



# Advanced algorithms

topological ordering,  
minimum spanning tree,  
Union-Find problem

Jiří Vyskočil, Radek Mařík

2012

# Subgraph

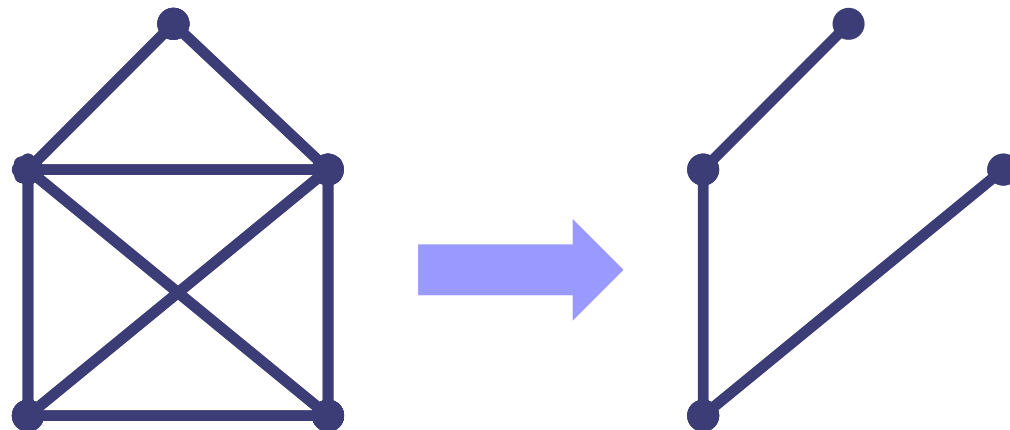
## ■ subgraph

- A graph  $H$  is a **subgraph** of a graph  $G$ , if the following two inclusions are satisfied:

$$V(H) \subseteq V(G)$$

$$E(H) \subseteq E(G) \cap \binom{V(H)}{2}$$

- In other words, a subgraph is created so that:
  - Some vertices of the original graph are removed.
  - All edges incident to the removed vertices and possibly some other edges are removed.



# DFS for the entire graph recursively

■ **input:** Graph  $G$ .

```
1) procedure DFS (Graph  $G$ ) {
2)   for each Vertex  $v$  in  $V(G)$  { state[ $v$ ] = UNVISITED;  $p[v]$  = null; }
3)   time = 0;
4)   for each Vertex  $v$  in  $V(G)$ 
5)     if (state[  $v$ ] == UNVISITED) then DFS-Walk( $v$ );
6)   }
```

```
7) procedure DFS-Walk(Vertex  $u$ ) {
8)   state[ $u$ ] = OPEN;  $d[u]$  = ++time;
9)   for each Vertex  $v$  in Neighbors( $u$ )
10)    if (state[ $v$ ] == UNVISITED) then { $p[v]$  =  $u$ ; DFS-Walk( $v$ ); }
11)   state[ $u$ ] = CLOSED;  $f[u]$  = ++time;
12) }
```

■ **output:** array  $p$  pointing to predecessor vertex, array  $d$  with times of vertex opening and array  $f$  with time of vertex closing.

# Topological ordering

- **topological ordering (topological sorting) of graph vertices**
  - Let graph  $G$  be DAG. Let's define binary relation  $R$  of **topological ordering** over vertices of graph  $G$  such as  $R(x,y)$  is valid iff there exists a directed path from  $x$  to  $y$ , that is, whenever  $y$  is reachable from  $x$ .
  - In other words: All vertices of graph  $G$  are assigned with numbers so that  $x \leq y$  holds for every pair of vertices  $x$  and  $y$  iff there is a directed path from  $x$  to  $y$ .  
Then relation  $\leq$  is a *topological ordering* over graph  $G$  with numbered vertices.
- **an implementation using the previous DFS algorithm**
  - The numbering vertices through array  $f$  with relation  $\leq$  is a topological order.



# Other uses of modified DFS

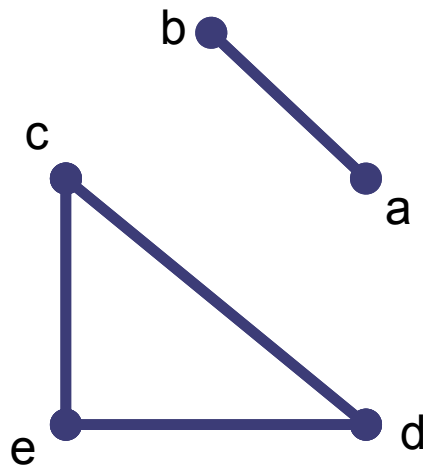
- Testing graph acyclicity
- Testing graph connectivity
- Searching for graph connected components
- Transformation of a graph to a directed forest.

# Connected component

- A connected component of graph  $G = (V, E)$  with regard to vertex  $v$  is a set

$$\mathcal{C}(v) = \{u \in V \mid \text{there exists a path in } G \text{ from } u \text{ to } v\}.$$

- In other words: If a graph is disconnected, then parts from which is composed from and that are themselves connected, are called *connected components*.

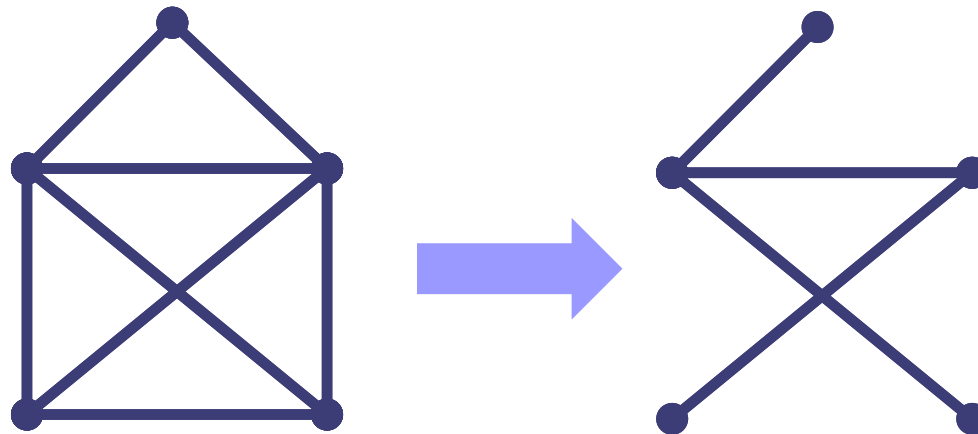


$$\mathcal{C}(a) = \mathcal{C}(b) = \{a, b\}$$

$$\mathcal{C}(c) = \mathcal{C}(d) = \mathcal{C}(e) = \{c, d, e\}$$

# Spanning tree

- graph spanning tree
  - Let  $G=(V,E)$  be a graph. A **Spanning tree of the graph  $G$**  is such a subgraph  $H$  of the graph  $G$  that  $V(G)=V(H)$  and  $H$  is a tree.



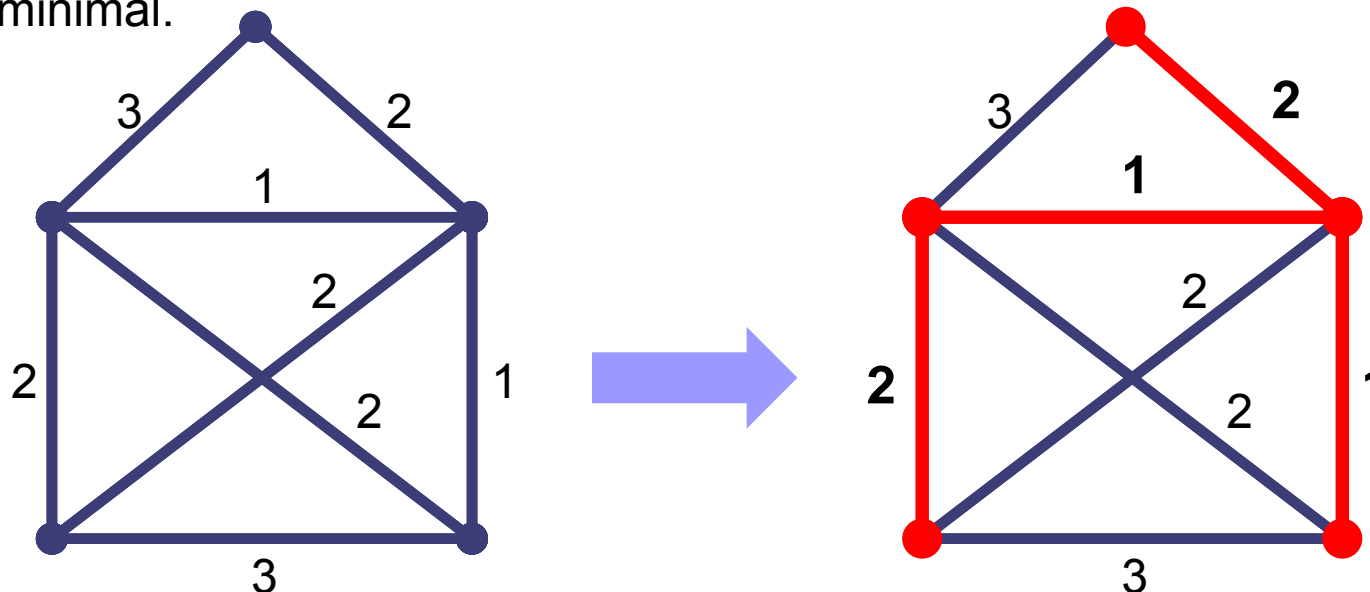
# Minimum spanning tree

- Minimum spanning tree

- Let  $G=(V,E)$  be a graph and  $w: E \rightarrow \mathbb{R}$  be its weight function.
- A **minimum spanning tree of the graph**  $G$  is such a tree  $K=(V,E_K)$  of the graph  $G$ , that

$$\sum_{e \in E_K} w(e) = w(K)$$

is minimal.





# Cut of graph

## ■ cut

□ A **cut of graph**  $G = (V, E)$  is a subset of edges  $F \subseteq E$  such that  
 $\exists U \subset V: F = \{\{u, v\} \in E \mid u \in U, v \notin U\}$ .

■ *Lemma:* Let  $G$  be a graph,  $w$  be its injective real-valued weight function,  $F$  be a cut of graph  $G$  and  $f$  be its lightest edge of cut  $F$  (crossing), then every minimum spanning tree  $K$  of graph  $G$  contains  $f \in E(K)$ .

□ *Proof by contradiction:* Let  $K$  be a minimum spanning tree and  $f = \{u, v\} \notin E(K)$ . Then there is a path  $P \subseteq K$  connecting  $u$  and  $v$ . The path has to cross the cut at least once. Therefore there is an edge  $e \in P \cap F$  and furthermore  $w(f) < w(e)$ . Let's consider  $K' = K - e + f$ . This graph is also a spanning tree of graph  $G$ , because the graph splits into two components by removing of the edge  $e$  and it merges back by adding of the edge  $f$ . Then  $w(K') = w(K) - w(e) + w(f) < w(K)$ .  $K'$  is also a minimum spanning tree.

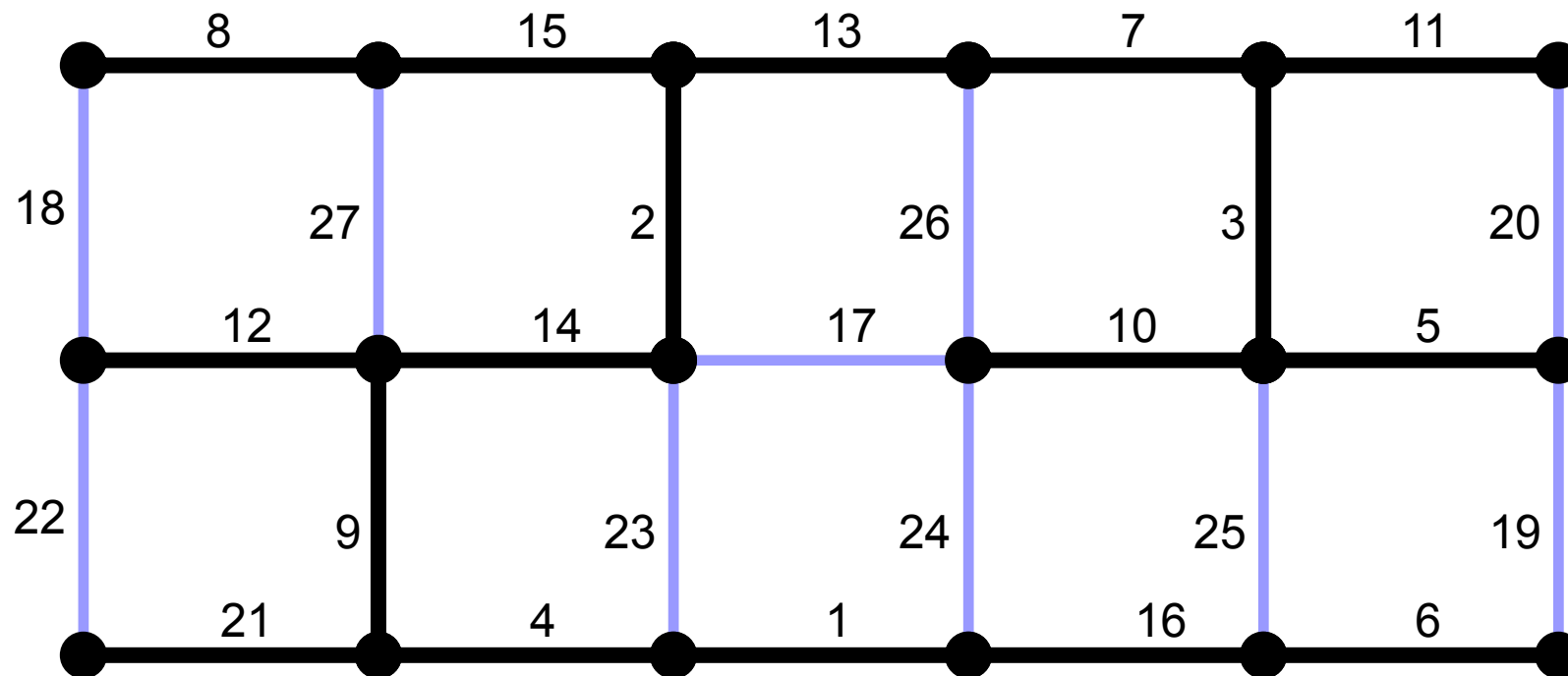
# Jarník (Prim)'s algorithm

- **input:** A graph  $G$  with a weight function  $w: E(G) \rightarrow \mathbb{R}$ . (Let  $|V(G)| = N$ .)
  - 1) Select an arbitrary vertex  $v_0 \in V(G)$ .
  - 2)  $K := (\{v_0\}, \emptyset)$ .
  - 3) **while**  $|V(K)| \neq |V(G)|$  { //  $|V(G)|$  loops
  - 4)     Select edge  $\{u, v\} \in E(G)$ ,  
      where  $u \in V(K)$  and  $v \notin V(K)$ , so that  
       $w(\{u, v\})$  is minimum.
  - 5)      $K := K + \text{edge } \{u, v\}$ .
  - 6) }
- **output:** a minimum spanning tree  $K$ .

Takes  $O(\log |V(G)|)$  time  
when a priority queue is  
used.

$O(\log |V(G)|)$  times  $|V(G)|$  loops yields  $O(|V(G)| \cdot \log |V(G)|)$  complexity.  
In short,  $O(N \cdot \log(N))$  complexity, when  $N = |V(G)|$ .

# Jarník (Prim)'s algorithm



# Jarník (Prim)'s algorithm

- Lemma: Jarník's algorithm stops after maximum  $|V(G)|$  steps and the result is a minimum spanning tree of the graph  $G$ .
  - In every iteration just one vertex is added to  $K$ , so the loop must stop after  $|V(G)|$  iteration in maximum.
  - The result graph  $K$  is a tree because only a leaf is always added to the tree. Furthermore,  $K$  has  $|V(G)|$  vertices – it is a spanning tree.
  - The edges among vertices of the tree  $K$  and the rest of the graph  $G$  determines a cut. The algorithm always adds the lightest edge of this cut to  $K$ . Following the previous lemma, all edges of  $K$  must belong to every minimum spanning tree. As  $K$  is a tree, then it must be a minimum spanning tree.

# Jarník (Prim)'s algorithm

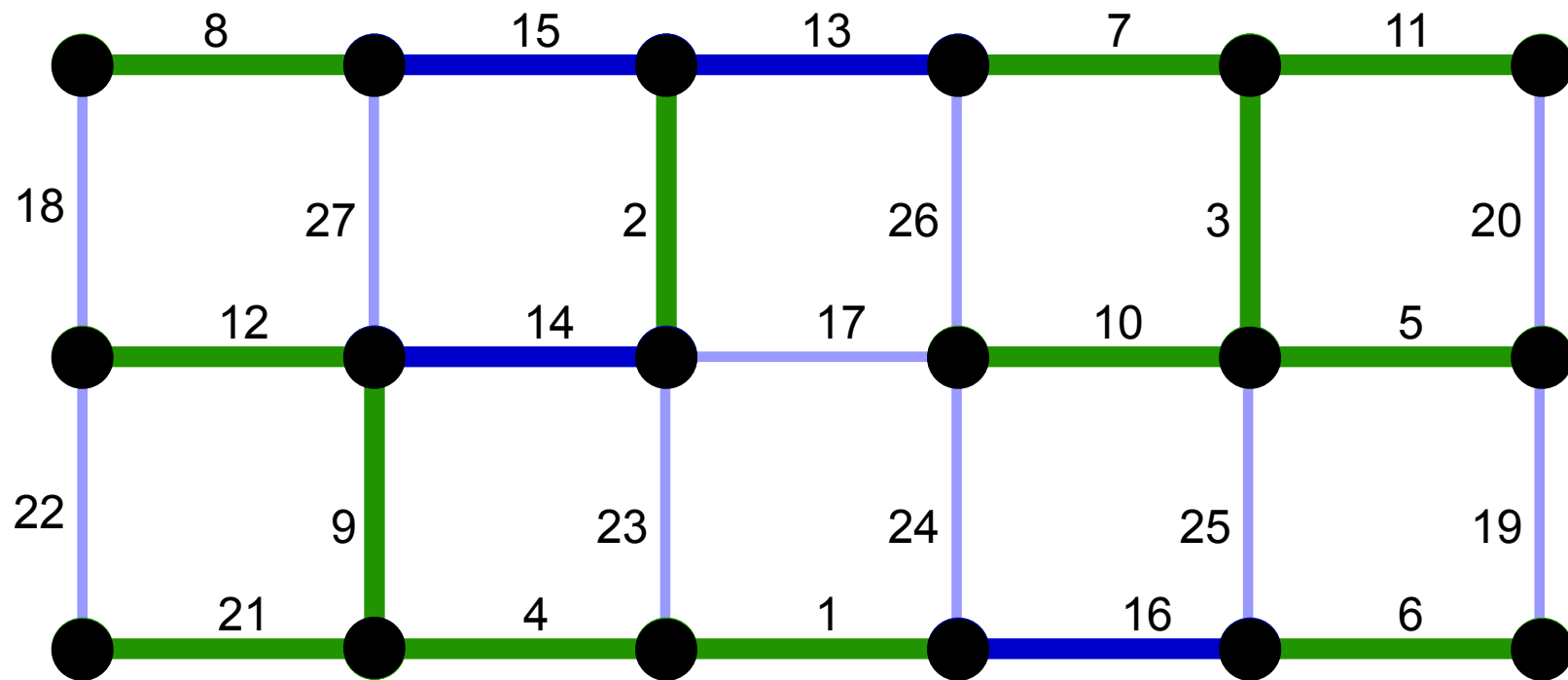
- implementations:
  - „straightforward”
    - Maintain which vertices and edges belong to the tree  $K$  and which not.
    - The time complexity is  $O(n \cdot m)$  where  $n = |V(G)|$  and  $m = |E(G)|$ .
  - improvements
    - Store  $D(v) = \min\{w(\{u,v\}) \mid u \in K\}$  for  $v \notin V(K)$ . During every iteration of the main loop we search through all  $D(v)$  (it takes  $O(n)$  time) and we check all neighbors  $D(s)$  for  $\{v,s\} \in E$  when a vertex  $v$  is added to  $K$  and its value is decreased if necessary ( $O(1)$  for each edge).
    - Time complexity is improved to  $O(n^2 + m) = O(n^2)$ .
    - The time complexity might be further improved using a suitable type of heap up to  $O(\log(n) \cdot m)$  (technically up to  $O(m + \log(n) \cdot n)$  with so called Fibonacci heap).

# Borůvka's algorithm

- **input:** A graph  $G$  with a weight function  $w: E(G) \rightarrow \mathbb{R}$ , where all weights are **different**.
  - 1)  $K := (V(G), \emptyset)$ .
  - 2) **while**  $K$  has at least two connected components {
  - 3) For all components  $T_i$  of graph  $K$  the *light incident edge*<sup>1</sup>  $t_i$  is chosen.
  - 4) All edges  $t_i$  are added to  $K$ .
  - 5) }
- **output:** a minimum spanning tree  $K$ .

<sup>1</sup> A *light incident edge* is an edge connecting a connected component  $T_i$  with another connected component while a weight of this edge is the lowest.

# Borůvka's algorithm



# Borůvka's algorithm

- theorem: Borůvka's algorithm stops after max.  $\lceil \log_2 |V(G)| \rceil$  and the result is a minimum spanning tree of the graph  $G$ .
  - After  $k$  iterations all components of the graph  $K$  have at least  $2^k$  vertices.
    - induction: Initially, all components consist of just one vertex.  
In each iteration, each component is merged with at least another neighboring one so that the size of components is at least doubled.
  - Therefore, after  $\lceil \log_2 |V(G)| \rceil$  iterations, the size of any component must be at least a number of all vertices of graph  $G$  and then the algorithm stops.
  - The edges between each connected component and the rest of graph determines a cut. Then all edges added to  $K$  must belong to a unique minimum spanning tree. Graph  $K \subseteq G$  is always a forest (= a set of trees disconnected to each other) and when the algorithm stops it will be equal to a minimum spanning tree.





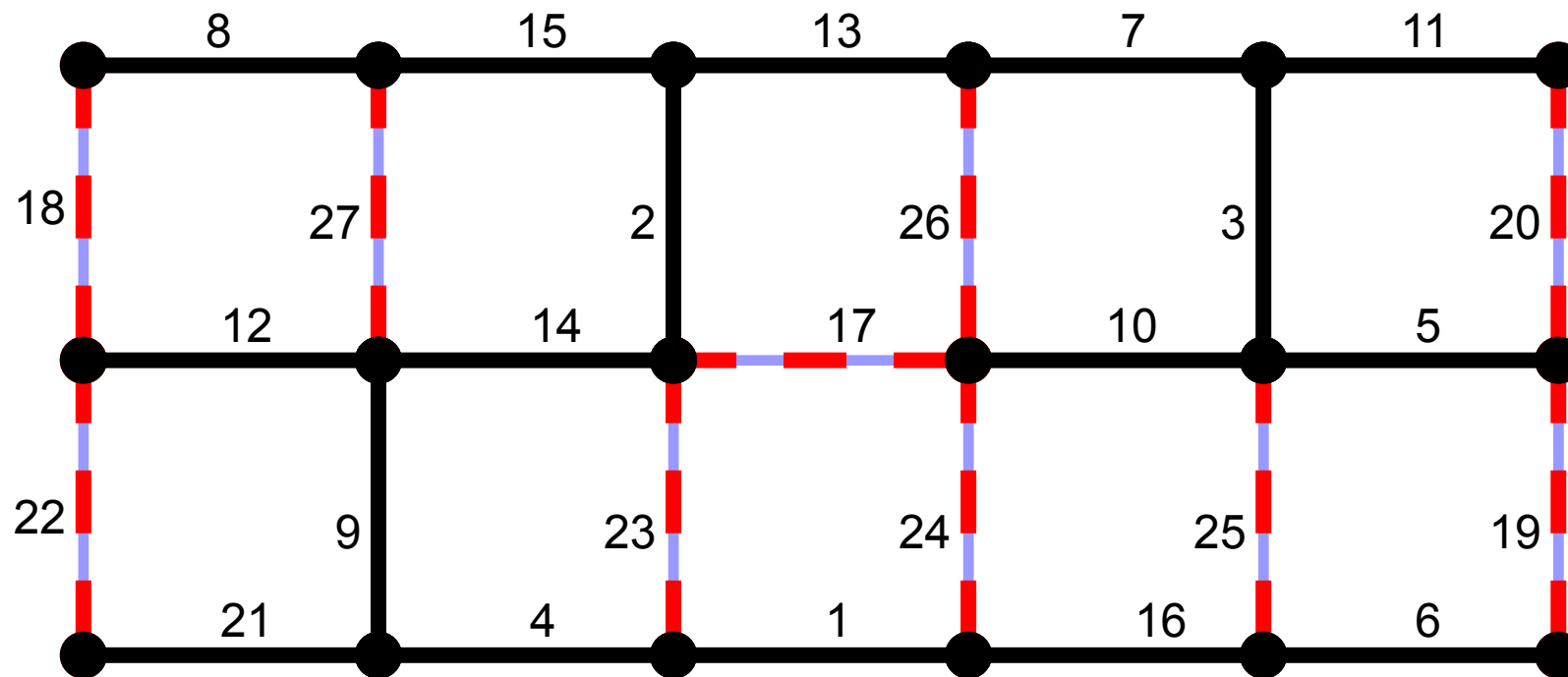
# Borůvka's algorithm

- Iteration implementation:
  - The forest is decomposed to connected components using DFS. Each vertex is assigned to a number of its component.
  - For each edge we find out to which component it belongs and we store the lightest edge only.
  - Therefore each iteration takes  $O(|E(G)|)$  time and the entire algorithm running time is  $O(|E(G)| \cdot \log|V(G)|)$ .

# Kruskal's („greedy“) algorithm

- **input:** A graph  $G$  with a weight function  $w: E(G) \rightarrow \mathbb{R}$ .
  - 1) Sort all edges  $e_1, \dots, e_{m=|E(G)|}$  from  $E(G)$  so that  $w(e_1) \leq \dots \leq w(e_m)$ .
  - 2)  $K := (V(G), \emptyset)$ .
  - 3) for  $i := 1$  to  $m$  {
  - 4)       if  $K + \text{edge } \{u, v\}$  is an acyclic graph then  
           $K := K + \text{edge } \{u, v\}$ .
  - 5) }
- **output:** a minimum spanning tree  $K$ .

# Kruskal's („greedy“) algorithm



# Kruskal's („greedy“) algorithm

- theorem: Kruskal's algorithm stops after  $|E(G)|$  iterations and returns a minimum spanning tree.
  - Each iteration of the algorithm processes just one edge, so the number of iterations is  $|E(G)|$ .
  - By induction we prove that  $K$  is always a subgraph of a minimum spanning tree: the empty initial  $K$  is a subgraph of anything (including a minimum spanning tree). Each added edge has the lowest weight in the cut separating a component of  $K$  from the rest of the graph (the remaining unprocessed edges of this cut are heavier). In opposite way, no edge that is not added to  $K$  cannot belong to a minimum spanning tree because it creates a cycle with edges already assigned to a minimum spanning tree.

# Kruskal's („greedy“) algorithm

## ■ implementation

- Sorting time is  $O(|E(G)| \cdot \log|E(G)|) = O(|E(G)| \cdot \log|V(G)|)$ .
- We can stop the main loop earlier. When we successfully add  $|V(G)| - 1$  edges to  $K$  then we can stop the algorithm because  $K$  has already reached a spanning tree.
- We need to maintain connected components of graph  $K$  so that we can recognize quickly if the current processed edge creates a cycle.
- Thus we need a structure for connected component maintenance which we can ask  $|E(G)|$ -times if two vertices belong to the same component (operation Find), and we merge just  $(|V(G)| - 1)$ -times two components to a single one (operation Union).

# Union-Find problem

- Let's have graph  $G = (V, E)$ .

Question: „Do vertices  $u$  and  $v$  belong to the same connected component of graph  $G$ ?“.

Sometimes the problem is called as incremental connected components or equivalence maintenance.

One representative is selected in each connected component. For sake of simplicity the representative of component  $\mathcal{C}(v)$  is labeled as  $r(v)$ .

If  $u$  and  $v$  belong to the same component then  $r(u) = r(v)$ .

The task might be accomplished using the following operations:

- $\text{FIND}(v) = r(v)$ , the operation returns the representative of connected component  $\mathcal{C}(v)$ .
- $\text{UNION}(u, v)$  merges connected components  $\mathcal{C}(u)$  and  $\mathcal{C}(v)$ . This reflects adding edge  $\{u, v\}$  into the graph.

# Union-Find problem

## ■ A simple solution:

- Let's assume all vertices are assigned with a number from 1 to  $n$ . Let's use an array  $R[1..n]$ , where  $R[i] = r(i)$ , i.e. the number of component  $C(i)$  representative.
- Operation  $\text{FIND}(v)$  just returns value  $R[v]$  and so it takes  $O(1)$ .
- To perform  $\text{UNION}(u, v)$  we find representatives  $r(u) = \text{FIND}(u)$  and  $r(v) = \text{FIND}(v)$ .  
If they are different then we process all items of array  $R$ . Any value of  $r(u)$  is rewritten to  $r(v)$ . It takes  $O(n)$  time.

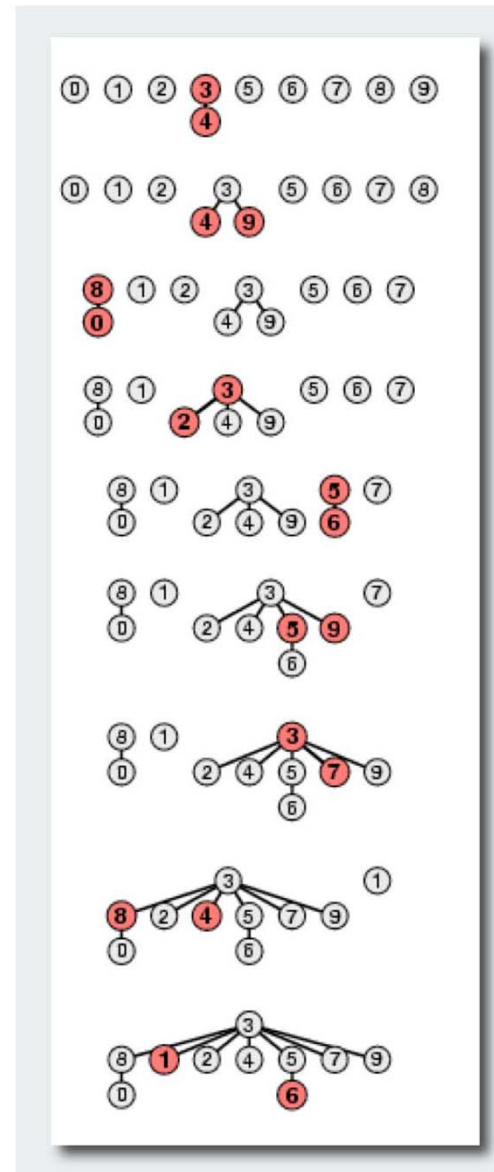
# Union-Find problem

- **An improved solution (using a directed tree):**
  - Each component is stored as a tree directed towards the root – every vertex has a pointer to its father, every root stores the size of the component. The root of each component serves as its representative.
  - Operation  $\text{FIND}(v)$  climbs from vertex  $v$  to the root that is returned.
  - To perform  $\text{UNION}(u, v)$  we find representatives  $r(u) = \text{FIND}(u)$  and  $r(v) = \text{FIND}(v)$ .  
If they differ then the root of smaller component is merged to the root of the bigger component. The size of new component is updated in its root.



# Union-Find problem

<b>3-4</b>	0	1	2	3	3	5	6	7	8	9
<b>4-9</b>	0	1	2	3	3	5	6	7	8	3
<b>8-0</b>	8	1	2	3	3	5	6	7	8	3
<b>2-3</b>	8	1	3	3	3	5	6	7	8	3
<b>5-6</b>	8	1	3	3	3	5	5	7	8	3
<b>5-9</b>	8	1	3	3	3	3	5	7	8	3
<b>7-3</b>	8	1	3	3	3	3	5	3	8	3
<b>4-8</b>	8	1	3	3	3	3	5	3	3	3
<b>6-1</b>	8	3	3	3	3	3	5	3	3	3



# Union-Find problem

- **An improved solution (using a directed tree):**
  - lemma: Union-Find tree of a depth  $h$  has at least  $2^h$  items.
  - By induction: If UNION merges a tree of the depth  $h$  with another tree of a depth smaller than  $h$ , then a depth of the result tree remains  $h$ . If two trees of the same depth  $h$  are merged, then the result tree has a depth  $h+1$ . By induction assumption we know that a tree of depth  $h$  has at least  $2^h$  vertices. Therefore the result tree of a depth  $h+1$  has at least  $2^{h+1}$  vertices.
  - A consequence: Time complexity of operation UNION and FIND is  $O(\log |V|)$ .
- The best known solution is amortized  $O(\alpha |V|)$  for both operations, where function  $\alpha$  is inverse Ackermann function.

# Kruskal's („greedy“) algorithm

- Kruskal's algorithm complexity:
  - Sorting takes time:  $O(|E(G)| \cdot \log|E(G)|) = O(|E(G)| \cdot \log|V(G)|)$ .
  - Then we need a structure for connected component maintenance which we can ask  $|E(G)|$ -times if two vertices belong to the same component (operation Find), and we merge just  $(|V(G)| - 1)$ -times two components to a single one (operation Union).
  - If the simple solution is used then the complexity of the algorithm is:  
 $O(|E(G)| \cdot \log|V(G)| + |E(G)| + |V(G)|^2) = O(|E(G)| \cdot \log|V(G)| + |V(G)|^2)$
  - If the improved solution using a directed tree is used then the complexity of the algorithm is:  
 $O(|E(G)| \cdot \log|V(G)| + |E(G)| \cdot \log|V(G)| + |V(G)| \cdot \log|V(G)|) = O(|E(G)| \cdot \log|V(G)|)$



# References

- Matoušek, J.; Nešetřil, J. *Kapitoly z diskretní matematiky*. Karolinum. Praha 2002. ISBN 978-80-246-1411-3.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). *Introduction to Algorithms (2nd ed.)*. MIT Press and McGraw-Hill. ISBN 0-262-53196-8.