**Minimum Spanning Tree Problem:** Given a graph $G = (V,E)$ with edge weights $w: E \to \mathbb{R}_{\geq 0}$ find a minimum weight subset of the edges that connects all the vertices.

The weight of a set $F \subseteq E$ is $\sum \{w(e) : e \in F\}$

**Examples**

Assuming the graph is connected, the edge subset will be a tree, called the minimum spanning tree.

There are several possible greedy approaches, with different implementation challenges:
- add minimum weight edge first, never build a cycle. Kruskal’s algorithm
- grow a connected graph from one vertex. Prim’s algorithm. **TODAY**
- throw away heavy edges, never disconnect.
**Prim’s MST algorithm:** Grow one connected component in a greedy fashion, i.e. by adding the minimum weight edge leaving the component.

**Correctness.** The exact same exchange argument works. In fact, we can prove one lemma that gives correctness of both algorithms (see the text, CLRS).
Implementing and analyzing Prim’s algorithm \( |V| = n, |E| = m \)

Prim’s algorithm

initialize \( C := \{s\}; \ T := \emptyset \)
while \( C \neq V \)
    find the min weight edge \( e = (u,v) \) with \( u \in C, v \in V - C \)
    \( T := T \cup \{ e \} \)
    \( C := C \cup \{ v \} \)

In general, we need to find the min weight edge leaving \( C \).

**Priority Queue.** Data structure to maintain a set of weighted elements with the operations:

- Find and delete the min. weight element
- Insert
- Delete

Can be implemented as a heap (see CS 240) with \( O(\log k) \) time per operation, \( k = \#\text{elements} \).

In our case, the current set of elements is \( \delta(C) = \text{edges leaving } C \). Then \( k \leq m \).

We must:

- find the min weight edge \( e \in \delta(C) \). This is Find-min.
- update \( \delta(C) \) when \( v \) is added to \( C \)
Implementing and analyzing Prim’s algorithm

$|V| = n$, $|E| = m$

How to update $\delta(C)$ when $v$ is added to $C$
- edges between $C$ and $v$ leave $\delta(C)$. Use Delete operations
- the other edges incident to $v$ join $\delta(C)$. Use Insert operations.

We can find these edges by going through $v$’s adjacency list.

**Note:** each edge enters $\delta(C)$ once and leaves $\delta(C)$ once.

<table>
<thead>
<tr>
<th># of Priority Queue operations:</th>
<th>Total runtime:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find-min</td>
<td>$O(n \log m) + O(m \log m)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$O(m \log n)$ assuming $m \leq n$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(\log m)$</td>
</tr>
</tbody>
</table>

Time per operation: $O(\log m)$

It is slightly more efficient to keep a priority Queue of vertices $V - C$, with $w(v) = \min$ weight of edge from $C$ to $v$. Then size of Queue $= k \leq n$.

Update is a Decrease-Key operation, $O(\log n)$. Still gives $O(m \log n)$ total.

A Fibonacci Heap (fancy Priority Queue) gives $O(n \log n + m)$.
Summary of Lecture 16, part 1

- Prim’s algorithm for Minimum Spanning Tree

What you should know from Lecture 16, part 1:

- what is Prim’s algorithm, how to implement, runtime

Next:

- Shortest paths
Shortest Paths in Edge Weighted Graphs

Recall that BFS from vertex v finds shortest paths from v to all vertices in an unweighted undirected graph.

**General input:** directed or undirected graph with weights on the edges.

**Example**

Note 1: Does a Minimum Spanning Tree always contain the shortest paths? No always

Note 2: Can we reduce to unweighted graphs and use BFS?

We will study several shortest path algorithms (directed/undirected, single source/all pairs). Non-negative/arbitrary edge weights.

**Today:** Dijkstra’s algorithm.
Dijkstra’s Shortest Path Algorithm, 1959.

**Input:** Graph or digraph $G = (V,E)$ with edge weights $w: E \rightarrow \mathbb{R}^{\geq 0}$ and a vertex $s$.

**Output:** Shortest paths from $s$ to every vertex in $V$.

**Note:** A **shortest** path is a path of minimum weight and the **weight** of a path $P \subseteq E$ is $\sum \{w(e) : e \in P\}$.

How do we give the output? as a **tree** of shortest paths.

**Idea of the algorithm:** Grow the tree of shortest paths starting from vertex $s$. 
Dijkstra’s Shortest Path Algorithm, 1959.

Idea of the algorithm: Grow the tree of shortest paths starting from vertex s.

General step: have the tree T of shortest paths to all vertices in set B. Initially B = {s}.
Choose edge e = (x,y) with x ∈ B, y ∉ B to minimize
\[ d(s,x) + w(x,y). \]
Call this d.

Then: \[ d(s,y) := d \]
add (x,y) to T, i.e. parent(y) := x
add y to B

This is greedy in the sense that we always add the vertex with the next min distance from s.

Claim: d is the min distance from s to y.
Dijkstra’s Shortest Path Algorithm, 1959.

Claim. d is the min distance from s to y. (Assuming d(s,x) is correct for all x in B.)
Note: This justifies the output being a tree.

Proof. Any path from s to y has weight \( \geq d \).

Consider any paths to y. Break up it into

\[ \Pi \]

- initial part in B
  \[ s \to u \]
- first edge of it leaving B
  \[ e = (u, v) \]
- rest of it
  \[ \Pi_2 \]

\[ w(\Pi) = w(\Pi_1) + w(e) + w(\Pi_2) \]
\[ \geq w(\Pi_1) + w(e) \] because edge weight are \( \geq 0 \) so \( w(\Pi_2) \geq 0 \)
\[ \geq d(s, u) + w(e) \geq d \] because \( d \) was chosen as min.

Therefore, by induction on I(B) the algorithm correctly finds d(s,v) for all vertices v.
A shortest path from s to v is recovered by following parent pointers from v.
Implementing and analyzing Dijkstra’s algorithm

Recall the plan:
Choose edge \( e = (x,y) \) with \( x \in B, \ y \notin B \) to minimize
\[ d(s,x) + w(x,y). \]

As for Prim’s algorithm, we can use a Priority Queue (a heap) whose elements are edges \( e = (x,y) \) with \( x \in B, \ y \notin B \) with value(e) = \( d(s,x) + w(x,y) \).

More efficient: a heap of vertices \( y \notin B \), where value of \( y \) is the “tentative distance” \( d(y) = \min \text{ weight path from } s \text{ to } y \) with all but the last edge in \( B \).

Dijkstra’s algorithm

initialize: \( d(v) := \infty \ \forall \ v; \ d(s) := 0; \ B := \emptyset; \) set up heap on \( V \)
while \( B \neq V \)

\[ y := \text{vertex of } V - B \text{ of min } d(y) \]

#Find-min (and delete it)

for each edge (y,z) do

\[ \text{if } d(y) + w(y,z) < d(z) \text{ then} \]

\[ d(z) := d(y) + w(y,z) \text{ and update the heap} \]

parent(z) := y

\[ \text{add } y \text{ to } B. \]
Implementing and analyzing Dijkstra’s algorithm

Runtime Analysis

Dijkstra’s algorithm

\[
\text{initialize: } \ d(v) := \infty \ \forall \ v; \ d(s) := 0; \ B := \emptyset; \ \text{set up heap on } V \\
\text{while } B \neq V \\
\quad y := \text{vertex of } V - B \text{ of min } d(y) \quad \# \text{Find-min (and delete it)} \\
\quad \text{for each edge } (y,z) \text{ do} \\
\quad \quad \text{if } d(y) + w(y,z) < d(z) \text{ then} \\
\quad \quad \quad d(z) := d(y) + w(y,z) \text{ and update the heap} \\
\quad \quad \text{parent}(z) := y
\]

Heap has \(O(n)\) elements. Find-min, Insert, Delete take \(O(\log n)\) time.

How to update the heap: The value (key) of \(z\) changes. Can Delete and re-Insert \(z\). Or use Decrease-Key operation. Either way, \(O(\log n)\).

# of Priority Queue operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>(n)</th>
<th>(m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find-min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total runtime:

\[
\mathcal{O}(n \log n) + \mathcal{O}(m \log n) = \mathcal{O}(m \log n) \text{ assuming } m \geq n.
\]
Edsger W. Dijkstra was known for many contributions to computer science, e.g. structured programming, concurrent programming. He designed the above shortest path algorithm to demonstrate the capabilities of a new computer (he found railway journeys in the Netherlands as the demonstration).

At that time (the 1950’s) the result was not considered important.

Dijkstra wrote:

At the time, algorithms were hardly considered a scientific topic. I wouldn’t have known where to publish it. . . . The mathematical culture of the day was very much identified with the continuum and infinity. Could a finite discrete problem be of any interest? The number of paths from here to there on a finite graph is finite; each path is a finite length; you must search for the minimum of a finite set. Any finite set has a minimum – next problem, please. It was not considered mathematically respectable. . .
Summary of Lecture 16

- Prim’s MST algorithm
- Dijkstra’s shortest path algorithm

What you should know from Lecture 16:

- what is Prim’s Min Spanning Tree algorithm, implementation, runtime
- what is Dijkstra’s algorithm, implementation, runtime
- conditions for Dijkstra to work
- similarities and differences Prim vs Dijkstra

Next:

- more algorithms for shortest paths in graphs