Chapter 2

Trees and arborescences

1. Introduction

Trees are a class of graphs that are widely used to organize data and solve problems. They describe hierarchical relationships and are suitable for recursive and/or efficient processing. For example, when sorting an array, textit{heap sort} (which is based on binary trees) has a more interesting complexity than other sorting algorithms. This chapter introduces the concept of trees in graph theory and explains how spanning trees are built for an arbitrary graph.

2. Basic concepts

2.1. Definitions and properties

A tree in mainly a simple graph. Hence, all considered graphs in this chapter are simple.

Definition 2.1

A *tree* is a undirected, connected and acyclic graph.

If the graph is of order n, it has $n-1$ edges (how can we show this?). The graph is figure 2.1 illustrates a tree with 5 vertices (and 4 edges).

Let G be a tree with order n (with $n \geq 2$). The following properties are equivalent:

- G is connected and acyclic.
- If the order of G is n then its size is $n-1$.
- G is connected and minimal, which means that removing one edge makes the graph non-connected.
- G is acyclic and maximal, which means that adding one edge makes the graph cyclic.
- There is exactly one walk between each pair of vertices in G.

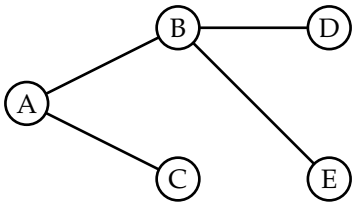


Figure 2.1 - example of a tree with order 5

It is very instructive to demonstrate these equivalences for interested students.

In a tree, any pendant vertex is called a *leaf*. In the tree in figure 2.1 , the leaves are: C, D, E. We can additionally state the following properties (the proof can serve as a useful exercise):

- A tree with an order greater than 1 has at least two leaves.
- A tree can be colored with two colors, making it a bipartite graph. Be careful; not all bipartite graphs are trees.

2.2. Forest

A *forest* is a network whose components are trees that may have an isolated vertex.

A forest is obtained by relaxing the connectedness constraints on trees. In other terms, a forest is an acyclical graph. Besides, any partial graph of a tree forms a forest. In figure 2.1 , we can remove the edge (B, D), to get the forest given in figure 2.2 , with the components: $\{A, B, C, E\}$ and $\{D\}$.

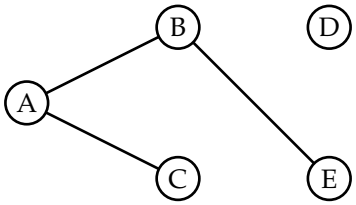


Figure 2.2 - example of a forest

2.3. Root and anti-root

In directed graphs, a *root* is a vertex that can reach every other vertex. In other terms, a vertex is a root if there is at least one path connecting it to each other vertex in the graph. An *anti-root* is a vertex that can be reached from every other vertex in the graph.

In the graph in figure 2.3 , vertex A is a root and vertex C is an anti-root.

If a graph does not contain a circuit, topological sorting places the possible root at the first level and the potential anti-root at the last level.

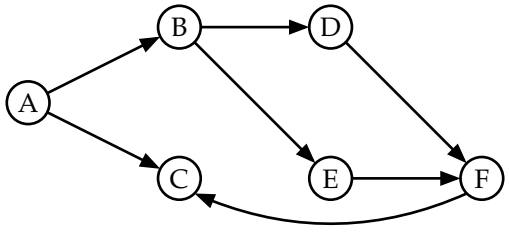


Figure 2.3 - example of a root and an anti-root

2.4. Arborescence

An *arborescence* is a directed graph G satisfying the following properties:

1. If the arcs of G are converted to edges, the resulting undirected graph is a tree (G is connected and acyclic).
2. G has a root S.
3. Every vertex $x \neq S$ has a single predecessor. (i.e. $\forall x \neq S : d^-(x) = 1$).

4. The third property can be stated alternatively as: there is only one path from the root S to any other vertex in G.

The directed graph in figure 2.4 is an arborescence. Its root is A. If the arcs of this graph are inverted, then we obtain an *anti-arborescence*.

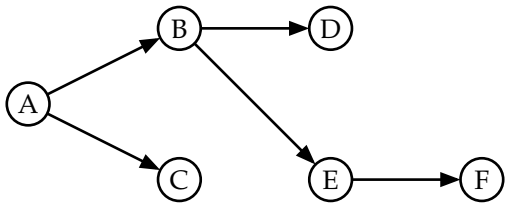


Figure 2.4 - example of an arborescence

Arborescences have a wide range of applications including:
- Hierarchical structure: such as dividing a book into chapters, sections, subsections, paragraphs, and so on.
- Genealogy: in a family, we can create a graph of the family members, with arcs representing parenthood relationships. This concept has been adopted by object-oriented programming.
- Many algorithms rely on arborescences to solve problems more efficiently.

3. Minimal spanning tree

3.1. Definitions

Definition 2.2

Let $G = (X, E)$ be a connected, undirected graph. A *spanning tree* is a partial graph of G that represents a tree.

It is possible to build multiple spanning trees for a given graph, the number of which is determined by the topology of G. For example, if G is the complete graph K_n, the number of spanning trees is n^{n-2}. The number of spanning trees in the complete bipartite graph $K_{m,n}$ is $m^{n-1} \times n^{m-1}$. In other cases, determining the number of spanning trees may need a more complex calculation.

Assume that G is valued, meaning that each edge has a value or *weight* (in this example, a real number). The weight can be a cost or a reward associated with the edge. Let $T = (X, E_T)$ be a spanning tree over G. The weight of T is defined as: $w(T) = \sum_{e \in E_T} w(e)$, where $w(e)$ is the weight of edge e. The *minimum spanning tree* is the one that minimizes $w(T)$.

In figure 2.5 , a spanning tree is shown in thick lines. However, this is not the minimum spanning tree (the weight equals 24 in this case).

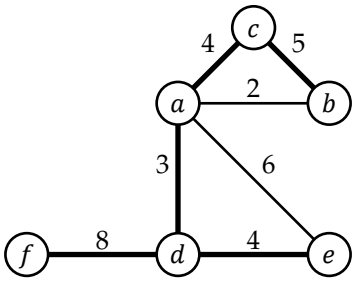


Figure 2.5 - example of a spanning tree of a valued graph

3.2. Kruskal's algorithm

This algorithm is used to build the minimum spanning tree over a valued undirected connected graph.

Algorithm of Kruskal to build the minimum spanning tree of a valued graph G

```
1  n = order(G)
2  m = size(G)
3  Let a_1, a_2, ..., a_m be the the sorted edges      ▷ w(a_k) ≤ w(a_{k+1}), ∀k
4  F = ∅, k = 1                                       ▷ initially, the spanning tree has no edge
5  while k ≤ m and card(F) < n − 1
6      if adding a_k does not create a cycle
7          F = F ∪ {a_k}
8      k = k + 1
```

We will apply Kruskal's algorithm to the graph in figure 2.6 .

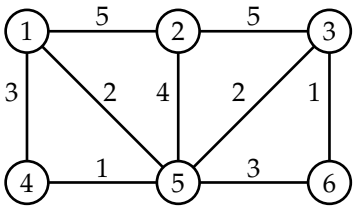
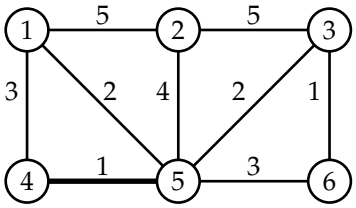
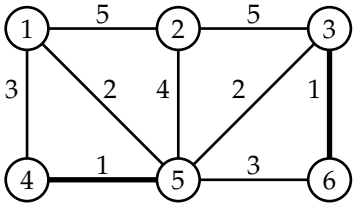
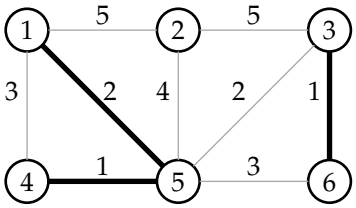
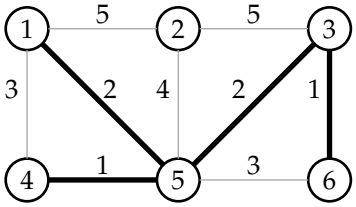


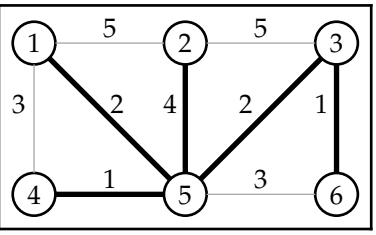
Figure 2.6 - example of a valued graph

Let us begin by sorting the edges:

Edge	(4,5)	(3,6)	(1,5)	(3,5)	(1,4)	(5,6)	(2,5)	(1,2)	(2,3)
Weight	1	1	2	2	3	3	4	5	5
Index	1	2	3	4	5	6	7	8	9

The algorithm then proceeds according to the values k:

k	**Decision**	**Graph**
1	Add edge (4,5)	
2	Add edge (3,6)	
3	Add edge (1,5)	
4	Add edge (3,5)	
5	Skip edge (1,4)	the same graph
6	Skip edge (5,6)	the same graph

7	Add edge (2,5) (end)	

The minimum spanning tree is given in the last row of the table. Its weight equals 10.

The minimum spanning tree has many applications. For instance, in a message-based communication system, the minimum spanning tree allows messages to be broadcast to all nodes in a network at a minimal cost.

3.3. Applying Kruskal's algorithm on a non-connected graph

Kruskal's algorithm can also be applied on non-connected graphs. However, the result is a minimum-spanning forest. Let us consider the graph in figure 2.7 , which has two components. We should adjust the loop's halt condition. The number of edges to be added equals the graph's order minus the number of components.

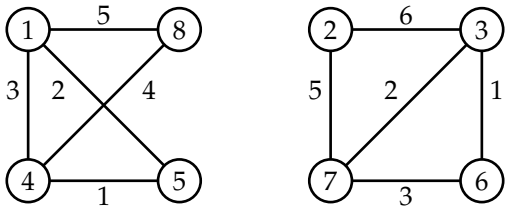
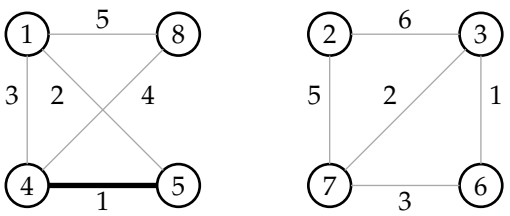
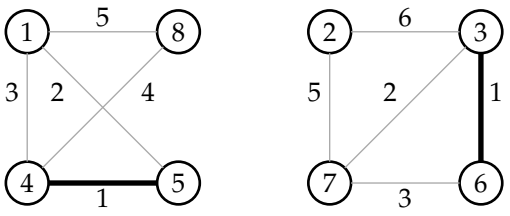
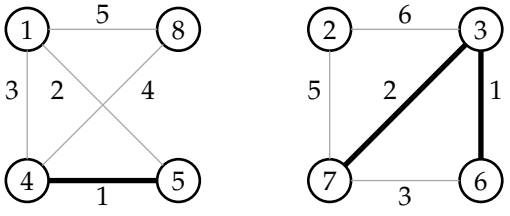


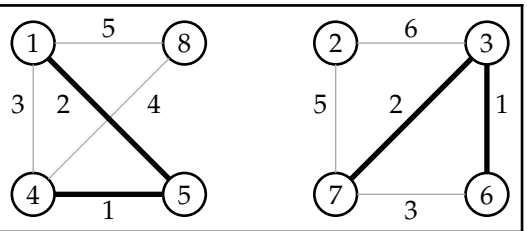
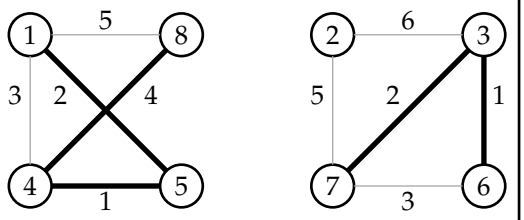
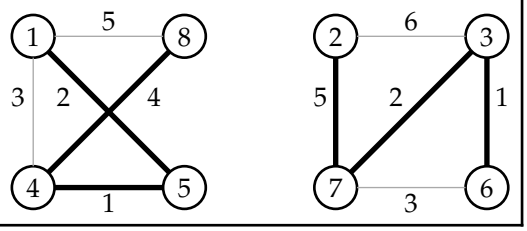
Figure 2.7 - example of a non-connected valued graph

Let us sort the edges:

Edge	(3,6)	(4,5)	(3,7)	(1,5)	(6,7)	(1,4)	(4,8)	(2,7)	(1,8)	(2,3)
Weight	1	1	2	2	3	3	4	5	5	6
Index	1	2	3	4	5	6	7	8	9	10

The algorithm proceeds as follows:

k	**Decision**	**Graph**
1	Add edge (4,5)	
2	Add edge (3,6)	
3	Add edge (3,7)	

4	Add edge (1,5)	
5	Skip edge(6,7)	The same graph
6	Skip edge (1,4)	The same graph
7	Add edge (4,8)	
8	Add edge (2,7) (end)	

Kruskal's algorithm is not the only algorithm that can build the minimum spanning tree. Other algorithm like Sollin's algorithm and Prim's algorithm can also be used.