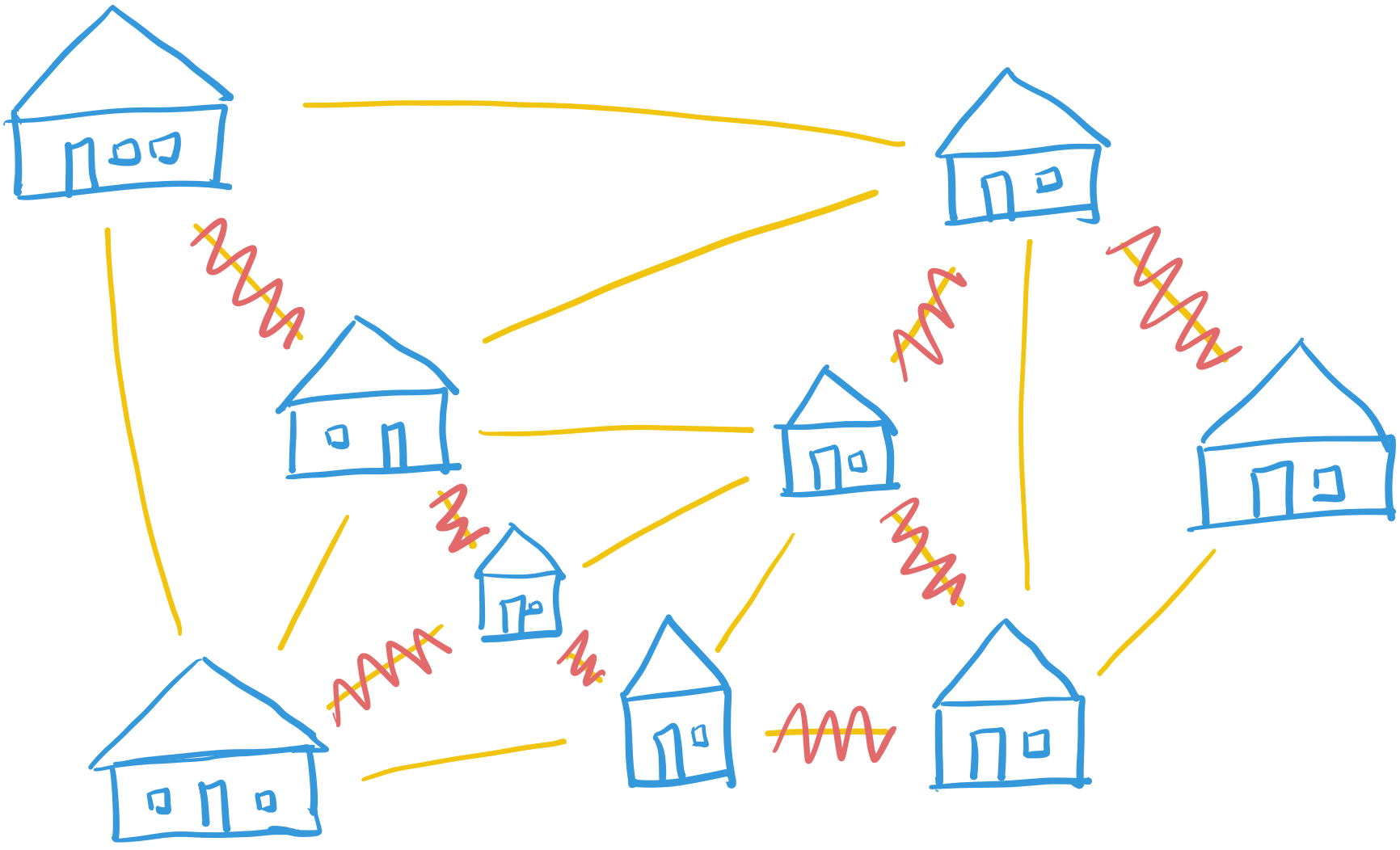
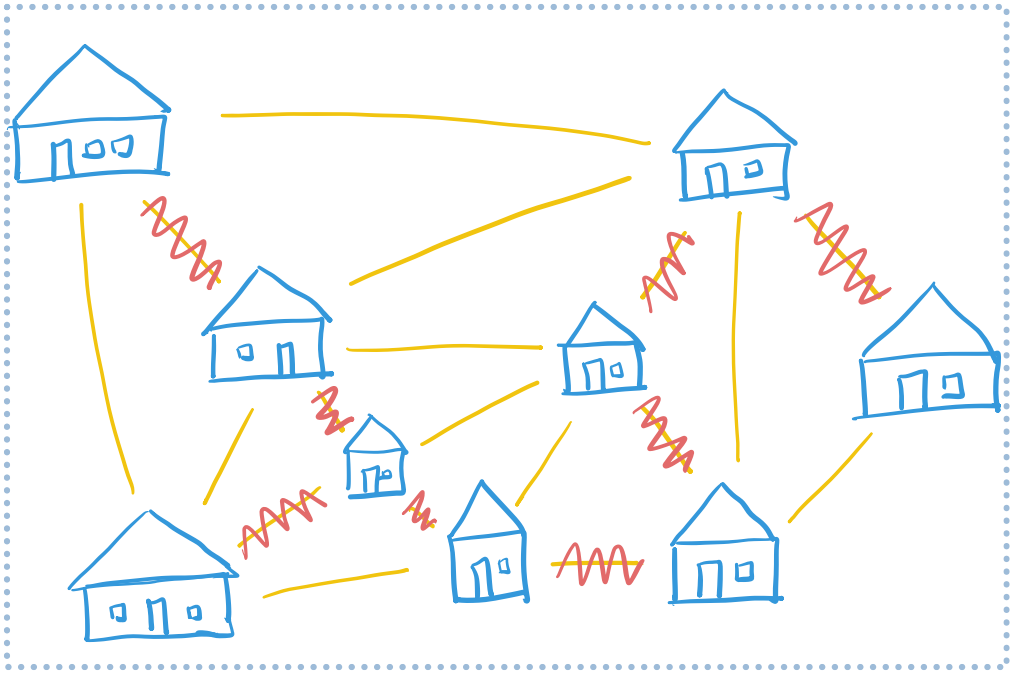
Spanning Trees



GOAL: connect town w/ minimum amount of electrical wire

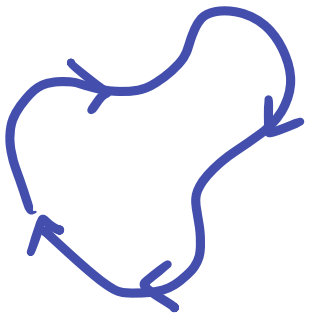
Quick observations



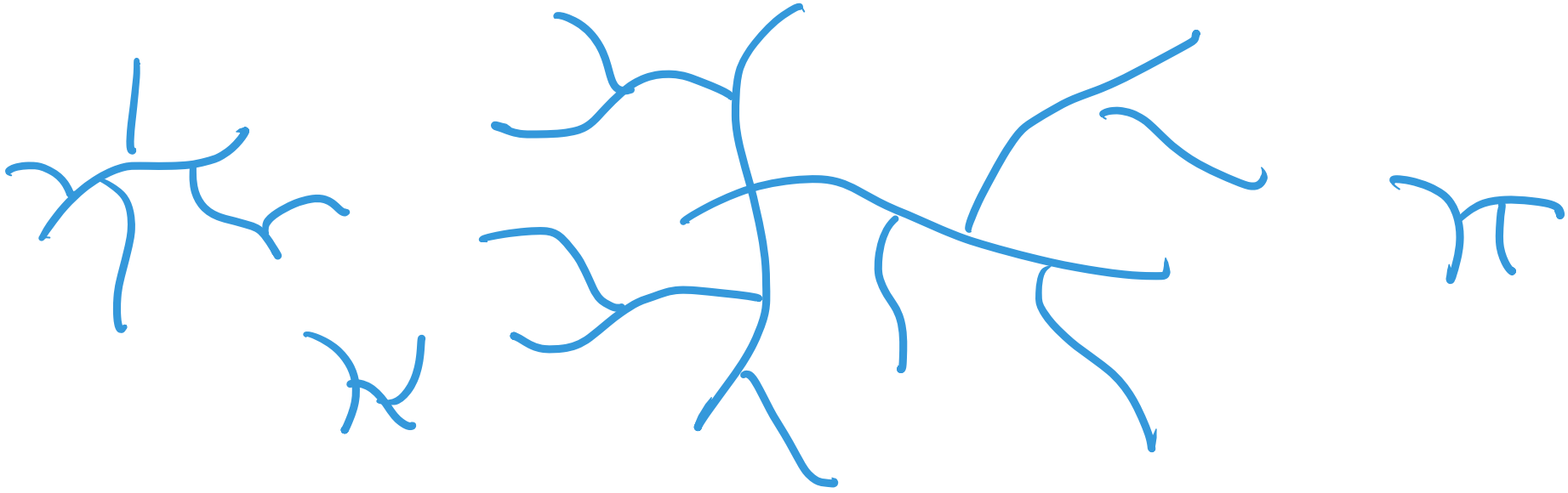
- no reason to have cycles
- biased towards short wires
- unused wires have alternative paths (which may be longer)

$G=(V,E)$ undirected, $F \subseteq E$ subset of edges.

"circuit" (nonempty) walk that starts/ends at same vertex



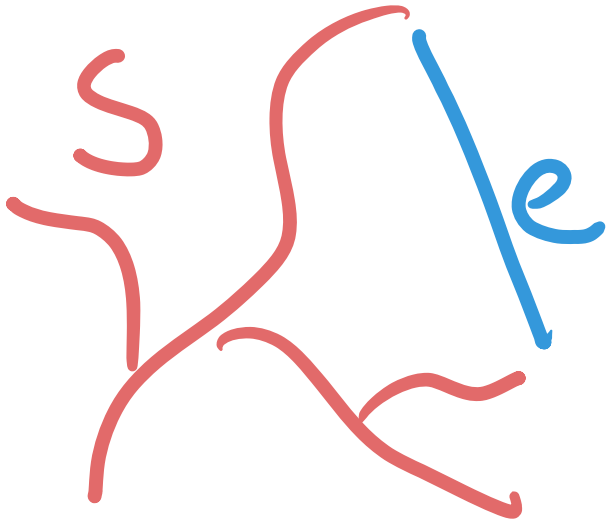
"forest": set of edges w/ no circuits



"tree": forest w/ one connected component

$G=(V,E)$ undirected, $S \subseteq E$

"S spans e"
endpoints of e are connected by S



$\text{span}(S) = \{e \in E : S \text{ spans } e\}$

"S is spanning" S spans every $e \in E$

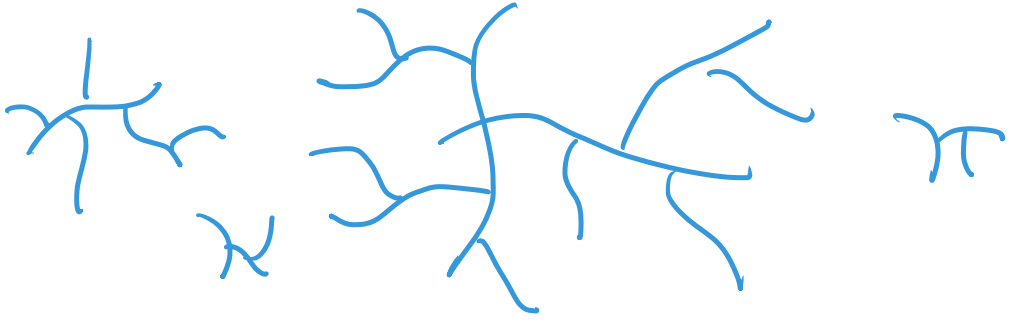
Two problems

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).

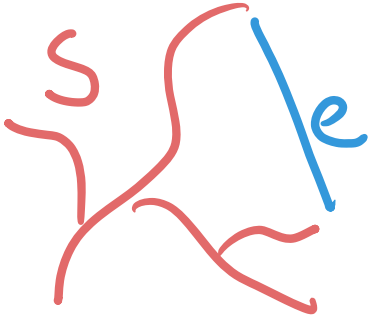
Two problems

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).

"forest": set of edges w/ no circuits



"S spans e"
endpoints of e are connected by S



"S is spanning" S spans every $e \in E$

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).

Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - 1 \leq |S|.$$

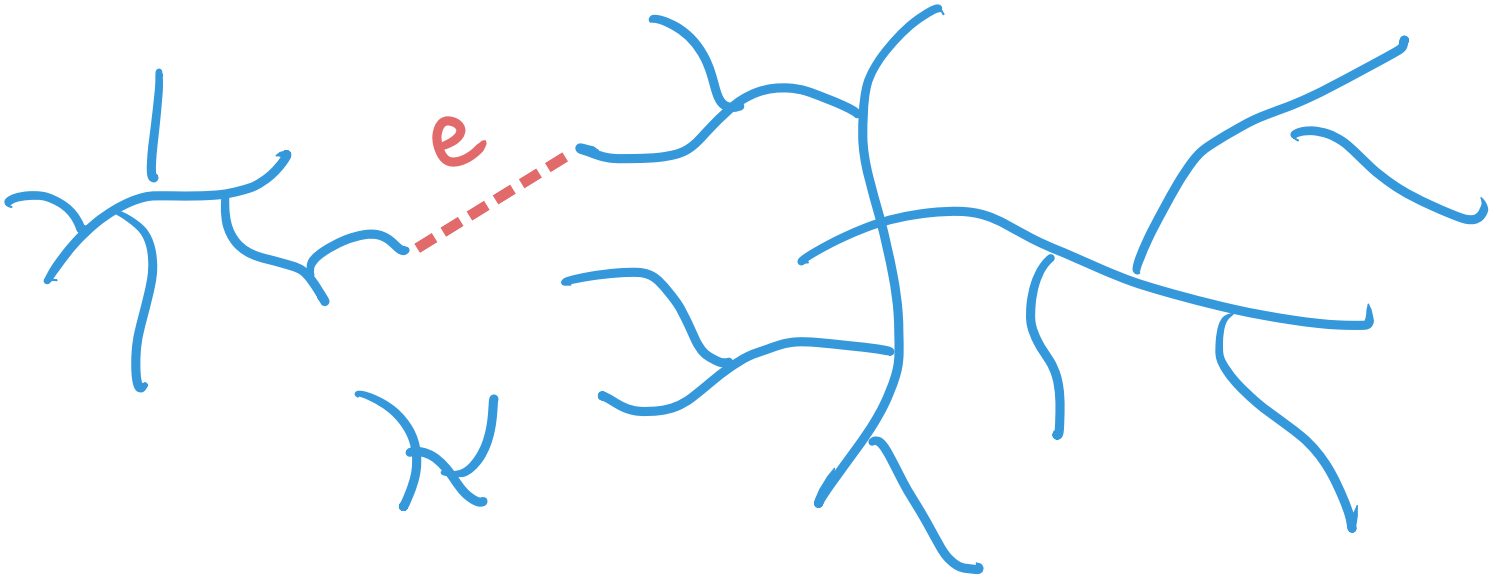
Counting argument based on # conn. comp.

Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - 1 \leq |S|.$$

Let F be a forest

Count # conn. comp. as we add F one edge at a time

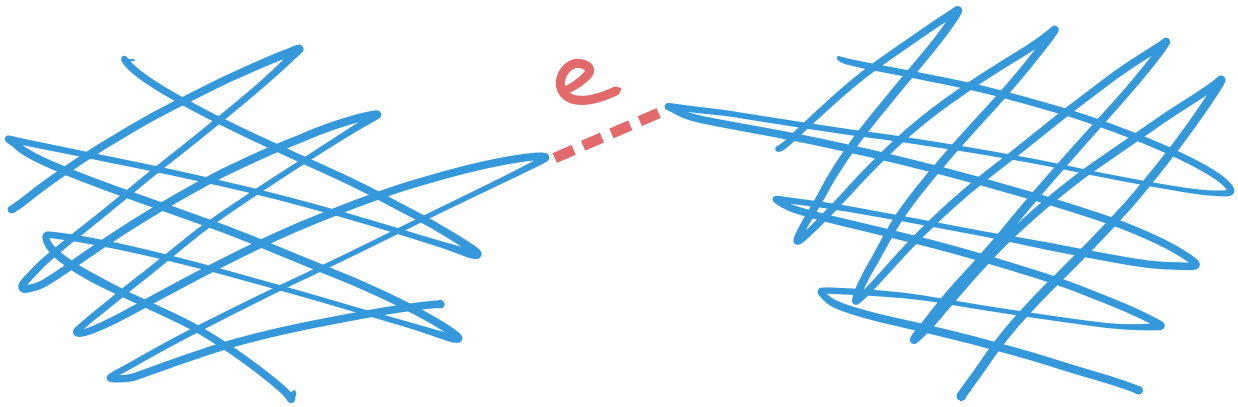


Let S be a spanning set

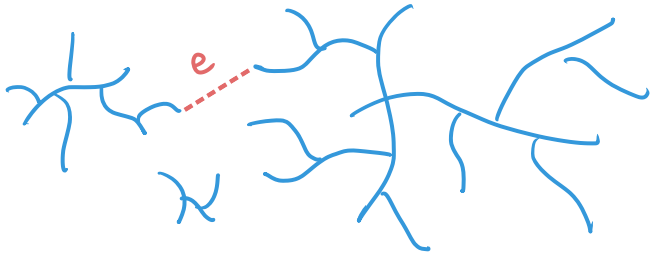
Count # conn. comp. as we delete edges from S

Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

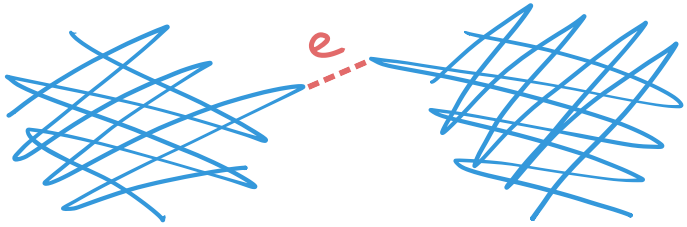
$$|F| \leq n - 1 \leq |S|.$$



Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*



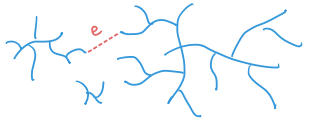
$$|F| \leq n - 1 \leq |S|.$$



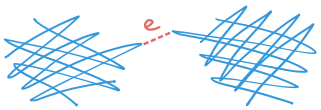
Lemma 13.2. *Let $G = (V, E)$ be an undirected graph with κ connected components. Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - \kappa \leq |S|.$$

Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*



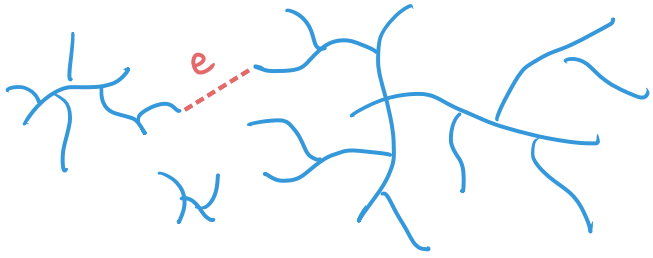
$$|F| \leq n - 1 \leq |S|.$$



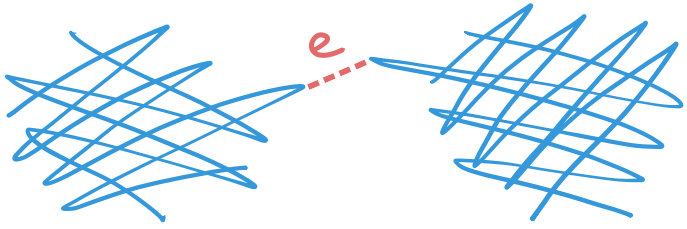
Lemma 13.2. *Let $G = (V, E)$ be an undirected graph with κ connected components. Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - \kappa \leq |S|.$$

Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*



$$|F| \leq n - 1 \leq |S|.$$



Lemma 13.2. *Let $G = (V, E)$ be an undirected graph with κ connected components. Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - \kappa \leq |S|.$$

Theorem 13.3. *Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges.*

1. *If F is a **maximal** forest, then F is a spanning forest.*
2. *If S is a **minimal** spanning set of edges, then S is a spanning forest.*

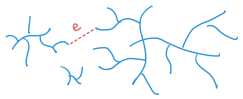
In particular, maximal forests are maximum forests, minimal spanning sets are minimum spanning sets, and in both cases they are spanning forests with $n - \kappa$ edges.

Theorem 13.3. *Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges.*

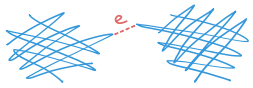
1. *If F is a **maximal** forest, then F is a spanning forest.*
2. *If S is a **minimal** spanning set of edges, then S is a spanning forest.*

In particular, maximal forests are maximum forests, minimal spanning sets are minimum spanning sets, and in both cases they are spanning forests with $n - \kappa$ edges.

Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*



$$|F| \leq n - 1 \leq |S|.$$



Lemma 13.2. *Let $G = (V, E)$ be an undirected graph with κ connected components. Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

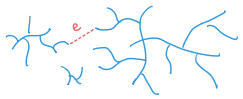
$$|F| \leq n - \kappa \leq |S|.$$

Theorem 13.3. *Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges.*

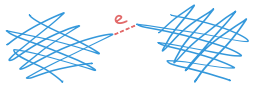
1. *If F is a **maximal** forest, then F is a spanning forest.*
2. *If S is a **minimal** spanning set of edges, then S is a spanning forest.*

In particular, maximal forests are maximum forests, minimal spanning sets are minimum spanning sets, and in both cases they are spanning forests with $n - \kappa$ edges.

Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*



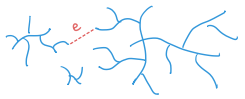
$$|F| \leq n - 1 \leq |S|.$$



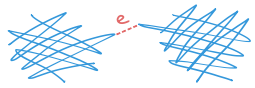
Lemma 13.2. *Let $G = (V, E)$ be an undirected graph with κ connected components. Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - \kappa \leq |S|.$$

Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*



$$|F| \leq n - 1 \leq |S|.$$



Lemma 13.2. *Let $G = (V, E)$ be an undirected graph with κ connected components. Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - \kappa \leq |S|.$$

Theorem 13.3. *Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges.*

1. *If F is a **maximal** forest, then F is a spanning forest.*
2. *If S is a **minimal** spanning set of edges, then S is a spanning forest.*

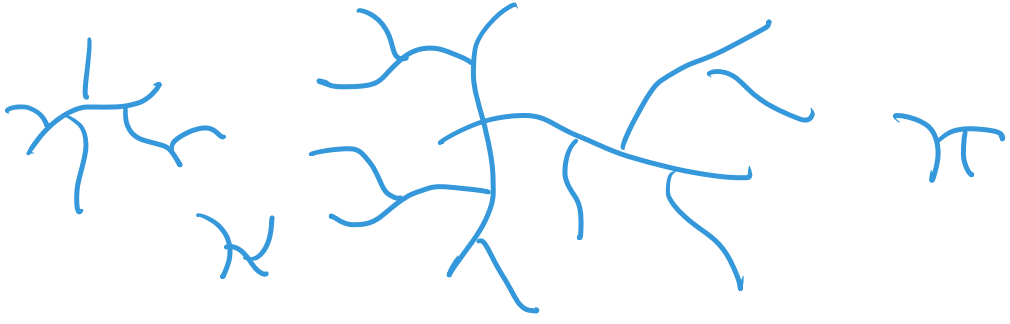
In particular, maximal forests are maximum forests, minimal spanning sets are minimum spanning sets, and in both cases they are spanning forests with $n - \kappa$ edges.

Corollary 13.4. *Every connected graph has a spanning tree, with $n - 1$ edges.*

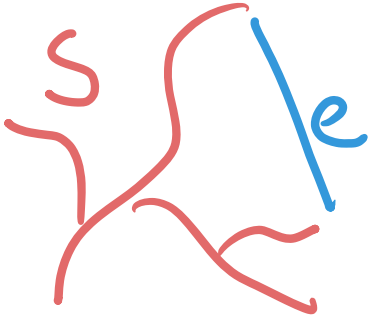
Two problems

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).

"forest": set of edges w/ no circuits



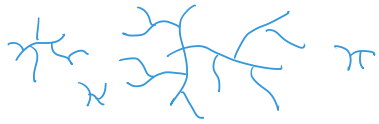
"S spans e"
endpoints of e are connected by S



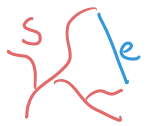
"S is spanning" S spans every $e \in E$

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).

"forest": set of edges w/ no circuits



"S spans e"
endpoints of e are connected by S



"S is spanning" S spans every $e \in E$

Good edges

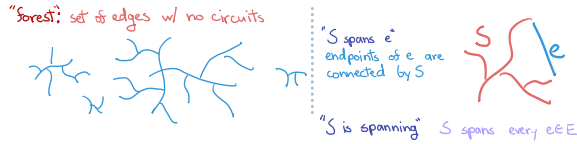
$e \in E$ is "good" if

For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),

$$w(e) < \max_{f \in F} w(f).$$

Good edges

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



e∈E is "good" if

For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),

$$w(e) < \max_{f \in F} w(f).$$

Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),* (e is good)

$$w(e) < \max_{f \in F} w(f).$$

Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

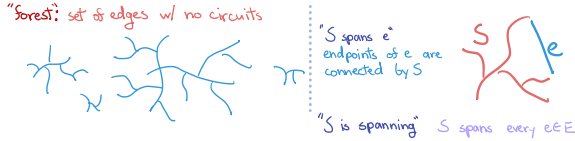
1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

Proof

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



e∈E is "good" if

For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),

$$w(e) < \max_{f \in F} w(f).$$

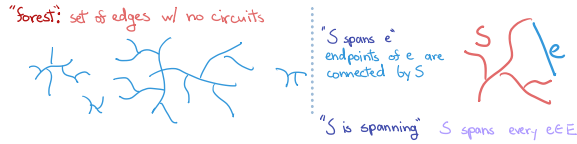
Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



e∈E is "good" if

For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),

$$w(e) < \max_{f \in F} w(f).$$

generic-MST$(G = (V, E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
      // Or throw an error if no such edge exists.
3. Return T.

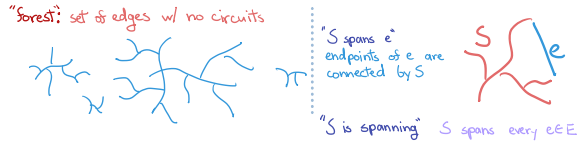
Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



e∈E is "good" if

For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),

$$w(e) < \max_{f \in F} w(f).$$

generic-MST$(G = (V, E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

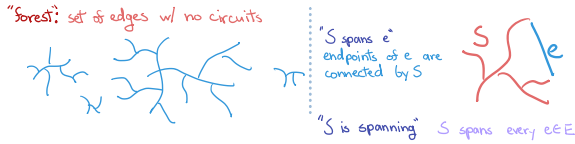
Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



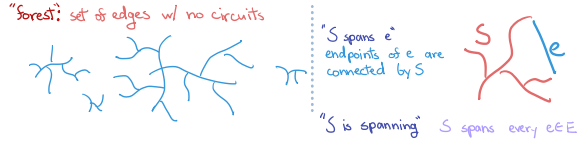

```
generic-MST(G = (V, E), w : E → ℝ)
1. T ← ∅
2. Until T is a spanning tree.
   A. Add a good edge e ∈ E \ span(T) to T
      // Or throw an error if no such edge exists.
3. Return T.
```

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

proof

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

"Greedy algorithm"

greedy-MST$(G = (V, E), w : E \to \mathbb{R})$

/* *Also known as Kruskal's algorithm.* */

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in increasing order of weight.
2. Set $T \leftarrow \emptyset$.
3. For $i = 1, \ldots, n$:

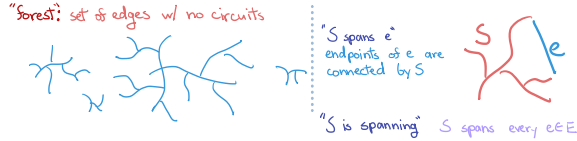
   /* *Below we discuss how to implement the following predicate efficiently.* */

   A. If $T + e_i$ is a forest:

      /* *e_i is the minimum weight edge not spanned by T* */

      1. Set $T \leftarrow T + e_i$.
4. Return T.

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),* (e is good)

$$w(e) < \max_{f \in F} w(f).$$

generic-MST$(G = (V, E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.

   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T

   // *Or throw an error if no such edge exists.*
3. Return T.

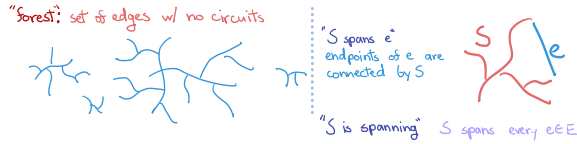
Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

greedy-MST$(G = (V, E), w : E \to \mathbb{R})$

/ Also known as Kruskal's algorithm. */*

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in increasing order of weight.
2. Set $T \leftarrow \emptyset$.
3. For $i = 1, \ldots, n$:
   \/ Below we discuss how to implement the following predicate efficiently. */*
   A. If $T + e_i$ is a forest:
      \/ e_i is the minimum weight edge not spanned by T */*
      1. Set $T \leftarrow T + e_i$.
4. Return T.

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

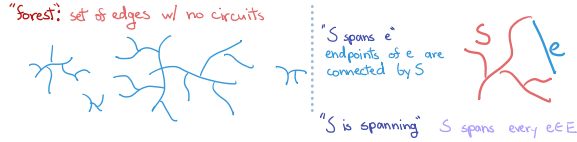
$$w(e) < \max_{f \in F} w(f).$$

generic-MST$(G = (V, E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

`generic-MST`$(G = (V,E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

"Parallel algorithm"

`parallel-MST`$(G = (V,E), w : E \to \mathbb{R})$

/ Also known as Borůvka's algorithm.*

**/*

1. $T \leftarrow \emptyset$
2. While T more than 1 connected component:
   A. For each component S of T, identify the smallest weight edge in $\partial(S)$.
   B. Add all of these edges to T.
3. Return T

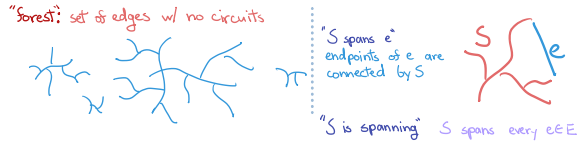
Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

`parallel-MST`$(G = (V,E), w : E \to \mathbb{R})$

/ Also known as Borůvka's algorithm. */*

1. $T \leftarrow \emptyset$
2. While T more than 1 connected component:
   A. For each component S of T, identify the smallest weight edge in $\partial(S)$.
   B. Add all of these edges to T.
3. Return T

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

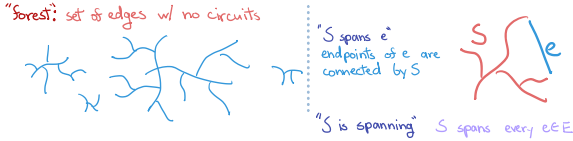
$$w(e) < \max_{f \in F} w(f).$$

`generic-MST`$(G = (V,E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

generic-MST($G = (V, E), w : E \to \mathbb{R}$)

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

"Search algorithm"

search-MST($G = (V, E), w : E \to \mathbb{R}$)

/ Also known as Prim's algorithm.*

1. Set $T \leftarrow \emptyset$ and $S \leftarrow \{v\}$ for an arbitrary vertex v.

/ We maintain the invariant that S is the connected component of v in T. */*

2. While T is not a spanning tree:

   / We can accelerate the following steps with a Fibonacci heap. */*

   A. Let e be the minimum weight edge in $\partial(S)$.

   B. Set $T \leftarrow T + e$, $S \leftarrow S \cup e$.

3. Return T.

**/*

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

search-MST($G = (V, E), w : E \to \mathbb{R}$)

/* *Also known as Prim's algorithm.* */

1. Set $T \leftarrow \emptyset$ and $S \leftarrow \{v\}$ for an arbitrary vertex v.

/* *We maintain the invariant that S is the connected component of v in T.* */

2. While T is not a spanning tree:

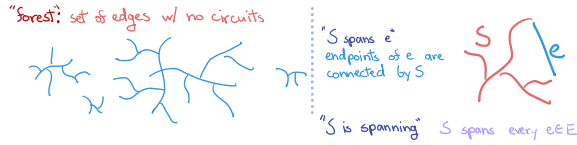
   /* *We can accelerate the following steps with a Fibonacci heap.* */

   A. Let e be the minimum weight edge in $\partial(S)$.

   B. Set $T \leftarrow T + e$, $S \leftarrow S \cup e$.

3. Return T.

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),* (e is good)

$$w(e) < \max_{f \in F} w(f).$$

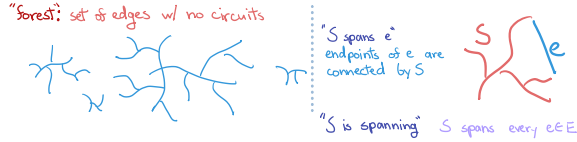
generic-MST($G = (V, E), w : E \to \mathbb{R}$)

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // *Or throw an error if no such edge exists.*
3. Return T.

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

Running times

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),* (e is good)

$$w(e) < \max_{f \in F} w(f).$$

`generic-MST`$(G = (V,E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

`greedy-MST`$(G = (V,E), w : E \to \mathbb{R})$

/ Also known as Kruskal's algorithm. */*

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in increasing order of weight.
2. Set $T \leftarrow \emptyset$.
3. For $i = 1, \ldots, n$:
   / Below we discuss how to implement the following predicate efficiently. */*
   A. If $T + e_i$ is a forest:
      / e_i is the minimum weight edge not spanned by T */*
      1. Set $T \leftarrow T + e_i$.
4. Return T.

`parallel-MST`$(G = (V,E), w : E \to \mathbb{R})$

/ Also known as Borůvka's algorithm. */*

1. $T \leftarrow \emptyset$
2. While T more than 1 connected component:
   A. For each component S of T, identify the smallest weight edge in $\partial(S)$.
   B. Add all of these edges to T.
3. Return T

`search-MST`$(G = (V,E), w : E \to \mathbb{R})$

/ Also known as Prim's algorithm. */*

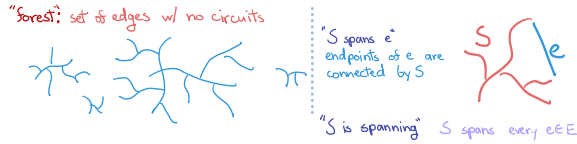
1. Set $T \leftarrow \emptyset$ and $S \leftarrow \{v\}$ for an arbitrary vertex v.

/ We maintain the invariant that S is the connected component of v in T. */*

2. While T is not a spanning tree:
   / We can accelerate the following steps with a Fibonacci heap. */*
   A. Let e be the minimum weight edge in $\partial(S)$.
   B. Set $T \leftarrow T + e$, $S \leftarrow S \cup e$.
3. Return T.

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

```
generic-MST(G = (V,E), w : E → ℝ)
1. T ← ∅
2. Until T is a spanning tree.
   A. Add a good edge e ∈ E \ span(T) to T
      // Or throw an error if no such edge exists.
3. Return T.
```

1. `union`(u,v): Given two elements u and v, take the union of the their sets.
2. `same-set`(u,v): Returns whether or not x and y are in the same set.
3. `new-set`(x): Given a new element x, creates the singleton set $\{x\}$.

Theorem 13.8. *There is a data structure that can implement the* `union`, `same-set`, *and* `new-set` *operations listed above that, for any m operations over n total elements, takes $O(m + n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function.*

```
greedy-MST(G = (V,E), w : E → ℝ)
/* Also known as Kruskal's algorithm.                              */
1. Initialize a new disjoint-union data structure U.
2. Call U.new-set(v) for all v ∈ V.
3. Sort and index E = {e_1, ..., e_n} in increasing order of weight.   // O(m log n)
4. Set T ← ∅.
5. For i = 1, ..., n
   A. Let e_i = {u,v}. Unless U.same-set(u,v):
      1. Set T ← T + e_i and call U.union(u,v).
6. Return T
```

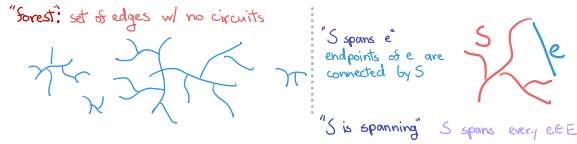
Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

`parallel-MST`$(G = (V,E), w : E \to \mathbb{R})$

/ Also known as Borůvka's algorithm. */*

1. $T \leftarrow \emptyset$
2. While T more than 1 connected component:
   A. For each component S of T, identify the smallest weight edge in $\partial(S)$.
   B. Add all of these edges to T.
3. Return T

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

(e is good)

$$w(e) < \max_{f \in F} w(f).$$

`generic-MST`$(G = (V,E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

search-MST($G = (V, E), w : E \to \mathbb{R}$)

/* *Also known as Prim's algorithm.* */

1. Set $T \leftarrow \emptyset$ and $S \leftarrow \{v\}$ for an arbitrary vertex v.

/* *We maintain the invariant that S is the connected component of v in T.* */

2. While T is not a spanning tree:

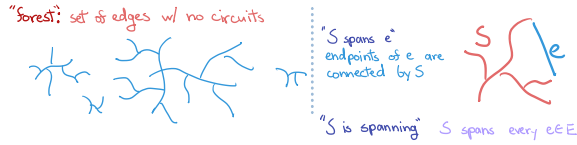
   /* *We can accelerate the following steps with a Fibonacci heap.* */

   A. Let e be the minimum weight edge in $\partial(S)$.

   B. Set $T \leftarrow T + e$, $S \leftarrow S \cup e$.

3. Return T.

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).



Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),* (e is good)

$$w(e) < \max_{f \in F} w(f).$$

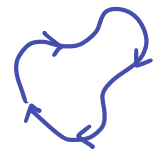
generic-MST($G = (V, E), w : E \to \mathbb{R}$)

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // *Or throw an error if no such edge exists.*
3. Return T.

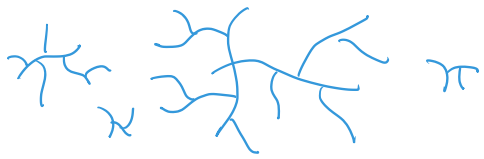
Chapter 13

Spanning trees

Let $G = (V, E)$ be an undirected graph with m edges and n vertices. Recall that a **circuit** in F is any nonempty walk that starts and ends at the same vertex. A **forest** is an edge set $F \subseteq E$ with no circuits (or equivalently, no cycles). A **tree** is a forest with one connected component.



A circuit



A forest.

For a set of edges $S \subseteq E$, and an edge $e \in E$, we say that ***S* spans *e*** if the endpoints of e are connected in S. We say that S is **spanning** if it spans every edge in E. We denote

$$\operatorname{span}(S) = \{e \in E : S \text{ spans } e\}.$$

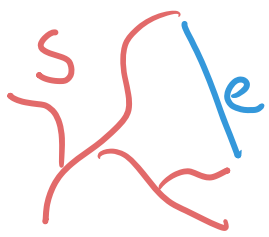


Fig. 13.1: S spans e.

Observer that $\operatorname{span}\big(\operatorname{span}(S)\big) = \operatorname{span}(S)$ for all F.

This chapter focus on the following two graph problems.

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight).
2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight, when the edges are weighted).

It is helpful to catalog the two problems as packing and covering problems, respectively. In the first problem, the goal is to find a set F with as many edges as possible, subject to not taking all the edges of any cycle. In general, problems where we want to take as many elements as possible subject to a set of maximum

capacity constraints are called *packing problems.* In the second problem, the goal is to take a set $S \subseteq E$ with as few edges as possible, subject to making sure S spans (i.e., "covers") every edge in e. In general, problems where we want to take as few as possible subject to a set of minimum requirements are called *covering problems.*

We will develop algorithms that solve both of the above problems. As we do so, it is worth paying attention to the *style* of our investigation. We will make a sequence of small, incremental observations about the above definitions. As we get to understand the definitions well, connections between the two problems emerge, technical issues melt away, and algorithms present themselves. In fact we will end up with several different algorithms for the above problems, and (if done right) it will be easy to see that they all work for the same reasons.

13.1 Unweighted graphs

We first focus on the unweighted versions of the problem. The two problems are to compute a *minimum cardinality spanning set* and a *maximum cardinality forest.* Throughout this section, let κ denote the number of connected components in G.

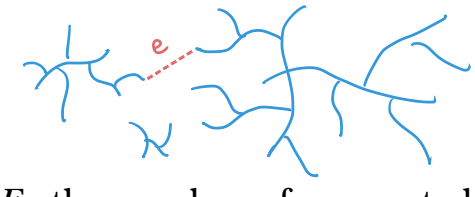
Locally optimal vs globally optimum. An easier problem is to obtain *maximal* forests and *minimal* spanning sets. A set $F \subseteq E$ is a *maximal forest* if for every edge $e \notin F$, $F + e$ is *not* a forest. A set $S \subseteq E$ is a *minimal spanning set* if for all $e \in S$, $S - e$ is not a spanning set. It is easy to find a maximal forest: start with $F \neq \emptyset$, and repeatedly add edges to F subject to keeping F a forest, until we are out of options. Similarly one can easily find a minimal spanning set: start with $S \subseteq E$, and repeatedly remove edges from S as long as S remains a spanning set. We will end up with a maximal forest F and a minimal spanning set S in polynomial time. These represent *locally optimal* solutions – adding or deleting a single edge either violates a constraint or hurts the cardinality. Unfortunately we do not know if they are *globally optimum* solutions – perhaps a particular sequence of additions / deletions will lead to a better solution than others.

Spanning sets are bigger than forests. Our first lemma concerns the maximum number of edges in a forest, and the minimum number of edges in a spanning set, in a connected graph. What is perhaps most interesting about the following lemma is the relationship it uncovers between the comparative sizes of forests and spanning sets: it says every spanning set has at least as many edges as any forest.

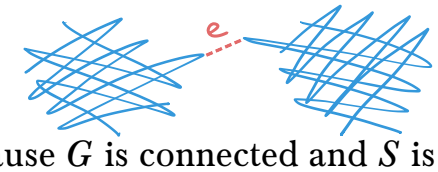
Lemma 13.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - 1 \leq |S|.$$

Proof. Consider the forest F. Imagine starting with an empty graph, and tracking the number of connected components as we add edges from F one-by-one. Initially we have n connected components each consisting of an isolated vertex. Each additional edge e cannot introduce a cycle, so the endpoints of e must be in two different connected components of the current graph. In particular, e reduces the number of connected components by 1. After adding all the edges from F, the number of connected components is at least 1, so we have added at most $n-1$ edges from F. That is, $|F| \leq n-1$.



Now consider the spanning set S. Imagine starting with S as a graph, and tracking the number of connected components of S as we remove edges one-by-one. Initially we have 1 connected component because G is connected and S is spanning. Each time we remove an edge e, the number of connected components increases by at most 1. At the end there are no edges left and n connected components. To reach n connected components, we must have removed at least $n-1$ edges from S. ■



The follow lemma extends the previous ones to disconnected graphs, by treating each connected component as if it were a connected graph on its own.

Lemma 13.2. *Let $G = (V, E)$ be an undirected graph with κ connected components. Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - \kappa \leq |S|.$$

Proof. Let $G_1 = (V_1, E_1), \ldots, G_\kappa = (V_\kappa, E_\kappa)$ be the connected components of G. For each i, let $F_i = F \cap E_i$ be the restriction of F to G_i, and let $S_i = F \cap E_i$ be the restriction of F to G_i. Let $n_i = |V_i|$. By lemma 13.1, we have

$$|F_i| \leq n_i - 1 \leq |S_i|$$

for each i. Summing over $i = 1, \ldots, \kappa$ gives the claimed inequality. ■

Locally optimal vs globally optimum, revisited. Lemma 13.2 establishes that for any forest F and any spanning set S, we have

$$|F| \leq |S|.$$

Our maximization problem is bounded above by our minimization problem. What is particularly nice is that the two problems can mutually certify each other to be optimal: *if* F is a forest, and S is a spanning set, such that

$$|F| \geq |S|,$$

then we know that F and S are both globally optimum solutions for their respective problems. Better yet, if a set of edges S was both spanning and acyclic, then it is *both* a maximum cardinality forest and a minimum cardinality spanning set. In the following theorem and proof, observe how the dual relationship between forests and spanning sets play off each other.

Theorem 13.3. *Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges.*

1. *If F is a **maximal** forest, then F is a spanning forest.*
2. *If S is a **minimal** spanning set of edges, then S is a spanning forest.*

In particular, maximal forests are maximum forests, minimal spanning sets are minimum spanning sets, and in both cases they are spanning forests with $n - \kappa$ edges.

Proof. For the first claim, let $F \subseteq E$ be a maximal forest, and suppose by contradiction that F is not spanning. Any edge e not spanned by F can be added to F without introducing a cycle. But then F is not maximal, a contradiction.

For the second claim, let $S \subseteq E$ be a minimal spanning edge set and suppose by contradiction that S is not a forest. In particular, S contains a cycle. Deleting an edge from a cycle in S does not effect connectivity. But then S is not minimal, a contradiction. ■

All minimal spanning edge have the same size, and therefore are all *minimum* size spanning sets. Likewise all maximal edge sets have the same size, and therefore are all *maximum* size forests. It is extremely convenient that minimal equals minimum and maximal equals minimum – it suggests a sort of discrete analogue of convexity and concavity. We also highlight they way that forests and spanning sets played off each other in the proof of theorem 13.3. This is our first example of a more general phenomena called *packing and covering duality*.

Theorem 13.3 proves that the following algorithms both produce a spanning forest:

1. Starting from $F = E$, repeatedly remove edges $e \in F$ as long as $F - e$ remains spanning.
2. Starting from $F = \emptyset$, repeatedly add edges $e \in E \setminus F$ to F as long as $F + e$ remains a forest.

The first algorithm terminates with a minimal spanning edge set, which by theorem 13.3 is a forest. The second algorithm terminates with a maximal forest, which by theorem 13.3 is also spanning.

An important special case is when a graph is connected. In this case, the spanning forest is a spanning *tree*.

Corollary 13.4. *Every connected graph has a spanning tree, with $n - 1$ edges.*

13.2 Weighted graphs

Let $G = (V, E)$ be a connected and undirected graph, with edge weights given by $w : E \to \mathbb{R}$. Consider the problem of computing the minimum weight spanning tree (MST) of G. We assume for simplicity that the edge weights are distinct. However the algorithms we develop will work for general weights as well, and we address this at the end of the section. It will also be able to see that the algorithms generalize to unconnected graphs as well, where we find a minimum weight spanning forest rather and not a tree.

Good edges. Thus, consider the problem of finding the minimum weight spanning tree where we assume the graph is connected and the edge weights are distinct. Our algorithms will all be based on the following key lemma.

Lemma 13.5. *Let $e \in E$. Then the following conditions are equivalent.*

1. *Every minimum weight spanning tree contains e.*
2. *For every set of edges $F \subseteq E - e$ that spans e (and doesn't include e),*

$$w(e) < \max_{f \in F} w(f).$$

Proof. (1) implies (2). Suppose every minimum weight spanning tree T contains e. Suppose by contradiction that (2) does not hold. Let $F \subseteq E$ be a set of edges spanning e where $w(e) > w(f)$ for all $f \in F$. In particular F contains a path P where $w(e) > w(f)$ for all $f \in P$.

Consider the forest $T - e$ obtained by removing e from T. Let $e = \{a, b\}$. $T - e$ has two connected components A and B, where A contains a and B contains b. As a path from $a \in A$ to $b \in B$, P must have an edge f crossing from A to B. This edge f is not spanned by $T - e$, so $T' \stackrel{\text{def}}{=} T - e + f$ is a forest. Since T' has $n - 1$ edges, it is also a spanning tree. It has weight

$$\bar{w}(T') = \bar{w}(T) - w(e) + w(f) \stackrel{\text{(a)}}{<} \bar{w}(T)$$

where (a) is because $w(f) < w(e)$ by assumption. But then T is not the minimum weight spanning tree, a contradiction.

(2) implies (1). Suppose (2) holds. Suppose by contradiction that T is a minimum weight spanning tree excluding e. Consider the unique path $P \subseteq T$ connecting the endpoints of e. Then P spans e, so P contains an edge f with $w(e) \leq w(f)$.

We claim that $T' = T - f + e$ is a spanning tree. We first observe that $T - f$ does not span e. Indeed, if $T - f$ did span e, then this implies a second path P' in T spanning e distinct from P, a contradiction. Thus $T - f$ is a forest that

does not span e, with $n-2$ edges. $T - f + e$ is then a forest with $n-1$ edges. Any forest with $n-1$ edges is a spanning tree.

Thus $T - f + e$. It has weight

$$\bar{w}(T') = \bar{w}(T) - w\big(f\big) + w(e) < \bar{w}(T),$$

which means that T is not the minimum weight spanning tree, a contradiction. ■

Let us call an edge e **good** if it satisfies condition (2) of lemma 13.5; in particular, all good edges are in all MST's.

Now, consider the following abstract algorithm that simply tries to add good edges. Now there are two issues with this approach (particularly in step (2.A)) which prevents it from being a "real" algorithm. First, it is not clear that good edges are always available. For the time being we throw an error if we fail to find a good edge. Second, it is not clear how to certify that an edge e is good. For the sake of discussion, we simply pretend we can identify good edges as well. Later we will identify concrete strategies to do so.

`generic-MST`$(G = (V,E), w : E \to \mathbb{R})$

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

`generic-MST` is very conservative, only adding edges that we know (by lemma 13.5) must be in every minimum weight spanning tree. However it is not clear that these edges are sufficient to find the minimum spanning tree. Still, almost tautologically, we have the following.

Lemma 13.6. *If* `generic-MST` *terminates, then it returns the minimum spanning tree, which is unique.*

Proof. Indeed, it returns a spanning tree T where, by lemma 13.5, every edge in T is in every spanning tree. ■

It remains to show that `generic-MST` always terminates. In particular, we need to show that an edge satisfying step (2.A) always exists.

13.2.1 The greedy algorithm

Lemma 13.7. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

Proof. Let $F \subseteq E - e$ be any set of edges spanning e. Since F spans e and H does not, F is not a subset of $\text{span}(H)$. Any edge $f \in F \setminus \text{span}(H)$ must have $w(f) > w(e)$ by choice of e, as required. ■

The key point to the above lemma is that the set H gives a **certificate** that e should be included the minimum weight spanning tree. Without such a certificate, it is hard to justify taking e. The lemma leads to perhaps our simplest implementation of `generic-MST`: repeated add the minimum weight edge that does not introduce a cycle, until we have a spanning tree. This called the **greedy algorithm** and is also known as Kruskal's algorithm.

`greedy-MST`$(G = (V, E), w : E \to \mathbb{R})$

/ Also known as Kruskal's algorithm. */*

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in increasing order of weight.
2. Set $T \leftarrow \emptyset$.
3. For $i = 1, \ldots, n$:
   \/ Below we discuss how to implement the following predicate efficiently. */*
   A. If $T + e_i$ is a forest:
      \/ e_i is the minimum weight edge not spanned by T */*
      1. Set $T \leftarrow T + e_i$.
4. Return T.

Running time and the disjoint union data structure. Correctness follows immediately as a special case of `generic-MST`. Running time is another matter and we briefly review the algorithm to set the stage. The algorithm sorts the edges by weight, and repeatedly adds edges in order as long as they do not introduce a cycle. Sorting is easy. However, checking that we do not introduce a cycle in step (3.A) is not so obvious. Given an edge $e_i = (u, v)$, we need to quickly figure out if the endpoints u and v are in the same components with respect to the current forest. One could do so by running a searching algorithm from u, which leads to a $O(mn)$ running time.

To do better, imagine the connected components of T over the course of the algorithm. Initially, when $T = \emptyset$, every vertex is in its own singleton set.

Thereafter, whenever we add an edge e_i to T, we also combine the connected components of the endpoints, replacing the two components with their *union*. Along the way, we are constant checking two vertices are already in the same component.

These operations are facilitated by the **disjoint union** data structure, which has the following interface.

1. `union`(u,v): Given two elements u and v, take the union of the their sets.
2. `same-set`(u,v): Returns whether or not x and y are in the same set.
3. `new-set`(x): Given a new element x, creates the singleton set $\{x\}$.

We plan to cover the disjoint union data structure later when we focus on data structures. For now we treat it as a black box with the following guarantees.

Theorem 13.8. *There is a data structure that can implement the* `union`, `same-set`, *and* `new-set` *operations listed above that, for any m operations over n total elements, takes $O(m + n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function.*

We refer to the wikipedia article for more on the inverse Ackerman function, suffice it to say that it is asymptotically smaller than $\log(n)$, or $\log(\log(n))$, or $\log(\log(\log(n)))$, or $\log(\log(\log(\log(n))))$...

Now we put everything together.

Theorem 13.9. *The greedy algorithm computes the MST in $O(m \log n)$ time.*

Proof. We have already addressed the running time above in lemma 13.6. For the running time, we observe that we make $O(m)$ calls to the `disjoint-union` data structure to maintain disjoint sets over n elements. The total running time from these operations is $O(m + n\alpha(n))$. The remaining operations take $O(m \log n)$ time, where the bottleneck is from sorting. ■

Note that the $O(m + n\alpha(n))$ bound for the `disjoint-union` data structure was overkill, since sorting was already a bottleneck.

13.2.2 A parallel algorithm

For a set of vertices $S \subset V$, the **cut** induced by S, denoted $\partial(S)$, is the set of edges with exactly one endpoint in S:

$$\partial(S) \stackrel{\text{def}}{=} \{\{u,v\} \in E : u \in S, v \notin S\}$$

The name "cut" comes from the fact that removing $\partial(S)$ from G cuts off S from the rest of the graph.

`greedy-MST`$(G = (V,E), w : E \to \mathbb{R})$

/ Also known as Kruskal's algorithm. */*

1. Initialize a new `disjoint-union` data structure U.
2. Call U`.new-set(`v`)` for all $v \in V$.
3. Sort and index $E = \{e_1, \dots, e_n\}$ in increasing order of weight. *// $O(m \log n)$*
4. Set $T \leftarrow \emptyset$.
5. For $i = 1, \dots, n$
   A. Let $e_i = \{u, v\}$. Unless U`.same-set(`u, v`)`:
      1. Set $T \leftarrow T + e_i$ and call U`.union(`u, v`)`.
6. Return T

Figure 13.2: The greedy algorithm for MST implemented with the disjoint union data structure.

Lemma 13.10. *Let $S \subset V$ be any set of vertices. Then the minimum weight edge in the cut induced by S is a good edge.*

Proof. Let $H = E - \partial(S)$. Then $E \setminus \text{span}(H) = \partial(S)$. The claim now follows from lemma 13.7. ■

This inspires the following instantiation of `generic-MST` with a parallel character: each round, for each connected component S of the current tree T, identify the smallest weight edge in $\partial(S)$. Add all of these edges to T.

`parallel-MST`$(G = (V,E), w : E \to \mathbb{R})$

/ Also known as Borůvka's algorithm. */*

1. $T \leftarrow \emptyset$
2. While T more than 1 connected component:
   A. For each component S of T, identify the smallest weight edge in $\partial(S)$.
   B. Add all of these edges to T.
3. Return T

An alternative perspective is to imagine *contracting* the edges as they are added to T. For an edge $e = \{u, v\}$, **contracting** e means we identify u and v as the same vertex. One way to implement this is to introduce a new vertex (say) z, replace u and v with z in every edge incident to u or v, and remove u and v from the graph.

The contraction version of the parallel algorithm immediately contracts every edge that is added to T. As we do so, every vertex in the contracted graph corresponds to a distinct connected component of T.

`parallel-MST`$(G = (V, E), w : E \to \mathbb{R})$

/ Also known as Borůvka's algorithm. */*

1. $T \leftarrow \emptyset$
2. While V has more than one vertex:
   A. For each vertex v, identify the smallest weight edge incident to v.
   B. Contract all the selected edges, and add the original, uncontracted versions of them to T.
3. Return T

Theorem 13.11. `parallel-MST` *computes the minimum spanning tree, and can be implemented to run in* $O(m \log(n))$.

Proof. Correctness follows from lemma 13.6 and lemma 13.10. For the running time, we first observe that there are at most $O(\log(n))$ iterations of the outer loop. This is because each iteration reduces the number of components by at least a factor of 2.

Next we claim that each iteration can be implemented in $O(m)$ time. We first analyze the first version that does not contract edges. To build some intuition, we point out that this is very easy in the first round: each vertex scans its list of incident edges to identify the smallest weight edge incident to that vertex. In subsequent iterations, some additional effort is required to do this on a per-component basis, but it is not too difficult either. By running BFS or DFS, we identify the connected component of each vertex, labeling the vertex with some identifier for the component. (e.g., a unique integer.) We then run through the edges and relabel the endpoints by the corresponding endpoints. Then, similar to the first iteration, it is easy to find the smallest edge cut by each component.

The contraction version is implicitly very similar and we analyze it for the sake of completeness. Each iteration, observe that contracting all the selected edges amounts to building a new (smaller) graph. Each round, one should label

the connected components formed by the selected edges, and contract each component all at once. Then the whole process takes $O(m)$ time. So again we spend $O(m)$ time per iteration.

In either version, a single iteration takes $O(m)$ time, and there are $O(\log n)$ iterations. This gives the claimed running time. ■

Beyond theoretical and sequential running times. Borůvka's algorithm is clearly very simple, and although it is not the fastest theoretical running time for MST we will see, the simplicity seems to produce very good performance in practice[1]. Another important property of Borůvka's algorithm it is that it is inherently *parallel*. Each vertex in the contracted graph can pick it's lightest weight edge independently. Borůvka's algorithm takes $O(\log n)$ time (up to lower order terms) on most natural distributed and parallel models of computation.

13.2.3 The search algorithm

Our final algorithm, like the previous one, also certifies its decisions by only taking the minimum weight edge of cuts induced by components of the spanning tree. Rather than adding many edges in parallel, this algorithm focuses on one component (which is initially one vertex) and keep adding the minimum weight edge to add the next component. Finding the next edge can be implemented very efficiently with a Fibonacci heap.

`search-MST`$(G = (V, E), w : E \to \mathbb{R})$

/ Also known as Prim's algorithm. */*

1. Set $T \leftarrow \emptyset$ and $S \leftarrow \{v\}$ for an arbitrary vertex v.

/ We maintain the invariant that S is the connected component of v in T. */*

2. While T is not a spanning tree:

   / We can accelerate the following steps with a Fibonacci heap. */*

   A. Let e be the minimum weight edge in $\partial(S)$.

   B. Set $T \leftarrow T + e$, $S \leftarrow S \cup e$.

3. Return T.

Theorem 13.12. `search-MST` *returns the MST, and can be implemented to run in* $O(m + n \log n)$ *time.*

[1] We also point out that the theoretically fastest algorithms [Cha00; KKT95], not covered here, are based on Borůvka's algorithm. See also exercise 13.9.

Proof. We maintain a Fibonacci heap [FT87] over $V \setminus S$. For each vertex $v \in V \setminus S$, we use the minimum weight of any edge from v to S as the priority for v. This may be $+\infty$ if (early on) there is no such edge, and can be revised as more vertices are added to S. With this setup, `remove-min` returns the next vertex to add to S.

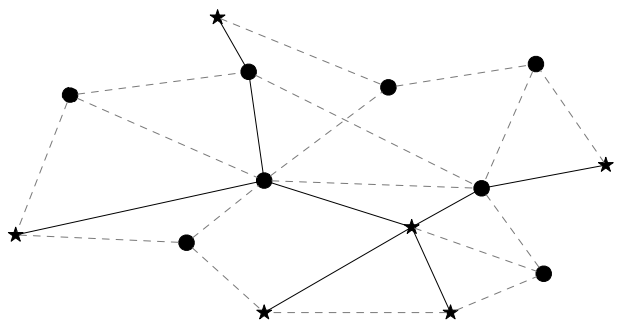
We call `decrease-key` once for every edge, and `remove-min` once for every vertex. This gives an overall running time of $O(m + n \log n)$. ■

13.2.4 When weights are not distinct

Above we assumed the edge weights are distinct which simplified the analysis substantially. In particular, when edge weights are distinct, the minimum spanning tree is unique.

We can drop this assumption without redoing the analysis, by the following thought experiment. Consider any of the concrete algorithms above. The algorithm - taking minimum weight edges in different ways - still mostly make sense, although the choice of edge may not be unique. The one caveat is Borůvka's algorithm, but as long as we break ties consistently, it is sensible. That said, consider any fixed run of one of these algorithms, which produces a tree T. As a thought experiment, we can perturb the edge weights so that (a) the edge weights are unique, and (b) the algorithm would still return the same set of edges. One such way is to add ϵi to the weight of an edge taken in the ith iteration, and ϵn to the weight of any edge not taken, where $\epsilon > 0$ is sufficiently small such that this change only acts as a tiebreaker. Our analysis above applies directly to this thought experiment, which in turn justifies the actual algorithm run on non-distinct weights.

13.3 Steiner trees



We now turn to a natural and useful generalization of spanning trees called **Steiner tree**. Let $G = (V, E)$ be an undirected graph. Let $T \subset V$ be a subset of vertices called **terminals**. A **Steiner tree** over T is a tree $F \subseteq E$ in which U is connected. For example, a spanning tree is a Steiner tree for $T = V$. Above is a

picture of a Steiner tree where terminals are indicated by ⋆'s, and excluded edges are drawn dashed.

Given positive edge weights $w : E \to \mathbb{R}$, the **minimum weight Steiner tree** problem is to find the Steiner tree whose edges have minimum total weights. Minimum weight Steiner tree is a very important problem in network design – interpreting the edge weights as costs, the minimum weight Steiner tree is the minimum cost network that connects a fixed set of points. Similarly it is very useful in automated VLSI design.

Theorem 13.13. *The Steiner tree problem is NP-Hard, even in unweighted graphs.*

Proof. We describe a reduction from 3-SAT. Let $f(x_1, \ldots, x_n)$ be a 3CNF with m clauses $C_1, \ldots, C_m$.

1. For each variable x_j, a terminal vertex.
2. For each clause C_i, a terminal vertex.
3. For each assignment such as $x_j = \texttt{true}$ or $x_j = \texttt{false}$, a (non-terminal) vertex.

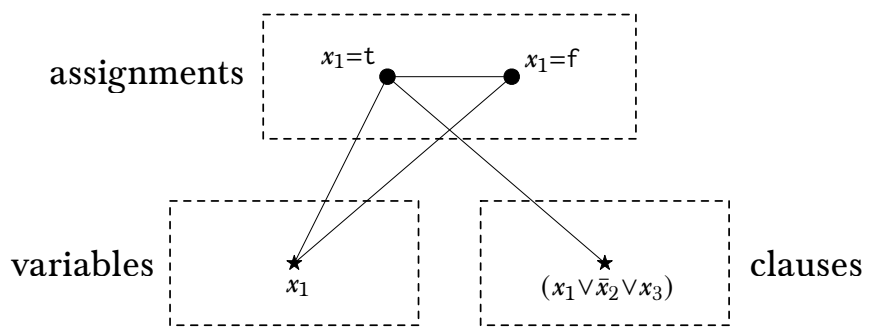
This creates a total of $3n + m + 1$ vertices. Next we create the edges.

4. An edge between every variable and its two assignments.
5. An edge between every clause and every satisfying assignment.
6. An edge between every pair of assignments.

This creates

$$2n + 3m + \binom{2n}{2} = O\left(m + n^2\right)$$

edges. A high-level diagram of the construction is given below.

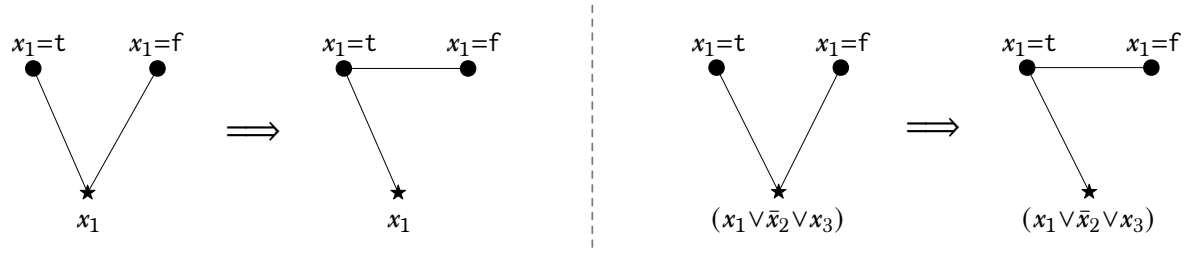


The idea behind the proof is as follows. Ultimately, we need to connect the variables and clauses, via assignments. Let us say that a Steiner tree "takes an assignment" if it the edge between the corresponding edge and assignment is included in the tree. The terminals corresponding to the terminals requires the Steiner tree to take at least one edge incident to each variable. The terminals corresponding to each clauses requires us to take at least one edge incident to every clause. We can also assume that every minimum Steiner tree has exactly one edge to each variable and to each clause, because when there is more than one, we can replace one of them with an edge between the corresponding assignments. This implies that every Steiner tree contains at least $m + n + n - 1 = m + 2n - 1$ edges – n for the variables, m for the clauses, and then (at least) $n - 1$ to connect the assignments of the variables to one another. A satisfying assignment for f maps to a Steiner tree with $m + 2n - 1$ edges by connecting the clauses and variables to the corresponding assignments with $m + n$, and then connecting the n assignments we are using arbitrarily with $n - 1$ edges. Conversely, any Steiner tree with $m + 2n - 1$ edges can be converted to one using exactly n assignments, one per variable, that gives a satisfying solution for f.

We now proceed through the proof more carefully. We claim that f is satisfiable iff there is a Steiner tree of size $m + 2n - 1$.

Suppose we have a satisfying assignment for x. We take the corresponding n edges between variables and assignments. For each clause, we select an edge corresponding to the single-variable assignment that satisfied that clause. Then we add any spanning tree over the assignment vertices, which adds $n - 1$ edges. This gives a Steiner tree of size $m + 2n - 1$.

Now, consider any Steiner tree T in the graph. We claim that if $|T| \leq m+2n-1$, then f is satisfiable. We first claim that we can modify T so that each variable is incident to exactly one edge, and each clause is incident to exactly one edge, in T. Indeed, suppose T has both edges incident to a variable x_i. We can delete one of those edges and replace it with the edge between the two assignments $x_i = \texttt{true}$ and $x_i = \texttt{false}$, and T remains a Steiner tree. Likewise if a clause is two edges incident to two different satisfying vertex assignments, we can delete one of these edges and replace it with an edge between the assignments. See the diagrams below.



Thus, for any Steiner tree T, we may assume that each variable and each clause is

incident to exactly one edge. Consider just these $m+n$ edges, and the assignments that are at the opposite endpoints. Suppose there are k of them. None of these assignments are connected to each other by the $m+n$ edges, and is at least one for each variable x_j. Thus we have $k \geq n$ connected components, and the rest of T must have at least $k-1 \geq n-1$ edges to connect them. That is $|T| \geq m+n+k-1 \geq m+2n-1$. We have $|T| = m+2n-1$ iff $k = n$, iff we are using one assignment for every variable. These assignments give a satisfying assignment for the SAT formula. ■

13.4 Exercises

Please see [Eri19, Chapter 7] for further exercises.

Exercise 13.1. Let $G = (V, E)$ be a connected graph, and $F, S \subseteq E$. Decide if the following statements are true or false.

1. F is spanning iff for every cycle C in G with k edges, F contains at least $k-1$ edges. [2]
2. S is spanning iff S contains a spanning forest.
3. S is spanning iff $|S| \geq n-1$.
4. G has distinct edge weights if G has a unique MST.
5. If $F \subseteq \text{span}(S)$, then $|F| \leq |S|$.
6. If F is a forest, and $F \subseteq \text{span}(S)$, then $|F| \leq |S|$.

Exercise 13.2. Consider the following algorithm that claims to compute the MST (in a connected graph).

`decremental-MST`

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in decreasing order of weight.
2. For $i = 1, 2, \ldots, n$:
   A. Unless $E - e_i$ is disconnected:
      1. $E \leftarrow E - e_i$.
3. Return E.

[2] To be clear, a cycle specifically means a circuit with no repeating vertices (except at the common vertex where the walk starts and ends).

Either

1. Describe an input graph where `decremental-MST` fails to find the MST, or
2. Prove that `decremental-MST` always returns the minimum weight spanning tree.

For simplicity you may focus on the setting where the edge weights are distinct.

Exercise 13.3. Let $G = (V, E)$ be an undirected graph with real-valued edge weights Recall that the MST minimizes the sum of edge weights over all spanning edge sets of G. Consider the problem of computing a spanning edge set $S \subseteq E$ minimizing the maximum edge weight, $\max_{e \in S} w(e)$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 13.4. Let $G = (V, E)$ be a undirected graph with distinct edge weights and let T be the MST. Suppose the weight of a single edge $e \in E$ is altered. The edge e may or may not be in T, and the weight may have been increased or decreased. Describe and analyze an algorithm that fixes the MST.[3]

Exercise 13.5. Consider the problem of computing the second smallest weight spanning tree of a weighted graph G. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 13.6. Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge lengths. The *bottleneck length* of a path in G is the length of the longest edge in the path. Prove that the MST of G contains the shortest bottleneck path between every pair of points.

Exercise 13.7. Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge weights $w : E \to \mathbb{R}$. For a spanning tree T, we say that the *bottleneck weight* of T is the maximum weight edge in T, $\max_{e \in T} w(e)$. Consider the problem of computing the minimum

1. Prove that the MST is also a minimum bottleneck weight spanning tree of G.

[3] Of course one could build an entirely new MST from scratch. Can one do better?

2. Design and analyze a $O(m+n)$-time algorithm for computing a minimum bottleneck weight spanning tree of G. (This is faster than any of our algorithms for MST.)[4]

Exercise 13.8. Recall the notion of "bottleneck shortest paths" from exercise 13.6. Design and analyze a $O(m+n)$-time algorithm for computing the shortest bottleneck (s,t)-path.

Exercise 13.9. Of the multiple algorithms we have seen for MST, recall the parallel approach (Borůvka's algorithm) and the search approach (Prim's algorithm) with Fibonacci heaps. There is also a hybrid approach running in $O\big((m+n)\log\log n\big)$ time where one runs the parallel algorithm for some number of iterations (say, k iterations for a parameter k TBD), and then runs the search algorithm thereafter. (Such an algorithm is correct because ultimately it only takes good edges.) Describe and analyze this algorithm.[5]

Exercise 13.10. Let $G=(V,E)$ be a connected graph with distinct edge weights and $k<n$. We define the k-MWF as the minimum weight forest among the forests with exactly k edges. Consider the problem of computing the k-MWF.

1. Call an edge e *k-great* if for every set of edges $F \subseteq E-e$ that (a) has size $|F| \le k$ and (b) spans e (and doesn't include e),

$$w(e) < \max_{f\in F} w\big(f\big).$$

   Prove that e is k-great iff every k-MWF contains e.

2. Design an analyze an algorithm for computing the k-MWF.

Exercise 13.11. For $p>0$, the L_p-norm of a vector $x\in\mathbb{R}^k$ is defined by

$$\|x\|_p \stackrel{\text{def}}{=} \left(\sum_{i=1}^{k}|x_i|^p\right)^{1/p}.$$

Two edge cases are $p=0$ and $p=\infty$: here we define $\|x\|_0$ as the number of indices i such that $x_i\neq 0$, and

$$\|x\|_\infty \stackrel{\text{def}}{=} \max_i |x_i|.$$

[4]Here's step 1: compute the median edge weight in $O(m)$ time.

[5]It may be helpful to obtain a running time generically as a function of k, and then optimize k.

Let $G = (V, E)$ be an undirected graph with real-valued edge lengths $\ell : E \to \mathbb{R}$. We can take inspiration from L_p-norms to define the L_p-distance in graphs. For a walk w with edges $e_1, \ldots, e_k$, and $p > 0$, we define the L_p-norm of w as the L_p-norm of the vector of edge weights in w,

$$\|w\|_p \stackrel{\text{def}}{=} \left(\sum_{i=1}^{k} |\ell(e_i)|^p \right)^{1/p}.$$

Analogously we define the L_0 and L_∞ norms of w based on the definition for vectors. We define the L_p-distance (for $p > 0$, $p = 0$, and $p = \infty$) between two vertices s and t is the infimum of $\|w\|_p$ over all walks from s to t.

Let $s, t \in V$ be fixed. Below, for different values of p, we consider the problem of computing the L_p-distance from s to t in G. For each of these problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. (2 pts.) The L_0-distance from s to t.
2. (2 pts.) The $L_{1/2}$-distance from s to t.
3. (2 pts.) The L_1-distance from s to t.
4. (2 pts.) The L_2-distance from s to t.
5. (2 pts.) The L_∞-distance from s to t.