Each edge is assigned a *weight* (or *cost*), denoted $\omega(e)$. It’s a nonnegative real number, shown on each edge above.

The *weight* of a spanning tree is the sum of the weights of its edges.

The spanning tree shown in red above has weight

$$6 + 5 + 10 + 11 + 2 + 20 + 15 = 68.$$ 

We’ll cover two algorithms to find a minimum weight spanning tree in a connected graph, *Prim’s Algorithm* and *Kruskal’s Algorithm*. 
• We’ll build a tree $T$ as a subgraph of $G$.

• Pick a vertex to start at. We’ll use $a$.

• Initialize $T$ to just vertex $a$ (in red) and no edges.
Consider all edges with one vertex in the tree and one vertex not in the tree:

\{a, b\}, \{a, f\}, \{a, e\}

Of those, pick one of smallest weight (if there’s a tie, pick arbitrarily):

\{a, b\}
Prim’s Algorithm

Consider all edges with one vertex in the tree and the other isn’t:
\{a, b\}, \{a, f\}, \{a, e\}

Of those, pick one of smallest weight (break ties arbitrarily):
\{a, b\}

Add it to the tree.

Repeat until all vertices are in the tree.
### Prim’s Algorithm

A weighted graph representation is shown below with nodes labeled from `a` to `h` and edges labeled with their corresponding weights. The graph is depicted with the following structure:

- **Nodes:** `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`
- **Edges and Weights:**
  - `a` to `b` (weight 5)
  - `b` to `c` (weight 11)
  - `c` to `d` (weight 20)
  - `e` to `f` (weight 6)
  - `f` to `g` (weight 12)
  - `g` to `h` (weight 15)

The weight of the minimum spanning tree, as calculated by Prim’s Algorithm, is **5**.
Prim’s Algorithm

Weight: 19
Prim’s Algorithm

Weight: 23
Prim’s Algorithm

Weight: 25
Prim’s Algorithm

Weight: 33
Prim’s Algorithm

Weight: 48
Greedy Algorithms

- Prim’s Algorithm (and the next two algorithms coming up) is a *greedy algorithm*.

- We use the optimal *local solution* for each step.

- We hope this gives the optimal *global solution* at the end, but greedy algorithms aren’t guaranteed to work; it has to be proved for each specific algorithm.

**Greedy algorithm for driving in rush hour**

- At each intersection or fork, take the option (straight or a turn) that looks fastest and goes towards your destination.

- The overall time may be worse, since you could hit a traffic jam, lower speed limits, etc. in that direction.

- A GPS navigation system with real-time traffic information for the whole city considers all routes instead of just this one step.
Proof of Prim’s Algorithm

Claim 1: At each stage of Prim’s Algorithm, $T$ is a tree.

- By the way we added edges (one vertex in $T$, the other vertex not in $T$), $T$ is connected and doesn’t have any cycles, so it’s a tree.

Claim 2: At each stage of Prim’s Algorithm, $T$ is contained in a minimum weight spanning tree of $G$.

- We will prove this by induction on $m = |E(T)|$. 
Proof of Prim’s Algorithm

Claim
At each stage of Prim’s Algorithm, $T$ is contained in a minimum weight spanning tree of $G$.

Proof (base case):
- We prove this by induction on $m = |E(T)|$.
- When we start, $T$ has one vertex and no edges, so it’s contained in every minimum spanning tree.
- Since the graph is finite, there is a finite number of spanning trees (even if it’s large), so there is one achieving a minimum weight.
Proof of Prim’s Algorithm

**Claim**

At each stage of Prim’s Algorithm, $T$ is contained in a minimum weight spanning tree of $G$.

**Proof (induction step):**

- Tree $T$ has $m$ edges ($0 \leq m < |E(G)|$), and by the induction hypothesis, it’s contained in a minimum weight spanning tree $S$.

- We select an edge $e$ to add to $T$, to yield $T \cup \{e\}$ with $m + 1$ edges.

**Case 1:** If $T \cup \{e\} \subset S$, we’re done.

**Case 2:** If $T \cup \{e\} \not\subset S$, we will show how to construct a different minimum weight spanning tree containing $T \cup \{e\}$. 

Proof of Prim’s Algorithm

Claim
At each stage of Prim’s Algorithm, $T$ is contained in a minimum weight spanning tree of $G$.

Proof (induction step, Case 2):

- $S$ is a spanning tree and $e$ isn’t in $S$, so $S \cup \{e\}$ has a cycle $C$.

- $C - e$ is a path in $S$ with one end in $T$ and the other end not in $T$.

- So, somewhere along path $C - e$, there’s at least one edge $f$ that connects a vertex in $T$ to a vertex not in $T$.

- Since $e$ is a minimum weight edge connecting a vertex in $T$ to a vertex not in $T$, we must have $\omega(e) \leq \omega(f)$.
Proof of Prim’s Algorithm

Claim
At each stage of Prim’s Algorithm, $T$ is contained in a minimum weight spanning tree of $G$.

Proof (induction step, Case 2, continued):

- Let $S' = (S \cup \{e\}) - \{f\}$.
  - We create a cycle by adding $e$, and break the cycle by removing $f$.
  - So $S'$ is also a spanning tree of $G$ (not necessarily min. weight), and $S'$ contains $T \cup \{e\}$.

- $\omega(S') = \omega(S) + \omega(e) - \omega(f)$, and $\omega(S') \leq \omega(S)$ since $\omega(e) \leq \omega(f)$.

- But $S$ is a minimum weight spanning tree, so $\omega(S) \leq \omega(S')$.

- Thus, $\omega(S) = \omega(S')$, so $S'$ is also a minimum weight spanning tree of $G$, and it contains $T \cup \{e\}$. The claim has been proved.
We’ll build a forest $F$ (red) as a subgraph of $G$.

Initialize $F$ to all of the vertices of $G$, but no edges.
Kruskal’s Algorithm

Pick an edge of min. weight connecting different components of $F$: \{c, g\}.

Add the edge to $F$.

Repeat until there are no edges connecting different components.

At each step, $F$ is a forest.

If $G$ is connected: The final result is a minimum weight spanning tree (similar proof to Prim’s Algorithm).

If $G$ isn’t connected: The final result is a forest with a minimum weight spanning tree of each connected component.
Kruskal’s Algorithm

Weight: 2
Kruskal’s Algorithm

Weight: 6
Kruskal’s Algorithm

Weight: 11
Two edges are candidates, \{e, f\} and \{g, h\}, both with weight 8. Pick either one.
Kruskal’s Algorithm

Weight: 25
The edges with vertices in different components are \( \{c, d\} \) (weight 20) and \( \{d, h\} \) (weight 15).

There are lower weight edges not in \( F \), but adding any of them would form a cycle.
Kruskal’s Algorithm

Weight: 48
Dijkstra’s Algorithm

Pick a vertex; we’ll use $a$.

We want to find minimum weight paths from $a$ to all vertices.

Minimum length paths (in # edges) corresponds to all edges having weight 1. We already solved that using breadth first search.
Create two arrays concerning the best path (smallest weight) we have found so far from $a$ to $v$, for each vertex $v$:

- $\text{dist}(v)$ is its weight.
  The only path we know so far is $a$ to itself (one vertex, length 0). Initialize to $\text{dist}(a) = 0$ and $\text{dist}(v) = \infty$ for all other vertices.

- $\text{prev}(v) = u$ means that its last edge is $\{u, v\}$.
  Initialize to $\text{prev}(v) = \text{undefined}$ for all vertices.
  We will see that $\text{prev}()$ encodes a tree.

- Also initialize $W = V(G)$. We will examine vertices one at a time.
Dijkstra’s Algorithm

Pick any vertex \( u \in W \) with \( \text{dist}(u) \) minimum. Here, \( u = a \).

Remove \( u \) from \( W \).

For each neighbor \( v \in N_G(u) \):

- Consider the best path from \( a \) to \( u \) so far, plus edge \( \{u, v\} \). This goes from \( a \) to \( v \).
- It has weight \( \text{dist}(u) + \omega(\{u, v\}) \).
- If this is smaller than \( \text{dist}(v) \), then set \( \text{dist}(v) = \text{dist}(u) + \omega(\{u, v\}) \) and \( \text{prev}(v) = u \).

Repeat the above steps until \( W \) is emptied.
### Dijkstra’s Algorithm

#### Weighted Graph $W = \{a, b, c, d, e, f, g, h\}$

<table>
<thead>
<tr>
<th>$v$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{dist}(v)$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\text{prev}(v)$</td>
<td>$-\cdot$</td>
<td>$-\cdot$</td>
<td>$-\cdot$</td>
<td>$-\cdot$</td>
<td>$-\cdot$</td>
<td>$-\cdot$</td>
<td>$-\cdot$</td>
<td>$-\cdot$</td>
</tr>
</tbody>
</table>

**Vertex:**
- $u = a$

**Neighbors:**
- $b, f$

**Paths through $u$ to neighbors:**
- **To $b$:** $0 + 5 = 5 < \infty$, so set $\text{dist}(b) = 5$ and $\text{prev}(b) = a$.
- **To $e$:** $0 + 6 = 6 < \infty$, so set $\text{dist}(e) = 6$ and $\text{prev}(e) = a$. 
### Dijkstra’s Algorithm

**Vertex:**
- \( u = b \)

**Neighbors:**
- \( a, c, f, g \)

**Paths through \( u \) to neighbors:**
- **To \( a \):** \( 5 + 5 = 10 > 0 \), so it’s not better than the prior best path to \( a \). Also, it goes back to \( u \)’s parent \( (a - b - a) \), so it’s not a path. We can always skip \( u \)’s parent; it never improves the prior best path.
- **To \( c \):** \( 5 + 11 = 16 < \infty \), so set \( \text{dist}(c) = 16 \) and \( \text{prev}(c) = b \).
- **To \( f \):** \( 5 + 10 = 15 > 12 \), so it’s not better.
- **To \( g \):** \( 5 + 12 = 17 < \infty \), so set \( \text{dist}(g) = 17 \) and \( \text{prev}(e) = b \).
### Dijkstra’s Algorithm

#### Vertex:
- \( u = e \)

#### Neighbors:
- \( a, f \)

#### Paths through \( u \) to neighbors:
- **To \( f \):** \( 6 + 8 = 14 > 12 \), so it’s not better.

---

<table>
<thead>
<tr>
<th>( v )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
<th>( g )</th>
<th>( h )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{dist}(v) )</td>
<td>0</td>
<td>5</td>
<td>16</td>
<td>( \infty )</td>
<td>6</td>
<td>12</td>
<td>17</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( \text{prev}(v) )</td>
<td>( - )</td>
<td>( a )</td>
<td>( b )</td>
<td>( - )</td>
<td>( a )</td>
<td>( a )</td>
<td>( b )</td>
<td>( - )</td>
</tr>
</tbody>
</table>

\[ W = \{ c, d, e, f, g, h \} \]
Dijkstra’s Algorithm

Vertex:
- \( u = f \)

Neighbors:
- \( a, b, e, g \)

Paths through \( u \) to neighbors:
- **To \( b \):** \( 12 + 10 = 22 > 5 \), so it’s not better.
- **To \( e \):** \( 12 + 8 = 20 > 6 \), so it’s not better.
- **To \( g \):** \( 12 + 4 = 16 < 17 \), so it’s better than the prior path!
  
  Set \( \text{dist}(g) = 16 \) and \( \text{prev}(g) = f \).
  
  This replaces the edge we previously had to \( g \).
Dijkstra’s Algorithm

Vertex: c and g are tied! We’ll use \( u = c \).

Neighbors:

Paths through \( u \) to neighbors:

- **To d:** \( 16 + 20 = 36 < \infty \), so set \( \text{dist}(d) = 36 \) and \( \text{prev}(d) = c \).
- **To g:** \( 16 + 2 = 18 > 16 \), so it’s not better.
- **To h:** \( 16 + 10 = 26 < \infty \), so set \( \text{dist}(h) = 26 \) and \( \text{prev}(h) = c \).
Dijkstra’s Algorithm

Vertex:

Neighbors:

Paths through $u$ to neighbors:

- **To $b$:** $16 + 12 = 28 > 5$, so it’s not better.
- **To $c$:** $16 + 2 = 18 > 16$, so it’s not better.
- **To $h$:** $16 + 8 = 24 < 26$, so it’s better than the prior path!
  Set $\text{dist}(h) = 24$ and $\text{prev}(h) = g$.
  This replaces the edge we previously had to $h$. 

$$W = \{d, g, h\}$$
Dijkstra’s Algorithm

**Vertex:**
- \( u = h \)

**Neighbors:**
- \( c, d, g \)

**Paths through \( u \) to neighbors:**
- **To \( c \):** \( 24 + 10 = 34 \) > 16, so it’s not better.
- **To \( d \):** \( 24 + 15 = 39 \) > 36, so it’s not better.

---

**Graph and Table:**

- **Graph:** A weighted graph with vertices \( a, b, c, d, e, f, g, h \) and weights on the edges.
- **Table:**
  
  \[
  \begin{array}{cccccccc}
    v & a & b & c & d & e & f & g & h \\
    \text{dist}(v) & 0 & 5 & 16 & 36 & 6 & 12 & 16 & 24 \\
    \text{prev}(v) & - & a & b & c & a & a & f & g \\
  \end{array}
  \]

  - \( W = \{d, h\} \)
Vertex: $u = d$

Neighbors: $v, h$

Paths through $u$ to neighbors:

To $h$: $36 + 15 = 51 > 24$, so it’s not better.
Dijkstra’s Algorithm

We are done!

For a minimum weight path from $a$ to $h$, work backwards from $h$ by iterating $\text{prev}(v)$ until we reach $a$:

- $\text{prev}(h) = g$
- $\text{prev}(g) = f$
- $\text{prev}(f) = a$

Put these in the forwards direction: $a, f, g, h$

It works the same for minimum weight paths from $a$ to any vertex.
There are also algorithms beyond what we’re covering, including:

- The *Bellman-Ford Algorithm* allows for negative edge weights (as long as that doesn’t result in any cycles of negative weight).

- The *Floyd-Warshall Algorithm* allows for negative edge weights (same restriction) and it simultaneously computes the minimum distance between all pairs of vertices.