

Elementary Introduction to Graph Theory

(01YEIG 2026/2027)

Lecture 6



Francesco Dolce
dolcefra@fit.cvut.cz

October 30, 2026

updated: October 23, 2026

PDF available at the address:
dolcefra.pages.fit/ens/2627/YEIG-lecture-06.pdf

Solutions of Exercises in the previous Lecture

Solution of Exercise [Isomorphism of rooted trees]. Let us use the radix order to compare encoding of the children of a given node.

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

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

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

The nodes of the rooted tree $T_2(\mathbf{C})$ have encoding

$$C(\mathbf{E}) = C(\mathbf{A}) = \mathbf{01}, \quad C(\mathbf{D}) = C(\mathbf{B}) = \mathbf{0.01.1}$$

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

Since $C(\mathbf{d}) \neq C(\mathbf{C})$ we can conclude that $T_1(\mathbf{d}) \not\cong_R T_2(\mathbf{C})$,

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

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

and

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

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

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

which implies $T_3(\mathbf{r}) \cong_R T_4(\mathbf{R}) \cong_R T_5(\mathbf{s})$.

1 Construction of a spanning tree using ordered edges

A different algorithm for obtaining a spanning tree T from a graph $G = (V, E)$ is given by adding an edge at a time while avoiding the creation of cycles.

1. Let us assume that the edges of E are ordered: $e_1 < e_2 < \dots < e_m$.
2. Construct a sequence of subgraphs

$$G_0 \subseteq G_1 \subseteq \dots \subseteq G_m$$

such that

- $G_0 = (V, \emptyset)$, i.e., we start with no edges;
- for every $i = 1, \dots, m$ we define $G_i = (V, E_i)$ (i.e., all graphs have the same set of vertices), where

$$E_i = \begin{cases} E_{i-1} \cup \{e_i\} & \text{if this does not create a cycle,} \\ E_{i-1} & \text{otherwise.} \end{cases}$$

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 1: TreeSearchEdges(G)

Input: a connected graph $G = (V, E)$ with $E = \{e_1 < e_2 < \dots < e_m\}$

Output: a spanning tree $T = (V, E')$ of G

```

1  $E' = \emptyset, i = 1$ 
2  $T = (V, E')$ 
3 repeat
4   if  $(T + e_i \text{ is acyclic})$  then
5      $E' \leftarrow E' \cup \{e_i\}$ 
6      $i \leftarrow i + 1$ 
7 until  $\#E' = \#V - 1$ 
```

Example 1 Let us consider the graph on the left of Figure 1 with the order of edges given by $e_1 < e_2 < \dots < e_{12}$. The spanning tree obtained using the above algorithm is shown on the right of the same figure. For example, we do not add e_6 since doing so would create the cycle (b, c, e, d) .

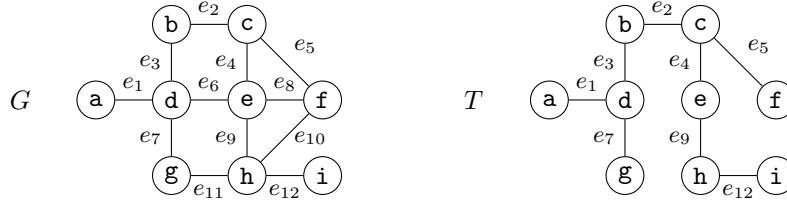


Figure 1: A graph and one of its spanning trees.

2 Weighted graphs

In some applications, edges represent a cost, distance, length, capacity, or other numerical quantity. This motivates the following definition.

A *weighted graph* (G, w) is a graph $G = (V, E)$ together with a function $w : E \rightarrow \mathbb{R}$, called *weight function*, assigning to each edge e its *weight* $w(e)$. The *weight* of a subgraph $H = (U, F)$ of G is defined by $w(H) = \sum_{e \in F} w(e)$.

Example 2 Let G be the simple graph in Figure 2 where the weight of each edge is represented above it. Let P be the path (a, d, e, f, g) in G (edges are in blue in Figure 2). Then $w(G) = 22$ and $w(P) = 14$.

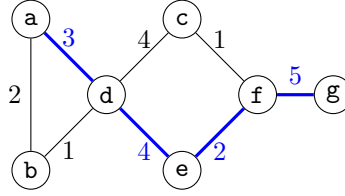


Figure 2: A weighted graph.

3 Minimum spanning trees

Let (G, w) be a weighted graph, with $G = (V, E)$. A *minimum spanning 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. A classical one is the so-called Kruskal's algorithm

Algorithm 2: $\text{Kruskal}((G, w))$

Input: weighted graph (G, w) with $G = (V, E)$ **Output:** minimum spanning tree $T = (V, E')$ of G 1 sort E s.t. $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$ 2 return $\text{TreeSearchEdges}(G)$

Example 3 ČEZ wants to connect some major cities (**P**rague, **B**rno, **L**iberec, **Ú**stí nad Labem, **O**lomouc, **H**radec Králové, **Z**lín, **Č**eské Budějovice) to the electricity network.

The network has to be connected, otherwise some cities would not be connected to the electricity network. 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 $\binom{8}{2} = 28$ edges. Let's order them according to their weight:

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

The minimum spanning tree of the complete graph using Kruskal's algorithm is shown in Figure 3

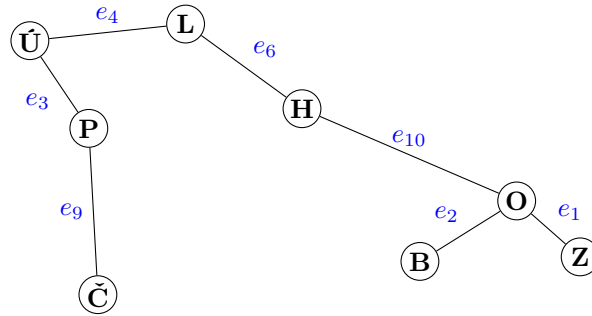


Figure 3: A minimum spanning tree of K_8 .

Therefore,

$$\omega(T) = 51 + 65 + 69 + 73 + 83 + 122 + 124 = 587.$$

Exercise [Minimum spanning tree]. Let (G, w) be the weighted graph represented in Figure 4, where each edge is represented with its **weight**. Find a minimum spanning tree of G .

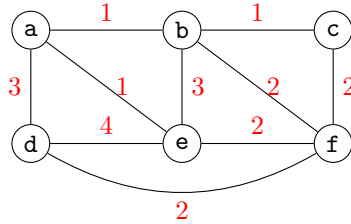


Figure 4: A weighted graph.

4 Time complexity and the Union-Find scheme

The time complexity of the algorithms seen above depends on the used data structures. For instance, in Kruskal's Algorithm, time complexity heavily depends on the data structure used to test whether adding an edge creates a cycle.

To test that, it is enough to remember for each vertex v the component to which the vertex belongs: a new edge $\{x, y\}$ creates a cycle if and only if the vertices x and y already belong to the same component.

A naive implementation could be to store, for each vertex, the label of current connected component.

The operation $\text{FIND}(v)$ that finds the label of the component containing the vertex v has a time complexity $\mathcal{O}(1)$. This seems to be perfect.

However, we need to implement another operation: the $\text{UNION}(\cdot)$ operation. Suppose that we are trying to add an edge $e = \{x, y\}$. If x and y are in the same component, we skip e . If x and y are in a different component – i.e. if $\text{FIND}(x) \neq \text{FIND}(y)$ – then we add e to the graph. This operation merges the components containing x and y into a single component.

In our naive implementation, we have to go through the whole array used to store the labels of components and to perform the merging.

For example, if x is in component A and y is in component B , we must scan the entire array, and replace all instances of B with A .

This operation has time complexity $\mathcal{O}(n)$, where $n = \#V$.

The total time complexity of Kruskal's Algorithm with naive implementation is, thus, $\mathcal{O}(m \cdot n)$, where $m = \#E$ and $n = \#V$, since we have at most m steps of the algorithm.

We can consider a more elaborate implementation of UNION-FIND scheme.

- We represent individual components by trees.
- The root of a tree is the label of the component, or a representative of the component.
- $\text{FIND}(a)$: find out the root of the tree representing the component containing the vertex a . The complexity is proportional to the height of the tree, i.e., $\mathcal{O}(\ln n)$.
- $\text{UNION}(a, b)$: we need to merge two components. We merge the trees representing both components so that the height of the resulting tree is as small as possible. To achieve this goal we attach the root of the tree with fewer vertices to the root of the tree with more vertices.

In this way the height increases at most by 1. The complexity is $\mathcal{O}(1)$ – if we remember the length of trees and we ran $\text{FIND}(\cdot)$ before $\text{UNION}(\cdot)$ – or $\mathcal{O}(\ln n)$.

- The total time complexity of the scheme is $\mathcal{O}(m \cdot \ln n)$, where $m = \#E$ and $n = \#V$.

Therefore, the total running time of the Kruskal's Algorithm is $\mathcal{O}(m \cdot n) + \mathcal{O}(m \ln n) = \mathcal{O}(m \ln n)$.

Example 4 Let G be a graph having two components A and X as in the left of Figure 5. We can check that $\text{FIND}(\mathbf{b}) = A$ and $\text{FIND}(\mathbf{z}) = X$. Two possible representative trees for the components are shown in the centre of the same figure (we select $\mathbf{a}$ as representative of A and $\mathbf{x}$ as representative of X). The union of the two components containing $\mathbf{b}$ and $\mathbf{z}$ is shown on the right of the figure.

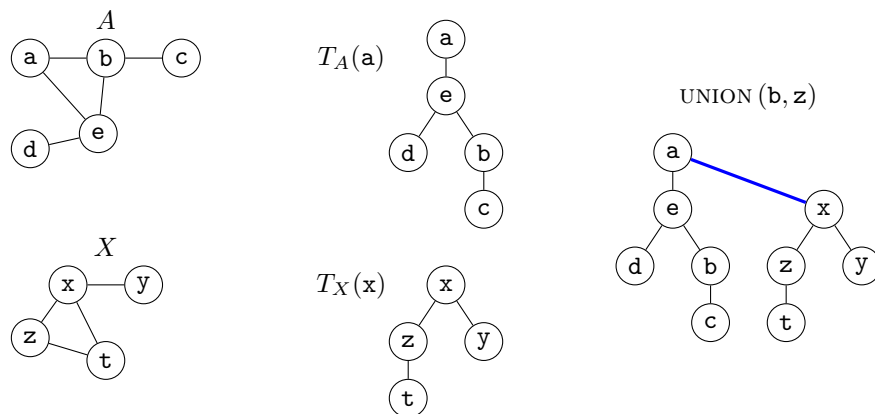


Figure 5: UNION-FIND scheme using trees.

We discussed the UNION-FIND scheme to demonstrate how algorithmic efficiency can depend on the choice of data structure. This scheme, also called *disjoint-set data structure* is a very general data structure that has different applications.

In practice, a faster version of the UNION-FIND scheme is usually used: tree representation with path compression. This version achieves near constant *amortized* time complexity for each operation. More precisely, for a sequence of m UNION-FIND operations on a set with n nodes, the total time required is $\mathcal{O}(m \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.

5 Travelling Salesperson Problem

The *Travelling Salesperson Problem* (TSP) is a classical $\mathcal{NP}$ -hard problem: given a weighted graph (G, w) find a minimum-weight cycle that visits every vertex of G exactly once.



Exercise [Trick-or-Treat Spooky Problem (TSP)]. It is Halloween Night and you decide to go around in your neighbourhood, visit every house, collect as many candies as you can and then return home to eat them.

The distances between the houses are represented in Table 2 and Figure 6.

Find the optimal route that allows you to leave and return to your home (H) collecting treats at every door, without having to knock at the same door twice.

	H	b	c	d	e
H	–	3	6	7	4
b	3	–	2	5	6
c	6	2	–	3	5
d	7	5	3	–	2
e	4	6	5	2	–

Table 2: Distance between houses in the neighbourhood.

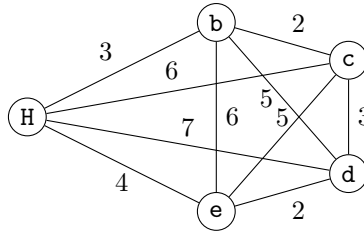


Figure 6: A plan of the neighbourhood.

6 Euler tours

Let $G = (V, E)$ be a (not necessarily simple) graph of order n and size m . A walk $(v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \cdots \xrightarrow{e_m} v_m)$ is called an *Euler trail* if for every $1 \leq i, j \leq m$ with $i \neq j$ we have $e_i \neq e_j$. Thus, an Euler trail in G is a trail that contains every edge, i.e., every edge in E is traversed exactly once.

An Euler trail that starts and ends at the same vertex is called an *Euler tour*.

A graph is called *Eulerian* if it admits an Euler tour.

Example 5 Let G be the simple graph in Figure 7. An Euler trail in G is given by the edge sequence $(e_5, e_1, e_2, e_3, e_4, e_8, e_7, e_6)$. There is no Euler tour in G , so the graph is not Eulerian.

One can also check that the graph K in the same figure is not Eulerian and does not contain any Euler trail neither.



Figure 7: Two non-Eulerian graphs.

A necessary condition for having an Euler trail or an Euler tour in a graph is connectedness. This condition is not sufficient, though, as shown in Example 5.

Theorem 1 (Euler (1736)) *A connected graph is Eulerian if and only if it is even.*

Proof.

($\Rightarrow$) Every vertex appearing k times in an Euler tour must have degree $2k$.
 ($\Leftarrow$) Let G be a connected even graph. We construct an Euler tour in the following way.

- 0)
 - $i \leftarrow 1$.
 - Let us choose any starting vertex v_i and a colour c_i .
- 1) Walk, as long as possible, along the edges (at random) of G . When leaving a vertex choose an edge with no colour. While traversing an edge, colour it using c_i .
- 2) If it is no longer possible to continue, we have returned to the starting vertex v_i and, moreover, all edges incident with v_i have already been coloured.
 - a) If there are no colourless edges go to Step 3) (RECONSTRUCTION).
 - b) If there are some colourless edges left:
 - $i \leftarrow i + 1$.
 - Choose a new colour c_i that has not been used so far.
 - Choose a new starting vertex v_i such that there are some colourless as well as some coloured edges incident with it (it is possible to find such a vertex v_i due to the assumption that G is connected).
 - Go to Step 1).
- 3) (RECONSTRUCTION) All edges of G have been coloured, i.e., the set E is partitioned by a set of disjoint closed walks $\{W_i\}_i$. We have to join these walks into one Euler tour.
 - i) Start at v_1 , follow the edges of the first closed walk.

- ii) If we meet a starting vertex v_k of a closed walk W_k which has not yet been processed (i.e., joined into the Euler tour), we "transfer" to W_k , traverse the whole closed walk W_k before returning back to the edges of the previous walk.

■

By applying rule *ii*) in Theorem 1, the walks can "nest" several times and they can be processed in an order different from the order in which they have been created.

Example 6 Let H be the graph in Figure 8.

- (Step 0) We start by choosing the vertex **b** and the colour **red**.
 (Step 1) We consider the closed walk $W_r = (b \xrightarrow{e_1} d \xrightarrow{e_2} c \xrightarrow{e_3} a \xrightarrow{e_4} b \xrightarrow{e_5} e \xrightarrow{e_6} c \xrightarrow{e_7} b)$.
 (Step 2) We choose the vertex **d** and the colour **green**.
 (Step 1) We consider the closed walk $W_g = (d \xrightarrow{e_8} h \xrightarrow{e_9} g \xrightarrow{e_{10}} d)$.
 (Step 2) We choose the vertex **g** and the colour **blue**.
 (Step 1) We consider the closed walk $W_b = (g \xrightarrow{e_{11}} f \xrightarrow{e_{12}} e \xrightarrow{e_{13}} g)$.
 (Step 2) All edges are coloured.
 (Step 3) The Euler tour is $(e_1, e_8, e_9, e_{11}, e_{12}, e_{13}, e_{10}, e_2, e_3, e_4, e_5, e_6, e_7)$.

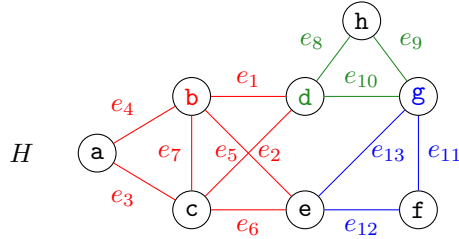


Figure 8: Finding an Euler tour in a graph.

The proof of Theorem 1 gives an algorithm to find Euler tours in an Eulerian graph. There is, however, a more efficient (but slightly more complicated) algorithm based on Tarry's traversal algorithm with complexity $\mathcal{O}(m+n)$, where m and n are respectively the size and the order of the graph.

- Choose a starting vertex v_0 .
- Traverse G using as long as possible the two rules:
 (R1) no edge can be used twice in the same direction;

- a) an edge which has not yet been used,
- b) an edge used to arrive to u with the exception of the edge of first arrival,
- c) the edge of first arrival to u .

- When there it is no longer possible to proceed in the traversal using rules (R1) and (R2): we have returned to v_0 , every edge has been traversed exactly twice (once in every direction); and one can prove that the return sequence corresponds to an Euler tour.

Using rules (R1) and (R2) we can find a closed walk

where we colour in blue the first arrival to a vertex, and in red the return sequence.

$$(e_7, e_6, e_{13}, e_{10}, e_8, e_9, e_{11}, e_{12}, e_5, e_4, e_3, e_2, e_1).$$


11

Theorem 2 *A connected graph has an Euler trail if and only if the number of vertices with odd degree is at most 2 (i.e., either 0 or 2).*

Example 8 The graphs G and K in Example 5 are not Eulerian. Indeed, they are connected but not even, since, e.g., $d_G(\mathbf{e}) = 3$ and $d_K(A) = 5$.

The graph G has an Euler trail, since only $\mathbf{d}$ and $\mathbf{e}$ have odd degrees.

The graph K does not have an Euler trail since it has 4 vertices of odd degree.