7.3 Spanning trees

We know that trees are connected graphs with the minimal number of edges. Hence trees become very useful in applications where our goal is to connect some places using the least number of connections. Let us form an undirected graph as follows: Vertices are places, edges are connections between them. Now, we are looking for a subgraph which is a factor (we need to connect **all** places), and which is a tree – but at the same time a factor of the given graph – such subgraphs will be called a spanning tree.

7.3.1 Definition. Given a connected graph G. A factor of G which is a tree is called a *spanning tree* of G. □

The following proposition characterizes graphs that have a spanning tree.

Proposition. A graph G has a spanning tree if and only it it is connected.

Justification. If a graph has a spanning tree it must be connected because a spanning tree is a connected graph.

Assume that G is connected. If G does not have a circuit then it is itself a tree, so it is its (only) spanning tree. Assume that G has a circuit, say C. Let us remove one edge of C from G. We obtain a subgraph of G, say G_1 which is connected (we removed an edge from a circuit). Therefore if G_1 does not have a circuit, then it is a tree and also a spanning tree of G. If G_1 contains a circuit we proceed in a similar way: we remove one edge of an existing circuit. After removing a finite number of edges (in fact $|E| - (|V| - 1)$) we get a connected graph with $|V| - 1$ edges, and it has to be a tree. So it is a spanning tree of G.

7.3.2 A Minimal Spanning Tree. In some applications, we know the price of an edge (e.g. the price which has to be paid for construction of a given connection). In this case we are not interested in an arbitrary spanning tree but in a spanning tree that has the least sum of prices of chosen edges. And this is, roughly speaking, a minimal spanning tree.

Definition. Given a connected graph G together with a mapping c which assigns to every $e \in E(G)$ a number $c(e)$.

A *minimal spanning tree* of $G = (V, E)$ is a spanning tree $K = (V, L)$ such that $\sum_{e \in L} c(e)$ is the smallest one (among all spanning trees of G). □

Let us mention that the number $c(e)$ is often called the *weight of* e or the *price of* e. Also, a graph G together with a mapping $c\colon E(G) \to \mathbb{R}$ is called a *weighted graph*. We will denote a weighted graph by (G, c).

Proposition. Any connected weighted graph (G, c) has a minimal spanning tree (which is not necessary unique). □

Justification. We already know that any connected graph has a spanning tree. On the other hand, there are only finitely many spanning trees (as $n - 1$ element subsets of a finite set of edges). So there must be a spanning tree with the least possible weight. □

A minimal spanning tree does not need to be unique; indeed, take any connected graph G with n vertices and n edges ($n > 2$), and define $c(e) = 1$ for every edge e. Then any spanning tree of G is a minimal one, and there are at least two spanning trees. (Note that a graph with n vertices and n edges cannot be a tree itself.)

7.3.3 An Algorithm for Finding a Minimal Spanning Tree. There are several algorithms that find a minimal spanning tree for a given connected weighted graph (G, c). We will show only one of them, the Kruskal's algorithm. It is an examples of so called "greedy algorithms", in other words, algorithms that in each step choose the most "promising" edge.

Kruskal's Algorithm for Finding a Minimal Spanning Tree.

Input: A connected graph G with the weight function c.
Output: A minimal spanning tree of G represented by its set of edges L.

1. Sort edges by their weights into non decreasing sequence, i.e.

$$c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m).$$

   Put $L := \emptyset$.

2. Go through edges in the given order. Insert e_i into L if and only if it does not close a circuit in L. If e_i closes a circuit, skip e_i and continue with e_{i+1}.

3. If L has $n-1$ edges (n is the number of vertices of G) end the algorithm, (V, L) is a minimal spanning tree.

□

Remarks.
1. If the input is a disconnected graph then the above algorithm will end with $|L| < |V|-1$ edges. So we do not need to check beforehand whether a given graph is connected.

2. If several edges have the same weight then the ordering in the first step of the algorithm "determines" the minimal spanning tree which the algorithm finds.

7.3.4 Example. Given an undirected graph $G = (V, E)$ with $V = \{1, \ldots, 7\}$ by the following matrix of weights (it means, at the position (i, j) we have either $c(\{i, j\})$ if $\{i, j\} \in E$, or "$-$" if $\{i, j\} \notin E$). Find a minimal spanning tree in (G, c) using the Kruskal's algorithm

$$\begin{pmatrix} - & 6 & 9 & - & - & - & 9 \\ 6 & - & 2 & 1 & 3 & - & - \\ 9 & 2 & - & 1 & - & - & 15 \\ - & 1 & 1 & - & 10 & 13 & 3 \\ - & 3 & - & 10 & - & 10 & 1 \\ - & - & - & 13 & 10 & - & 15 \\ 9 & - & 15 & 3 & 1 & 15 & - \end{pmatrix}$$

Solution. First we sort the edges (we state the weight of an edge in the brackets)

$$e_1 = \{2,4\}(1), e_2 = \{3,4\}(1), e_3 = \{5,7\}(1), e_4 = \{2,3\}(2), e_5 = \{2,5\}(3), e_6 = \{4,7\}(3),$$

$$e_7 = \{1,2\}(6), e_8 = \{1,3\}(9), e_9 = \{1,7\}(9), e_{10} = \{4,5\}(10), e_{11} = \{5,6\}(10), e_{12} = \{4,6\}(13),$$

$$e_{13} = \{3,7\}(15), e_{14} = \{6,7\}(15).$$

Put $L = \emptyset$.

Now we will go through edges in the given order and include e_i into L if and only if it does not close a circuit.

1. e_1 does not close a circuit, hence $L := \{\{2,4\}\}$.
2. e_2 does not close a circuit, hence $L := \{\{2,4\}, \{3,4\}\}$.
3. e_3 does not close a circuit, hence $L := \{\{2,4\}, \{3,4\}, \{5,7\}\}$.
4. e_4 closes a circuit formed by e_1, e_2 and e_4, hence L is the same as in 3.
5. e_5 does not close a circuit, hence $L := \{\{2,4\}, \{3,4\}, \{5,7\}, \{2,5\}\}$.
6. e_6 closes a circuit formed by e_1, e_5, e_3 and e_6, hence L is the same as in 5.
7. e_7 does not close a circuit, hence $L := \{\{2,4\}, \{3,4\}, \{5,7\}, \{2,5\}, \{1,2\}\}$.
8. e_8 closes a circuit formed by e_1, e_2, e_7 and e_8, hence L is the same as in 7.
9. e_9 closes a circuit formed by e_3, e_5, e_7 and e_9, hence L is the same as in 7.
10. e_{10} closes a circuit formed by e_1, e_5 and e_{10}, hence L is the same as in 7.
11. e_{11} does not close a circuit, hence $L := \{\{2,4\}, \{3,4\}, \{5,7\}, \{2,5\}, \{1,2\}, \{5,6\}\}$.

Since L contains $7-1=6$ edges, therefore

$$L = \{\{2,4\},\{3,4\},\{5,7\},\{2,5\},\{1,2\},\{5,6\}\}$$

is the set of edges of a minimal spanning tree of G. The weight (price) of L is

$$c(L) = 1+1+1+3+6+10 = 22.$$

7.4 Directed Trees

In this section we will introduce the notion of a rooted tree – a directed tree that contains a vertex r from which any other vertex is reachable by a directed path.

7.4.1 A Root.

Definition. Given a directed graph $G = (V, E, \varepsilon)$. We say that a vertex $r \in V$ is a *root* of G if there exists a directed path from r to any vertex of G. □

Note that in the definition of a root we can require existence of a directed walk instead of a directed path. Let us also note that a directed graph can have more than one root; indeed, in a cycle every vertex is a root. There are also directed graphs that do not have a root – find an example.

7.4.2 A Rooted Tree. A directed graph with a root which, at the same time, is a tree plays an important role in applications. For example, data structures based on rooted trees are widely used in computer science.

Definition. A directed graph which is a tree and contains a root is called a *rooted tree*. □

Since every graph with a root is connected (the opposite does not hold), we could define a rooted tree as a directed graph with a root that does not contain a circuit.

7.4.3 Proposition. Every rooted tree contains precisely one root. □

Justification. Assume, in contrary, that a rooted tree has roots r_1 and r_2, $r_1 \neq r_2$. Then there exists a directed path P_1 from r_1 to r_2 (indeed, r_1 is a root), as well as a directed path P_2 from r_2 to r_1 (indeed, r_2 is a root). Moreover, P_1 together with P_2 form a closed walk, and every closed walk contains at least one circuit. And this contradicts the fact that a tree does not contain a circuit. □

7.4.4 Remark. Given a rooted tree $G = (V, E)$ with its root r. Then for every vertex $v \in V$ there is a unique directed path from r to v. Indeed, there is a directed path form r to v by definition, and if there are two different directed paths then there is a closed undirected walk that always contains a circuit, which contradicts the fact that G is without circuits.

7.4.5 Successor, Predecessor, Leaf. We can distinguish different "types" of vertices in a rooted tree.

Definition. Let $G = (V, E)$ be a rooted tree. If (u, v) is an edge of G then u is called *predecessor* of v, and v is called a *successor* of u. A vertex which does not have a successor is called a *leaf*. □

Note that any leaf must have in-degree 1 and out-degree 0, so $d(v) = 1$ for any leaf. The other implication does not hold. A root of a tree may also have degree 1; but in this case it is the out-degree.

7.4.6 Levels, the Hight of a Rooted Tree. Vertices of a rooted tree can be divided into so called levels according to the number of edges that the only directed path from the root to the vertex has.

Definition. Given a rooted tree $G = (V, E)$ with the root r. A vertex $v \in V$ *belongs to k-th level* if the unique directed path from r to v contains precisely k edges.

The *hight of a directed tree* is the biggest k such that there is a vertex in k-th level. □

Remarks. 1. The hight of a rooted tree is, in fact, the number of edges in the longest directed path from r (to a leaf).

2. Notice that the root itself forms the 0-th level. Indeed, the only directed path from r to r is the trivial path containing 0 edges.

7.4.7 A Subtree Determined by a Vertex.

Definition. Given a rooted tree G. A *subtree generated by* a vertex v is the subgraph of G induced by the set of all vertices directly reachable from v. □

Remark. It is easy to see that a subtree generated by v is again a rooted tree, its root is v.

7.4.8 Binary Rooted Trees. Rooted trees where each vertex has at most two successors play an important role in data structures.

Definition. A rooted tree is called a *binary rooted tree* if every vertex has at most two successors.

In binary rooted trees we speak about the *right successor* and the *left successor* of a vertex v. A *right subtree*, a *left subtree*, of v is the subtree generated by the right successor, or the left successor of v, respectively. □

An example of a binary rooted tree is a data structure called a heap. It is a basis of the heap sort, one of the fast sorting algorithms.

7.5 Acyclic Graphs

Trees are connected graphs without circuits. Acyclic graphs are directed graphs that do not contain a cycle. Unlike trees, acyclic graphs do not need to be connected.

7.5.1 Definition. A directed graph is called *acyclic* if it does not contain a cycle. □

We will give another characterization of acyclic graphs; acyclic graphs are those directed graphs that admit a topological sort of vertices or/and a topological sort of edges.

7.5.2 Topological Sort of Vertices.

Definition. Given a directed graph $G = (V, E, \varepsilon)$ with n vertices. A sequence of vertices

$$v_1, v_2, \ldots, v_n$$

is called a *topological sort*, also *topological ordering*, if the following holds: for every edge e with initial vertex v_i and terminal vertex v_j it necessarily holds that $i < j$. □

Informally, edges go from a vertex with a smaller index to a vertex with a bigger index.

7.5.3 Topological Sort of Edges.

Definition. Given a directed graph $G = (V, E, \varepsilon)$ with m edges. A sequence of edges

$$e_1, e_2, \ldots, e_m$$

is called a *topological sort*, also *topological ordering*, if for every two edges e_i, e_j for which the terminal vertex of e_i is the initial vertex of e_j it necessarily holds that $i < j$. □

7.5.4 Proposition. In every acyclic graph there exists at least one vertex with in-degree 0. □

Justification. We proceed by contradiction. Assume that every vertex of a graph G has its in-degree at least 1. Choose any vertex v, and denote it by v_1. Since $d^-(v_1) \geq 1$, there is an edge e_1 with $TV(e_1) = v_1$. Denote by v_2 the vertex $IV(e_1)$. We have $v_2 \neq v_1$; indeed, otherwise e_1 is a cycle. Since $d^-(v_2) \geq 1$, there is an edge e_2 with $TV(e_2) = v_2$. Denote v_3 the initial vertex of e_2. Then v_3 is a new vertex; indeed, if $v_3 = v_1$ or $v_3 = v_2$ then G contains a cycle. Again, $d^-(v_3) \geq 1$, hence there is e_3 such that $TV(e_3) = v_3$, and $v_4 = IV(e_3)$ must be a new vertex, otherwise G contains a cycle.

In such a way, we get a directed path $v_n, e_{n-1}, v_{n-1}, e_{n-2}, \ldots, v_2, e_1, v_1$ containing all vertices of G. But $d^-(v_n) \geq 1$, so there must be an edge e_n with $TV(e_n) = v_n$. On the other hand, $IV(e_n)$ must be among vertices $v_1, \ldots, v_n$ (indeed, we do not have more vertices); therefore G contains a cycle – a contradiction.

7.5.5 Theorem. Given a directed graph G. Then the following conditions are equivalent:

1. G is an acyclic graph;
2. G has a topological sort of vertices;
3. G has a topological sort of edges.

Justification. First, we show that 2 implies 3. Assume that $v_1, v_2, \ldots, v_n$ is a topological sort of vertices of G. We list at first all edges with the initial vertex v_1, then all edges with the initial vertex v_2, etc. and finally all edges with the initial vertex v_n. This sequence contains all edges, indeed, every edge has its initial vertex, hence was listed. The fact that it is a topological sort of edges of G is evident.

3 implies 2: Let $e_1, e_2, \ldots, e_m$ be a topological sort of edges of G. We go through edges in this order and we list the initial vertex of the edge if it has not been listed before. In this way we get a sequence $v_1, v_2, \ldots, v_k$ of (not all) vertices of G. We add, in an arbitrary order, the remaining vertices. The sequence $v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, v_n$ is a topological sort of vertices of G.

It is easy to notice that if a graph G is not acyclic then it does not have a topological sort of vertices; indeed, a cycle cannot be topologically ordered. The fact that any acyclic graph has a topological sort of vertices will be shown by an algorithm bellow that constructs a topological sort of vertices.

7.5.6 An Algorithm for Topological Sort. The algorithm is based on the proposition 7.5.4. First, we find all vertices with in-degree 0, since any such vertex v can be the first vertex of a topological sort of vertices. Then we remove all edges e with the initial vertex v. This will be done by decreasing the in-degree of $TV(e)$ by 1. If the new in-degree of $TV(e)$ becomes 0, then $TV(e)$ may be inserted in the topological sort of G. The set M will always contain all vertices with the actual in-degree 0; i.e. vertices that can be listed in a topological sort of G.

More precisely:

An Algorithm for Finding a Topological Sort of Vertices.

Input: An acyclic graph G.
Output: A topological sort of vertices of G.

1) For each vertex v we calculate its in-degree $d^-(v)$.

2) The set M contains all vertices with in-degree 0.
We put $i := 1$.

3) While $M \neq \emptyset$ we do

3a) We choose a vertex v from M and delete it from M.
We put $v_i := v$, $i := i + 1$.

3b) For every edge e with $IV(e) = v$ we do
$d^-(TV(e)) := d^-(TV(e)) - 1$. If $d^-(TV(e)) = 0$ we insert $TV(e)$ into the set M.