Special Mathematics Lecture

Graph theory

Nagoya University, Spring 2024

Lecturer: Serge Richard
Teaching assistant: Sota Kitano

Goals of these Lectures notes:

Provide the necessary background information for understanding the main ideas of graph theory. These notes correspond to 14 lectures lasting 90 minutes each.

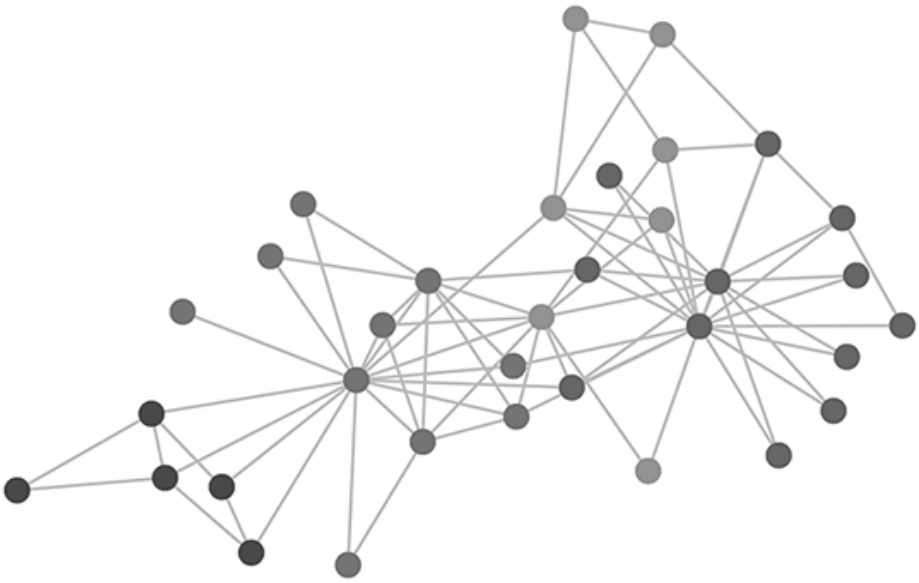


Figure 1: A finite graph

Website for this course:
http://www.math.nagoya-u.ac.jp/~richard/SMLspring2024.html

Comments or corrections are welcome:
richard@math.nagoya-u.ac.jp

Contents

Chapter 1

The basics

1.1 Graphs

In this section we provide the main definitions about graphs.

Definition 1.1 (Graph)**.** *A* graph *G consists in a pair $G = (V, E)$ of two sets together with a map $i : E \to V \times V$ assigning to every $e \in E$ a pair (x, y) of elements of V*[1]*. Elements of V are called* vertices*, elements of E are called* edges*. If $i(e) = (x, y)$, the vertices x and y are also called the* endpoints *of e.*

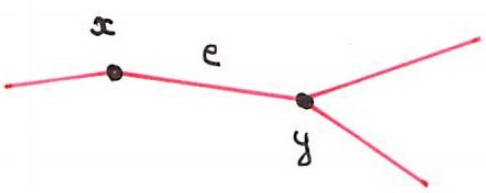


Figure 1.1: One edge, two vertices

We say that the graph is *undirected* or *unoriented* if we identify the pairs (x, y) and (y, x) in $V \times V$, while the graph is *directed* or *oriented* if we consider (x, y) distinct from (y, x) in $V \times V$. For undirected graphs, when $i(e) = (x, y)$ we say that the edge e links x to y or y to x, without any distinction, or that e is an edge between x and y, or between y and x. For directed graphs, if $i(e) = (x, y)$ we often call x *the initial vertex* for e, or *the origin* of e, while y is called *the terminal vertex* or *the target*. In this case, we also set $\mathrm{o} : E \to V$ and $\mathrm{t} : E \to V$ with $\mathrm{o}(e) = x$ and $\mathrm{t}(e) = y$ such that $i(e) = (\mathrm{o}(e), \mathrm{t}(e))$, the *origin* and the *terminal* maps, see Figure 1.2. Some authors also use *head* and *tail* for the terminal vertex and the initial vertex, respectively. For directed or undirected graphs, we also say that x and y are *connected* or *adjacent* whenever there exists an edge (directed or not) between them.

One observes that Definition 1.1 allows any graph to have *loops*, when $i(e) = (x, x)$, and *multiple edges* between the same vertices, namely when for some fixed $x, y \in V$ there exist $e_1, \dots, e_n$ with $i(e_j) = (x, y)$ for all $j \in \{1, \dots, n\}$, see Figures 1.3 and 1.4.

Note that directed graphs are also called *digraphs*, and that graphs with loops and / or multiple edges are also called *multigraphs*. If a graph contains both directed edges (often represented by arrows) and undirected edges (just represented by a segment), we call it *a mixed graph*. Clearly, an undirected graph can be obtained from a directed graph by forgetting the information about the direction (one simply identifies (x, y) with (y, x) in $V \times V$), while a directed graph can be constructed from an undirected one by assigning a direction to each edge (for example by fixing the origin of each edge).

[1] According to this definition one should write $G = (V, E, i)$ but shall keep the shorter and common notation $G = (V, E)$.

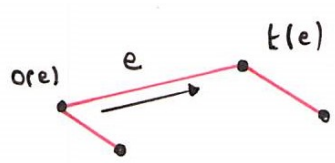


Figure 1.2: Oriented edge

Figure 1.3: Loop

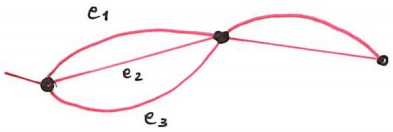


Figure 1.4: Multiple edges

Remark 1.2 (Simple graph)**.** *When a graph has no loop and no multiple edge, we say that the graph is* simple*. In such a case, the set E can be identified with a subset of $V \times V$. Indeed, an edge can be simply written $e = (x, y)$ since there is no ambiguity about the indexation. In an undirected simple graph, the notations (x, y) and (y, x) would represent the same edge, while for a directed simple graph they would not.*

Definition 1.3 (Finite graph, order, and size)**.** *A graph $G = (V, E)$ is* finite *if V and E contain only a finite number of elements. A graph is infinite if either V or E (or both) contain(s) an infinite number of elements. In the infinite case, it is assumed that the sets V and E are countable. For finite graph, the* order *of G, denoted by $|G|$, corresponds to the cardinality of V, while the* size *of G, denoted by $\|G\|$, corresponds to the cardinality of E.*

Let us provide a few definitions related to vertices.

Definition 1.4 (Degree and neighbourhood)**.** *Let x be a vertex of a graph $G = (V, E)$.*

(i) The degree of x*, or* valence of x*, denoted by* $\deg(x)$*, corresponds to the number of edges connected at x, with a loop giving a contribution of 2,*

(ii) The set of neighbours of x, denoted by $N(x)$, corresponds to the set of vertices connected to x by an edge,

A vertex x with $\deg(x) = 1$ is sometimes called *a leaf*, and a vertex x with $\deg(x) = 0$ is said to be *isolated*. However, one has to be careful for graphs admitting loops. Is a vertex having only one (or more) loop(s) and no other link isolated or not ? The answer depends on the authors. In principle, we shall consider that a vertex which has no link to any other vertex is isolated, even if it possesses some loops.

Based on these notions, we define *the minimum degree of a graph* as $\delta(G) := \min\{\deg(x) \mid x \in V\}$ and *the maximum degree of a graph* as $\Delta(G) := \max\{\deg(x) \mid x \in V\}$. Also, a graph is k-regular if $\deg(x) = k$ for all $x \in V$, see Figure 1.5.

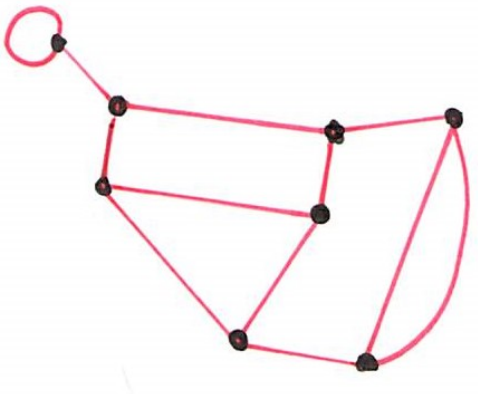


Figure 1.5: A 3-regular graph

Let us state an easy result based on the notion of degree.

Lemma 1.5 (Euler's degree-sum theorem)**.** *The sum of the degrees of the vertices of a finite graph is twice the number of edges.*

Consider now a graph $G = (V, E)$ and another graph $G' = (V', E')$ with $E' \subset E$ and $V' \subset V$, and with $i' = i|_{E'}$ whenever it is defined. In this case G' is called a *subgraph* of G, and one says that G contains G'. This notion is rather simple, but one can be more precise.

Definition 1.6 (Induced subgraph)**.** *A subgraph $G' \subset G$ is* an induced subgraph *if, for all $x, y \in V'$ and all $e \in E$ with $i(e) = (x, y)$ one has $e \subset E'$. We also say that V'* induces *or* spans *G' in G, and write $G' = G[V']$.*

From this definition, one can define the suppression of vertices. If $G = (V, E)$ is a graph and if $U \subset V$, then we write $G - U$ for $G[V \setminus U]$. In other words, $G - U$ corresponds to the graph containing all vertices of $V \setminus U$ and all edges of G which do not have an endpoint in U. For edges, if $F \subset E$, one write $G - F$ for the graph $(V, E \setminus F)$.

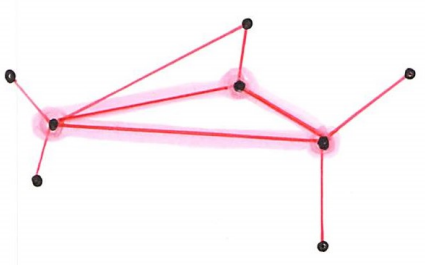


Figure 1.6: An induced graph in pink

Remark 1.7 (Union of graphs)**.** *The notion of union of two graphs needs to be defined with great care. Indeed, let us consider $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. If we consider a disjoint union (denoted by $\sqcup$), then $G := G_1 \sqcup G_2$ with $G := (V, E)$ and $V = V_1 \sqcup V_2$, $E = E_1 \sqcup E_2$, with no identification between some elements of the sets V_j, or of the sets E_j, for $j \in \{1, 2\}$. If we want to identify some elements, then one has to do it very precisely.*

One more important definition related to the division of a graph into two parts:

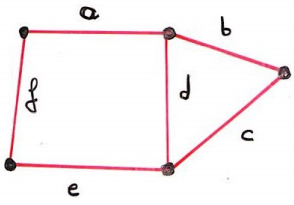
Definition 1.8 (Bipartite graph)**.** *An undirected graph $G = (V, E)$ is* bipartite *if the set of its vertices can be divided into two subsets V_1 and V_2 such that any $e \in E$ has one endpoint in V_1 and the other endpoint in V_2. The sets V_1 and V_2 are called* the bipartition subsets.

Figure 1.7: Bipartite graph

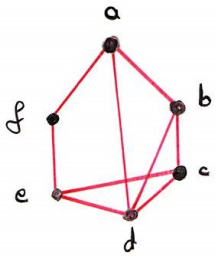
It is rather clear that a bipartite graph can not have any loop. On the other hand, multiple edges do not prevent a graph to be bipartite. There exists also a kind of duality between some graphs, as provided in the following definition.

Definition 1.9 (Line graph)**.** *The* line graph *of an undirected graph $G = (V, E)$ (without loop) consists in a new graph $L(G) := (V', E')$ with $V' = E$ and two vertices in V' are adjacent if and only if they had a common vertex in G.*

The representation of a line graph is provided in Figure 1.8. Note that the definition of a line graph for an undirected graph with loop does not seem to be completely clear and standard.



(a) Original graph G



(b) Line graph $L(G)$

Figure 1.8: Graph and its line graph

Note that there exists a lot of classical graphs which are presented in any book, as for example in Section 1.2 of [GYA]. We shall not present these examples except when necessary.

1.2 Walks and paths

As in the previous section, the following definitions depend slightly on the authors. We always choose the definitions which look quite general and flexible.

Definition 1.10 (Walk)**.** *A walk W of length N on a graph $G = (V, E)$ is an alternating sequence*

$$W = (x_0, e_1, x_1, e_2, \dots, x_{N-1}, e_N, x_N)$$

with the requirement that $i(e_j) = (x_{j-1}, x_j)$ for $j \in \{1, \dots, N\}$,

An illustration of a walk is given in Figure 1.9.

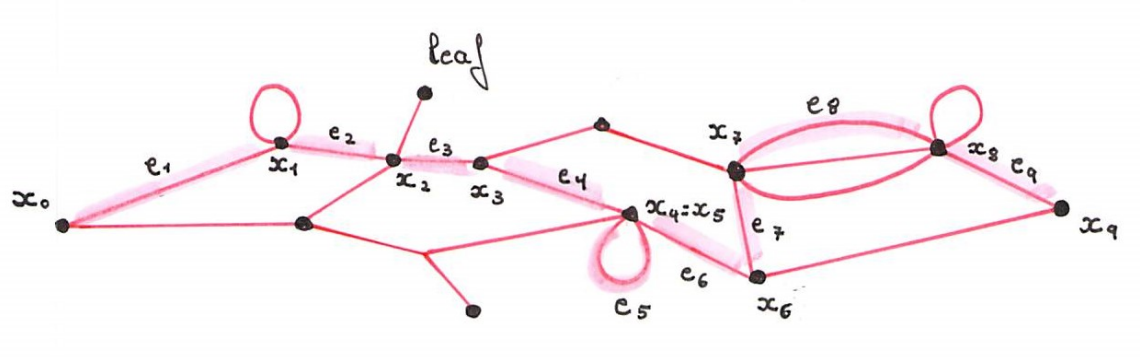


Figure 1.9: One walk with one loop included

Let us observe that this definition is quite flexible. Indeed, there is no restriction about intersection of a walk with itself. Also this definition is valid for directed and undirected graphs: for the former, it means that a walk is always going in the direction of the arrows. Note also that this definition of walk is compatible with loops and multiple edges, and take them into account. We say that the above walk *starts at* x_0 and *ends at* x_N, or is from x_0 to x_N. We also say that the walk is *closed* if $x_0 = x_N$.

Remark 1.11. *For simple graphs, a walk is uniquely defined by the sequence $(x_j)_{j=0}^N$ since multiple edges or loops are not allowed. The list of edges is therefore not necessary.*

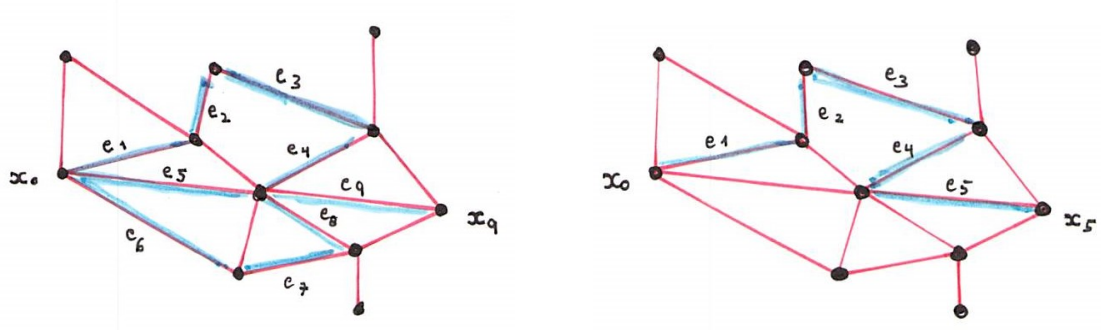
For $k \in \{1, 2\}$, consider two walks W_k with start at x_0^k and ends at $x_{N_k}^k$. We say that these walks are *composable* if $x_{N_1}^1 = x_0^2$, and in this case we define their *composition*. This operation consists in defining the new walk $W = W_1 W_2$ with

$$W = (x_0^1, e_1^1, x_1^1, e_2^1, \dots, x_{N_1-1}^1, e_{N_1}^1, x_{N_1}^1, e_1^2, x_1^2, e_2^2, \dots, e_{N_2}^2, x_{N_2}^2).$$

This new walk is of length $N_1 + N_2$.

One walk can also be concatenated: Consider $W = (x_0, e_1, x_1, e_2, \dots, x_{N-1}, e_N, x_N)$ and suppose that $x_j = x_k$ for some $0 \le j < k \le N$. Then one concatenated walk consists in removing $x_{j+1}, \dots, x_k$ and $e_{j+1}, \dots, e_k$ to the alternating sequences defining the walk W. One thus get a new walk starting at x_0 and ending at x_N which is "shorter" than the initial walk. Note that several concatenations might be possible on a given walk, and do not always lead to the same resulting walk, see Figure 1.10

The notion of walks is convenient because the addition of two composable walks is again a walk. However, walks have some drawbacks because "the walker is allowed to do some detours". Let's be more efficient !



(a) Before concatenation (b) After concatenation

Figure 1.10: One concatenation of a walk

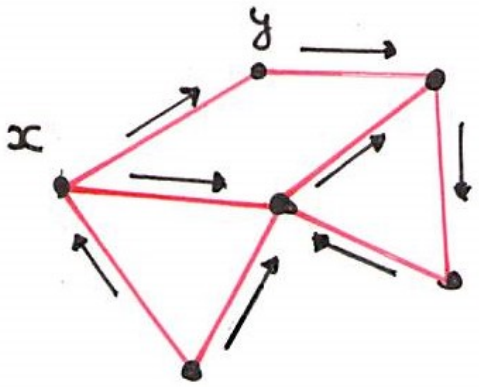


Figure 1.11: $\mathrm{d}(x, y) = 1$ but $\mathrm{d}(y, x) = \infty$

Definition 1.12 (Trail and path)**.** *A* trail *is a walk with no repeated edges. A* path *is a trail with no repeated vertices, except possibly the endpoints x_0 and x_N. The length of a trail or of a path corresponds to the length of the corresponding walk.*

Note that for multiple edges, one has to be careful when defining a trail, since edges linking the same two vertices can still appear in a trail, if each of them does not appear more than once. On the other hand, in a path this is not possible since two vertices would appear at least twice.

The following notions could have been defined in terms of walks, but they are always realized by a path. For that reason, it is more natural to express them in terms of paths.

Definition 1.13 (Distance in a graph)**.** *Given two vertices x, y in a graph G,* the distance $\mathrm{d}(x, y)$ between x and y *corresponds to the length of the shortest path between x and y. If there is no path from x to y, one sets* $\mathrm{d}(x, y) = \infty$.

It is clear that for undirected graphs, one has $\mathrm{d}(x, y) = \mathrm{d}(y, x)$. For directed graphs, $\mathrm{d}(x, y)$ can be different from $\mathrm{d}(y, x)$, see Figure 1.11.

Definition 1.14 (Eccentricity)**.** *The* eccentricity $\mathrm{ecc}(\cdot) : V \to [0, \infty]$ *in a graph $G = (V, E)$ is defined as the distance between a given vertex x to the vertex farthest to x, namely*

$$\mathrm{ecc}(x) := \max_{y \in V} \mathrm{d}(x, y).$$

Two additional notions for a graph can be defined in terms of the eccentricity:

Definition 1.15 (Diameter and radius)**.**

(i) The diameter $\mathrm{diam}(G)$ *of a graph* $G = (V, E)$ *is defined by the maximal eccentricity on the graph, namely*

$$\mathrm{diam}(G) := \max_{x \in V} \mathrm{ecc}(x) = \max_{x,y \in V} \mathrm{d}(x, y).$$

(ii) The radius $\mathrm{rad}(G)$ *of a graph* $G = (V, E)$ *is the minimum of the eccentricities, namely*

$$\mathrm{rad}(G) := \min_{x \in V} \mathrm{ecc}(x).$$

In a very vague sense, one can think about these two notions respectively as the diameter of a ball containing the entire graph, and as the maximum radius of a ball contained in the graph and centered at the best place (the "center" of the graph, as defined below).

Let us emphasize that these two concepts can take the value ∞. It should also be noted that these notions can be different for a directed graph and for the subjacent undirected graph, once the direction on the edges have been removed. Related to the notion of radius of a graph, one can also look for the "center" of a graph.

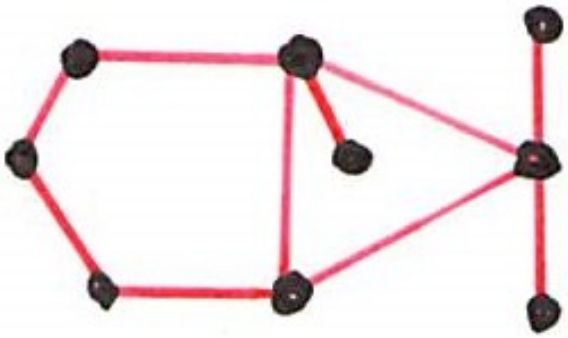


Figure 1.12: $\mathrm{diam}(G) = 4$, $\mathrm{rad}(G) = 2$

Definition 1.16 (Central vertex)**.** *A* central vertex *of a graph* G *is a vertex with minimum eccentricity, which means* x *is a central vertex if* $\mathrm{ecc}(x) = \mathrm{rad}(G)$.

This definition does not imply that there is only one central vertex. In fact, one can even construct graphs for which all vertices are the central vertex. Even if uniqueness does not hold in general, existence always holds: there always exists at least one central vertex (with the exception of the trivial graph with no vertex).

Paths can also be used for defining the notion of connected graphs.

Definition 1.17 (Connected)**.** *An undirected graph* $G = (V, E)$ *is* connected *if for any* $x, y \in V$ *there exists a path between* x *and* y*. A directed graph is* connected *if the underlying undirected graph is connected.*

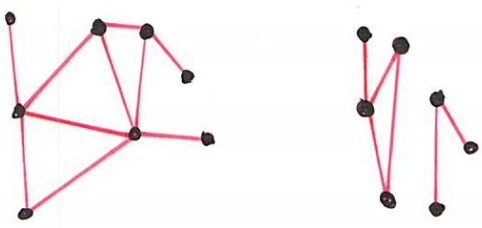


Figure 1.13: One connected graph, one not connected graph

We observe that this definition fits with our definition of an isolated vertex (when it has no link to any vertex different from itself). Indeed, any such point is isolated, and any graph having such a vertex would not be connected. Note also that with Definition 1.17, the direction is suppressed. For directed graphs, some authors

say that they are *weakly connected* when they are connected with the notion defined above. This is in contrast with the following definition, which is more useful for directed graphs:

Definition 1.18 (Strongly connected)**.** *An directed graph $G = (V, E)$ is* strongly connected *if for any $x, y \in V$ there exists one path from x to y.*

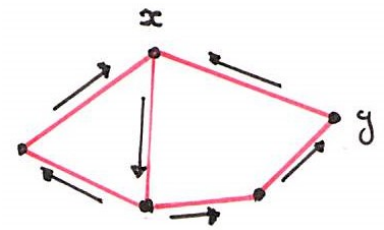


Figure 1.14: Strongly connected graph

Observe that for strongly connected graphs, the distances $d(x, y)$ and $d(y, x)$ are never equal to ∞, for any pair of vertices (x, y). However, these two quantities can still be different, see Figure 1.14.

1.3 Cycles

The notion of closed walks has already been introduced, and since paths are special instance of walks, closed paths are also already defined. A name is given to non-trivial closed paths (here non-trivial means a path not reduced to a single vertex).

Definition 1.19 (Cycle)**.** *A* cycle *is a (non-trivial) closed path.*

Observe that for simple graphs, a cycle has always a length of at least 3. On the other hand, for graphs with loops or multiple edges, a cycle can be of length 1 (for a loop) or of length 2 (between 2 vertices linked by multiple edges). If a graph has no cycle, it is called *acyclic*, and we shall come back to them subsequently. For graphs with cycles, we can wonder what is the length of the shortest cycle ?

Definition 1.20 (Girth)**.** *The* girth *of a graph G is the length of the shortest cycle in G, and it is denoted by* girth(G)*. If G is acyclic, then its girth is ∞.*

In the next statement, a characterization of bipartite graphs in terms of cycles is provided. A proof is available in [GYA, Thm. 1.5.4].

Theorem 1.21. *An undirected graph is bipartite if and only if it has no cycle of odd length.*

The following extension of this result has been proposed by Duc Truyen Dam, and the proof has been provided by Chang Sun (both former G30 students).

Theorem 1.22 (Strongly connected bipartite graphs)**.** *A strongly connected oriented graph G is bipartite if and only if it has no cycle of odd length.*

Figure 1.15: A Hamiltonian graph

Let us now introduce a cycles with an additional property:

Definition 1.23 (Hamiltonian cycle and graph)**.** *A cycle that includes every vertex of a graph is called a* Hamiltonian cycle*. A graph having a Hamiltonian cycle is called* a Hamiltonian graph*.*

Let us also stress that a Hamiltonian cycle has to go through every vertices, but it will not use all edges of a graph in general (and is not allowed to use twice any vertex). These cycles are important because their existence means that there exists a closed path visiting all vertices once. Hamiltonian cycles are also related to the famous *travelling salesman problem*. What is the analogue definition for edges instead of vertices ? The answer is

provided below. Note that we consider trails instead of paths, which means that a vertex can be visited more than once, but any edge can be used only once.

Definition 1.24 (Eulerian trail and graph)**.**

(i) An Eulerian trail *is a trail that contains every edge of a graph.*

(ii) An Eulerian tour *is a closed Eulerian trail.*

(ii) An Eulerian graph *is a connected graph which possesses an Eulerian tour.*

Let us remark that there are natural questions related to the above two notions. Given a connected graph $G = (V, E)$, does it posses a Hamiltonian cycle, or is it an Eulerian graph ? One answer for simple graphs is provided in [Die, Thm. 1.8.1] while the proof for more general graphs is given in [GYA, Thm. 4.5.11].

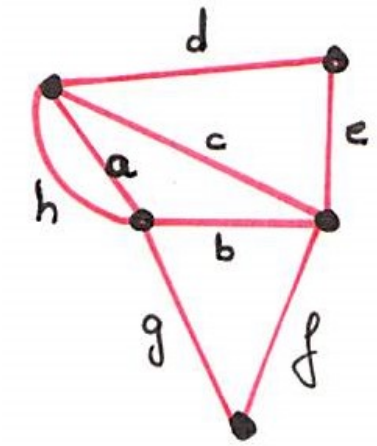


Figure 1.16: Eulerian graph

Theorem 1.25. *A connected, undirected and finite graph is Eulerian if and only if every vertex has even degree.*

Let us finally gather in the next statement a few results which link some of the notions introduced so far. We also recall that $\delta(G)$ and $\Delta(G)$ denote the minimal and the maximal degree of a graph.

Theorem 1.26. *Let G be a simple undirected finite graph:*

(i) G contains a path of length $\delta(G)$ and a cycle of length at least $\delta(G)+1$ (provided $\delta(G) \geq 2$).

(ii) If G contains a cycle, then $\mathrm{girth}(G) \leq 2\,\mathrm{diam}(G) + 1$.

(iii) If $\mathrm{rad}(G) = k$ *and* $\Delta(G) = d \geq 3$*, then G contains at most $\frac{d}{d-2}(d-1)^k$ vertices.*

1.4 Weighted graphs

Additional information can be encoded in a graph. In particular, a weight can be added to each vertex and / or to each edge.

Definition 1.27 (Weighted graph)**.** *A* weighted graph $G = (V, E, \omega)$ *is a graph (V, E) together with two maps $\omega_V : V \to \mathbb{R}$ and $\omega_E : E \to \mathbb{R}$. The notation ω refers to the pair (ω_V, ω_E).*

Note that quite often, one considers these maps with value in $(0, \infty)$ instead of $\mathbb{R}$, and that the index V or E is often drop. We then have $\omega : V \to \mathbb{R}$ and $\omega : E \to \mathbb{R}$, and this does not lead to any confusion. Weighted graphs are very natural and useful in applications. In this framework one has the following definition:

Definition 1.28 (Weighted length)**.** *The* weighted length *of a walk in a weighted graph is given by the sum of the weight on the corresponding edges.*

Clearly, this definition is also valid for trails, paths or cycles, since they are special instances of walks. For a graph without weights, it corresponds to the original notion of length of a walk if one endows the graph with the constant weight 1 on every edge (and on every vertex). Note that in a weighted graph, the shortest path (disregarding weights) between two vertices might not be the one with the smallest weighted length. In

applications, one has therefore to specify which quantity has to be minimized: the unweighted length, or the weighted length ? Of course, it depends on the purpose.

Chapter 2

Representations and structures

In this chapter, we first introduce a few ways to encode the information contained in a graph. Then, we develop the notion of isomorphisms, and list some invariant structures of a graph.

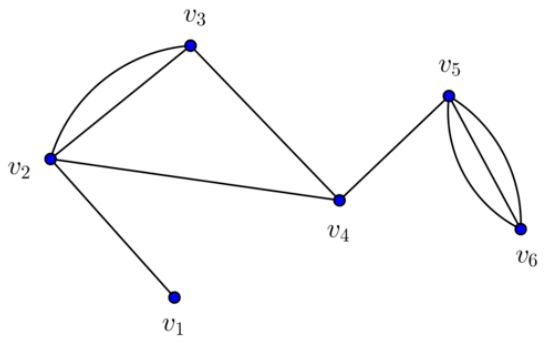
2.1 Matrix representations

Since linear algebra contains a large set of powerful tools, it is rather natural to use this theory for analysing graphs. There exist several ways to represent a graph with matrices. The figures about adjacency matrices are borrowed from [2]. Note that in these figures, vertices are denoted by v_j while in the text they are written x_j.

Definition 2.1 (Adjacency matrix)**.** *Let $G = (V, E)$ be a finite graph, and set $V = \{x_1, \ldots, x_N\}$. The* adjacency matrix A_G of G *is a $N \times N$ matrix with entries*

$$a_{jk} = \#\{e \in E \mid i(e) = (x_j, x_k)\}$$

with the convention that $(x_j, x_k) = (x_k, x_j)$ if the G is undirected, and that a loop satisfying $i(e) = (x_j, x_j)$ is counted twice for an undirected graph, but only once for a directed graph.

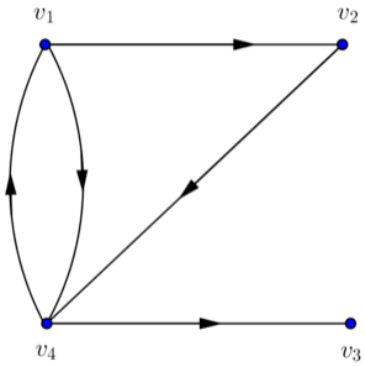


$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 1 & 0 & 0 \\ 0 & 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 3 & 0 \end{pmatrix}$$

Figure 2.1: Adjacency matrix of an undirected graph

It easily follows from this definition that the adjacency matrix of an undirected graph is a symmetric matrix, since $a_{jk} = a_{kj}$, see Figure 2.1. In addition, the relation

$$\deg(x_j) = \sum_k a_{jk} = \sum_k a_{kj}$$



$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 2.2: Non symmetric adjacency matrix

holds for any undirected graph and any $x_j \in V$. Note that these properties are not true in general for a directed graph. However, let us define the *indegree* and the *outdegree* of a vertex of a directed graph:

$$\deg_{\text{in}}(x) = \#\{e \in E \mid i(e) = (y, x) \text{ with y arbitrary}\} \tag{2.1.1}$$

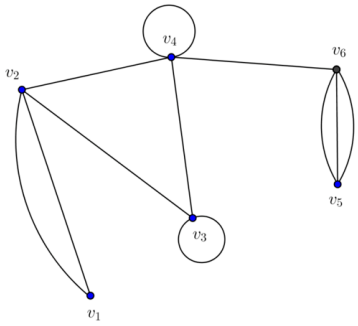
and

$$\deg_{\text{out}}(x) = \#\{e \in E \mid i(e) = (x, y) \text{ with y arbitrary}\}. \tag{2.1.2}$$

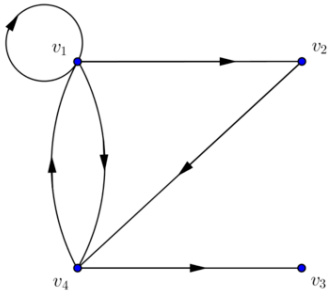
Clearly, one has $\deg_{\text{in}}(x) + \deg_{\text{out}}(x) = \deg(x)$. Then, if G is a directed graph, the following relations hold, see Figure 2.2:

$$\sum_k a_{jk} = \deg_{\text{out}}(x_j) \quad \text{and} \quad \sum_j a_{jk} = \deg_{\text{in}}(x_k).$$

Note also that the convention of counting twice a loop for undirected graph is coherent with the degree 2 attached to a loop in Definition 1.4, see Figure 2.3. However, this choice has also some drawbacks, and the convention is not universal. For example, this convention leads to wrong result in the next statement about the powers of the adjacency matrix. Note that in this statement and in the sequel, we take the convention that $\mathbb{N} := \{1, 2, 3, \dots\}$.



$$\begin{pmatrix} 0 & 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 1 & 3 & 0 \end{pmatrix}$$



$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 2.3: Adjacency matrices in the presence of loops

Proposition 2.2. *Let G be a graph and let A_G be the adjacency matrix (with the convention that a loop provides a contribution 1 on the diagonal also for undirected graphs). For any $r \in \mathbb{N}$ the entry $(A_G^r)_{jk}$ of the r^{th} power of A_G is equal to the number of walks of length r from x_j to x_k.*

Proof. Let A_G be a $N \times N$ adjacency matrix. When $r = 1$, $(A_G^r)_{jk}$ is the number of walks of length 1 from x_j to x_k by the definition of the adjacency matrix. Then, let $r \in N$ be given and suppose $(A_G^r)_{jk}$ is the number of walks of length r from x_j to x_k. Consider $(A_G^{r+1})_{jk}$, and let us compute this entry by using the summation of multiplication of entries of two matrices.

$$(A_G^{r+1})_{jk} = \sum_{l=1}^{N} (A_G^r)_{jl}(A_G^1)_{lk}. \tag{2.1.3}$$

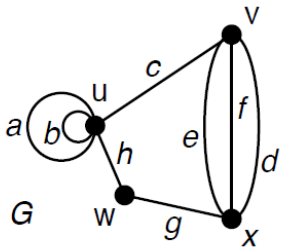
By the assumption and definitions mentioned above, $(A_G^r)_{jl}$ is the number of walks of length r from x_j to x_l, and $(A_G^1)_{lk}$ is the number of walks of length 1 from x_l to x_k. Hence, $(A_G^r)_{jl}(A_G^1)_{lk}$ is the number of walks of length $r+1$ consisting of two walks, one is the walk of length r from x_j to x_l and another one is the walk of length 1 from x_l to x_k. It follows that $\sum_l (A_G^r)_{jl}(A_G^1)_{lk}$ indicates the number of all walks of length $r+1$ from x_j to x_k. Thus, we can regard the l.h.s. of (2.1.3) as the number of walks of length $r+1$ from x_j to x_k. One finishes the proof by an induction argument. □

Let us also mention that adjacency matrices can also be used for checking if two graphs are isomorphic, see the following sections. Indeed, if A_G and $A_{G'}$ correspond to the adjacency matrices of two finite graphs with the same order, then a reordering of the vertices on one graph should lead to two identical adjacency matrices if the graphs are isomorphic. However, this approach is very time and energy consuming, and therefore very inefficient.

We now provide another tool involving matrices. Unfortunately, the definition is not exactly the same for directed or undirected graphs. Also, these definitions depend slightly on the authors, especially for the value associated with a loop.

Definition 2.3 (Incidence matrix of an undirected graph)**.** *Let $G = (V, E)$ be a finite undirected graph with $V = \{x_1, \dots, x_N\}$ and $E = \{e_1, \dots, e_M\}$. The incidence matrix I_G of G consists in the $N \times M$ matrix with entries*

$$i_{j\ell} = \begin{cases} 0 & \text{if } x_j \text{ is not an endpoint of } e_\ell, \\ 1 & \text{if } x_j \text{ is an endpoint of } e_\ell, \\ 2 & \text{if } e_\ell \text{ is a loop at } x_j. \end{cases}$$



$$I_G = \begin{array}{c} \\ u \\ v \\ w \\ x \end{array} \begin{array}{c} \begin{array}{cccccccc} a & b & c & d & e & f & g & h \end{array} \\ \begin{pmatrix} 2 & 2 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Figure 2.4: Incidence matrix of an undirected graph, see Fig. 2.6.4 of [GYA]

The following properties can be easily inferred from this definition:

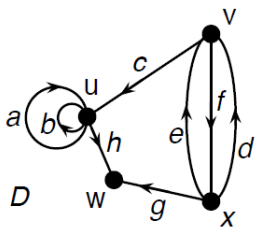
Lemma 2.4. *For the incidence matrix of a finite undirected graph the following relations hold:*

$$\sum_{\ell=1}^{M} i_{j\ell} = \deg(x_j) \quad \text{and} \quad \sum_{j=1}^{N} i_{j\ell} = 2.$$

For a directed graph, the incidence matrix is defined as follows:

Definition 2.5 (Incidence matrix of a directed graph)**.** *Let $G = (V, E)$ be a finite directed graph with $V = \{x_1, \ldots, x_N\}$ and $E = \{e_1, \ldots, e_M\}$. The incidence matrix I_G of G consists in the $N \times M$ matrix with entries*

$$i_{j\ell} = \begin{cases} 0 & \text{if } x_j \text{ is not an endpoint of } e_\ell, \\ 1 & \text{if } x_j \text{ is the target of } e_\ell, \\ -1 & \text{if } x_j \text{ is the origin of } e_\ell, \\ 2 & \text{if } e_\ell \text{ is a loop at } x_j. \end{cases}$$

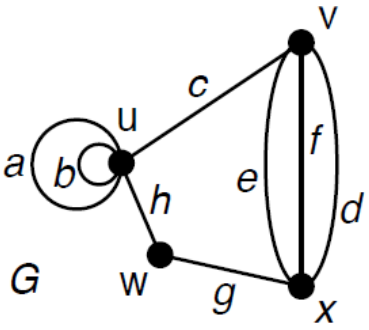


$$I_G = \begin{array}{c} \\ u \\ v \\ w \\ x \end{array} \begin{array}{c} \begin{array}{cccccccc} a & b & c & d & e & f & g & h \end{array} \\ \begin{pmatrix} 2 & 2 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & -1 & -1 & 1 & -1 & 0 \end{pmatrix} \end{array}$$

Figure 2.5: Incidence matrix of an undirected graph, see Fig. 2.6.5 of [GYA]

Note that one of the undesirable features of these matrices is that they contain many zeros. One can be more economical by keeping only the non-zero information but one loses the power of matrices. The incidence tables corresponds to new representations.

Definition 2.6 (Incidence table of an undirected graph)**.** *Let $G = (V, E)$ be a finite undirected graph with $V = \{x_1, \ldots, x_N\}$ and $E = \{e_1, \ldots, e_M\}$. The incidence table $I_{V:E}(G)$ lists, for each vertex x_j, all edges e_ℓ having x_j as one endpoint.*



$$I_{V:E}(G) = \begin{array}{ll} \underline{u}: & a\ b\ c\ h \\ \underline{v}: & c\ d\ e\ f \\ \underline{w}: & g\ h \\ \underline{x}: & d\ e\ f\ g \end{array}$$

Figure 2.6: Incidence table of an undirected graph, see Ex. 2.6.6 of [GYA]

For directed graphs, the tables have to be duplicated.

Definition 2.7 (Incidence tables of a directed graph)**.** *Let $G = (V, E)$ be a finite directed graph with $V = \{x_1, \ldots, x_N\}$ and $E = \{e_1, \ldots, e_M\}$. The incoming incidence table $in_{V:E}(G)$ lists, for each vertex x_j, all edges e_ℓ having x_j as a final point (target), while the outgoing incidence table $out_{V:E}(G)$ lists, for each vertex x_j, all edges e_ℓ having x_j as an initial point (origin).*

$$in_{V:E}\,(\mathsf{G}) = \begin{array}{lccc} \underline{u}: & a & b & c \\ \underline{v}: & d & e & \\ \underline{w}: & g & h & \\ \underline{x}: & f & & \end{array} \qquad out_{V:E}\,(\mathsf{G}) = \begin{array}{lccc} \underline{u}: & a & b & h \\ \underline{v}: & c & f & \\ \underline{w}: & & & \\ \underline{x}: & d & e & g \end{array}$$

Figure 2.7: Incidence tables for the graph of Figure 2.5, see Ex. 2.6.7 of [GYA]

2.2 Isomorphisms

Our general aim is to provide some efficient tools for deciding when two graphs contain the same information, even if they are represented quite differently. What characterizes a graph is its pattern of connections, and the direction on edges for directed graphs, but the way they are represented does not matter. For example, the two graphs of Figure 2.8 correspond to the same graph, even if they do not look similar.

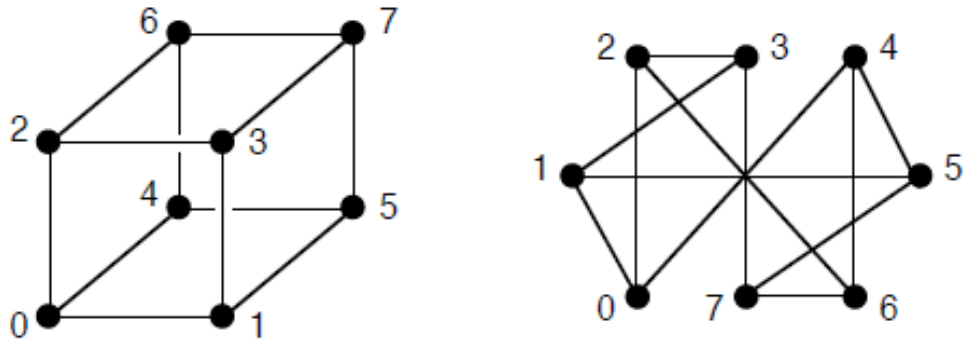


Figure 2.8: Two representations of the same graph, see. Fig. 2.1.1 of [GYA]

We say that these two pictures represent the same graph because any vertex has the same adjacent vertices on both representations. Clearly, if the graph has loop(s) or multiple edges, or if the graph is directed, we would like to have these properties similarly represented in the two pictures. The correct notion encoding all the necessary information is provided in the next definition.

Definition 2.8 (Isomorphism of graph)**.** *Let $G = (V, E)$ and $G' = (V', E')$ be two graphs, with internal map denoted respectively by i and by i'. A map $f : G \to G'$ is* an isomorphism of graphs *if $f = (f_V, f_E)$ with $f_V : V \to V'$ and $f_E : E \to E'$ satisfy*

(i) f_V and f_E are bijections,

(ii) For any $e \in E$ with $i(e) = (x, y)$ in $V \times V$, one has $i'(f_E(e)) = (f_V(x), f_V(y))$ in $V' \times V'$.

Whenever such an isomorphism exists, we say that G and G' are isomorphic, and write $G \cong G'$.

Note that this definition holds for the general definition of a graph provided in Definition 1.1, see Figures 2.9 and 2.10. Once again, if the graph is undirected, the pairs (x, y) and (y, x) are identified in $V \times V$, and the same for the pairs $(f_V(x), f_V(y))$ and $(f_V(y), f_V(x))$ in $V' \times V'$, but this property does not hold for directed graphs. In the special case of simple graphs, as presented in Remark 1.2, the above definition can be slightly simplified since an edge is uniquely defined by its endpoints, see Figure 2.11. Observe finally that another way to present

Figure 2.9: Isomorphism of a graph with loops and multiple edges

the second condition of Definition 2.8 is to say the following diagram is commutative:

$$\begin{array}{ccc} E & \xrightarrow{\ i\ } & V \times V \\ \downarrow{\scriptstyle f_E} & & \downarrow{\scriptstyle f_V \times f_V} \\ E' & \xrightarrow{\ i'\ } & V' \times V' \ . \end{array}$$

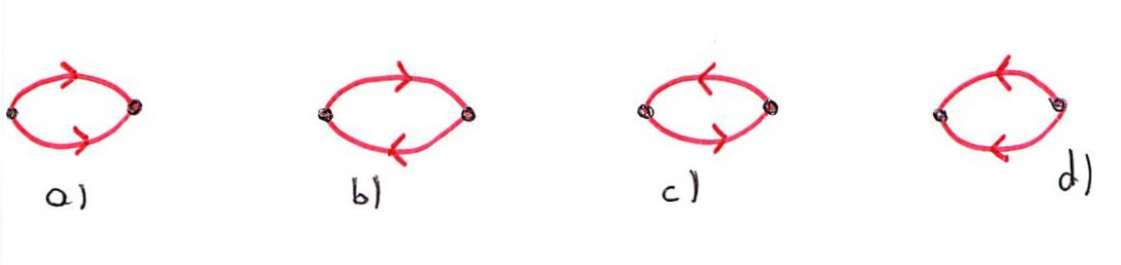


Figure 2.10: a) and d) are isomorphic, b) and c) are isomorphic

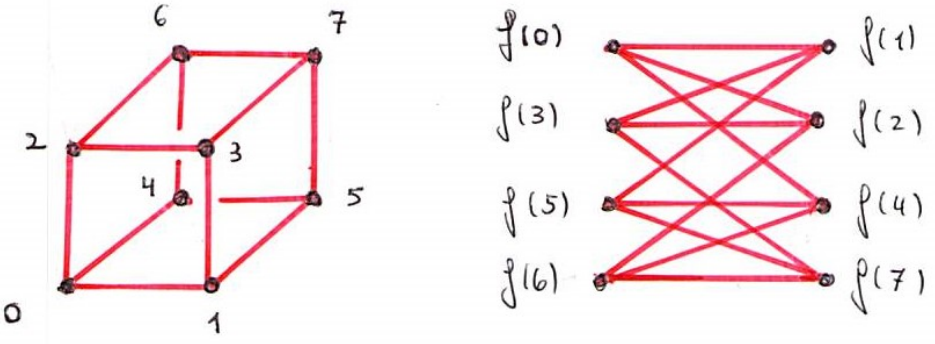


Figure 2.11: Isomorphism of a simple graph

Remark 2.9. *One observes that the notion of isomorphisms is an* equivalence relation. *Indeed, $G \cong G$ (reflexive property) by considering the identify map for the graph isomorphism; if $G \cong G'$, then $G' \cong G$ (symmetric property) because f^{-1} also defines an isomorphism of graph; if $G \cong G'$ (through a map f) and $G' \cong G''$ (through a map f'), then $G \cong G''$ (transitive property) because the composition of maps $f' \circ f$ also defines an isomorphism of graph, as it can be easily checked.*

Deciding when two graphs are isomorphic is a hard and famous problem, the so-called *graph-isomorphism problem*. Except for very small graphs, it is very time consuming. However, by looking at specific quantities, one can often easily show that two graphs are not isomorphic. Such quantities are presented in the next definition.

Definition 2.10 (Graph invariant)**.** *A* graph invariant *is a property of a graph which is preserved by isomorphisms.*

In other terms, such a quantity is the same in any representation of a graph. Thus, if this quantity is not the same in two graphs, one can directly say that these two graphs are not isomorphic. Let us list a few quantities which are clearly graph invariants, additional examples will appear in this chapter. We recall that the notation $N(x)$ for the set of neighbours of x has been introduced in Definition 1.4.

Proposition 2.11 (Graph invariants 1)**.** *Let $G = (V, E)$ be a graph. The following quantities are graph invariants:*

(i) The order and the size (see Definition 1.3), with the convention that these quantities can take the value ∞,

(ii) The set of degrees (see Definition 1.4), namely $\{\deg(x) \mid x \in V\}$*,*

(iii) The set of degrees of neighbours, namely

$$\{\{\deg(y) \mid y \in N(x)\} \text{ for any } x \in V\},$$

(iv) The set of lengths of walks, of trails, or of paths in G, see Definitions 1.10 and 1.12,

(v) The diameter, the radius and the girth (see Definitions 1.15 and 1.20).

Let us observe that for directed graphs, a refined version of (*ii*) and (*iii*) exists. We recall that the notion of indegree and outdegree have been introduced in (2.1.1) and (2.1.2), respectively. Then, the following quantities are directed graphs invariants:

(ii') The set if indegrees and outdegrees, namely $\{\deg_{\text{in}}(x) \mid x \in V\}$*, and* $\{\deg_{\text{out}}(x) \mid x \in V\}$*,*

(iii') The indegrees and outdegrees of neighbours, namely

$$\{\{\deg_{\text{in}}(y) \mid y \in N(x)\} \text{ for any } x \in V\},$$

and

$$\{\{\deg_{\text{out}}(y) \mid y \in N(x)\} \text{ for any } x \in V\}.$$

Note that these invariants could be even further refined by considering separately the set of neighbours of x which are connected by an edge e satisfying either $o(e) = x$ or $t(e) = x$. We leave the definition of these invariants to the interested reader.

2.3 Automorphisms and symmetries

Identifying the symmetries of a graph is often useful, even if it is not an easy task. Clearly, symmetries should not depend on the representation but should again be an intrinsic property. The following definition contains the necessary notion for dealing with symmetries of a graph.

Definition 2.12 (Automorphism)**.** *Let G be a graph. An isomorpism from G to G is called an* automorphism.

Clearly, any graph possesses an automorphism, the identity map. In addition, by the properties of the equivalence relation mentioned in Remark 2.9, one observes that the set of automorphisms of a graph is in fact a group: the composition of automorphisms is associative, and every automorphism has an inverse (it corresponds to the map f^{-1} mentioned in Remark 2.9). One speaks about the *automorphisms group* of a graph. The main idea now is to look at the size of this group. If this group is big, then the graph has several symmetries, while if the group contains only the identity element, then the graph has no symmetry at all. Observe that for simple undirected graphs, automorphisms can be completely described by permutations of the set of vertices.

Figure 2.12 contains three representations of the same simple graph, called the Petersen graph. At first glance, it is not easy to see that these three graphs are isomorphic, but this can be checked by looking at the edges connected at any vertex. Then, what about automorphisms ? It is clear on the picture (a) that any rotation by $2\pi k/5$ with $k \in \{0, 1, 2, 3, 4\}$ defines an automorphism. A reflection symmetry by a vertical axis is also clear on figure (a). On figures (b) and (c) a reflection symmetry by a vertical axis is also clear, but these three reflection symmetries do not correspond to the same automorphisms of G.

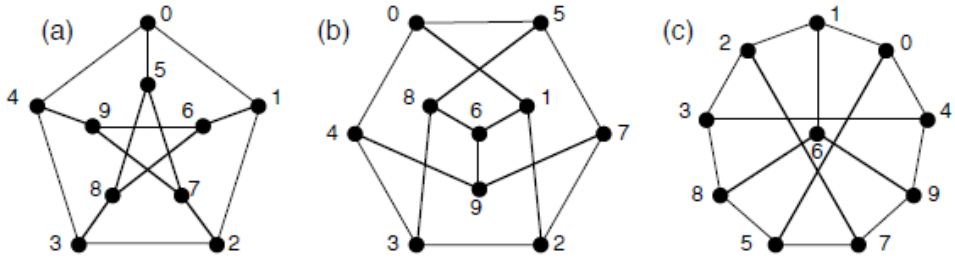


Figure 2.12: Three representations of the Petersen graph, see. Fig. 2.2.3 of [GYA]

Let us try to describe these automorphisms by using a convenient notation. More information on the permutation group can be found in the Appendix A.4 of [GYA] or in Wikipedia [1]. One way to describe the rotation by $2\pi/5$ of figure (a) is to write

$$(0\ 1\ 2\ 3\ 4)(5\ 6\ 7\ 8\ 9)$$

describing the action of the automorphism: $0 \to 1$, $1 \to 2$, $2 \to 3$, $3 \to 4$, $4 \to 0$ and $5 \to 6$, $6 \to 7$, $7 \to 8$, $8 \to 9$, $9 \to 5$. In a similar way, the three mentioned reflection symmetries can be described by $(1\ 4)(2\ 3)(6\ 9)(7\ 8)(0)(5)$, $(0\ 5)(1\ 8)(2\ 3)(4\ 7)(6)(9)$ and $(0\ 2)(3\ 4)(5\ 7)(8\ 9)(1)(6)$.

Whenever a group acts on an object, a useful concept is the one of orbit. Here, we keep in mind the action of the automorphism group acting on a graph, but the definition is more general. Note that since G is already used for a graph, we use the notation H for the group in the next definition.

Definition 2.13 (Orbit)**.** *Let H be a group acting on a set X, with an action denoted by $h(x) \in X$ for $x \in X$ and $h \in H$. For any $x \in X$ the* orbit of x *is the set $\{h(x) \mid h \in H\}$ and is denoted by* $\mathrm{Orb}(x)$.

In other words, $\mathrm{Orb}(x)$ corresponds to all points taken by x when a group H acts on this point. It is easily observed that for any $x, y \in X$ one has either $\mathrm{Orb}(x) = \mathrm{Orb}(y)$ or $\mathrm{Orb}(x) \cap \mathrm{Orb}(y) = \emptyset$, and no other alternative. Let us consider the graph given in Figure 2.13. This graph has again several automorphisms obtained by reflection symmetries by a vertical axis, a horizontal axis, but also the one obtained by the combination of these two automorphisms. If we list them with the notation introduced above one gets

$$(1)(2)(3)(4)(5)(6)(7)(8)$$

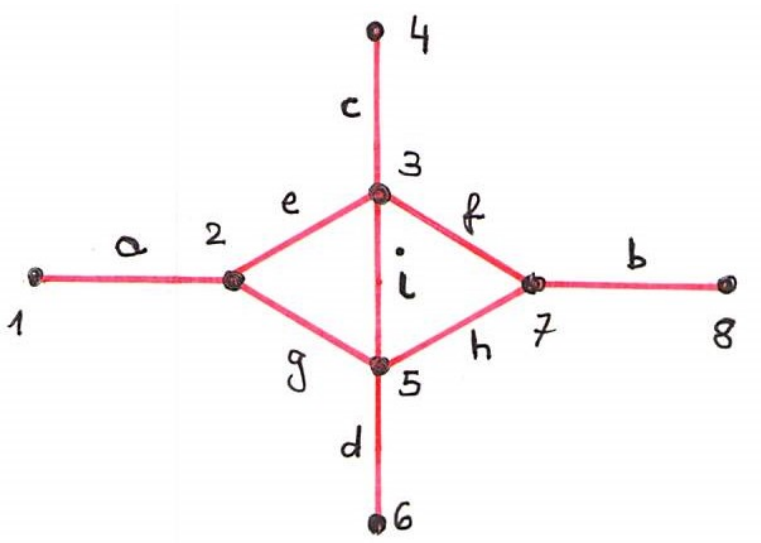


Figure 2.13: A graph with several symmetries

$$(1\ 8)(2\ 7)(3)(4)(5)(6)$$
$$(1)(2)(3\ 5)(4\ 6)(7)(8)$$
$$(1\ 8)(2\ 7)(3\ 5)(4\ 6).$$

Let us now describe the orbits of the vertices and of the edges under the group generated by these four automorphisms. For this example, the *vertex orbits* are

$$\mathrm{Orb}(1) = \mathrm{Orb}(8) = \{1, 8\}, \quad \mathrm{Orb}(2) = \mathrm{Orb}(7) = \{2, 7\},$$
$$\mathrm{Orb}(4) = \mathrm{Orb}(6) = \{4, 6\}, \quad \mathrm{Orb}(3) = \mathrm{Orb}(5) = \{3, 5\},$$

while the *edge orbits* are

$$\mathrm{Orb}(a) = \mathrm{Orb}(b) = \{a, b\}, \quad \mathrm{Orb}(c) = \mathrm{Orb}(d) = \{c, d\},$$
$$\mathrm{Orb}(e) = \mathrm{Orb}(f) = \mathrm{Orb}(g) = \mathrm{Orb}(h) = \{e, f, g, h\}, \quad \mathrm{Orb}(i) = \{i\}.$$

Observe that these notions apply to directed graphs as well, but the orientation is one more ingredient to take into account. For example, the graph represented in Figure 2.14 has a group of automorphism reduced to the identity only.

Figure 2.14: No symmetry

Let us still state some easy properties of elements on orbits. These properties can be deduced from Proposition 2.11.

Lemma 2.14.

(i) All vertices in one orbit have the same degree (and the same indegree and outdegree for directed graphs),

(ii) All edges in one orbit have the same pair of degrees at their endpoints (and the same indegrees and outdegrees for directed graphs).

Let us check what are the vertex orbits and the edge orbit of Figure 2.12 ? Quite surprisingly, this graph has only one vertex orbit (containing all vertices) and one edge orbit (containing all edges). It means that given two vertices x and y, there exists one automorphism sending x on y, and a similar observation holds for any pair of edges. In such a case, we speak about a *vertex transitive graph* and a *edge transitive graph*.

2.4 Subgraphs

Some properties of a graph can be determined by the existence of some subgraphs inside it.

One example is Theorem 1.21 about undirected bipartite graphs and the existence of cycles of odd length. In this section we gather several notions related to subgraphs, not all these notions are related to each others.

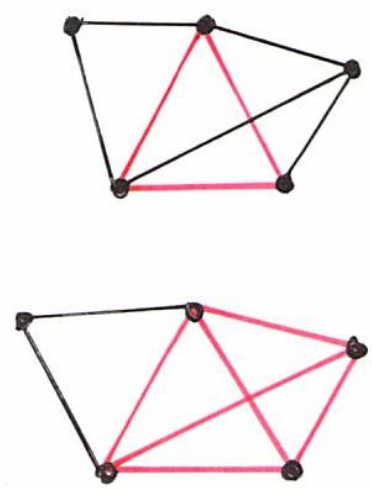


Figure 2.15: 1 clique, 1 maximal clique

Definition 2.15 (Clique)**.** *Let $G = (V, E)$ be an undirected graph and consider a subset $S \subset V$. This set S is called* a clique *if for any $x, y \in S$ with $x \neq y$ there exists $e \in E$ with $i(e) = (x, y)$.*

Note that the first part of the definition means that every two distinct vertices in S are adjacent. One speaks about a *a maximal clique S* if there is no clique S' with $S \subset S' \subset V$, see Figure 2.15. Observe also that this requirement is a maximality condition, and that some authors include this requirement in the definition of a clique. The notion of clique is interesting for the next definition.

Definition 2.16 (Clique number)**.** *The* clique number *$w(G)$ of a graph G corresponds to the number of vertices of a largest clique in G.*

Note that there might be several cliques containing $w(G)$ vertices. Thus, there is no uniqueness for the "largest" clique, but the clique number is uniquely defined. In a vague sense, this clique number gives the maximal number of vertices which are tightly connected to each others, see Figure 2.16. Two concepts complementary to the notions of clique and clique number are:

Definition 2.17 (Independent set and independence number)**.** *Let $G = (V, E)$ be an undirected graph and consider a subset $S \subset V$. This set S is called* independent *if no pair of vertices in S is connected by any edge in G. The* independence number *$\alpha(G)$ of a graph G corresponds to the number of vertices of a largest independent set in G.*

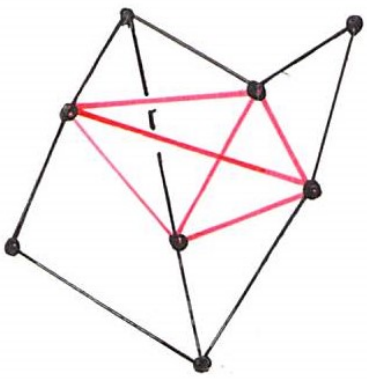


Figure 2.16: $w(G) = 4$

As before, there is no uniqueness for the largest independent set in G, but the independence number is uniquely defined, see Figure 2.17 for example. Note that these last notions extend directly to directed graphs. However, for the notion of a clique, it is not so clear what would be the most useful extension ? Should we use the notion of a clique in the underlying undirected graphs (when orientation is suppressed), or should we look for pair of edges connected by directed edges in both directions ? The choice of the most suitable notion would certainly depend on the applications.

A somewhat related (but more global) notion is provided in the next definition.

Definition 2.18 (Component)**.** *A* component *of a graph G is a maximal connected subgraph of G.*

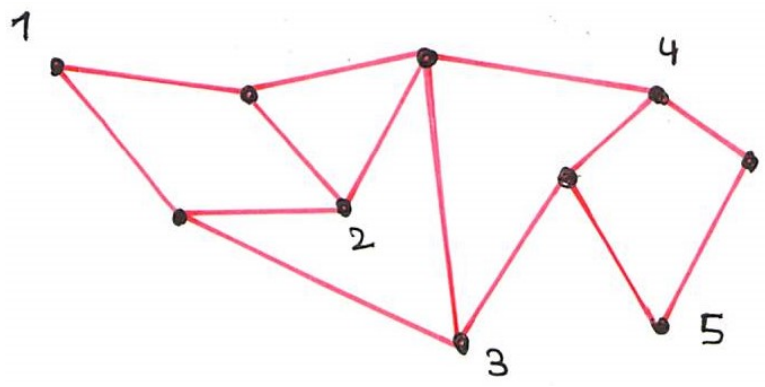


Figure 2.17: Independence number: $\alpha(G) = 5$

Figure 2.18: A graph with 4 components

In other words, a connected subgraph G' is a component of G if G' is not a proper subgraph of any connected subgraph of G. Here, proper simply means *different*. It thus follows that any graph is made of the disjoint union of its components. The number of components of G will be denoted by $c(G)$, see Figure 2.18.

Since orientation does not play any role in the definition of connected graphs, it also does not play any role in the definition of a component. Note that an alternative definition could be provided in terms of paths: For any pair of vertices in one component there exists a path (with the direction on the edges suppressed) having these vertices as endpoints, and the edges for all these possible paths belong to same component of the graph.

Recall that the suppression of a vertex or an edge from a graph has been introduced in Section1.1. Together with the notion of component, we can now select some vertices or edges which are more important than others. More precisely, the following definitions identify the most vulnerable parts of a graph, see also Figures 2.19 and 2.20.

Definition 2.19 (Vertex-cut and cut-vertex)**.** *Let $G = (V, E)$ be a graph.*

(i) A vertex-cut *is a set of vertices $U \subset V$ such that $G - U$ has at least one more component than G.*

(ii) A vertex $x \in V$ is called a cut-vertex *or a* cutpoint *if $\{x\}$ is a vertex-cut.*

Definition 2.20 (Edge-cut and cut-edge)**.** *Let $G = (V, E)$ be a graph.*

(i) An edge-cut *is a set of edges $F \subset E$ such that $G - F$ has at least one more component than G.*

(ii) An edge $e \in E$ is called a cut-edge *or a* bridge *if $\{e\}$ is an edge-cut.*

These notions will be used again when graph's connectivity will be discussed. For the time being, let us simply complement the content of Proposition 2.11 with a few more graph invariants.

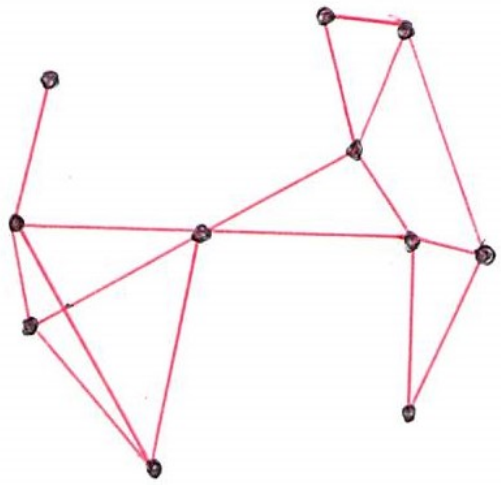


Figure 2.19: Graph with two cut-vertices

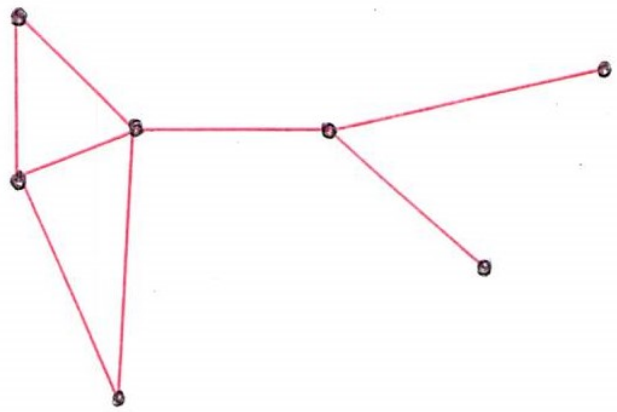


Figure 2.20: Graph with three cut-edges

Proposition 2.21 (Graph invariants 2)**.** *Let $G = (V, E)$ be a graph. The following quantities are graph invariants:*

(i) For undirected graphs, the clique number and the independence number, namely $w(G)$ and $\alpha(G)$,

(ii) The number of components, namely $c(G)$,

(iii) The number of distinct cutpoints or bridges.

Chapter 3

Trees

Trees play a central role in graph theory, and are at the root of many algorithms. We first present the theoretical part, and subsequently describe several applications.

3.1 Trees and forests

We first provide the definition of a tree in the general setting that we have introduced so far. Usually, trees and directed trees are treated separately.

Definition 3.1 (Tree)**.** *A* tree *is a connected graph whose underlying undirected graph has no cycle.*

Note first that the acyclicity condition prevents any tree to have a loop or any multiple edges. For that reason, trees are always simple graphs, as defined in Remark 1.2.

For undirected graphs the above definition reduces to a connected and acyclic graph. For directed graphs, this definition does not see the orientation on the edges, and is probably not the most interesting definition. Indeed, the notion of connected graph is based on the underlying graph, and the acyclicity property is also imposed on the underlying undirected graph. For directed graphs, a more interesting notion is the one of acyclic digraph, see Figure 3.1. These oriented graphs have no cycle, but the underlying unoriented graph can have cycles. Acyclic digraphs have also several applications, see [3]. For simplicity, we shall simply say that a directed graph has no *undirected cycle* whenever the underlying undirected graph has no cycle. Directed graphs which are trees are also called *oriented trees*, *polytrees*, or *singly connected network*. In the sequel, whenever we want to emphasize that the tree considered is also an oriented graph, we shall call it *an oriented tree*, and accordingly an *unoriented tree* will be a tree without orientation, see Figure 3.2.

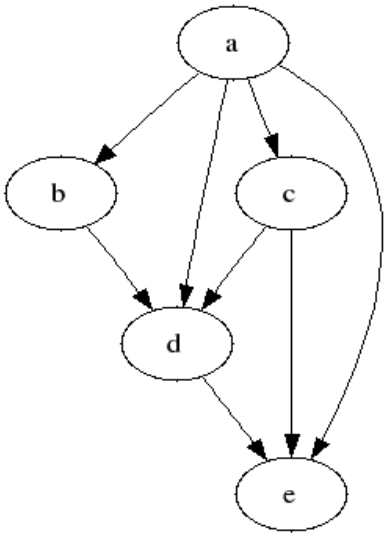


Figure 3.1: An acyclic digraph

Recall that a leaf is a vertex of degree 1. It is not difficult to observe (and prove) that any finite tree containing at least one edge has also at least two leaves. In other words, a non-trivial tree must have at least two leaves ☺. Also, if a tree is made

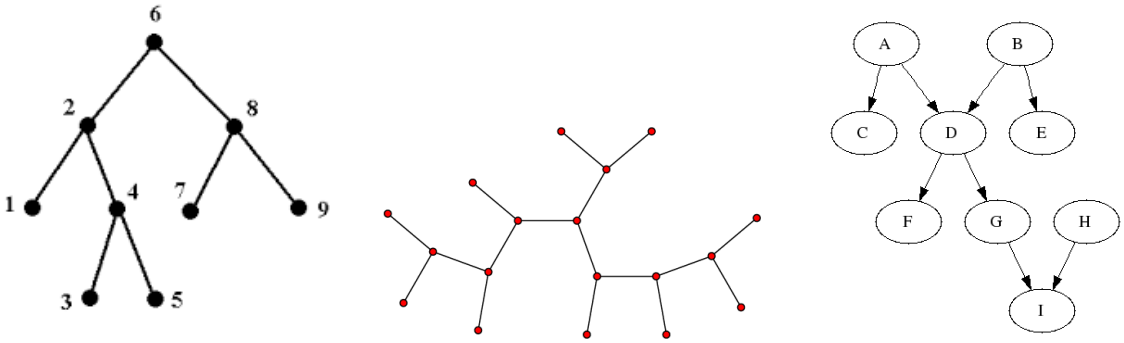


Figure 3.2: Three trees: two unoriented, one oriented

of n vertices, it contains exactly $n-1$ edges. Note that a graph with no undirected cycle is called *a forest*, see Figure 3.3, and that such a forest is made of the disjoint union of trees, each of them defining a component of the graph, see Definition 2.18. Some authors use the terms *polyforest* or *oriented forest* for the disjoint union of oriented trees.

We now provide some equivalent definitions of a tree. The proof is provided in [GYA, Thm. 3.1.8] for undirected graphs.

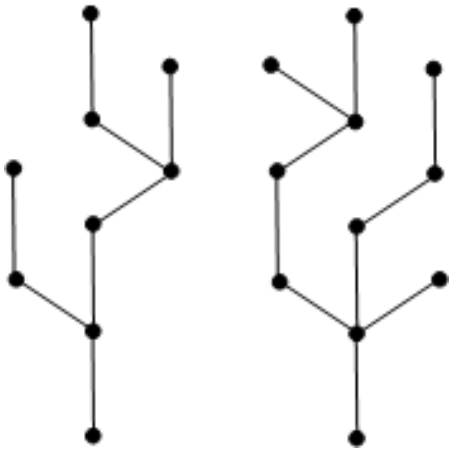


Figure 3.3: One forest

Proposition 3.2. *Let G be a graph with n vertices. The following statements are equivalent:*

(i) *G is a tree,*

(ii) *G contains no undirected cycle and has $n-1$ edges,*

(iii) *G is connected and has $n-1$ edges,*

(iv) *G is connected and every edge is a cut-edge, see Definition 2.20,*

(v) *Any two vertices of G are connected by exactly one unoriented path (when the orientation on the edges is disregarded),*

(vi) *G contains no undirected cycle, and the addition of any new edge e on the graph generates a graph with exactly one undirected cycle.*

Recall that the notion of a central vertex has been introduced in Definition 1.16. Such a vertex has the property of being at a minimum distance to all other vertices, and therefore is located at a "strategic position". This position is usually not unique, and examples with several central vertices are easy to construct. For trees, the situation is completely different. In fact, the following statement has already been proved in 1869, but note that it applies only to unoriented graphs.

Theorem 3.3. *For any finite unoriented tree, there exists only one or two central vertices.*

The proof is not difficult but relies on several lemmas, see pages 125 and 126 of [GYA]. Let us just emphasize the main idea: If x is a central vertex in an unoriented tree, then x is still a central vertex in the induced tree obtained by removing all leaves. By the process of removing leaves iteratively, one finally ends up with a tree consisting either of one single vertex, or of two vertices connected by an edge. This unique vertex or the two vertices correspond to the central vertices of the initial unoriented tree.

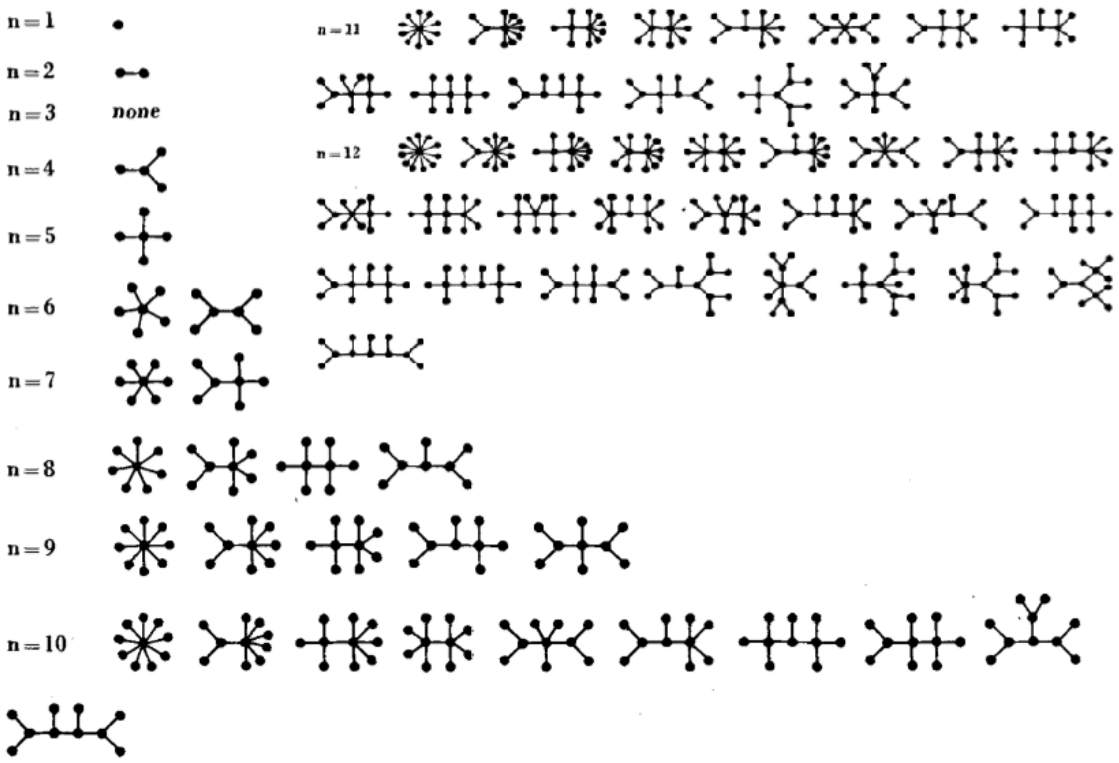


Figure 3.4: Irreducible trees with less than 12 vertices, see [4]

Note that this almost unicity of the central vertex of a tree can be used for the definition of the *root* of a tree. Before introducing rooted trees, and for fun, let us introduce one more notion:

Definition 3.4 (Irreducible tree)**.** *An* irreducible tree*, or* series-reduced tree *is an unoriented tree in which there is no vertex of degree* 2*.*

Note that there exists a classification of such trees, modulo isomorphisms. The table of the ones with less that 12 is provided in Figure 3.4.

3.2 Rooted trees

In a tree, it is sometimes important to single out one vertex. This idea is contained in the next definition.

Definition 3.5 (Rooted tree)**.** *A* rooted tree *is tree with a designated vertex called* the root*.*

On drawings, the root of a rooted tree is often put at a special place (top, bottom, left or right of the picture), see figure 3.5. Note that in this definition, the choice of the root is arbitrary. However, in applications there often exists a natural choice for the root, based on some specific properties of this vertex. We mention a few examples in the next definition, but other situations can take place.

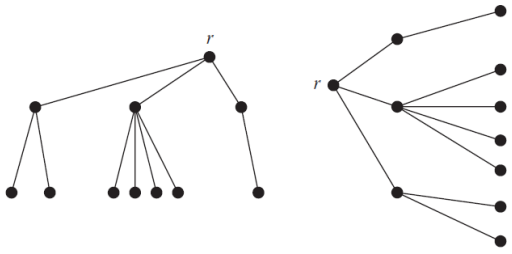


Figure 3.5: Two trees with root r

Definition 3.6 (Shortest path tree, arborescence and anti-arborescence)**.**

(i) *A* shortest path tree *is an unoriented rooted tree for which the root is the unique central vertex,*

(ii) *An* arborescence *or* out-tree *is a rooted oriented tree with all edges pointing away from the root, and an* anti-arborescence *or* in-tree *is a rooted oriented tree with all edges pointing towards the root,*

Let us illustrate these definitions: In Figure 3.6, the first tree is a rooted unoriented tree without any special property, the second tree corresponds to a shortest path tree, while the third tree is an arborescence. Note that underlying graphs for the first and the second tree are the same, only the choice of a specific vertex as a root makes them look different. As a consequence, these two trees are isomorphic as graphs, but not as rooted trees (for which the two roots should be in correspondence).

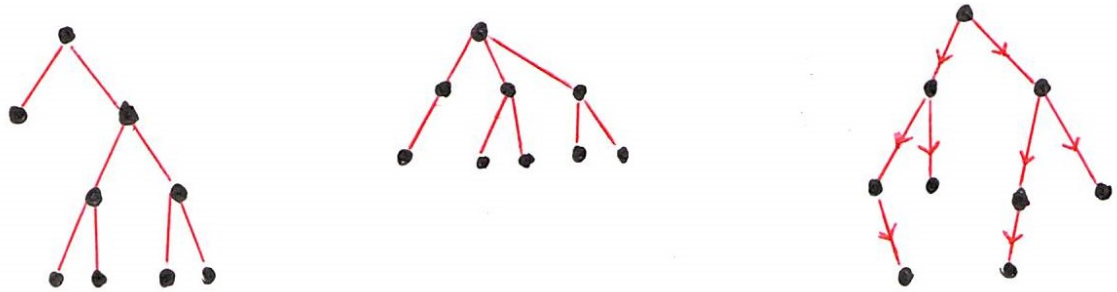


Figure 3.6: Three trees

Let us now introduce some names related to vertices.

Definition 3.7. *Let G be an unoriented rooted tree, or an arborescence, with root denoted by r.*

(i) *The* height*, or the* depth*, or the* level *of a vertex x corresponds to the distance* $\mathrm{d}(r, x)$*,*

(ii) *The* height of the tree *is the greatest level, or equivalently the length of the longest path with one endpoint at r,*

(iii) *The* ancestors *or* ascendants *of a vertex x is the set of all vertices contained in the path from r to x, while the* descendants *of x is the set of all y having x as an ancestor. One speaks about* proper ancestors *of x and* proper descendants *of x when x is not included in these sets,*

(iv) *The* parent *of a vertex x is the ancestor y satisfying* $\mathrm{d}(r, y) = \mathrm{d}(r, x) - 1$*, and a child of x is a descendant y satisfying* $\mathrm{d}(r, y) = \mathrm{d}(r, x) + 1$*, with the convention that the root has no parent, and a leaf has no child,*

(v) *Two vertices having the same parent are called* siblings*,*

(vi) *An* internal vertex *of a tree is a vertex which possesses at least one child.*

It is easily observed that a vertex x has only one parent but is allowed to have several children. Let us also mention that these notions can also be applied to anti-arborescence, if the distance $\mathrm{d}(r, x)$ is replaced by $\mathrm{d}(x, r)$, and the directions of paths are reversed. In the sequel we shall usually not mention anti-arborescences, but keep in mind that any information on arborescences can be adapted to anti-arborescences. On the other hand, one observes that the notions introduced above do not really fit with arbitrary oriented rooted trees, since given an arbitrary vertex x, the distances $\mathrm{d}(r, x)$ and $\mathrm{d}(x, r)$ could be infinite.

Let us now discuss the regularity of trees.

Definition 3.8 (p-ary tree, complete p-ary tree)**.** *Let p be a natural number.*

(i) *A* p-ary tree *is an unoriented rooted tree or an arborescence, in which every vertex has at most p children, and at least one of them possesses p children,*

(ii) *A* complete p-ary tree *is an unoriented rooted tree or an arborescence in which every internal vertex has p children, and each leave of the tree has the same depth.*

Figure 3.7: A 3-ary tree and a complete 2-ary tree

Another useful notion can be defined for the rooted trees considered so far. Note however that it is an additional structure which is added repeatedly to the children of each vertex.

Definition 3.9 (ordered tree)**.** *An* ordered tree *is an unoriented rooted tree or an arborescence in which the children of each vertex are assigned with a fixed ordering.*

On drawing, the ordering is often represented by the respective position of the children of any given vertex. The primary example of an ordered tree is the *binary tree*, a 2-ary tree with possibly a *left child* and a *right child* for each vertex. Another example is the *ternary tree*, a 3-ary tree with children distinguished into *left child*, *mid child* and *right child*.

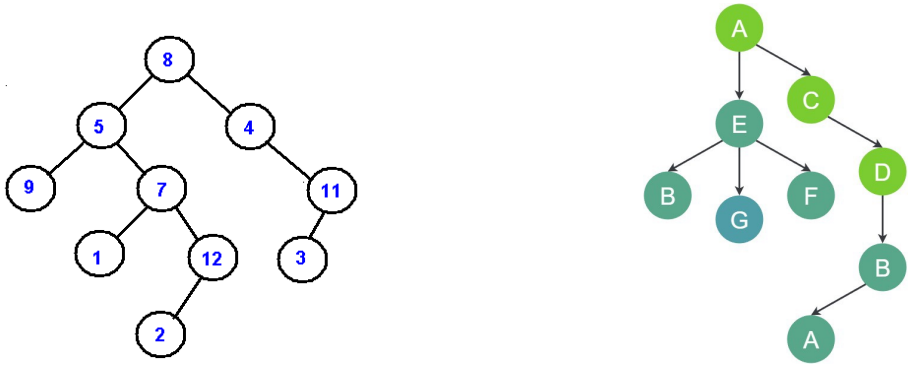


Figure 3.8: One binary tree and one ternary tree

Let us now list a few applications of rooted trees, more will be presented in the following sections. The forthcoming pictures are all borrowed from [GYA, Sec. 3.2].

Example 3.10 (Decision tree)**.** *A* decision tree *is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It often lists all possible sequences, and provides a final weight (for example probability or cost) to each path in the tree, see Figure 3.9 and [5].*

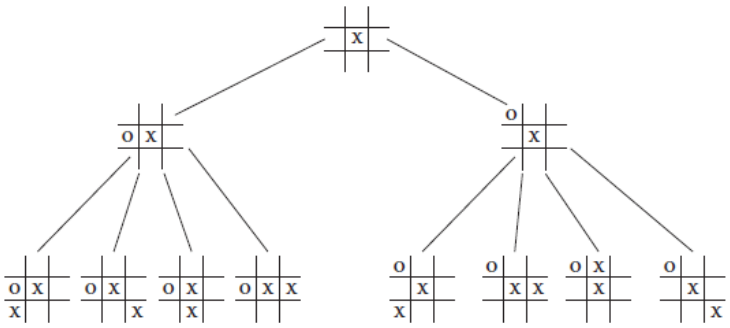


Figure 3.9: The first three moves of tic-tac-toe

Example 3.11 (Tree data structure)**.** *Trees are widely used whenever data contains a hierarchical structure, see Figure 3.10. The notion of parent and children can then be used efficiently.*

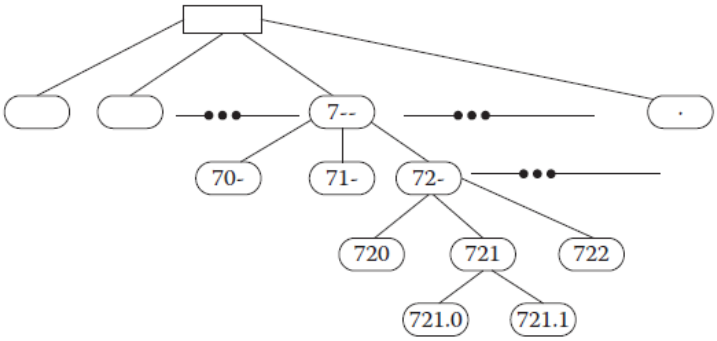


Figure 3.10: Classification in libraries is often based on a tree data structure

Example 3.12 (Sentence parsing)**.** *Rooted trees can be used to parse a sentence in any language, see Figure 3.11. For such an application, a predefined structure of the tree is applied to a sentence.*

Let us finally mention one application of the notion of shortest path tree provided in Definition 3.6. Given a connected unoriented graph and choosing one vertex x, it is always possible to realize one shortest path tree with root x. Note that this tree is usually not unique, but allows us to get a good representation of the distance between x and any other vertex of the graph.

3.3 Traversals in binary trees

In this section we introduce a basic tool for encoding or decoding some information stored in binary trees, see the left picture in Figure 3.8. Most of the constructions apply to more general ordered p-ary trees as well. Note that this section and the following ones are very much oriented towards computer science.

Definition 3.13 (Graph traversal)**.** A graph traversal *or* a graph search *is the process of visiting systematically each vertex in a graph.*

Here "visiting" means either collect the data, or compare or perform the data, or update the data stored at a vertex. Since each vertex are visited successively, a graph traversal also corresponds to endowing the vertices of a graph with a global ordering. For a general graph, a traversal can be almost arbitrary, but for trees (and in particular for binary trees) some traversals are rather natural. Note that we consider planar trees in the sense

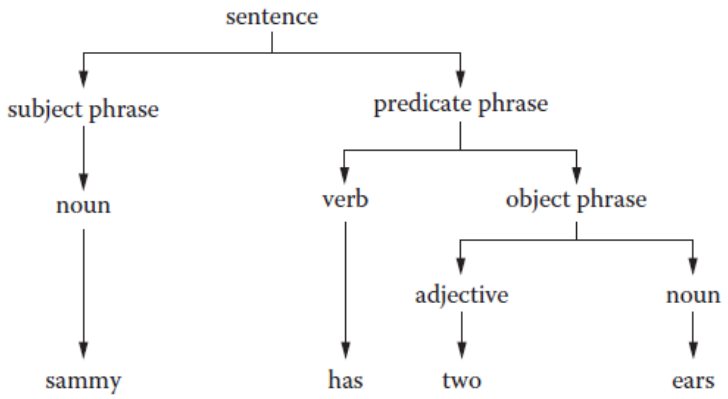


Figure 3.11: Parsing a sentence

that the order on the tree is indexed by left and right. As a consequence, given a vertex x of the graph, its left subtree $L(x)$ and its right subtree $R(x)$ are clearly defined, see Figure 3.12.

The general recursive pattern for traversing a binary tree is this: Go down one level to the vertex x. If x exists (is non-empty) execute the following three operations in a certain order: (L) Recursively traverse x's left subtree, (R) Recursively traverse x's right subtree, (N) Process the current node x itself. Return by going up one level and arrive at the parent of x. The following examples are the most used traversals:

Figure 3.12: Left subtree $L(x)$ and right subtree $R(x)$

Definition 3.14 (Traversal of binary trees)**.** *Let G be a binary tree, with the root represented at the top.*

(i) The level-order traversal *consists in enumerating the vertices in the top-to-bottom, left-to-right order,*

(ii) The pre-order traversal *or* NLR *is defined recursively by 1) process the root, 2) perform the pre-order traversal of the left subtree, 3) perform the pre-order traversal of the right subtree,*

(iii) The in-order traversal *or* LNR *is defined recursively by 1) perform the in-order traversal of the left subtree, 2) process the root, 3) perform the in-order traversal of the right subtree,*

(iv) The post-order traversal *or* LRN *is defined recursively by 1) perform the post-order traversal of the left subtree, 2) perform the post-order traversal of the right subtree, 3) process the root .*

Let us illustrate these traversals with Figure 3.13 from [6].

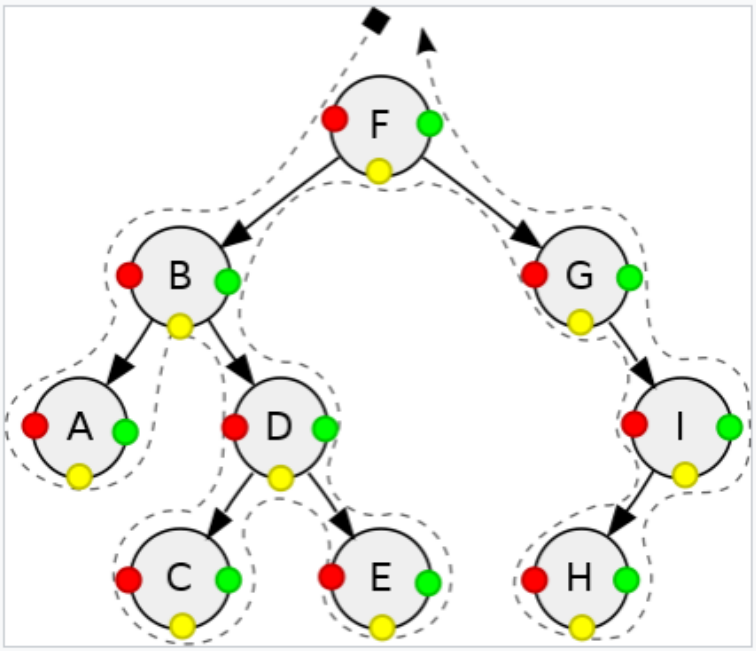


Figure 3.13: Traversals of a binary tree

(i) Level-order traversal:
F, B, G, A, D, I, C, E, H,

(ii) Pre-order traversal (NLR) in red:
F, B, A, D, C, E, G, I, H,

(iii) In-order traversal (LNR) in yellow:
A, B, C, D, E, F, G, H, I,

(iv) Post-order traversal (LRN) in green:
A, C, E, D, B, H, I, G, F.

In the next section we gather several applications of binary trees. Traversals will also play a role for reading a tree.

3.4 Applications

In applications, the letter T is often used for trees instead of the letter G which was more natural for general graph. In this section, we follow this general trend and use the letter T instead of G.

3.4.1 Arithmetic expression trees

Let us first look at an application of the in-order traversal for arithmetic expressions. A similar application holds for boolean expressions.

Definition 3.15 (Arithmetic expression tree)**.** *An* arithmetic expression tree *is a binary tree, with arithmetic expressions (operators) at each internal vertex, and constants or variables at each leaf.*

Clearly, such a tree can be read by the different traversals introduced in Definition 3.14. In this setting, the most natural traversal is the in-order traversal. However, when printing the expression contained in such a tree, opening and closing parentheses must be added at the beginning and ending of each expression. The interest of these trees is precisely that they take care of the ordering of the operations. As every subtree represents a sub-expression, an opening parenthesis is printed at its start and the closing parenthesis is printed after processing all of its children. For example, the arithmetic expression tree represented in Figure 3.14 corresponds to the expression $((5 + z)/ - 8) * (4^2)$, see also [7]. Note that following the in-order traversal, one should write $8-$ and not -8. However, the sign $-$ is in fact slightly misleading since it represents here the operation "take the opposite". When applied to the number 8, the outcome is indeed -8. The operation "take the opposite" or "take the inverse" are called *unary operators* because they require only one child, and not two children as most of the other operations.

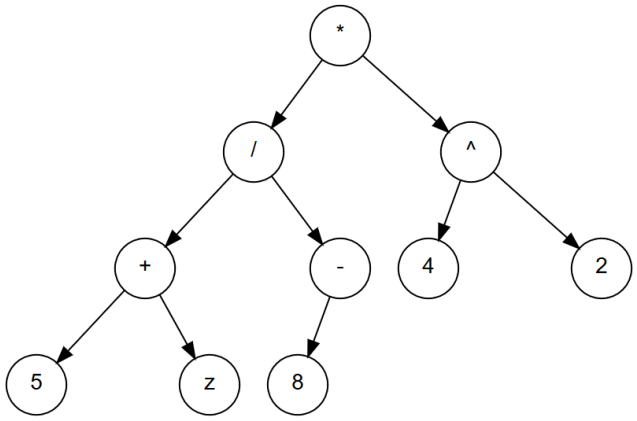


Figure 3.14: An arithmetic expression tree

3.4.2 Binary search trees

We now introduce an application of binary trees for the efficient search of data. In the next definition, we consider *a totally ordered set* $(S, \leq)$, which means a set S with a binary operation $\leq$ satisfying the following three conditions for any $a, b, c \in S$:

(i) Antisymmetry: If $a \leq b$ and $b \leq a$, then $a = b$,

(ii) Transitivity: If $a \leq b$ and $b \leq c$, then $a \leq c$,

(iii) Connexity: Either $a \leq b$ or $b \leq a$.

Clearly, $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$ are totally ordered sets, but introducing this notion gives us more flexibility. When $a \leq b$ we say that a is smaller than or equal to b.

Definition 3.16 (Binary search trees (BST))**.** Binary search trees (BST)*, also called* ordered *or* sorted binary trees *is a binary tree* $T = (V, E)$ *together with a (weight) function* $\omega : V \to S$*, with* $(S, \leq)$ *a totally ordered set, such that for any* $x \in V$*:*

- $\omega(y) \leq \omega(x)$ *for any* $y \in L(x)$*,*
- $\omega(x) \leq \omega(y)$ *for any* $y \in R(x)$*,*

where $L(x)$ and $R(x)$ are the left and and the right subtree defined below x. The values at the vertices, namely $\{\omega(x) \in S \mid x \in V\}$, are called the keys.

Two examples of binary search trees are presented in Figure 3.15. Let us emphasize that binary search trees do not consist only in the values of the keys: the structure of the tree and accordingly the position of each vertex is important. For example, even though the two binary search trees of Figure 3.15 contain the same keys, they are very different and exhibit different responses to a research algorithm. It takes only four comparisons to determine that the number 20 is not one of the key stored in the left tree of Figure 3.15, while the same conclusion is obtained only after nine comparisons in the right tree.

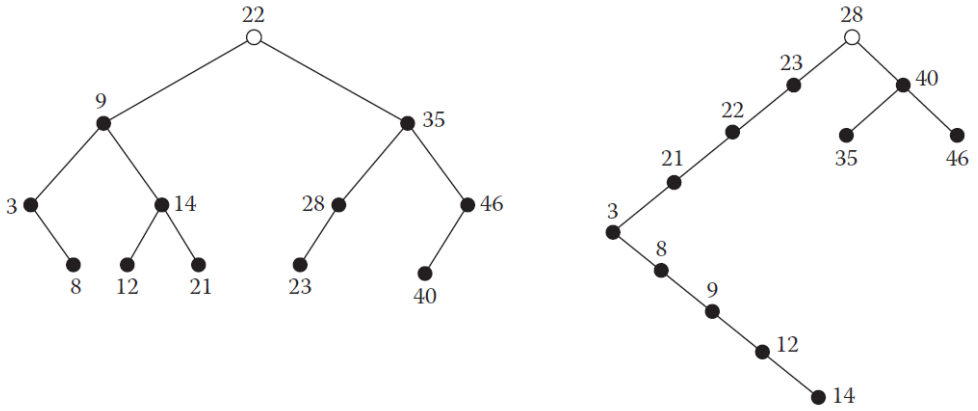


Figure 3.15: Two binary search trees, see Figure 3.4.1 of [GYA]

In practice, BST have a better behaviour if the two subtrees at each vertex contain roughly the same number of vertices. In such a case, we say that the binary tree is *balanced*.

It is easily observed that the smallest key in a BST is always stored in the most left vertex. This can be found starting from the root and proceeding always to the left until one reaches a vertex with no left-child. Similarly, the largest value is always attached to the most right vertex. This vertex can be found starting from the root and proceeding always to the right until a vertex with no right-child is reached.

More generally, the vertex corresponding to a certain key can be found by a simple iteration process, excluding always either a left subtree or a right subtree from the rest of the search. For a balanced tree containing n vertices, such a search requires an average of $O(\ln(n))$ operations[2]. Indeed, for a balanced tree, the relation between the height h of the tree and the number n is of the form $2^h \cong n$, which means that a leaf can be reached in about $\log_2(n)$ steps.

Two other primary operations can be performed on BST, namely the *insert operation* and the *delete operation*. The first one consists in adding a vertex corresponding to a prescribed new key by first looking at the right position for this new vertex. The second operation consists in eliminating a vertex but keeping the structure of a BST. These operations can be studied as an exercise.

3.4.3 Huffman trees

In this section we discuss the use of binary trees for creating efficient binary codes.

[2]For a strictly positive function ζ, the notation $f \in O(\zeta(n))$ means that $\left|\frac{f(n)}{\zeta(n)}\right| \leq c$ for some $c < \infty$ and all n, while $f \in o(\zeta(n))$ means $\lim_{n\to\infty} \frac{f(n)}{\zeta(n)} = 0$. These notations give us an indication about the growth property of the function f without looking at the details.

Definition 3.17 (Binary code)**.** *A* binary code *is a bijective map between a finite set of symbols (or alphabet) and a set of finite sequences made of* 0 *and* 1*. Each sequence is called* a bit string *or* codeword*.*

In this definition, the set of symbols can be arbitrary, like a set of letters, a set of words, a set of mathematical symbols, and so on. Also, the finite sequences of 0 and 1 can either be all of the same length, or have a variable length. In the latter case, it is important that a given sequence does not correspond to the first part of another sequence. In that respect, the following definition is useful:

Definition 3.18 (Prefix code)**.** *A* prefix code *is a binary code with the property that no bit string is the intial part of any other bit string.*

It is quite clear that an ordered tree can be associated with any binary code. For this, it is sufficient to associate the value 0 to the edge linking a father to the left child (if any), and a value 1 to the edge linking a father to the right child (if any). In such a construction, the difference between a binary code and a prefix code is clearly visible: in the former case a symbol can be associated to any vertex, while in the latter case a symbol is only associated to leaves, see Figure 3.16.

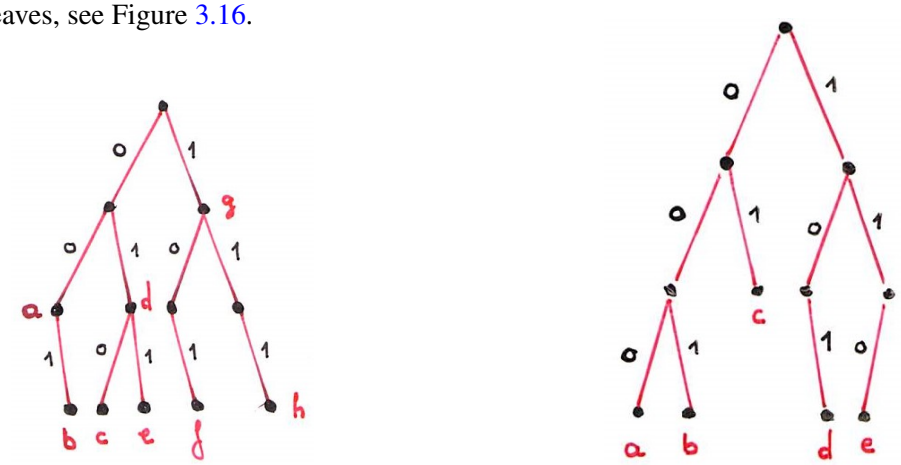


Figure 3.16: Trees of a binary code and of a prefix code

The code ASCII is a binary code in which each bit string has a fixed length. On the other hand, the example presented in Figure 3.17 corresponds to a prefix code together with the associated binary tree.

letter	*a*	*b*	*c*	*d*	*e*	*f*	*g*
codeword	000	0010	0011	0101	011	100	101

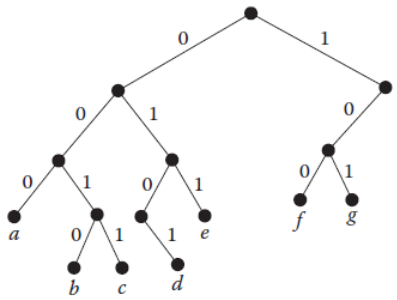


Figure 3.17: A prefix code and the corresponding tree, see Figure 3.5.1 of [GYA]

For some applications it would be be quite natural to use short codewords for symbols which appear frequently.

For example, in a prefix code for an English text, one would like to associate a short bit string to the letter e which appears quite often, and accept a longer bit string for a letter which is much more rare. One efficient way to realize such a prefix code is to use the *Huffman code*. It is based on one additional information associated with any symbol: its frequency or its weight. The following definition is based on the notion of weighted length for weighted graphs already introduced in Definition 1.28. We shall also use the information about the height or depth of a vertex, as introduced in Definition 3.7. Recall that the depth corresponds to the length $\mathrm{d}(r, x)$ of the path between the root r and a vertex x. In the present setting, this length is also equal to the number of elements of the codeword associated to a vertex.

Definition 3.19 (Weighted depth)**.** *Let T be a tree with leaves $\{x_1, x_2, \ldots, x_n\}$, and let $\{\omega_i\}_{i=1}^n \subset [0, \infty)$ be weights associated with the leaves. Then the* weighted depth $\omega(T)$ *of the tree is defined by*

$$\omega(T) := \sum_{i=1}^{n} \omega_i \, \mathrm{d}(r, x_i).$$

Input: a set $\{s_1, \cdots, s_l\}$ of symbols; a list $\{w_1, \cdots, w_l\}$ of weights,
where w_i is the weight associated with symbol S_i.
Output: a binary tree representing a prefix code for a set of symbols whose codewords have minimum average weighted length.
Initialize F to be a forest of isolated vertices, labeled $s_1, \cdots, s_l$, with respective weights $w_1, \ldots, w_l$.
For $i = 1$ to $l - 1$
Choose from forest F two trees, T and T', of smallest weights in F.
Create a new binary tree whose root has T and T' as its left and right subtrees, respectively.
Label the edge to T with a 0 and the edge to T' with a l.
Assign to the new tree the weight $w(T) + w(T')$
Replace trees T and T' in forest F by the new tree.
Return F.

Figure 3.18: Huffman algorithm, from Algorithm 3.5.1 of [GYA]

Note that if the weight ω_i associated to a leaf x_i is related to the frequency of the letter s_i corresponding to that leaf, and if $\sum_i \omega_i = 1$, then the weighted depth provides an information about the length of the transcription of the text with this prefix code: the weighted depth corresponds to the average length of the bit strings used for the transcription.

Now, given a list of symbols $S := \{s_1, s_2, \ldots, s_l\}$ and a list of weights $\{\omega_1, \omega_2, \ldots, \omega_l\}$, the Huffman algorithm corresponds to constructing a tree T which minimizes the weighted depth $\omega(T)$. Note however that the solution is not unique, and the lack of unicity appears rather clearly in the algorithm presented in Figure 3.18. The resulting tree is called *a Huffman tree* and the resulting prefix code is called *a Huffman code*. Clearly, these outcomes depend on the given weights. An example of such a construction is provided in Figure 3.19. Let us finally mention the result which motivates the construction presented above:

Theorem 3.20 (Huffman's theorem)**.** *Given a list of weights $\{\omega_1, \omega_2, \ldots, \omega_l\}$, the Huffman algorithm presented in Figure 3.18 generates an binary tree T which minimizes the weighted depth $\omega(T)$ among all binary trees.*

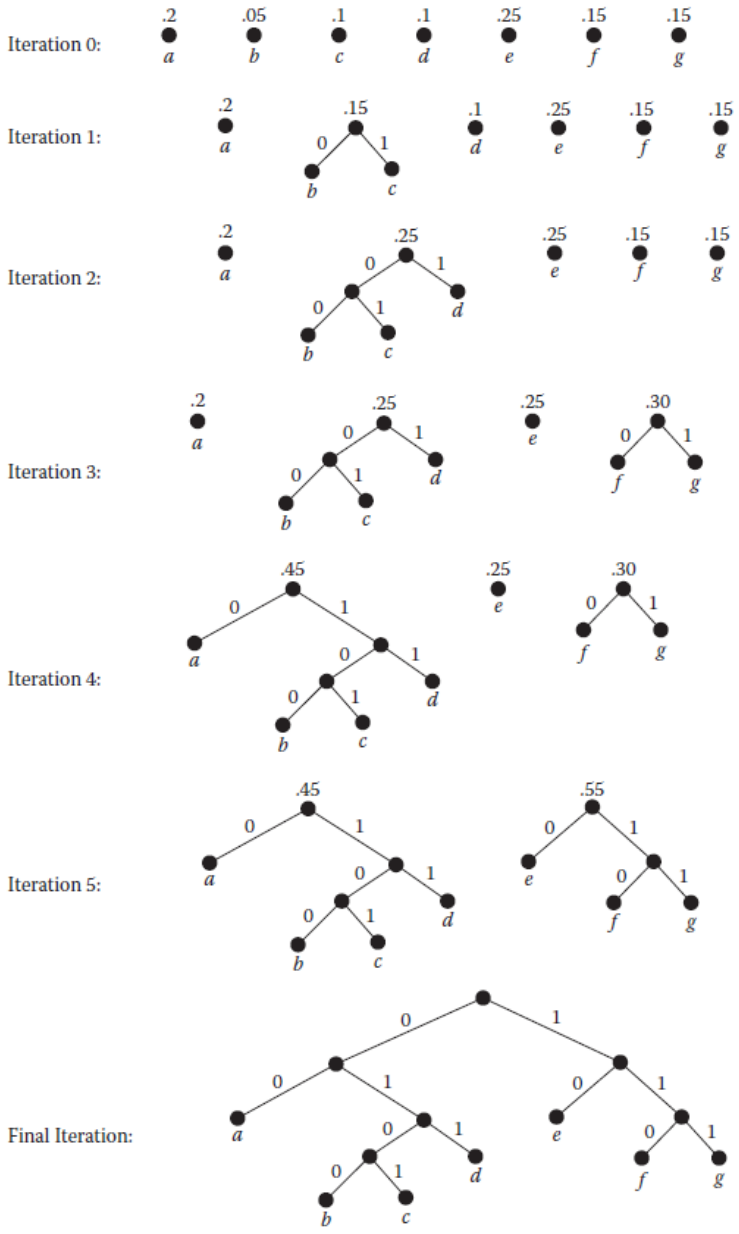


Figure 3.19: Application of Huffman algorithm, from Example 3.5.3 of [GYA]

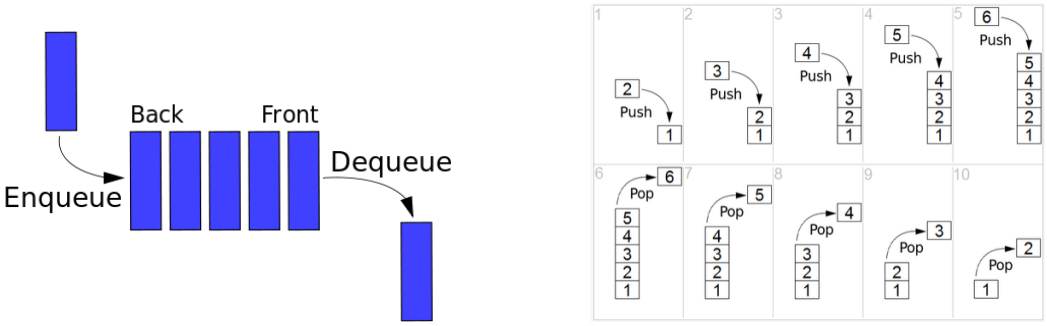
3.4.4 Priority trees

Let us present one more application of binary trees. First of all, we introduce some ideas more related to computer science.

Definition 3.21 (Abstract data type)**.** *An* abstract data type *is a set of objects together with some operations acting on these objects.*

Two such objects are quite common: 1) A *queue*, which is a set of objects that are maintained in a sequence which can be modified by the addition of new objects (enqueue) at one end of the sequence and the removal of objects (dequeue) from the other end of the sequence. A queue is also called FIFO (First In, First Out), and a

representation of a queue is provided in Figure 3.20a. 2) A *stack*, which is a set of objects that are maintained in a sequence which can be modified by the addition of new object on the top of the sequence (push operation) or removed also on the top of the sequence (pop operation). A stack is also called LIFO (Last In, First out), and a representation of a stack is provided in Figure 3.20b.



(a) A queue (b) A stack

Figure 3.20: Two standard abstract data types

We now generalize these two examples.

Definition 3.22 (Priority queue)**.** *A* priority queue *is a set of objects, each of which is assigned a* priority*, namely an element of a totally ordered set. The* operation of addition (enqueue) *consists simply in adding one more object together with its priority to the set, while the* operation of removal (dequeue) *consists always in removing the object with the largest priority. If two objects share the largest priority, an additional selection rule has to be prescribed.*

Note that the queue and the stack already mentioned as special instances of priority queues. In the former one, the lowest priority is always given to the newest object, while in the latter the largest priority is always given to the newest object. One can always represent a priority queue in a *linked list*, with the links sorted by decreasing priorities. Note that a linked list is also an abstract data type consisting in a set of objects, where each object points to the next in the set, see Figure 3.21. However, a better suited and more efficient implementation of a priority queue can be obtained with priority trees, as introduced below. A priority tree corresponds in fact to the most natural representation of a priority queue.

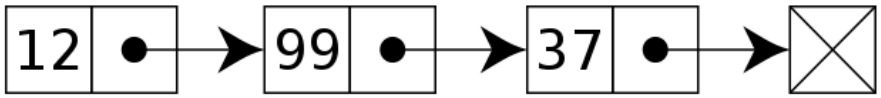


Figure 3.21: A linked list, with a terminal object (terminator) represented by a box

For priority trees, recall first that complete p-ary trees were introduced in Definition 3.8. We now weaken a little bit the completeness requirement.

Definition 3.23 (Left-completeness)**.** *A binary tree of height h is called* left-complete *if the following conditions are satisfied:*

(i) Every vertex of depth $h-2$ or less has two children,

(ii) There is at most one vertex at depth $h-1$ that has only one child (a left-one)

(iii) No vertex at depth $h-1$ has fewer children than another vertex at depth $h-1$ to its right.

An example of a left-complete binary tree of height 3 is presented in Figure 3.22. In the sequel we shall endow the vertices of such a tree with one additional information. However, in order to keep the greatest generality, let us first introduce a notion slightly weaker than the totally ordered set already mentioned. A *partially ordered set* S consists in a set S with a binary operation $\leq$ satisfying the following three conditions for any $a, b, c \in S$:

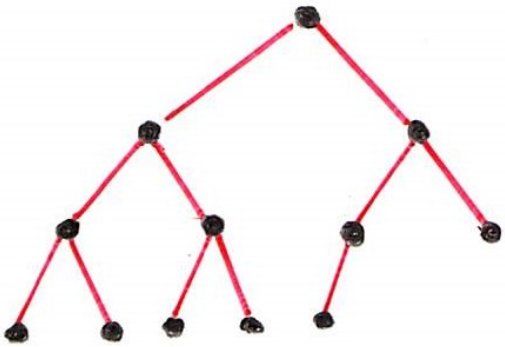


Figure 3.22: A left-complete binay tree

(i) Reflexivity: $a \leq a$,

(ii) Antisymmetry: If $a \leq b$ and $b \leq a$, then $a = b$,

(iii) Transitivity: If $a \leq b$ and $b \leq c$, then $a \leq c$.

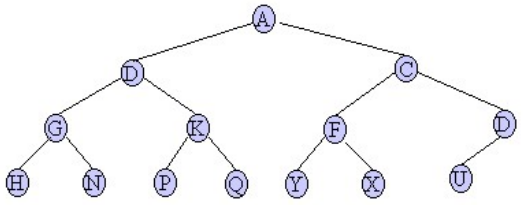
Clearly, a totally ordered set is also a partially ordered set, but the converse is not true. The main difference with a totally ordered set is that some elements a and b, the relations $a \leq b$ and $b \leq a$ could both be wrong. In such a case, we say that a and b are not comparable. An example of partially ordered set is provided by the set of all subsets of $\mathbb{R}$ with $A \leq B$ if $A \subset B$, for any subsets A and B of $\mathbb{R}$. For example, with intervals one has $(1, 2) \leq (0, 4)$, but $(0, 4)$ and $(1, 5)$ are not comparable.

We can now introduce a general definition of priority trees:

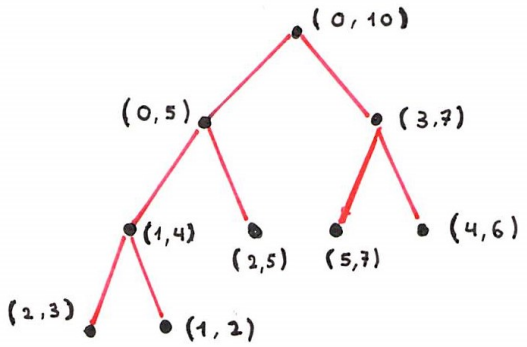
Definition 3.24 (Priority tree)**.** A priority tree *is a left-complete binary tree* $T = (V, E)$ *together with a map* $\omega : V \to S$*, with* $(S, \leq)$ *a partially ordered set, such that for any* $x \in V$ *and any child* y *of* x *one has* $\omega(y) \leq \omega(x)$*. The value of* S *are called the* priorities*.*

Binary search trees (BST), also called *ordered* or *sorted binary trees* is a binary tree $T = (V, E)$ together with a (weight) function $\omega : V \to S$, with $(S, \leq)$ a totally ordered set, such that for any $x \in V$:

- $\omega(y) \leq \omega(x)$ for any $y \in L(x)$,
- $\omega(x) \leq \omega(y)$ for any $y \in R(x)$,



(a) The alphabetical total ordering



(b) The inclusion partial ordering

Figure 3.23: Two examples of priority trees

Two examples of priority trees are presented in Figure 3.23. Note that Figure 3.23a is based on a totally ordered set, while Figure 3.23b is based on a partially ordered set. Note also that even though a vertex can not have a larger priority than its parent, it can have a larger priority than a sibling of its parent. From the definition, the piorities of a parent and of a child are always comparable, but the priorities of two siblings, or of elements which are further away, do not need to be comparable.

By comparing the definition of priority queue and of priority trees, it is quite clear that a priority queue can be represented in a priority tree. In fact, only special intances of priority trees are used for representing priority queues: the ones for which the priorities are chosen in a totally ordered sets[3].

When a priority queue is represented by a priority tree, the dequeue operation consists simply in extracting the root of the tree, but how can one obtain again a priority tree ? In fact, the removal and the addition of any element in a priority tree can be implemented by the following algorithms. The insertion of an arbitrary element in a priority tree can be implemented by the algorithm *Priority tree insert* presented in Figure 3.24. An example is provided in Figure 3.25.

Input: a priority tree T and a new entry x.
Output: tree T with x inserted so that T is still a priority tree.
Append entry x to the first vacant spot v in the left-complete tree T.
While $v \neq root(T)$ AND $priority(v) > priority(parent(v))$
Swap v with $parent(v)$.

Figure 3.24: Priority tree insert, from algorithm 3.6.1 of [GYA]

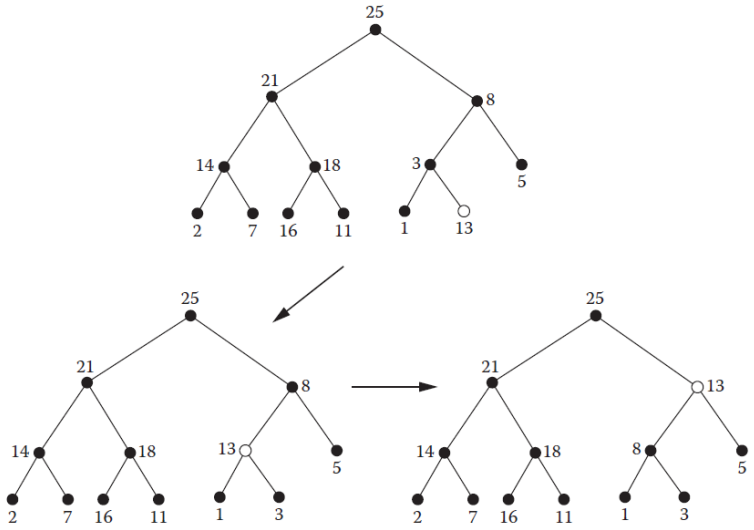


Figure 3.25: The insertion of 13 in the priority tree, see Figure 3.6.3 of [GYA]

The removal of an arbitrary element of a priority tree (as for example the root) can be implemented by the algorithm *Priority tree delete* presented in Figure 3.26. An example is provided in Figure 3.27.

3.5 Counting trees

Let us conclude this chapter with a question related to combinatorics: how many binary trees of n vertices can one construct ? For this question one has to remember that binary trees are ordered 2-ary trees (which implies that they are rooted). Clearly, if $n = 1$, there is only one such tree. If $n = 2$, two solutions exist: a root with a left child, or a root with a right child. The solutions for $n = 3$ are presented in Figure 3.28, but what about bigger n ?

[3] A more general notion of priority queue with a partially ordered set is possible, but the corresponding operations are not well defined, or not really natural.

Input: a priority tree T and an entry x in T.
Output: tree T with x deleted so that it remains a priority tree.
Replace x by the entry y that occupies the rightmost spot at the bottom level of T.
While y is not a leaf AND [$priority(y) \leq priority\ (leftchild\ (y))$.
OR $priority\ (y) \leq priority\ (rightchild(y))$]
If $priority\ leftchild\ (y) > priority\ (rightchild\ (y))$
Swap y with $leftchild\ (y)$.
Else swap y with $rightchild(y)$

Figure 3.26: Priority tree delete, from algorithm 3.6.2 of [GYA]

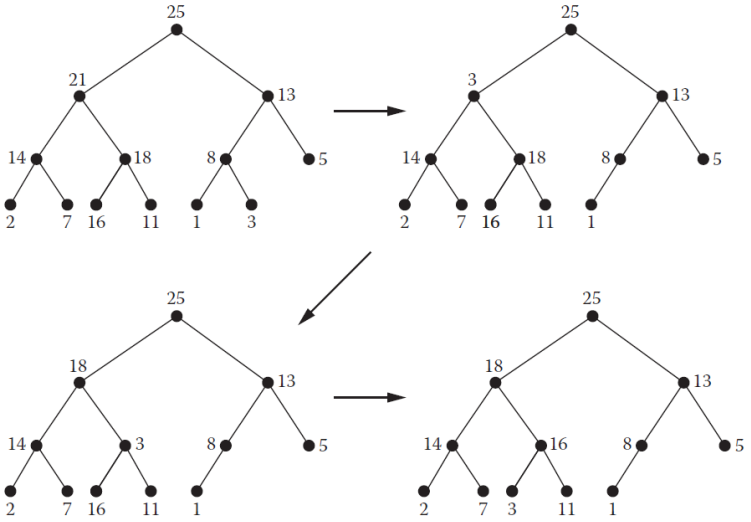


Figure 3.27: The deletion of 21 in the priority tree, see Figure 3.6.4 of [GYA]

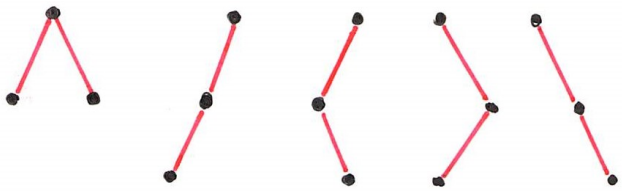


Figure 3.28: Rooted ordered trees with 3 vertices

One approach is by recursion. Let C_n denote the number of such binary trees of n vertices. As shown above, $C_1 = 1$, $C_2 = 2$ and $C_3 = 5$. Now, given the root of a tree containing n vertices, this root has a left subtree and a right subtree. If the left subtree contains j vertices (with $0 \leq j \leq n-1$) then the right subtree contains $n-j-1$ vertices. In such a case, there exists C_j possible binary subtrees for the left subtree, and C_{n-j-1} subtrees for the right subtree, making a total of $C_j C_{n-j-1}$ possible and different trees. Note that this formula holds if we fix by convention that $C_0 = 1$. Since j can vary between 0 and $n-1$ and since the solutions obtained for different j are all different, one obtains the recurrence relation

$$C_n = C_0 C_{n-1} + C_1 C_{n-2} + C_2 C_{n-3} + \ldots C_{n-2} C_1 + C_{n-1} C_0.$$

This relation can also be written more concisely:

$$C_n = \sum_{j=0}^{n-1} C_j C_{n-j-1}$$

and is called the *Catalan recursion*. Then numbers C_n are known as the *Catalan numbers*, and appear in various counting problems, see [9]. A closed formula exists for the computation of these numbers, In fact one has

$$C_n = \frac{1}{n+1}\binom{2n}{n},$$

where $\binom{2n}{n}$ denote a binomial coefficient. Several proofs of this result are presented in [9].

Another formula which will appear later on is the so-called *Cayley's formula*. This formula counts the number of non-isomorphic trees with *n labeled vertices*. These labels can be identified with a different weight assigned to each vertex, see Section 1.4 for the definition of weighted graphs. For weighted graphs, any isomorphism has to respect the weights, which means that the functions (f_V, f_E), from the weighted graph $G = (V, E, \omega)$ to the weighted graph $G' = (V', E', \omega')$, introduced in Definition 2.8 have to satisfy for any $x \in V$ and $e \in E$

$$\omega'(f_V(x)) = \omega(x) \quad \text{and} \quad \omega'(f_E(e)) = \omega(e).$$

Obviously, if only the vertices (or the edges) are endowed with weights, only one of these conditions has to be satisfied.

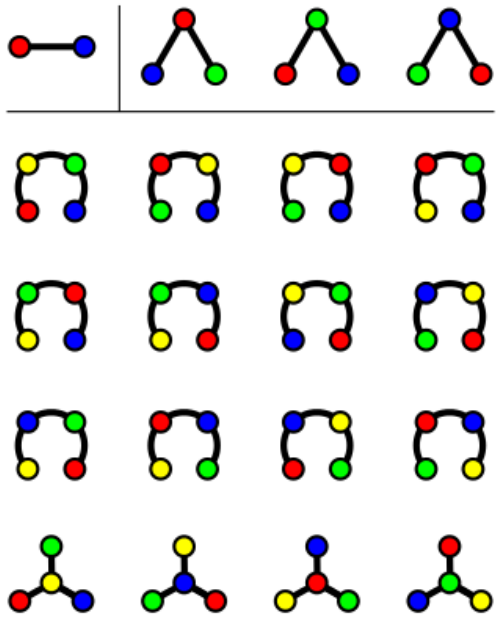


Figure 3.29: Trees with 2, 3 and 4 labeled vertices, see [10]

In Figure 3.29 labels on vertices are indicated by colors, and only trees are considered. All non isomorphic trees of $2, 3$ and 4 labeled vertices are presented. Calyley's formula states that for n labeled vertices, the number of non-isomorphic trees is n^{n-2}. Several proofs are indicated on [10], and one is fully presented in [GYA, Sec. 3.7].

Chapter 4

Spanning trees

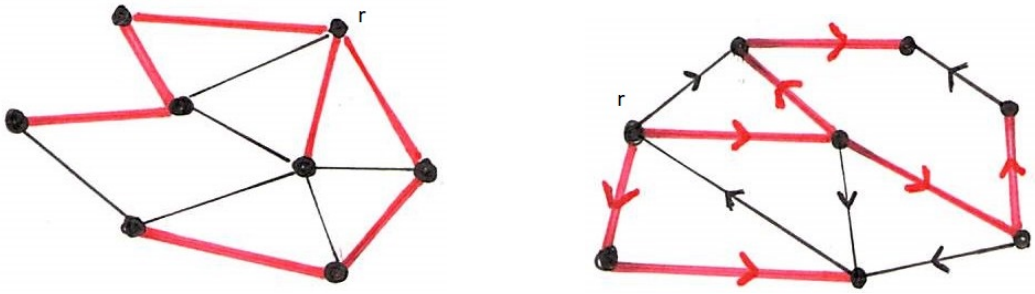
In this chapter we study spanning trees and their construction. These trees have many applications, and nice mathematical properties ☺.

4.1 Growing trees

Before stating the main definition of this section, let us recall that an arborescence is a directed rooted tree with all edges pointing away from the root. As a consequence, there exists a unique (oriented) path from the root to any vertex of the arborescence.

Definition 4.1 (Spanning tree)**.** *Let G be a connected graph. A spanning tree T of G is a subgraph of G which is either an unoriented tree or an arborescence which includes every vertex of G.*

A spanning tree T is a subgraph which induces or spans G, as introduced in Definition 1.6. For undirected graphs, a spanning tree can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices. Note that usually one fixes a root for the tree, but it is not strictly necessary. It is rather clear that for undirected connected graphs, a spanning tree always exists (and often it is non unique). On the other hand, for directed graphs, the requirement that the spanning tree is also an arborescence makes its existence less likely, but it is only with this additional property that spanning trees are useful for directed graphs, see Figure 4.1.



(a) An unoriented tree (b) An arborescence

Figure 4.1: Two spanning trees with root r

Our next aim is to construct such spanning trees. There exists several algorithms, but they all rely on a few definitions which are introduced now.

Definition 4.2 (Tree edge, vertex edge, frontier edge)**.** *Let G be a connected graph, and T be a subgraph of G which is a tree.*

(i) *A* tree edge *or a* tree vertex *is an edge or a vertex of G which belongs to T. A* non-tree edge *or a* non-tree vertex *is an edge or a vertex of G which does not belong to T.*

(ii) *A* frontier edge *is an non-tree edge with one endpoint in T (called* tree endpoint*) and the other not in T (called* non-tree endpoint*). If G is directed, the non-tree endpoint has to correspond the target. The set of all frontier edges if denoted by* $\mathrm{Front}(G, T)$.

The frontier edges $\mathrm{Front}(G, T)$ of an undirected graph is represented in Figure 4.2. For directed graph, a frontier edge is also called *a frontier arc* and the requirement is that the edge points outside of the tree. The general role of frontier edges is to be added to the existing tree (or arborescence in case of directed graphs) and to make these structures grow, as easily shown with the following lemma.

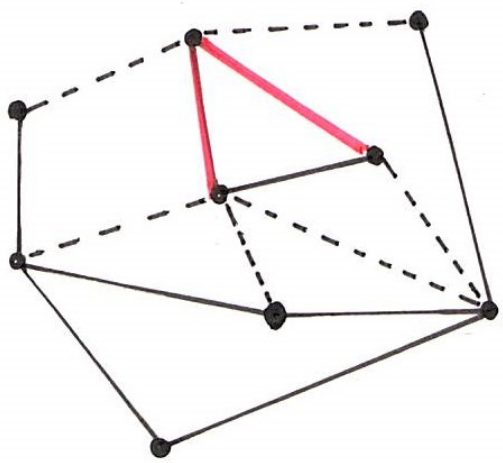


Figure 4.2: A graph, a tree, and the frontier edges as dashed lines

Lemma 4.3. *Let G be a connected graph, and T be a subgraph of G which is an unoriented tree or an arborescence. The addition of a frontier edge eto T generates a new subgraph of G which is an unoriented tree or an arborescence. This new tree is denoted by $T \sqcup \{e\}$.*

Given an oriented tree or an arborescence in G, one natural question is about the choice of a frontier edge. Indeed, there often exist several edges which are frontier edges, how can one choose a particular one and based on which criteria ? Each criterion (selection rule) corresponds in fact to a different algorithm. Note that choosing one element inside the set of all frontier edges can be either a deterministic operation or a random operation. Note also that once a frontier edge has been chosen, this set has to be updated. Indeed, at least this frontier edge has to be removed from the list, but possibly other frontier edges have to be removed, and new frontier edges might become available. Thus, growing an unoriented tree or an arborescence inside a connected graph G always consists in a series of several operations:

Algorithm 4.4 (Grow a tree)**.**

(i) *Fix an initial vertex x_0 of G, set $T_0 := \{x_0\}$ and fix $i := 0$,*

(ii) Choose one element e_{i+1} *of* Front(G, T_i)*, set* $T_{i+1} = T_i \sqcup \{e_{i+1}\}$*, and set* $i := i + 1$,

(iii) Repeat (ii) until Front$(G, T_i) = \emptyset$.

Let us make a few observations about this algorithm. The trivial initial tree in indeed just defined by a vertex, while at each subsequent step only the additional edge is mentioned. These information uniquely determine the tree. As mentioned before the choice of $e_{i+1} \in \text{Front}(G, T_i)$ will be determined by a prescribed procedure, and we shall see several subsequently. For an undirected finite graph, the algorithm will stop once a spanning tree has been obtained. For an arbitrary directed graph, this is much less clear, and the process might stop much before a spanning tree is obtained. The success of obtaining a spanning tree in this case will highly depend on the choice of the initial vertex (the root) and of the structure of the directed graph G. For example, in Figure 4.3 it is possible to create an arborescence starting from the vertex a but not from the vertex b. On the other hand, if a graph (directed or undirected) contains an infinite number of vertices, the algorithm might never end. Note finally that even if Front(G, T_{i+1}) is different from Front(G, T_i), it might not be necessary to compute this set from scratch but some information can be inferred from Front(G, T_i).

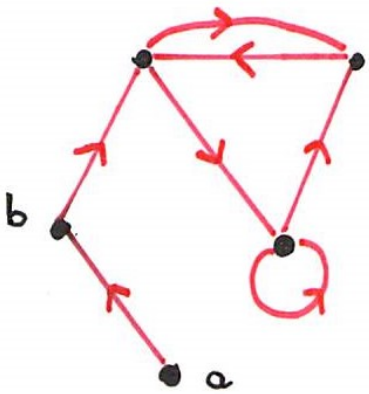
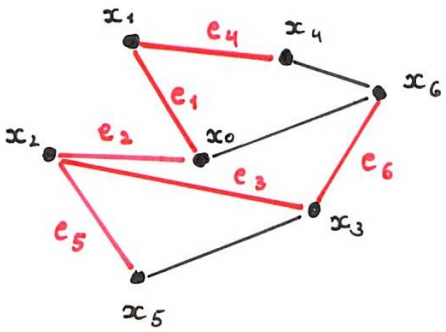
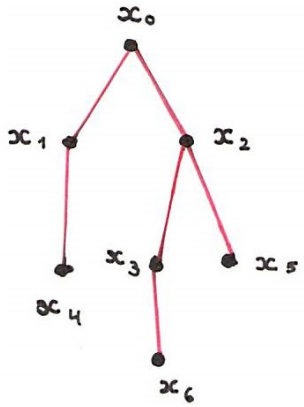


Figure 4.3: A digraph

Remark 4.5 (Discovery number)**.** *In the point (ii) of the above algorithm, we have written* "Choose one element e_{i+1} of Front(G, T_i), set $T_{i+1} = T_i \sqcup \{e_{i+1}\}$" *and not simply* "Choose one element e of Front(G, T_i), set $T_{i+1} = T_i \sqcup \{e\}$" *which would have been sufficient. The interest in the first notation is that it keeps an ordering in the growth of the tree. In fact, this ordering is called* the discovery number *and can be associated uniquely to each edge or to each vertex of the tree. For the edge, the discovery number of* e_i *is simply* i*, while for the vertices, we set* x_i *for the non-tree endpoint of the edge* e_i *(before this endpoint becomes also part of the tree). The function associating its discovery number to any vertex of the tree is often called the* dfnumber*-function. The discovery number also endows the tree with the structure of an ordered tree, see Figures 4.4a and 4.4b. With this notation, one can write precisely* $T_i = (V_i, E_i)$ *with* $V_i = \{x_0, x_1, \ldots, x_i\}$ *and* $E_i = \{e_1, e_2, \ldots, e_i\}$.



(a) A graph with a spanning tree



(b) The corresponding ordered tree

Figure 4.4: A graph, a spanning tree, and the resulting ordered tree

Let us now suppose that the process of growing a tree has ended up in a spanning tree. Except if the graph G itself was a tree, otherwise some edges of G do not belong to the tree, they are non-tree edges. These edges

can be divided into two sets: the *skip-edges* and the *cross-edges*. Skip-edges link two vertices which are in the same "family", one being an ancestor of the other one, while cross-edges link two vertices which are not in the same "family", none being an ancestor of the other one. Note that for skip-edges in directed graphs, one speaks about *back-edge* or *back-arc* if the target of the edge is the ancestor while it is a *forward-edge* or *forward-arc* if the tail of the edge is the ancestor. Cross-edges of directed graphs can also be separated into two subclasses, those linking a vertex with a discovery number to a vertex with a larger one, and those linking a vertex with a discovery number to a vertex with a smaller one. Note that loops have not been considered in this classification and should be considered as a family in itself.

As final note, let us observe that only connected graphs have been considered in this section. Clearly, the process of growing a free will not be able to visit more than one component of a graphs made of several components. However, it is not difficult to extend the construction and develop the growth of a forest. The missing necessary step is to allow the start of a new tree in a component different from the initial one. By iterating this procedure, one ends up with a forest and can define *a spanning forest*.

4.2 Depth-first and breadth-first search

We present here two classical solutions for choosing the element of Front(G, T_i) in the part (ii) of Algorithm 4.4.

The main idea of *Depth-first search (DFS)* is to start at the root node and explores as far as possible along each branch before backtracking. For that purpose, the frontier edge $e_{i+1} \in \text{Front}(G, T_i)$ is chosen with a tree endpoint at x_j with the largest number j (starting from i and then backward). Whenever more than one edge satisfy this condition (one speaks about *ties*) a priority rule has to be imposed. This priority can be either random, or based on some *a priori* information. For example, if indices had been attributed to edges or to vertices, they can be used to implementing an additional selection rule. Such a rule is called a *default priority*.

One tree constructed with the depth-first search will naturally be called a *depth-first search tree*. Usually, such a tree is not unique, and it is surely not unique if two edges in Front(G, T_i) had their tree endpoint at x_i. However, one easy property of depth-first search trees is provided in the next statement. Its proof can either be found in [GYA, Prop. 4.2.1] or by a minute of thought.

Lemma 4.6. *For an undirected graph, any depth-first search tree has no cross-edges.*

Let us add two remarks which link the depth-first search to two already introduced concepts. Firstly, by using a slightly extended version of the pre-order traversal as introduced in Definition 3.14 on a depth-first search tree one reproduces the discovery order of the edges in the original graph, see Figure 4.5. Note that the mentioned extension corresponds to an extension of the pre-order traversal to general ordered trees, and not only to binary trees. Secondly, when the growth of a tree is implemented, the natural way to store the frontier edges is to to use a priority queue, see Definition 3.22. However, for depth-first search tree the structure of a stack is sufficient. Indeed, the newest frontier edges are given the highest priority by being pushed onto the stack (in increasing default priority order, if there is more than one new frontier edge). The recursive aspect of depth-first search also suggests the feasibility of implementation as a stack, see Figure 3.20b and Figure 4.6.

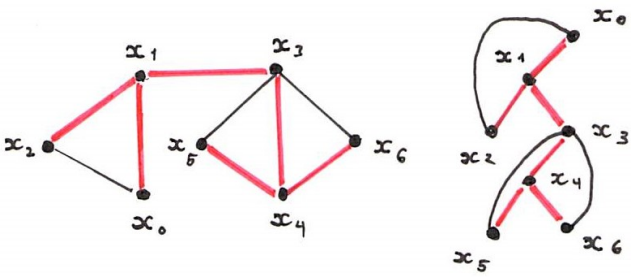


Figure 4.5: A DFS tree

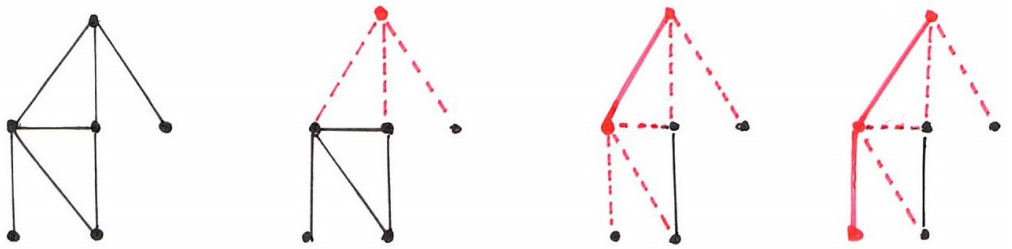


Figure 4.6: A growing DFS tree with Front(G, T_i) in dashed lines

Let us now move to *Breadth-first search (BFS)*. This time, the main idea is to start at the tree root and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level. For that purpose, the frontier edge $e_{i+1} \in \text{Front}(G, T_i)$ is chosen with a tree endpoint at x_j with the minimal number j (starting from $j = 0$ and then upward). Again, whenever more than one edge satisfy this condition a default priority is used.

One tree constructed with the breadth-first search will naturally be called a *breadth-first search tree*. Usually, such a tree is not unique, and it is surely not unique if two edges in Front(G, T_i) shared the same tree endpoint. This time, by using a slightly extended version of the level-order traversal as introduced in Definition 3.14 on a breadth-first search tree one reproduces the discovery order of the edges in the original graph, see Figure 4.7. Note that the mentioned extension corresponds to an extension of the level-order traversal to general ordered trees, and not only to binary trees. Additional properties of breadth-first search trees are provided in the next statement. The proof can either be found in [GYA, Sec. 4.2] or by a minute of thought. We recall that the level of a vertex in a tree has been introduced in Definition 3.7 and that the dfnumber-function has been introduced in Remark 4.5.

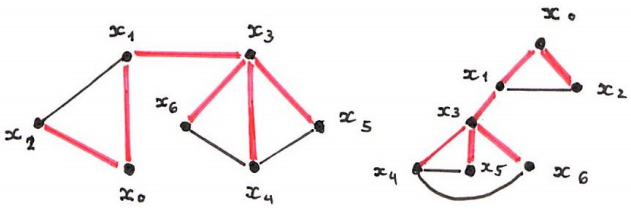


Figure 4.7: A BSF tree

Lemma 4.7.

(i) Let x, y be two vertices in a breadth-first search tree, then the property level(y) > level(x) *implies* dfnumber(y) > dfnumber(x)*,*

(ii) Any breadth-first search tree provides the shortest path tree of an unoriented graph with a given root, see also Definition 3.6.

The appropriate data structure to store the frontier edges in a breadth-first search is a queue, since the frontier edges that are oldest have the highest priority, see Figure 4.8.

A comparison between a DFS tree and a BFS tree is provided in Figure 4.9. Starting from the vertex v, it represents the trees obtained after eleven iterations of the Algorithm 4.4.

4.3 Applications of DFS

In this section we present some applications of depth-first search to connected and finite graphs. Extensions to non-connected ones can be done by considering the components separately.

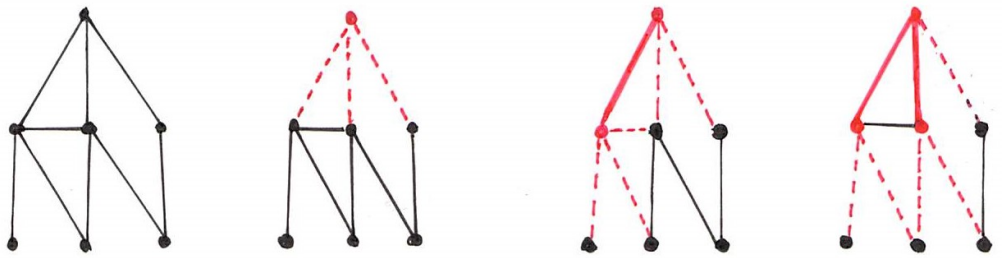


Figure 4.8: A growing BFS tree with Front(G, T_i) in dashed lines

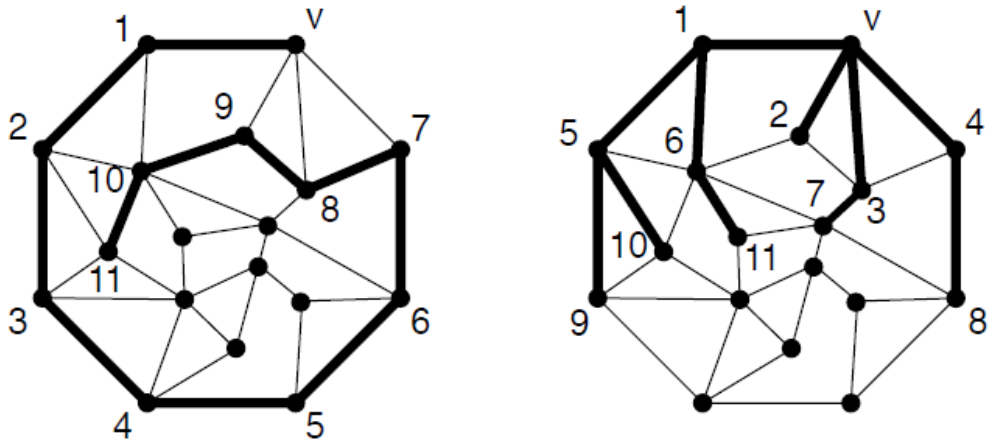


Figure 4.9: A DFS tree and a BFS tree after 11 iterations, see also Figure 4.2.2 of [GYA]

Let us recall that in Algorithm 4.4 the central idea of depth-first search is to look for an edge $e_{i+1} \in \text{Front}(G, T_i)$ with a tree endpoint at x_j with the largest number j (at most i). The following definition is related to this quest.

Definition 4.8 (Finished vertex)**.** *In a depth-first search, a discovered vertex is* finished *when all its neighbours have been discovered and those with higher discovery number are all finished.*

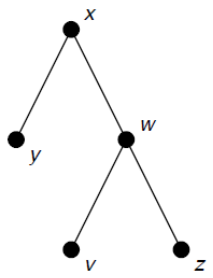
An example of a depth-first search is provided in Figure 4.10. Since the graph is simple, the name of each edge is indicated by the name of its two endpoints (xy means the edge between the vertices x and y). The column "nextEdge" corresponds of the edge which has been chosen among the frontier edges.

As mentioned in the previous section, a natural generalization of the pre-order traversal applied to a DFS tree provides the discovery order of the edges inside the original graph. Similarly, a slightly extended version of the post-order traversal, also introduced in Definition 3.14, applied to a DFS tree provide the list of the finished vertices, in the order they appear during the search. This property can be illustrated for example with Figure 4.5.

Usually, a depth-first search generates a walk since one has to backtrack several times in the construction of a spanning tree. In that respect, the following definition is natural.

Definition 4.9 (dfs-path)**.** *A* dfs-path *is a path produced by executing a depth-first search and by stopping the iteration right before one backtracks for the first time.*

Let us consider G undirected, and let x denote the endpoint of a dfs-path. Clearly, this path is also a tree, and



	nextEdge	vertex(discovery #)	frontier-edge set	vertex(finish #)
Initialization:		$x(0)$	$\{xy, xw\}$	
Iteration 1:	xw	$w(1)$	$\{xy, wv, wz\}$	
Iteration 2:	wv	$v(2)$	$\{xy, wz\}$	$v(1)$
Iteration 3:	wz	$z(3)$	$\{xy\}$	$z(2), w(3)$
Iteration 4:	xy	$y(4)$	$\emptyset$	$y(4), x(5)$

Figure 4.10: 4 iterations of a depth-first search, see also Example 4.4.1 of [GYA]

the vertex x is finished. One easily observes that all neighbours of x belong to the path, since otherwise the path could be extended. The following statement can then be easily deduced from this simple observation.

Lemma 4.10. *Let G be an undirected graph, and let x denote the endpoint of a dfs-path. Then either* $\deg(x) = 1$*, or x and all its neighbours belong to a cycle of G.*

Proof. The case $\deg(x) = 1$ is clear. Suppose now that $\deg(x) \geq 2$, and let y be a neighbour of x with the smallest value dfnumber among all neighbours of x. By the previous observation all neighbours of x are contained on the path between y and x. Since y is also a neighbour of x it means that there is a non-tree edge, see Definition 4.2, which links x to y. Thus, all neighbours of x are on a cycle, as claimed. □

Quite pleasantly, this result directly provides a proof to the statement (i) of Theorem 1.26. Namely, if G is a simple and undirected finite graph with minimum degree $\delta(G) \geq 2$, it contains a cycle of length at least equal to $\delta(G) + 1$. Clearly, if $\delta(G) \leq 1$, then the statement is not true but nevertheless the graph contains a path of length $\delta(G)$, as claimed in the statement of the theorem.

Let us now use the depth-first search for finding cut-edge (bridge) as introduced in Definition 2.20. Observe firstly that in an undirected and connected graph, an edge is a bridge if and only if it does not belong to any cycle in a graph, see also [GYA, Corol. 2.4.2]. Whenever a graph has some bridge(s), the following definition is natural.

Definition 4.11 (Bridge component)**.** *Let G be a connected graph, and let B be the set of all its bridges.* A bridge component *of G is a component of the graph $G - B$.*

It clearly follows from this definition and from the previous observation that the edges and the vertices of any cycle in a connected graph belong to the same bridge component. Let us go one step further in the construction.

Definition 4.12 (Contraction)**.** *Let $H = (V_H, E_H)$ be a subgraph of a graph $G = (V, E)$. The* contraction of H to a vertex *is the replacement of V_H by a single vertex k. Any edge between a vertex of V_H and a vertex of $V \setminus V_H$ is replaced by an edge between k and the same element of $V \setminus V_H$, while all edges between vertices of V_H do not appear in the contraction.*

Figure 4.11 provides an illustration of a contraction of three vertices.

Figure 4.11: Contraction of $V_H = \{a, b, c\}$ into the vertex k

As a result of these definitions and observations, one easily deduces the following important result:

Proposition 4.13. *Let G be a connected and undirected graph. The graph that results from contracting each bridge component of G to a vertex is a tree.*

Figure 4.12 corresponds to the content of this proposition.

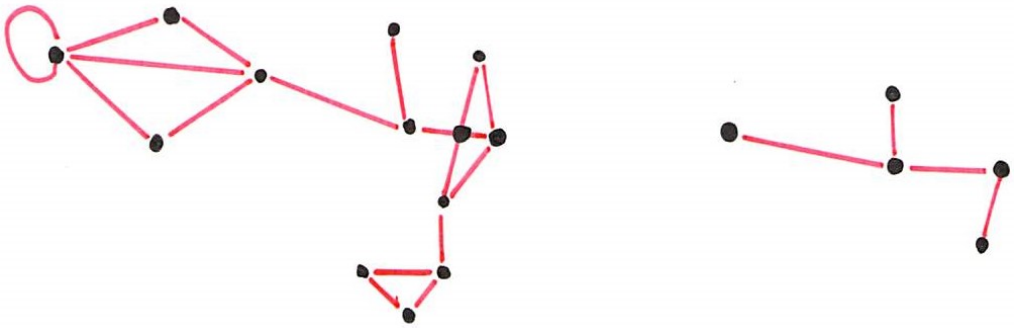


Figure 4.12: Before and after contracting each bridge component of G

Before stating the algorithm which allows us to identify the bridges of a graph, let us still observe that if x is a vertex of a connected graph G with $\deg(x) = 1$, then the bridge components of G is made of the component consisting on x only, and on the bridge components of $G - \{x\}$. The algorithm for determining cut-edges is presented below. Note that it has been developed for undirected graphs, and this assumption should be added at the beginning of the statement.

Let us emphasize that the bridges are important because they correspond to the "weaknesses" of a graph. By removing them, a connected graph becomes disconnected. Note that a similar procedure exists for cut-vertices, as introduced in Definition 2.19. The construction is of the same type and can be studied independently. We refer to [GYA, p. 196–200] for more information.

Input: a connected graph G.
Output: the cut-edges of G.
    Initialize graph H as graph G.
    While $|V_H| > 1$
        Grow a dfs-path to the first vertex t that becomes finished.
        If $deg(t) = 1$
            Mark the edge incident on t as a bridge.
            $H := H - t$.
        Else /$*$ vertex t and all its neighbors lie on a cycle C^* /
            Let H be the result of contracting cycle C to a vertex.
    Return the edges of H

Figure 4.13: How to find bridges, from algorithm 4.4.1 of [GYA]

4.4 Minimum spanning trees and shortest paths

In this section we refine the algorithm presented in Section 4.1 when a weighted connected graph is considered, see Definition 1.27. Note that weights will be attached only to edges and not to vertices, which means that we consider only edge-weights and accordingly *edges-weighted graphs*. Our aim is to construct a spanning tree with the minimum total edge-weight, the so-called *minimum spanning tree problem*. Since for a graph with n vertices, there could exist up to n^{n-2} trees, see Section 3.5, one is forced to look for an efficient algorithm.

The main idea for growing a tree with minimum total edge-weight is to use Algorithm 4.4 and to choose at each step (ii) the edge $e_{i+1} \in \text{Front}(G, T_i)$ with the smallest edge-weight. If there is more than one edge in $\text{Front}(G, T_i)$ with the smallest edge-weight, then the default priority can choose one of them. Note that this procedure is referred to as Prim's algorithm, since it has been proposed by R.C. Prim in 1957. It remains then to show that this procedure leads for undirected graphs to a spanning tree with minimum total edge-weight. This can be done by an inductive proof as presented for example in [GYA, Prop. 4.3.1]. Figure 4.14 provides the example of an edges-weighted graph with a minimum spanning tree.

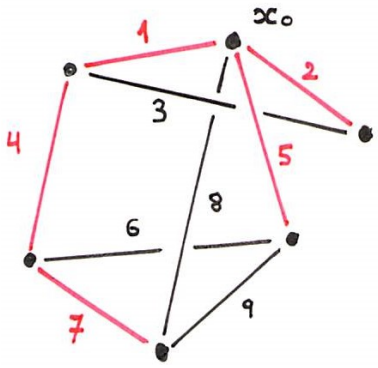


Figure 4.14: A minimum spanning tree

Let us mention that a generalization of the this problem consists in prescribing only a subset of vertices. More precisely, let $G = (V, E, \omega)$ be a connected edge-weighted graph, and let $U \subset V$. The *Steiner-tree problem* consists in finding a minimum total weight tree in G containing all vertices of U. Clearly, the special case $U = V$ corresponds to the minimum spanning tree problem. The Steiner-tree problem has been extensively studied, see [11]. A special instance of this problem consists in considering only two prescribed vertices in the graph (the set U contains only two elements). We now develop this situation.

For solving this problem, recall that the weighted length of a walk has been introduced in Definition 1.28. It corresponds to the sum of the weight on the corresponding edges, and if W denotes a walk in the graph, we write $\omega(W)$ for its weighted length. Accordingly, if x, y are vertices of a weighted graph, it would be natural

to define

$$\mathrm{d}_\omega(x,y) = \min\{\omega(W) \mid W \text{ is a walk from } x \text{ to } y\}.$$

However, this notion suffers from two weaknesses. The first one has already been observed: there might be no walk between x and y, in which case we set $\mathrm{d}_\omega(x,y) = \infty$. The second problem is more serious: if the graph contains cycles with negative length, then most of the distances would be equal to $-\infty$. In order to avoid this situation, the minimal requirement is to impose that the graph has no such cycle of negative length. One stronger requirement is to impose that all weights belong to $[0,\infty)$. Note that if we further impose $\omega(e) > 0$ for any edge e and if the graph is undirected, then the distance d_ω defined above endows the weighted graph $G = (V,E,\omega)$ with a metric[4]. If the graph is directed and/or if the weight is not strictly positive, d_ω does not correspond to a metric in general. Note finally that in the general setting, this "distance" might represent various quantities.

From now on, let us assume for simplicity that $\omega(e) \geq 0$, but mention that an extension with the only requirement of the absence of cycles of negative length exists (Floyd–Warshall algorithm). As an easy consequence, one always has $\mathrm{d}_\omega(x,x) = 0$ for any $x \in G$. In the sequel we construct more than just the weighted path from a prescribed x_0 to a prescribed y with the minimum weight, we construct such a path from x_0 to any vertex y in the graph. In fact, we construct a tree with root x_0, and the minimum weighted path from x_0 to any y is then uniquely defined by the tree. This tree is called *a Dijkstra tree*, since it has been proposed by E. Dijkstra in 1959. The construction is again based on Algorithm 4.4 with a clever choice of $e_{i+1} \in \mathrm{Front}(G,T_i)$. However, since an additional information has to be kept during the process, we provide below the updated version of the algorithm.

Before this, let us adapt an already old concept. Since the edges in $\mathrm{Front}(G,T_i)$ have a tree endpoint and a non-tree endpoint, it is rather natural to use the origin map $\mathrm{o} : E \to V$ and the target map $\mathrm{t} : E \to V$ already introduced in Section 1.1. With this notation $\mathrm{o}(e)$ will denote the tree endpoint, while $\mathrm{t}(e)$ corresponds to the non-tree endpoint. Note that in the present framework this notation is natural both for oriented and non-oriented edges.

Algorithm 4.14 (Dijkstra's tree algorithm)**.**

(i) Fix an initial vertex x_0 of G, set $T_0 := \{x_0\}$ and fix $i := 0$,

(ii) Choose the element $e_{i+1} \in \mathrm{Front}(G,T_i)$ which satisfies

$$\mathrm{d}_\omega(x_0,\mathrm{t}(e_{i+1})) := \min_{e\in\mathrm{Front}(G,T_i)} \Big(\mathrm{d}_\omega(x_0,\mathrm{o}(e)) + \omega(e)\Big). \tag{4.4.1}$$

Set $T_{i+1} = T_i \sqcup \{e_{i+1}\}$, and set $i := i+1$,

(iii) Repeat (ii) until $\mathrm{Front}(G,T_i) = \emptyset$.

Note that if more than one edge satisfies condition (4.4.1), then the default priority is applied. Note also that this algorithm is well defined, since $\mathrm{d}_\omega(x_0,\mathrm{o}(e))$ always corresponds to $\mathrm{d}_\omega(x_0,x_j)$ for some $j \in \{0,1,\dots,i\}$. This comes from the fact that for any $e \in \mathrm{Front}(G,T_i)$ one has $\mathrm{o}(e) = x_j$ for some $j \in \{0,1,\dots,i\}$. It is clear that the implementation of this algorithm requires that the value $\mathrm{d}_\omega(x_0,x_{i+1})$ has to be kept in memory each time the new edge e_{i+1} is chosen. There exists several practical implementations of this algorithm, but we do not develop this any further. The correctedness of this algorithm can be proved by induction over the trees. One version is provided in [GYA, Thm. 4.3.3]. Alternatively, there are plenty of well documented websites on Dijkstra's tree algorithm over the internet, see also [12]. Note finally that Dijkstra's algorithm has been

[4]Recall that a metric is a map $\mathrm{d} : V \times V \to [0,\infty)$ satisfying the three conditions: 1. $\mathrm{d}(x,y) = 0 \Leftrightarrow x = y$, 2. $\mathrm{d}(x,y) = \mathrm{d}(y,x)$, and 3. $\mathrm{d}(x,y) \leq \mathrm{d}(x,z) + \mathrm{d}(z,y)$ for any $x,y,z \in V$.

developed in several directions. For example, one could compute simultaneously the minimum weighted paths from any x to any y, and not only from a fixed x to any y.

An example of the Dijkstra's algorithm is presented in Figure 4.15. The edge-weights are indicated above the edges, while the yellow disks contain an information about the distance from the root (upper-left vertex) to the corresponding vertex: Once the vertex is visited, it corresponds to $\mathrm{d}_\omega(x_0, x_i)$ and before it is visited it corresponds to a preliminary result when a comparison of the type (4.4.1) is computed (preliminary distance). This preliminary distance can only decrease, or stay constant if no path with a smaller weighted length is discovered inside the graph. For that reason, these preliminary distances are often set to ∞ before the start of the algorithm. At each step in the algorithm, one updates some of these values only if a smaller weighted length is found.

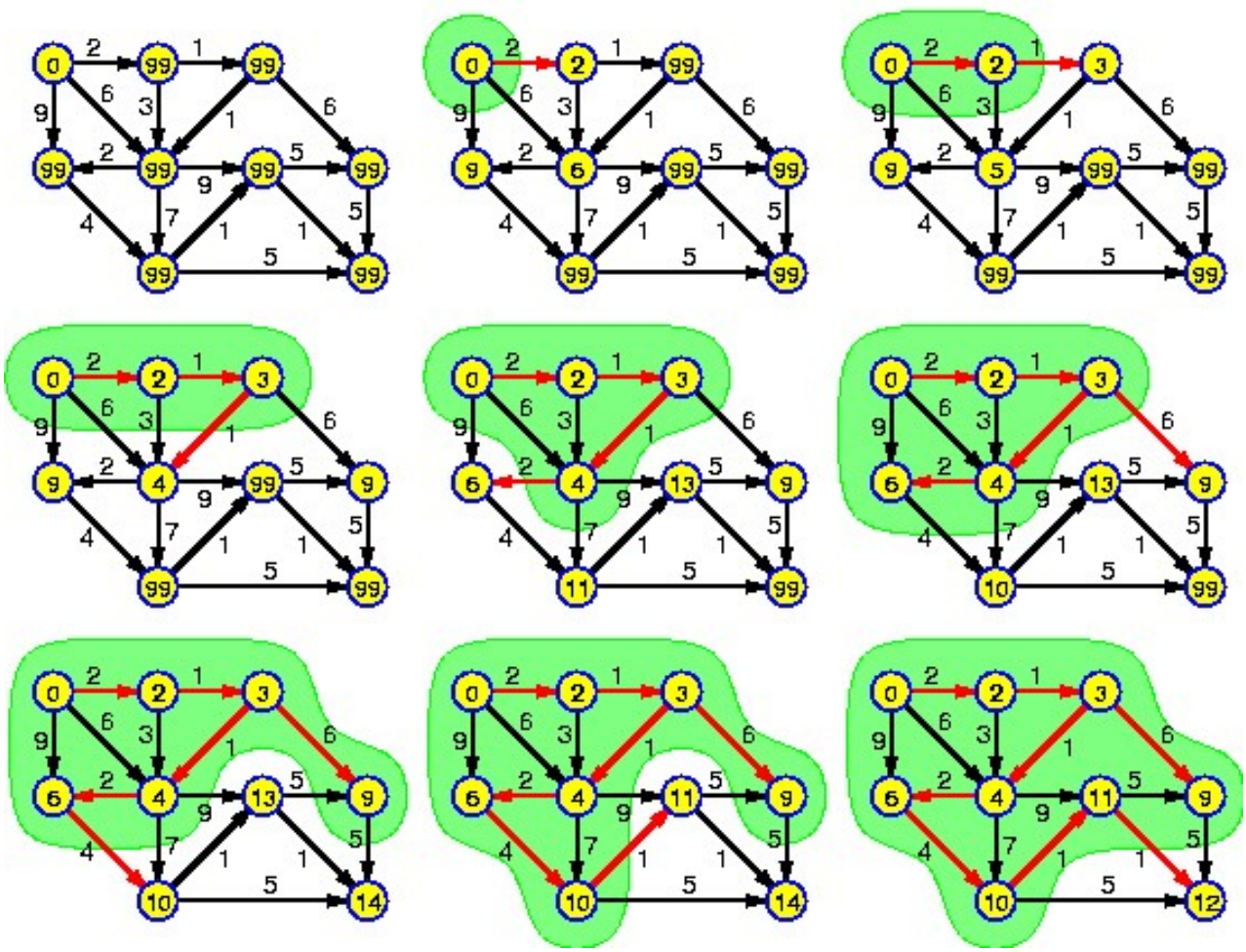


Figure 4.15: Application of Dijkstra's algorithm, from [13]

Chapter 5

Connectivity

Connectivity is an important concept in graph theory. Graphs with a dense connectivity or with a very weak connectivity do not present the same vulnerability towards the removal of some vertices or edges. Beside the notions of vertex-cut or edge-cut, and cut-vertex (=cutpoint) or cut-edge (=bridge) provided in Definitions 2.19 and 2.20, more refined concepts have to be introduced. Also, since loops do not play any role for the connectivity of a graph, the graphs considered in this chapter will be loopless.

5.1 Vertex and edge connectivity

Let us start by providing a concept measuring the density of connectivity in a graph. For clarity, recall that an operation *disconnects* a connected graph if after this operation the graph is no more connected. Let us however acknowledge that the notion of connectivity does not fit well with the notion of orientation. In fact, for digraphs more refined concepts are necessary, and will be introduced in subsequent chapters. As a consequence, the main statement of this section, namely Theorem 5.4, is applicable to unoriented graphs only. Its statement would be wrong for oriented graphs.

Definition 5.1 (vertex or edge connectivity)**.** *Let G be a connected graph.*

(i) The vertex connectivity *$\kappa_V(G)$ of G is the minimum number of vertices whose removal can either disconnect G or reduce it to a 1-vertex graph.*

(ii) The edge connectivity *$\kappa_E(G)$ of G is the minimum number of edges whose removal can disconnect G.*

Note that the minimum degree $\delta(G)$ already introduced in Section 1.1 must satisfy $\kappa_E(G) \leq \delta(G)$ (otherwise, one easily gets a contradiction). In fact, the two connectivities are not independent, one has

$$\kappa_V(G) \leq \kappa_E(G) \leq \delta(G). \tag{5.1.1}$$

The proof is left as an exercise. Now, in relation with these definitions one also sets:

Definition 5.2 (k-connectedness)**.** *Let G be a connected graph, and let $k \in \mathbb{N}$.*

*(i) The graph G is k-*vertex connected *(or simply k-*connected*) if $\kappa_V(G) \geq k$,*

*(ii) The graph G is k-*edge connected *if $\kappa_E(G) \geq k$,*

These notions are useful for discussing any network *survivability*, which is the capacity of a network to stay connected after some edges or vertices are removed. For example, if the vertices of the graph are divided into two subsets V_1 and V_2, then the number of edges between V_1 and V_2 is always greater or equal to $\kappa_E(G)$. An example of vertex connectivity and edge connectivity is provided in Figure 5.1.

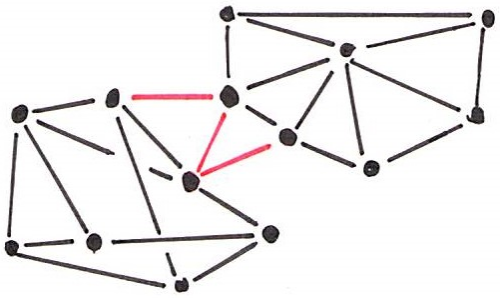


Figure 5.1: $\kappa_V(G) = 2$ and $\kappa_E(G) = 3$

Recall that internal vertices of a tree have been introduced in Definition 3.7. For a path which is not a cycle, the internal vertices correspond to all vertices of the path except its two endpoints. In order to discuss the vulnerability of a network, the following definition is useful:

Definition 5.3 (Internally disjoint paths)**.** *Let x, y be distinct vertices in a graph G. A family of paths from x to y is said to be* internally disjoint *if no two paths in the family have an internal vertex in common.*

A representation of two such path is provided in Figure 5.2. Already in 1932, H. Whitney provided a characterization of 2-connected graphs in terms of internally disjoint paths, namely: Any connected and unoriented graph with at least 3 vertices is 2-connected if and only if each pair of vertices in G admit two internally disjoint paths between them. There are several proofs of this *Whitney's 2-connected characterization* available on Internet. In fact, a more general characterization of 2-connected graphs can obtained, see also Figure 5.3. Its proof can be done as an exercise.

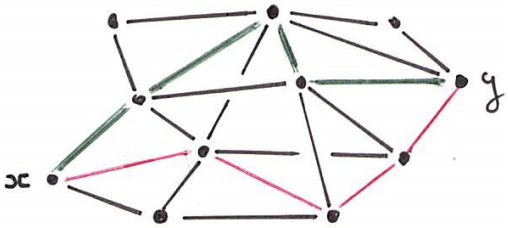


Figure 5.2: 2 internally disjoint paths

Theorem 5.4. *Let G be a connected, unoriented and finite graph with at least 3 vertices. The following statements are equivalent:*

(i) *G is 2-connected,*

(ii) *For any two vertices, there exists a cycle containing both,*

(iii) *For any vertex and any edge, there is a cycle containing both,*

(iv) *For any two edges, there is a cycle containing both,*

(v) *For any two vertices and one edge, there is a path from one vertex to the other one that contains the edge,*

(vi) *For any three distinct vertices, there is a path from the first to the third and containing the second,*

(vii) *For any three distinct vertices, there is a path containing any two of them and not the third one.*

Let us emphasize that 2-connected unoriented graphs can be seen as stable structures with respect to the deletion of an arbitrary vertex. Indeed, by the above statement (vii) it means that 2 vertices can always be joined by a path, even if another arbitrary vertex of the graph as been removed. Let us add that a similar description of 3-connected, unoriented and finite graphs also exists and is provided for example in [Die, Sec. 3.2].

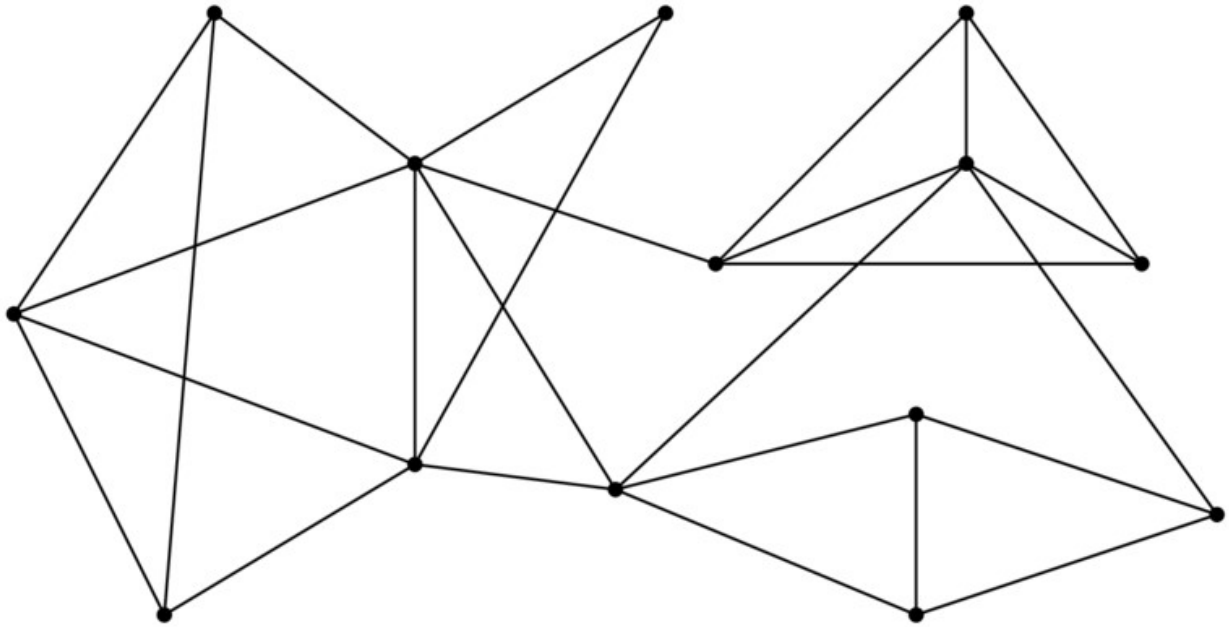


Figure 5.3: A 2-connected graph on which the above statement can be observed

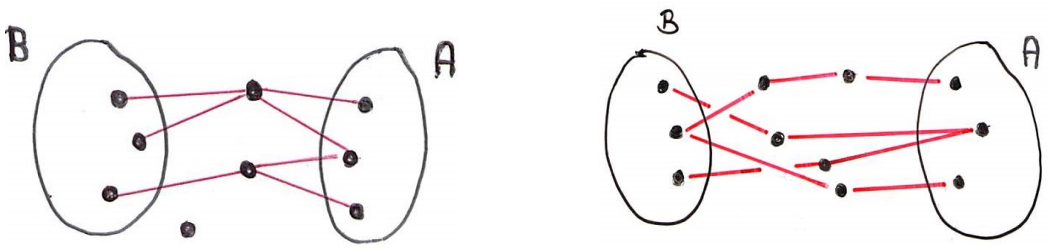
5.2 Menger's theorem

The aim of this section is to present Menger's theorem, one important result in graph theory. In order to state a rather general version of this theorem, we first extend some of the definitions already introduced. Note that in the following definitions, the sets A or B can not be the empty set.

Definition 5.5 (A-B-path)**.** *Let $G = (V, E)$ be a graph, and let $A \subset V$ and $B \subset V$. An A-B path is a path in G with its starting vertex in A, its end vertex in B, and no internal vertices in A or in B.*

Examples of A-B paths are presented in Figure 5.4a. With this first notion at hand, we naturally extend Definition 5.3.

Definition 5.6 (Internally disjoint A-B paths)**.** *Let $G = (V, E)$ be a graph, and let $A \subset V$ and $B \subset V$. A family of A-B paths is said to be* internally disjoint *if no two paths in the family have an internal vertex in common.*



(a) A-B paths

(b) Internally disjoint A-B paths

Figure 5.4: Subsets of vertices, and paths between them

Internally disjoint A-B paths are presented in Figure 5.4b. Clearly, if $A = \{x\}$ and $B = \{y\}$ for two vertices x, y of G, one comes back to Definition 5.3. Note that the interest of this definition is that the endpoints of the

different paths can be different elements of A and B.

For the next definition, recall that whenever $G = (V, E)$ is a graph and $S \subset V$, the graph $G - S$ corresponds to the induced graph $G[V \setminus S]$ as defined in Definition 1.6.

Definition 5.7 (A-B separator)**.** *Let $G = (V, E)$ be a connected graph, and let $A \subset V$ and $B \subset V$. A set $S \subset V$ is an A-B* separator *if $G - S$ contains no A-B path*[5]*. We also say that S* separates *the set A and B in G.*

An A-B separator is presented in Figure 5.5. Observe that this definition is related to Definition 2.19 about vertex-cut, but is more flexible, since it does not imply that $G-S$ is disconnected. Indeed, looking carefully at this definition, the notion of orientation is taken into account. More precisely, the definition of A-B path holds for directed graphs, and Definition 5.7 also takes care of orientation. An illustration of this concept is provided in Figure 5.6.

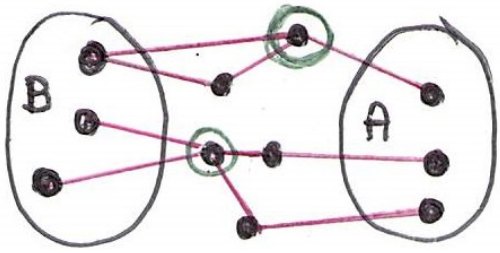


Figure 5.5: A-B separator (in green)

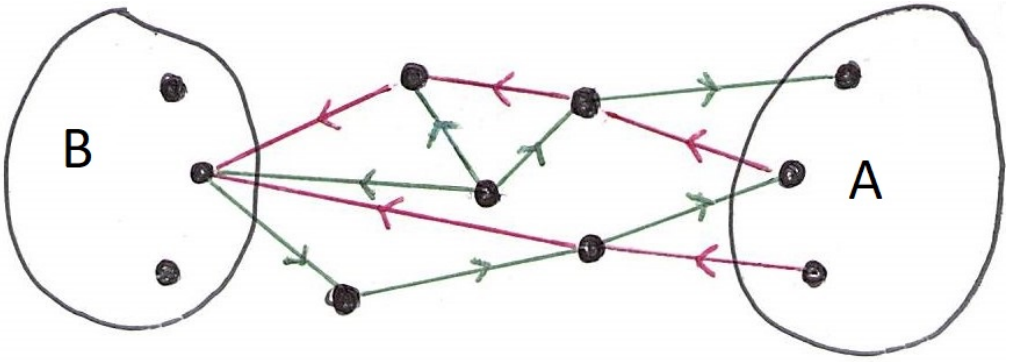


Figure 5.6: A-B paths with orientation (in red)

Now, given a graph $G = (V, E)$ and for two sets $A, B \subset V$, two related problems can easily be formulated:
Minimization problem: Determine the minimum number $\kappa(A, B)$ of vertices contained in any A-B separator.
Maximization problem: Determine the maximum number $\ell(A, B)$ of internally disjoint A-B paths.

A rather general version of Menger's theorem can now be stated. Note that quite often it is stated for $A = \{x\}$ and $B = \{y\}$, and only for unoriented graph. On the other hand, it is quite clear that only simple graphs can be considered: loops do not play any role, but multiple edges would clearly lead to wrong statements (one could add edges and paths without changing the number of vertices). In the next statement, we assume that $\kappa(A, B) \geq 1$, which impose that any A-B path is not reduced to a single edge (with no internal vertex). Without this assumption, the statement is simply not correct.

Theorem 5.8 (Menger's theorem)**.** *Let G be a connected, simple and finite graph, and let $A \subset V$ and $B \subset V$. If $\kappa(A, B) \geq 1$, then the equality $\kappa(A, B) = \ell(A, B)$ holds, or in other terms the minimum number vertices contained in any A-B separator is equal to the maximum number of internally disjoint A-B paths.*

Note that one inequality is easy to prove, namely $\ell(A, B) \leq \kappa(A, B)$. Indeed, let S denote an A-B separator set containing $\kappa(A, B)$ elements. Since S is A-B separating, each A-B path must contain at least one vertex of

[5]Note in particular that this definition implies that $S \cap A = \emptyset = S \cap B$.

S. If we impose that the paths are internally disjoint, it implies that there exists at most $\kappa(A, B)$ such paths. This directly leads to the stated inequality. Unfortunately the equality is more difficult to prove, but several proofs exist. In [Die, Sec. 3.3] three proofs are provided for undirected graphs; in [GYA, Sec. 5.3 & 10.3] one version for undirected and one version for directed graphs are provided, but only in the case $A = \{x\}$ and $B = \{y\}$. In [14] two versions of the proof are also provided. Note finally that an extension for infinite graphs also exists, but one has to be more cautious about equalities of the form $\infty = \infty$.

Let us present two consequences of the previous result. Since it is related to the notion of connectivity, it will hold for undirected graphs only. Indeed, for such graphs A-B paths are equal to B-A paths, which is not true in general for directed graphs.

Proposition 5.9. *Let G be a connected, simple, unoriented and finite graph containing at least one pair of non-adjacent vertices. Then the vertex connectivity $\kappa_V(G)$ satisfies*

$$\kappa_V(G) = \min\{\kappa(\{x\}, \{y\}) \mid x, y \text{ non-adjacent vertices of } G\}.$$

The proof of this statement is provided in [GYA, Lem. 5.3.5], while the proof of the following theorem is available in [GYA, Thm. 5.3.6]. Note that the following statement is a generalization of the characterization of 2-connected graphs in terms of internally disjoint paths provided in Theorem 5.4.(ii).

Theorem 5.10 (Whitney's k-connected characterization)**.** *Let G be a connected, simple, unoriented and finite graph, and let $k \in \mathbb{N}$. Then G is k-connected if and only if for any pair x, y of vertices of G there exist at least k internally disjoint path between x and y.*

Let us still mention in this section that there exist analogues of Menger's theorem and its consequences in terms of edges instead of vertices. More precisely, the notion of edges disjoint paths can be introduced, and separator can be expressed in terms of edges instead of vertices. Then, an edge form of Menger's theorem can be formulated, and a statement about edge connectivity holds as well.

5.3 Blocks and block-cutpoint graphs

The decomposition of a graph into blocks reveals its coarse structure, its skeleton. After this decomposition, a bipartite tree can then be constructed, which encodes the main structure of the graph. Note that this decompositions holds for undirected graphs. We also recall that all graphs in this chapter are considered as loopless.

Recall that the notion of cut-vertex has been introduced in Definition 2.19.

Definition 5.11. *A* block *of an undirected graph G is a maximal connected subgraph which does not contain any cut-vertex (any cut-vertex of the subgraph, but it can contain a cut-vertex of the graph G).*

Recall that the notion of *maximal* means that there is not a larger structure with the same properties. As a consequence of this definition, a block is either a maximal 2-connected subgraph containing at least three vertices, or a *dipole*, or an isolated vertex. A dipole consists in two vertices connected by one or several edges. Note that for simple graphs, these dipoles are often called *bridges with their endpoints* in the literature. Some properties of blocks can be easily deduced, and we refer to [Die, Sec. 3.2] or to [GYA, Sec. 5.4] for the proofs.

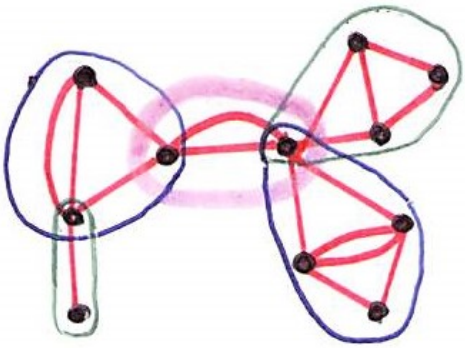


Figure 5.7: 1 graph, 5 blocks

Lemma 5.12. *Let G be an undirected and loopless graph.*

(i) Two blocks of G can overlap in at most one vertex, which is then a cut-vertex of G,

(ii) Every edge of G lies in a unique block,

(iii) Cycles of G are confined in blocks.

Based on the above property, a bipartite tree can be constructed. It reflects the structure of the initial graph. The construction is called *the block-cutpoint graph* $\mathrm{BC}(G)$ of G and goes as follows: Let $G = (V, E)$ be the initial graph, and let $\mathrm{BC}(G) = (W, F)$ be the block-cutpoint graph. The bipartition W_1, W_2 of W is defined by: each vertex of W_1 corresponds to a block of G, each vertex of W_2 corresponds to a cut-vertex of G. An element of W_2 is connected to an element of W_1 if the corresponding cut-vertex belongs to the corresponding block. It is then easy to check that the resulting bipartite graph is also a tree, see Figure 5.8.

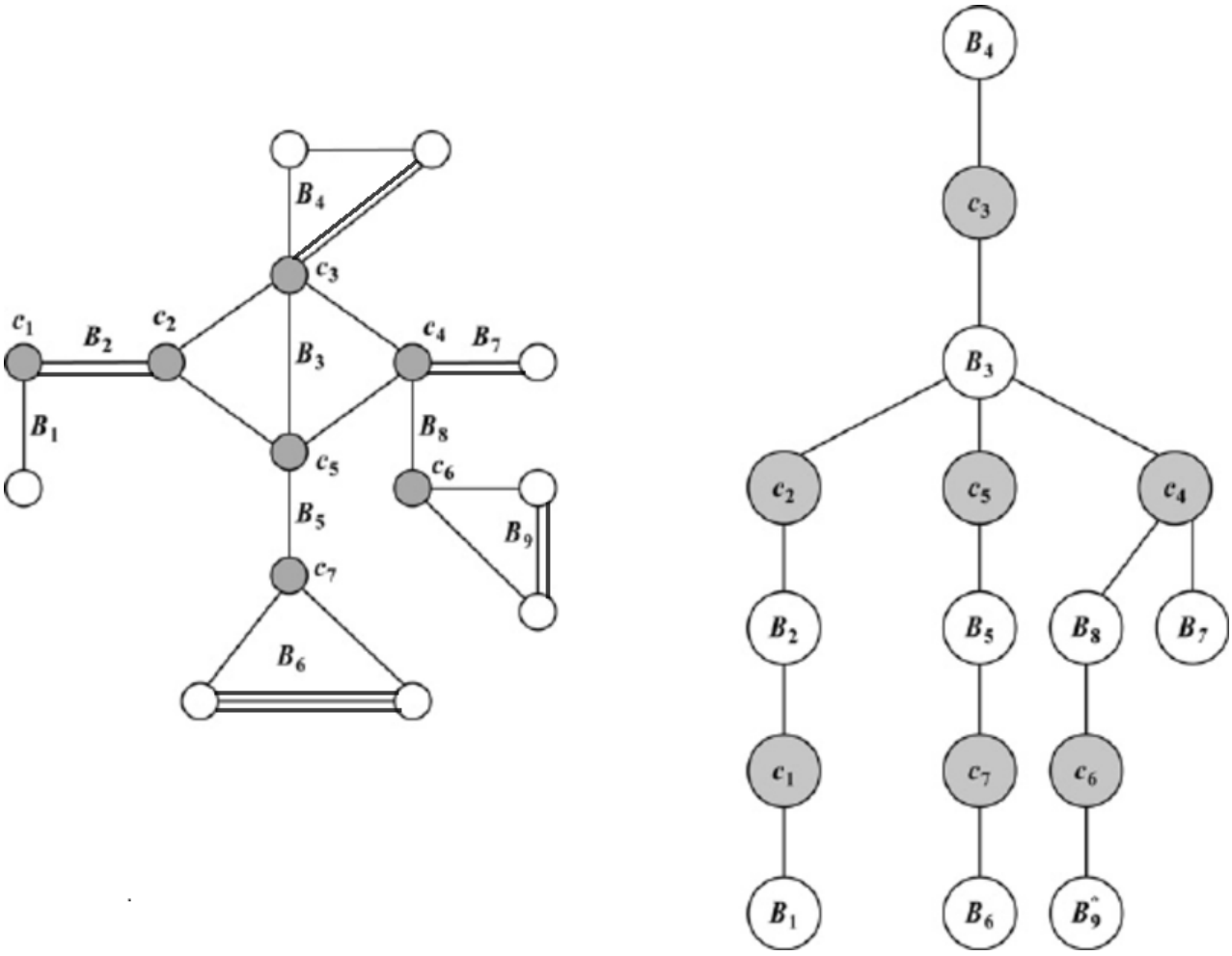


Figure 5.8: A connected graph and its block-cutpoint graph

Bibliography

[BG] S. Baase, A. van Gelder, *Computer algorithms, Introduction to design and analysis*, Addison-Wesley, 2000.

[CH] J. Clark, D.A. Holton, *A first look at graph theory*, Wold Scientific, 1991.

[Die] R. Diestel, *Graph theory*, Fifth edition, Springer, 2017.

[GYA] J.L. Gross, J. Yellen, M. Anderson, *Graph theory and its applications*, CRC press, 2019.

[KMS] I. Kiss, J. Miller, P. Simon, *Mathematics of epidemics on networks*, Springer, 2017.

[Ma] B. Maurer, The King Chicken Theorems, Mathematics Magazine Vol. 53, (1980), pp. 67-80.

[Mo] J.W. Moon, Topics on tournaments, Holt, Rinehart and Winston, Inc., 1968.

[Ne] M. Newman, Networks, second edition, Oxford University Press, 2018.

[1] https://en.wikipedia.org/wiki/Permutation_group

[2] https://proofwiki.org/wiki/Definition:Adjacency_Matrix

[3] https://en.wikipedia.org/wiki/Directed_acyclic_graph

[4] https://thespectrumofriemannium.wordpress.com/tag/homeomorphically-irreducible-tree/

[5] https://en.wikipedia.org/wiki/Decision_tree

[6] https://en.wikipedia.org/wiki/Tree_traversal

[7] https://en.wikipedia.org/wiki/Binary_expression_tree

[8] https://en.wikipedia.org/wiki/Binary_search_tree

[9] https://en.wikipedia.org/wiki/Catalan_number

[10] https://en.wikipedia.org/wiki/Cayley's_formula

[11] https://en.wikipedia.org/wiki/Steiner_tree_proble

[12] https://en.wikipedia.org/wiki/Dijkstra's_algorithm

[13] https://steemit.com/popularscience/@krishtopa/dijkstra-s-algorithm-of-finding-optimal-paths

[14] https://en.wikipedia.org/wiki/Menger's_theorem