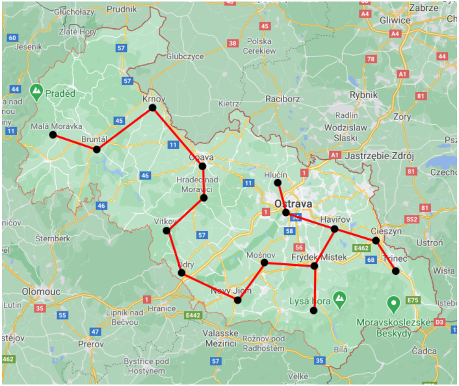
Minimum Spanning Trees I: Prim's algorithm
COMS20010 2020, Video 5-3

John Lapinskas, University of Bristol

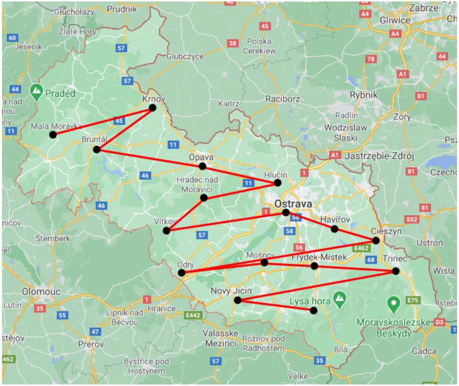
Say you're trying to build a regional power grid for Moravia, like Otakar Borůvka in 1926.



You need every town to be connected to every other town, and you want to spend as little as possible. So you want something like **this**,

Motivation

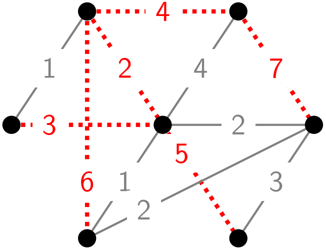
Say you're trying to build a regional power grid for Moravia, like Otakar Borůvka in 1926.



You need every town to be connected to every other town, and you want to spend as little as possible. So you want something like this, not like **this**.

Formal definition

We think of this situation as a connected weighted graph $G = ((V, E), w)$: the vertices are towns, and $w(x, y)$ is the cost of building a connection from x to y. (In this case, E would contain every possible edge.)



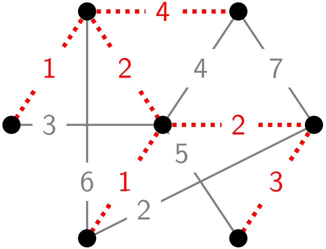
Total weight:

$4 + 2 + 7 + 3 + 6 + 5 = 27$

In other words, we seek a subtree T of G with $V(T) = V$ (a **spanning tree**)... whose total weight $\sum_{e \in E(T)} w(e)$ is as small as possible.

Formal definition

We think of this situation as a connected weighted graph $G = ((V, E), w)$: the vertices are towns, and $w(x, y)$ is the cost of building a connection from x to y. (In this case, E would contain every possible edge.)



Total weight:

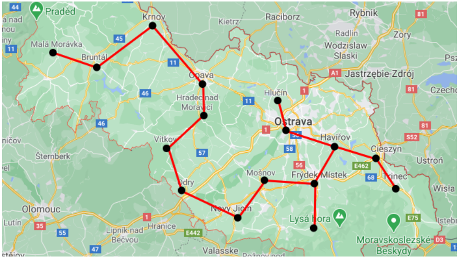
$4 + 1 + 2 + 2 + 1 + 3 = 13$

In other words, we seek a subtree T of G with $V(T) = V$ (a **spanning tree**)... whose total weight $\sum_{e \in E(T)} w(e)$ is as small as possible.

This is called a **minimum spanning tree**.

Wait a second...

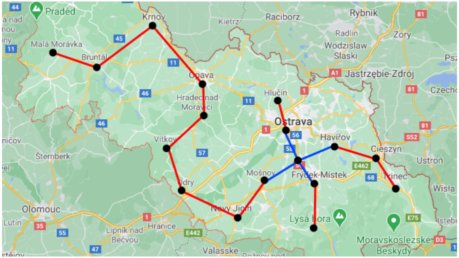
Strictly speaking, this might not be the **best** possible solution.



What if we could introduce new vertices?

Wait a second...

Strictly speaking, this might not be the **best** possible solution.



What if we could introduce new vertices?

This version of the problem is called **minimum Steiner tree**. But:

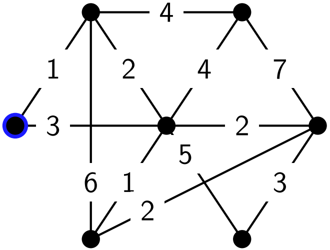
- this is “NP-hard” (read: no polynomial-time algorithm);
- all the approximation algorithms are based on minimum spanning tree;
- using a minimum spanning tree is already “good enough” — at worst twice the weight of a minimum Steiner tree (see problem sheet).

Prim's algorithm: The idea

Input: A connected weighted graph $G = ((V, E), w)$. **Output:** A minimum spanning tree of G.

A **minimum spanning tree** is a subtree T of G covering all of G's vertices, whose total weight $\sum_{e \in E(T)} w(e)$ is as small as possible.

We work **greedily**: pick an arbitrary start vertex, then grow it into a spanning tree by always choosing one of the cheapest available edges.

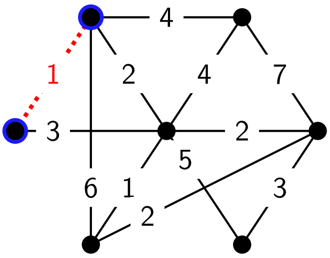


Prim's algorithm: The idea

Input: A connected weighted graph $G = ((V, E), w)$. **Output:** A minimum spanning tree of G.

A **minimum spanning tree** is a subtree T of G covering all of G's vertices, whose total weight $\sum_{e \in E(T)} w(e)$ is as small as possible.

We work **greedily**: pick an arbitrary start vertex, then grow it into a spanning tree by always choosing one of the cheapest available edges.

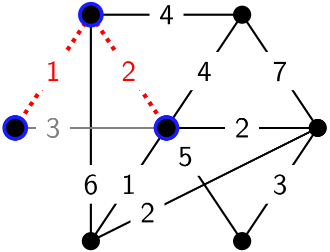


Prim's algorithm: The idea

Input: A connected weighted graph $G = ((V, E), w)$. **Output:** A minimum spanning tree of G.

A **minimum spanning tree** is a subtree T of G covering all of G's vertices, whose total weight $\sum_{e \in E(T)} w(e)$ is as small as possible.

We work **greedily**: pick an arbitrary start vertex, then grow it into a spanning tree by always choosing one of the cheapest available edges.

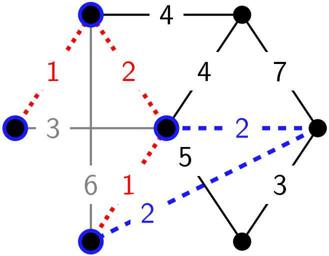


Prim's algorithm: The idea

Input: A connected weighted graph $G = ((V, E), w)$. **Output:** A minimum spanning tree of G.

A **minimum spanning tree** is a subtree T of G covering all of G's vertices, whose total weight $\sum_{e \in E(T)} w(e)$ is as small as possible.

We work **greedily**: pick an arbitrary start vertex, then grow it into a spanning tree by always choosing one of the cheapest available edges.



When there's a tie, we break it arbitrarily.

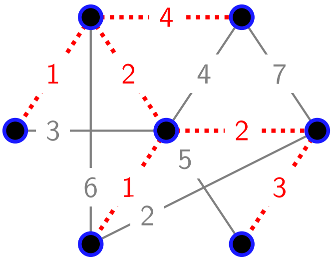
(The choice will only affect *which* minimum spanning tree we get.)

Prim's algorithm: The idea

Input: A connected weighted graph $G = ((V, E), w)$. **Output:** A minimum spanning tree of G.

A **minimum spanning tree** is a subtree T of G covering all of G's vertices, whose total weight $\sum_{e \in E(T)} w(e)$ is as small as possible.

We work **greedily**: pick an arbitrary start vertex, then grow it into a spanning tree by always choosing one of the cheapest available edges.



When there's a tie, we break it arbitrarily.

(The choice will only affect *which* minimum spanning tree we get.)

Prim's algorithm: Formal version and correctness

Input: A connected weighted graph $G = ((V, E), w)$. **Output:** A minimum spanning tree of G.

A **minimum spanning tree** is a subtree T of G covering all of G's vertices, whose total weight $\sum_{e \in E(T)} w(e)$ is as small as possible.

Formally: Let $T_1 = (\{v\}, \emptyset)$ for some arbitrary $v \in V$.

Let E_i be the set of edges from $V(T_i)$ to $V \setminus V(T_i)$.

Form T_{i+1} by adding a lowest-weight edge $e_i \in E_i$ to T_i, so

$$V(T_{i+1}) = V(T_i) \cup e_i \text{ and } E(T_{i+1}) = E(T_i) \cup \{e_i\}.$$

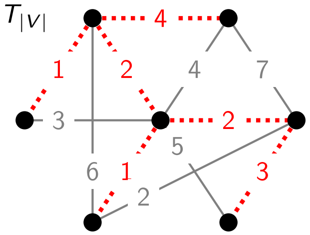
Prim's algorithm is to calculate and return $T_{|V|}$. Why does this work?

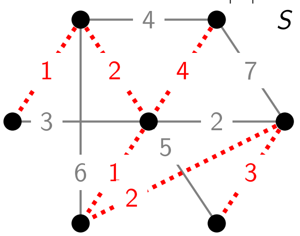
It returns a spanning tree because it's basically breadth-first search!
We just pick a lowest-weight edge at each stage rather than using a queue.

To prove it's a **minimum** spanning tree, we use an exchange argument.

That is, we show we can turn any minimum spanning tree into $T_{|V|}$ without increasing its weight (like with interval scheduling).

$T_{|V|}$ **is minimum:** Let S be a minimum spanning tree with $S \neq T_{|V|}$.



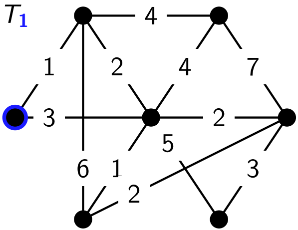


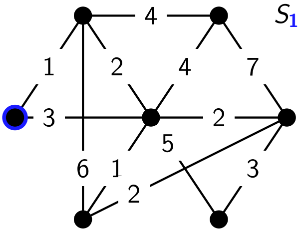
Let $S_i = S[V(T_i)]$, and let $I = \min\{i \colon S_i \neq T_i\}$.

We have $S_1 = T_1$ and $S_{|V|} \neq T_{|V|}$, so $2 \leq I \leq |V|$.

Prim's algorithm: Correctness II

$T_{|V|}$ **is minimum:** Let S be a minimum spanning tree with $S \neq T_{|V|}$.



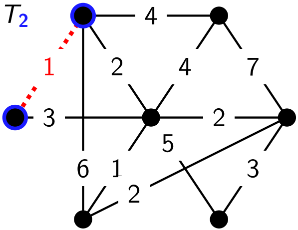


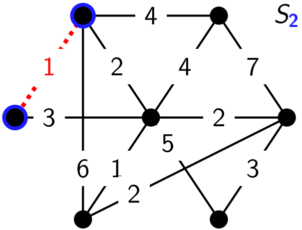
Let $S_i = S[V(T_i)]$, and let $I = \min\{i \colon S_i \neq T_i\}$.

We have $S_1 = T_1$ and $S_{|V|} \neq T_{|V|}$, so $2 \leq I \leq |V|$.

Prim's algorithm: Correctness II

$T_{|V|}$ **is minimum:** Let S be a minimum spanning tree with $S \neq T_{|V|}$.



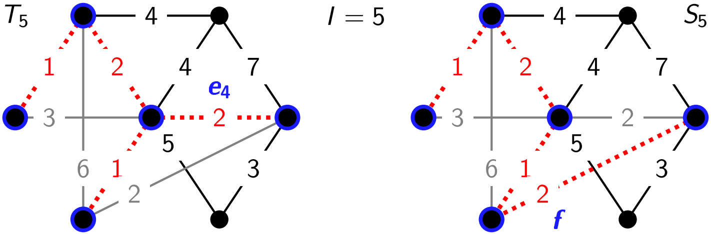


Let $S_i = S[V(T_i)]$, and let $I = \min\{i : S_i \neq T_i\}$.

We have $S_1 = T_1$ and $S_{|V|} \neq T_{|V|}$, so $2 \leq I \leq |V|$.

Prim's algorithm: Correctness II

$T_{|V|}$ **is minimum:** Let S be a minimum spanning tree with $S \neq T_{|V|}$.



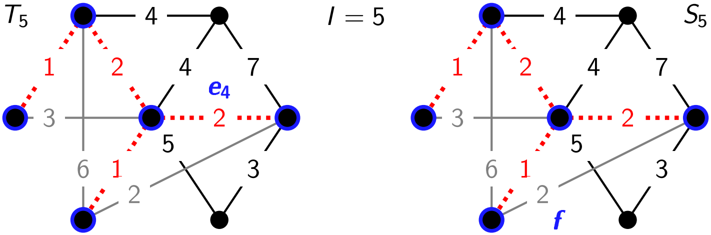
Let $S_i = S[V(T_i)]$, and let $I = \min\{i\colon S_i \neq T_i\}$.

We have $S_1 = T_1$ and $S_{|V|} \neq T_{|V|}$, so $2 \leq I \leq |V|$.

$S_{I-1} = T_{I-1}$ is a tree, so there is only one edge f from $V(S_{I-1})$ to $V \setminus V(S_{I-1})$ in S_I (or S_I would contain a cycle). Remove it and replace it with e_{I-1}.

Prim's algorithm: Correctness II

$T_{|V|}$ **is minimum:** Let S be a minimum spanning tree with $S \neq T_{|V|}$.



Let $S_i = S[V(T_i)]$, and let $l = \min\{i \colon S_i \neq T_i\}$.

We have $S_1 = T_1$ and $S_{|V|} \neq T_{|V|}$, so $2 \leq l \leq |V|$.

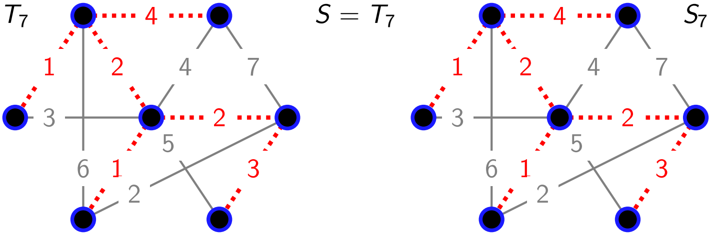
$S_{l-1} = T_{l-1}$ is a tree, so there is only one edge f from $V(S_{l-1})$ to $V \setminus V(S_{l-1})$ in S_l (or S_l would contain a cycle). Remove it and replace it with e_{l-1}.

Weight doesn't increase: True by Prim's choice of e_{l-1}. ✓

Still a tree: Since there is only one edge f, $S[V \setminus V(S_{l-1})]$ is a tree as well (by the FLoT). Joining two disjoint trees by an edge gives another tree (by the FLoT). ✓

Prim's algorithm: Correctness II

$T_{|V|}$ **is minimum:** Let S be a minimum spanning tree with $S \neq T_{|V|}$.



Let $S_i = S[V(T_i)]$, and let $I = \min\{i \colon S_i \neq T_i\}$.

We have $S_1 = T_1$ and $S_{|V|} \neq T_{|V|}$, so $2 \leq I \leq |V|$.

$S_{I-1} = T_{I-1}$ is a tree, so there is only one edge f from $V(S_{I-1})$ to $V \setminus V(S_{I-1})$ in S_I (or S_I would contain a cycle). Remove it and replace it with e_{I-1}.

Weight doesn't increase: ✓ **Still a tree:** ✓

So S is now a spanning tree which is "one edge closer" to $T_{|V|}$.

By repeating the process, we can turn S **into** $T_{|V|}$ without increasing its weight. Hence $w(S) \geq w(T_{|V|})$. Since S was minimum, we're done! $\square$

Prim's algorithm: Implementation

Literally just breadth-first search with a priority queue!

Algorithm: BFS

Input : **Connected weighted** graph $G = ((V, E), w)$.
Output : **A minimum spanning tree for G**.

Number the vertices of G **arbitrarily** as $v_1, \dots, v_n$.
Let $\mathtt{L}[i] \leftarrow \infty$ for all $i \in [n]$.
Let $\mathtt{L}[1] \leftarrow 0$, $\mathtt{pred}[1] \leftarrow \mathtt{None}$.
Let `queue` be a **length-$|E|$ priority** queue containing all tuples (v, v_j) with $\{v, v_j\} \in E$,
  using their edge weights as priorities.
while `queue` *is not empty* **do**
  Remove front tuple (v_i, v_j) from `queue`.
  if $\mathtt{L}[j] = \infty$ **then**
    Add (v_j, v_k) to `queue` for all $\{v_j, v_k\} \in E$, $k \neq i$.
    Set $\mathtt{L}[j] \leftarrow \mathtt{L}[i] + 1$, $\mathtt{pred}[j] = i$.
Return `pred`.

Time analysis: As with breadth-first search, each edge is only processed twice. Processing each edge now takes $\Theta(\log |E|)$ worst-case time, so overall the algorithm runs in $O(|E| \log |E|)$ time. (Note $|E| \geq |V|$.)

Prim's algorithm: Implementation

Literally just breadth-first search with a priority queue!

Algorithm: BFS

Input : **Connected weighted** graph $G = ((V, E), w)$.
Output : **A minimum spanning tree for G.**

Number the vertices of G **arbitrarily** as $v_1, \dots, v_n$.
Let $\mathtt{L}[i] \leftarrow \infty$ for all $i \in [n]$.
Let $\mathtt{L}[1] \leftarrow 0$, $\mathtt{pred}[1] \leftarrow \mathtt{None}$.
Let $\mathtt{queue}$ be a **length-$|E|$ priority** queue containing all tuples (v, v_j) with $\{v, v_j\} \in E$,
  using their edge weights as priorities.
while $\mathtt{queue}$ *is not empty* **do**
  Remove front tuple (v_i, v_j) from $\mathtt{queue}$.
  if $\mathtt{L}[j] = \infty$ **then**
    Add (v_j, v_k) to $\mathtt{queue}$ for all $\{v_j, v_k\} \in E$, $k \neq i$.
    Set $\mathtt{L}[j] \leftarrow \mathtt{L}[i] + 1$, $\mathtt{pred}[j] = i$.
Return $\mathtt{pred}$.

Like with Dijkstra, we could "improve" this to $O(|E| + |V| \log |V|)$ time (with a much worse constant) by using a Fibonacci heap in place of the priority queue.