Clique if and only if $\overline{G}$ has an independent set with k-vertices. So Independent Set is NP-hard. □

□

We provide one more NP-hardness proof to illustrate that not all NP-hard problems are in NP. We introduce the Hamiltonian Cycle and TSP-OPT problems.

Definition 204. Hamiltonian Cycle

- Instance: Let $G(V, E)$ be a graph.
- Decision: Does G contain a cycle visiting every vertex in G?

And the Traveling Salesman optimization problem is defined as follows.

Definition 205. TSP-OPT

- Instance: Let $G(V, E, W)$ be a weighted graph where $W : E \to \mathbb{R}_+$ is the weight function.

- Solution: Find the minimum cost Hamiltonian Cycle in G.

We first note that Hamiltonian Cycle is NP-complete, though we won't prove this. In order to show TSP-OPT is NP-hard, we reduce from Hamiltonian Cycle.

Theorem 206. TSP-OPT is NP-hard.

Proof. We show Hamiltonian Cycle $\leq_p$ TSP-OPT. Let G be a graph with n vertices and a Hamiltonian cycle C. We construct a weighted K_n as follows. Each edge in K_n corresponding to C is weighted 0. All other edges are weighted 1. So any minimum weight Hamiltonian cycle in K_n has weight at least 0. We show that G has a Hamiltonian cycle if and only if the minimum weight Hamiltonian cycle in the K_n has weight 0. Suppose first G has a Hamiltonian cycle C. We trace along C in K_n to obtain a Hamiltonian cycle of weight 0 in K_n. Conversely, suppose K_n has a Hamiltonian cycle of weight 0. By construction, this corresponds to the Hamiltonian cycle C in G. We conclude that Hamiltonian Cycle $\leq_p$ TSP-OPT, so TSP-OPT is NP-hard. □

10 Hash Tables

In this section, we discuss hash tables. Our exposition is adopted from [Fen09].

A *hash table* is a data structure that is used to relate (key, value) pairs. Precisely, a hash table stores an array $A[1, \ldots, m]$. We also have a $h(x)$ that takes an element x and returns a position in the array. We place element x in position $h(x)$. It may be the case that there are two distinct elements x_1, x_2 such that $h(x_1) = h(x_2)$. We refer to such an occurrence as a *collision*. Observe that if we wish to store $n > m$ elements in our hash table, then we will have collisions.

There are two methods to resolve collisions. The first method is called *chaining*, where each element of A is a doubly-linked list with head and tail pointers. When placing element x into position $h(x)$, we prepended x to the start of the linked list stored in $A[h(x)]$. The second method is called *open addressing*, which works by searching for alternative locations in the hash table store element x. Note that in order to keep the runtimes reasonable for the insertion, deletion, and lookup operations, we need to increase the size of A as the number of elements n increases. In this section, we focus exclusively on hash tables that use chaining to resolve collisions.

The worst case runtime for the insertion is $\Theta(1)$, and the worst case for the deletion, and lookup operations for a hash table degenerates to $\Theta(n)$. This occurs when our hash function maps every element to the same cell in our array. Effectively, this is no different than managing a linked list. In practice, hash tables perform much better. In order to achieve better performance, we want a hash function $h(x)$ that is "random-like." Precisely, we say that $h(x)$ satisfies the *Simple Uniform Hashing Assumptions* if $h(x)$ satisfies the following two conditions.

- For any key k and any index $i \in \{1, \ldots, m\}$ $\Pr[h(k) = i] = 1/m$. That is, the probability that $h(x)$ maps the element k into position i is $1/m$.
- For any two keys k_1, k_2 and any index i, $\Pr[h(k_1) = i]$ and $\Pr[h(k_2) = i]$ are independent.

Our goal is to analyze the average-case complexity of a hash table, where the underlying hash function $h(x)$ satisfies the Simple Uniform Hashing Assumption. Note that if $h(x)$ distributes n keys uniformly and independently to different positions in A, then the expected length of a linked list is: $\alpha := n/m$. We call α the *load factor*. Now inserting an element into a linked list takes time $\Theta(1)$. For deletion and lookup, we have to find the element. This breaks down into two cases: whether the given key belongs to the hash table or whether it does not. We first consider the case of an unsuccessful search; that is, when the key is not in the hash table. We assume that our hash function $h(x)$ takes $\Theta(1)$ time to compute.

Lemma 207. Let $\mathcal{H}$ be a hash table with the underlying array $A[1, \ldots, m]$ and hash function $h(x)$ satisfying the Simple Uniform Hashing Assumptions. Suppose that $\mathcal{H}$ resolves collisions with chaining. Now suppose that $k \notin \mathcal{H}$. Then the lookup and deletion operations have expected runtime $\Theta(1 + \alpha)$.

Proof. We leave the details as an exercise. □

We now determine the expected runtime of the deletion and lookup operations if the key belongs to the hash table.

Lemma 208. Let $\mathcal{H}$ be a hash table with the underlying array $A[1, \ldots, m]$ and hash function $h(x)$ satisfying the Simple Uniform Hashing Assumptions. Suppose that $\mathcal{H}$ resolves collisions with chaining. Now suppose that $k \in \mathcal{H}$. Then the lookup and deletion operations have expected runtime $\Theta(1 + \alpha)$.

Proof. Suppose we have keys $k_1, \ldots, k_n$ that are inserted one at a time into $\mathcal{H}$, which is initially empty. As each key is prepended to the start of the corresponding linked list, the number of steps required to search for the key k is 1 plus the number of keys inserted after k into the same position as k (that is, the number of elements in the linked list that appear after k). For $i, j \in \{1, \ldots, n\}$, denote X_{ij} as the event that $h(k_i) = h(k_j)$.

As the deletion operation takes $\Theta(1)$ additional time after finding an element, it suffices to analyze the average runtime of the lookup operation. If $k \in \mathcal{H}$, the expected runtime is:

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

Here, the inner summation

$$\sum_{j=i+1}^{n} X_{ij}$$

denotes the number of elements inserted into the same list as i after i. Now by linearity of expectation, we have that:

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}1+\frac{1}{n}\sum_{i=1}^{n}\frac{n-i}{m} \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{nm}\left(\frac{n(n-1)}{2}\right) \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{n}{2m}-\frac{1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{1}{2m}.
\end{aligned}$$

Thus, the expected runtime of a successful search is $\Theta(1+\alpha)$, as desired. □

Remark 209. Note that if we never resize the underlying array $A[1,\ldots,m]$ in our hash table, then m is constant. So the lookup and deletion operations have expected runtime $\Theta(1+n/m)=\Theta(n)$, which achieves the worst case bound. Our desired complexity bounds will dictate when to resize. For instance, if we want the expected runtime complexity for the lookup and deletion operations to be constant, then it is necessary for $m=\Theta(n)$.

Example 210. Suppose for our complexity goals, we want the load factor $\alpha \leq 1/2$. In this case, we resize our lookup table whenever the load factor is strictly greater than $1/2$. Our goal is to get a sense of when the lookup table resizes. For this example, suppose our lookup table is initialized to have $m=10$ elements.

- Observe that for the first five keys $k_1,\ldots,k_5$, $\alpha \leq 5/10 = 1/2$. When we add k_6, $\alpha = 6/10 > 1/2$. So we double the size of the array. We now have an array of size $m=20$.
- Now when we add keys $k_7,\ldots,k_{10}$, $\alpha \leq 10/20 = 1/2$. When we add k_{11}, $\alpha = 11/20 > 1/2$. So we double the size of the array. We now have an array of size $m=40$.
- When we add keys $k_{12},\ldots,k_{20}$, $\alpha \leq 20/40 = 1/2$. When we add k_{21}, $\alpha = 21/40$. So we double the size of the array. We now have an array of size $m=80$.

A Notation

A.1 Collections

Definition 211. 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 212. 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 213. 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 214. 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 215. 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 216. 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.

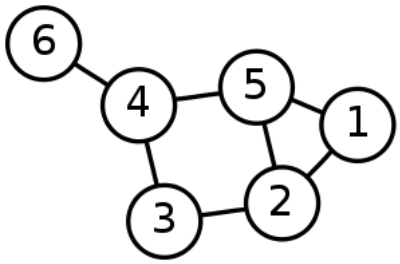
The exposition in this section has benefitted from [Die10, Lev20, Wes00].

B.1 Introduction to Graphs

Definition 217 (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.

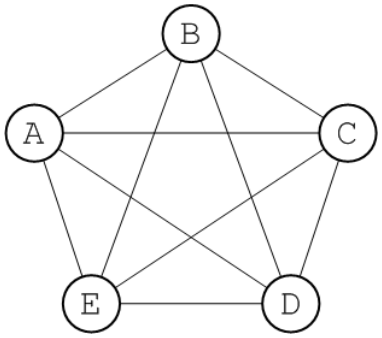
Example 218. Let $G(V, E)$ be the graph where $V = \{1, 2, \ldots, 6\}$ and $E = \{12, 15, 23, 25, 34, 45, 46\}$. This graph is pictured below.



We now introduce several common classes of graphs.

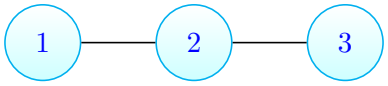
Definition 219 (Complete Graph)**.** The complete graph, denoted K_n, has the vertex set $V = \{1, 2, \ldots, n\}$ and edge set E which consists of **all** two-element subsets of V. That is, K_n has all possible edges between vertices.

Example 220. The complete graph on five vertices K_5 is pictured below.



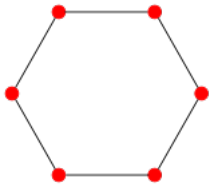
Definition 221 (Path Graph)**.** The path graph, dentoed P_n, has vertex set $V = \{1, 2, \ldots, n\}$ and the edge set $E = \{\{i, i+1\} : 1 \leq i \leq n-1\}$.

Example 222. The path on three vertices P_3 is shown below.



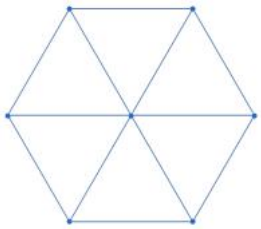
Definition 223 (Cycle Graph)**.** Let $n \geq 3$. The cycle graph, denoted C_n, has the vertex set $V = \{1, 2, \ldots, n\}$ and the edge set $E = \{\{i, i+1\} : 1 \leq i \leq n-1\} \cup \{\{1, n\}\}$.

Example 224. 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.



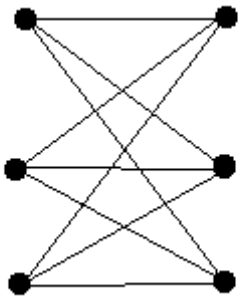
Definition 225 (Wheel Graph)**.** Let $n \geq 4$. The wheel graph, denoted W_n, is constructed by joining a vertex n to each vertex of C_{n-1}. So we take $C_{n-1} \dot{\cup} n$ and add the edges vn for each $v \in [n-1]$.

Example 226. The wheel graph on seven vertices W_7 is pictured below.



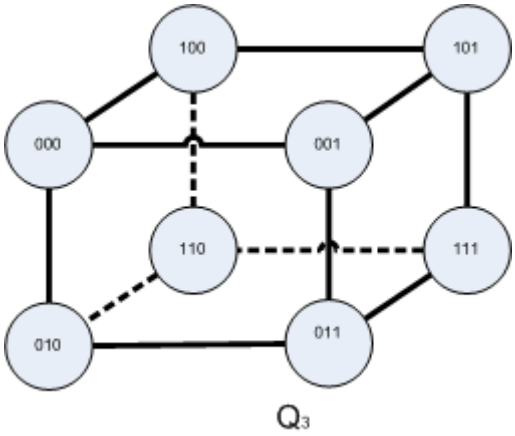
Definition 227 (Bipartite Graph)**.** A bipartite graph $G(V, E)$ has a vertex set $V = X \dot{\cup} Y$, with edge set $E \subseteq \{xy : x \in X, y \in Y\}$. That is, no two vertices in the same part of V are adjacent. So no two vertices in X are adjacent, and no two vertices in Y are adjacent.

Example 228. 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 229 (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 230. 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.



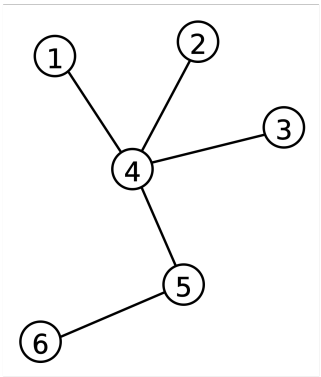
Definition 231 (Connected Graph)**.** A graph $G(V, E)$ is said to be connected if for every $u, v \in V(G)$, there exists a $u-v$ path in G. A graph is said to be *disconnected* if it is not connected; and each connected subgraph is known as a *component*.

Example 232. 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 233 (Tree)**.** A Tree is a connected, acyclic graph.

Example 234. A path is an example of a tree. Additional examples include the binary search tree, the binary heap, and spanning trees of graphs.

Example 235. The following is an example of a tree.



Definition 236 (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 237. Each vertex in the Cycle graph C_n has degree 2. In Example 218, $\deg(6) = 1$ and $\deg(5) = 3$.

Theorem 238 (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 Erdós-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 239 (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 240. 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 241 (Adjacency Matrix)**.** Let $G(V, E)$ be a graph. The adjacency matrix A is an $n \times n$ matrix where:

$$A_{ij} = \begin{cases} 1 & : ij \in E(G) \\ 0 & : ij \notin E(G) \end{cases} \tag{31}$$

Example 242. Consider the adjacency matrix for the graph K_5:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{32}$$

Theorem 243. Let $G(V, E)$ be a graph, and let A be its adjacency matrix. For each $n \in \mathbb{Z}^+$, A_{ij}^n 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_{ij}^m 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_{ij}^k and A_{ij} count the number of $i - j$ walks of length k and 1 respectively. Observe that:

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

So A_{ix}^k 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. □

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 244. 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 245. 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 244, 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.

[CG18] Aaron Clauset and Joshua A. Grochow, *Csci 3104, cu boulder- lecture 7*, January 2018.

[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/`.

[Fen09] Stephen A. Fenner, *Csce 750: Analysis of algorithms course notes*, 2009.

[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/.

[IS10] Hiroshi Imai and Christian Sommer, *Lecture 1- april 8*, 2010.

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

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

[Loe18] Nicholas A. Loehr, *Combinatorics*, CRC Press Taylor & Francis Group, 2018.

[McQ09] William McQuain, *Algorithm analysis i*, 2009.

[Mou16a] Lalla Mouatadid, *Greedy algorithms: Interval scheduling*, 2016, http://www.cs.toronto.edu/ lalla/373s16/notes/ISP.pdf.

[Mou16b] ———, *Network flows: The max flow/min cut theorem.*

[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.