Elementary Introduction to Graph Theory

(01EIG 2025/2026)

Lecture 4

Francesco Dolce
dolcefra@fit.cvut.cz

October 17, 2025

updated: October 17, 2025

PDF available at the address: dolcefra.pages.fit/ens/2526/EIG-lecture-04.pdf

Solution of Exercise in previous Lecture. Let $G = (V, E)$ and $H = (U, F)$ be two isomorphic graph with isomorphism φ. Each vertex $v \in V$ is sent to a vertex $\varphi(v) \in U$ having the same degree. Indeed, φ is, in particular, a bijection between $\mathrm{N}_G(v)$ and $\mathrm{N}_H(\varphi(v))$. Thus,

$$d_G(v) = \#\mathrm{N}_G(v) = \#\mathrm{N}_H(\varphi(v)) = d_H(\varphi(v)).$$

In the first graph all vertices have degree 2, while in the second one we have a vertex of degree 3, two vertices of degree 2 and one vertex of degree 1. Thus G and H cannot be isomorphic.

Solution of Exercise in previous Lecture. Let us use the radix order to compare coding of children of the same node.

a) The nodes of the rooted tree $T_1(\mathtt{d})$ have coding

$$C(\mathtt{a}) = C(\mathtt{e}) = \mathtt{01}, \qquad C(\mathtt{b}) = \mathtt{0.01.1}, \qquad C(\mathtt{c}) = \mathtt{0.0011.1}$$

and $C(\mathtt{d}) = \mathtt{0.01.000111.1}$, since $\mathtt{01} <_{rad} 000111$.

The nodes of the rooted tree $T_2(\mathtt{C})$ have coding

$$C(\mathtt{E}) = C(\mathtt{A}) = \mathtt{01}, \qquad C(\mathtt{D}) = C(\mathtt{B}) = \mathtt{0.01.1}$$

and $C(\mathtt{C}) = \mathtt{0.0011.0011.1}$, since $\mathtt{0011} =_{rad} \mathtt{0011}$.

Since $C(\mathtt{d}) \neq C(\mathtt{C})$ we can conclude that $T_1(\mathtt{d}) \not\sim_R T_2(\mathtt{c})$,

b) The nodes of the rooted trees $T_3(\mathtt{r}), T_4(\mathtt{R})$ and $T_5(\mathtt{s})$ have coding

$$C(\mathtt{a}) = C(\mathtt{d}) = C(\mathtt{c}) = C(\mathtt{A}) = C(\mathtt{C}) = C(\mathtt{D}) = C(\mathtt{z}) = C(\mathtt{x}) = C(\mathtt{y}) = \mathtt{01},$$

and

$$C(\mathtt{b}) = C(\mathtt{B}) = C(\mathtt{w}) = \mathtt{0.01.1}.$$

Since $\mathtt{01} =_{rad} \mathtt{01} <_{rad} \mathtt{0011}$, we have

$$C(\mathtt{r}) = C(\mathtt{R}) = C(\mathtt{s}) = \mathtt{0.01.01.0011.1},$$

which implies $T_3(\mathtt{r}) \sim_R T_4(\mathtt{R}) \sim_R T_5(\mathtt{s})$.

1 Directed graphs

A *directed graph* or *digraph* (or *oriented graph*) is a pair $D(V, E)$, where V is a (finite) set, and $a \in V \times V$ for every $a \in E$.

Elements of V are called *vertices*, while elements of E are called *arcs* (or *directed edges*, or *oriented edges*).

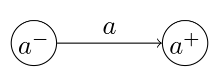
Given an arc $a = (u, v) \in E$, we call u the *tail* of a and v the *head* of a. Both u, v are the *ends* of a. The vertex u is called an *in-neighbour* of v, and the vertex v a *out-neighbour* of u. It is thus natural to define

$$\mathrm{N}^-_G(v) = \{u \mid (u, v) \in E\} \quad \text{and} \quad \mathrm{N}^+_G(v) = \{u \mid (v, u) \in E\}.$$

The *in-degree* and the *out-degree* of a vertex v are defined respectively as

$$d^-_G(v) = \#\mathrm{N}^-_G(v) \quad \text{and} \quad d^+_G(v) = \#\mathrm{N}^+_G(v).$$

Given an arc $a = (u, v)$, we also denote by $a^- = u$ its starting vertex and by $a^+ = v$ its ending vertex.



A graph $G = (V', E')$ is the *underlying graph* of a digraph D if $V' = V$ and $\{u, v\} \in E'$ if $(u, v) \in E$.

Example 1 The digraph $D = (V, E)$, with vertices $V = \{\mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}\}$ and arcs $E = \{(\mathtt{u}, \mathtt{v}), (\mathtt{u}, \mathtt{w}), (\mathtt{v}, \mathtt{x}), (\mathtt{x}, \mathtt{v}), (\mathtt{y}, \mathtt{v}), (\mathtt{y}, \mathtt{y})\}$ is shown in Figure 1 on the left. One can check that $d^-_G(\mathtt{v}) = 3$ and $d^+_G(\mathtt{v}) = 1$.

The underlying graph G of D is shown on the same figure on the right.

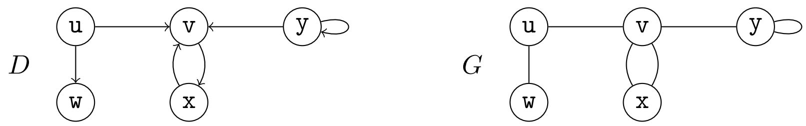


Figure 1: A digraph D (left) and its underlying graph G (right).

An *orientation* of a graph $G = (V, E)$ is a digraph obtained by choosing a direction for every edge in E. The *converse* of a digraph $D = (V, E)$ is the digraph $D' = (V', E')$ with set of vertices $V = V'$ and seet of arcs given by $(u, v) \in E \Leftrightarrow (v, u) \in E'$.

Clearly, both a digraph and its converse are orientations of the same graph.

Example 2 An orientation of the graph $G = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\}\})$ is the digraph $D = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{(\mathtt{a}, \mathtt{b}), (\mathtt{c}, \mathtt{b})\})$. The converse of D is the digraph $D' = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{(\mathtt{b}, \mathtt{a}), (\mathtt{b}, \mathtt{c})\})$ (see Figure 2).

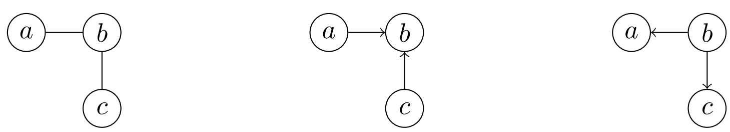


Figure 2: A graph (left) and two of its possible orientations (centre and right).

2 Branching

An orientation of a rooted tree such that every vertex but the root has in-degree 1 is called a *branching*.

It is clear that (u, v) is an arc in a branching if and only if v is a child of u in the correspective tree.

Example 3 Let $T(\mathtt{r})$ be the rooted tree shown on the left of Figure 3. A branching D of $T(\mathtt{r})$ is shown on the right of the same Figure 3.

Figure 3: A rooted tree (left) and its branching (right).

3 Graph Traversal Algorithms (GTS)

A *graph traversal* is the process of visiting each vertex of a (connected) graph. It is a key ingredient of many graph algorithms.

Tarry's algorithm. The oldest known GTS algorithm, Tarry's algorithm, uses two rules while traversing the graph:

(R1) every edge can be used at most once in every direction;

(R2) the edge of the first arrival to a vertex can be used (in the opposite direction) only if there is no other possibility.

Theorem 1 *Tarry's traversal is finite.*

Moreover, if there is no way to continue with the traversal, we have returned to the starting vertex, and, moreover, every edge of G has been used exactly twice.

Example 4 Let G be the graph in Figure 4. If the starting point is the vertex `a`, a possible traversal is the walk (`a,d,b,c,f,e,d,g,h,i`).

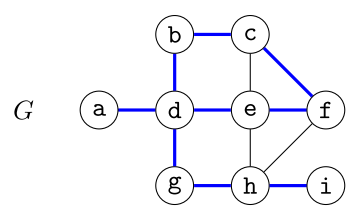


Figure 4: The walk (`a,d,b,c,f,e,d,g,h,i`) is a traversal.

Note that in a traversal we do not necessarily use all edges, and we may visit the same vertex more than once.

Today, the two primary graph traversal (or searching) algorithms are:

- *Breadth-First Search* (using queues).
- *Depth-First Search* (using stacks),

The following algorithm can be used with D either a queue, for BFS, or a stack, for DFS.

Algorithm 1: `Traversal`(G)

Input: graph G
Init: choose a vertex and put it in D
repeat
  pop a vertex from D
  process it
  put all its unprocessed neighbours in D
until D *is empty*

Example 5 A possible traversal of the graph G in Figure 4 starting from `a` is given by the queue (using BFS):

$$\emptyset \to (\mathtt{a}) \to (\underline{\mathtt{a}}, \mathtt{d}) \to (\underline{\mathtt{d}}, \mathtt{b}, \mathtt{e}, \mathtt{g}) \to (\underline{\mathtt{b}}, \mathtt{e}, \mathtt{g}, \mathtt{c}) \to (\underline{\mathtt{e}}, \mathtt{g}, \mathtt{c}, \mathtt{f}, \mathtt{h})$$
$$\to (\underline{\mathtt{g}}, \mathtt{c}, \mathtt{f}, \mathtt{h}) \to (\underline{\mathtt{c}}, \mathtt{f}, \mathtt{h}) \to (\underline{\mathtt{f}}, \mathtt{h}) \to (\underline{\mathtt{h}}, \mathtt{i}) \to (\underline{\mathtt{i}}) \to \emptyset.$$

Example 6 A possible traversal of the graph G in Figure 4 starting from `a` is given by the stack (using DFS):

$$\emptyset \to [\mathtt{a}] \to [\mathtt{a}, \mathtt{d}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{c}]$$
$$\to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{c}, \mathtt{b}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{c}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{h}]$$
$$\to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{h}, \mathtt{g}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{h}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{h}, \mathtt{i}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{h}]$$
$$\to [\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}] \to [\mathtt{a}, \mathtt{d}, \mathtt{e}] \to [\mathtt{a}, \mathtt{d}] \to [\mathtt{a}] \to \emptyset.$$

We can use a GTS algorithm, e.g., to find the components of a (not necessarily connected) graph G:

1. use DFS from a starting point (i.e., any vertex) $v \in V$;
2. if the algorithm stops, then all vertices are processed: we have the component of G containing v.

4 Tree-Search Algorithms: BFS vs DFS

Let $G = (V, E)$ be a graph. A *spanning tree* of G is a spanning graph of G that is a tree, i.e., a tree $T = (V, E')$ with $E' \subseteq E$.

A graph that is not connected cannot contain any spanning tree.

On the other hand, any connected graph contains at least one spanning tree.

Example 7 Let G be the graph in Figure 5 on the left. Two possible spanning trees are represented on the right of the same figure.

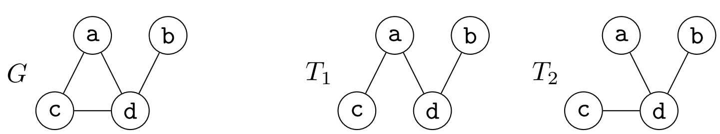


Figure 5: A graph and two of its possible spanning trees.

One can modify Algorithm 1 to construct the spanning tree of a graph. For instance, the following algorithm uses BFS, and a queue, to construct a spanning tree of a given graph.

Algorithm 2: `Breadth-FirstSearch`(G)

Input: graph G
Output: a spanning tree $T = (V, E)$ of G
Init: choose a vertex v and append it to Q

$V = \{v\}$, $E = \emptyset$
repeat
    let x be the head of Q
    foreach $y \in \mathrm{N}_G(x)$ **do**
        if $y \notin V$ **then**
            append y to Q
            $V \leftarrow V \cup \{y\}$
            $E \leftarrow E \cup \{x, y\}$ // y is a child of x
    remove x from Q
until Q *is empty*

Example 8 Let G be the graph in Figure 4. A spanning tree is given by the following queue (see also Figure 6):

$$\emptyset \to (\mathtt{a}) \stackrel{\{\mathtt{a},\mathtt{d}\}}{\to} (\mathtt{a},\mathtt{d}) \to (\mathtt{d}) \stackrel{\{\mathtt{d},\mathtt{b}\}}{\to} (\mathtt{d},\mathtt{b}) \stackrel{\{\mathtt{d},\mathtt{e}\}}{\to} (\mathtt{d},\mathtt{b},\mathtt{e}) \stackrel{\{\mathtt{d},\mathtt{g}\}}{\to} (\mathtt{d},\mathtt{b},\mathtt{e},\mathtt{g})$$
$$\to (\mathtt{b},\mathtt{e},\mathtt{g}) \stackrel{\{\mathtt{b},\mathtt{c}\}}{\to} (\mathtt{b},\mathtt{e},\mathtt{g},\mathtt{c}) \to (\mathtt{e},\mathtt{g},\mathtt{c}) \stackrel{\{\mathtt{e},\mathtt{f}\}}{\to} (\mathtt{e},\mathtt{g},\mathtt{c},\mathtt{f})$$
$$\stackrel{\{\mathtt{e},\mathtt{h}\}}{\to} (\mathtt{e},\mathtt{g},\mathtt{c},\mathtt{f},\mathtt{h}) \to (\mathtt{g},\mathtt{c},\mathtt{f},\mathtt{h}) \to (\mathtt{c},\mathtt{f},\mathtt{h}) \to (\mathtt{f},\mathtt{h}) \to (\mathtt{h})$$
$$\stackrel{\{\mathtt{h},\mathtt{i}\}}{\to} (\mathtt{h},\mathtt{i}) \to (\mathtt{i}) \to \emptyset.$$

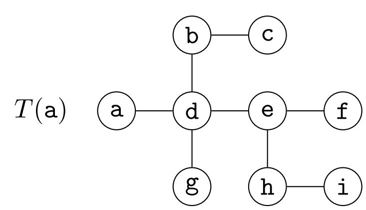


Figure 6: A spanning tree obtained by BFS.

Remember that, for a rooted tree $T(r)$ the level of a vertex v in T is defined as $\ell_T(v) = d_T(r, v)$.

Let $T(r)$ be a spanning tree of a connected graph G obtained by BFS . Then it is easy to show that for every vertex v of G we have $\ell_T(v) = d_G(r, v)$, i.e., the distance between r and v in the original graph is the same as the distance in the spanning tree.

Example 9 Let G be the graph in Example 4 and $T(\mathtt{a})$ be the rooted spanning tree obtained from it in Example 8. One can check that we have, e.g., $d_G(\mathtt{a}, \mathtt{h}) = d_T(\mathtt{a}, \mathtt{h}) = 3$.

Similarly we can construct a spanning tree of a graph using DFS, with a stack as data structure.

Algorithm 3: `Depth-FirstSearch(`G`)`

Input: graph G
Output: a spanning tree $T = (V, E)$ of G
Init: choose a vertex v and push it to the top of S

$V = \{v\}$, $E = \emptyset$
repeat
  let x be the top of S
  if $\mathrm{N}_G(x) \setminus V$ *is not empty* **then**
    choose $y \in \mathrm{N}_G(x) \setminus V$
    push y to the top of S
    $V \leftarrow V \cup \{y\}$
    $E \leftarrow E \cup \{x, y\}$ // y is a child of x
  else
    remove x from S
until S *is empty*

Example 10 Let G be the graph in Figure 4. A spanning tree is given by the following stack (see also Figure 7):

$$\begin{aligned}&\emptyset \to [\mathtt{a}] \stackrel{\{\mathtt{a},\mathtt{d}\}}{\to} [\mathtt{a},\mathtt{d}] \stackrel{\{\mathtt{d},\mathtt{e}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e}] \stackrel{\{\mathtt{e},\mathtt{f}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \stackrel{\{\mathtt{f},\mathtt{c}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c}]\\ &\stackrel{\{\mathtt{c},\mathtt{b}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c},\mathtt{b}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \stackrel{\{\mathtt{f},\mathtt{h}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}]\\ &\stackrel{\{\mathtt{h},\mathtt{g}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h},\mathtt{g}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}] \stackrel{\{\mathtt{h},\mathtt{i}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h},\mathtt{i}]\\ &\to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \to [\mathtt{a},\mathtt{d},\mathtt{e}] \to [\mathtt{a},\mathtt{d}] \to [\mathtt{a}] \to \emptyset.\end{aligned}$$

$T(\mathtt{a})$

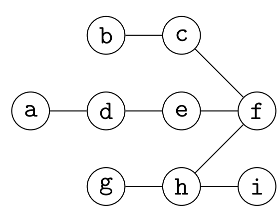


Figure 7: A spanning tree obtained by DFS.

5 Tree-Search Algorithms: ordered edges

A different algorithm to obtain a spanning tree T from a graph $G = (V, E)$ is given by adding a vertex at a time without creating any cycles.

1. Let us assume that the edges of E are ordered: $e_1 < e_2 < \ldots < e_m$.

2. Construct a sequence of subgraphs

$$G_0 \subseteq G_1 \subseteq \cdots \subseteq G_k$$

such that

- $G_0 = (V, \emptyset)$, i.e., we start with no edges;
- for every i we define $V(G_i) = V$, i.e., all graphs have the same set of vertices, and

$$E(G_{i+1}) = \begin{cases} E(G_i) \cup \{e_i\} & \text{if by adding } e_i \text{ no cycle is created in } E(G_i), \\ E(G_i) & \text{otherwise.} \end{cases}$$

3. We stop when $\#E_i = \#V - 1$.

In the last step, we rely on Euler's formula for trees.
The algorithm can be written in pseudo-code as follows.

Algorithm 4: `TreeSearchEdges(`G`)`

Input: graph $G = (V, E)$ with $E = \{e_1 < e_2 < \ldots < e_m\}$
Output: spanning tree $T = (V, E')$ of G
$E' = \emptyset$, $i = 1$
repeat
  if *($T + e_i$ is acyclic)* **then**
    $E' \leftarrow E' \cup \{e_i\}$
  $i \rightarrow i + 1$
until $\#E' = \#V - 1$

Example 11 Let us consider the graph of Example 4 with edges ordered as on the left of Figure 8: $e_1 < e_2 < \ldots < e_{12}$. The spanning tree obtained using the algorithm above is shown on the right of the same figure. For instance, we do not add e_6 since this would add a cycle (b, c, e, d).

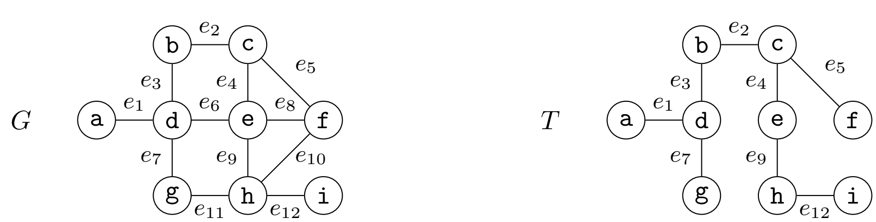


Figure 8: A graph and one of its spanning tree.

6 Weighted graphs

In some applications it is important to consider edges differently and to associate to each of them a different number.

A *weighted graph* (G, w) is a graph $G = (V, E)$ together with a function $w : E \to \mathbb{R}$, called *weight function*, associating to each edge e its *weight* $w(e)$. Given a subgraph $H = (U, F)$ of G, we call $w(H) = \sum_{e \in F} w(e)$ the *weight* of H.

Example 12 Let G be the simple graph in Figure 9 where the weight of each edge is represented above it. Let P be the path $(\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g})$ in G (edges are in blue in Figure 9). Then $w(G) = 22$ and $w(P) = 14$.

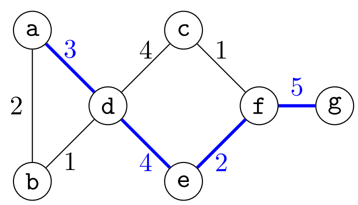


Figure 9: A weighted graph.

7 Minimal spanning trees

Let (G, w) be a weighted graph, with $G = (V, E)$. A *minimum spanning tree* (or *optimal tree*) of G is a spanning tree $T = (V, E')$ of G such that the cost $w(T) = \sum_{e \in E'} w(e)$ of T is minimal.

Several known algorithms exist for finding a minimum spanning tree. The most used is the so-called Kruskal's algorithm

Algorithm 5: `Kruskal`$((G, w))$

Input: weighted graph (G, w) with $G = (V, E)$
Output: minimal spanning tree $T = (V, E')$ of G
sort E s.t. $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
return `TreeSearchEdges`(G, E)

Example 13 ČEZ wants to connect some major cities (**P**rague, **B**rno, **L**iberec, **Ú**stí nad Labem, **O**lomouc, **H**radec Králove, **Z**lín, **Č**eské Budějovice) to the electricity network. The network has to be connected, otherwise some towns will not get the electricity. On the other hand, ČEZ wants to keep the building costs as low as possible. Thus, they want to build a connected graph with the lowest possible total cost of links. Assuming that the cost of building a link between two cities is proportional to the distance between these cities (see Table 1), find an optimal subgraph.

km	**P**	**B**	**L**	**Ú**	**O**	**H**	**Z**	**Č**
P	–	185	88	69	209	101	252	124
B		–	207	246	65	126	77	157
L			–	73	204	83	254	204
Ú				–	258	137	305	191
O					–	122	51	213
H						–	171	169
Z							–	234
Č								–

Table 1: Distance between cities in Czechia.

Using the distance between the cities we can construct a weighted complete graph K_8 on 8 vertices $\{\mathbf{P}, \mathbf{B}, \mathbf{L}, \mathbf{Ú}, \mathbf{O}, \mathbf{H}, \mathbf{Z}, \mathbf{Č}\}$. This complete graph has $\frac{8 \cdot 7}{2} = 24$ edges. Let's order them according to their weight:

$$
\begin{array}{llll}
e_1 = \{\mathbf{O}, \mathbf{Z}\}, & e_2 = \{\mathbf{B}, \mathbf{O}\}, & e_3 = \{\mathbf{P}, \mathbf{Ú}\}, & e_4 = \{\mathbf{L}, \mathbf{Ú}\}, \\
e_5 = \{\mathbf{B}, \mathbf{Z}\}, & e_6 = \{\mathbf{L}, \mathbf{H}\}, & e_7 = \{\mathbf{P}, \mathbf{L}\}, & e_8 = \{\mathbf{P}, \mathbf{H}\}, \\
e_9 = \{\mathbf{P}, \mathbf{Č}\}, & e_{10} = \{\mathbf{O}, \mathbf{H}\}, & e_{11} = \{\mathbf{B}, \mathbf{H}\}, & e_{12} = \{\mathbf{Ú}, \mathbf{H}\}, \\
e_{13} = \{\mathbf{B}, \mathbf{Č}\}, & e_{14} = \{\mathbf{H}, \mathbf{Č}\}, & e_{15} = \{\mathbf{H}, \mathbf{Z}\}, & e_{16} = \{\mathbf{P}, \mathbf{B}\}, \\
e_{17} = \{\mathbf{Ú}, \mathbf{Č}\}, & e_{18} = \{\mathbf{L}, \mathbf{O}\}, & e_{19} = \{\mathbf{L}, \mathbf{Č}\}, & e_{20} = \{\mathbf{P}, \mathbf{O}\}, \\
e_{21} = \{\mathbf{B}, \mathbf{L}\}, & e_{22} = \{\mathbf{O}, \mathbf{Č}\}, & e_{23} = \{\mathbf{Z}, \mathbf{Č}\}, & e_{24} = \{\mathbf{B}, \mathbf{Ú}\}, \\
e_{25} = \{\mathbf{P}, \mathbf{Z}\}, & e_{26} = \{\mathbf{L}, \mathbf{Z}\}, & e_{27} = \{\mathbf{Ú}, \mathbf{O}\}, & e_{28} = \{\mathbf{Ú}, \mathbf{Z}\}.
\end{array}
$$

The minimal spanning tree of the complete graph using Kruskal's algorithm is shown in Figure 10

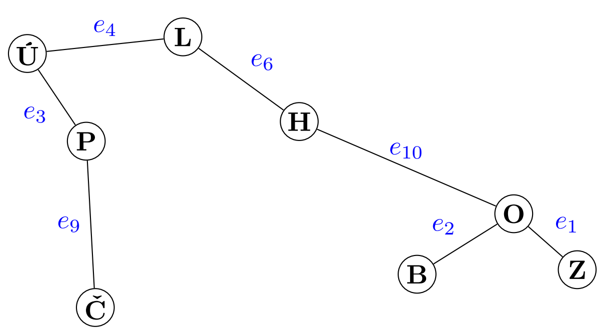


Figure 10: A minimal spanning tree of K_8.

Exercise. Let (G, w) be the weighted graph represented in Figure 11, where each edge is represented with its weight. Find a minimal spanning tree of G.

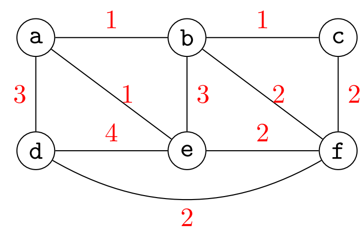


Figure 11: A weighted graph.