SS 2022/23

Algorithms

Chapter 5.1
Minimum Spanning Trees

Martin Dietzfelbinger

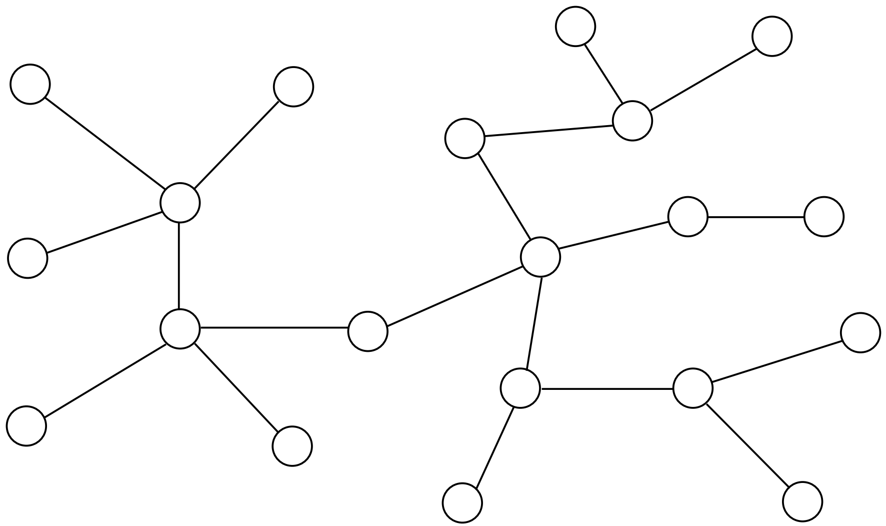
July 2022

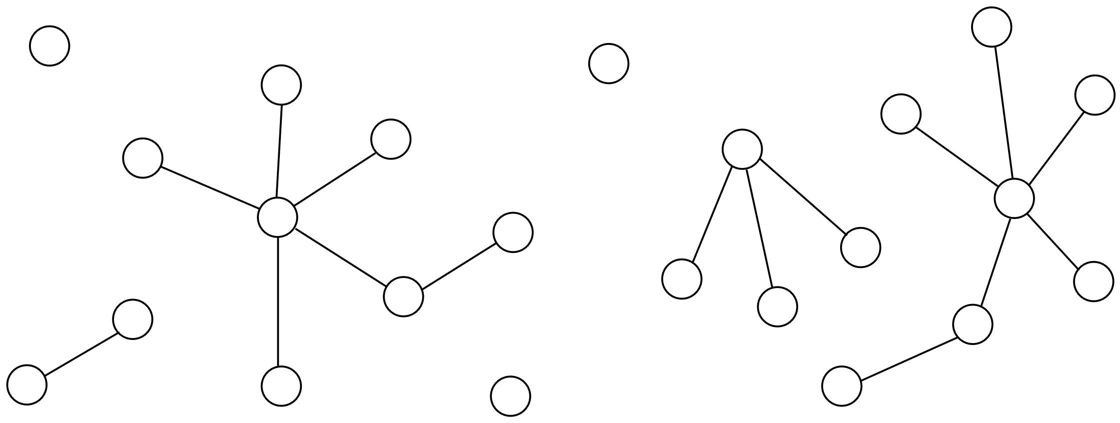
5.1.1 Basics

Reminder

(a) An undirected graph $G = (V, E)$ is called **acyclic** if there is no cycle in G.

(b) A graph G is called a **(free) tree** if it is connected and acyclic. – *Example*:





Acyclic graphs are also called **(free) forests**.

Fundamental facts about trees

If $G = (V, E)$ is a tree with n nodes and m edges, the following statements hold:

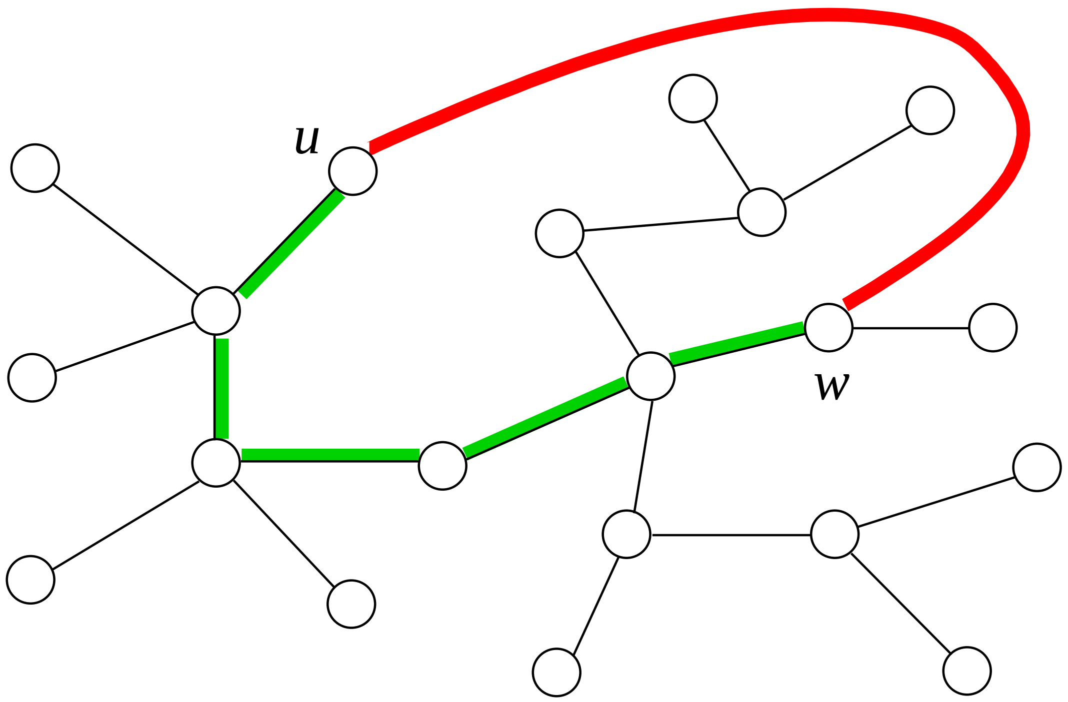
(a) $m = n - 1$.
(b) For each pair u, v of nodes there is exactly one simple path from u to v.

Furthermore we have:

If $G = (V, E)$ is a graph with $n - 1$ edges and it is acyclic, it is a tree.

If $G = (V, E)$ is a graph with $n - 1$ edges and it is connected, it is a tree.

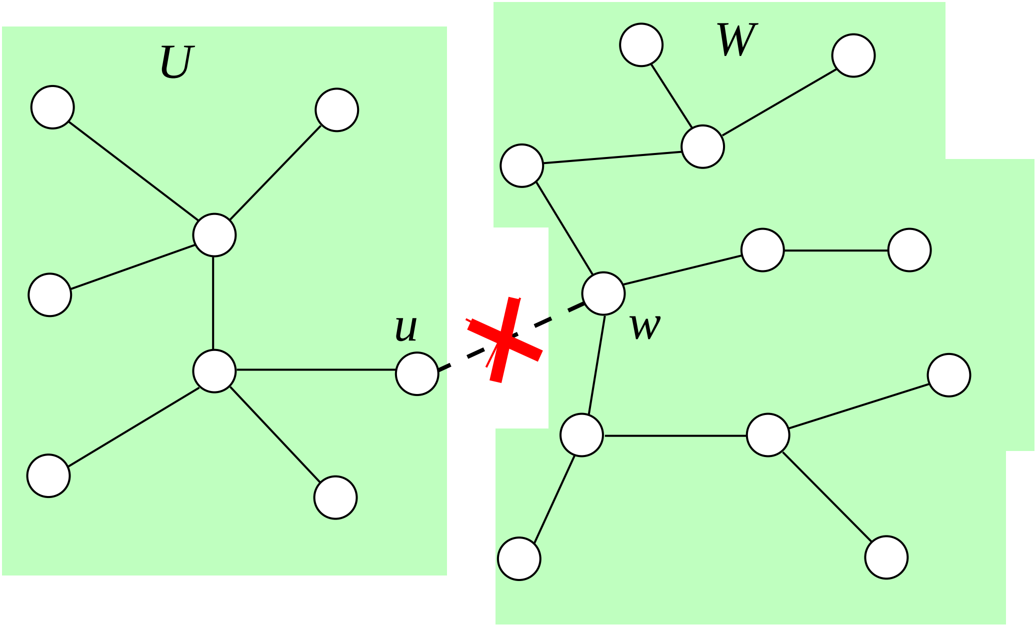
(1) Adding an edge (u, w) to a tree G creates exactly one cycle (consisting of (u, w) and the unique **path** from u to w in G).



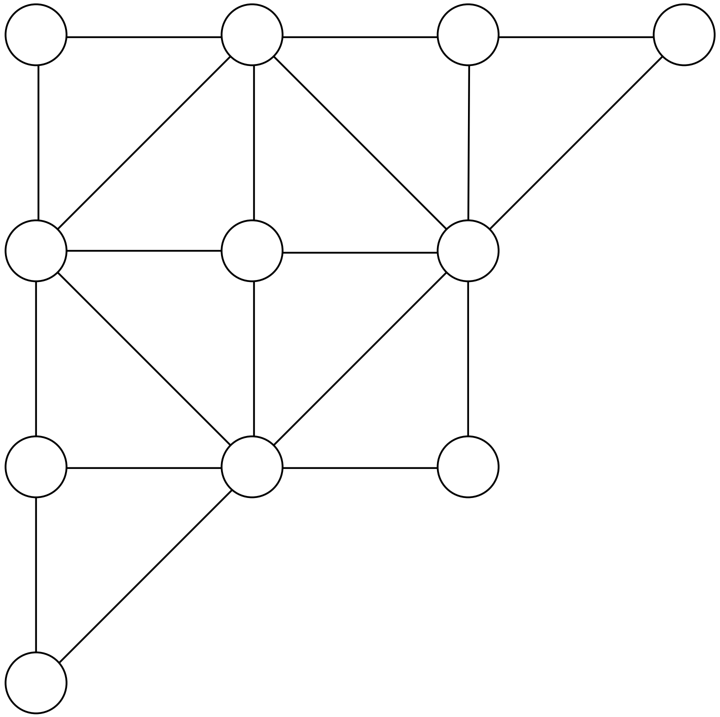
(2) Removing an edge (u, w) from G makes the graph split in 2 components:

$U = \{v \in V \mid \ v$ reachable from u via edges in $E - \{(u, w)\}\}$;

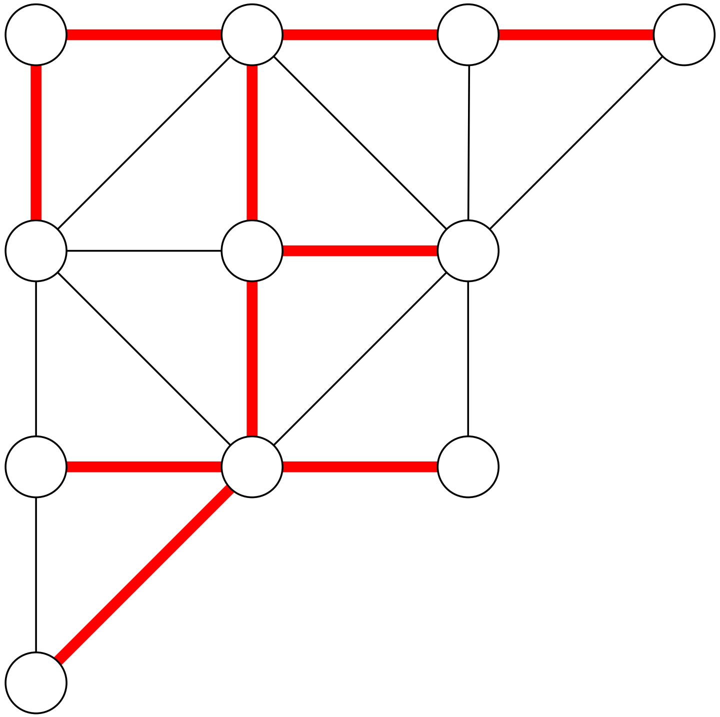
$W = \{v \in V \mid \ v$ reachable from w via edges in $E - \{(u, w)\}\}$;



Example:

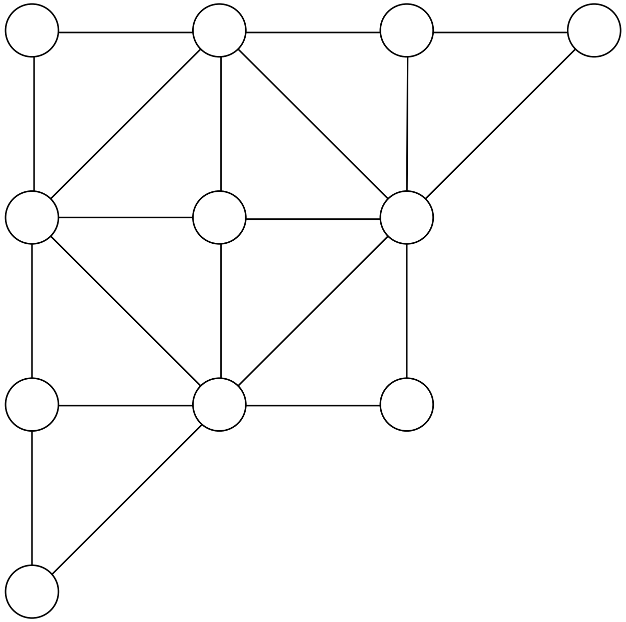


Example:



Definition 5.1.1

For $G = (V, E)$ a connected graph a set $T \subseteq E$ of edges is called a **spanning tree** for G if (V, T) is a tree.



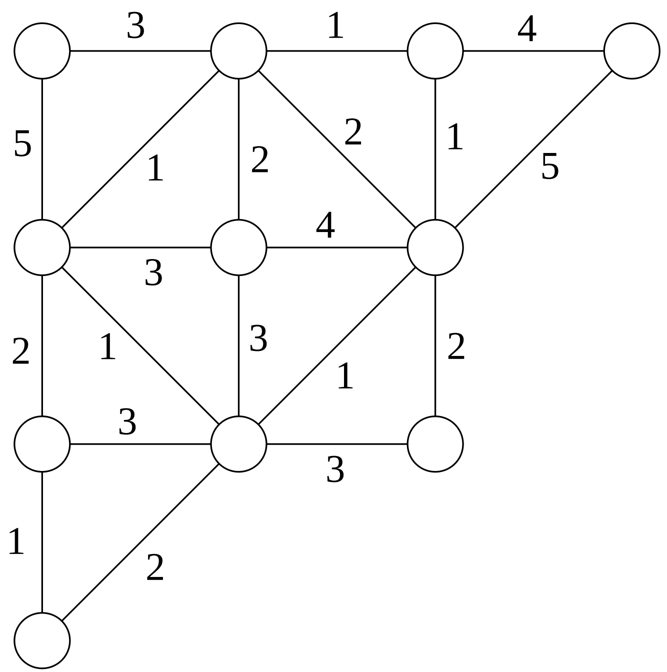
Observe: Every connected graph has a spanning tree.

(Start with E. While there is a cycle, remove some edge from some cycle.
At some point: no cycle is left.

Taking away a cycle edge never destroys connectedness, so the final result is connected and acyclic: a tree.)

Definition 5.1.2

Let $G = (V, E, c)$ be a **weighted graph**, i.e. $c\colon E \to \mathbb{R}$ is a "*weight function*" or "*cost function*".



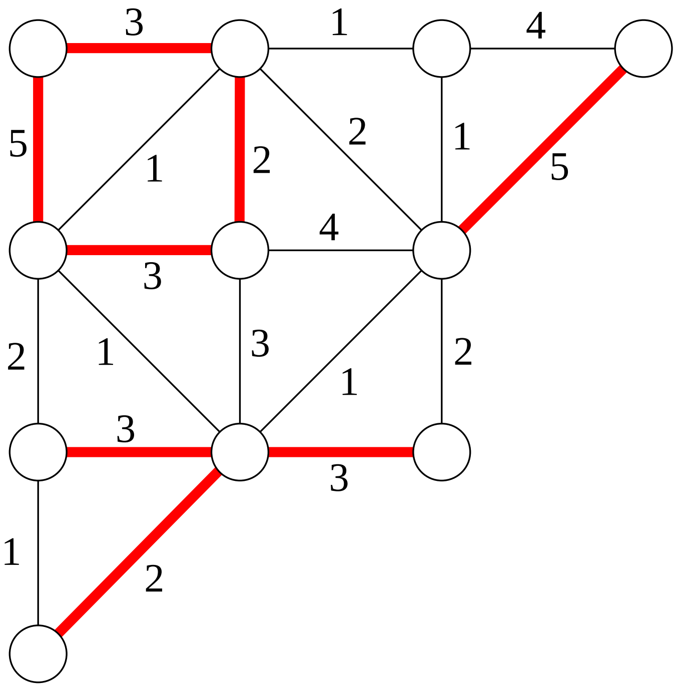
Weighted graphs model: road networks – computer networks – electric power networks . . .

edge costs model: building cost – cost for leasing cable use – cost for leasing equipment for transmitting data via radio waves . . .

Btw: Multiply edge weights by "million Euros".

(a) The **(total) weight** of a subset $E' \subseteq E$ of edges is defined as

$$\boldsymbol{c(E')} := \sum_{e \in E'} c(e).$$

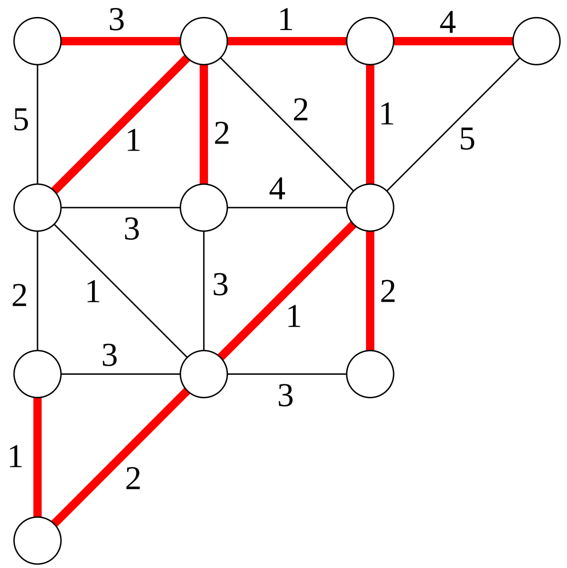


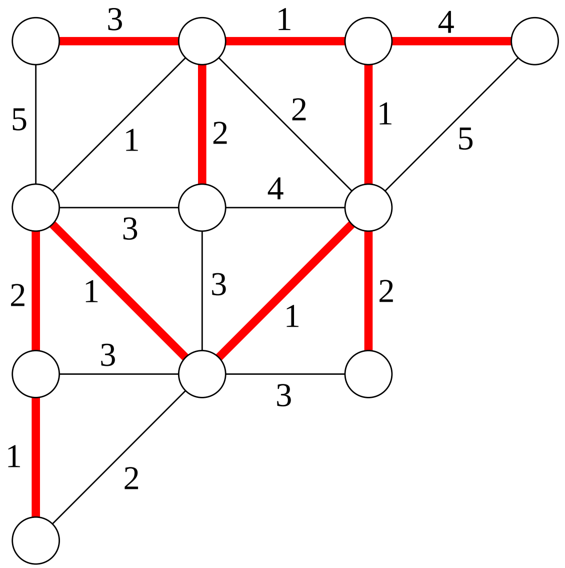
Total weight $c(\boldsymbol{E'}) = 3 + 5 + 2 + 5 + 3 + 3 + 3 + 2 = 26.$

(b) Let G be a connected graph. A spanning tree $T \subseteq E$ for G is called a **minimum spanning tree (MST)** for G if

$$c(T) = \min\{c(T') \mid T' \text{ spanning tree of } G\},$$

i.e. if $c(T)$ is minimal among all spanning trees of G.





Two MSTs, both with total weight 18.

Obvious: Each graph has an MST.
(There are only finitely many spanning trees.)

Beware: There may be several different MSTs (with the same weight, of course).

Task: Given $G = (V, E, c)$, find an MST T for G.

Here: **"Jarník/Prim algorithm"***
"Kruskal's algorithm"**

Typical for the algorithm paradigm **"greedy"**:

Build solution **step by step**, choosing one edge after the other.

In each step make the decision that **momentarily looks best**.

Never undo a decision.

---

* Invented 1930 by Vojtěch **Jarník**, re-invented 1957 by Robert C. **Prim** and 1959 by Edsger W. **Dijkstra**.

** Invented 1956 by Joseph **Kruskal**.

5.1.2 Jarník/Prim algorithm

S: Set S of nodes, the nodes "reached so far".
R: Set R of edges, the edges "chosen so far".

(1) Choose an arbitrary (start) node $s \in V$.
$\mathtt{S} \leftarrow \{s\}$; $\mathtt{R} \leftarrow \emptyset$;

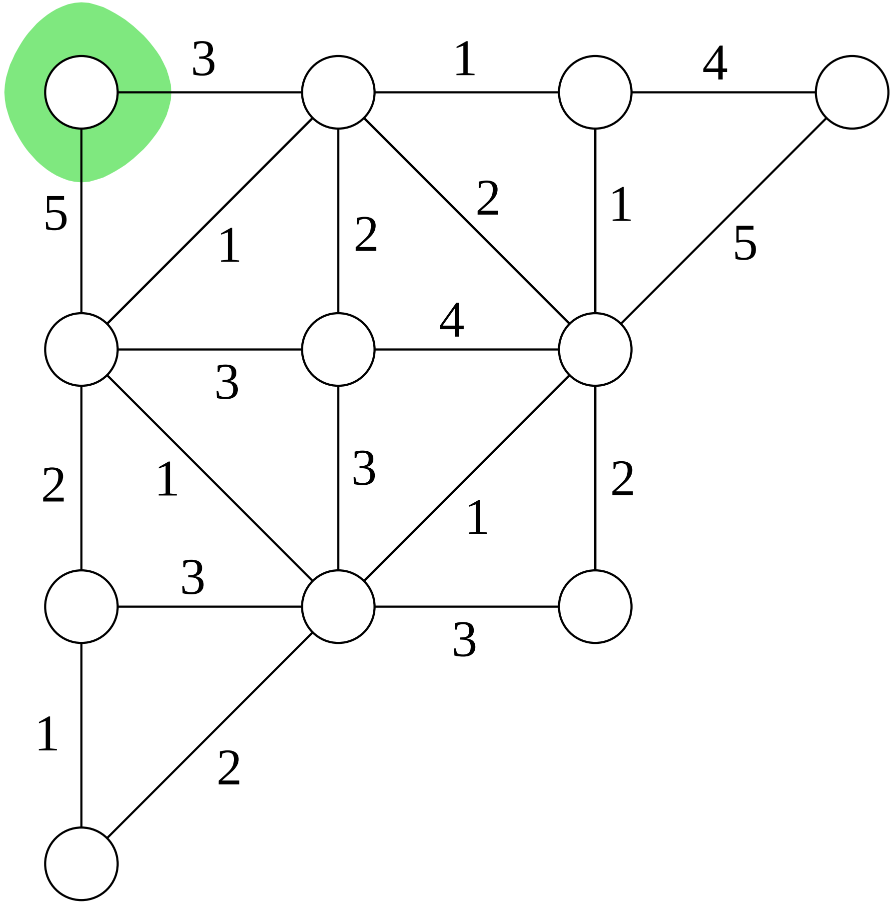
(2) Repeat $(n-1)$ times:

Find $w \in \mathtt{S}$ and $u \in V - \mathtt{S}$ s.t.
$c(w,u)$ is **minimal** among all values $c(w',u')$, $w' \in \mathtt{S}$, $u' \in V - \mathtt{S}$.

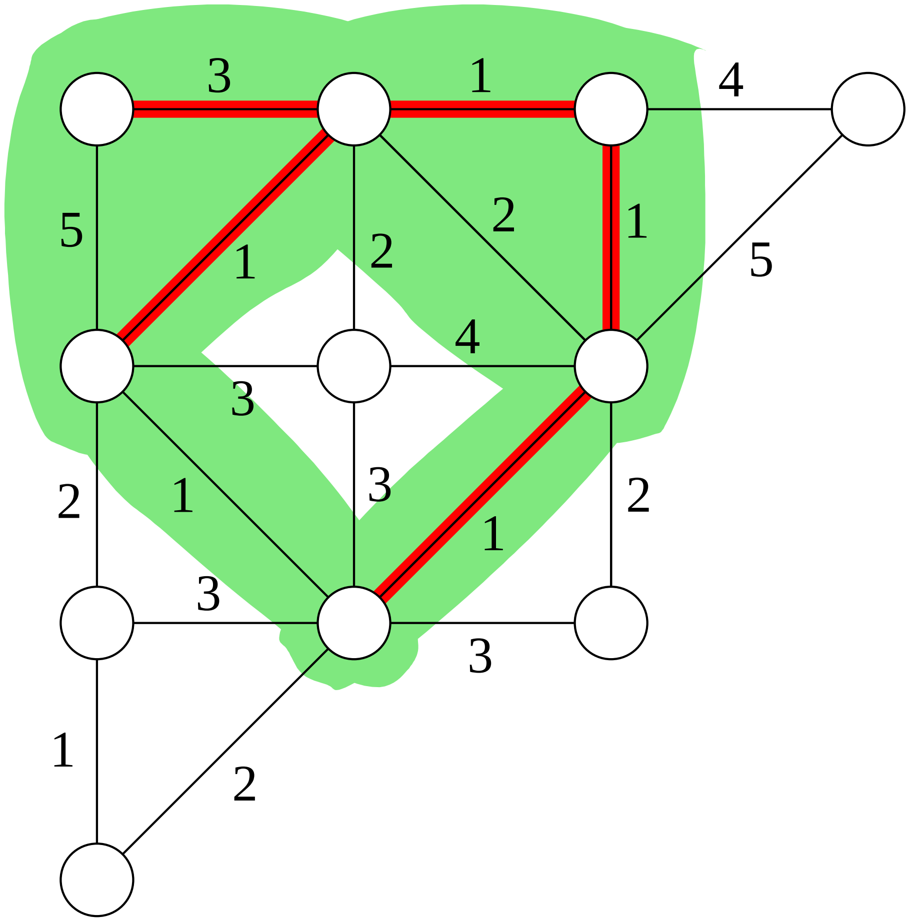
$\mathtt{S} \leftarrow \mathtt{S} \cup \{u\}$; // add node to S
$\mathtt{R} \leftarrow \mathtt{R} \cup \{(w,u)\}$; // add edge to R

(3) Output: R.

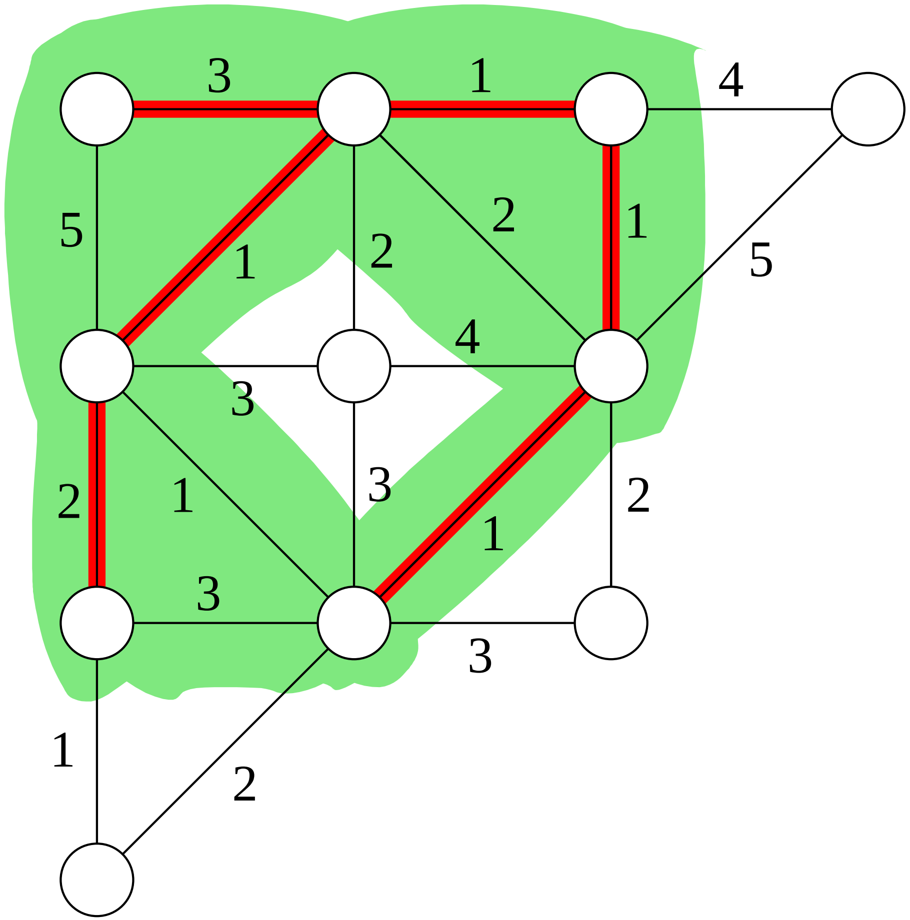
Example (Jarník/Prim):



Example (Jarník/Prim):



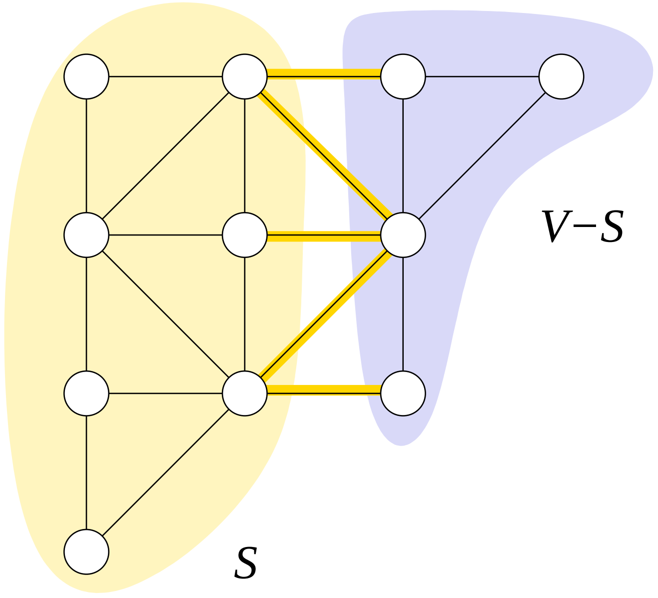
Example (Jarník/Prim):



The cut property

For proving the algorithm of Jarník/Prim correct we use the **"cut property"**

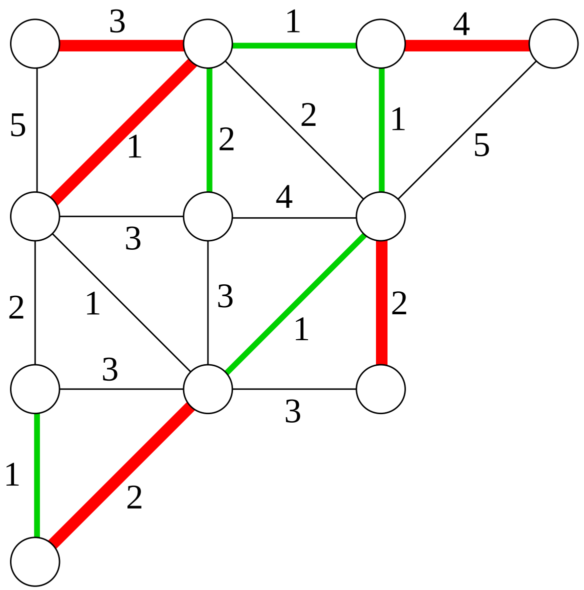
A partition $(S, V - S)$ with $\emptyset \neq S \neq V$ is called a **cut**.



The cut property

Definition 5.1.3

A set $R \subseteq E$ is called **extendible** (to an MST), if there is an MST T s.t. $R \subseteq T$.

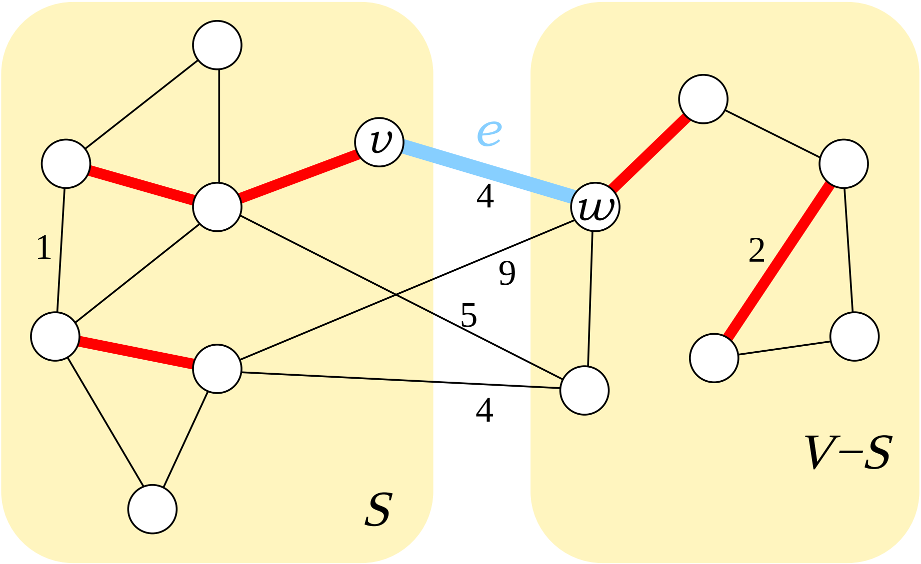


$\boldsymbol{R}$ is extendible, because there is an MST $\boldsymbol{T} \supseteq \boldsymbol{R}$.

Claim (Cut Property):

Assume $\boldsymbol{R} \subseteq E$ is extendible
and $(S, V-S)$ is a cut s.t.
there is no edge in $\boldsymbol{R}$ from a node in S to a node in $V-S$,
and assume that $e = (v, w)$, $v \in S$, $w \in V-S$ is an edge
that **minimizes** $c((v', w'))$, $v' \in S$, $w' \in V-S$.

Then $\boldsymbol{R} \cup \{e\}$ is also extendible.



The cut property

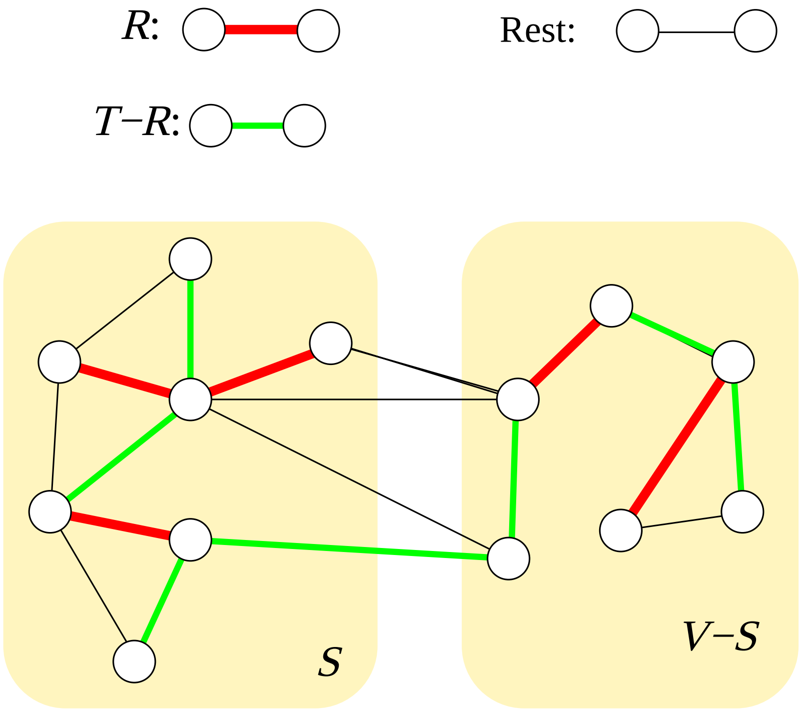
Proof:

Let $R \subseteq E$, let $T \supseteq R$ be an MST; let $(S, V - S)$ be a cut, let e be as assumed.

Case 1: If $e \in T$, we have $R \cup \{e\} \subseteq T$, hence $R \cup \{e\}$ is extendible.

The cut property

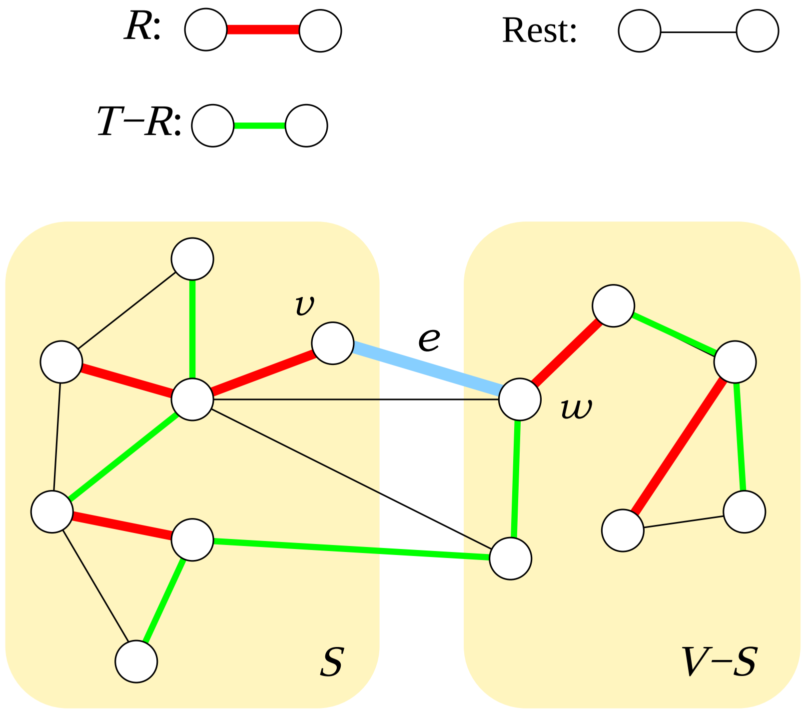
Case 2: $e \notin T$.



MST T with $R \subseteq T$.

The cut property

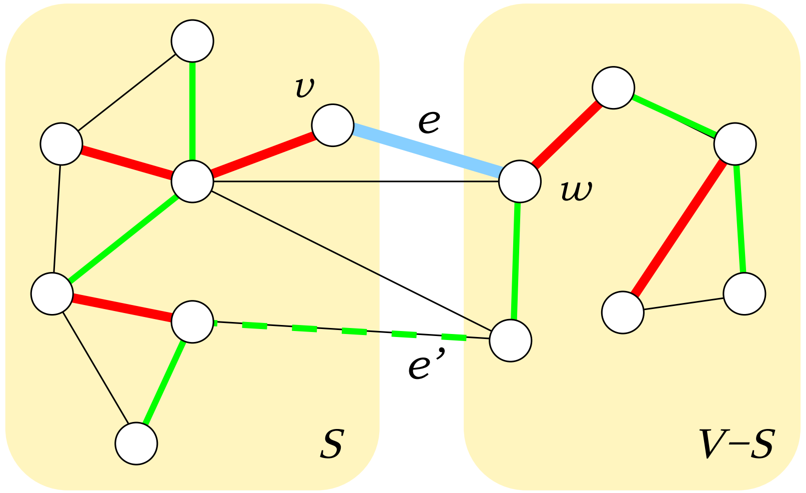
Case 2: $e \notin T$.



$e = (v, w)$ minimizes $c((v', w'))$, $v' \in S$, $w' \in V - S$.

The cut property

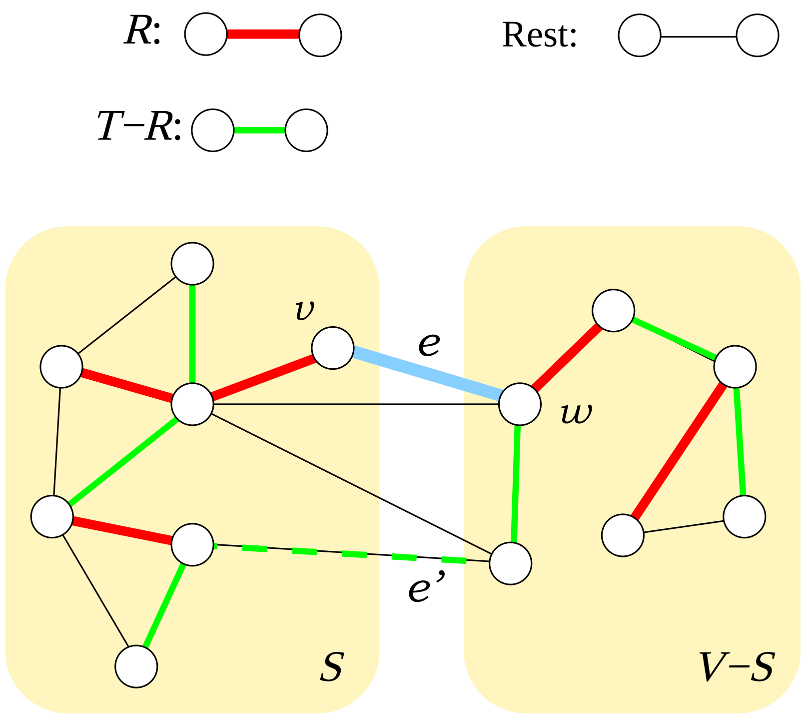
Case 2: $e \notin T$.



Path from v to w in T must change from S to $V - S$ at some edge e'.
We obtain a cycle in $T \cup \{e\}$ with e and e' on it.

The cut property

Case 2: $e \notin T$.



New tree $T_e := (T - \{e'\}) \cup \{e\} \supseteq R \cup \{e\}$ is a spanning tree.
$c(T_e) - c(T) = c(e) - c(e') \leq 0$, hence T_e also optimal, hence $R \cup \{e\}$ is extendible.$\square$

Correctness of the Jarník/Prim algorithm:

R_i: Edge set (size i), after round i in R.
S_i: Node set (size $i+1$), after round i in S.

Since in every round an edge and a node is added to a connected graph, creating no cycles, every graph (S_i, R_i) is a tree.

Since R_{n-1} is a tree with $n-1$ edges, R_{n-1} is a spanning tree.

Must show: Minimality, i.e. R_{n-1} is an MST for G.

Inductive claim IC(i): R_i is extendible.
(This is easily proved by induction on $i = 0, 1, \ldots, n-1$, with the cut property.)

Then IC$(n-1)$ says that $T \supseteq R_{n-1}$ for some MST T.
Since $|T| = n - 1 = |R_{n-1}|$, we get $T = R_{n-1}$, hence R_{n-1} is an MST.

Missing: Details of the implementation. There is a great similarity with Dijkstra's algorithm. (We use a priority queue, for $w \notin S$ the value `dist`[w] is the length of an edge (v, w) with $v \in S$ that minimizes $c((v, w))$.)

Theorem 5.1.1

The Jarník/Prim algorithm can be implemented using a priority queue, realized as a binary heap. Then it finds a minimum spanning tree for any given weighted connected graph $G = (V, E, c)$ in time $\boldsymbol{O(m \log n)}$, or $\boldsymbol{O(|E| \log |V|)}$.

Jarnik/Prim(G, s) // (full version with priority queue)

Input: Weighted connected graph $G = (V, E, c)$, $V = \{1, \ldots, n\}$, $s \in V$ (arbitrary);
Output: MST T for G.
Auxiliary structures: PQ: priority queue, initially empty; inS$[1..n]$, p$[1..n]$: as above

for w **from** 1 **to** n **do**
    dist[w] $\leftarrow \infty$; inS[w] $\leftarrow$ *false*; p[w] $\leftarrow -1$;
dist[s] $\leftarrow 0$; p[s] $\leftarrow -2$; PQ.insert(s);
while not PQ.isempty **do**
    u $\leftarrow$ PQ.extractMin; inS[u] $\leftarrow$ *true*;
    for all vertices w with (u, w) $\in E$ **and not** inS[w] **do**
        dd $\leftarrow c$(u, w); // the only difference to Dijkstra's algorithm!
        if p[w] ≥ 0 **and** dd $<$ dist[w] **then**
            dist[w] $\leftarrow$ dd; PQ.decreaseKey(w,dd); p[w] $\leftarrow$ u;
        if p[w] $= -1$ **then** // w is found
            dist[w] $\leftarrow$ dd; p[w] $\leftarrow$ u; PQ.insert(w);
Ausgabe: $T = \{(w, \mathrm{p}[w]) \mid \mathrm{inS}[w] = \textit{true}, w \neq s\}$. // set of the chosen edges

5.1.3 Kruskal's algorithm

This algorithm also solves the **MST problem**.

We use a different method than Jarník/Prim, but also "greedy":
Start with $\boldsymbol{R} = \emptyset$. Then do $n-1$ rounds.

In each round:

Choose an edge $\boldsymbol{e} \in E - \boldsymbol{R}$ of minimum weight that does not close a cycle with $(V, \boldsymbol{R})$, and add e to $\boldsymbol{R}$.

It is clear that one can organize this as follows:

Scan the edges in increasing order of their weight. Add e to $\boldsymbol{R}$ if and only if e does not close a cycle with the current $\boldsymbol{R}$.

Kruskal's algorithm

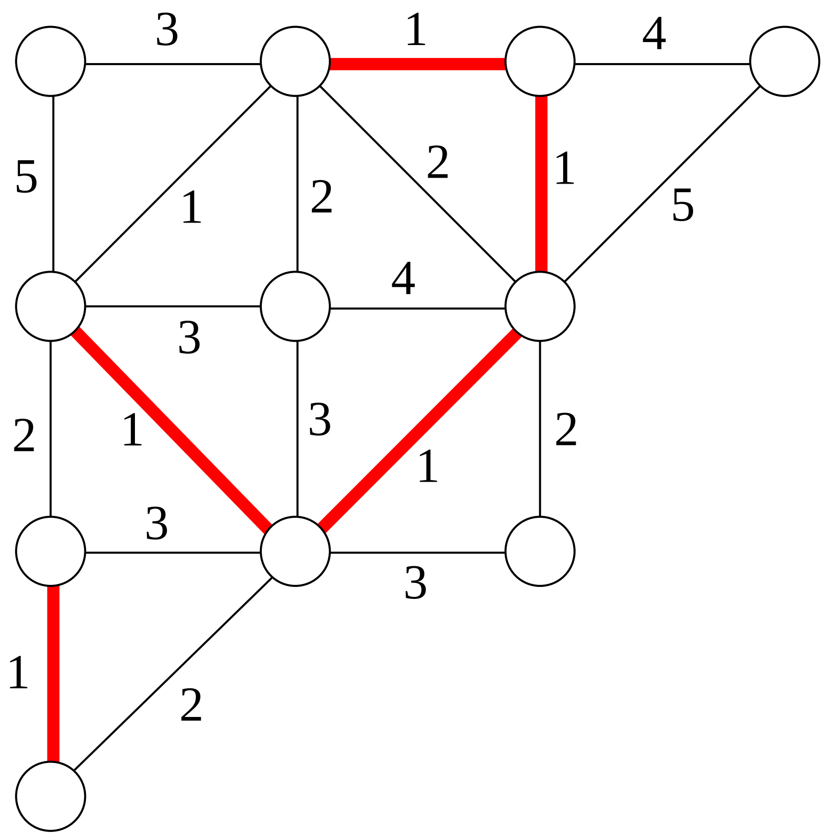
Step 1: Sort edges $e_1, \ldots, e_m$ according to their weights $c(e_1), \ldots, c(e_m)$ in increasing order, and re-label.
Afterwards: $c(e_1) \leq \cdots \leq c(e_m)$.

Step 2: $\mathtt{R} \leftarrow \emptyset$.

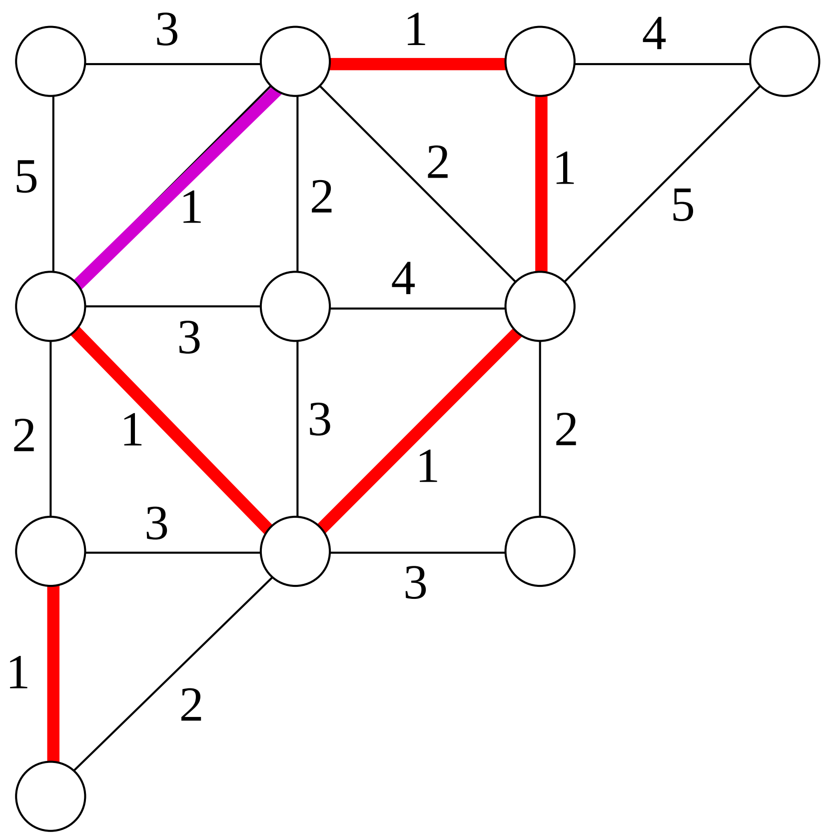
Step 3: **for** $i = 1, 2, \ldots, m$ **do**
    if $\mathtt{R} \cup \{e_i\}$ is acyclic **then** $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$
        // otherwise, i.e. if e_i closes a cycle, R does not change.
    // Optional: End loop as soon as $|\mathtt{R}| = n - 1$.

Step 4: Output R.

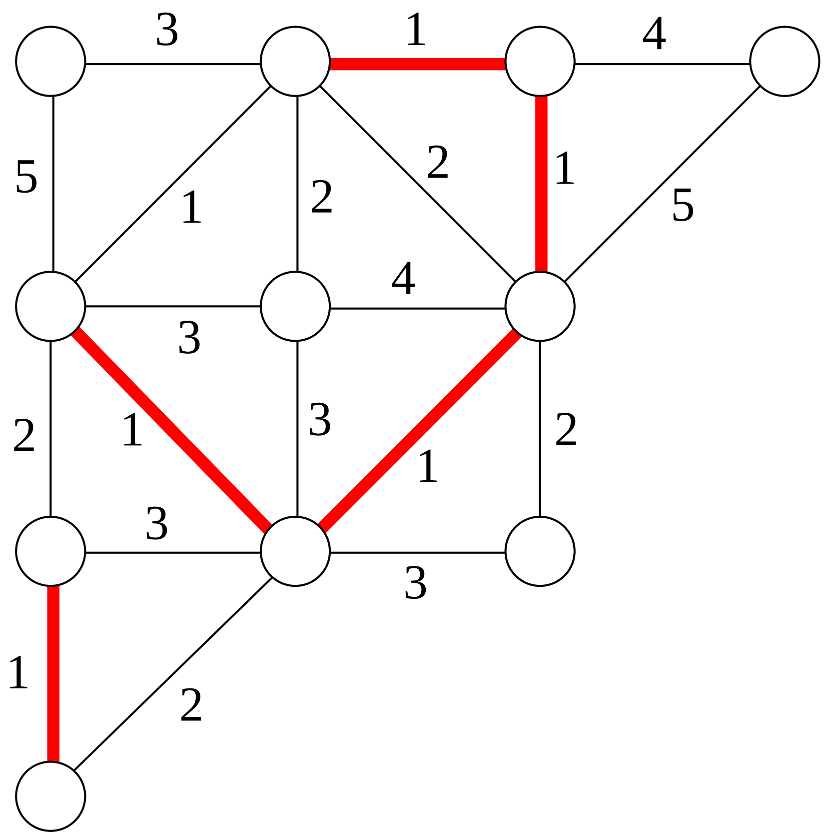
Example (Kruskal):



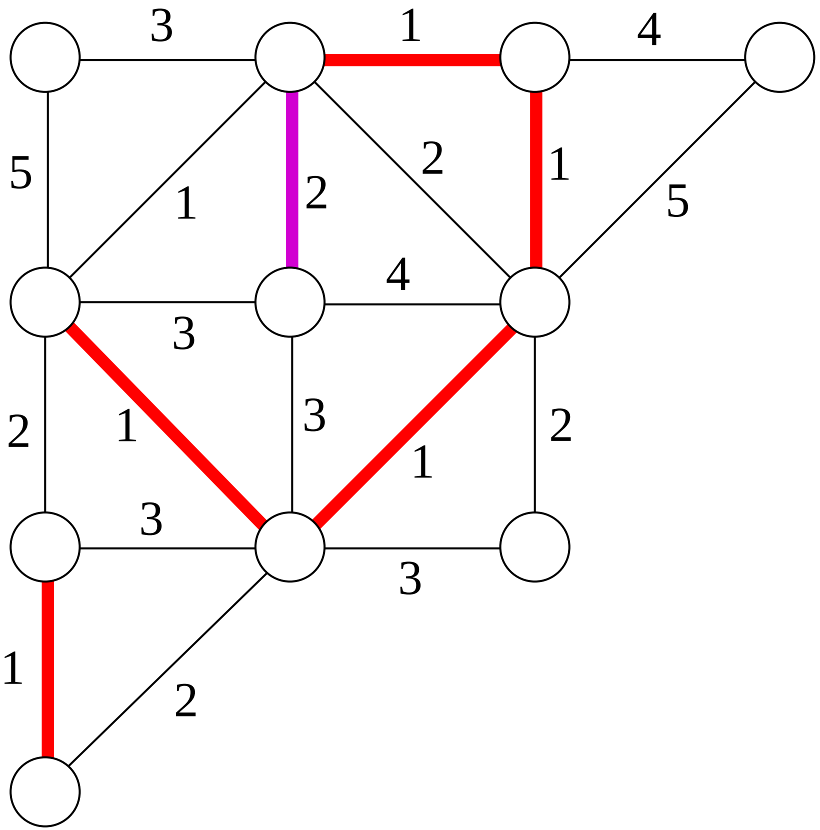
Example (Kruskal):



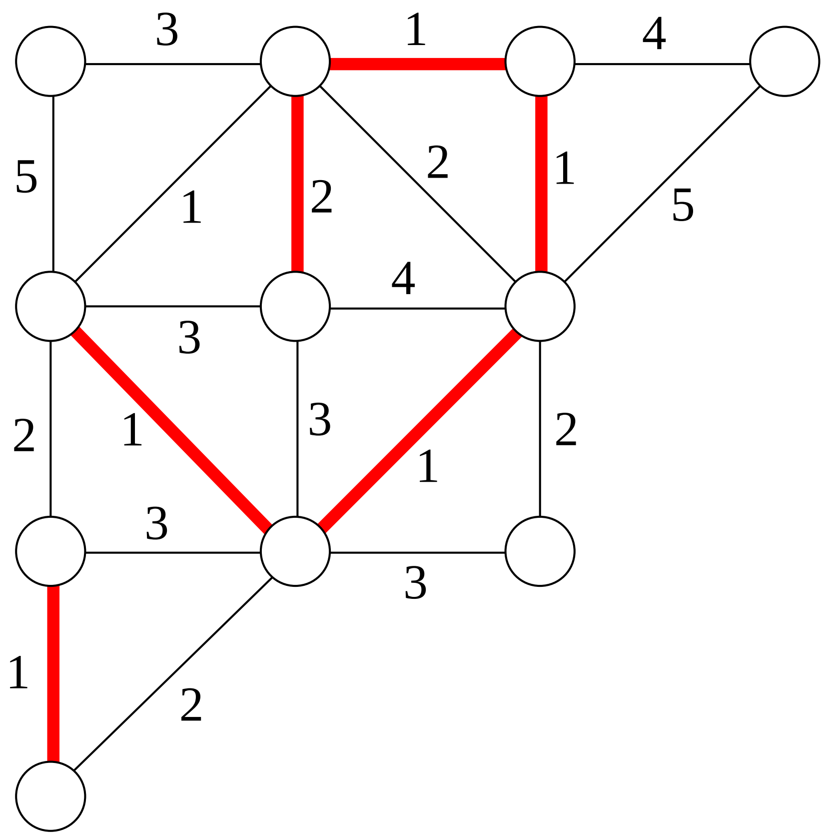
Example (Kruskal):



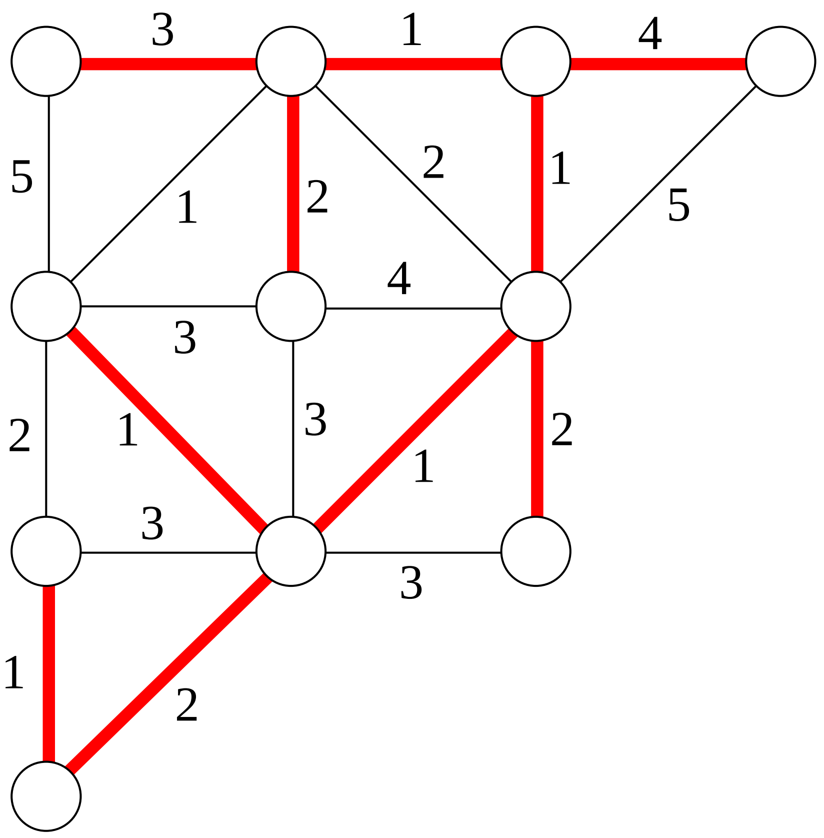
Example (Kruskal):



Example (Kruskal):



Example (Kruskal):



Remarks on 1) Correctness; 2) Computation time

Correctness proof:

R_i: Edge set in R after treating e_i.

One shows by induction on $i = 0, \ldots, m$: R_i is extendible. (Not hard with the cut property.)

Then $R_m \subseteq T$ for an MST T.
But R_m is also connected, hence it is a tree, hence $R_m = T$.

Induction Claim IC(i): R_i is extendible.

Proof:

Basis: $R_0 = \emptyset$ is extendible (since there are MSTs).
I.H.: $1 \leq i \leq m$ and R_{i-1} is extendible.
I.S.: We execute round i with edge e_i.

Case 1: $R_{i-1} \cup \{e_i\}$ has a cycle. Then $R_{i-1} = R_i$ is extendible.

2. Fall: $R_{i-1} \cup \{e_i\}$ is acyclic. – Let $e_i = (v, w)$. Define

$S :=$ Connected component of v in (V, R_{i-1}).

Then obviously no edge in R_{i-1} connects S and $V - S$.

Since $R_{i-1} \cup \{e_i\}$ is acyclic, we have $w \in V - S$.

Easy: $c(e_i)$ is minimal among all $c(e')$ with $e' = (v', w')$, $v' \in S$, $w' \in V - S$.

By the **cut property** we get: $R_i = R_{i-1} \cup \{e_i\}$ is extendible. □

IC(m) says that $R_m \subseteq T$ for some MST T.

But we also have $T \subseteq R_m$.

(Let $e \in T$. Then $e = e_i$ for some i and e is tested in round i.
Now $R_{i-1} \subseteq R_m \subseteq T$ and $e_i \in T$, hence $R_{i-1} \cup \{e_i\} \subseteq T$, hence $R_{i-1} \cup \{e_i\}$ is acyclic, hence the algorithm puts e_i into R_i, hence $e = e_i \in R_m$.)

So $R_m = T$, and R_m is an MST.

Computation time:

With a suitable data structure ("Union-Find data structure", implementation with trees, see below) the acyclicity test in Step 3 can be carried out in time $O(\log n)$.

Total time for Kruskal's algorithm:

$$\underbrace{O(m \log m)}_{\text{sorting}} + \underbrace{m \cdot O(\log n)}_{\text{loop}} = O(m \log n).$$

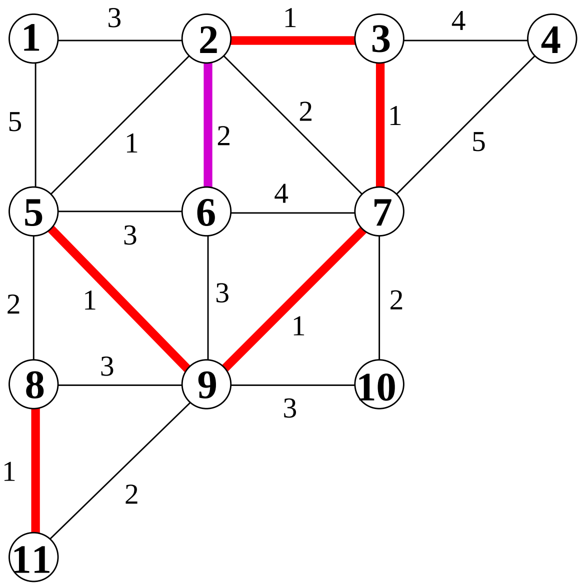
(Details to follow. Note that $n - 1 \leq m < n^2/2$, hence $\log_2 m = \Theta(\log_2 n)$.)

5.1.4 Auxiliary data structure: Union-Find

Union-Find data structures are used as an **auxiliary structure** for several algorithms, in particular for Kruskal's algorithm.

Intermediate situation in Kruskal's algorithm:
Set $R = R_{i-1} \subseteq E$, so that (V, R) is a forest.
Next edge: $e = e_i = (v, w)$.
Must decide whether e closes a cycle with R.

I.e.:



For two nodes v and w decide whether (V, R) contains a path from v to w.
(Here: $v = 2$, $w = 6$.)
Possible, but slow: do depth-first-search in (V, R) each time.

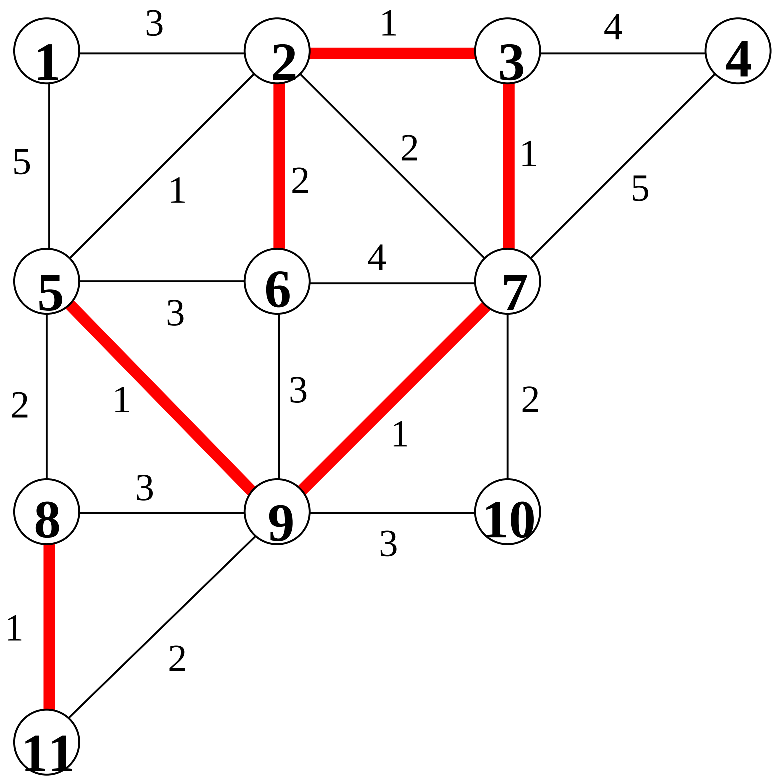
Clever: We do not need to represent the edges in R, only the **node sets** of the connected components of (V, R).

In the picture this are the sets

$\{1\}, \{2, 3, 5, 7, 9\}, \{4\}, \{6\}, \{8, 11\}, \{10\}$.

We wish to figure out (fast) whether **two nodes are in the same component (set)**.

When the algorithm puts a new edge into R, we have to form the **union** of two of the sets (and throw away the two old sets).



New sets: $\{1\}, \{2, 3, 5, 6, 7, 9\}, \{4\}, \{8, 11\}, \{10\}$.

This operation also should be fast.

Abstract task:

A **partition** of $V = \{1, 2, \ldots, n\}$ consists of disjoint nonempty subsets of V, whose union is V:

$$V = \{1, 2, \ldots, n\} = S_1 \cup S_2 \cup \cdots \cup S_\ell,$$

where $S_1, S_2, \ldots, S_\ell$ are **disjoint**.

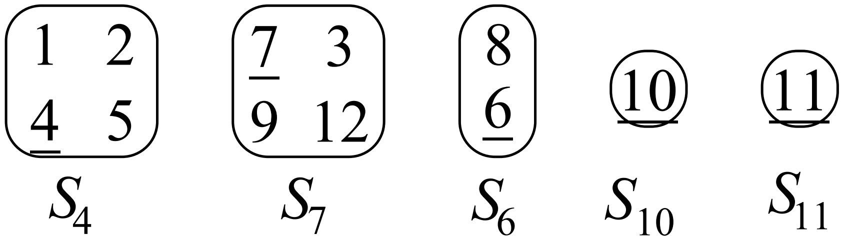
We consider **"dynamic"** partitions, which can be changed by operations.

Task: Maintain a "dynamic partition" of the set $\{1, 2, \ldots, n\}$ under operations

init	(Initialization)
union	(Union of two of the sets)
find	("In which set is v?").

Deviation from book: Our ground set is $V = \{1, 2, \ldots, n\}$, we do not use an insert operation.

Example: $n = 12$.



In each set S of the partition we have chosen a **representative** $r \in S$. This representative acts as S's **name**. We write $\boldsymbol{S_r}$ for the set with representative r.

Operations:

init(n): Given $n \geq 1$, generate the "discrete partition" with the n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. Thus: $S_v = \{v\}$ and $r(v) = v$, for $1 \leq v \leq n$.

find(v): Given $v \in \{1, \ldots, n\}$, return the representative $r(v)$ of the set $S_{r(v)}$ that (currently) contains v.

union(s, t): The arguments s and t must be representatives of **different classes** S_s and S_t. The operation removes S_s and S_t from the partition and adds $S_s \cup S_t$ to it. As representative of this new set $S_s \cup S_t$ use s or t.

In the example, **union**$(4, 10)$ removes the sets $S_4 = \{1, 2, \underline{4}, 5\}$ and $S_{10} = \{\underline{10}\}$ and adds $S'_{10} = \{1, 2, 4, 5, \underline{10}\}$.

Kruskal's algorithm with Union-Find data structure

Input: Weighted connected graph $G = (V, E, c)$ with $V = \{1, \ldots, n\}$.

Step 1: Sort edges $e_1, \ldots, e_m$ in increasing order $c_1 = c(e_1), \ldots, c_m = c(e_m)$ of weights.

Result: Sorted edge list $e_1 = (v_1, w_1, c_1), \ldots, e_m = (v_m, w_m, c_m)$.

Step 2: $\mathtt{R} \leftarrow \emptyset$; initialize Union-Find structure for $\{1, \ldots, n\}$.

Step 3: **for** $i = 1, 2, \ldots, m$ **do**:

$\mathtt{s} \leftarrow$ **find**(v_i); $\mathtt{t} \leftarrow$ **find**(w_i);
if $\mathtt{s} \neq \mathtt{t}$ **then begin** $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$; **union**$(\mathtt{s}, \mathtt{t})$ **end**;
// Optional: Quit loop as soon as $|\mathtt{R}| = n - 1$.

Step 4: **return** $\mathtt{R}$.

Theorem 5.1.2

(a) Kruskal's algorithm in the implementation just given is correct.

(b) The execution time of the algorithm is $O(m \log n)$ if one implements the Union-Find data structure with **trees**.

Proof: (a) (Correctness) One (easily) shows by induction that after i rounds the sets in the union-find structure are the connected components of $(V, \{e_1, \ldots, e_i\})$, and these are the same as the connected components of the forest (V, R_i) ($R_i =$ content of R after i rounds).

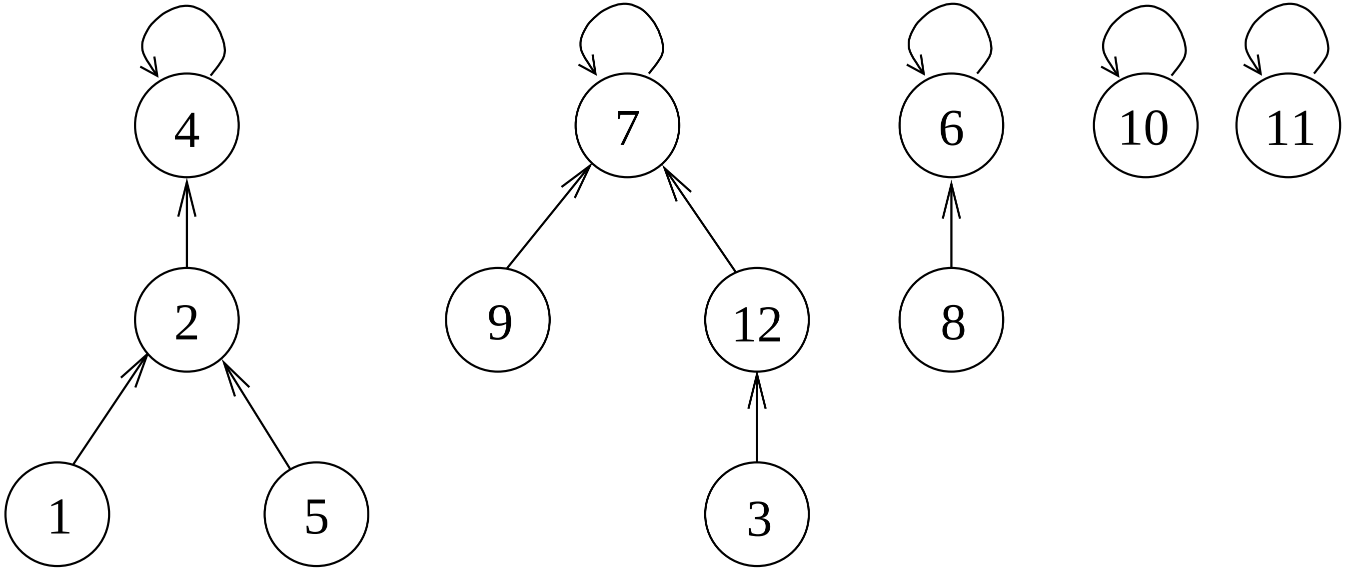
Hence "$\mathtt{s} \leftarrow$ **find**(v_i); $\mathtt{t} \leftarrow$ **find**(w_i); **if** $\mathtt{s} \neq \mathtt{t}$ ..." tests whether $e_i = (v_i, w_i)$ closes a cycle with (V, R_{i-1}).

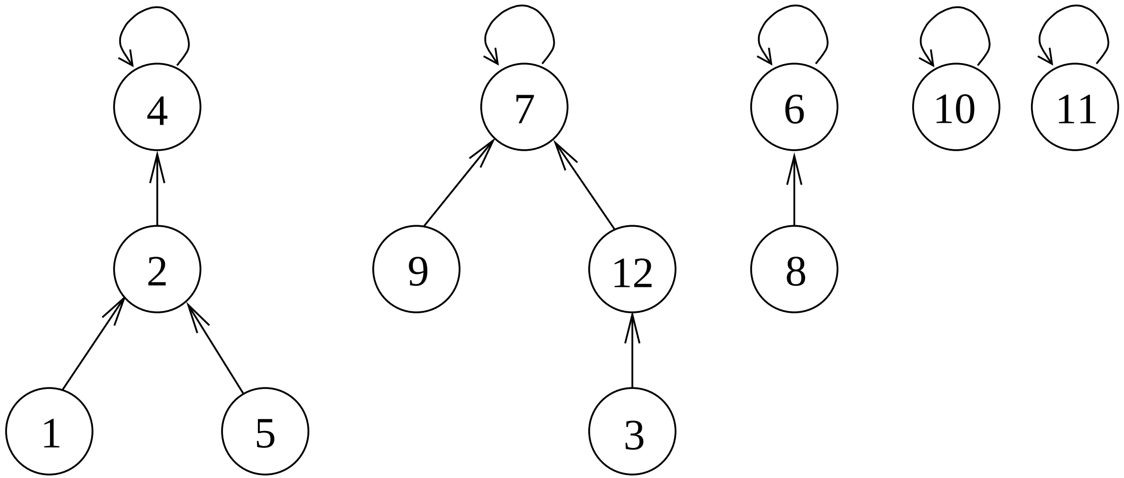
(b) (Execution time): see below.

Tree implementation of Union-Find

An attractive implementation of the Union-Find data structure uses a forest **with edges directed towards the roots**.

Example: Partition $\{1, 2, \underline{4}, 5\}, \{3, \underline{7}, 9, 12\}, \{\underline{6}, 8\}, \{\underline{10}\}, \{\underline{11}\}$ is represented by:





For each set S_t there is exactly one tree B_t.

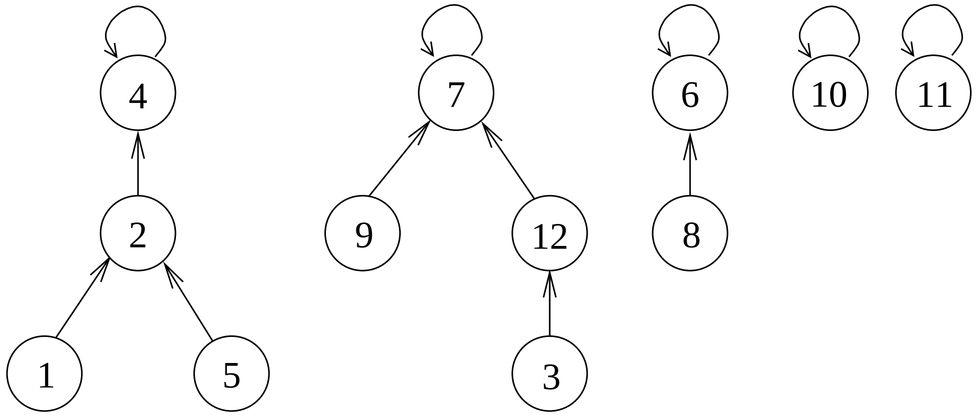
Each element $v \in S_t$ is a node in tree B_t.

In each tree, all arrows point towards the root:
$p(v)$ is the predecessor of v; the root is the representative r; the root points to itself as a predecessor: $p(v) = v$ if and only if v is a representative.

Central property: Starting at v, always following the arrows will get us to the root, i.e., the representative of v.

A very efficient representation of such a forest uses only an array
`p[1..n]: array of int`;
for node v the entry $\texttt{p}[v]$ gives the predecessor $p(v)$.

The forest in the example



is given by the following array:

	1	2	3	4	5	6	7	8	9	10	11	12
p :	2	4	12	4	2	6	7	6	7	10	11	7

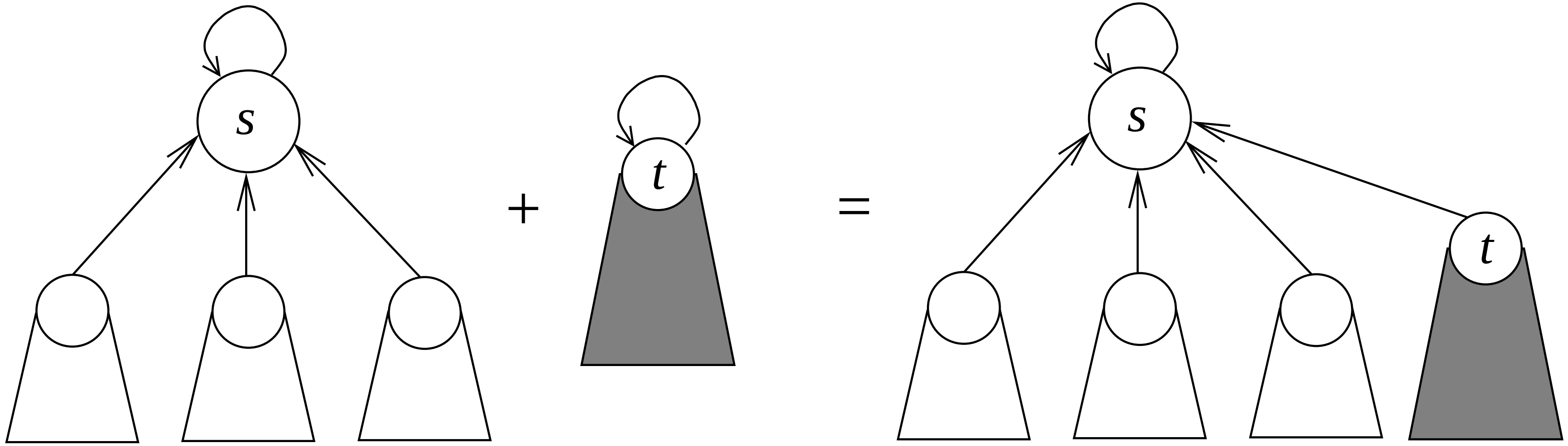
Implementation of **find**(v):

```
procedure find(v)
    u ← v;
    // follow arrows until root
    uu ← p[u];
    while uu ≠ u do
        u ← uu;
        uu ← p[u] ;
    return u.
```

Time: $\Theta($*depth*$(v)) = \Theta($depth of v in its tree$)$.

union(s,t): Given are two representatives s and t.

We make one of the representatives the child of the other one, i.e., we let $\mathrm{p}(s) \leftarrow t$ **or** $\mathrm{p}(t) \leftarrow s$.



? Which of the two options is preferable?

Bad choice: Carry out **union**$(v, v+1)$, $v = 1, \ldots, n-1$, by $\mathrm{p}(v) \leftarrow v+1$, $v = 1, \ldots, n-1$.

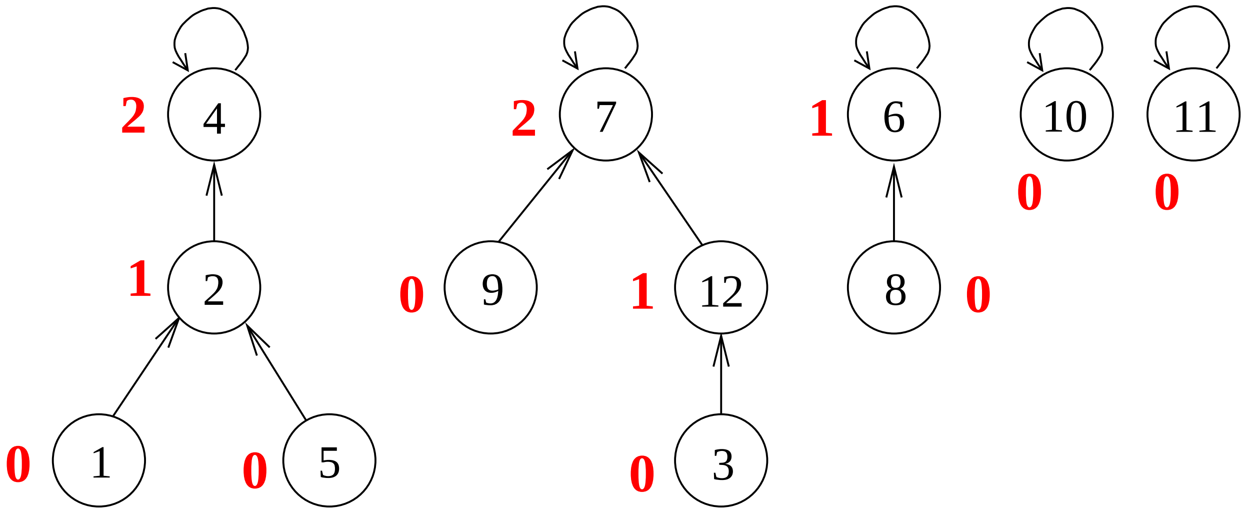
This gives the tree

Now **find** operations are very expensive!

find(v) for each $v \in \{1, \ldots, n\}$ gives total cost $\Theta(n^2)$!

Trick: For each node v keep a number *rank*(v) **(rank)**, in an array `rank[1..`n`]: array of int`, with

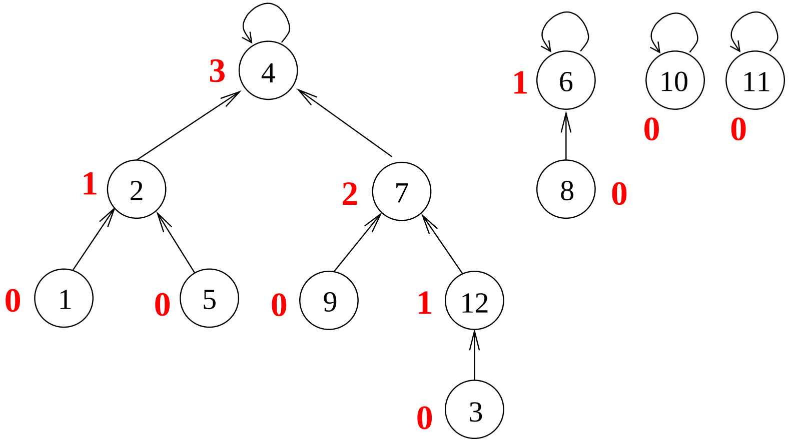
$$\texttt{rank}[v] = \text{depth of subtree with root } v.$$



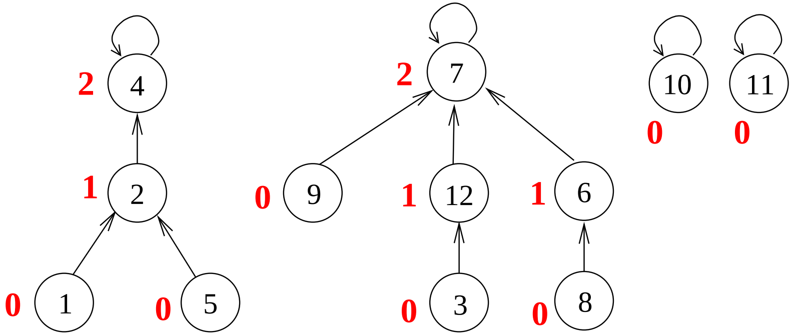
union(s,t): the root with the bigger rank becomes the root of the new, larger tree ("*union by rank*"). If the ranks are the same it does not matter which node we choose as new root – **(precisely) in this case** the rank of the node that remains a root increases by 1.

Examples:

union$(4,7)$ would give:



union$(6,7)$ would give:



Operations:

Prozedur init(n) // Initialization

(1) Generate `p`, `rank`: arrays of length n for `ints`

(2) **for** `v` **from** 1 **to** n **do**

(3) `p[v]` $\leftarrow$ `v`; `rank[v]` $\leftarrow 0$;

// n trees, consisting of only the root, which has rank 0

Cost: $\Theta(n)$.

prozedure union(s,t)

// Assumption: s,t are different representatives/roots

// I.e.: $\texttt{p}[s] = s$ and $\texttt{p}[t] = t$

(1) **if** $\texttt{rank}[s] > \texttt{rank}[t]$

(2) **then** $\texttt{p}[t] \leftarrow s$

(3) **elseif** $\texttt{rank}[t] > \texttt{rank}[s]$

(4) **then** $\texttt{p}[s] \leftarrow t$

(5) **else** $\texttt{p}[t] \leftarrow s$; $\texttt{rank}[s] \leftarrow \texttt{rank}[s] + 1$.

Cost: $O(1)$.

Theorem 5.1.3

The implementation of Union-Find just described has the following properties:

a) It is correct (i.e. it has the prescribed I/O behaviour).

b) **init**(n) takes time $\Theta(n)$;
find(v) takes time $O(\log n)$;
union(s, t) takes time $O(1)$.

Proof: (a) is clear.

(b) We observe:

Claim: If s is root of the tree B_s and $h = \textit{rank}(s)$, then B_s contains at least 2^h nodes.

Proof of claim: A root of rank h is created when two trees are joined whose roots both have rank $h-1$. From this the claim follows by an easy induction over $h = 0, 1, \ldots$.

Since the number of nodes is n, the largest rank h that can occur satisfies $2^h \leq n$, or $h \leq \log_2 n$.

Since rank(s) also gives the depth of the tree with root s, no node v can be more than $\log_2 n$ steps away from its root, so **find**(v) takes time $O(\log n)$.

$\square$

THE END

Many thanks for coming and taking part!

All the best in the exam!