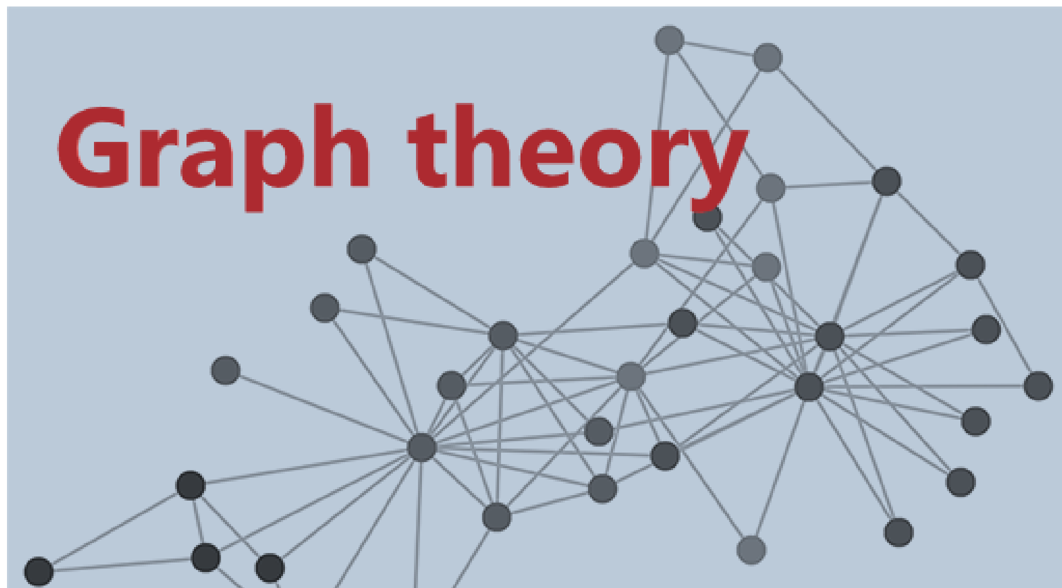
Special math lectures



Spring Semester 2020

S. Richard

Teaching assistant: N. Tsuzu

Contents

1 The basics **7**
- 1.1 Graphs . . . 7
- 1.2 Walks and paths . . . 10
- 1.3 Cycles . . . 14
- 1.4 Weighted graphs . . . 15
- 1.5 Appendix . . . 16
  - 1.5.1 Strongly connected oriented graph and bipartiteness . . . 16
  - 1.5.2 Travelling Salesman Problem . . . 18
  - 1.5.3 $\delta(G)$, $\Delta(G)$, $\mathrm{rad}(G)$, $\mathrm{diam}(G)$, $\mathrm{girth}(G)$, and all that . . . 23

2 Representations and structures **25**
- 2.1 Matrix representations . . . 25
- 2.2 Isomorphisms . . . 29
- 2.3 Automorphisms and symmetries . . . 32
- 2.4 Subgraphs . . . 35

3 Trees **39**
- 3.1 Trees and forests . . . 39
- 3.2 Rooted trees . . . 41
- 3.3 Traversals in binary trees . . . 46
- 3.4 Applications . . . 47
  - 3.4.1 Arithmetic expression trees . . . 47
  - 3.4.2 Binary search trees . . . 48
  - 3.4.3 Huffman trees . . . 49
  - 3.4.4 Priority trees . . . 53
- 3.5 Counting binary trees . . . 56
- 3.6 Appendix . . . 59
  - 3.6.1 Operations on binary search trees . . . 59
  - 3.6.2 An Improved Inserting Algorithm to Binary Search Trees . . . 62

4 Spanning trees **69**
- 4.1 Spanning trees and their growth . . . 69
- 4.2 Depth-first and breadth-first search . . . 72

4.3 Applications of DFS 75
4.4 Minimum spanning trees and shortest paths 78
4.5 Appendix 81
4.5.1 A few problems on spanning trees 81
4.5.2 Greedy algorithm 83
4.5.3 Application of graph theory in route search algorithm for route guidance system in automobiles 86
4.5.4 Floyd—Warshall algorithm 95

5 Connectivity **99**
5.1 Vertex and edge connectivity 99
5.2 Menger's theorem 101
5.3 Blocks and block-cutpoint graphs 104
5.4 Appendix 106
5.4.1 Some inequalities 106

6 Optimal traversals **107**
6.1 Eulerian trails 107
6.2 Postman tour 108
6.3 Hamiltonian paths and cycles 111
6.4 The traveling salesman problem 113

7 Graph colorings **119**
7.1 Vertex-colorings 119
7.2 Plane graphs 124
7.3 Map-colorings 127
7.4 Appendix 129
7.4.1 The five color theorem 129
7.4.2 Some problems related to plane graphs 133

8 Directed graphs **137**
8.1 Strongly connected components 137
8.2 Tournaments 143
8.3 Project scheduling 146

9 Flows **151**
9.1 Capacity, flows and cuts 151
9.2 Maximum flow problem 154
9.3 Applications 157
9.3.1 Flow and Menger's theorem 157
9.3.2 Matching 158
9.3.3 Transversals 160
9.4 Appendix 161
9.4.1 Hall's marriage theorem 161

10 Random graphs: the $\mathcal{G}(n,p)$ model **165**
10.1 Basic results 165
10.2 Components 169
10.3 Clustering coefficient and path lengths 173
10.4 Weaknesses 175

11 The configuration model **179**
11.1 Construction, and basic properties 179
11.2 Additional properties 182
11.3 Community structure, or modularity 186

12 Epidemics on graphs **191**
12.1 Basic models 191
12.1.1 The **SI**-model 191
12.1.2 The **SIR**-model 192
12.1.3 Other models 195
12.2 Percolation 195
12.3 Epidemic on graphs and percolation 198
12.4 Time dependent evolution 199

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*.*

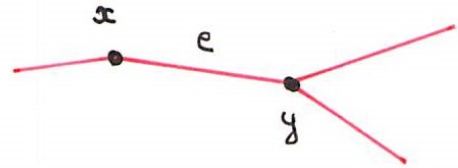


Fig. 1.1. One edge, two vertices

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 $i(e) = \big(\mathrm{o}(e), \mathrm{t}(e)\big)$, 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 $e_1, \ldots, e_n$ with $i(e_j) = (x, y)$ for all $j \in \{1, \ldots, n\}$, 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

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

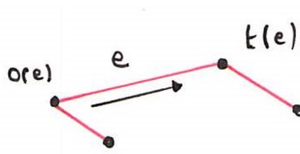


Fig. 1.2. Oriented edge

Fig. 1.3. Loop

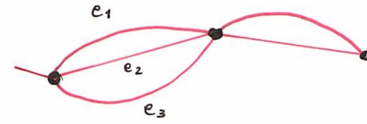


Fig. 1.4. Multiple edges

(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).

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.

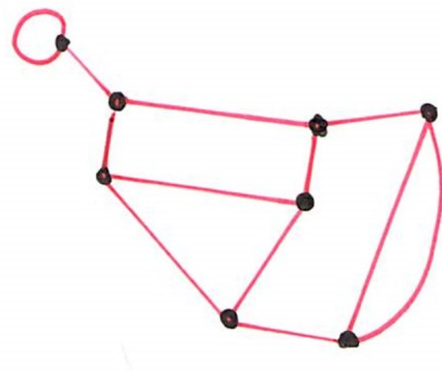


Fig. 1.5. A 3-regular graph

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.

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 $i' = i$ 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 $(V, E \setminus F)$.

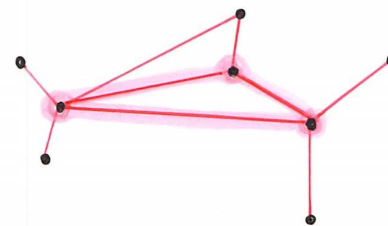


Fig. 1.6. An induced graph in pink

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)**.** *A 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.

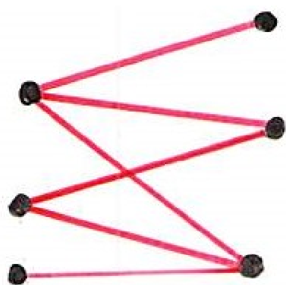
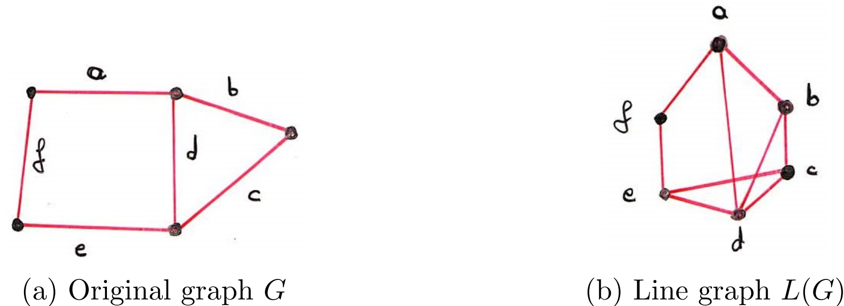


Fig. 1.7. Bipartite graph

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 $L(G)$

Fig. 1.8. Graph and its line graph

Note taht there exist 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 $G = (V, E)$ consists of two sequences $(x_j)_{j=0}^N \subset V$ and $(e_j)_{j=1}^N \subset E$ with $i(e_j) = (x_{j-1}, x_j)$. We write $W = \big((x_j)_{j=0}^N, (e_j)_{j=1}^N\big)$ for such a walk.*

Note that another presentation for a walk is the following:

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

with the requirement that $i(e_j) = (x_{j-1}, x_j)$ for $j \in \{1, \ldots, N\}$, see also Figure 1.9.

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 $(e_j)_{j=1}^N$ is therefore not necessary.*

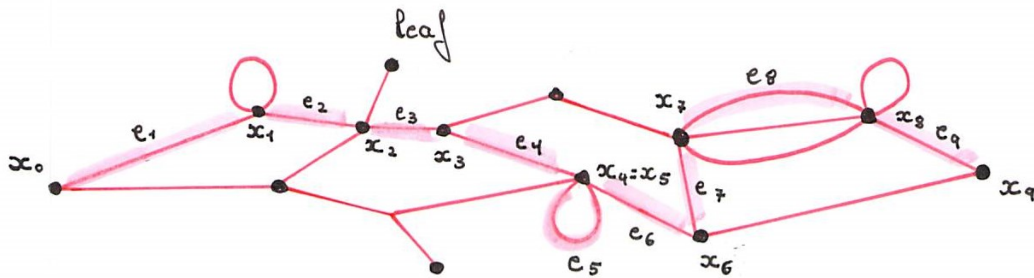
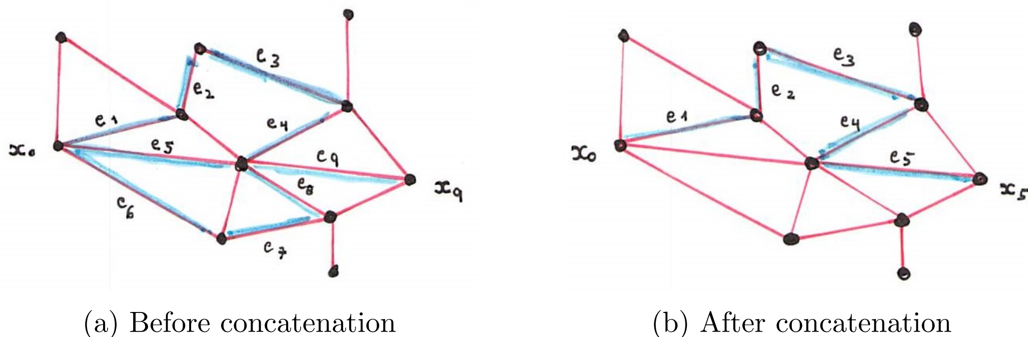


Fig. 1.9. One walk with one loop included

For $k \in \{1,2\}$, consider two walks $W_k = \big((x_j^k)_{j=0}^{N_k}, (e_j^k)_{j=1}^{N_k}\big)$. 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

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

This new walk is of length $N_1 + N_2$. One walk can also be concatenated: Consider $W = \big((x_j)_{j=0}^N, (e_j)_{j=1}^N\big)$ and suppose that $x_j = x_k$ for some $0 \leq j < k \leq N$. Then one concatenated walk consists in removing $x_{j+1}, \dots, x_k$ and $e_{j+1}, \dots, e_k$ to the 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



(a) Before concatenation (b) After concatenation

Fig. 1.10. One concatenation of a walk

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 !

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.*

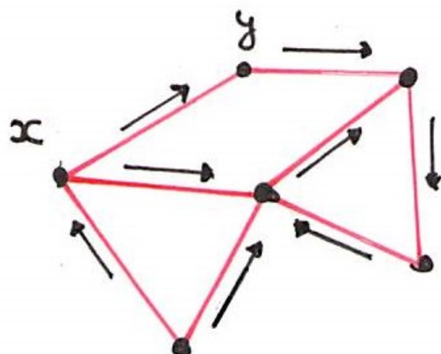


Fig. 1.11. $\mathrm{d}(x, y) = 1$ but $\mathrm{d}(y, x) = \infty$

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.

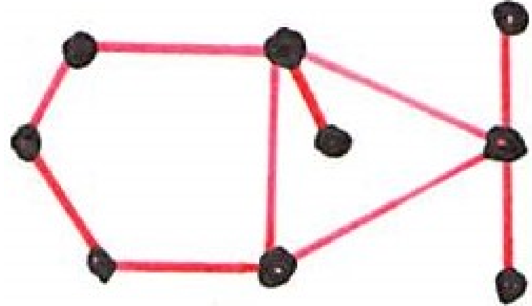


Fig. 1.12. $\mathrm{diam}(G) = 4$, $\mathrm{rad}(G) = 2$

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.*

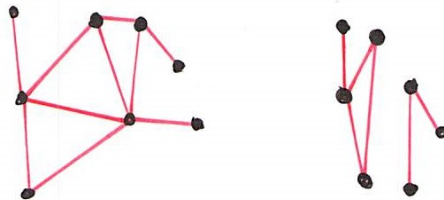


Fig. 1.13. One connected graph, one not connected graph

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.*

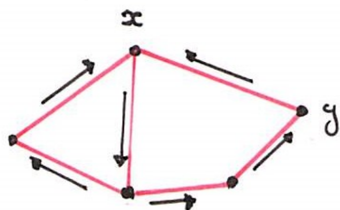


Fig. 1.14. Strongly connected graph

Observe that for strongly connected graphs, the distances $\mathrm{d}(x, y)$ and $\mathrm{d}(y, x)$ 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 in Chapter 3. 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* $\mathrm{girth}(G)$*. 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. *A 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 (provided in the appendix of this section) has been provided by Chang Sun.

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.*

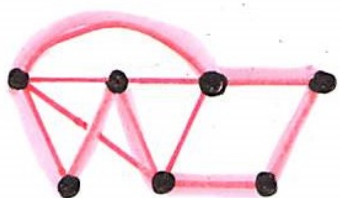


Fig. 1.15. A Hamiltonian graph

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 existience means that there exists a closed path visiting all vertices once. Hamilltonian cycles are also related to the famous *travelling salesman problem*, see Section 1.5.2. 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].

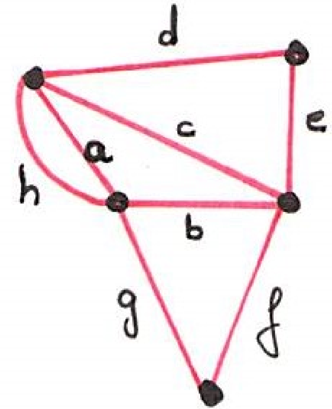


Fig. 1.16. Eulerian graph

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. A proof is provided in Section 1.5.3. 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 will depend on the purpose.

1.5 Appendix

1.5.1 Strongly connected oriented graph and bipartiteness

Proof of Theorem 1.22. Necessity ($\Rightarrow$): Suppose G is bipartite, then each step in a walk switches between the two bipartitions. Any closed walk (if exists) requires the walk to end on the same side as it started, and this forces the total number of steps to be even.

Sufficiency ($\Leftarrow$): Let G be a graph with at least two vertices and no cycle of odd length. Choose arbitrary $x, y \in V$. Let L_1 be a shortest $y \to x$ path, and let L_2 be a shortest $x \to y$ path. Let a_1 be the first vertex common to both paths from y (go reversely with path L_2), and let a_{k+1} be the first vertex common to both paths from a_k until one reaches $a_n = x$. Let C_k be the cycles obtained by sections of L_1 and L_2 between a_k and a_{k-1}, see Figure 1.17.

Clearly, the length of the composition of L_1 and L_2 is $\mathrm{d}(y, x) + \mathrm{d}(x, y) = \sum_{k=1}^{n} c_k$ with c_k being the length of C_k. Given that there exists no cycle with odd length, all c_k are even. Thus $\mathrm{d}(y, x) + \mathrm{d}(x, y)$ is even and therefore $\mathrm{d}(y, x) = \mathrm{d}(x, y)$ modulo 2.

Let us now pick a vertex u from G, and define a partition (X_0, X_1) of V as follows

$$
\begin{aligned}
X_0 &:= \{x \in V \mid \mathrm{d}(u, x) = 0 \pmod 2\} = \{x \in V \mid \mathrm{d}(x, u) = 0 \pmod 2\} \\
X_1 &:= \{x \in V \mid \mathrm{d}(u, x) = 1 \pmod 2\} = \{x \in V \mid \mathrm{d}(x, u) = 1 \pmod 2\}.
\end{aligned}
$$

It remains to show that this partition defines a bipartition for G.

If (X_0, X_1) is not a bipartition of G, then there are two vertices in one of the sets, say v and w that are joined by an edge. We denote this edge by e, and assume with no

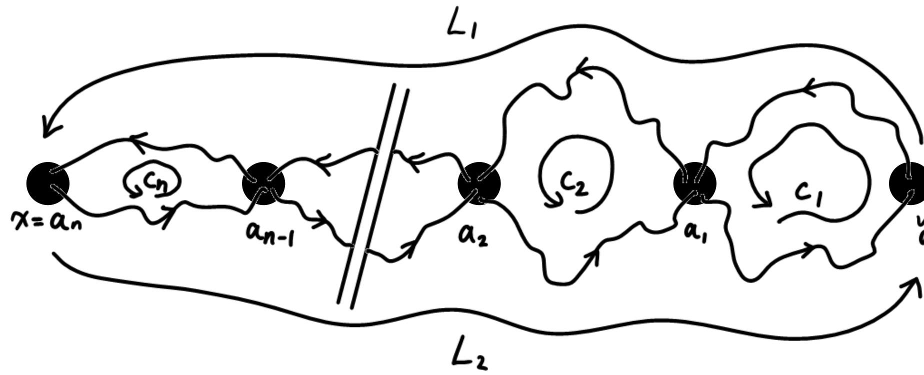


Fig. 1.17. Construction for the sufficiency part

loss of generality that this edge starts at v and ends at w. Let P_1 be a shortest $u \to v$ path, and let P_2 be a shortest $w \to u$ path. By definition of the sets X_0 and X_1, the length of these paths are both even or both odd. Starting from vertex u, let z be the last vertex common to both paths, see Figure 1.18.

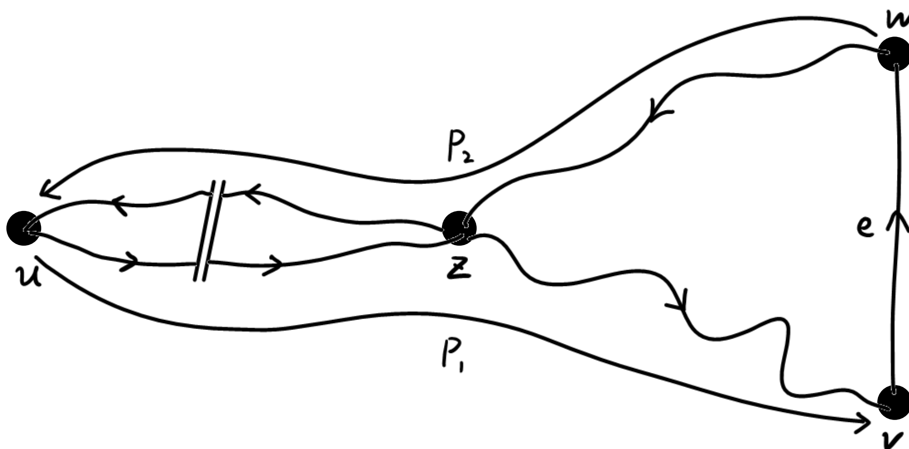


Fig. 1.18. For the proof of the bipartition

As both $\mathrm{d}(w,u)$ and $\mathrm{d}(u,v)$ are odd or even, and $\mathrm{d}(u,z) = \mathrm{d}(z,u) \pmod 2$, one has

$$
\begin{aligned}
&\mathrm{d}(w,z) + \mathrm{d}(z,v) \\
=&\mathrm{d}(w,u) - \mathrm{d}(z,u) + \mathrm{d}(u,v) - \mathrm{d}(u,z) \pmod 2 \\
=&\big(\mathrm{d}(w,u) + \mathrm{d}(u,v)\big) - \big(\mathrm{d}(u,z) + \mathrm{d}(z,u)\big) \pmod 2 \\
=&0 \pmod 2
\end{aligned}
$$

Therefore, the cycle passing from w to z on P_2, then from z to v on P_1, and finally

from v to u by e, is of length $\mathrm{d}(w,z)+\mathrm{d}(z,v)+1=1 \pmod 2$. It is thus a cycle of odd length, which contradicts the assumption. Thus, (X_0, X_1) is a bipartition for G. □

1.5.2 Travelling Salesman Problem

The material of this section has been studied and written by Quang Nhat Nguyen.

Definition 1.29 (Travelling Salesman Problem)**.** *Let $G = (V, E, \omega)$ be a* weighted *Hamiltonian graph, with $V = \{x_1, x_2, \dots, x_n\}$, $i : E \to V \times V$ determines the edges, and $\omega : E \to \mathbb{R}_+$ determines the weighted edge lengths. The problem of finding its shortest Hamiltonian cycle is called the Travelling Salesman Problem (abbrv. TSP).*

Note: *If G is undirected, the TSP is called* symmetric TSP *(sTSP). If G is directed, the TSP is called* asymmetric TSP *(aTSP).*

This is one of the classical problems in Computer Science, and is an introductory problem to Dynamic Programming. The need of computer power arises when we consider a large graph (eg. one that contains hundreds or thousands of vertices) and want to find its shortest Hamiltonian cycle. Before discussing the approaches, let us introduce the *distance matrix.*

Definition 1.30 (Distance matrix)**.** *The $n \times n$ matrix D with elements*

$$D_{ij} := \min\{\omega(e) \mid e \in E,\ i(e) = (x_i, x_j)\} \quad \text{if there exists such } e$$
$$D_{ij} := \infty \quad \text{if there is no such } e$$

is called the distance matrix *of the weighted graph G.*

In simpler terms, the element D_{ij} denotes the shortest edge that originates at x_i and terminates at x_j. It can be seen that if the graph G is undirected, the matrix D is symmetric. If there is no edge from x_i to x_j, $D_{ij} = \infty$. If there are multiple edges from x_i to x_j, we only consider the shortest edge because our concern is the shortest Hamiltonian cycle.

A naïve approach to this problem would be for the computer to consider all permutations of the vertices, see if one permutation can make a cycle, and if it does then record the cycle's length for comparison. This method has a time complexity of $O(n!)$, and thus is not desirable when dealing with a large database.

Proposition 1.31. *One can reduce the time complexity of $O(n!)$ of the naïve method to $O(n^2 \times 2^n)$ by implementing Bellman-Held-Karp Algorithm.*

In the sequel, we study the Bellman-Held-Karp Algorithm. First, let us acknowledge the following.

Proposition 1.32. *The shortest Hamiltonian cycle does not depend on the choice of the starting vertex.*

This is obvious because the Hamiltonian cycle has cyclic symmetry, thus changing the starting vertex does not change the order of the other vertices in the cycle. Therefore, let us start constructing the desired path from x_1.

Definition 1.33. *Let $S \subset V\backslash\{x_1\} \equiv \{x_2, x_3, \dots, x_n\}$ be a subset of size $s\,(1 \leq s \leq n-1)$. For each vertex $x_i \in S$, we define $\text{cost}(x_i, S)$ as the length of the shortest path from x_1 to x_i which travels to each of the remaining vertices in S once. More precisely, we implement this function by the following recursion formula:*

$$\text{cost}(x_i, S) = \min_{x_j}\{\text{cost}(x_j, S\backslash\{x_i\}) + D_{ji}\} \tag{1.1}$$

in which $x_j \in S\backslash\{x_i\}$. In case S has size 1, we define:

$$\text{cost}(x_i, S) = D_{1i} \tag{1.2}$$

Using the above recursive formula, we can gradually construct the cost function for subset S of size from 1 to $n-1$. Because of this recursive feature, which means that the current step is the base for the next step, the implementation of this algorithm in a programming language is called Dynamic Programming.

When we reach subset S of size $n-1$, which means $S = V\backslash\{x_1\}$, the only thing left is to find:

$$\begin{aligned}\text{Shortest Hamiltonian cycle} &= \min_{x_i}\{\text{cost}(x_i, S) + D_{i1}\} \\ &\equiv \min_{x_i}\{\text{cost}(x_i, V\backslash\{x_1\}) + D_{i1}\}\end{aligned}$$

with $x_i \in S \equiv V\backslash\{x_1\}$.

Proposition 1.34 (Time complexity of this algorithm)**.** *The time complexity of this algorithm is: $O(n^2 \times 2^n)$.*

Proof. This algorithm considers 2^{n-1} subsets of $V\backslash\{x_1\}$. For each subset, one computes the cost function for all of its elements, which there are no more than n of them. For each execution of the cost function, one again computes no more than n values based on the values obtained from the previous step. In total, the time complexity of this algorithm is: $O(n^2 \times 2^n)$. □

Remark 1.35. *Compared to the naïve method which has time complexity $O(n!)$, the time complexity of this algorithm, $O(n^2 \times 2^n)$, is much better. For example, if $n = 100$:*

$$O(n!) = O(100!) = O(9.33 \times 10^{157})$$
$$O(n^2 \times 2^n) = O(100^2 \times 2^{100}) = O(1.27 \times 10^{34})$$

which is about 7.35×10^{123}, or 0.7 septillion googols, times faster.

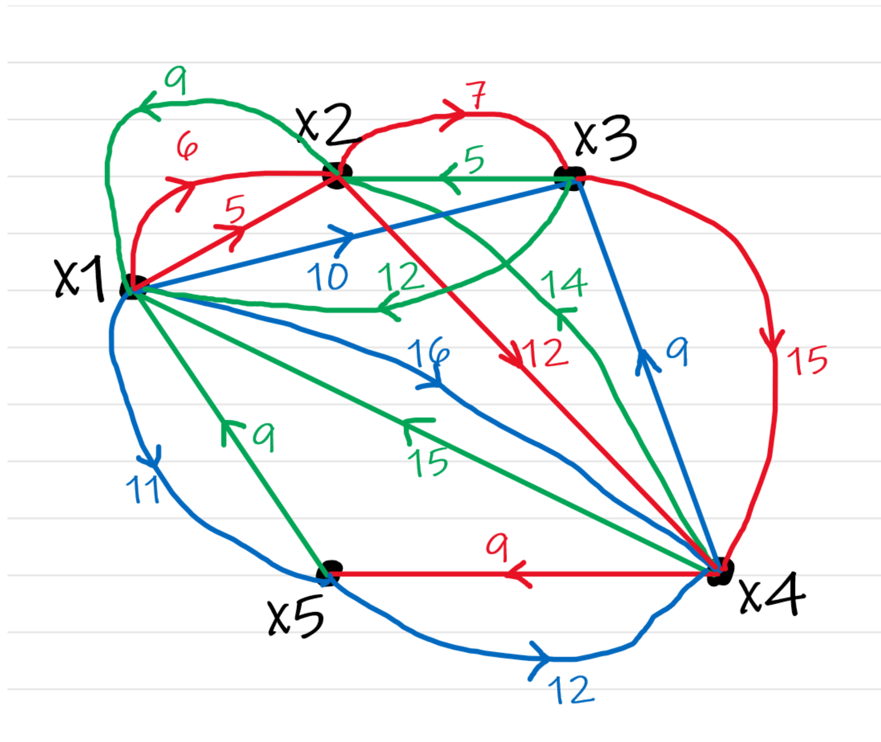


Fig. 1.19. Graph for a sample asymmetric TSP

Let us now illustrate the algorithm through the following sample problem. The distance matrix can be generated as follows:

$$\begin{bmatrix} 0 & 5 & 10 & 16 & 11 \\ 9 & 0 & 7 & 12 & \infty \\ 12 & 5 & 0 & 15 & \infty \\ 15 & 14 & 9 & 0 & 9 \\ 9 & \infty & \infty & 12 & 0 \end{bmatrix}$$

Now that we have the distance matrix, we can proceed with the recursive formula. First, let us consider subsets S of size 1:

Subset S	cost function	Result
$S = \{x_2\}$	$\text{cost}(x_2, \{x_2\}) = D_{12}$	5
$S = \{x_3\}$	$\text{cost}(x_3, \{x_3\}) = D_{13}$	10
$S = \{x_4\}$	$\text{cost}(x_4, \{x_4\}) = D_{14}$	16
$S = \{x_5\}$	$\text{cost}(x_5, \{x_5\}) = D_{15}$	11

Table 1.1: Process for subsets S of size 1

Next we consider subsets S of size 2 as follows:

Subset S	cost function	Result
$S=\{x_2,x_3\}$	$\text{cost}(x_2,\{x_2,x_3\})=\min\{\text{cost}(x_3,\{x_3\})+D_{32}\}=10+5$	15
	$\text{cost}(x_3,\{x_2,x_3\})=\min\{\text{cost}(x_2,\{x_2\})+D_{23}\}=5+7$	12
$S=\{x_2,x_4\}$	$\text{cost}(x_2,\{x_2,x_4\})=\min\{\text{cost}(x_4,\{x_4\})+D_{42}\}=16+14$	30
	$\text{cost}(x_4,\{x_2,x_4\})=\min\{\text{cost}(x_2,\{x_2\})+D_{24}\}=5+12$	17
$S=\{x_2,x_5\}$	$\text{cost}(x_2,\{x_2,x_5\})=\min\{\text{cost}(x_5,\{x_5\})+D_{52}\}=11+\infty$	∞
	$\text{cost}(x_5,\{x_2,x_5\})=\min\{\text{cost}(x_2,\{x_2\})+D_{25}\}=5+\infty$	∞
$S=\{x_3,x_4\}$	$\text{cost}(x_3,\{x_3,x_4\})=\min\{\text{cost}(x_4,\{x_4\})+D_{43}\}=16+9$	25
	$\text{cost}(x_4,\{x_3,x_4\})=\min\{\text{cost}(x_3,\{x_3\})+D_{34}\}=10+15$	25
$S=\{x_3,x_5\}$	$\text{cost}(x_3,\{x_3,x_5\})=\min\{\text{cost}(x_5,\{x_5\})+D_{53}\}=11+\infty$	∞
	$\text{cost}(x_5,\{x_3,x_5\})=\min\{\text{cost}(x_3,\{x_3\})+D_{35}\}=10+\infty$	∞
$S=\{x_4,x_5\}$	$\text{cost}(x_4,\{x_4,x_5\})=\min\{\text{cost}(x_5,\{x_5\})+D_{54}\}=11+12$	23
	$\text{cost}(x_5,\{x_4,x_5\})=\min\{\text{cost}(x_4,\{x_4\})+D_{45}\}=16+9$	25

Table 1.2: Process for subsets S of size 2

For the subsets S of size 3, we have the following table:

Subset S	cost function	Result
$S=\{x_2,x_3,x_4\}$	$\text{cost}(x_2,\{x_2,x_3,x_4\})=\min\{25+5,25+14\}$	30
	$\text{cost}(x_3,\{x_2,x_3,x_4\})=\min\{30+7,17+9\}$	26
	$\text{cost}(x_4,\{x_2,x_3,x_4\})=\min\{15+12,12+15\}$	27
$S=\{x_2,x_3,x_5\}$	$\text{cost}(x_2,\{x_2,x_3,x_5\})=\min\{\infty,\infty\}$	∞
	$\text{cost}(x_3,\{x_2,x_3,x_5\})=\min\{\infty,\infty\}$	∞
	$\text{cost}(x_5,\{x_2,x_3,x_5\})=\min\{\infty,\infty\}$	∞
$S=\{x_2,x_4,x_5\}$	$\text{cost}(x_2,\{x_2,x_4,x_5\})=\min\{23+14,\infty\}$	37
	$\text{cost}(x_4,\{x_2,x_4,x_5\})=\min\{\infty,\infty\}$	∞
	$\text{cost}(x_5,\{x_2,x_4,x_5\})=\min\{\infty,17+9\}$	26
$S=\{x_3,x_4,x_5\}$	$\text{cost}(x_3,\{x_3,x_4,x_5\})=\min\{23+9,25+\infty\}$	32
	$\text{cost}(x_4,\{x_3,x_4,x_5\})=\min\{\infty,\infty\}$	∞
	$\text{cost}(x_5,\{x_3,x_4,x_5\})=\min\{25+\infty,25+9\}$	34

Table 1.3: Process for subsets S of size 3

Note: Exlanation for the first entry:

$$\begin{aligned}\text{cost}(x_2,\{x_2,x_3,x_4\}) &= \min\{\text{cost}(x_3,\{x_3,x_4\})+D_{32},\text{cost}(x_4,\{x_3,x_4\})+D_{42}\}\\ &= \min\{25+5,25+14\}=30\end{aligned}$$

For the subsets S of size 4, which means $S=\{x_2,x_3,x_4,x_5\}$, we have the following:

cost function	Evaluation	Result
$\text{cost}\{x_2, \{x_2, x_3, x_4, x_5\}\}$	$\min\{\text{cost}(x_3, \{x_3, x_4, x_5\}) + D_{32}, \text{cost}(x_4, \{x_3, x_4, x_5\}) + D_{42}, \text{cost}(x_5, \{x_3, x_4, x_5\}) + D_{52}, \} = \min\{32+5, \infty, \infty\}$	37
$\text{cost}\{x_3, \{x_2, x_3, x_4, x_5\}\}$	$\min\{\text{cost}(x_2, \{x_2, x_4, x_5\}) + D_{23}, \text{cost}(x_4, \{x_2, x_4, x_5\}) + D_{43}, \text{cost}(x_5, \{x_2, x_4, x_5\}) + D_{53}, \} = \min\{37+7, \infty, \infty\}$	44
$\text{cost}\{x_4, \{x_2, x_3, x_4, x_5\}\}$	$\min\{\text{cost}(x_2, \{x_2, x_3, x_5\}) + D_{24}, \text{cost}(x_3, \{x_2, x_3, x_5\}) + D_{34}, \text{cost}(x_5, \{x_2, x_3, x_5\}) + D_{54}, \} = \min\{\infty, \infty, \infty\}$	∞
$\text{cost}\{x_5, \{x_2, x_3, x_4, x_5\}\}$	$\min\{\text{cost}(x_2, \{x_2, x_3, x_4\}) + D_{25}, \text{cost}(x_3, \{x_2, x_3, x_4\}) + D_{35}, \text{cost}(x_4, \{x_2, x_3, x_4\}) + D_{45}, \} = \min\{\infty, \infty, 27+9\}$	36

Table 1.4: Process for subsets S of size 4

For the final step, we have:

$$\begin{aligned}\text{Shortest Hamiltonian cycle} &= \min\{\text{cost}\{x_2, \{x_2, x_3, x_4, x_5\}\} + D_{21},\\ &\qquad \text{cost}\{x_3, \{x_2, x_3, x_4, x_5\}\} + D_{31},\\ &\qquad \text{cost}\{x_4, \{x_2, x_3, x_4, x_5\}\} + D_{41},\\ &\qquad \text{cost}\{x_5, \{x_2, x_3, x_4, x_5\}\} + D_{51}\}\\ &= \min\{37 + 9, 44 + 12, \infty, 36 + 9\}\\ &= 45\end{aligned}$$

This result means that this graph has *at least* three Hamiltonian cycles, and the shortest one has length 45. To trace back the order of the vertices in the shortest one, one simply trace back the vertices x_i in the minimal cost functions $\text{cost}(x_i, S)$. In this sample problem, the back-tracing order will be:

$$x_1 \leftarrow x_5 \leftarrow x_4 \leftarrow x_2 \leftarrow x_3 \leftarrow x_1$$

However, this back-tracing order is also correct:

$$x_1 \leftarrow x_5 \leftarrow x_4 \leftarrow x_3 \leftarrow x_2 \leftarrow x_1$$

So, the shortest Hamiltonian cycle has the following forward-going order:

$$x_1 \rightarrow x_3 \rightarrow x_2 \rightarrow x_4 \rightarrow x_5 \rightarrow x_1$$

or:

$$x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5 \rightarrow x_1$$

One can easily look at Figure 1.19 and check that the above Hamiltonian cycles both have length 45. This concludes the illustration of Bellman-Held-Karp Algorithm, which is based on Dynamic Programming

1.5.3 $\delta(G)$, $\Delta(G)$, rad(G), diam(G), girth(G), and all that

The material of this section has been studied and written by Bui Tu Ha.

Proof of Theorem 1.26. Observe that since the graph G is simple and undirected, a path is uniquely defined by the enumeration of the successive vertices. In this proof, the description of the paths will be done accordingly.

(i) Let $\delta(G) = k$. Start at any vertex v_0. If $k \geq 1$ then v_0 is adjacent to some vertex v_1. The path (v_0, v_1) has length 1. If $k \geq 2$ then v_1 is adjacent to a vertex $v_2 \neq v_0$. The path (v_0, v_1, v_2) has length 2. If $k \geq 3$ then v_2 is adjacent to a vertex $v_3 \notin \{v_0, v_1\}$. The path (v_0, v_1, v_2, v_3) has length 3. We repeat this argument until we have chosen v_k adjacent to v_{k-1} and $v_k \notin \{v_0, v_1, ..., v_{k-2}\}$. As a result, G contains the path $(v_0, v_1, v_2, ..., v_k)$, whose length is $k = \delta(G)$.

Assume that the longest path in G is $M = (v_0, v_1, ..., v_{m-1}, v_m)$. Note that v_m cannot be adjacent to a vertex $x \notin \{v_0, v_1, ..., v_{m-1}\}$ because it would produce a path longer than M, a contradiction. Then we have v_m adjacent to at least k vertices belonging to the set $\{v_0, v_1, ..., v_{m-1}\}$. Consequently, at least one of these k vertices is in the set $H = \{v_0, v_1, ..., v_{m-k}\}$. When v_m is adjacent to one vertex in H, it produces a cycle of length $\geq k + 1$ (The equality holds for the case v_m and v_{m-k} are adjacent). Thus, if $\delta(G) \geq 2$ then G contains a cycle of length at least $\delta(G) + 1$.

(ii) Let C be the shortest cycle in G. By definition, the length of C is girth(G). Assume that girth$(G) \geq 2$diam$(G)+2$. Then, there exists two vertices x and y in C such that the two paths between x and y in C are the path of length diam$(G)+1$, called P_C, and the other path whose length is at least diam$(G)+1$. The distance between x and y in C is $\mathrm{d}_C(x,y) = \mathrm{diam}(G)+1$. Let $\mathrm{d}_G(x,y)$ be the distance between x and y in G and it corresponds to the shortest path P_G in G between these two vertices. Observe that $\mathrm{d}_G(x,y) \leq \mathrm{diam}(G)$ (based on the definition of diameter) while $\mathrm{d}_C(x,y) = \mathrm{diam}(G)+1$. As a result, P_G is not a subgraph of C. P_G and P_C have two independent vertices in common, namely z_1 and z_2, which satisfy the following condition: The path contained in P_G between z_1 and z_2 and the path contained in P_C between z_1 and z_2 do not have any vertex in common, except z_1 and z_2. In some graphs, it is possible to choose $z_1 \equiv x$ or $z_2 \equiv y$. However, it is not always true for any graph. For example, the graph shown in Figure 1.20 has z_1 and z_2 different from x and y.

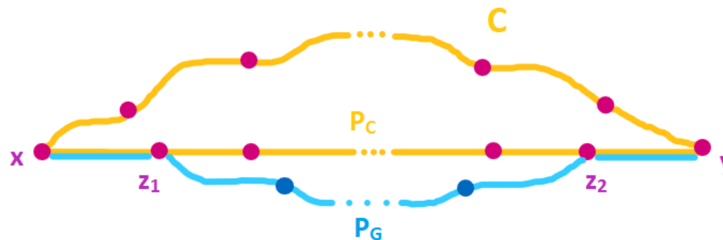


Fig. 1.20. A shorter cycle

The composition of the path contained in P_G between z_1 and z_2 and the path contained in P_C between z_1 and z_2 forms a cycle of length: $L \leq \mathrm{d}_G(x,y) + \mathrm{d}_C(x,y) = \mathrm{diam}(G) + \big((\mathrm{diam}(G)+1\big) = 2\mathrm{diam}(G)+1$. Thus, this cycle is even shorter than C, which is a contradiction. Therefore, the assumption is not correct and we have proved $\mathrm{girth}(G) \leq 2\mathrm{diam}(G)+1$. One example that the equality holds: G is a cycle having $(2n+1)$ vertices, then $\mathrm{diam}(G) = n$ and $\mathrm{girth}(G) = 2n+1 = 2\mathrm{diam}(G)+1$.

(iii) Let c be a central vertex of G. Let V_i be the set of vertices of G at distance i from c, with $i \in \{0,1,...,k\}$. Then $V(G) = \cup_0^k V_i$. By induction, we are going to prove that for any $i \in \{0,1,...,k\}$,

$$|V_{i+1}| \leq d(d-1)^i \tag{1.3}$$

It is clear that $|V_1| \leq d \Rightarrow |V_1| \leq d(d-1)^0$, which means that (1.3) is correct for $i=0$. For $i \in \{1,...,k-1\}$, we are going to show that if $|V_i| \leq d(d-1)^{i-1}$ is true then $|V_{i+1}| \leq d(d-1)^i$ is true. Indeed, observe that for $i \geq 1$, each vertex in V_i is connected to at most $(d-1)$ vertices in V_{i+1}, see Figure 1.21. As a consequence, one has

$$|V_{i+1}| \leq (d-1)\,|V_i| \leq (d-1)d(d-1)^{i-1} = d(d-1)^i.$$

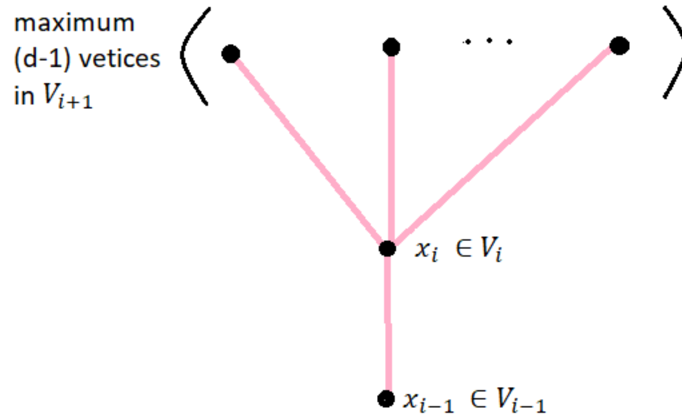


Fig. 1.21. Maximal number of adjacent vertices in V_{i+1}

Thus, for the number of vertices contained in G one has when $d \geq 3$:

$$|V(G)| = \left|\cup_0^k V_i\right| = 1 + \sum_{i=1}^{k} |V_i| \leq 1 + \sum_{i=0}^{k-1} d(d-1)^i = 1 + d\sum_{i=0}^{k-1}(d-1)^i$$
$$= 1 + d\frac{(d-1)^k - 1}{(d-1)-1} = 1 + \frac{d}{d-2}\big((d-1)^k - 1\big) = \frac{d}{d-2}(d-1)^k - \frac{2}{d-2}$$
$$< \frac{d}{d-2}(d-1)^k.$$

Therefore, G contains at most $\frac{d}{d-2}(d-1)^k$ vertices. $\square$

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.

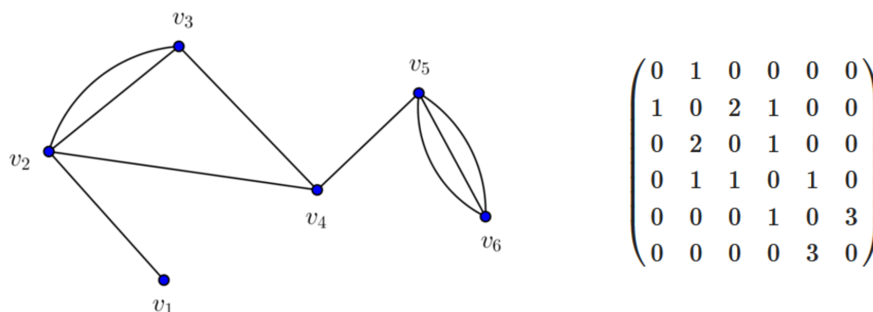
2.1 Matrix representations

Since linear algebra contains a large set of powerful tools, it is rather natural to use this theory for analyzing 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 $V = \{x_1, \dots, x_N\}$. 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.



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

Fig. 2.1. Adjacency matrix of an undirected graph

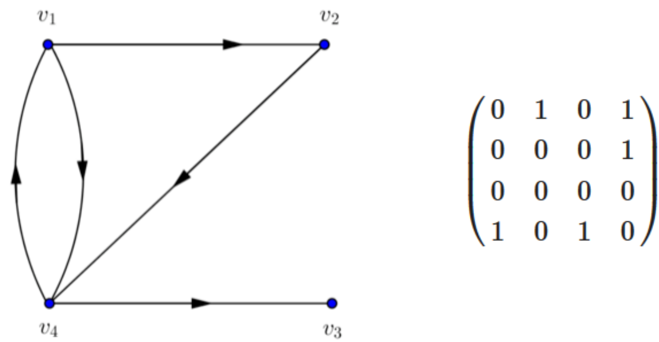


Fig. 2.2. Non symmetric adjacency matrix

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 . Note that this is not true in general for a directed graph. However, let us define the *indegree* and the *outdegree* of a vertex of a directed graph:

$$\deg_{\text{in}}(x) = \#\{e \in E \mid i(e) = (y, x) \text{ with y arbitrary}\} \tag{2.1}$$

and

$$\deg_{\text{out}}(x) = \#\{e \in E \mid i(e) = (x, y) \text{ with y arbitrary}\}. \tag{2.2}$$

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.

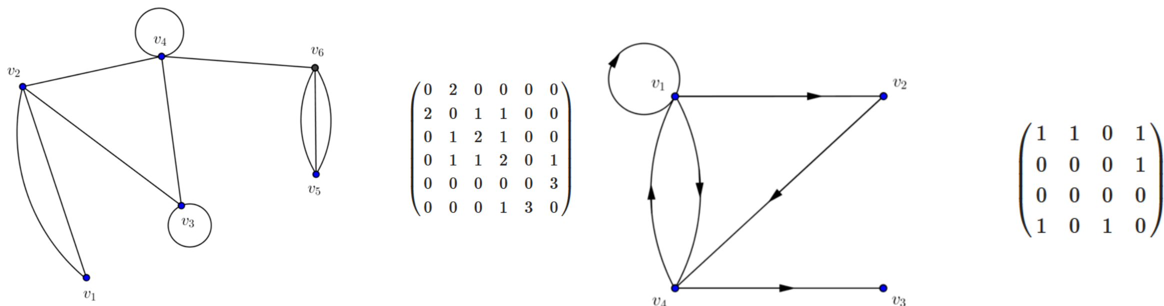


Fig. 2.3. Adjacency matrices in the presence of loops

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 $\big(A_G^r\big)_{jk}$ of the rth power of A_G is equal to the number of walks of length r from x_j to x_k.*

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

$$\big(A_G^{r+1}\big)_{jk} = \sum_{l=1}^{N} \big(A_G^r\big)_{jl} \big(A_G^1\big)_{lk}. \tag{2.3}$$

By the assumption and definitions mentioned above, $\big(A_G^r\big)_{jl}$ is the number of walks of length r from x_j to x_l, and $\big(A_G^1\big)_{lk}$ is the number of walks of length 1 from x_l to x_k. Hence, $\big(A_G^r\big)_{jl}\big(A_G^1\big)_{lk}$ 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 $\sum_l \big(A_G^r\big)_{jl}\big(A_G^1\big)_{lk}$ 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 (2.3) 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.

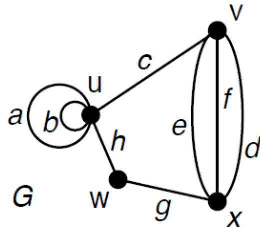
Definition 2.3 (Incidence matrix of an undirected graph)**.** *Let $G = (V, E)$ be a finite undirected graph with $V = \{x_1, \dots, x_N\}$ and $E = \{e_1, \dots, e_M\}$. The incidence matrix I_G of G consists in the $N \times M$ matrix with entries*

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

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:*

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



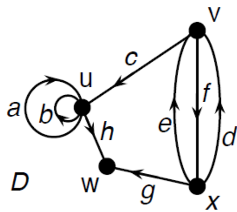
$$I_G = \begin{array}{c} \\ u \\ v \\ w \\ x \end{array} \begin{array}{c} \begin{array}{cccccccc} a & b & c & d & e & f & g & h \end{array} \\ \begin{pmatrix} 2 & 2 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}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

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

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 $V = \{x_1, \dots, x_N\}$ and $E = \{e_1, \dots, e_M\}$. The incidence matrix I_G of G consists in the $N \times M$ matrix with entries*

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



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

Fig. 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 $V = \{x_1, \dots, x_N\}$ and $E = \{e_1, \dots, e_M\}$. The incidence table $I_{V:E}(G)$ lists, for each vertex x_j, all edges e_ℓ having x_j as one endpoint.*

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 $V = \{x_1, \dots, x_N\}$ and $E = \{e_1, \dots, e_M\}$. 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).*

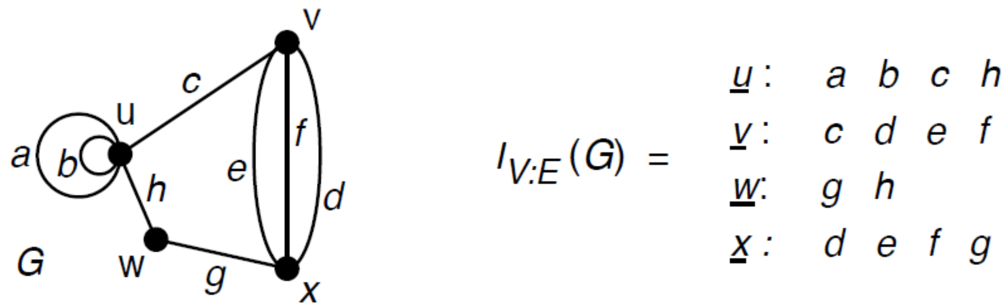


Fig. 2.6. Incidence table of an undirected graph, see Ex. 2.6.6 of [GYA]

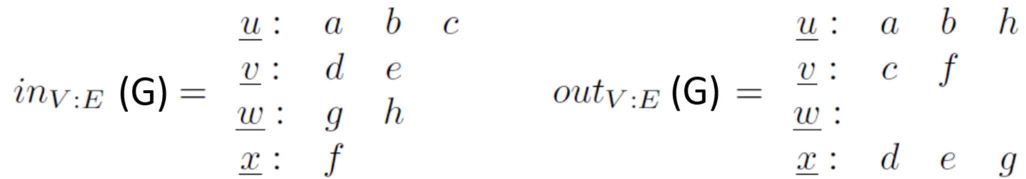


Fig. 2.7. Incidence tables for the graph of Figure 2.5, see Ex. 2.6.7 of [GYA]

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.

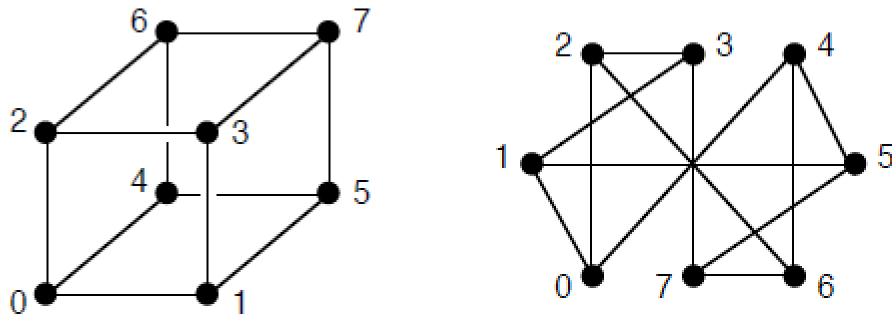


Fig. 2.8. Two representations of the same graph, see. Fig. 2.1.1 of [GYA]

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 $i'\big(f_E(e)\big) = \big(f_V(x), f_V(y)\big)$ in $V' \times V'$. Whenever such an isomorphism exists, we say that G and G' are isomorphic, and write $G \cong G'$.

Fig. 2.9. Isomorphism of a graph with loops and multiple edges

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 $\big(f_V(x), f_V(y)\big)$ and $\big(f_V(y), f_V(x)\big)$ 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 the second condition of Definition 2.8 is to say the following diagram is commutative:

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

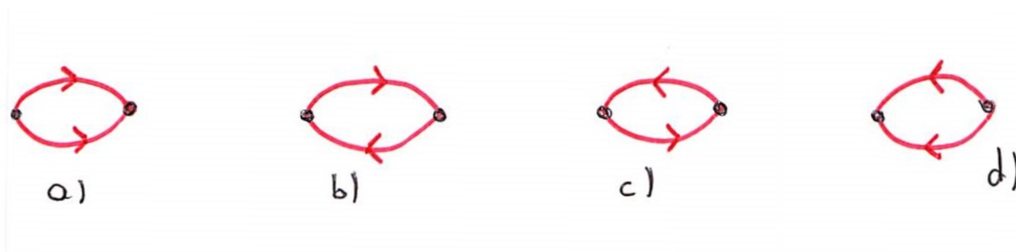


Fig. 2.10. a) and d) are isomorphic, b) and c) are isomorphic

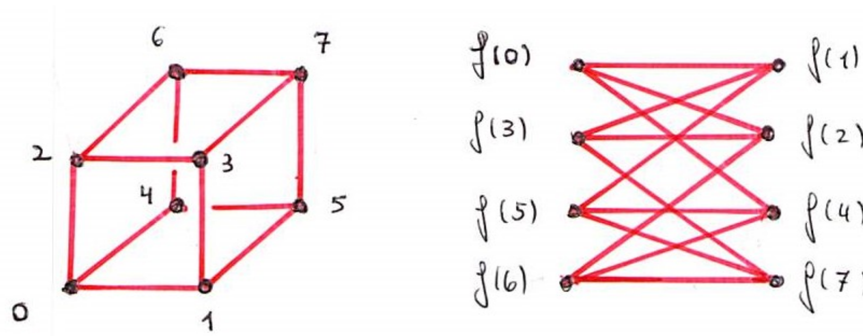


Fig. 2.11. Isomorphism of a simple graph

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. For simplicity, we shall also drop the indices V and E in f_V and f_E and write f for both functions. This should not lead to any confusion.

Proposition 2.11 (Graph invariants 1)**.** *Let $G = (V, E)$ and $G' = (V', E')$ be two graphs, and let $f : G \to G'$ be an isomorphism of 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 degree (see Definition 1.4), namely $\deg\big(f(x)\big) = \deg(x)$ for any $x \in V$,

(iii) The degrees of neighbours, namely

$$\{\deg(y) \mid y \in N(x)\} = \big\{\deg(y') \mid y' \in N\big(f(x)\big)\big\}$$

for any $x \in V$,

(iv) *The length of a walk, a trail or a path in* G *and their respective image in* G' *through* f *(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) and (2.2), respectively. Then, for directed graphs, a more precise version of (*ii*) and (*iii*) says that the following quantities are directed graphs invariants:

(ii') *The indegree and outdegree, namely* $\deg_{\text{in}}\big(f(x)\big) = \deg_{\text{in}}(x)$, *and* $\deg_{\text{out}}\big(f(x)\big) = \deg_{\text{out}}(x)$ *for any* $x \in V$,

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

$$\{\deg_{\text{in}}(y) \mid y \in N(x)\} = \big\{\deg_{\text{in}}(y') \mid y' \in N\big(f(x)\big)\big\}$$

and

$$\{\deg_{\text{out}}(y) \mid y \in N(x)\} = \big\{\deg_{\text{out}}(y') \mid y' \in N\big(f(x)\big)\big\}$$

for any $x \in V$.

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. Before looking at one concrete example, let us mention that symmetries are often related to permutations.

Figure 2.12 contains three representations of the same 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.

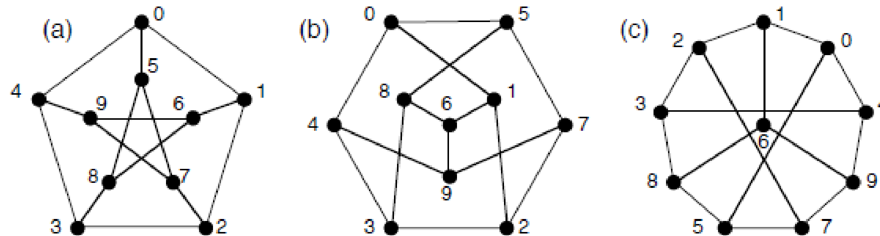


Fig. 2.12. Three representations of the Petersen graph, see. Fig. 2.2.3 of [GYA]

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

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

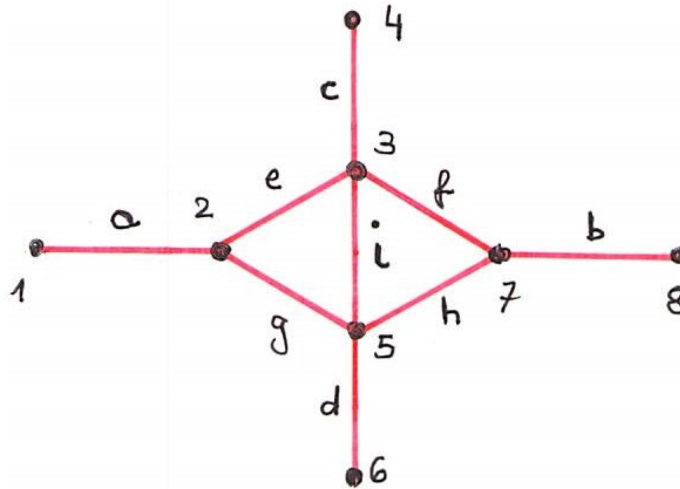


Fig. 2.13. A graph with several symmetries

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.

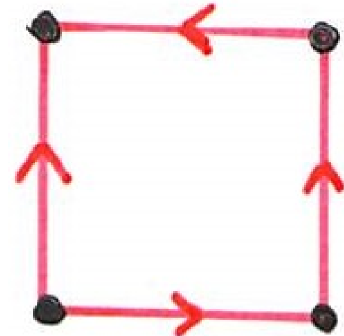


Fig. 2.14. No symmetry

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.

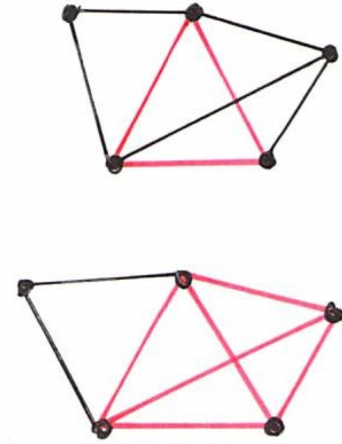


Fig. 2.15. 1 clique, 1 maximal clique

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.*

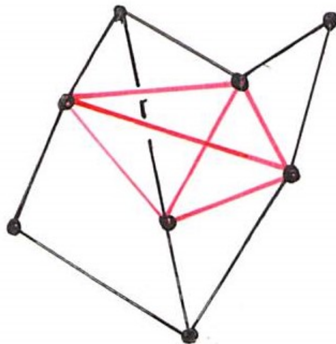


Fig. 2.16. $w(G) = 4$

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.*

As before, there is no uniqueness for the largest independent set in G, but the independence number is uniquely defined. 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

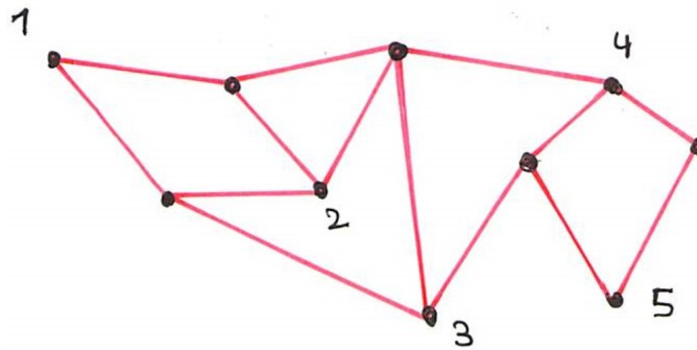


Fig. 2.17. Independence number: $\alpha(G) = 5$

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.*

Fig. 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 $c(G)$.

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.*

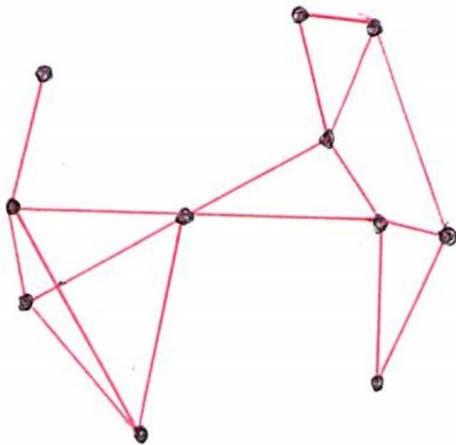


Fig. 2.19. Graph with two cut-vertices

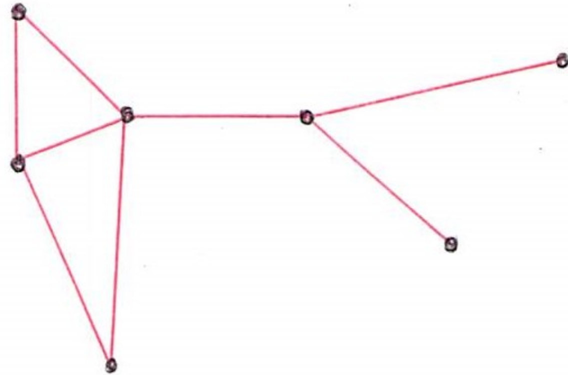


Fig. 2.20. Graph with three cut-edges

(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.

Proposition 2.21 (Graph invariants 2)**.** *Let $G = (V, E)$ and $G' = (V', E')$ be two graphs, and let $f : G \to G'$ be an isomorphism of graph. The following quantities are graph invariants:*

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

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

(iii) The number of distinct cutpoints or bridges.

More generally, if G possesses n distinct subgraphs having a certain property, then G' has to possess n distinct subgraphs having the same property.

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. Indeed, the notion of connected graph is based on the underlying graph, and the acyclicity property is also imposed on the underlying undirected graph. Note also that for directed graphs, the absence of cycle for the underlying undirected graph is stronger than the absence of cycles for the directed graph see Figure 3.1 which is an acyclic digraph but with a cyclic underlying undirected graph. 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,

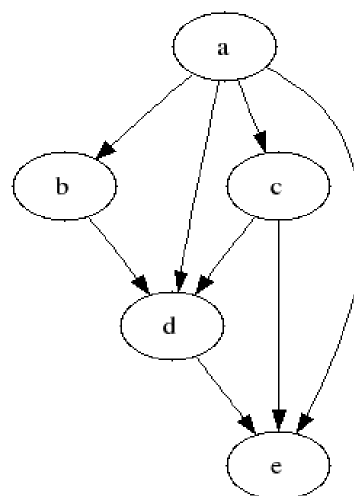


Fig. 3.1. An acyclic digraph

we shall call it *an oriented tree*, and accordingly an *unoriented tree* will be a tree with no orientation on its edges, see Figure 3.7.

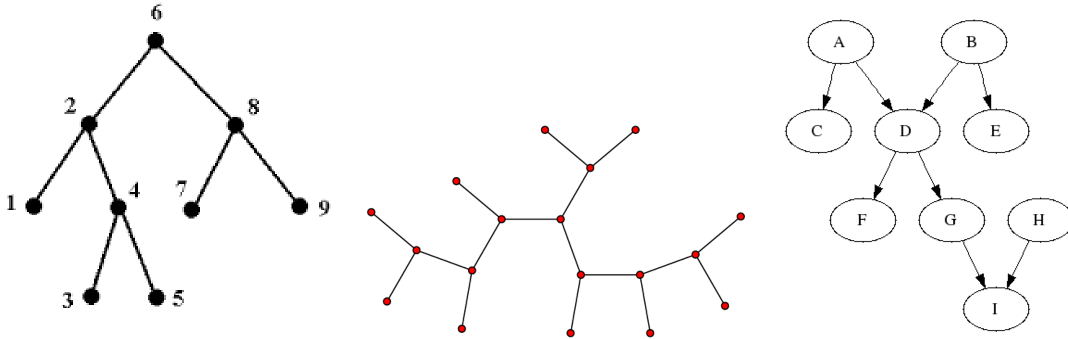


Fig. 3.2. Three trees: two unoriented, one oriented

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 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* whenever the trees are oriented trees.

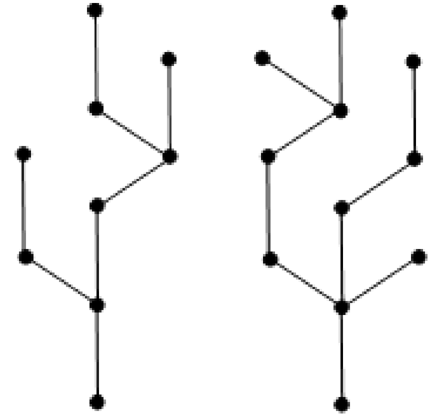


Fig. 3.3. One forest

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

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 trees.

Theorem 3.3. *For any unoriented finite 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.

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.

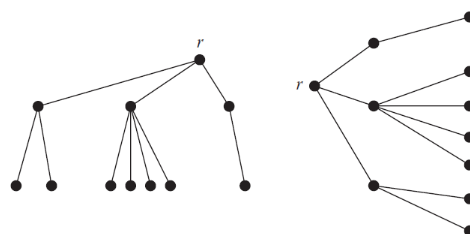


Fig. 3.5. Two trees with root r

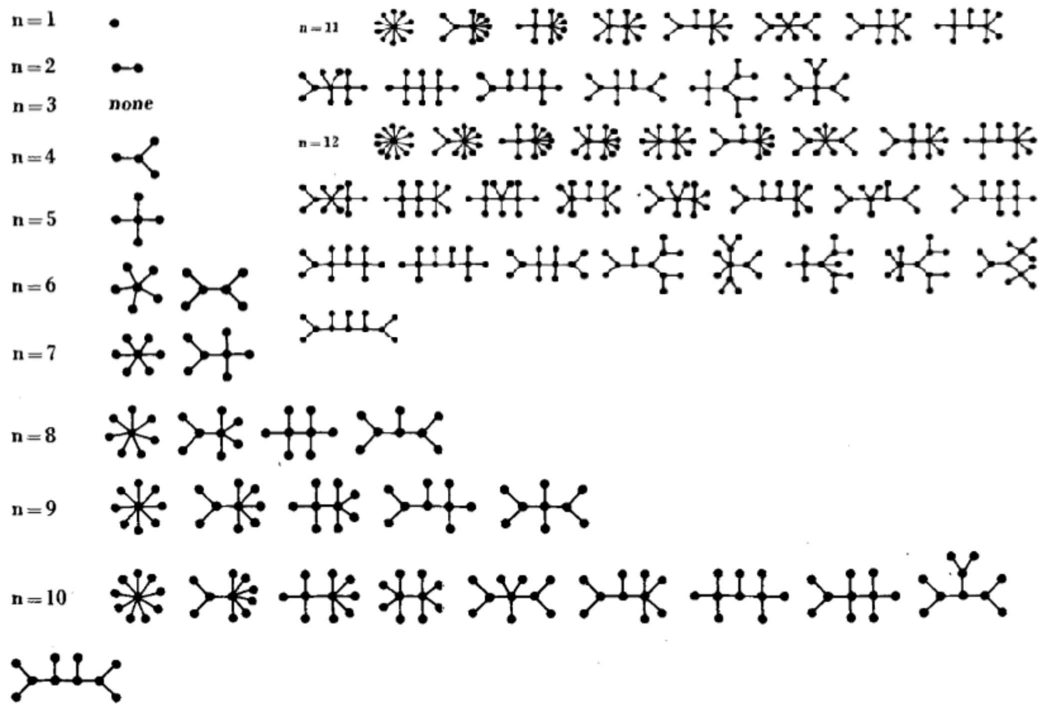


Fig. 3.4. Irreducible graphs with less than 12 vertices, see [4]

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

(i) A shortest path tree *is an unoriented 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).

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* $\mathrm{d}(r,x)$*,*

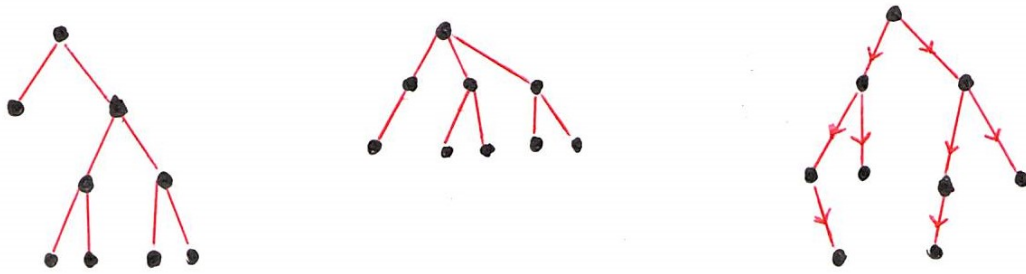


Fig. 3.6. Three trees

(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.*

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.

Fig. 3.7. A 3-ary tree and a complete 2-ary tree

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*.

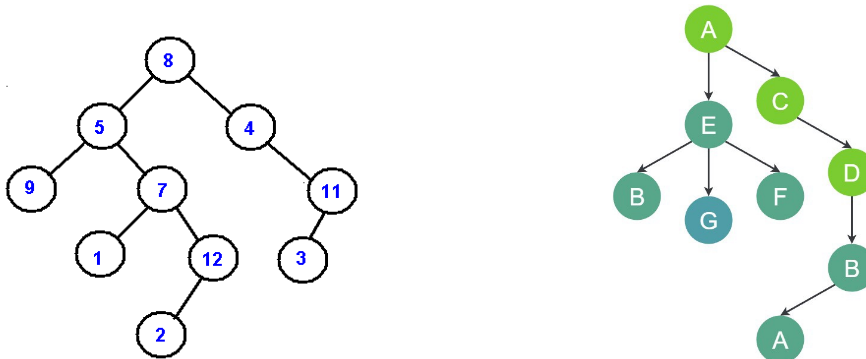


Fig. 3.8. One binary tree and one ternary tree

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].*

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.*

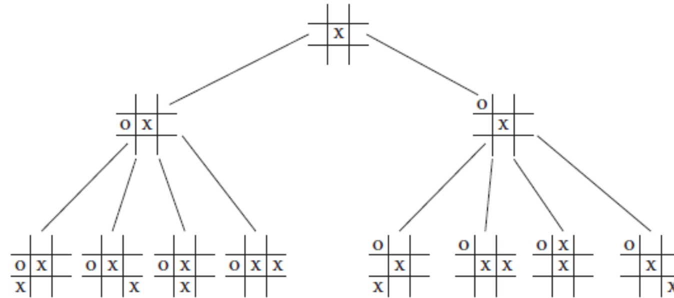


Fig. 3.9. The first three moves of tic-tac-toe

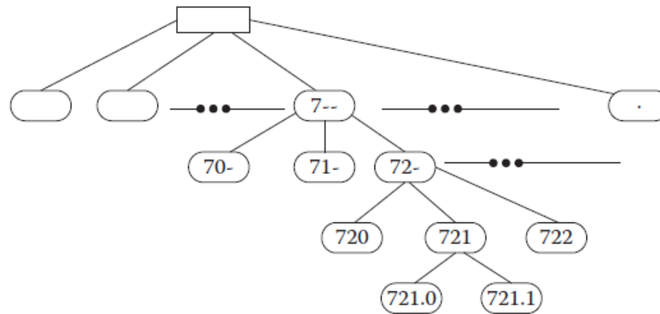


Fig. 3.10. Classification in libraries is often based on a tree data structure

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.*

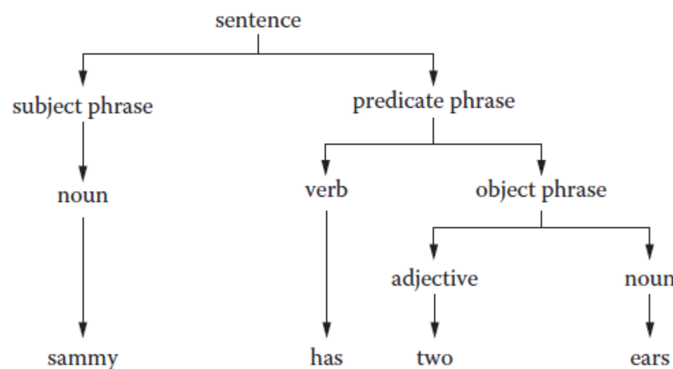


Fig. 3.11. Parsing a sentence

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 that the order on the tree is indexed by left and right. As a consequence, given a vertex of the graph, its left subtree and its right subtree 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:

Fig. 3.12. Left and right subtrees

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 .*

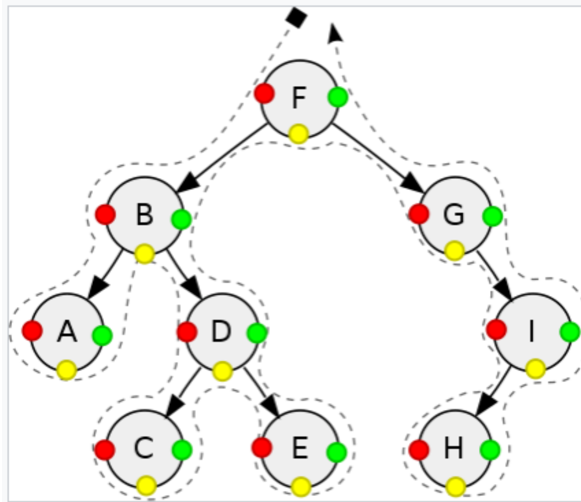


Fig. 3.13. Traversals of a binary tree

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

(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. As every subtree represents a subexpression, 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 resented in Figure 3.14 corresponds to the expression $\big((5+z)/-8\big)*(4^2)$, 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.

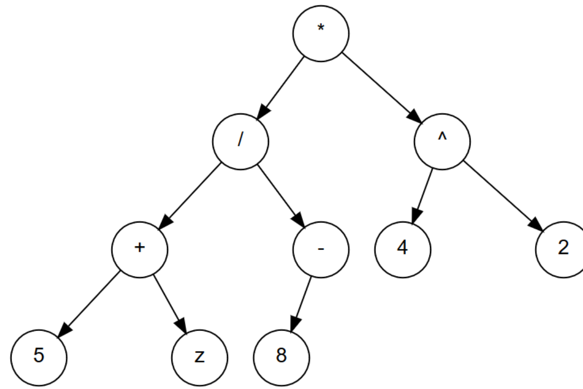


Fig. 3.14. An arithmetic expression tree

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, 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 b, or that b is larger than a.

Definition 3.16 (Binary search trees (BST))**.** Binary search trees (BST)*, also called* ordered *or* sorted binary trees *is a binary tree where each vertex x contains a value from a totally ordered set, and such that*

- *Vertices of the left subtree of x contain values which are smaller than or equal to the value contained in the vextex x,*
- *Vertices in the right subtree of x contain values which are larger than or equal to the value contained in the vertex x.*

If the totally ordered set is denoted by S, the valued at the vertex x, namely an element of S, is often called *a key*. We then say that the key at x has to be larger than the key at each vertex of the left subtree, and smaller than the key at each vertex of its right subtree. 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.

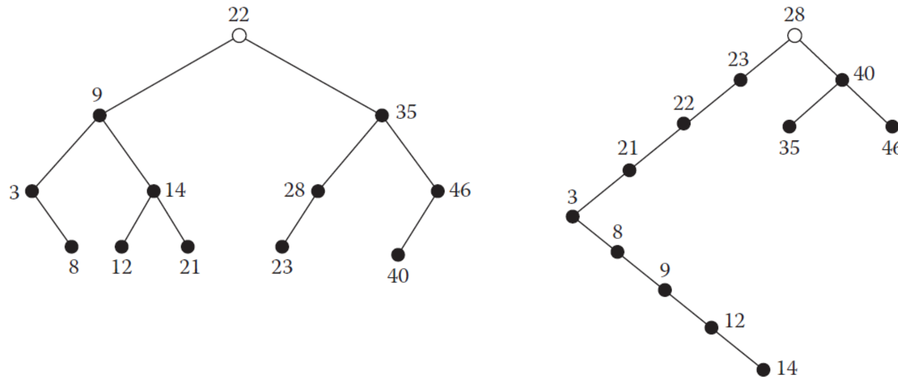


Fig. 3.15. Two binary search trees, see Figure 3.4.1 of [GYA]

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 $O\big(\ln(n)\big)$ operations[1]. 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 are described in Section 3.6.1, see also [8] for additional information.

3.4.3 Huffman trees

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

[1]For a strictly positive function ζ, the notation $f \in O\big(\zeta(n)\big)$ means that $\big|\frac{f(n)}{\zeta(n)}\big| \leq c$ for some $c < \infty$ and all n, while $f \in o\big(\zeta(n)\big)$ 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 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.

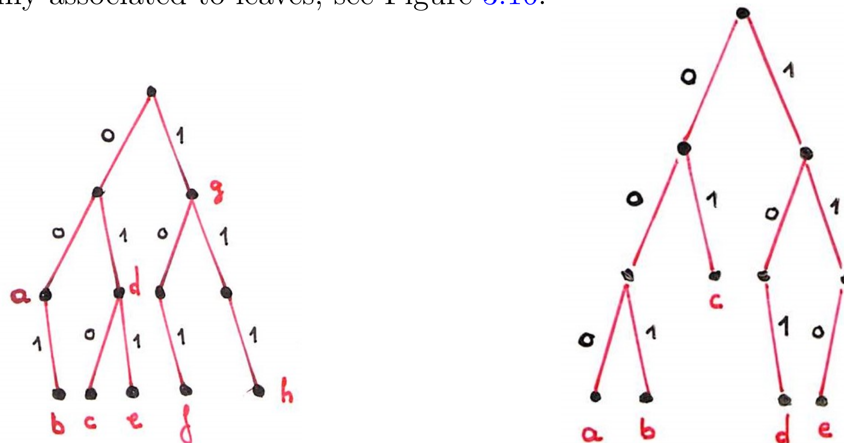


Fig. 3.16. Trees of a binary code and of a prefix code

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.

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,

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

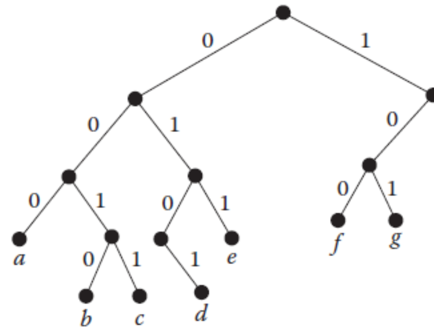


Fig. 3.17. A prefix code and the corresponding tree, see Figure 3.5.1 of [GYA]

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 $\{x_1, x_2, \dots, x_n\}$, 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 $w_1, \dots, w_l$.
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 1.
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.

Fig. 3.18. Huffman algorithm, from Algorithm 3.5.1 of [GYA]

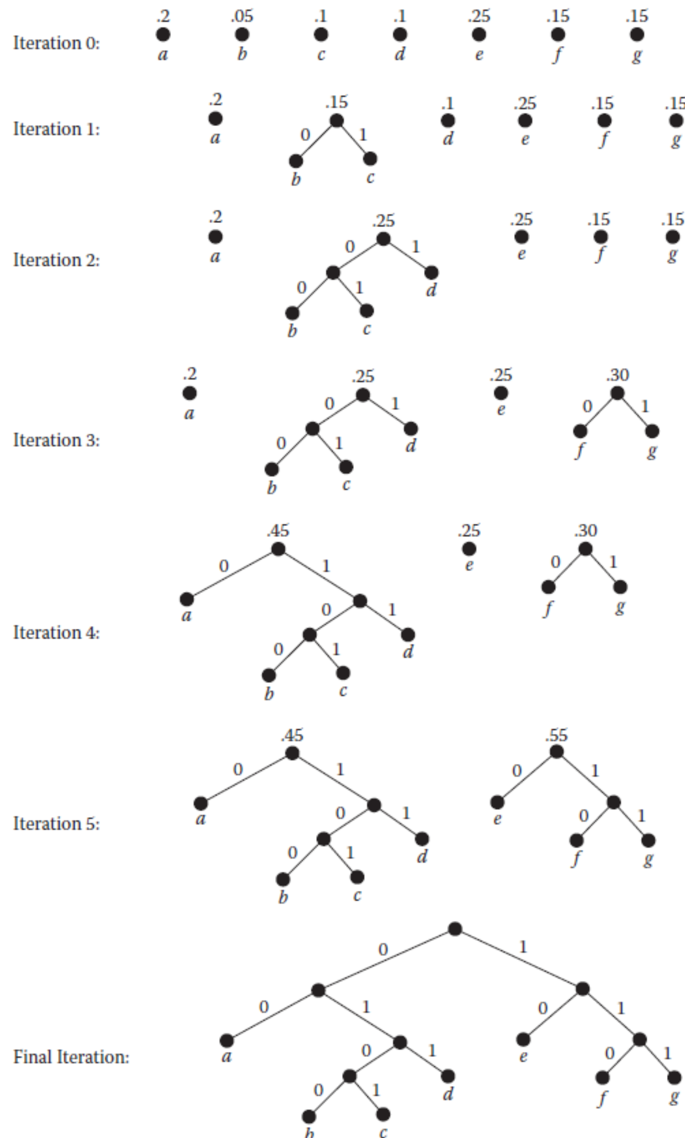


Fig. 3.19. Application of Huffman algorithm, from Example 3.5.3 of [GYA]

Note that if the weight ω_i associated to a leaf x_i is related to the frequency of the letter 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 $S := s_1, s_2, \dots, x_l$ and a list of weights $\{\omega_1, \omega_2, \dots, \omega_l\}$, 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:

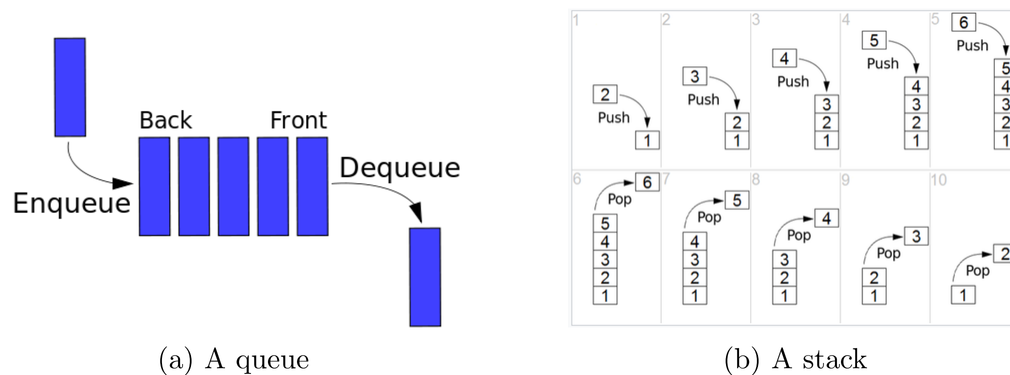
Theorem 3.20 (Huffman's theorem). *Given a list of weights* $\{\omega_1, \omega_2, \dots, \omega_l\}$*, the Huffman algorithm presented in Figure 3.18 generates an binary tree* T *which minimizes the weighted depth* $\omega(T)$ *among all binary trees.*

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

Fig. 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.

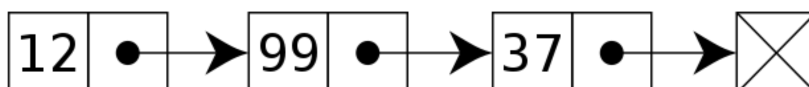


Fig. 3.21. A linked list, with a terminal object (terminator) represented by a box

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 $a, b, c \in S$:

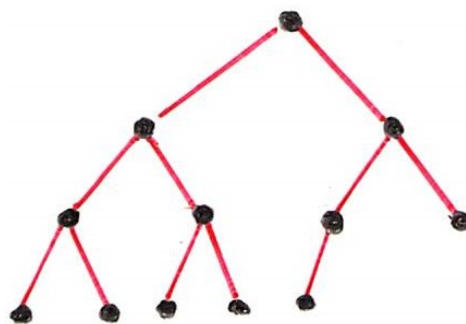


Fig. 3.22. A left-complete binay tree

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

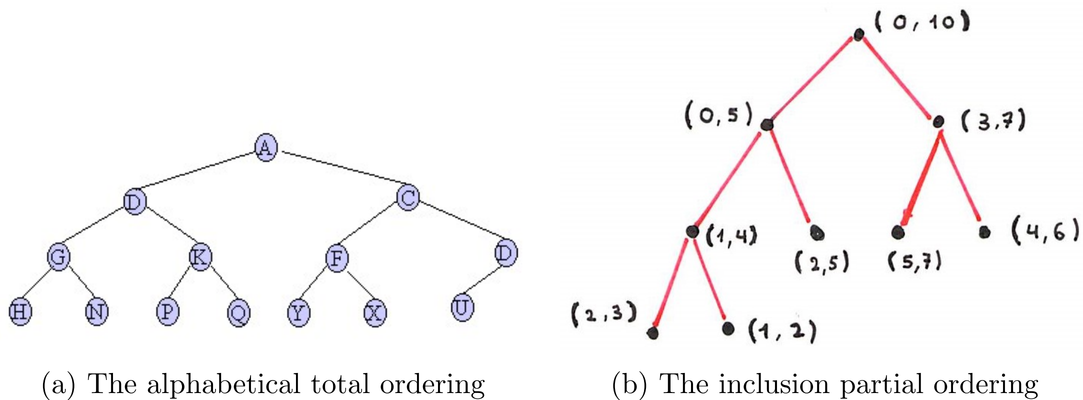
(ii) Antisymmetry: If $a \leq b$ and $b \leq a$, then $a = b$,

(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 whose vertices are labeled by the elements, called* priorities, *of a partially ordered set and such that no vertex has a priority larger than its parent.*



(a) The alphabetical total ordering (b) The inclusion partial ordering

Fig. 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[2].

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.

[2] 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 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 $parent(v)$.

Fig. 3.24. Priority tree insert, from algorithm 3.6.1 of [GYA]

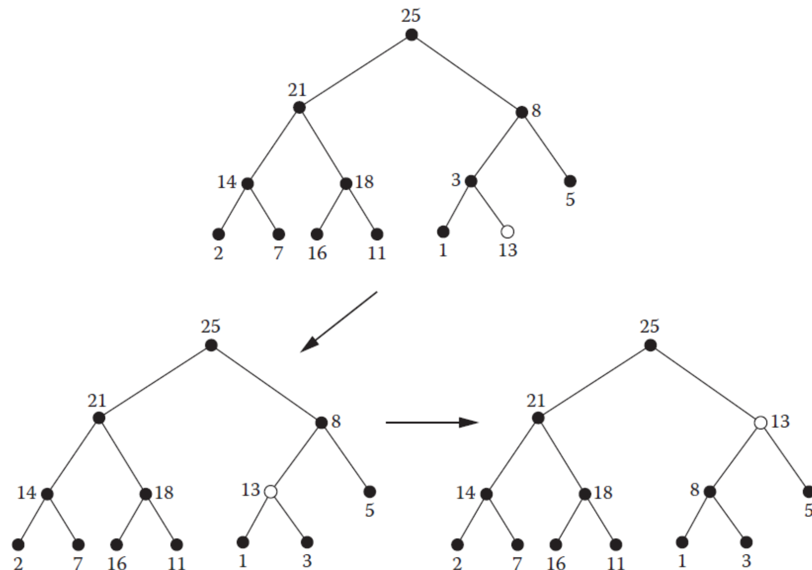


Fig. 3.25. The insertion of 13 in the priority tree, see Figure 3.6.3 of [GYA]

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 binary trees

Let us conclude this chapter with a question related to combinatoric: 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 ?

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

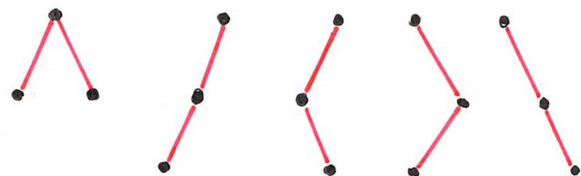


Fig. 3.28. Rooted ordered trees with 3 vertices

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 $rightchild(y)$

Fig. 3.26. Priority tree delete, from algorithm 3.6.2 of [GYA]

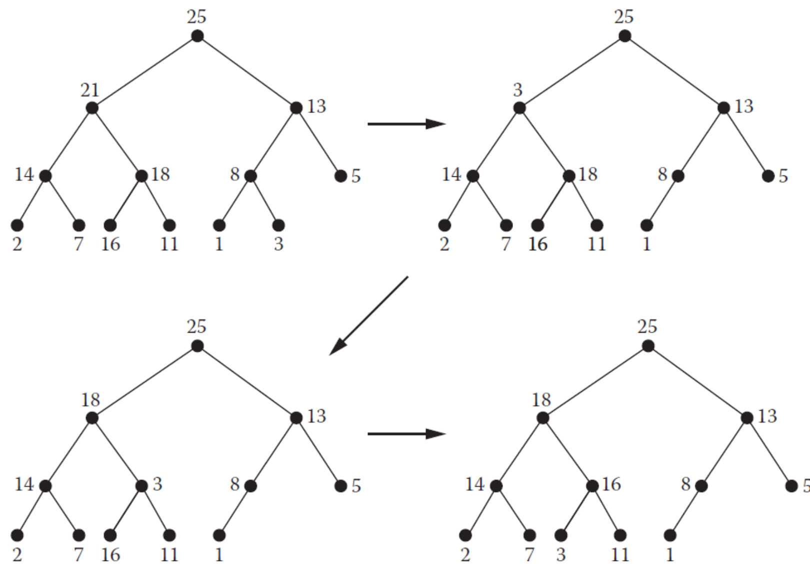


Fig. 3.27. The insertion of 21 in the priority tree, see Figure 3.6.4 of [GYA]

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

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

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 $e \in E$

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

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

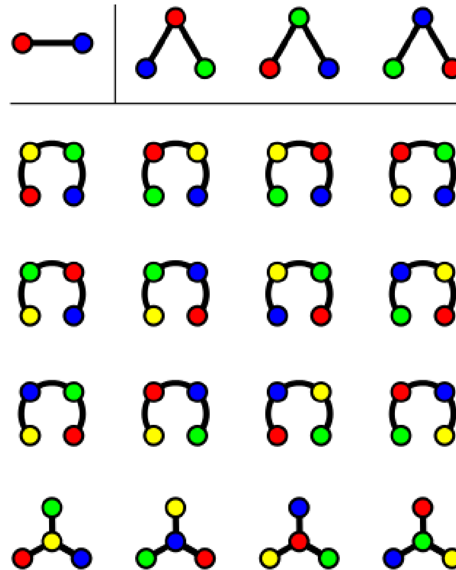


Fig. 3.29. Trees with 2, 3 and 4 labeled vertices, see [10]

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].

3.6 Appendix

3.6.1 Operations on binary search trees

The material of this section has been studied and written by Quang Nhat Le. It presents the most common operation performed on a binary search tree, namely the searching algorithms, the insert algorithm and the delete algorithm.

We shall first present the search operation and show that this operation can be supported in time $O(h)$ on a binary search tree of height h. Then we will discuss how the operations of insertion and deletion cause the dynamic set of a binary search tree change but still remain its properties. As we shall see, modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate.

The searching algorithm: Given a pointer to the root of the tree and a key k, searching algorithm will return a pointer to a node with key k if one exists, otherwise it returns nothing (or $NULL$). The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 3.30.

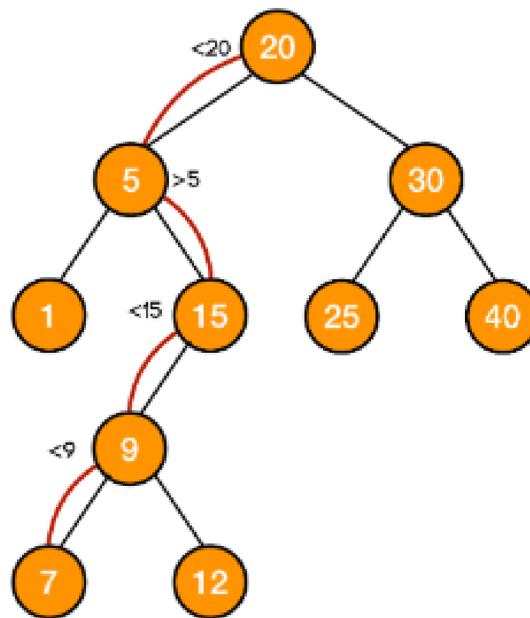


Fig. 3.30. Searching in BST

For example in Figure 3.30 one has to search for key 7. We start at the root node as current node. Since the search key's value (7) is smaller than current node's key(20) and the current node has *a left child*, we will search for the value in the left subtree left. Next, we search to the right since the search key's value (7) is greater than current node' key(5). This procedure will continue recursively in the subtree until the search key's value matches the current node's key.

In general, for each node x it encounters, it compares the key k with $key(x)$. If the two keys are equal, the search terminates. If k is smaller than $key(x)$, the search continues in the left subtree of x, since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $key(x)$, the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of this search operation is $O(h)$, where h is the height of the tree.

The insert algorithm: We can't insert any new node anywhere in a binary search tree because the tree after the insertion of the new node must follow the binary search tree property. Therefore, to insert an element, we first search for that element and if the element is not found, then we insert it.

Let us take an example similar to the previous search operation, but instead of finding the key 7 we will insert it into the tree in Figure 3.31. We will use a temporary pointer and go to the place where the node is going to be inserted. Here, we are starting from the root of the tree and then moving to the $left$ subtree if the data of the node to be inserted is less than the current node. Otherwise, we are moving $right$.

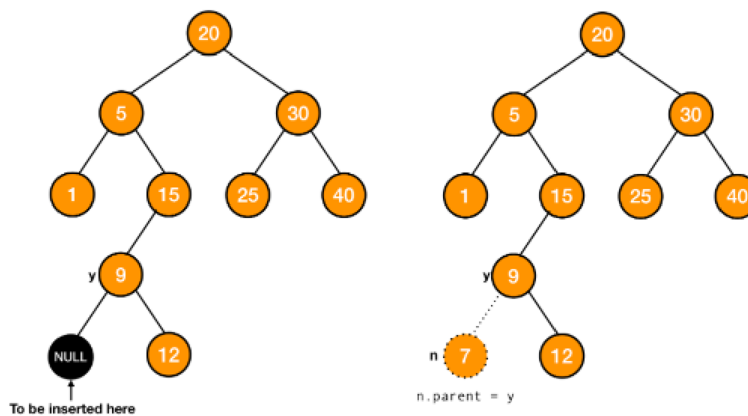


Fig. 3.31. Inserting in BST

Hence the general way to explain the insert algorithm is:

1. Always insert new node as leaf node
2. Start at root node as current node
3. If new node's key $<$ current's key
    - If current node has a $left$ child, search $left$
    - Else add new node as current's $left$ child
4. If new node's key $>$ current's key
    - If current node has a $right$ child, search $right$

- Else add new node as current's *right* child

5. This process continues, until the new node is compared with a leaf node

Remark 3.25. *There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.*

The delete algorithm: The last operation we need to do on a binary search tree to make it a full-fledged working data structure is to delete a node.

We can proceed in a similar manner to delete any node that has one child (or no child), but what can we do to delete a node that has two children? We are left with two links, but have a place in the parent node for only one of them. An answer to this dilemma, first proposed by *T. Hibbard* in 1962, is to delete a node x by replacing it with its *successor*.

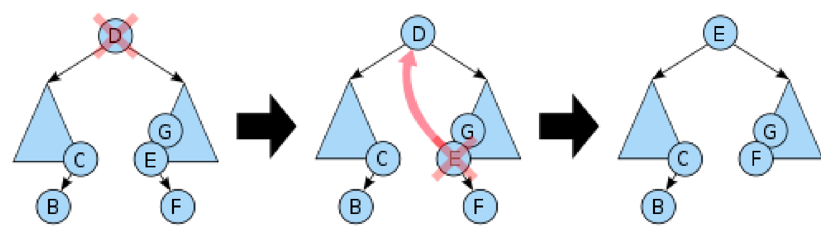


Fig. 3.32. Deleting a node with 2 children in BST

There are 3 possible cases:

1. Deleting a node with no children: simply remove the node from the tree.

2. Deleting a node with one child: remove the node and replace it with its child

3. Deleting a node with two children: call the node to be deleted D. Do not delete D. Instead, choose either its in-order predecessor node or its in-order successor node as replacement node E as Figure 3.32 . Copy the user values of E to D.If E does not have a child simply remove E from its previous parent G. If E has a child, say F, it is a right child. Replace E with F at E's parent.

Remark 3.26. *In all cases, when D happens to be the root, make the replacement node root again.*

Here is another example to better understanding 3 cases, see Figure 3.33:

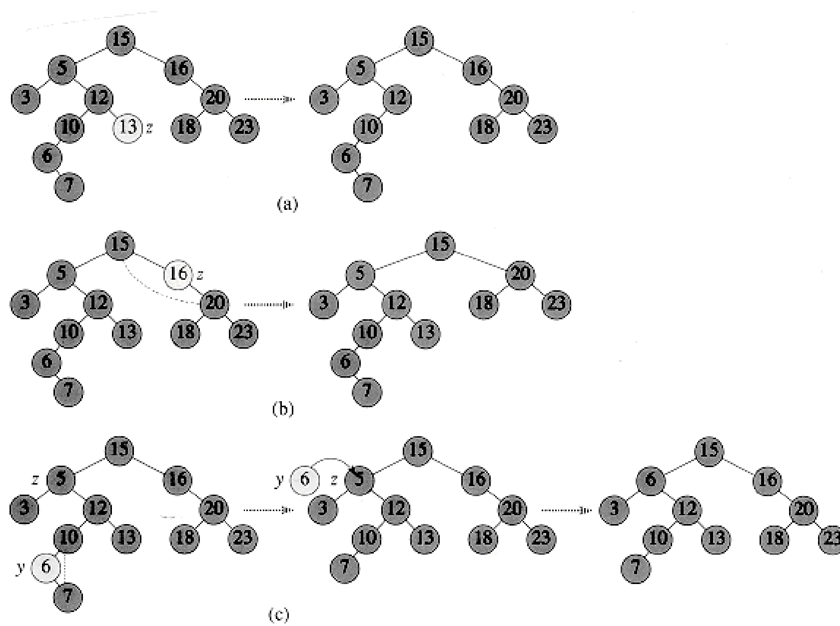


Fig. 3.33. Deleting a node z from a binary search tree

In case (a) we can see that if z has no children, we just remove it. For case (b) since z has only one child, we splice out z. And in (c) z has two children, we splice out its successor , which has at most one child, and then replace the contents of z with the contents of the successor.

While this method does the job, it has a flaw that might cause performance problems in some practical situations. The problem is that the choice of using the successor is arbitrary and not symmetric.

This section is based on the following references:

1) https://www.codesdope.com/course/data-structures-binary-search-trees/

2) http://staff.ustc.edu.cn/ csli/graduate/algorithms/book6/chap13.htm

3) https://en.wikipedia.org/wiki/Binary_search_tree

4) https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BST.html

3.6.2 An Improved Inserting Algorithm to Binary Search Trees

This section has been created by Liyang Zhang and Arata Suzuki.

Introduction

In Subsection 3.6.1 Quang Nhat Le has described several algorithms on binary search trees. However, the inserting algorithm has a drawback. If we insert a lot of new nodes

into a binary search tree, for example, to insert 41, 42, 43, ..., 100 into Figure 3.31, then the height of the tree will grow much faster than the logarithm of number of nodes, leading to long time for subsequenting searching, inserting or deleting operations.

Therefore, it seems good to put some reasonable restriction on the height h of the tree resulted by the inserting operation, for example,

$$h \le \log_2 n + 2 \tag{3.1}$$

with n denoting the order of the graph after the inserting operation. This kind of restriction makes sure that the resulting binary search tree has a relatively low height, and that the searching, inserting and deleting operations are relatively fast.

We have created an improved inserting algorithm, based on the original one, to meet a given restriction on the height of the resulting binary search tree. The time complexity of this algorithm is still $O(h)$, as is the original one. However, we meet a difficulty in a situation shown in Figure 3.36. In this case we have either to ignore the restriction on height, keeping the time complexity $O(h)$; or to reorder contents of $O(2^h)$ nodes, with a time complexity $O(n) = O(2^h)$, since there is only one element that is out of order. The improvement part is 100% original.

General Idea

The general idea of this algorithm is below.

- Try to insert the new node as a leaf, in the same way as the algorithm introduced by Quang Le.
- If the height of the resulting graph is less than or equal to the restriction, return the new graph.
- If the height of the resulting graph exceeds the restriction, do not insert the new node here. Instead, move up along the tree and search for a node with only one child, and insert the new node as the other child, or to say, at the "empty site".
- Then, rearrange the tree by comparing repeatively the parent and two children, and correcting their order. Return it after the order is corrected completely.

Figure 3.34 and Figure 3.35 are used as examples to help explaining the algorithm.

Notations

$\mathcal{N}$:= current node.

$^*\mathcal{N}$:= the content of current node.

$h(v)$:= height of node v. The height of the root node is 0.

$h_{\max}$:= the restriction on the height of the resulting tree.

In the algorithm below, ROMAN letters are the main texts or the algorithm, and *ITALIAN* letters are comments.

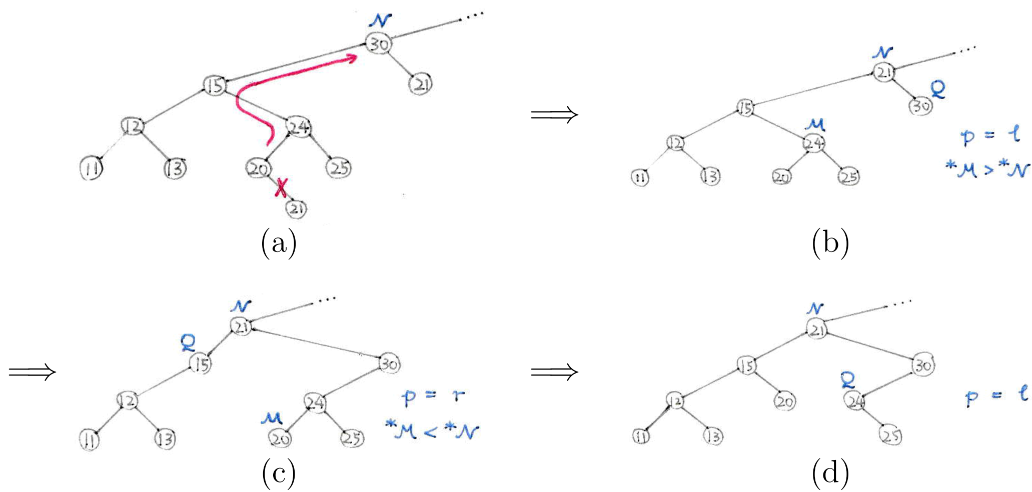


Fig. 3.34. the first case of inserting.

Algorithm

Input: binary search tree $\mathcal{T}$ on a totally ordered set $(\mathbb{S}, \leq)$; element $x \in \mathbb{S}$ to be added into $\mathcal{T}$; restriction on the height $h_{\max}(n)$ dependent on the order n of the tree.

Output: binary search tree $\mathcal{T}_f$ resulted from the insertion.

1. Calculate $h_{\max} := h_{\max}(\text{order}(\mathcal{T}) + 1)$.

2. If $\text{order}(\mathcal{T}) \geq 2^{h_{\max}+1} - 1$, the graph is full and any insertion is impossible under this restriction.

3. Set $\mathcal{N}$ = root node.

4. While $h(\mathcal{N}) < h_{\max}$

   - If $x ==^* \mathcal{N}$
     - return $\mathcal{T}$.
   - Else if $x <^* \mathcal{N}$
     - If $\mathcal{N}$ has a left child $\mathcal{L}(\mathcal{N})$, set $\mathcal{N} = \mathcal{L}(\mathcal{N})$ and continue the loop.
     - Else, put x into a new node as the left child of $\mathcal{N}$, and return $\mathcal{T}$.
   - Else
     - If $\mathcal{N}$ has a right child $\mathcal{R}(\mathcal{N})$, set $\mathcal{N} = \mathcal{R}(\mathcal{N})$ and continue the loop.
     - Else, put x into a new node as the right child of $\mathcal{N}$, and return $\mathcal{T}$. $(*)$

   Above is the algorithm introduced by Quang Le. Below is original.

5. *When the loop above finishes before returning, we must have now $h(\mathcal{N}) == h_{\max}$.*

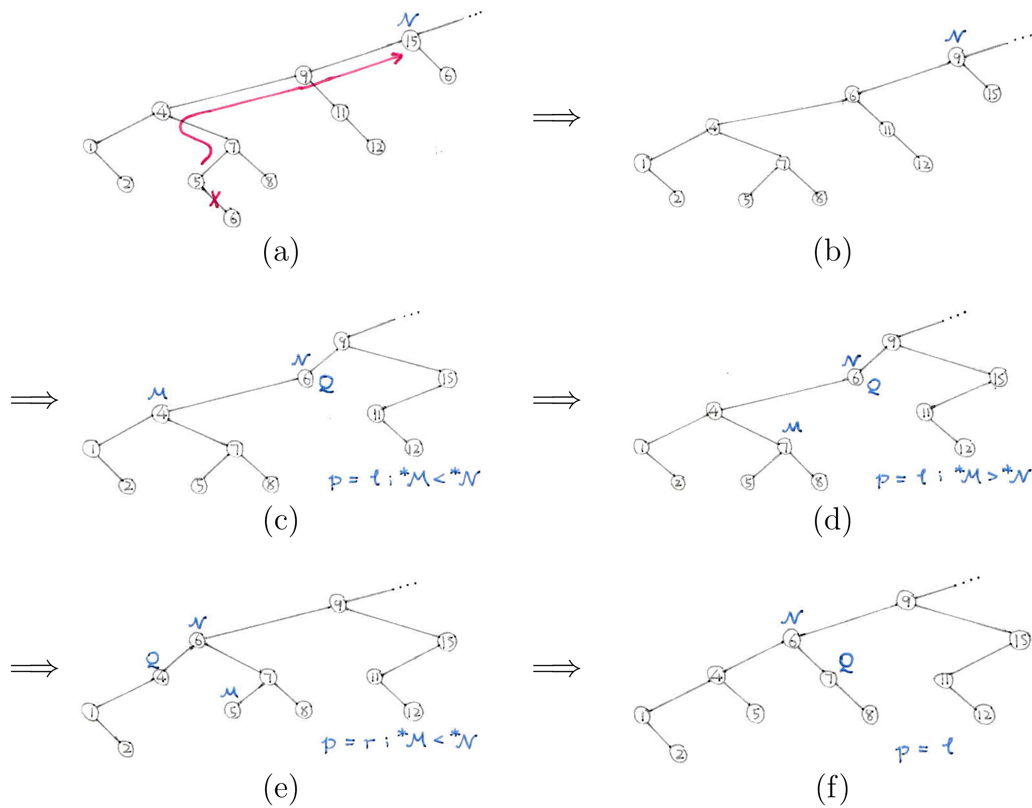


Fig. 3.35. the second case of inserting.

- If $x ==^* \mathcal{N}$, return $\mathcal{T}$.
- *Else, we cannot add a node as $\mathcal{N}$'s child anymore, since we have reached h_{max} already.*

  Therefore, we add it at the closest empty space, and adjust the neiboring nodes later.

Now we go back to the first node we encounter with only 1 child.

6. Firstly, set $\mathcal{N} = \mathcal{P}(\mathcal{N})$ with $\mathcal{P}(\mathcal{N})$ denoting the parent of $\mathcal{N}$.
7. While $h(\mathcal{N}) \geq 0$
   - If $\mathcal{N}$ has 2 children, set $\mathcal{N} = \mathcal{P}(\mathcal{N})$.
   - Else if $\mathcal{N}$ has only 1 left child $\mathcal{L}(\mathcal{N})$
     - Create a right child of $\mathcal{N}$ called $\mathcal{R}(\mathcal{N})$, and set ${}^*\mathcal{R}(\mathcal{N}) = x$.
     - *In the downward loop before, the left child must have been chosen here, since otherwise that downward loop would have terminated and algorithm finished in $(*)$.*

       Thus, we have either ${}^*\mathcal{L}(\mathcal{N}) <^* \mathcal{R}(\mathcal{N}) <^* \mathcal{N}$ or ${}^*\mathcal{R}(\mathcal{N}) <^* \mathcal{L}(\mathcal{N}) <^* \mathcal{N}$.

- If ${}^*\mathcal{L}(\mathcal{N}) <^* \mathcal{R}(\mathcal{N}) <^* \mathcal{N}$ *(See Figure 3.34 as an example.)*
  (a) Exchange ${}^*\mathcal{N}$ with ${}^*\mathcal{R}(\mathcal{N})$.
  (b) *By this construction, $\mathcal{R}(\mathcal{N})$ is a leaf, but $\mathcal{L}(\mathcal{N})$ must have children. Since $\mathcal{L}(\mathcal{N})$ is unchanged, its left subtree contain only elements smaller than ${}^*\mathcal{L}(\mathcal{N})$, and its right subtree contain only elements greater than ${}^*\mathcal{L}(\mathcal{N})$. But its right subtree may contain elements greater than ${}^*\mathcal{N}$, which leads to a problem.*
  Now we search for all such nodes and move them to be the children of $\mathcal{R}(\mathcal{N})$. Since ${}^*\mathcal{N}$ becomes ${}^*\mathcal{R}(\mathcal{N})$, elements of the right subtree of $\mathcal{L}(\mathcal{N})$ always lie between $\mathcal{L}(\mathcal{N})$ and $\mathcal{R}(\mathcal{N})$.
  (c) Create a new pointer $\mathcal{M}$, and initialize it to be $\mathcal{R}(\mathcal{L}(\mathcal{N}))$.
  $\mathcal{M}$ is the node to be examined whether to move.
  $\mathcal{N}$ is fixed, and is the root of the problematic subtree.
  (d) Create 2 new pointers $\mathcal{Q}$ and $\mathcal{Q}'$, and initialize both to be $\mathcal{R}(\mathcal{N})$.
  $\mathcal{Q}$ is the node which $\mathcal{M}$ is going to be moved to be the child of. $\mathcal{Q}'$ serves as a temporary variable.
  (e) Create a flag $p \in \{l, r\}$, and initialize it to be l.
  $p = l$ iff $\mathcal{M}$ is in the left subtree of $\mathcal{N}$; $p = r$ iff $\mathcal{M}$ is in the right subtree of $\mathcal{N}$.
  (f) While $\mathcal{M} \neq$ NULL and $h(\mathcal{M}) \leq h_{\max}$
    - If $p = l$ and ${}^*\mathcal{M} <^* \mathcal{N}$,
      $\mathcal{M}$ and its left subtree should not be moved, and we examine its right subtree.
      - Set $\mathcal{M} = \mathcal{R}(\mathcal{M})$.
    - Else if $p = l$ and ${}^*\mathcal{M} >^* \mathcal{N}$,
      $\mathcal{M}$ should be at the proper place in right subtree of $\mathcal{N}$, and we examine the left subtree of $\mathcal{M}$.
      - Set $\mathcal{Q}' = \mathcal{P}(\mathcal{M})$.
      - Move the node $\mathcal{M}$ to be the left child of $\mathcal{Q}$.
      - Set $p = r$.
      - Set $\mathcal{M} = \mathcal{L}(\mathcal{M})$ and $\mathcal{Q} = \mathcal{Q}'$.
    - Else if $p = r$ and ${}^*\mathcal{M} <^* \mathcal{N}$,
      $\mathcal{M}$ should be at the proper place in left subtree of $\mathcal{N}$, and we examine the right subtree of $\mathcal{M}$.
      - Set $\mathcal{Q}' = \mathcal{P}(\mathcal{M})$.
      - Move the node $\mathcal{M}$ to be the right child of $\mathcal{Q}$.
      - Set $p = l$.
      - Set $\mathcal{M} = \mathcal{R}(\mathcal{M})$ and $\mathcal{Q} = \mathcal{Q}'$.
    - Else if $p = r$ and ${}^*\mathcal{M} >^* \mathcal{N}$,
      $\mathcal{M}$ and its right subtree should not be moved, and we examine its left subtree.

- ∘ Set $\mathcal{M} = \mathcal{L}(\mathcal{M})$.

∗ *These are the only 4 cases.*

(g) *After this loop, everything should have been sorted out.* Return $\mathcal{T}$.

∘ Else if $^*\mathcal{R}(\mathcal{N}) <^* \mathcal{L}(\mathcal{N}) <^* \mathcal{N}$ *(See Figure 3.35 as an example.)*

(a) Keep the nodes, and reorder their contents such that $^*\mathcal{L}(\mathcal{N}) <^* \mathcal{N} <^* \mathcal{R}(\mathcal{N})$.

(b) *By this construction, $\mathcal{R}(\mathcal{N})$ is a leaf, but $\mathcal{L}(\mathcal{N})$ must have children. Since $^*\mathcal{L}(\mathcal{N})$ becomes $^*\mathcal{N}$, every element in the right subtree of $\mathcal{L}(\mathcal{N})$ is now greater than $^*\mathcal{N}$. And since $^*\mathcal{N}$ becomes $^*\mathcal{R}(\mathcal{N})$, every element in the right subtree of $\mathcal{L}(\mathcal{N})$ is now smaller than $^*\mathcal{R}(\mathcal{N})$. Therefore, we should move the whole right subtree of $\mathcal{L}(\mathcal{N})$ to be the left subtree of $^*\mathcal{R}(\mathcal{N})$.*

(c) Move the node $\mathcal{R}(\mathcal{L}(\mathcal{N}))$ to be the left child of $\mathcal{R}(\mathcal{N})$.

(d) *Since $^*\mathcal{L}(\mathcal{N})$ becomes $^*\mathcal{N}$, every element in the left subtree of $\mathcal{L}(\mathcal{N})$ is now smaller than $^*\mathcal{N}$. But since the newly added $^*\mathcal{R}(\mathcal{N})$ becomes $^*\mathcal{L}(\mathcal{N})$, the left subtree of $\mathcal{L}(\mathcal{N})$ may now contain elements greater than $^*\mathcal{L}(\mathcal{N})$. Therefore, we search for all such nodes and move them into the right subtree of $\mathcal{L}(\mathcal{N})$.*

(e) Create a new pointer $\mathcal{M}$, and initialize it to be $\mathcal{L}(\mathcal{L}(\mathcal{N}))$.
$\mathcal{M}$ is the node to be examined whether to move.

(f) Set $\mathcal{N} = \mathcal{L}(\mathcal{N})$.
$\mathcal{N}$ is fixed, and is the root of the problematic subtree.

(g) Create 2 new pointers $\mathcal{Q}$ and $\mathcal{Q}'$, and initialize both to be $\mathcal{N}$.
$\mathcal{Q}$ is the node which $\mathcal{M}$ is going to be moved to be the child of. $\mathcal{Q}'$ serves as a temporary variable.

(h) Create a flag $p \in \{l, r\}$, and initialize it to be l.
$p = l$ iff $\mathcal{M}$ is in the left subtree of $\mathcal{N}$; $p = r$ iff $\mathcal{M}$ is in the right subtree of $\mathcal{N}$.

(i) While $\mathcal{M} \neq$ NULL and $h(\mathcal{M}) \leq h_{\max}$

∗ If $p = l$ and $^*\mathcal{M} <^* \mathcal{N}$,
$\mathcal{M}$ and its left subtree should not be moved, and we examine its right subtree.
- ∘ Set $\mathcal{M} = \mathcal{R}(\mathcal{M})$.

∗ Else if $p = l$ and $^*\mathcal{M} >^* \mathcal{N}$,
$\mathcal{M}$ should be at the proper place in right subtree of $\mathcal{N}$, and we examine the left subtree of $\mathcal{M}$.
- ∘ Set $\mathcal{Q}' = \mathcal{P}(\mathcal{M})$.
- ∘ If $\mathcal{Q} == \mathcal{N}$, move the node $\mathcal{M}$ to be the right child of $\mathcal{N}$.
- ∘ Else, move the node $\mathcal{M}$ to be the left child of $\mathcal{Q}$.
- ∘ Set $p = r$.

- Set $\mathcal{M} = \mathcal{L}(\mathcal{M})$ and $\mathcal{Q} = \mathcal{Q}'$.

* Else if $p = r$ and $^*\mathcal{M} <^* \mathcal{N}$,
  $\mathcal{M}$ should be at the proper place in left subtree of $\mathcal{N}$, and we examine the right subtree of $\mathcal{M}$.
  - Set $\mathcal{Q}' = \mathcal{P}(\mathcal{M})$.
  - If $\mathcal{Q} == \mathcal{N}$, move the node $\mathcal{M}$ to be the left child of $\mathcal{N}$.
  - Else, move the node $\mathcal{M}$ to be the right child of $\mathcal{Q}$.
  - Set $p = l$.
  - Set $\mathcal{M} = \mathcal{R}(\mathcal{M})$ and $\mathcal{Q} = \mathcal{Q}'$.
* Else if $p = r$ and $^*\mathcal{M} >^* \mathcal{N}$,
  $\mathcal{M}$ and its right subtree should not be moved, and we examine its left subtree.
  - Set $\mathcal{M} = \mathcal{L}(\mathcal{M})$.
* *These are the only 4 cases.*

(j) *After this loop, everything should have been sorted out.* Return $\mathcal{T}$.

- Else, $\mathcal{N}$ *has only 1 right child* $\mathcal{R}(\mathcal{N})$*.* Do the same as the case of only 1 left child, while replacing every word "left" with "right", and vice versa. Then return $\mathcal{T}$.

8. If the algorithm does not return before this loop finish, it means that this algorithm has not encountered any node with only 1 child, including the root node. *(See Figure 3.36 as an example.)* We have not yet found a way to solve this problem within the time complexity of $O(h)$. Our choices are either to ignore the restriction, set $h_{\max} = \infty$, and carry out the algorithm (now same as the algorithm introduced by Quang Le) again; or to insert the new node at an empty place, and reorder all contents of the nodes. The time complexity of the latter choice is unfortunately $O(n) = O(2^h)$, since there is only one element that is out of order.

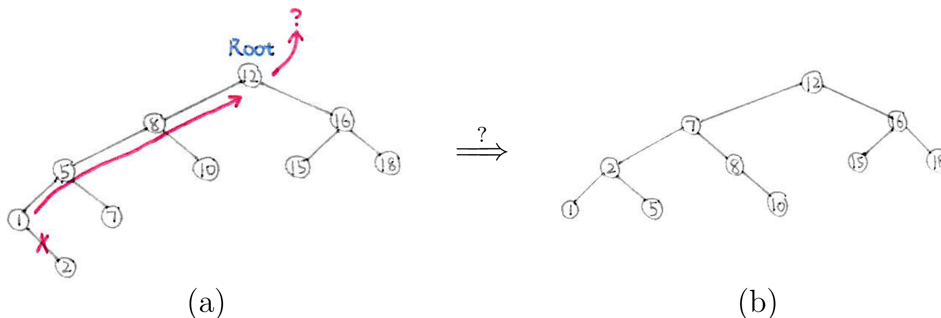


Fig. 3.36. an example with algorithm still not found within $O(h)$.

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 Spanning trees and their growth

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.

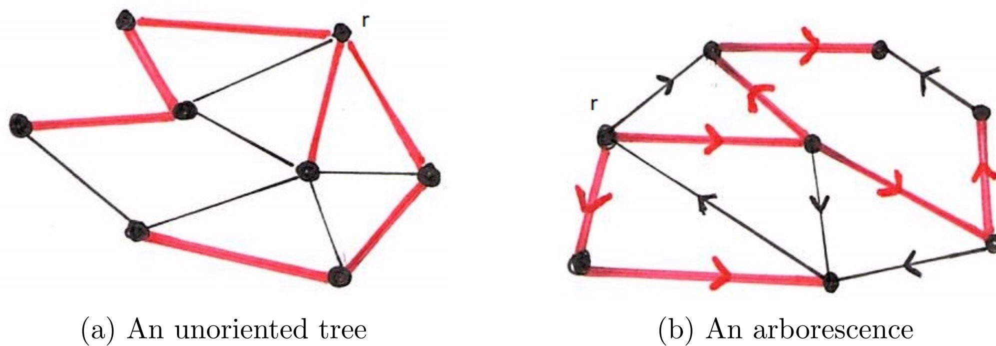
Definition 4.1 (Spanning tree). *Let G be a connected graph. A spanning tree T of G is a subgraph of G which is either an unoriented tree or an arborescence which includes every vertex of G.*

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.

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.*



(a) An unoriented tree (b) An arborescence

Fig. 4.1. Two spanning trees with root r

(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.

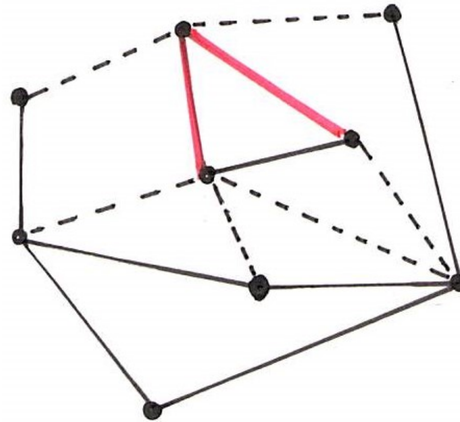


Fig. 4.2. A graph, a tree, and the frontier edges as dashed lines

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 to T generates a new subgraph of G which is an unoriented tree or an arborescence.*

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* $\mathrm{Front}(G, T_i)$*, set* $T_{i+1} = T_i \sqcup \{e_{i+1}\}$*, and set* $i := i+1$*,*

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

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 $e_{i+1} \in \mathrm{Front}(G, T_i)$ 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 $\mathrm{Front}(G, T_{i+1})$ is different from $\mathrm{Front}(G, T_i)$, it might not be necessary to compute this set from scratch but some information can be inferred from $\mathrm{Front}(G, T_i)$.

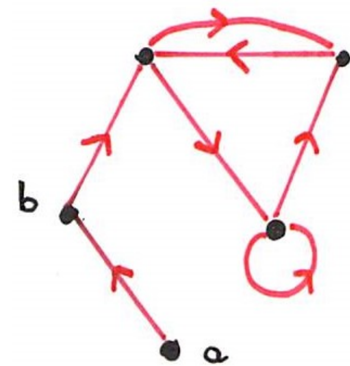
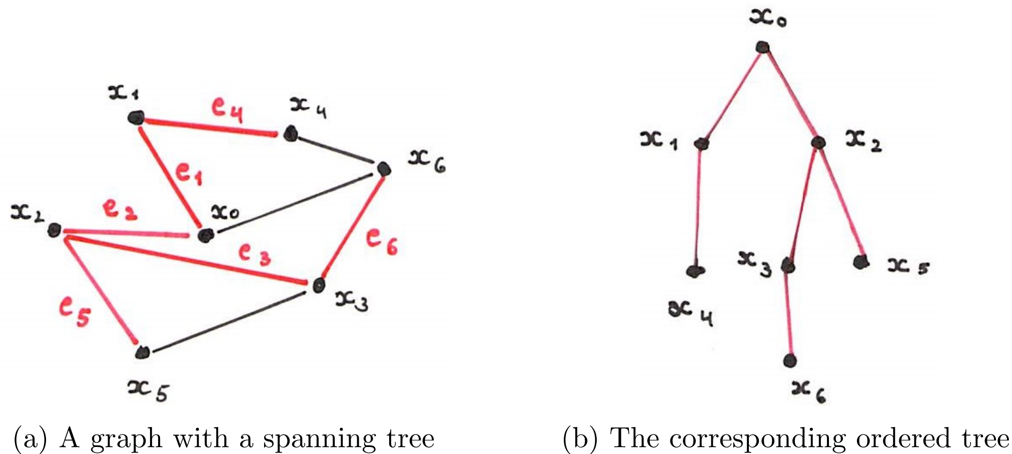


Fig. 4.3. A digraph

Remark 4.5 (Discovery number)**.** *In the point (ii) of the above algorithm, we have written* "Choose one element e_{i+1} of $\mathrm{Front}(G, T_i)$, set $T_{i+1} = T_i \sqcup \{e_{i+1}\}$" *and not simply* "Choose one element e of $\mathrm{Front}(G, T_i)$, 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 $V_i = \{x_0, x_1, \dots, x_i\}$ and $E_i = \{e_1, e_2, \dots, e_i\}$.*



(a) A graph with a spanning tree (b) The corresponding ordered tree

Fig. 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 $\mathrm{Front}(G, T_i)$ 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 $\text{Front}(G, T_i)$ 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.

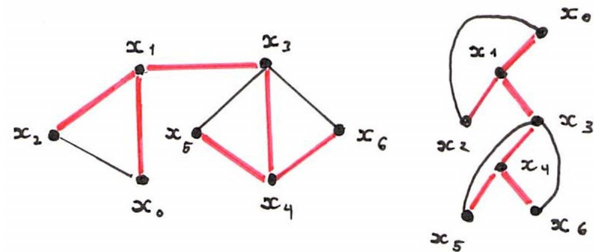


Fig. 4.5. A DFS tree

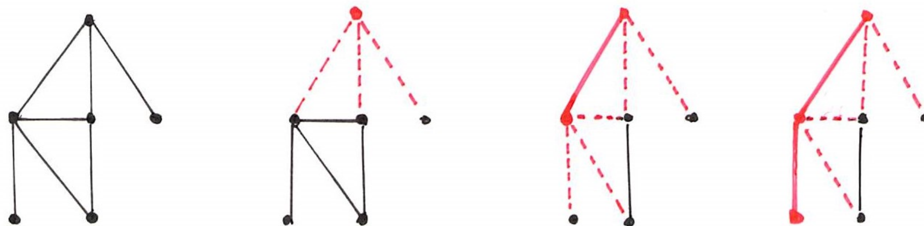


Fig. 4.6. A growing DFS tree with $\text{Front}(G, T_i)$ in dashed lines

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 $e_{i+1} \in \mathrm{Front}(G, T_i)$ 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 $\mathrm{Front}(G, T_i)$ 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.

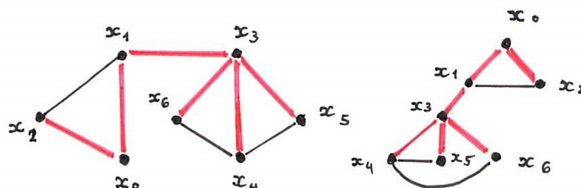


Fig. 4.7. A BSF tree

Lemma 4.7.

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

(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.

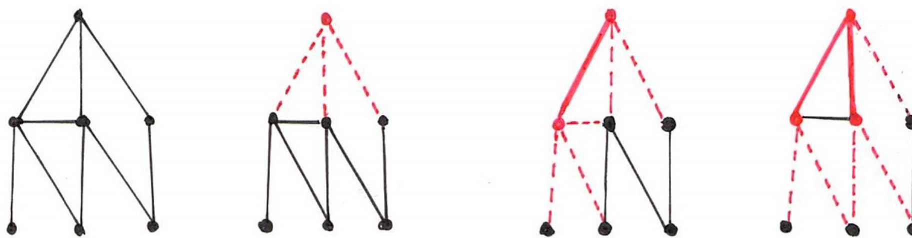


Fig. 4.8. A growing BFS tree with $\mathrm{Front}(G, T_i)$ in dashed lines

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.

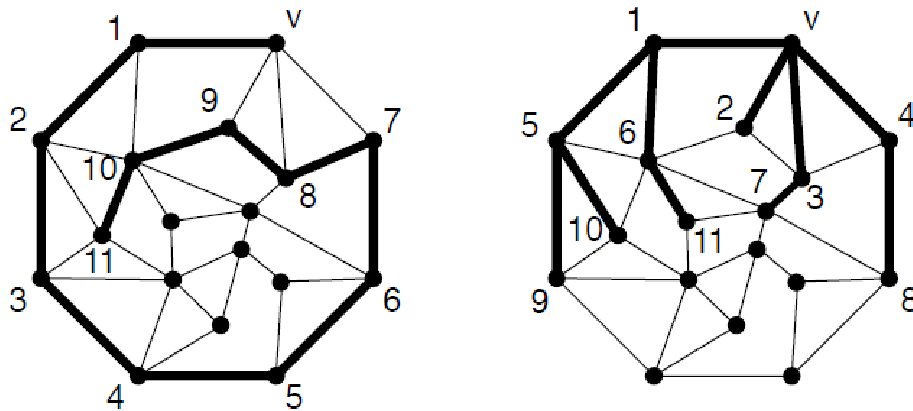


Fig. 4.9. A DFS tree and a BFS tree after 11 iterations, see also Figure 4.2.2 of [GYA]

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.

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.

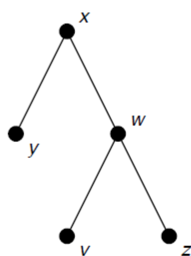
Definition 4.8 (Finished vertex)**.** *In a depth-first search, a discovered vertex is* finished *when all its neighbours have been discovered and those with higher discovery number are all finished.*

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.*



	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	$y(4)$	$\emptyset$	$y(4), x(5)$

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

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

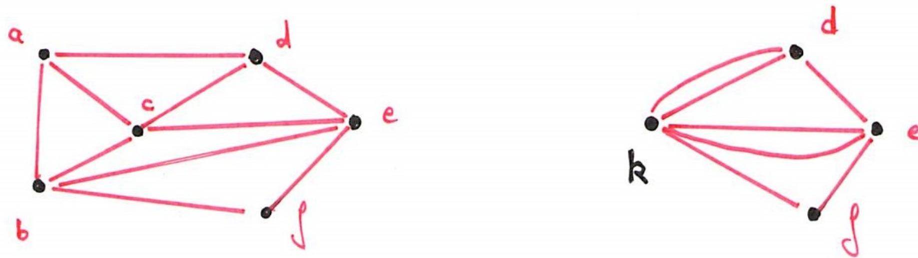


Fig. 4.11. Contraction of $V_H = \{a, b, c\}$ into the vertex k

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.

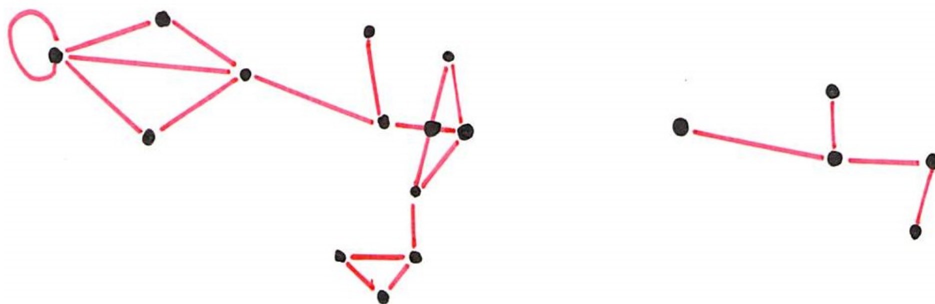


Fig. 4.12. Before and after contracting each bridge component of G

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

> *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

Fig. 4.13. How to find bridges, from algorithm 4.4.1 of [GYA]

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.

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.

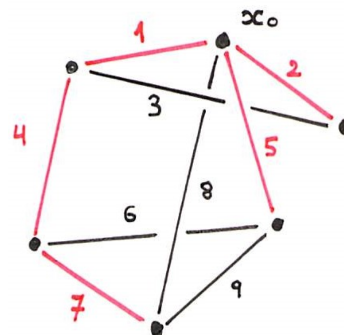


Fig. 4.14. A minimum spanning tree

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.

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

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

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[1]. 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), see Section 4.5.4. 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.

[1] 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 $x, y, z \in V$.

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

$$\mathrm{d}_\omega\big(x_0, \mathrm{t}(e_{i+1})\big) := \min_{e \in \mathrm{Front}(G,T_i)} \Big(\mathrm{d}_\omega\big(x_0, \mathrm{o}(e)\big) + \omega(e)\Big). \tag{4.1}$$

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 (4.1), then the default priority is applied. Note also that this algorithm is well defined, since $\mathrm{d}_\omega\big(x_0, \mathrm{o}(e)\big)$ 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]. Another version with several examples is provided on the very well documented website [12], see also [13]. Note finally that Dijkstra's algorithm has been 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 (4.1) 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.

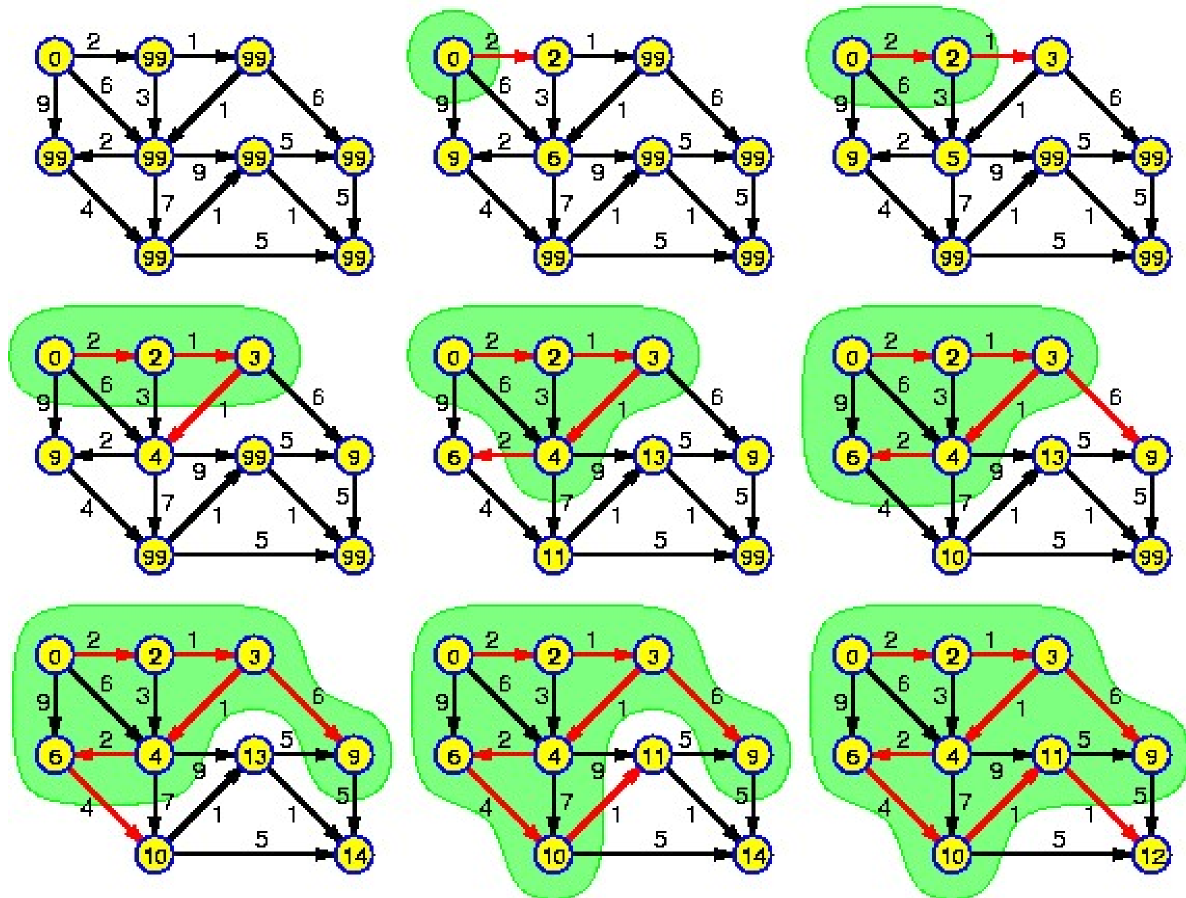


Fig. 4.15. Application of Dijkstra's algorithm, from [14]

4.5 Appendix

4.5.1 A few problems on spanning trees

The material of this section has been provided by Duc Truyen Dam and by Atsuya Watanabe.

Exercise 4.15. *Prove that a simple graph G is connected if and only if G has a spanning tree.*

Proof. If G has a subgraph G_1 which is a spanning tree, then all vertices in G are connected by a path in the spanning tree G_1. Hence, G is connected.

If G is connected, let's consider a subgraph G_1 that is connected and contains all the vertices and has the least number of edges. If G_1 has a cycle, then there exist 2 vertices a and b that are connected by an edge and another path. Deleting that edge, all vertices are still connected since a and b are still connected. However, by deleting this edge connecting a and b, we obtain a new subgraph which is connected and contains all the vertices with one less edge than G_1 and therefore our initial G_1 is not a connected

subgraph containing all the vertices with the least number of edges. By contradiction, G_1 does not have any cycle. Thus, G_1 is a spanning tree. □

Exercise 4.16. *Prove that if a connected graph G is not a tree, then G has at least three spanning trees.*

Proof. Since G is a connected graph but not a tree, then G has a cycle. From problem 1, G has one spanning tree T_1. Let C denote the cycle in G. Since the tree does not have any cycle, there exists an edge e of C that does not belong to T_1. The endpoints of e are denoted by x and y. The spanning tree includes x and y. Thus, there exists a path D in T_1 connecting x and y in the spanning tree which does not include the edge e.

Now let T_2 be constructed from T_1 by deleting an edge d_1 on the path D and connecting x and y by e. Let T_3 be constructed from T_1 by deleting an edge d_2 on the path D and connecting x and y by e. We can choose d_1 and d_2 different from each other since the path connecting x and y in T_1 has a length of at least 2. If not, the graph T_1 would have multiple edges, and would not be a tree.

Therefore, by this construction, the connected graph G has at least 3 spanning trees. □

The following Problem is related to Lemma 4.7 about the construction of a shortest path tree once a root has been fixed.

Exercise 4.17. *Let v be a vertex in a connected simple and finite graph G. Prove that there exists a spanning tree T in the graph G such that distances to every vertex from v are the same in G and in T.*

Proof. Let $G = (V, E)$ be a graph.

1. We construct the graph G_1 containing the path from v to all vertices with the following constraints:

- First, we construct the shortest path of length 1 from v to all vertices in its neighborhood. Since we are considering simple graph, each path is unique.
- We construct the shortest path between v and x_i whose length is greater than or equal to 2 with the following constraints: If this shortest path between v and x_i includes the edge x_jx_i connecting x_j to x_i, then this shortest path is constructed by the shortest path from v to x_j and the edge x_jx_i (we do this in order to make sure there is no cycle that is constructed by two shortest path of the same shortest length from v to x_j).

We now prove that the path between v to x_i is unique by induction: We have x_i that are neighborhood of v, then the paths are unique and of length 1 due to the first point above. Assume that the shortes path from v to x_j is constructed to be unique. Then for any vertex x_i such that the shortest path between v and x_i contains x_j, the shortest path from v to x_j is unique since the path between v and x_i is unique, and the edge between x_j and x_i is also unique. By induction, the shortest path between v and any vertex x_i of graph G is unique. By construction, G_1 includes all the shortest path from v to other vertices, therefore G_1 contains all the vertex in V. Let $G_1 = (V, E_1)$

where E_1 is the set of the edges in all of these shortest paths we constructed. By this construction, the distance between v and each other vertex is the same in G_1 and in G.

2. Let us prove that G_1 is a spanning tree:
- G_1 is connected since for vertices x and y different from v, x is connected to v and y is connected to v, thus x and y are connected.
- G_1 contains all the vertices in the graph.
- Suppose there is a cycle C in G_1. Consider a vertex x_1 in the cycle C, then there are 2 different path from v to x_1. This is a contradiction to our construction and uniquness of the shortest path from v to each vertex. Therefore, by contradiction, G_1 is acyclic.
-Therefore, G_1 is a spanning tree of G that satisfies the distance between v and any vertex is the same in G and in G_1. □

4.5.2 Greedy algorithm

This section has been studied and written by Dam Truyen Duc and by Atsuya Watanabe.

To make Greedy Algorithm, firstly, we propose an algorithm that gives a spanning tree. For a simple connected graph $G = G(V, E)$:

```
Let E = e_1, e_2, e_3, ..., e_m.

begin
T := ∅
for i = 1, 2, 3, ..., m do

begin
if T + e_i does not contain a cycle
then T ← T + e_i
end
output T
end
```

Let's prove that this algorithm gives a spanning tree of connected graph $G(V, E)$.

Proof. By construction, If adding an edge e_1 creates a circle in T, then we do not add it. Therefore, T has no cycles.

If T does not contain all vertices, since G is connected, then there exists a vertex x_j in G and not in T with at least one edge e with one end point is x_j and another endpoint is a vertex in T, and e does not belong to T (otherwise, all the vertices that is not connected to T by any edges would construct a component in G that disjoint from T). $T + e$ contains x_j, which is a leaf, and T does not have any cycle. Thus $T + e$ does not have any cycle and e should have been added to T. This is a contradiction. Therefore, T contains all vertices.

Assume that T is not connected, then T will consists of componets $C_1, C_2, \ldots, C_k$ in which each component is connected. Consider $D = union\ of\ C_2, C_3, ...C_k$. C_1 contains V_1 set of vertices and D contains V_2 set of vertices. As T contains all vertices, the disjoint union of V_1 and V_2 is V. D is not connected with C_1, which means T does not have any edge joining a vertex of C_1 with a vertex of D. If G has an edge e_i that joininting C_1 and D, then e_i would be a bridge from C_1 and D. As bridge is not in any cycle of G, e_i would have been added to T. This is not valid since the spanning tree T is assumed to be completely obtained after the algorithm. Therefore, G does not have an edge joininting C_1 and D, which is again a contradiction since G is connected. Thus, T is connected.

By those properties proved above, T is a connected acyclic graph that includes all of the vertices. Therefore, the proposed algorithm provides T as a spanning tree of G. □

Let's consider the *maximum weight tree* problem.

Maximum weight tree

Let $G = (V, E)$ is a connected graph.

$\omega : E \to \mathbb{R}$. $\omega(e)$ is the weight of the edge e.

For spanning tree T, $\omega(T) = \sum_{e \in T} \omega(e)$. The problem is:

Find a spanning tree of maximum weight.

To solve this problem, let's consider the *Greedy Algorithm.*

Greedy Algorithm

Label the edges of the simple connected graph G(V,E) so that $E = e_1, e_2, e_3, \ldots, e_m$ where
$\omega(e_1) \geq \omega(e_2) \geq \cdots \geq \omega(e_m)$.

begin $T = \emptyset$
for $i = 1, 2, \ldots, m$ do
begin
if $T + e_i$ does not contain a cycle
then $T \leftarrow T + e_i$
Output T
end

By observation, Greedy Algorithm always adds the maximum weight edge in the edges left which does not make a cycle with previously chosen edges. To prove that Greedy Algorithm gives the maximum weight tree, I will prove the proposition below.

Proposition 4.18. *A simple acyclic graph of n vertices and k edges ($k < n$) has exactly $n - k$ components.*

Proof. We will prove by induction: The number of vertices n is fixed. For $k = 1$, The graph contains $n - 2$ vertices which have no edges, and 2 vertices connected each other by one edge. Thus, it has $n - 2 + 1 = n - 1$ components. Therefore, our statement is true for $k = 1$.

Assume that the statement is true for k edges. Consider a simply acyclic graph $G(k + 1)$ with n vertices and $k + 1$ edges ($k + 1 < n$). Then, this graph $G(k + 1)$ is equivalent to graph $G(k)$ of n and k edges plus an edge e connecting 2 vertices of graph $G(k)$. If e connects 2 vertices of the same component of $G(k)$, then that component plus e will make a cycle. which is a contradiction as there is no cycle in graph $G(k+1)$. On the other hand, if e connects 2 vertices of different components of $G(k)$, then those 2 components join into 1 component. Thus, the total number of components of $G(k+1)$ is 1 component less than the total number of components of $G(k)$, $(n-k)-1 = n-(k+1)$ components.

Thus, if the proposition is correct for k edges and n vertices, it is also correct with $k + 1$ edges and n vertices. By induction, this proposition is proved. □

Proposition 4.19. *Let G be a simple connected weighted graph. Then the spanning tree T given by the Greedy algorithm is a maximum weighted spanning tree.*

Proof. A tree of n vertices has $n - 1$ edges. Let the edges of the greedy tree be $e_1^*, e_2^*, \ldots, e_{n-1}^*$, in order of choice. Note that $\omega(e_i^*) \geq \omega(e_{i+1}^*)$ since neither makes a cycle with $e_1^*, e_2^*, \ldots, e_{n-1}^*$ and our algorithm consider the edges with the decreasing order of weight.

Let $f_1, f_2, \ldots, f_{n-1}$ be the edges of any other tree where

$$\omega(f_1) \geq \omega(f_2) \geq \cdots \geq \omega(f_{n-1}).$$

We want to show that

$$\omega(e_i^*) \geq \omega(f_i),\ 1 \leq i \leq n - 1. \tag{4.2}$$

Let's prove by contradiction. Suppose (1) is false, namely, there exist $k > 0$ such that

$$\omega(e_i^*) \geq \omega(f_i),\ 1 \leq i < k \text{ and } \omega(e_k^*) < \omega(f_k).$$

Then, we have the inequality $\omega(e_i^*) \geq \omega(f_i) \geq \omega(f_k) > \omega(e_k^*)$. Thus, in the set$(e_1, e_2, \ldots e_m)$ we ordered before constructing the graph, if f_k is the edge e_{j_1} and e_k^* is the edge e_{j_2}, then $j_1 < j_2$. Thus, f_k should be considered before e_k^* is considered in our algorithm of building the greedy tree. Because of that, each $f_i, 1 \leq i \leq k$ is either one of $e_1^*, e_2^*, \ldots, e_{k-1}^*$ or makes cycle with $e_1^*, e_2^*, \ldots, e_{k-1}^*$. Otherwise one of the f_i (which is considered before e_k^* in our algorithm) would have been chosen in preference to e_k^*.

Let the components of forest $(V, \{e_1^*, e_2^*, \ldots, e_{k-1}^*\})$ be $C_1, C_2, \ldots, C_{n-k+1}$. Since $e_1^*, e_2^*, \ldots, e_{k-1}^*$ does not make any cycle, we use the proposition proved above, and this forest of n vertices and $k - 1$ edges has $n - k + 1$ components. Each $f_i, 1 \leq i \leq k$

has both of its endpoints in the same component, since f_i either makes a cycle is one of the $e_1^*, e_2^*, \dots e_{k-1}^*$. If f_i is one of those edges, then it obviously belongs to one of the component. If f_i makes a cycle with those edges, then it connects 2 vertices that are already connected by a path built with $e_1^*, e_2^*, \dots, e_{k-1}^*$. Thus, f_i connects 2 vertices of one component. This confirms that each $f_i, 1 \leq i \leq k$ has both of its endpoints in the same component.

Let μ_i be the number of f_j which have both endpoints in C_i and let ν_i be the number of vertices of C_i. Then, taking all components into account, we obtain that:

$$\mu_1 + \mu_2 + \dots \mu_{n-k+1} = k \tag{4.3}$$

$$\nu_1 + \nu_2 + \dots \nu_{n-k+1} = n. \tag{4.4}$$

Suppose that $\mu_t < \nu_t$ for all t in $1 \leq t \leq n-k+1$, then $\mu_t \leq \nu_t - 1$ since there are natural numbers. Then,

$$\begin{aligned} k = \sum_{i=1}^{n-k+1} \mu_i \ &\leq \ \sum_{i=1}^{n-k+1} (\nu_i - 1) \\ &\leq \ \sum_{i=1}^{n-k+1} \nu_i - (n-k+1) \\ &\leq \ k-1 \end{aligned}$$

which is a contradiction. Thus, it follows from (4.3) and (4.4) that there exists t such that

$$\mu_t \geq \nu_t. \tag{4.5}$$

Relation (4.5) indicates that there exists compoment C_t such that the number of edges f_j in C_t is larger than or equal to the number of vertices in C_t. Therefore, these edges f_j in C_t must contain a cycle, which is a contradiction since the set f_j we are considerd is a set of edges belong to a spannning tree, which is acyclic.

Therefore, by this contradiction, (4.2) is true, which means we have our constructed graph as the maximum weight graph. From the above, we get the conclusion that Greedy Algorithm gives maximum weight tree. □

4.5.3 Application of graph theory in route search algorithm for route guidance system in automobiles

This section has been studied and written by Bui Tu Ha.

Route Guidance System

Route Guidance System is one of the major elements of travel and transportation management, which contributes to Intelligent Transportation System (ITS). Using maps,

arrows, and/or a voice interface, Route Guidance System provide users with turn-by-turn guidance to a destination. Typical applications of Route Guidance Systems are car navigation system, delivery truck route planning system, fright route finder,...

Route Guidance Systems process route search based on Route Search Algorithm. After origin point, destination point and departure time are set, routes are searched in a road map (road network) according to predefined criteria (e.g. shortest travel time, shortest distance, minimal CO_2 emission, ...). The most common criterion is often the shortest travel time.

Route search problem is treated using the knowledge of Graph Theory. Road network is considered as plane graphs. It is expressed by nodes (vertices) and links (edges / arcs). Nodes indicate road intersections while links indicate road segment between two adjacent intersections. Another element is weight of link. It can also be regarded as impedance or cost of link, for example, travel time or travel distance. Weight of link is determined according to observation by road side infrastructure (e.g. loop-coil / ultrasonic vehicle detector, radio wave beacon), observation by probe vehicle system (e.g. taxi fleets), estimation based on traffic volume, as well as some other information. In addition, if links in road network have no restriction for moving direction, we use undirected graph. Then in many cases, weight on a link is the same in both direction. By contrast, if links have restriction for moving direction (e.g. vehicles must keep left), a directed graph is used. In this case, it is possible that weight becomes different when we move in opposite direction between two adjacent nodes.

Route Search Algorithm

Route Search Algorithm plays a key role in the application of Route Guidance System. Why is it necessary? When the size of road network increases, the number of routes also increases. More specifically, in case of a grid network having x links along horizontal side and y links along vertical sides (x and y are positive integer) , the number of paths between the origin and the destination as shown in the Figure 1 becomes $n = (x + y)!/(x!y!)$. This number can be large, for instance, $n = 70$ if $(x, y) = (4, 4)$; $n = 126$ if $(x, y) = (4, 5)$. Thus, an efficient Route Search Algorithm is necessary.

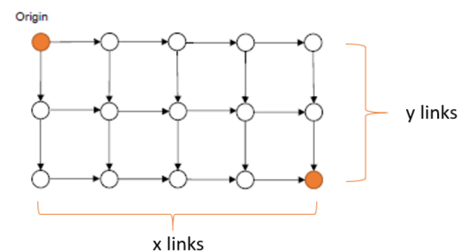


Fig. 4.16. A grid network

Route Search Algorithm is often used to solve shortest path problem, i.e. to find a route (path) between to locations with minimized total weight. There are three main types of shortest path problem:

- Single-Source Shortest Path problem: to find the shortest routes from a source node to all other nodes in the network (typical algorithm: Dijkstra algorithm, Bellman-Ford algorithm)

- Single-Pair Shortest Path problem: to find the shortest route between two pre-determined locations (typical algorithm: A* algorithm). Every known algorithm takes just as long as solving Single-Source.
- All-Pairs Shortest Path problem: to find the shortest routes between every pair of nodes in the network (typical algorithm: Warshall-Floyd algorithm).

The use of Warshall-Floyd algorithm involves matrix calculation and it is difficult to apply this to a large road network. Thus, we do not usually consider this algorithm. In the next part, we shall recall the Dijkstra algorithm (see Section 4.4), then discuss about A* algorithm, which is a revised method of Dijkstra's algorithm. After that, Heap Data Structure will be introduced as a method to mitigate a weakness of Dijkstra algorithm and A* algorithm. In the end, we will study Bellman-Ford Algorithm and compare it with Dijkstra algorithm.

Dijkstra's Algorithm Dijkstra's algorithm can be perceived as the most popular and frequently used route search algorithm for Route Guidance System. To implement this algorithm, we need to assume non-negative edge weight. We recall in Figure 4.17 the Algorithm 4.14 of Section 4.4:

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*

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

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

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

Fig. 4.17. Algorithm 4.14

With this expression, for each discovery number i ($i \geq 0$), we need to update the set of frontier edges, $\mathrm{Front}(G, T_i)$. From the viewpoint of programming, we shall use an one dimensional array to store the set of non-tree endpoints of the frontier edges. Choosing an element in this node set can be an alternative of "Choose the element $e_{i+1} \in \mathrm{Front}(G, T_i)$" in Algorithm 4.14.

Overall, for shortest path problem using Dijkstra algorithm, it is more convenient to classify nodes into three node sets: V(Visited), T (Tentative) and U (Unvisited).

Nodes in set V are the nodes that have been visited with fixed distance and route from origin. The set T is comprised of nodes that have been visited but distance and

route from origin have not been fixed. Accordingly, this set provides the information about frontier edges. The last node set is the set U, which contains nodes that have not been visited and distance and route from origin are unknown.

* Remark: The term “distance” mentioned above implies the value of function d_w in Algorithm 4.14. In other cases, d_w can be regarded as a function that indicates travel time or travel cost from origin.

Based on the definition of set V, T and U, we can express the Dijkstra's algorithm in another way which have underlying meaning similar to the Algorithm 4.14.

Step 0:

- Assign a beginning value of distance (infinity) to all nodes
- All nodes are in set U

Step 1:

- Set the distance of the origin node to zero
- Origin node is moved from set U to set V
- Set the origin node as “base node”

Step 2:

- For the based node, consider all of its unvisited or tentative neighbors and calculate their tentative distance via the current base node.
- For each of these neighbors, compare the tentative distance newly calculated to the tentative distance calculated before.
- If the newly calculated distance is smaller, the tentative distance is updated and the upstream node is changed to the current base node.
- Newly calculated neighbors are moved from set U to set T

Step 3:

- From set T, the node with the smallest tentative distance is extracted and set as new base node.
- This new base node is moved from set T to set V.

Step 4:

- If the base node is the destination node, then stop. The algorithm has finished.
- Otherwise, go to step 2.

Next, let's practice with an example of finding shortest path using Dijkstra's algorithm.

Exercise 4.20. *Search the shortest path from the origin to the destination using Dijkstra's algorithm in Figure 4.18.*

The step-by-step solution is provided in Figures 4.19 and 4.20.

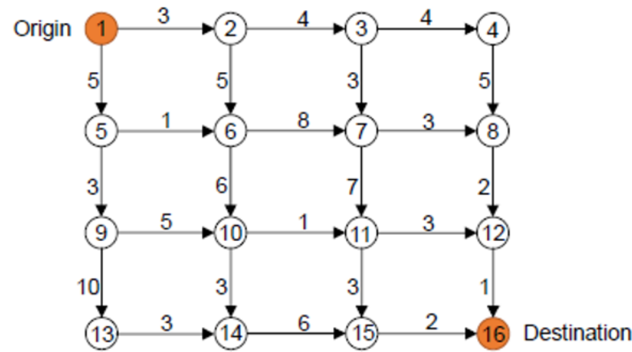


Fig. 4.18. Map for Exercise 4.20

A* Algorithm A* algorithm is proposed in 1968 as an improved algorithm of Dijkstra algorithm. [2] It is an improved method based on Dijkstra algorithm, with modification only in the step 2 of Dijkstra algorithm.

Let n be a node in set T, or equivalently, the non-tree endpoint of a frontier edge: For Dijkstra's algorithm:

$$\text{Tentative distance } D(n) = \text{Tentative distance from origin node}$$

For A* algorithm:

$$\text{Tentative distance } D'(n) = D(n) + \text{Heuristic } h(n)$$

where $h(n)$ is the estimation of the minimum distance from node n to destination node. It must be positive and smaller than the true minimum distance to the destination node. Practically, $h(n)$ is the length of the straight line from node n to destination node using X-Y coordinate.

Heap Data Structure

In the practical case of large road network, the number of nodes in set T can reach several thousands or more. However, in Dijkstra algorithm and A* algorithm, we only need to find one node with the smallest tentative distance (step 3) and the process to search this node consumes the most amount of processing time. To tackle this weakness of Dijkstra algorithm and A* algorithm, we shall use an implementation of priority queue - Heap Data Structure (Binary Heap). In fact, Heap Data Structure can be perceived as a priority tree (see Definition 3.24) of total ordered set with priorities (elements which label vertices) that are numeric values . Heap Data Structure efficiently searches the node with the minimal tentative distance. Based on the tentative distance values, nodes in set T are ordered in a specific way.

In computer science, a *heap* is a specialized tree-based data structure which is essentially an almost complete binary tree that satisfies the heap property:

[2] Hart P.E., Nilson N.J. and Raphael B. (July 1968), "A Formal Basis for the Heuristic Determination of Minimal Cost Paths". IEEE Transaction on Systems Science and Cybernetics 4 (2): 100-107

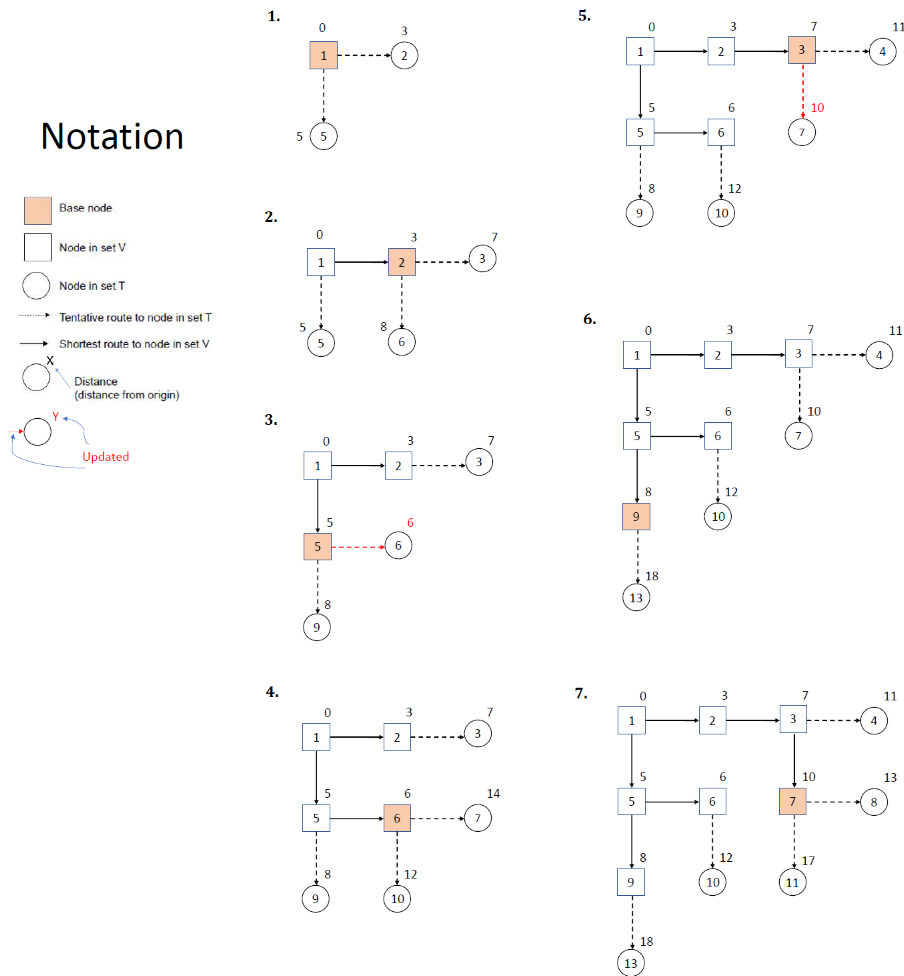


Fig. 4.19. First part of the solution to Exercise 4.20

- In a max heap, for any given node C (child node), if P is a parent node of C, then the key (the value) of P is greater than or equal to the key of C.
- In a min heap, the key of P is less than or equal to the key of C. The node at the "top" of the heap (with no parents) is the root node.

For Dijkstra's algorithm and A* algorithm, we shall use the property of min heap as heap condition.

Heap condition: Tentative distance of parent node is less than or equal to tentative distance of each child node.

When a node is added to set T (and distance of some tentative nodes is updated), heap is updated as follows:
(1) Insertion of a new node

- (1)-1: Add the new node to the last position of the bottom level

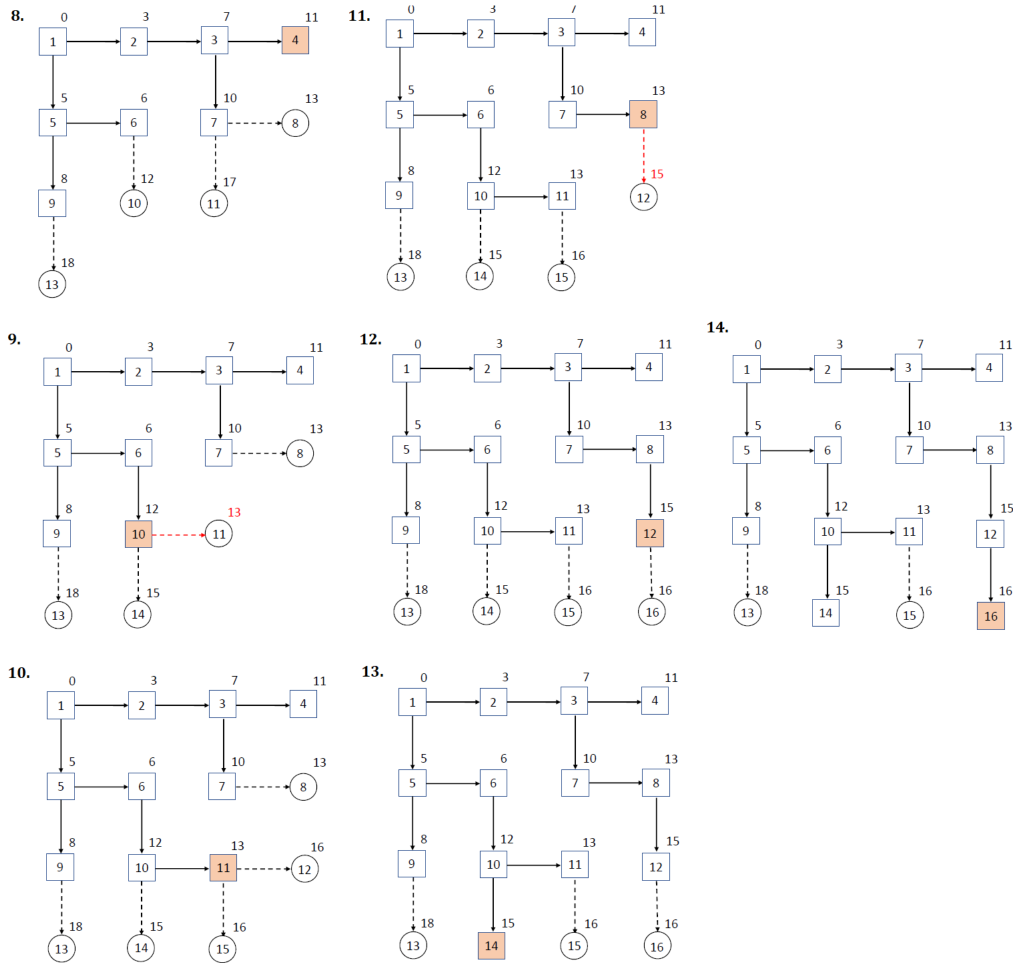


Fig. 4.20. Second part of the solution to Exercise 4.20

- (1)-2: Compare the added element with its parent. If they are in the correct order (according the heap condition), then stop. If not, swap the node with its parent. (Note that the number of swap is at most the height of the tree ($\log_2 |V|$) and this takes $O(\log(|V|))$ time.

  Step (1)-2 is repeated until the structure satisfies the heap condition.

(2) Deletion of root node, which is the node with the smallest tentative distance. In addition, the removed node will be the next base node.

(3) Bubble-down operation is performed to restore the structure

- (3)-1: Move that last node on the bottom level to the root of the heap
- (3)-2: Compare the new root with its children. If they are in the correct order, stop. If not, swap it with its smaller child. Note that the number of swap is at most the height of the tree ($\log_2 |V|$) and this takes $O(\log(|V|))$ time.

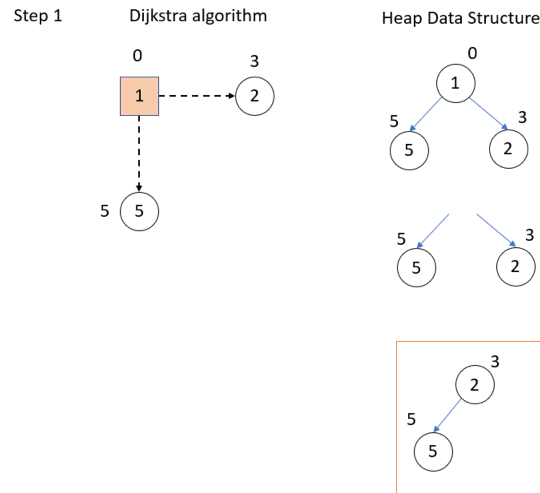


Fig. 4.21. Step 1

Step (3)-2 is repeated until the structure satisfies the heap condition.

These operations are represented respecively in Figures 4.21, 4.22, and 4.23.

* Remark: How to find the target node for comparison (i.e to find the parent node or a child node) ?

Binary heap can be memorized by using a single dimensional array. The tree nodes have a natural ordering: row by row (starting from the root node) and moving left to right within each row. If there are n nodes, this ordering specifies their positions $1, 2, ..., n$ in the array. Moving up an down the tree is easily simulated on the array, using that node number j has parent $\lfloor j/2 \rfloor$, and children $2j$ and $2j + 1$.

Bellman-Ford Algorithm

Beside Dijkstra's algorithm, Bellman-Ford (BF) algorithm is another Single-Source Shortest Path (SSSP) algorithm. However, BF algorithm is not ideal for most SSSP problems because of its high time complexity, which is $O(|V| \cdot |E|)$. Meanwhile, Dijkstra's algorithm using binary heap is much faster with time complexity $O((|V| + |E|) \log |V|)$.

Nonetheless, Dijkstra's algorithm has one disadvantage: it can fail when negative link weights exist, see Figure 4.24. This problem can be mitigated by the use of BF algorithm, which can treat negative value of link weight. However, we still have to assume no cycle of negative length for BF algorithm. After understanding how BF algorithm works, see Figure 4.25, you may find out that it is because a cycle of negative length possibly leads to infinite reduction of distance (to $-\infty$) and consequently, some vertices

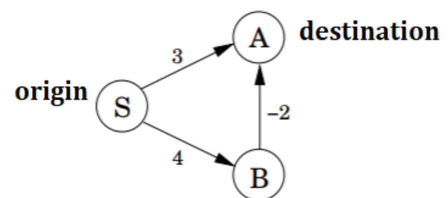


Fig. 4.24. One negative weight

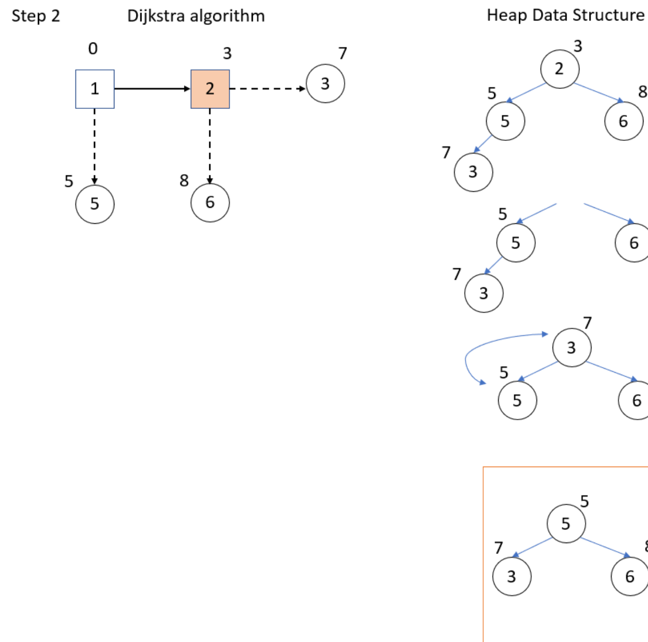


Fig. 4.22. Step 2

can not be visited.

Firstly, distance from s to each vertex u is set to infinity. When the total number of vertices in G is $|V|$, we do (at most [3]) $|V|-1$ iterations and each iteration will update along all edges. An example of Bellman-Ford algorithm is shown in Figure 4.26.

This section is based on the following references:

1. Tomio Miwa (2020), *Route Search Method*, Intelligent Transportation System Lecture, Nagoya University.

2. T. Cormen, *Introduction to algorithms*, United States of America: The MIT Press Cambridge, Massachusetts London, England. pp. 151–152. 2009.

3. P. Black, Entry for heap in Dictionary of Algorithms and Data Structures. Online version. U.S. National Institute of Standards and Technology, 14 December 2004. Retrieved on 2017-10-08 from https://xlinux.nist.gov/dads/HTML/heap.html

4. D. Walden (January 2008), The Bellman-Ford algorithm and "distributed Bellman-Ford", https://www.researchgate.net/publication/250014977_THE_BELLMAN - FORD_ALGORITHM_AND_DISTRIBUTED_BELLMAN-FORD

5. https://code.google.com/archive/p/eclipselu/downloads

[3] We sometimes make the algorithm stop earlier if nothing improves with more iterations.

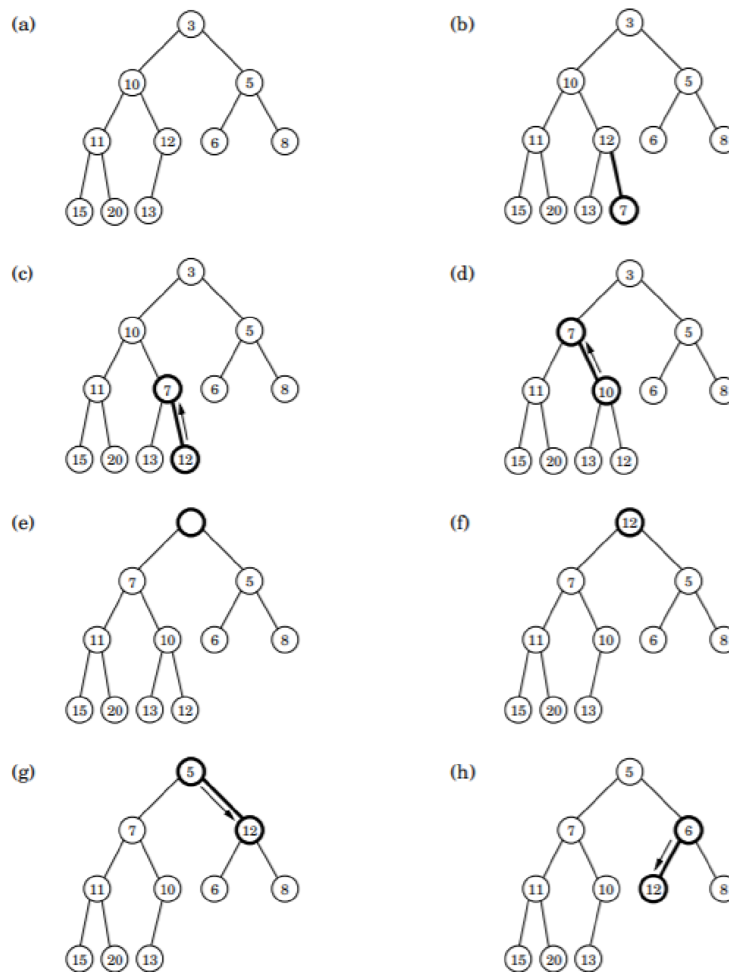


Fig. 4.23. (a) A binary heap with 10 elements. Only the key values are shown. (b)-(d) The intermediate “bubble-up” steps in inserting an element with key 7. (e)-(g) The “sift-down” steps in a delete-min operation [5]

4.5.4 Floyd—Warshall algorithm

This section has been studied and written by Liyang Zhang.

Introduction

Here I introduce an algorithm to find the path with the smallest edge between every pair of vertices at once in an oriented edge-weighted finite graph with negative weight edges permitted, but without any negative weight circle.

Dijkstra algorithm cannot deal with graphs with negative weighted edges, since it relies on a fact that if all weights are non-negative, adding an edge can never make a path shorter[1]. It fails to find the cheaper path when a largely negative edge is hidden

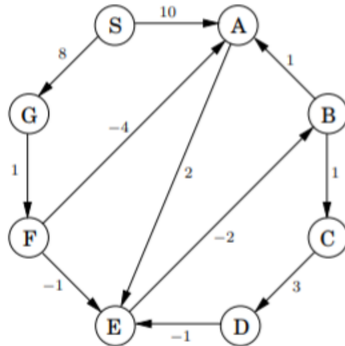
```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)


procedure update((u, v) ∈ E)
dist(v) = min{dist(v), dist(u) + l(u, v)}
```

Fig. 4.25. The Bellman-Ford Algorithm for Single-Source Shortest Path [5]



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Fig. 4.26. The Bellman-Ford Algorithm illustrated on a sample graph [5]

after a largely positive edge [1], for example the path from point 2 to 3 in Figure 4.27 (a).

Now let me introduce Floyd—Warshall Algorithm, which is applicable in oriented edge-weighted finite graphs with negative weight edges permitted, but without any negative weight circle, with the graph of Figure 4.27 (a) as example.

My main references are [2], [3], and [4].

Algorithm

- Input: an oriented edge-weighted finite graph G, vertices labelled by consecutive integers from 1 to $n = \text{order}(G)$.

- Output: a matrix $M \in \mathbb{R}^{n \times n}$ with M_{ij} showing the least weight among paths from vertex i to vertex j,
  and a set of shortest routes from each vertex to each vertex.

Fig. 4.27. (a) an oriented graph with negative edges; (b) the matrix M^0 listing the weight of its all edges.

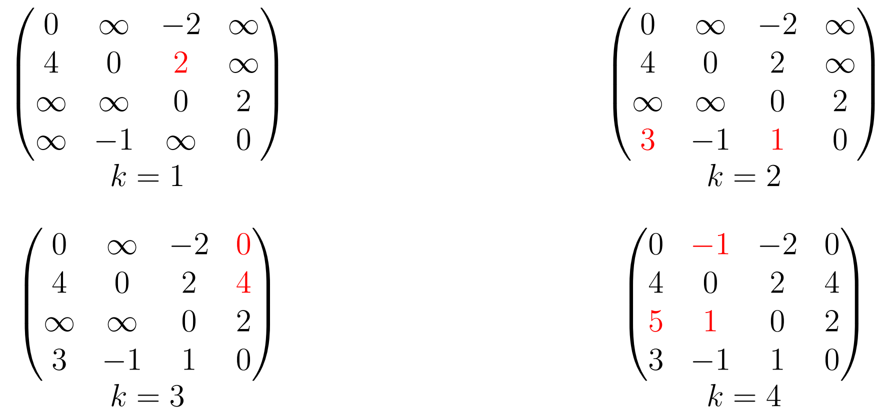


Fig. 4.28. Floyd–Warshall Algorithm on the graph of Figure 4.27 (a).
After examining all points, at $k = 4$, we have the final result.

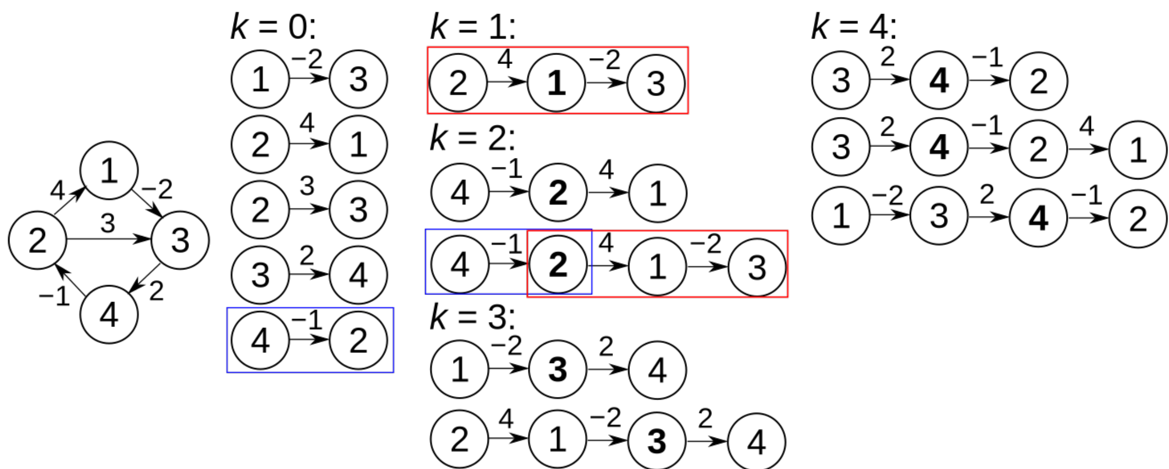


Fig. 4.29. the temporary cheapest paths between each pair of vertices after each step[2].
The final cheapest paths are in the last step $k = 4$.

1.
- Write down a matrix $M \in \mathbb{R}^{n\times n}$, with 0 on the diagonal, and the weight of edge from the ith vertex to the jth one on M_{ij}, and ∞ on M_{ij} if there is no edge from the ith vertex to the jth one. [*Example in Figure 4.27 (b)*]
- Save down each edge as a route. [*Example in Figure 4.29 ($k = 0$)*]

2.
- For each vertex k from 1 to n, For each ordered pair of vertices (i, j) with i and j from 1 to n different from k, If $M_{ij} > M_{ik} + M_{kj}$,
- Then let $M_{ij} = M_{ik} + M_{kj}$, [*Example in Figure 4.28*]
- Delete the old route from i to j,
- And save down the new route as the route from i to k that we already have connected to the route from k to j that we already have. [*Example in Figure 4.29 ($k = 1, 2, 3, 4$)*]
  Notice that in the case of our example, as shown in Figure 4.29, at $k = 1$, we have replaced the route $2 \to 3$ by $2 \to 1 \to 3$, and $2 \to 1 \to 3$ is now the route from 2 to 3 that we already have; then, at $k = 2$, we find that it is shorter to go from 4 to 3 via 2. Here, we need to add the route $4 \to 2 \to 1 \to 3$ instead of $4 \to 2 \to 3$, as stated in the algorithm. Indeed, $4 \to 2 \to 1 \to 3$ is shorter than $4 \to 2 \to 3$, and the algorithm gives the correct thing.

3.
- If for any i from 1 to n, $M_{ii} = 0$, then return M;
- Else if $M_{ii} < 0$, then the input graph has at least one negative circle starting and ending at the vertex i.
  Since the diagonal elements are initially 0 and only decreases in the following steps, $M_{ii} > 0$ is impossible.

Justification

The second step means that if the cheapest path from i to j not passing $k, k+1, ..., n$ is more expensive than the cheapest one passing k but not passing $k+1, ..., n$, then we replace it. After the loop from 1 to n, all possible midway points are examined once. In a figure without negative loop, paths passing a point more than once cannot be cheaper than its concatenated one, so it is enough to examine all points once and only once.

References:

[1] www.quora.com/Why-doesnt-Dijkstra-work-with-negative-weight-graphs

[2] en.wikipedia.org/wiki/Floyd-Warshall_algorithm

[3] ithelp.ithome.com.tw/articles/10209186 (in Chinese)

[4] youtube.com/watch?v=4OQeCuLYj-4 (video of 4.5 minutes)

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

$$\kappa_V(G) \leq \kappa_E(G) \leq \delta(G). \tag{5.1}$$

The precise proof is provided in Section 5.4.1. 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.

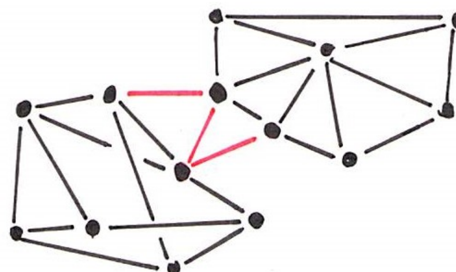


Fig. 5.1. $\kappa_V(G) = 2$ and $\kappa_E(G) = 3$

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.

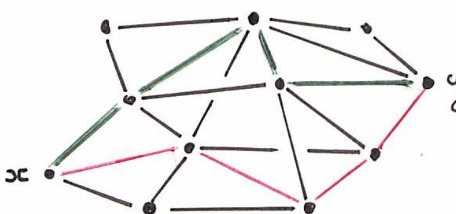


Fig. 5.2. 2 internally disjoint paths

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.*

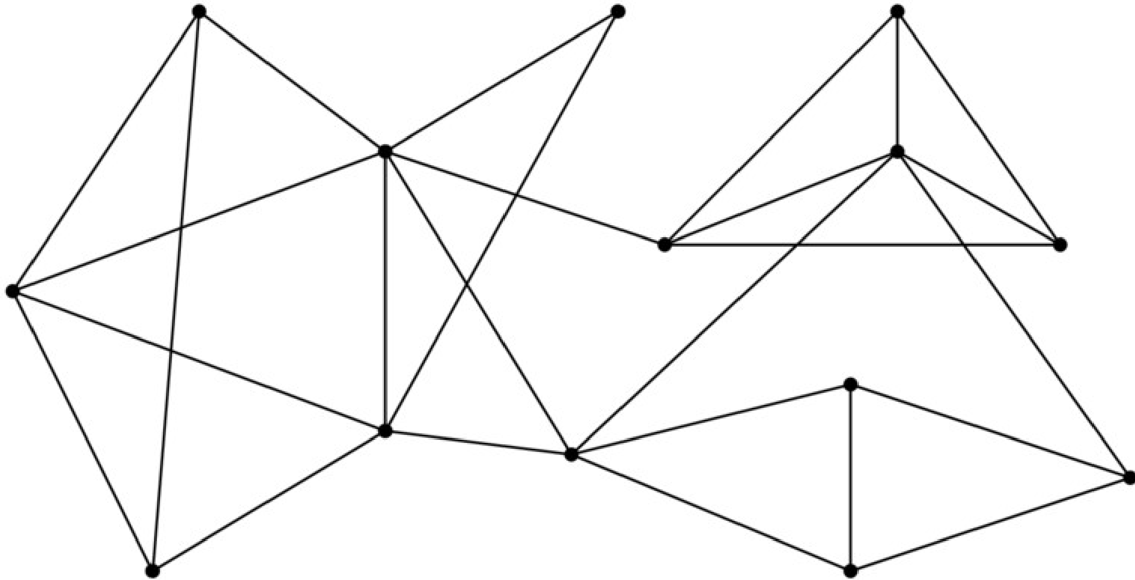


Fig. 5.3. A 2-connected graph on which the above statement can be observed

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].

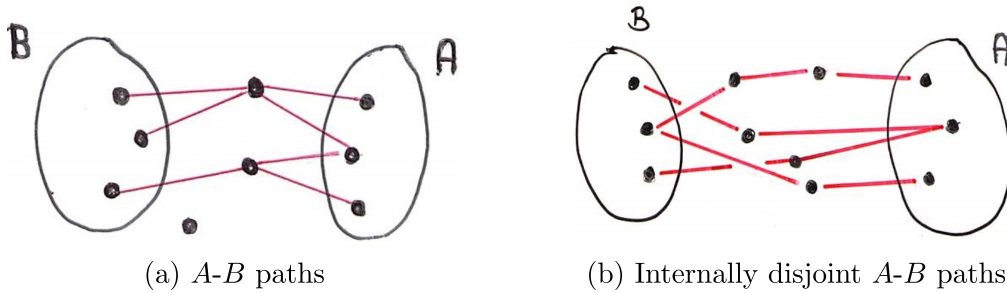
5.2 Menger's theorem

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

Fig. 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*[1]*. 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.

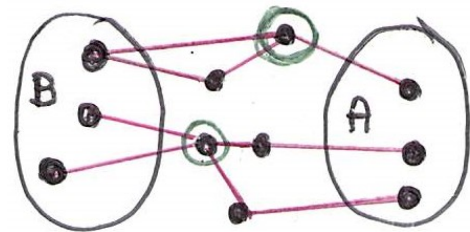


Fig. 5.5. A-B separator (in green)

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.

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

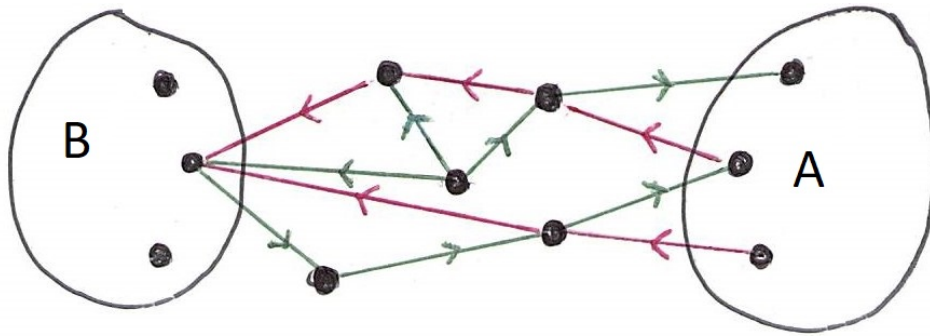


Fig. 5.6. A-B paths with orientation (in red)

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 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 [15] 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*

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

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.

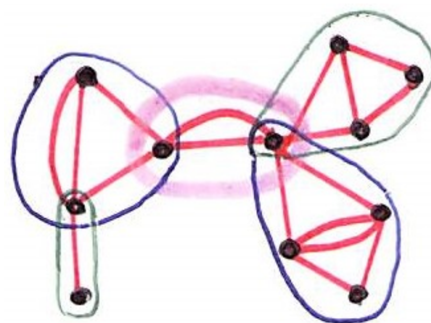


Fig. 5.7. 1 graph, 5 blocks

Lemma 5.12. *Let G be a 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.

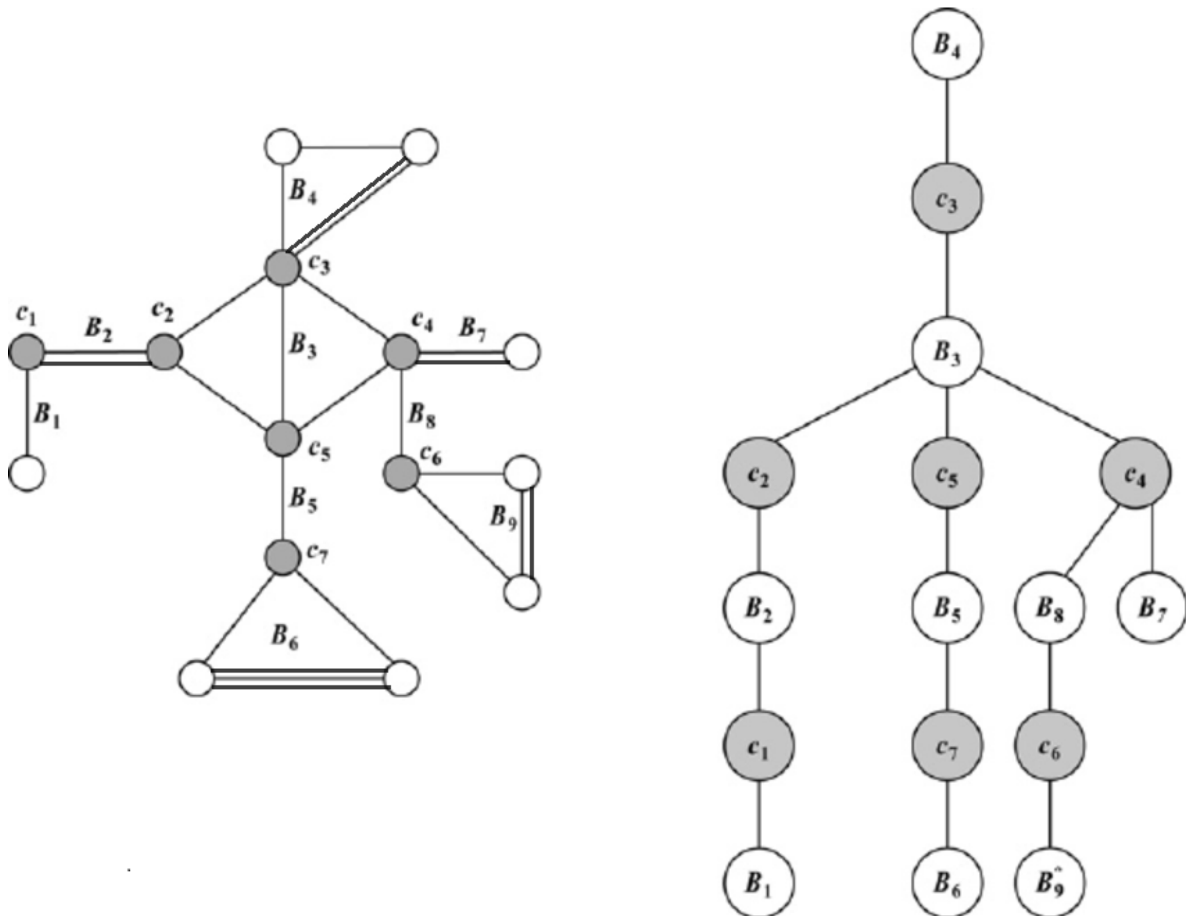


Fig. 5.8. A connected graph and its block-cutpoint graph

5.4 Appendix

5.4.1 Some inequalities

.

The material of this section has been studied and written by Quang Nhat Nguyen and Arata Suzuki. Its aim is to prove the two inequalities contained in (5.1).

Lemma 5.13. *For any undirected and loopless graph $G = (V, E)$ one has:*

(i) $\kappa_E(G) \leq \delta(G)$,

(ii) $\kappa_V(G) \leq \kappa_E(G)$.

Proof. (i) Let us assume that the minimum number of edges whose removal can disconnect $G(V, E)$ is:

$$\kappa_E(G) = k > \delta(G). \tag{5.2}$$

Consider a vertex $x_i \in V$ satisfying: $\deg(x_i) = \delta(G)$. If one removes all the edges $e \in E$ satisfying: $e = (x_j, x_i), x_j \in V$, the vertex x_i will then have degree 0 and thus is disconnected from G. The number of such edges e mentioned above is: $k' = \deg(x_i) = \delta(G)$. This number is smaller than the number of k which was assumed to be the minimum and thus contradicts our initial assumption. Therefore: $\kappa_E(G) \leq \delta(G)$.

(ii) Denote $E^* \subset E$ a set of edges that correspond to the edge connectivity $\kappa_E(G)$. This means that if one denotes $\#(E^*)$ as the number of elements in E^* then: $\#(E^*) = \kappa_E(G)$. Denote $V^* \subset V$ as the set of endpoints of all of edges in E^*, and let G^* be a subgraph of G and defined by the sets V^* and E^*.

If we consider the case that between any two vertices in V^* there can be a maxmimum of one edge in E^*, then all possible values for the number of elements in V^*, denoted as $\#(V^*)$, have to be within the range:

$$\kappa_E(G) \leq \#(V^*) \leq 2\kappa_E(G) \tag{5.3}$$

For any graph G^* in this case, there is always a choice of $\kappa_E(G)$ vertices such that they are the endpoints of all edges in E^*. If one removes these vertices from G^*, all of the edges in E^* will be removed as well. If this happens, the graph G will then become disconnected because the edges in E^* correspond to the edge connectivity $\kappa_E(G)$. It follows that $\kappa_V(G)$ is smaller than $\kappa_E(G)$.

In the case one allows more than one edge between two vertices, the minimum possible value of $\#(V^*)$ may become smaller than $\kappa_E(G)$. If it is, the number of vertices that are needed to be remove to make G disconnected will be equal to this minimum possible value, which is even smaller than $\kappa_E(G)$.

The above fact has proved that there is a choice of $\kappa_E(G)$ (or less) vertices whose removal can either disconnect G or reduce it to a 1-vertex graph. Therefore: $\kappa_V(G) \leq \kappa_E(G)$. □

Chapter 6

Optimal traversals

In Definition 3.13 a graph traversal was introduced as the process of visiting systematically each vertex in a graph. This definition can naturally be extended to the process of visiting systematically all edges of a graph. In this chapter, we discuss some problems which often reduce to finding an optimal traversal, under some constraints. Having more tools available, we also revisit or extend some results mentioned earlier.

6.1 Eulerian trails

Eulerian trails have been introduced in Definition 1.24 and correspond to a trail containing all edges of a graph, but once and only once. On the other hand, vertices can be visited more than once. If the trail is closed one speaks about an Eulerian tour.

Eulerian tours are intimately linked to the Seven Bridges of Königsberg's problem, see [16]. The negative resolution of this problem by Leonhard Euler in 1736 laid the foundations of graph theory. With the notation introduced so far, the problem consists in establishing if the graph on the right of Figure 6.1 is an Eulerian graph (a graph with an Eulerian tour).

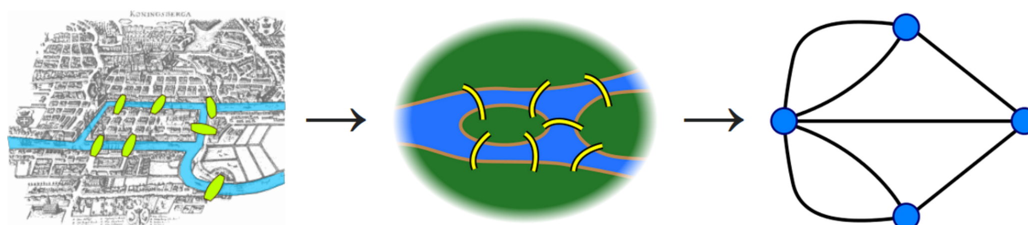
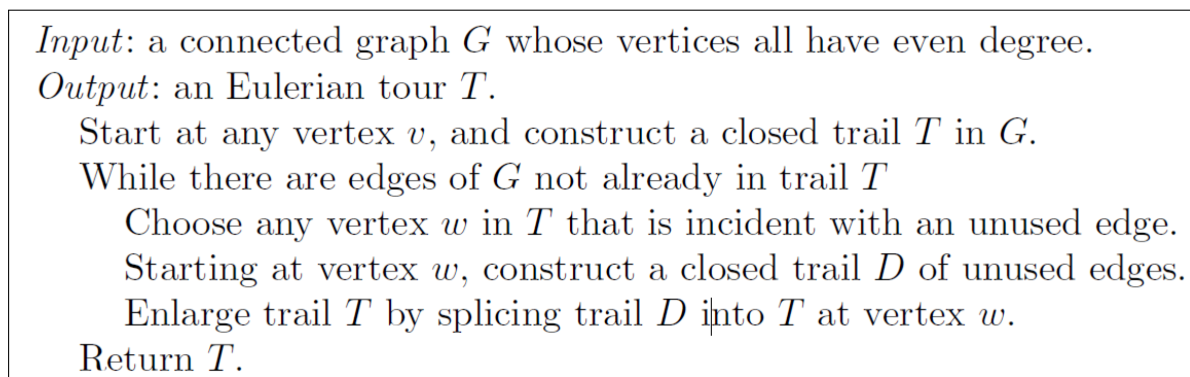


Fig. 6.1. A map of Königsberg and the corresponding graph, see [16]

As already mentioned, the answer is negative, since a characterization of Eulerian graphs require that all its edges have an even degree, see Theorem 1.25. On the other hand, for an unoriented finite graph whose vertices all have an even number of edges, a rather simple algorithm exists for finding one Eulerian tour. This algorithm is provided

in Figure 6.2. By one minute of thought, one easily concludes that this algorithm is correct if and only if every vertex has a even degree.



Input: a connected graph G whose vertices all have even degree.
Output: an Eulerian tour T.
Start at any vertex v, and construct a closed trail T in G.
While there are edges of G not already in trail T
Choose any vertex w in T that is incident with an unused edge.
Starting at vertex w, construct a closed trail D of unused edges.
Enlarge trail T by splicing trail D into T at vertex w.
Return T.

Fig. 6.2. Eulerian tour algorithm, from Algorithm 6.1.1 of [GYA]

Let us still mention a few extensions of the above result. First of all, Eulerian trails which are not closed can also be useful. Indeed, such trails would correspond to trail visiting all edges of the graph, but with an initial point and a final point which could be different. With this weaker requirement the following result can rather easily be obtained, see also [GYA, Thm. 6.1.1].

Theorem 6.1. *A connected, undirected and finite graph admits an open Eulerian trail if and only if it has exactly two vertices of odd degree. Furthermore, the initial and the final vertices of any Eulerian trail must be the two vertices of odd degree.*

Two additional results exist explicitly for directed graphs. For completeness, we state them, and leave the proofs as an exercise. Recall that the notions of indegree and outdegree of a vertex in a directed graph have been introduced in (2.1) and (2.2).

Theorem 6.2. *(i) A connected, directed and finite graph $G = (V, E)$ is Eulerian if and only if $\deg_{\text{in}}(x) = \deg_{\text{out}}(x)$ for any $x \in V$.*

(ii) A connected, directed and finite graph $G = (V, E)$ admits an open Eulerian trail if and only if there exist $x, y \in G$ with $\deg_{\text{in}}(x) + 1 = \deg_{\text{out}}(x)$, $\deg_{\text{in}}(y) = \deg_{\text{out}}(y) + 1$, and otherwise $\deg_{\text{in}}(z) = \deg_{\text{out}}(z)$ for all $z \in V \setminus \{x, y\}$.

Note finally that extensions of these results to infinite graphs are not so trivial. Indeed there exist infinite graphs with vertices of even degree everywhere but which do not admit the natural extension of an Eulerian tour. Additional information and some references on this infinite problem can be found on [16].

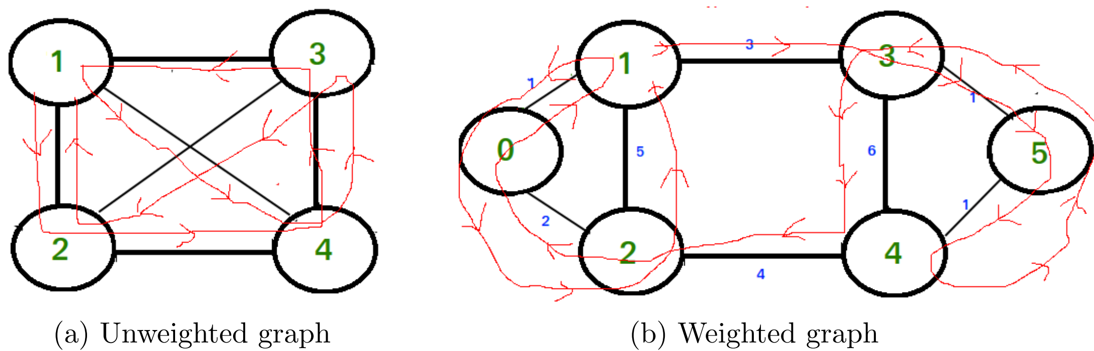
6.2 Postman tour

In the previous section, it was possible to visit all edges of a connected, undirected and finite graph once and only once if and only if all vertices had an even degree. What

about a graph with vertices having arbitrary degrees ? It might not be possible to visit all edges without visiting some twice, or more, but it is certainly possible to visit all of them at least once. In that respect the following definition is natural.

Definition 6.3 (Postman tour)**.** *A* postman tour[1] *on a connected and finite graph is a closed walk that uses each edge of the graph at least once. If the graph is endowed with edges weight, an* optimal postman tour *is a postman tour with the minimum total edge-weight.*

Note that if the graph is not endowed with specific weight, one can always consider that a weight 1 is associated with each edge, and in this case the optimal postman tour corresponds to a shortest postman tour. Two examples of optimal postman tour are presented in Figure 6.3. Let us also remind that a somewhat related question has already been investigated in Section 4.4, when the minimum spanning tree problem was considered. However, the aim is different since the postman has to visit all edges. On the other hand, if the graph is an Eulerian graph, one easily observes that the solution of the optimal postman tour is simply given by the sum of the weights on the edges. Indeed, any Eulerian tour would visit all edges once, each of them giving its contribution to the total weight.



(a) Unweighted graph (b) Weighted graph

Fig. 6.3. Two optimal postman tours, from [17]

There exists an algorithm for solving the optimal postman problem, which is presented in Algorithm 6.5 for undirected graph. The directed version is slightly more complicated. The idea behind the algorithm is to artificially add some weighted edges between vertices with odd degrees, and choose the edges such that these additional weights are kept to a minimum value. At the end of the construction, any Euler tour can be chosen and it has the minimum weight. In order to understand the construction a few more definitions are necessary.

Definition 6.4 (Matching, perfect matching)**.** *A* matching *in a graph $G = (V, E)$ is a set $F \subset E$ such that no two edges in F have a common endpoint. A* perfect matching

[1] Also called *Chinese postman tour*, in honor of the Chinese mathematician Mei-ko Kwan (also translated Meigu Guan) who introduced the problem in 1962.

in a graph G is a matching F in which every vertex of G is one endpoint of an element of F.

A matching and a perfect matching are represented in Figure 6.4. In the construction below, the notion of perfect matching will appear in a complete graph. More precisely, a *complete graph* is an undirected graph in which every pair of distinct vertices is connected by a unique edge. The complete graph with n vertices is often denoted by K_n and possess $n(n-1)/2$ edges. For such a graph, perfect matching are easily represented, see Figure 6.5 for K_6. The number of different perfect matching for K_n can be computed and corresponds to $(n-1)!!$. Here, the notation $n!!$ denotes the double factorial or semifactorial function. The expression of this function is slightly different for n odd or n even, namely for n even one has

Fig. 6.4. A matching, a perfect matching

$$n!! = \prod_{k=1}^{\frac{n}{2}} (2k) = n(n-2)(n-4)\cdots 4 \cdot 2$$

while for n odd one has

$$n!! = \prod_{k=1}^{\frac{n+1}{2}} (2k-1) = n(n-2)(n-4)\cdots 3 \cdot 1.$$

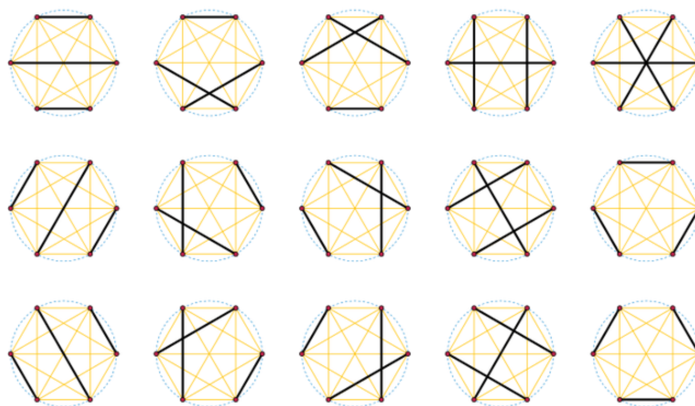


Fig. 6.5. Perfect matching for K_6, from [18]

Note that in the following algorithm, we assume that the graph is not Eulerian, since otherwise any Eulerian tour is an optimal postman tour, and there is no need for any algorithm.

Algorithm 6.5 (Optimal postman tour)**.** *Let G be a connected, finite, undirected and non Eulerian graph.*

(i) Determine the set $S = \{x_1, x_2, \dots, x_n\}$ *of all vertices with odd degree (n is always even),*

(ii) Construct the complete graph K_n *on the vertices* S*, which means the graph with all edges* e_{jk} *with* $i(e_{jk}) = (x_j, x_k)$ *for* $x_j, x_k \in S$ *and* $x_j \neq x_k$*,*

(iii) For $x_j, x_k \in S$*, find in* G *the path* P_{jk} *between* x_j *and* x_k *with a minimal weight* ω_{jk}*, and assign this weight* ω_{jk} *to the edge* e_{jk}*,*

(iv) Determine a perfect matching F_{perfect} *in* K_n *(containing* $n/2$ *edges) with the requirement that the total edge-weight of* F_{perfect} *is a minimum among all perfect matching,*

(v) On the graph G *add all weighted paths* P_{jk} *corresponding to edges* e_{jk} *in* F_{perfect}*. This augmented graph, denoted by* G^**, is an Eulerian graph,*

(vi) Choose any Eulerian tour in G^**; it is an optimal postman tour.*

Note that in the above algorithm, the method for choosing the perfect matching has not been discussed yet. We also mention that there exist several extensions of this problem. For example, different weights can be considered on an edge whenever this edge is visited several times. The directed version of the postman tour exists, and also the windy version. We refer to [GYA, Sec. 6.2] for other extensions, and to internet for numerous related problems.

6.3 Hamiltonian paths and cycles

Recall that Hamiltonian cycles and Hamiltonian graphs have already been introduced in Definition 1.23. More generally, one sets:

Definition 6.6 (Hamiltonian path). *A* Hamiltonian path *is a path in a graph that contains all vertices of the graph. If the path is closed, one speaks about* a Hamiltonian cycle*, and whenever a graph admits a Hamiltonian cycle, one calls it a* Hamiltonian graph*.*

Note that for a cycle, all vertices can be visited only once, except the initial endpoint and the final endpoint which have to coincide. Let us also mention an easy observation: loops or undirected multiple edges do not change the property of a graph of being a Hamiltonian graph or not. For directed graph, adding multiple edges can change the situation if one adds edges with the reserved orientation. Quite surprisingly, looking for Hamiltonian cycles turns out to be much more complicated than

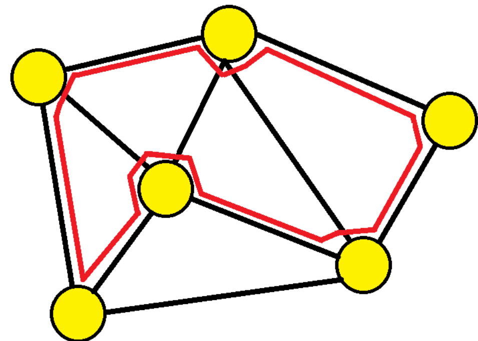


Fig. 6.6. A Hamiltonian graph

looking for Eulerian tours. There is no simple characterization of Hamiltonian graphs and there is no fast algorithm for determining Hamiltonian paths. However, there exit sufficient conditions for a graph to be Hamiltonian that apply to a large class of graphs. There also exist conditions which show that a graph can not be a Hamiltonian graph.

Let us start by mentioning some easy rules which can be used for showing that a graph is not Hamiltonian. It is based on the observation that only two edges adjacent to a vertex can be used in a Hamiltonian cycle. These rules are:

(i) If a vertex x has degree 2, both incident edges must be used in any Hamiltonian cycle,

(ii) During the construction of a Hamiltonian cycle, no cycle can be formed until all vertices are visited,

(iii) If two edges of a given vertex have to be used for a Hamiltonian cycles, then all the other adjacent edges can be disregarded.

The justification of the rules (i) and (iii) are quite clear. For (ii), it is enough to observe that whenever a cycle is created, its initial point and its final point have to be the same, which means that this vertex is visited twice. If this cycle is not the Hamiltonian cycle, then visiting twice a vertex is not allowed. This prevents the existence of any cycle before the final Hamiltonian cycle. Based on these rules, the following exercise can be done.

Exercise 6.7. *Show that the following two graphs are not Hamiltonian graphs.*

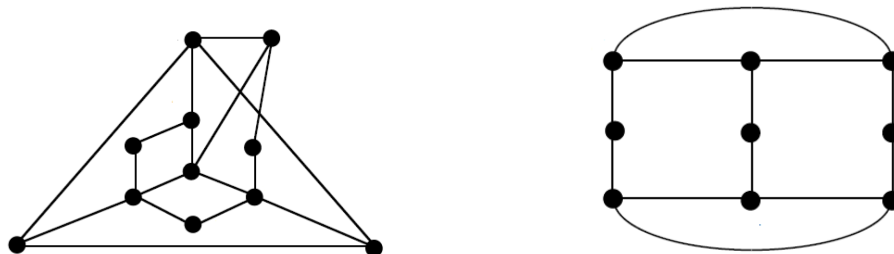


Fig. 6.7. Two non-Hamiltonian graphs, from [GYA, Sec. 6.3]

Another interesting exercise shows that Hamiltonian cycles and Hamiltonian paths are indeed different.

Exercise 6.8. *Prove that the Petersen graph presented in Figure 2.12 admits a Hamiltonian path but no Hamiltonian cycle. Some information for this proof can be obtained from[19].*

As mentioned above, there also exist some sufficient conditions for a graph to be a Hamiltonian graph. We provide such a result both for undirected and for directed graphs. Note however that these results are not really efficient for graphs with a large number of vertices: they also require a large number of edges.

Theorem 6.9 (Ore, 1960)**.** *Let G be a simple undirected graph with n vertices and $n \geq 3$. If $\deg(x) + \deg(y) \geq n$ for each pair of non-adjacent vertices x and y, then G is a Hamiltonian graph. In particular, if $\deg(x) \geq \frac{n}{2}$ for any x, then G is a Hamiltonian graph.*

The proof of the above theorem can be found in [GYA, Thm. 6.3.1]. It is not a completely trivial proof, and it is based on a contradiction argument. Let us emphasize an easy and useful consequence of this result: For $n \geq 3$ any complete undirected graph K_n is a Hamiltonian graph. In fact, the number of different Hamiltonian cycles in K_n is $(n-1)!/2$. For this computation, cycles that are the same apart from their starting point are not counted separately.

Theorem 6.10 (Woodall 1972)**.** *Let G be a simple directed graph with n vertices. If for any vertices x and y with no edge from x to y one has $\deg_{\text{out}}(x) + \deg_{\text{in}}(y) \geq n$, then G is a Hamiltonian graph. In particular, if $\deg_{\text{out}}(x) \geq \frac{n}{2}$ and $\deg_{\text{in}}(x) \geq \frac{n}{2}$ for any x, then G is a Hamiltonian graph.*

6.4 The traveling salesman problem

Hamiltonian cycles are related to the famous travelling salesman problem (TSP) which has already been introduced in Section 1.5.2. Recall that this problem consists in determining the shortest Hamiltonian cycle in a given weighted graph. Note that we assume in this section that all weights are non-negative. If the graph is undirected one speaks about the symmetric TSP (sTSP) while if the graph is directed, one speaks about the asymmetric TSP (aTSP). Also, let me remind that *shortest Hamiltonian cycle* means a Hamiltonian cycle with the minimum total weight (it might not be unique). Such a problem appears when a salesman wants to visit n cities once before returning home. The weight on the edges can represent the distance between the cities, or the cost of the transportation. Note that loops do not play any role for this problem, and multiple edges going in the same direction can be avoided by considering always the one with the minimum weight. On the other hand, in directed graphs, two edges going in opposite directions can not be simplified.

As we have seen in the previous section, not all graphs admit a Hamiltonian cycle. There are several ways for avoiding this situations. For example, one can allow some back-and-forths which are not too costly, or complete the graph with artificial edges of arbitrarily large weights. Once a complete graph is obtained, it is sure that Hamiltonian cycles exist.

A solution to the aTSP has been provided in Section 1.5.2. However, as mentioned in Proposition 1.31, if n denotes the number of vertices of the graph, the complexity of the algorithm provided there is of order $O(n^2 \times 2^n)$. For large n, using this approach would require too much time, and therefore this algorithm can not be applied. In such a situation, one should not look for the shortest solution, but to a solution close to the best one. This approach is often based on the following concept:

Definition 6.11 (Heuristic)**.** *A* heuristic *or* heuristic function *is a guideline that helps in choosing from several possible alternatives for a decision step. A* heuristic algorithm *is an algorithm whose steps are guided by heuristics. This is usually achieved by trading optimality, completeness, accuracy, or precision for speed.*

According to [20] the trade-off criteria for deciding whether to use a heuristic for solving a given problem include the following:

1. *Optimality*: When several solutions exist for a given problem, does the heuristic guarantee that the best solution will be found ? Is it actually necessary to find the best solution ?

2. *Completeness*: When several solutions exist for a given problem, can the heuristic find them all ? Do we actually need all solutions ? Many heuristics are only meant to find one solution.

3. *Accuracy or precision*: Can the heuristic provide a confidence interval for the purported solution ? Is the error bar on the solution unreasonably large ?

4. *Execution time*: Is this the best known heuristic for solving this type of problem ? Some heuristics converge faster than others. Some heuristics are only marginally quicker than classic methods.

The simplest sTSP heuristic is based on the nearest neighbour, as shown in Figure 6.9. The leading idea of this algorithm is to always choose the cheapest way to go somewhere. The framework is a complete graph, which can always be realized, as mentioned above. The implementation of this algorithm is very easy, but its performance can be pretty bad, as illustrated in the example of Figure 6.8. Indeed, by applying this algorithm on this graph, one gets a Hamiltonian cycle of total weight of $1,000,003$, while a clever choice would lead to a Hamiltonian cycle of total weight 6. The weakest point of this algorithm is that it does not look for a global minimum weight, but looks for the minimum weight only at every step.

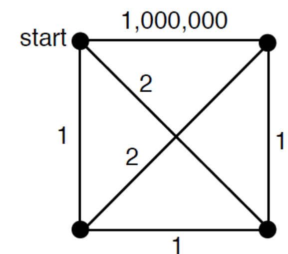


Fig. 6.8. A weighted graph

Another heuristic algorithm for sTSP is based on the minimum spanning tree introduced in Section 4.4. The framework is again a complete graph. It also uses the characterization of Eulerian graphs, namely that any graph with an even degree at every vertex admits an Eulerian tour. The main idea is to follow paths in the minimum spanning tree, as long as possible, and jump to another part of the spanning tree once an already visited vertex is reached. The precise form of this algorithm is provided in Figure 6.10. If we summarize it very briefly, it consists in three steps: 1) Find the minimum spanning tree T^* of the initial weighted graph, 2) Duplicate every edge of T^*, 3) Return a Hamiltonian cycle obtained by taking some shortcuts on the duplicate tree. An illustration of this procedure is provided in Figure 6.11.

Input: a weighted complete graph.
Output: a sequence of labeled vertices that forms a Hamiltonian cycle.
  Start at any vertex v.
  Initialize $l(v) = 0$.
  Initialize $i = 0$.
  While there are unlabeled vertices
    $i := i + 1$
    Traverse the cheapest edge that joins v to an unlabeled vertex, say w.
    Set $l(w) = i$.
    $v := w$

Fig. 6.9. Nearest neighbour algorithm, from Algorithm 6.4.1 of [GYA]

Input: a weighted complete graph G.
Output: a sequence of vertices and edges that forms a Hamiltonian cycle.
  Find a minimum spanning tree T^* of G.
  Create an Eulerian graph H by using two copies of each edge of T^*.
  Construct an Eulerian tour W of H.
  Construct a Hamiltonian cycle in G from W as follows:
    Follow the sequence of edges and vertices of W until the next edge in the sequence is joined to an already visited vertex. At that point, skip to the next unvisited vertex by taking a shortcut, using an edge that is not part of W. Resume the traversal of W, taking shortcuts whenever necessary, until all the vertices have been visited. Complete the cycle by returning to the starting vertex via the edge joining it to the last vertex.

Fig. 6.10. Double tree algorithm, from Algorithm 6.4.2 of [GYA]

So far, we have not discussed the performance of these algorithms. Indeed, as mentioned above, there is always a trade-off between rapidity but also accuracy. In order to discuss the accuracy, more assumptions on the edge-weights have to be imposed. The following assumption is rather natural.

Definition 6.12. *Let $G = (V, E, \omega)$ be a weighted graph, and let e_{xy}, e_{xz} and e_{zy} be any elements of E satisfying $i(e_{xy}) = (x, y)$, $i(e_{xz}) = (x, z)$ and $i(e_z y) = (z, y)$ for some vertices $x, y, z \in V$. Then G is said to satisfy the* triangle inequality *if the following inequality holds:*

$$\omega(e_{xy}) \leq \omega(e_{xz}) + \omega(e_{zy}).$$

Note that this condition is so natural that it is implicitly assumed in most works about TSP. If the edge-weights represent a distance or the cost of a transportation, this condition is satisfied. Now, if we assume that the weighted graph we consider satisfies the triangle inequality, then a comparison between the solution provided by the double tree algorithm and the optimal solution can be inferred.

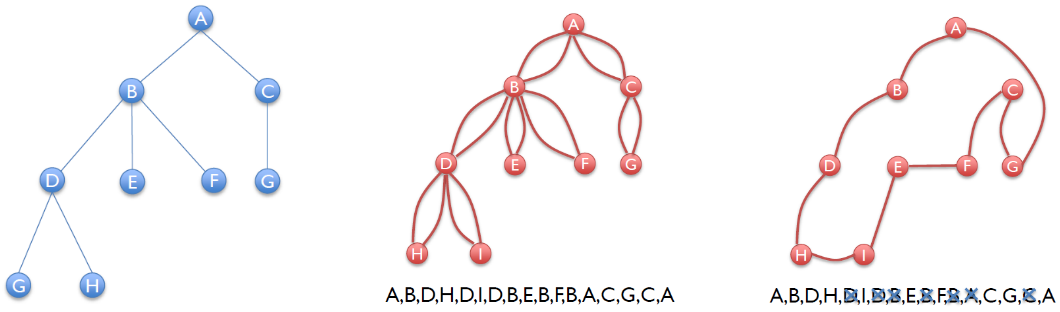


Fig. 6.11. The three main steps of the double tree algorithm

Lemma 6.13. *Let G be a undirected weighted and complete graph satisfying the triangle inequality. Then the solution for the sTSP produced by the double tree algorithm provided in Figure 6.11 is never worst than twice the optimal value.*

Proof. Let C^* be an optimal solution for the sTSP, and let T^* be a minimum spanning tree constructed on G. Let W be the Eulerian tour constructed on the two copies of T^*, as indicated in the double tree algorithm, and let C be the Hamiltonian cycle obtained by the double tree algorithm. If $\omega(X)$ denotes the total edge-weight of the graph X then one has clearly

$$\omega(T^*) \leq \omega(C^*) \quad \text{and} \quad \omega(W) = 2\omega(T^*).$$

In addition, since any shortcut corresponds to an edge in the initial graph, one infers from the triangle inequality that $\omega(C) \leq \omega(W)$. By putting these inequality together one gets:

$$\omega(C) \leq \omega(W) = 2\omega(T^*) \leq 2\omega(C^*)$$

which is the desired inequality. □

We shall now improve the double tree algorithm. Indeed, part of the construction in the algorithm is not optimized: the shortcuts have been chosen rather randomly. But a clever solution has already been introduced in Algorithm 6.5, and it was based on the choice of an optimal perfect matching. Thus, the main steps in the new algorithm will be: 1) Find the minimum spanning tree T^* of the initial weighted graph, 3) Find a minimum perfect matching M^* between the vertices of odd degree of T^*, 3) Return $T^* + M^*$ and take some shortcuts. Let also provide the details:

Algorithm 6.14 (Christofides's algorithm)**.** *Let G be an undirected weighted and complete graph satisfying the triangle inequality.*

(i) Create a minimum spanning tree T^* of G,

(ii) Let O be the subgraph of G induced by the vertices with odd degree in T^*,

(iii) Find a minimum perfect matching M^* in the subgraph O,

(iv) *Combine the edges of M^* and of T^* in an Eulerian graph H,*

(v) *Construct an Eulerian tour W of H,*

(vi) *Construct a Hamiltonian cycle in G from W, as in the double tree algorithm, namely: follow the sequence of edges and vertices of W until the next edge in the sequence is joined to an already visited vertex. At that point, skip to the next unvisited vertex by taking a shortcut, using an edge that is not part of W. Resume the traversal of W, taking shortcuts whenever necessary, until all the vertices have been visited. Complete the cycle by returning to the starting vertex via the edge joining it to the last vertex.*

An illustration of this construction is provided in Figure 6.12.

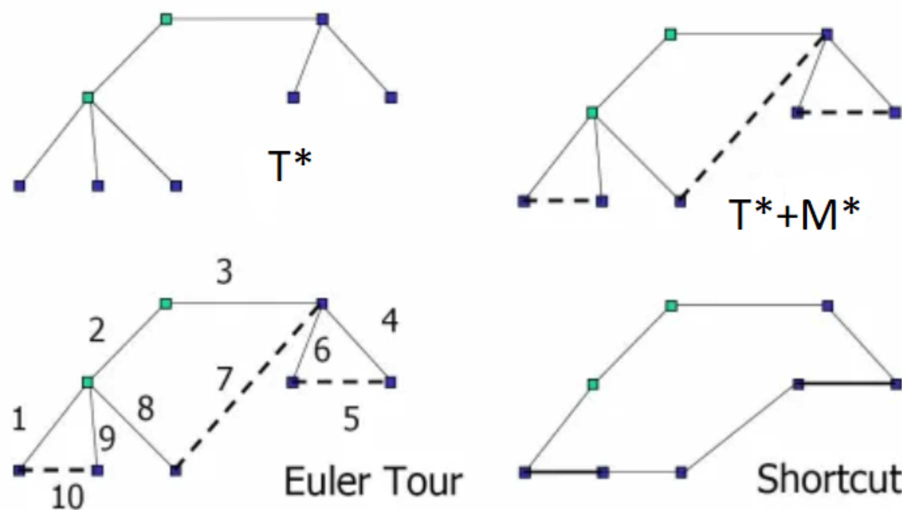


Fig. 6.12. The four main steps of Christofides's algorithm, see [21]

The interest of this improved algorithm can be seen in the following statement:

Lemma 6.15. *Let G be a undirected weighted and complete graph satisfying the triangle inequality. Then the solution for the sTSP produced by Christofides's algorithm provided in Algorithm 6.14 is never worst than $\frac{3}{2}$ times the optimal value.*

Proof. Let C^* be an optimal solution for the sTSP, and let T^* be a minimum spanning tree constructed on G. One always has $\omega(T^*) \leq \omega(C^*)$. Consider O and M^* as described in Algorithm 6.14, and let us show that $\omega(M^*) \leq \frac{1}{2}\omega(C^*)$.

For that purpose, let us enumerate the vertices of O in cyclic order around C^* and call them $\{x_1, x_2, \dots, x_j\}$ with j an even number. Consider then a split of C^* into two sets of paths: the ones starting at x_k with k even, and the ones starting at x_k with k odd. Each of these two sets of paths define a perfect matching of O that matches the two endpoints of each path. The weight of these perfect matching is at most equal to the weight of the corresponding paths, by the triangle inequality. Since these two sets

of paths partition the edges of C^*, one of the two sets has at most half of the weight of C^*. Thus, the corresponding perfect matching has a weight that is also at most half the weight of C^*. As a consequence, the minimum perfect matching can not have a larger weight, which means that $\omega(M^*) \leq \frac{1}{2}\omega(C^*)$, as stated.

Finally, adding the weights of T^* and M^* gives the weight of the Euler tour W, which is thus at most $\frac{3}{2}\omega(C^*)$. Thanks to the triangle inequality, shortcuts do not increase the weight, so the weight of the output is also at most $\frac{3}{2}\omega(C^*)$. □

Let us mention that Christofides's algorithm has been the best heuristic algorithm for more than 30 years for the aTSP. It is only since 2010 that some improvements have been proposed. Apparently, the best current algorithm provides a result which is never worst than 1.4 times the optimal value, see A. Sebö and J. Vygen, Combinatorica 34 (2014), 597–629. Let us however note that the setting is slightly different: all weight are 1 and the graph G is not complete. In this framework one looks for a minimum length closed walk in G that visits every vertex at least once. Equivalently, one looks for the shortest Hamiltonian cycle in the *metric closure of* G. Here, the metric closure of a connected, undirected graph G consists in the complete weighted graph $\bar{G} = (\bar{V}, \bar{E}, \bar{\omega})$ where $\bar{V} = V$, $\bar{E}$ contains all possible edges between the elements of $\bar{V}$, and $\bar{\omega}(e) = 1$ if $e \in E$, while for $e \in \bar{E} \setminus E$ the weight $\bar{\omega}(e)$ is given by the shortest distance in G between the two endpoints of e.

For digraphs, the TSP is much harder, and significants results have only been obtained during the last couple of years. For the aTSP, the framework is a strongly connected digraph with non-negative weights, see Definition 1.18 for the notion of strongly connected. It is also assumed that the graph is Eulerian, which corresponds to the equality of the indegree and the outdegree at every vertex, as mentioned in Theorem 6.2. For such graphs, the best result so far is provided in the next statement (explanations are provided after the statement).

Theorem 6.16 (Theorem 1.1 of [22]). *There is a polynomial-time algorithm for aTSP that returns a tour of value at most* 506 *times the Held-Karp lower bound.*

As mentioned in Section 1.5.2, the Bellman-Held-Karp Algorithm would require too much time for a large graph, and therefore can not be implemented. However, there exists a lower bound for the minimum weight for a TSP (oriented or not), the so-called Held-Karp lower bound (HK). There exist estimates about the difference between the Held-Karp lower bound, and the minimum value of the aTSP, and the current estimate seems to be $\omega(\text{aTSP}) \lesssim 2\,\text{HK}$. The main difference between the content of Theorem 6.16 and this estimate is that the theorem provides a constructive solution for the Hamiltonian cycle, while the lower estimate does not. As already mentioned, investigations on the aTSP are currently taking place, and lots of information (rather advanced) are available on internet.

Chapter 7

Graph colorings

In this chapter we discuss the colorings of graphs obtained by putting colors on the vertices. Such colorings have several practical applications. All graphs in this chapter are undirected, since orientation does not play any role in this context.

7.1 Vertex-colorings

In this section, colors are applied to vertices. In fact these colors can be identified with weights assigned to vertices, and this is how they are often represented: a number assigned to a vertex corresponds to a color put on this vertex. In other context, one speaks about labeled vertices. For internal coherence we shall continue using the notation introduced in Section 1.4 on weighted graphs. In the following definition, C represents a finite set whose elements are called *colors*. Usually, one sets $C = \{1, 2, 3, \ldots k\}$, but a set of colors or a set of letters can also be used. For several applications, it is useful to have a total order on C. For numbers or letters, this is clear, for colors one can just set a bijection between a set of numbers and the set of colors.

Definition 7.1 (Vertex-coloring)**.** *Let $G = (V, E)$ be a loopless graph and let C be a set containing k elements. A* vertex k-coloring *or simply* k-coloring *of G is a map $\omega : V \to C$ such that $\omega(x) \neq \omega(y)$ whenever x and y are the two endpoints of an edge in E.*

Note that some authors would speak about a *proper k-coloring* for this definition, and about a *k-coloring* if the last condition of the definition is not imposed. However, since this condition is always the key condition, it seems natural to include it in the main definition. Note also that the graph is loopless, because any loop would directly invalidate this definition.

Definition 7.2 (Color class)**.** *For a k-coloring of G, the set of all vertices sharing the same color is called* a color class*.*

In mathematical terms, it would be natural to define a color class by $\omega^{-1}(j)$ for any $j \in C$. Indeed, this notation corresponds to the set of all elements x of V such that

$\omega(x) = j$. Since $\omega(x)$ is well defined for any $x \in V$, each vertex belong to one and only one color class. For that reason, the set of color classes defines *a partition* of V. Namely, if we denote by $V_1, V_2, \dots, V_k$ the color classes, then $\cup_j V_j = V$ and $V_j \cap V_\ell = \emptyset$ for any $j \neq \ell$. Let us also emphasize that any $e \in E$ has its endpoints in two different sets V_j.

It is clear that a finite graph always admits a k-coloring for k large enough. On the other hand, for small k it is not clear that a given graph admits a k-coloring since the condition about the endpoint of any edge could be impossible to satisfy. In this context, the following definition is quite natural.

Definition 7.3 (k-colorable and chromatic number)**.** *A loopless graph G is* k-colorable *if it admits a k-coloring. The* vertex chromatic number*, or simply* chromatic number *of G denotes the minimum number k required for a k-coloring of G. This number is denoted by $\chi(G)$, and if $\chi(G) = k$, the graph G is said to be* k-chromatic*. A $\chi(G)$-coloring is called a* minimum coloring.

Clearly, a k-chromatic graph is k-colorable, but it is not $(k-1)$-colorable. A few examples of chromatic numbers can be easily computed. Note that C_n denotes the cycle graph consisting of a cycle with n vertices.

Graph G	$\chi(G)$
no edge	1
bipartite graph	2
non trivial tree	2
cycle graph C_n with n even	2
cycle graph C_n with n odd	3
complete graph K_n	n

Table 7.1: Chromatic numbers

Finding the chromatic number of a given graph is usually not an easy task. In fact finding an upper bound is quite simple, but showing that there does not exist any k-coloring for some small k is a hard problem. Nevertheless some results can be easily obtained. Before presenting them, we introduce the simplest algorithm in this context:

Algorithm 7.4 (Sequential vertex-coloring)**.** *Let G be a loopless finite graph with vertices $\{x_1, x_2, \dots, x_N\}$, and let $C = \{1, 2, \dots\}$.*

(i) Fix $i := 1$,

(ii) Define $\omega(x_i)$ as the smallest element of C not used by any vertex x_j adjacent to x_i with $j < i$, and set $i := i + 1$,

(iii) Repeat (ii) until $i = N + 1$.

This algorithm always return a coloring of G but it is rarely a minimum coloring. The example of an application of this algorithm is provided in Figure 7.1: the first figure

indicates the initial ordering of the vertices, the second figure presents the result of the sequential vertex-coloring, while the last figure corresponds to a minimum coloring of the graph. However, note that there was some arbitrariness in indexing the vertices in a certain order $x_1, x_2, \dots$. By choosing a different initial ordering another coloring (probably with a different number of colors) would have been obtained. An interesting observation is that there always exists an initial ordering of the vertices which would lead with this algorithm to a minimum coloring. However, finding this very good initial ordering is not simpler than looking directly for a minimum coloring.

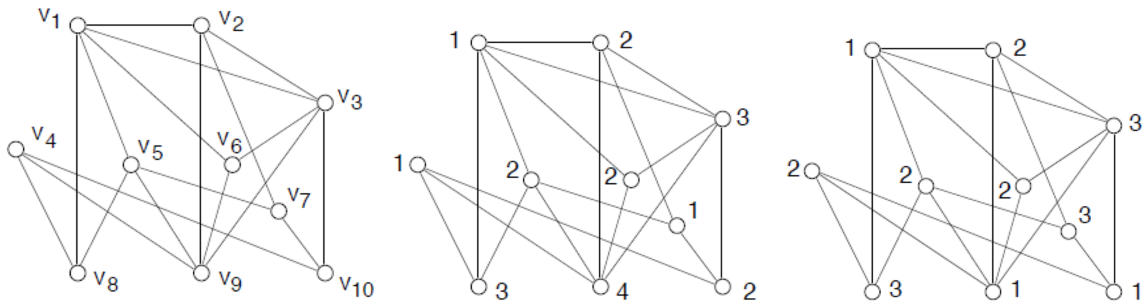


Fig. 7.1. Initial ordering, sequential vertex-coloring, minimum coloring, from [GYA, Sec. 8.1]

Let us now state and prove some easy results. Recall that the maximal degree $\Delta(G)$ of a graph has been introduced in Section 1.1, the clique number $w(G)$ has been introduced in Definition 2.16, and the independence number $\alpha(G)$ has been introduced in Definition 2.17.

Lemma 7.5. *For any loopless finite graph G, one has $\chi(G) \leq \Delta(G) + 1$.*

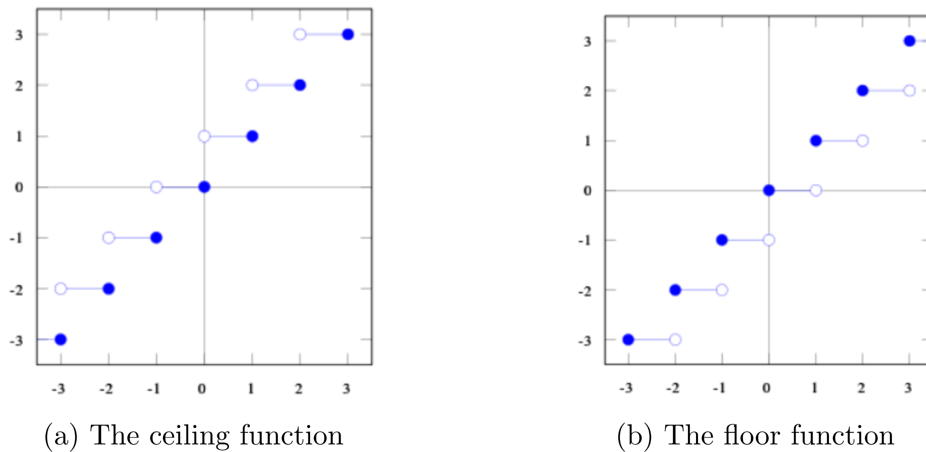
Proof. By using the sequential vertex-coloring algorithm, no more than $\Delta(G)+1$ colors will ever be used, no matter what is the initial ordering of the vertices. □

Lemma 7.6. *For any loopless finite graph, one has $\chi(G) \geq w(G)$.*

Proof. Since the elements of a clique are all mutually connected, it is necessary to use k colors for a clique containing k elements. The statement follows directly from this observation. □

For the next lemma, we introduce the *ceiling function*: For any $s \in \mathbb{R}$ we write $\lceil s \rceil$ for the least integer greater than or equal to s. This function is clearly related to the *floor function*: For any $s \in \mathbb{R}$ we write $\lfloor s \rfloor$ for the greatest integer less than or equal to s The graphs of these two functions are represented in Figure 7.2, and additional properties can be found in [23].

Lemma 7.7. *For any finite graph $G = (V, E)$ one has $\chi(G) \geq \left\lceil \frac{|V|}{\alpha(G)} \right\rceil$, where $|V|$ denotes the cardinality of V, namely the order of G.*



(a) The ceiling function (b) The floor function

Fig. 7.2. Two integer valued functions, from [23]

Proof. Since each color class contains at most $\alpha(G)$ vertices, the number of different color classes must be at least equal to $\left\lceil \frac{|V|}{\alpha(G)} \right\rceil$. □

Let us add one more easy observation: For any finite graph G and any subgraph H of G one has $\chi(G) \geq \chi(H)$. This is quite clear since any minimum coloring of G is also a minimum coloring of H. Note however that in general a minimum coloring of H can not be used as a starting point for a coloring of G. This observation can be used for guessing some lower bound for $\chi(G)$. Indeed, if one subgraph H of G is not k-colorable for some k, then the graph G itself won't be k-colorable. In such a case, we say that the subgraph H is a k-*obstruction*.

Let us provide now a result which sharpen the easy Lemma 7.5. The statement has been proved by Brooks in 1941, and several proofs are available over the internet, see also [GYA, Thm. 8.1.21].

Theorem 7.8 (Brooks' theorem)**.** *For any connected, undirected, loopless and finite graph G, one has $\chi(G) \leq \Delta(G)$, unless G is a complete graph K_n or a cycle graph C_n with n odd, in which case $\chi(G) = \Delta(G) + 1$.*

Remark 7.9. *For some applications, it might be useful to look for a k-coloring of a graph even if k is smaller than the chromatic number of the graph ! In such a case, one consider k-colorings by disregarding some edges for which the condition of not having the same color at their endpoint does not hold. Then, one look for the k-coloring which minimizes the number of edges which have to be excluded. Alternatively, one can consider edge weights and try to minimize the total weight of the edges which have to be disregarded. If the weight corresponds to the importance of an edge, it means that some edges of lower importance can be disregarded.*

We end this section with a heuristic algorithm. Based on our intuition, when coloring a graph a vertex with a large degree should be considered before a vertex with a small

degree. In addition, for two vertices with the same degree, the one having a denser subgraph generated by its neighbours should be treated first, see Definition 1.4 for the notion of neighbours. We shall say that a vertex x is *uncolored* if no value to $\omega(x)$ has been attributed so far. We also call the *colored degree* of a vertex x the number of different colors that have been assigned to vertices adjacent to x.

Algorithm 7.10 (Largest-degree-first algorithm)**.** *Let G be a loopless and finite graph, and let $C = \{1, 2, \dots\}$.*

(i) Set $i = \Delta(G)$,

(ii) Among all uncolored vertices of degree i, choose a vertex x with a maximum colored degree, and set $\omega(x) = k$ with k the smallest possible color,

(iii) Repeat (ii) as long as there exists some uncolored vertices of degree i,

(iv) Set $i := i - 1$ as long as $i \geq 1$, and go back to (ii).

An application of this algorithm is provided in Figure 7.3. In this case, this algorithm has a better outcome than the sequential vertex-coloring algorithm. However, this is not always the case, it all depends on the initial ordering for the sequential vertex-coloring algorithm. On the other hand, observe that no initial ordering is necessary for the largest-degree-first algorithm.

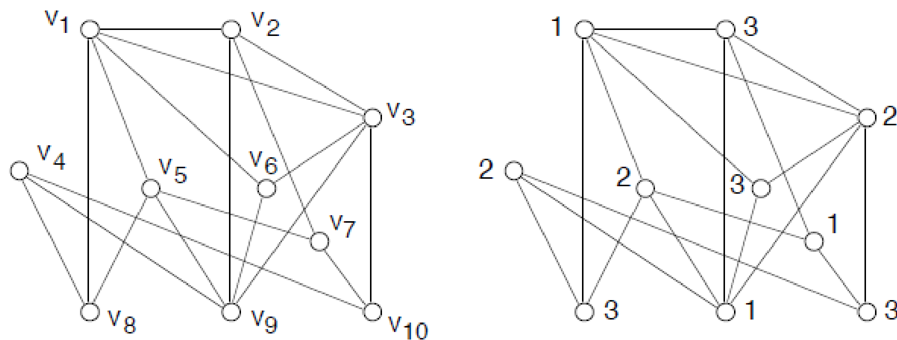


Fig. 7.3. An application of the largest-degree-first algorithm, see Figure 8.1.4 of [GYA]

Remark 7.11. *There exists also a notion of* edge-coloring *and the theory can be developed as above. For certain applications, this approach is even more natural, but these two theories are very close to each other of some dualities between graphs, see Definition 7.23 and its generalization. Some information about edge-coloring can be found in [Die, Sec. 5.3], in [CH, Sec. 6.5], or in [GYA, Sec. 8.3].*

7.2 Plane graphs

In the previous section, representations of a graph did not play any role. For other applications, the representation is as important as the graph itself, and the ambient space for the representation is also important. In this section we shall stick to the ambient space $\mathbb{R}^2$ (the usual plane). Note however that representations on other surfaces (like on a sphere or on a 2-torus) are also important.

Definition 7.12 (Plane graph). *A* plane graph *is a finite graph $G = (V, E)$ with the set of vertices V given by $\{x_1, x_2, \ldots, x_N\} \subset \mathbb{R}^2$, with the set of edges E given by a finite family of simple arcs (bijective and bicontinuous images of $[0, 1]$) having endpoints in V, and such that the interior of any arc contains no vertex and no point of any other edge.*

In simpler terms, a plane graph is sometimes defined by a planar drawing of a finite graph with no edge-crossing, but the above definition is certainly more precise. Note that this definition allows loops and multiple edges, which is sometimes not accepted in the definition of a plane graph (it depends on the authors). In this setting, we call *faces of the plane graph G* the open subsets defined by $\mathbb{R}^2 \setminus G$. For any plane graph G, there is always one face which is unbounded (called *the outer face*) and a finite number of bounded faces (called *the inner faces*). The set of faces of G is denoted by $F(G)$, see Figure 7.4 which contains 4 faces. For two distinct faces, we say that they are *adjacent* if they are separated by one (or more) edge. Equivalently, they are adjacent if their closure in $\mathbb{R}^2$ contain at least one common edge.

For plane graphs, there exists a quite famous formula linking the number of vertices, the number of edges, and the number of faces (including the unbounded one).

Theorem 7.13 (Euler's theorem). *Let $G = (V, E)$ be a connected plane graph, then the following equality holds:*

$$|V| - |E| + |F(G)| = 2, \tag{7.1}$$

where $|F(G)|$ denotes the number of faces of G.

There exist many proofs of this result, which can be stated in a more general framework. For simple plane graph we refer for example to [Die, Thm 4.2.9], or to [24] for twenty different proofs of this result. We shall now derive additional relations on plane graphs. We first give a definition related to the boundary of a face.

Definition 7.14 (Size of a face). *Let G be a plane graph, and let $f \in F(G)$ be one of its faces. The* size of *f is the number of edges of G on a boundary walk around f. We set* size(f) *for the size of the face f.*

If the plane graph is simple, the size of a face is rather easy to compute, and this number is always bigger than or equal to 3. For plane graphs with loops or multiple edge, one has to be more careful, and this number can be 1 or 2. For example, in Figure 7.4, the face f_1 is of size 1, the face f_2 is of size 2, the face f_3 is 3, and the face f_4 is of size 6. Based on this definition, a simple relation based on faces can be obtained:

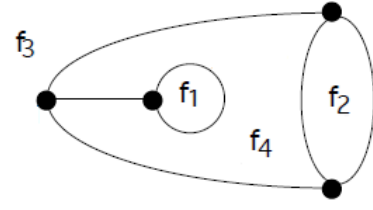


Fig. 7.4. Plane graph with 4 faces

Lemma 7.15 (Face-size relation)**.** *Let $G = (V, E)$ be a connected plane graph, and let $F(G)$ denote the set of its faces. Then the following relation holds*

$$2|E| = \sum_{f \in F(G)} \mathrm{size}(f). \tag{7.2}$$

Proof. Each edge either occurs once in each of two different face boundary walks or occurs twice in the same boundary walk. Thus, by definition of face-size, each edge contributes two sides to the sum. □

Recall now that the girth of a graph has been introduced in Definition 1.20. It corresponds to the length of the shortest cycle in a graph, as long as the graph is not a tree. Then, by a minute of thought, or by looking at [GYA, Prop. 7.5.5] one easily observes that for a plane graph which is not a tree the relation

$$\mathrm{girth}(G) \leq \min_{f \in F(G)} \mathrm{size}(f) \tag{7.3}$$

always holds. Based on this observation and on the previous lemma, one infers:

Lemma 7.16 (Face-edge relation)**.** *Let $G = (V, E)$ be a connected plane graph, and let $F(G)$ denote the set of its faces. Then the following relation holds:*

$$2|E| \geq \mathrm{girth}(G)\,|F(G)|. \tag{7.4}$$

Proof. The inequality is a direct consequence of the equality (7.2) together with the inequality (7.3). □

These various easy results lead to a rather important property of simple plane graphs. Recall that if G is simple, it has no loop and no multiple edge, and as a consequence its girth is always bigger than or equal to 3.

Theorem 7.17. *Let $G = (V, E)$ be a connected simple plane graph with $|V| \geq 3$. Then the following inequality holds*

$$|E| \leq 3|V| - 6. \tag{7.5}$$

Proof. From (7.4) with $\mathrm{girth}(G) \geq 3$ one infers that $2|E| \geq 3|F(G)|$, or equivalently $\frac{2}{3}|E| \geq |F(G)|$. By inserting this inequality in (7.1) one infers that $|V| - |E| + \frac{2}{3}|E| \geq 2$, which is equivalent to $|V| - \frac{1}{3}|E| \geq 2$. The statement follows easily from this inequality. □

Let us briefly mention some direct consequence of this result, proofs can be done as an exercise. For example, it follows from the previous theorem that the complete graph K_5 can not be represented as a plane graph. Also, any simple graph $G = (V, E)$ with $|V| = 8$ and $|E| \geq 19$ can not be represented as a plane graph. More generally we say that a graph which can not be represented as a plane graph that it is not a *planar graph*. In the next statement, we strengthen the previous result in the special case of a bipartite graph.

Theorem 7.18. *Let $G = (V, E)$ be a connected, simple, and bipartite plane graph with $|V| \geq 3$. Then the following inequality holds*

$$|E| \leq 2|V| - 4.$$

Proof. The proof is quite similar to the previous one, but this time the girth of a simple bipartite graph is at least 4 (it can not be 3 by the bipartiteness property). Thus one gets from (7.4) that $2|E| \geq 4|F(G)|$, or equivalently $\frac{1}{2}|E| \geq |F(G)|$. By inserting this inequality in (7.1) one infers that $|V| - |E| + \frac{1}{2}|E| \geq 2$, which is equivalent to $|V| - \frac{1}{2}|E| \geq 2$. The statement follows easily from this inequality. □

As for K_5 before, the previous result has an important consequence on the so-called graph $K_{3,3}$, see Figure 7.5. This graph is connected, simple and bipartite, with 6 vertices and 9 edges. As a consequence of the previous Theorem, this graph can not be represented as a plane graph, and therefore is not a planar graph. The two graphs K_5 and $K_{3,3}$ are sometimes referred to as the Kuratowski graphs. This name comes from the important result presented below. Before it, one needs to introduce two more concepts:

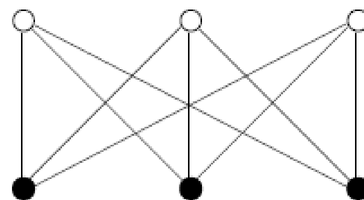


Fig. 7.5. The $K_{3,3}$ graph

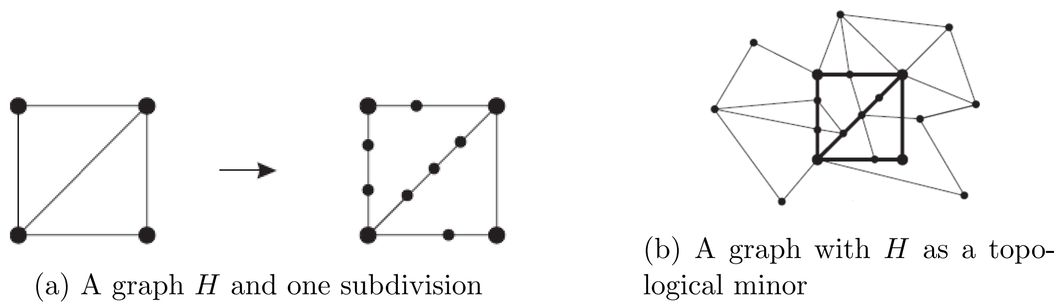
Definition 7.19 (Subdivision). *A* subdivision *of a graph G is a new graph obtained by subdividing some of the edges of G by adding new vertices on these edges. Any subdivision of G is denoted by $\mathcal{T}G$.*

A graph and one of its subdivision are presented in Figure 7.6a.

Definition 7.20 (Topological minor). *A graph H is a topological minor of a graph G if G contains $\mathcal{T}H$ as a subgraph.*

In Figure 7.6b the previous figure appears as a topological minor of the graph. We can now state a characterization of planar graphs. More precisely, the following theorem provides necessary and sufficient conditions for a graph to have a planar representation.

Theorem 7.21 (Kuratowski's theorem). *A graph G admits a plane graph representation (*i.e. is a planar graph*) if and only if G does not contain the graphs K_5 or $K_{3,3}$ as a topological minor.*



(a) A graph H and one subdivision

(b) A graph with H as a topological minor

Fig. 7.6. A subdivision and a topological minor, from Sec. 1.7 of [Die]

Note that necessity of the absence of these two graphs is quite simple, but the difficult part of the proof is the sufficiency. We refer to [Die, Sec. 4.4] for a proof in the case of simple graphs, or to [GYA, Sec. 7.4] for a proof without the assumption of simplicity. Note however that multiple edges or loops do not play a role here.

Having now a criterion for the planarity of graphs, we can state one of the best known results for plane graphs. As shown in the following section, this results implies that any map can be coloured with four colors.

Theorem 7.22 (Four colors theorem)**.** *Every loopless plane graph is* 4*-colorable.*

This theorem is one milestone in graph theory, and a lot of information on it are available, as for example in [25] or in every book on graph theory. Note that its proof has been one of the most challenging problem in computer's assisted mathematics. However, there exists a weaker statement which is perfectly accessible: Every loopless plane graph is 5-colorable. Its proof is provided in Section 7.4.1.

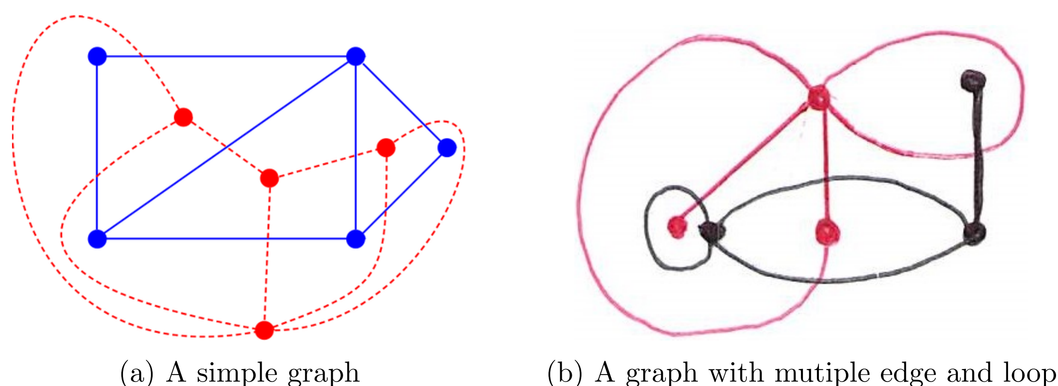
7.3 Map-colorings

Let us start by introducing the notion of plane duality. The idea is the following: Starting from a plane graph $G = (V, E)$, we construct a new plane graph G^* by first placing a new vertex in each face of G. This defines a set V^*. Edges are then added with the following rule: for any $e \in E$ we link the two vertices of V^* separated by e by an edge e^* crossing e; if e is incident with only one face, we attach a loop e^* to the vertex corresponding to that face, again crossing the edge e. The set of such e^* defines E^*, and the dual graph is $G^* = (V^*, E^*)$. Before stating a more precise definition, two examples are provided in Figures 7.7.

Definition 7.23 (Dual graph)**.** *Let $G = (V, E)$ and $G^* = (V^*, E^*)$ be two connected and plane graphs, with corresponding set of faces F and F^*. The graph G^* is* dual of G *if there exist bijections*

$$F \ni f \mapsto x^*(f) \in V^*, \qquad E \ni e \mapsto e^* \in E^*, \qquad V \ni x \mapsto f^*(x) \in F^*$$

satisfying the following conditions



(a) A simple graph (b) A graph with mutiple edge and loop

Fig. 7.7. Dual graphs, in red

(i) $x^*(f) \in f$ *for any* $f \in F$,

(ii) e *intersects* G^* *only on one point,* e^* *intersects* G *only on one point, and these intersections correspond to an intersection between the interior of* e *and the interior of* e^*,

(iii) $x \in f^*(x)$ *for any* $x \in V$.

It is rather clear from this definition that any connected and plane graph admits a dual, and in fact the initial graph is the dual of its dual graph. In other words, the map $G \to G^*$ is an *involution*. Note also that this notion of a dual graph can be abstracted to more general graph (they don't have to be plane graphs anymore). In this extended framework, a graph is a planar graph if and only if its dual is a planar graph. We shall not develop this theory here, but refer to [Die, Sec. 4.6] for more information. We also mention one additional property which can be proved as an exercise. In fact, [CH, Sec. 5.6] contains several nice properties of the dual graph which can be obtained rather easily.

Exercise 7.24. *An edge of* G *is a loop if and only if the associated edge* e^* *is a bridge in* G^*.

Let us now define a special instance of a connected plane graph. The name is surprisingly natural, as we can easily observe.

Definition 7.25 (Map)**.** *A* map *is a connected plane graph with no bridge.*

As a consequence of Exercise 7.24, the dual graph of a map is a plane graph with no loop. In fact, the absence of loop in the dual graph is one of the interest of the definition of a map. The following definition is a reminiscence of the a vertex k-coloring.

Definition 7.26 (Map-coloring)**.** *Let* G *be a plane graph without bridge (*i.e. *a map), and let* C *be a set containing* k *elements. A* map k-coloring *of* G *is a function* $F(G) \to C$ *with the requirement that any two adjacent faces are colored differently.*

Let us observe that the absence of bridge is a necessary requirement for the existence of a map k-coloring. Indeed, the graphs presented in Figure 7.8 contain bridges and do not accept any map-coloring.

Fig. 7.8. Graphs with a bridge

We can finally make the link between the notion of map-coloring and the four colors theorem stated in Theorem 7.22. Indeed, since any map G has a dual graph which is loopless, the four colors theorem applies to its dual graph $G*$. Then, since any vertex x^* of G^* belong to a unique face of G, one can color this face with the color of x^*. The condition that any edge in G^* has two endpoints of two different colors implies that two adjacent faces in G have also two different colors. By calling a region what has been named *a face*, one has thus proved:

Theorem 7.27. *No more than four colors are required to color the regions of any map so that no two adjacent regions have the same color.*

An illustration of this result is presented in Figure 7.9.

7.4 Appendix

7.4.1 The five color theorem

The material of this section has been studied and provided by Tomoya Tatsuno. It is based on references [Die, Sec. 5.1] and [27].

There is a very famous theorem in graph theory called the four color theorem, which states that every loopless plane graph is 4-colorable. As a consequence of this theorem, every map can be colored with at most four colors so that no two adjacent regions have the same color. Although the four color theorem is known to be very difficult to prove, there is a weaker version of this theorem that can be proven much more easily:

Theorem 7.28 (Five Color Theorem)**.** *Every loopless plane graph is* 5*-colorable.*

We shall first state an important lemma that we use in the proof of the five color theorem, which has already been proven in the lecture. First of all, the meaning of every terminology and notation used in this article is the same as in the lecture. However, let us just recall the definition of plane graphs for clarity.

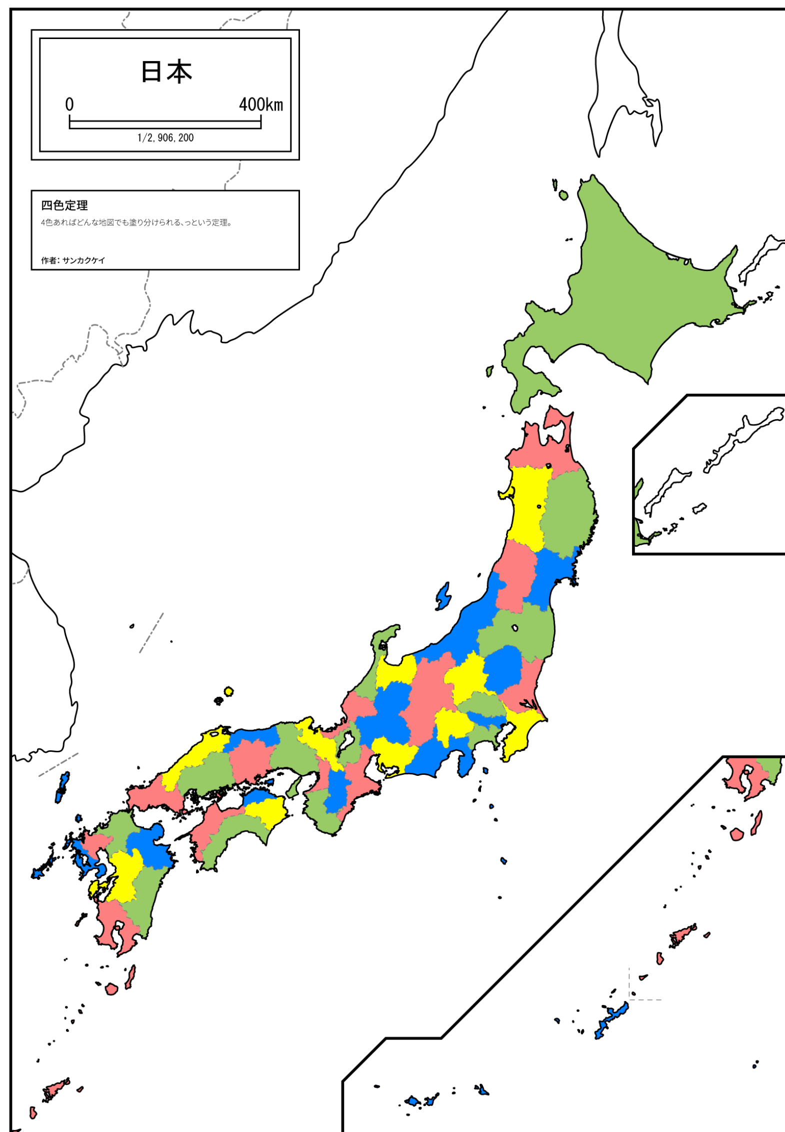


Fig. 7.9. Illustration of the four colors theorem, from [26]

Definition 7.29. *A **plane graph** is a finite graph $G = (V, E)$ with the set of vertices V given by $\{x_1, x_2, \ldots, x_N\} \in \mathbb{R}^2$, with the set of edges E given by a finite family of simple arcs (bijective and bicontinuous images of $[0, 1]$) having endpoints in V, and such that the interior of any arc contains no vertex and no point of any other edge.*

Note that this definition does not assume that plane graphs are simple. As written in the lecture notes, it depends on each author if plane graphs are assumed to be simple or not. However, it should be noted that when we discuss the coloring of loopless plane graphs, multiple edges do not play a role at all, and next lemma for a connected simple graph plays an important role in the proof of Theorem 7.28:

Lemma 7.30. *Let $G = (V, E)$ be a connected simple plane graph with $|V| \geq 3$. Then the following inequality holds:*

$$|E| \leq 3|V| - 6 \tag{7.6}$$

Observe that the assumption that G is simple cannot be eliminated. Indeed, if G is allowed not to be simple, then the inequality does not hold because we can choose two vertices and increase the number of edges between them arbitrarily without changing the number of vertices.

Let us now prove the main result of this section.

Proof of Theorem 7.28. Let $G = (V, E)$ be any loopless plane graph. First, we may assume that G is simple since multiple edges do not play a role at all in coloring. Since G is simple, we can use the notation $(x, y) \in V \times V$ for an edge e such that $i(e) = (x, y)$. Also, since G is a union of its connected components, it suffices to show that G is 5-colorable when G is connected. Hence we may assume that G is connected and simple. We give a proof by induction on $|V|$. If $|V| \leq 5$, then by assigning different colors to each vertex, G is 5-colorable. Suppose that the statement holds for $|V| = n$ for some integer $n \geq 5$. Let us show that the statement holds for $|V| = n + 1$. Let $|V| = n + 1$.
Claim 1 There exists $v \in V$ such that $\deg(v) \leq 5$.
Proof for Claim 1
Suppose for any $v \in V$, $\deg(v) \geq 6$. Since each vertex has at least 6 edges starting from it and one edge is shared by exactly two vertices, one has

$$6|V| \leq 2|E| \tag{7.7}$$

On the other hand, since G is a connected simple plane graph with $|V| \geq 3$, it follows from Lemma 7.30 that

$$|E| \leq 3|V| - 6 \tag{7.8}$$

Combining the equations (7.7) and (7.8), one has

$$6|V| \leq 6|V| - 12,$$

which is a contradiction. This proves Claim 1.

Then let v be a vertex of degree 5 or less, and let $H = G - \{v\}$. By the induction hypothesis, H is 5-colorable. Hence there exists a vertex 5-coloring $\omega : V \setminus \{v\} \to \{1, \ldots, 5\}$. If ω uses at most 4 colors for the neighbors of v, then we can color v by the color that is not used. Thus we let $\deg(v) = 5$ and assume that the neighbors of v have distinct colors for the rest of the proof.

Let D be an open small disk such that it meets only five edges starting from v and does not contain any other edges or vertices other than v. Let us label the intersection of those five edges with D according to their cyclic position in D as $s_1, \ldots, s_5$, and let (v, v_i) be the edge containing s_i. Without loss of generality, we may assume that $\omega(v_i) = i$ for each i. The purpose of taking D is to examine the behavior of our graph near v, see Figure 7.10.

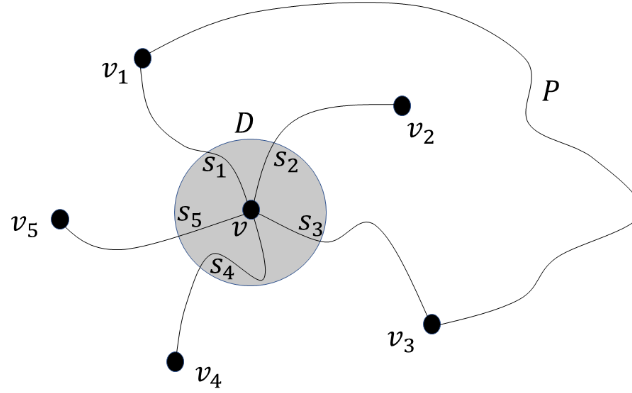


Fig. 7.10. The vertex with degree 5

Let P be any $\{v_1\} - \{v_3\}$ path in $H - \{v_2, v_4\}$. We define $H - P$ as follows: if V_P denote the set of all vertices contained in P, then $H - P := H - V_P$.

Claim 2 If there exists a $\{v_1\} - \{v_3\}$ path P in $H - \{v_2, v_4\}$, then there does not exist a $\{v_2\} - \{v_4\}$ path in $H - P$.

Proof for Claim 2

Let C be the cycle vv_1Pv_3v. Note that since G is simple, the cycle C is uniquely determined. It suffices to show that there does not exist a $\{v_2\} - \{v_4\}$ path in $G - C$. The notation $G - C$ is defined to be $G - V_C$, where V_C is the set of all vertices contained in C. Indeed, if there exists a $\{v_2\} - \{v_4\}$ path Q in $H - P$, then Q is a $\{v_2\} - \{v_4\}$ path in $G - C$. By taking a contraposition, if there does not exist a $\{v_2\} - \{v_4\}$ path in $G - C$, then there does not exist a $\{v_2\} - \{v_4\}$ path in $H - P$. Let $x_2 \in s_2$ and $x_4 \in s_4$. Since C is a cycle, which implies in particular that C has no self-intersection as a closed path in $\mathbb{R}^2$ by the property of plane graphs, $\mathbb{R}^2 \setminus C$ has exactly two components: a bounded open set A and an unbounded open set B. Note that strictly speaking, we are using Jordan curve theorem here. One has $x_2 \in A$ and $x_4 \in B$. Suppose there exists a $\{v_2\} - \{v_4\}$ path in $G - C$. By the property of plane graphs and $\{v_2, v_4\} \subseteq \mathbb{R}^2 \setminus C$, (v, v_2) and (v, v_4) contribute paths (in the topological sense) in $\mathbb{R}^2 \setminus C$ between x_2 and v_2 and between x_4 and v_4, respectively. Thus there exists a path in $\mathbb{R}^2 \setminus C$ between x_2 and x_4. It contradicts $x_2 \in A$ and $x_4 \in B$. This proves Claim 2.

Given $i, j \in \{1, \ldots, 5\}$, let $H_{i,j}$ be the subgraph of H induced by the vertices colored i or j.

Claim 3 We may assume that $H_{1,3}$ contains a $\{v_1\} - \{v_3\}$ path P in $H - \{v_2, v_4\}$.

Proof for Claim 3

Observe first that $H_{1,3}$ might consist in several components, and we call C_1 the component of $H_{1,3}$ containing v_1. If the component C_1 also contains v_3, then Claim 3 holds. Suppose that the component C_1 of $H_{1,3}$ containing v_1 does not contain v_3. If we interchange the colors 1 and 3 at all the vertices of C_1, we obtain another 5-coloring of H.

Then v_1 and v_3 are both colored 3 in this new coloring, and we may assign color 1 to v. Thus it suffices to deal with the case where the component C_1 of $H_{1,3}$ containing v_1 also contains v_3. This proves Claim 3.

Then the component C_2 of $H_{2,4}$ containing v_2 does not contain v_4. Indeed, if C_2 contains v_4, then C_2 is a $\{v_2\} - \{v_4\}$ path in $H - P$ since P is contained in $H_{1,3}$. This contradicts Claim 2. It means v_2 and v_4 lie in different components of $H_{2,4}$. If we interchange the colors 2 and 4 in C_2, we obtain a new 5-coloring of H. Since v_4 is not contained in C_2, v_2 and v_4 are colored 4 in this new coloring. Now v no longer has a neighbor colored 2. Thus we can assign color 2 to v. This completes the proof for Theorem 7.28. □

Note that Claim 1, proven by Lemma 7.30, was one of the most crucial steps. By Claim 1, one can find a vertex v that has a degree of 5 and one may assume neighbors of v have distinct colors since one has 5 colors. If we could show that for any loopless plane graph there exists a vertex v that has a degree of 4 or less, then by applying the proof for Theorem 7.28, the four color theorem could be proven. However, it is known that there exists a loopless plane graph such that every vertex has a degree of 5 or more, called an icosahedral graph. Thus the proof for Theorem 7.28 cannot be used to prove the four color theorem.

7.4.2 Some problems related to plane graphs

The content of this section has been studied and written by Eda Ruyshin, Kondo Ayaka and Bui Tu Ha.

Exercise 7.31. *Let $G = (V, E)$ be a connected simple plane graph with $|V| < 12$. Prove that G has a vertex of degree at most 4.*

Proof. If $|V| \leq 5$ then the maximal degree $\Delta(G)$ is at most 4. Thus, every vertex in G has degree at most 4. In the case $5 < |V| < 12$, since each edge adds two degree to the total degree of vertices and also based on Theorem 7.17, one has:

$$\sum_{x \in V} \deg(x) = 2\,|E| \leq 2(3|V| - 6) < 6|V| - |V| = 5|V|. \tag{7.9}$$

If every vertex has degree more than 4 (or equivalently, at least 5), then $\sum_{x \in V} \deg(x) \geq 5|V(G)|$, which contradicts (7.9). Hence, the graph G has a vertex of degree at most 4. □

Definition 7.32 (Complete bipartite graphs)**.** *The graph $G = (V, E)$ is a* complete bipartite graph *if it is a simple bipartite graph such that every vertex in one of the bipartition subsets is connected to every vertex in the other bipartition subset. If one subset of the bipartition contains r vertices, and the other subset s vertices, the corresponding graph is denoted by $K_{r,s}$.*

Let us now introduce some examples of complete bipartite graphs.

(1) Star graphs: For any positive integer s, the graph $K_{1,s}$ is called a *star graph*. All complete bipartite graphs which are trees are stars, see Figure 7.11. In addition, the graph $K_{1,3}$ is also called a *claw*.

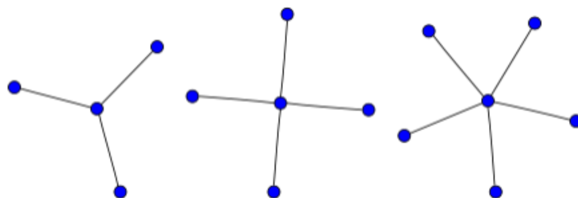


Fig. 7.11. The star graphs $K_{1,3}$,$K_{1,4}$ and $K_{1,5}$

(2) $K_{n,n}$ graphs when n is a positive integer: The graph $K_{3,3}$ is also called *utility graph*. As mentioned already, any plane graph cannot contain $K_{3,3}$ as a subgraph. Some examples are presented in Figure 7.12.

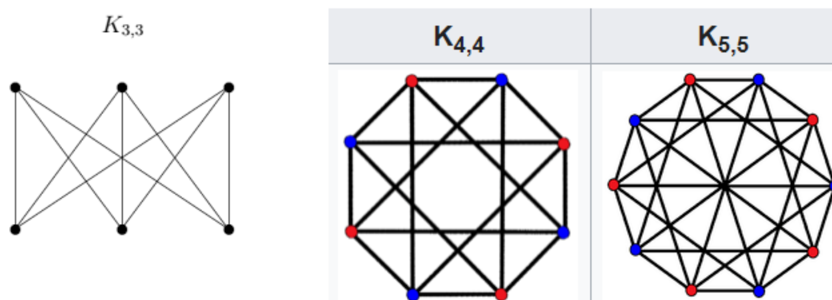


Fig. 7.12. The complete bipartite graphs $K_{3,3}$, $K_{4,4}$ and $K_{5,5}$

In Theorem 7.21 it is stated that $K_{3,3}$ can be used as a test for planarity. The next exercise uses this idea for an upper bound on the sum of the degrees of three vertices.

Exercise 7.33. *Prove that for any three vertices x, y, z of a simple plane graph $G = (V, E)$ with number of vertices at least 3, the sum of the degrees $\deg(x)+\deg(y)+\deg(z)$ is at most $2|V| + 2$.*

Proof. The sum $\deg(x)+\deg(y)+\deg(z)$ comprises in the connection among the vertices x, y, z themselves, and the connection between x, y, z and other vertices in G. If x, y, z are pairwise connected, the three pairwise edges among them make the sum $\deg(x) + \deg(y) + \deg(z)$ increase by 6. Since G cannot have $K_{3,3}$ as a subgraph, at most two vertices in $V \setminus \{x, y, z\}$ can be connected to all the three vertices x, y, z. With the exception of possibly two vertices, all remaining $|V| - 5$ vertices are adjacent to at most 2 vertices among x, y, z. Then, one infers

$$\deg(x) + \deg(y) + \deg(z) \leq 6 + 3 \times 2 + 2(|V| - 5) = 2|V| + 2.$$

□

Among all $K_{r,s}$ graphs, it is natural to wonder which ones are planar ?

Exercise 7.34. *The only $K_{r,s}$ graphs which are planar are for $(r,s) = (1,s)$ with $s \in \mathbb{Z}_+$ and for $(r,s) = (2,s)$ with $s \in \mathbb{Z}_+$.*

Proof. If $r > 3$ and $s > 3$ then $K_{r,s}$ contains $K_{3,3}$ as its subgraph and thus it is not a planar graph. Meanwhile, $K_{1,s}$ and $K_{2,s}$ can be plane graphs for any positive integer s. The graphs $K_{1,s}$ (also known as star graphs) can be drawn with one vertex in the center, surrounded by s vertices; the graphs $K_{2,s}$ can be drawn with s vertices on a line in the plane and the other two vertices, one on each side of this line, see Figure 7.13. □

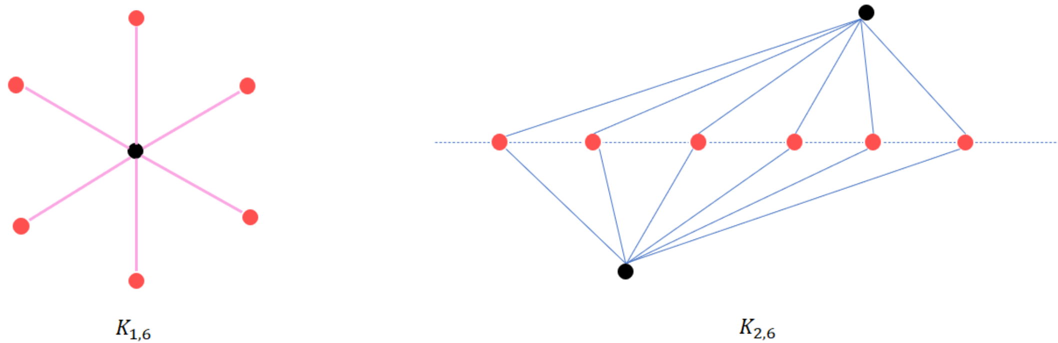


Fig. 7.13. An example of $K_{1,s}$ and $K_{2,s}$ with $s = 6$

Chapter 8

Directed graphs

In the previous chapters, we tried to present the theory simultaneously for directed graphs and for undirected graphs. However, some notions have been more naturally developed for undirected graphs, like the notion of connectivity or the chapter on graph colorings. In this chapter, we focus on directed graphs, study a few applications and develop some tools specifically for them.

8.1 Strongly connected components

Let us start by recalling that the notion of a connected graph was introduced in Definition 1.17, and that this notion does not see the orientation on edges of a directed graph. For such graphs, the notion of connectivity is sometimes called a *weak connectivity*, in contrast to the *strong connectivity* introduced in Definition 1.18. Recall that an oriented graph is strongly connected if there exists a path from x to y, for arbitrary vertices x and y. With our convention, it goes without saying that all paths on an oriented graph are oriented paths. However, the strong connectivity of the entire graph is often a requirement which is too strong. In that context, the following definition is useful.

Definition 8.1 (Strongly connected component)**.** *A* strongly connected component *of an oriented graph G is a maximal strongly connected subgraph of G. The vertices of a strong component are said to be* mutually reachable*.*

In other terms, for any vertices x and y in a strong component of a graph G, there exist a least one path from x to y and one path from y to x. On the other hand, for any x in a strong component and any y not in this strong component, either there does not exist a path from x to y, or there does not exist a path from y to x, or both do not exist. A graph with its strongly connected components is provided in Figure 8.1. As emphasized in this figure, the set of strongly connected components realizes a partition of the vertices of G. On the other hand, some edges do not belong to any strongly connected components. Based on these observations, one reduction of the initial graph is quite natural and useful.

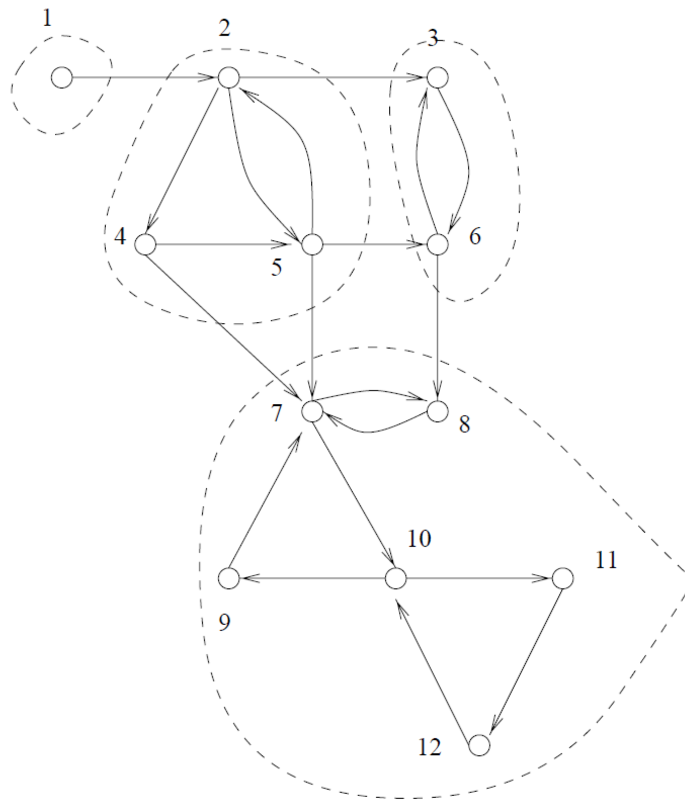


Fig. 8.1. A directed graph and its strongly connected components

Definition 8.2 (Condensation)**.** *Let G be a finite directed graph, and let $S := \{s_1, s_2, \dots, s_n\}$ be an enumeration of its strongly connected components. The* condensation *of G consists in the simple graph with vertices S and with edges defined by the following rule: for $j \neq k$ there exists an edge from s_j to s_k if one vertex of the strongly connected component s_j in G is linked to one vertex of the strongly connected component s_k in G.*

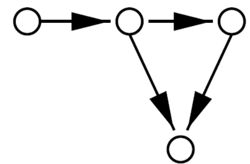


Fig. 8.2. A dag

The condensation of the graph of Figure 8.1 is provided in Figure 8.2. Note that a condensation is always an acyclic directed graph. Indeed, any cycle in a condensation would mean that some strongly connected components in G would not be maximal. A directed acyclic graph is often called *a dag*, see also Figure 3.1.

Let us now look at an algorithm for identifying the strongly connected components of a large graph. Note that there exist several algorithms for this task, as explained in [28]. Our approach will be based on the depth-first search (DFS) introduced in Section 4.2. The construction of a tree was provided in Algorithm 4.4, and it is the specific choice of a vertex in $\text{Front}(G, T_i)$ which characterizes the type of algorithm (dfs, bfs, or others).

We start recalling a few concepts for the growth of a tree. The discovery number function has been introduced in Remark 4.5. This function provides an index to the vertices according to their discovery during the algorithm. We also recall that the notions of skip-edges or cross-edges have been introduced at the end of Section 4.1. Figure 8.3 illustrates these concepts: the edges 1 and 3 are skip-edges, the former one being a back-edge while the latter one is a forward-edge. The edge 2 is a cross-edge since it links two vertices which belong to the tree but none is an ancestor of the other one. If the tree is constructed with Algorithm 4.4 following the dfs rule, then the following result can easily be obtained.

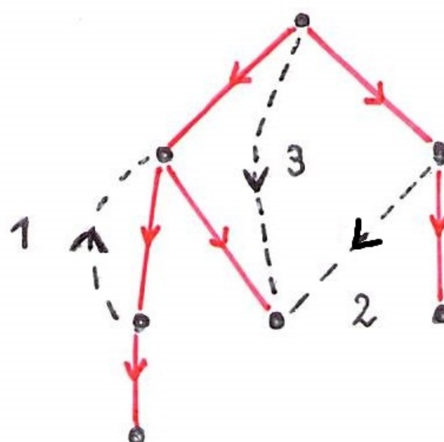


Fig. 8.3. A directed tree

Lemma 8.3. *Let e be a cross-edge of a depth-first search tree performed on a directed graph. If the origin of e is x and the terminal vertex of e is y, with x and y belonging to the tree, then one has* $\mathrm{dfnumber}(x) > \mathrm{dfnumber}(y)$.

The following proof has been studied and written by Bui Tu Ha, Zhang Liyang, Arata Suzuki, Tomoya Tatsuno, and Eda Ruyshin.

Proof. We are going to prove by contradiction. Assume that there exists a cross-edge $e = (x, y)$ such that $\mathrm{dfnumber}(x) < \mathrm{dfnumber}(y)$. After conducting DFS, we obtained the dfs-tree, denoted by T. The cross-edge $e = (x, y)$ implies that x and y are not in the same "family", in other words, neither of them is the ancestor of the other one. This means that there exists a subtree T_x of T which contains x but not y and a subtree T_y of T which contains y but not x , together with a vertex a in T being the root of the minimal subtree of T containing both T_x and T_y, see Figure 8.4. Among all the roots corresponding to subtrees of T that contains both T_x and T_y, the vertex a has the largest depth.

Based on the rules of DFS, since $\mathrm{dfnumber}(x) < \mathrm{dfnumber}(y)$, one has to finish discovering all vertices in T_x, then backtrack to a, before proceeding to T_y. However, the existence of the edge e, which points from x to y, means that x is not finished yet, which contradicts to the operation of DFS. Hence, the assumption is not correct and thus $\mathrm{dfnumber}(x) > \mathrm{dfnumber}(y)$. □

We shall now present an algorithm for identifying the strongly conected components of a graph, following the approach of [GYA, Sec. 9.5]. A more complete (but longer) presentation is also available in [BG, Sec. 7.5]. For the subsequent algorithm, additional functions have to be introduced. The setting is the construction of a dfs-tree for a digraph $G = (V, E)$.

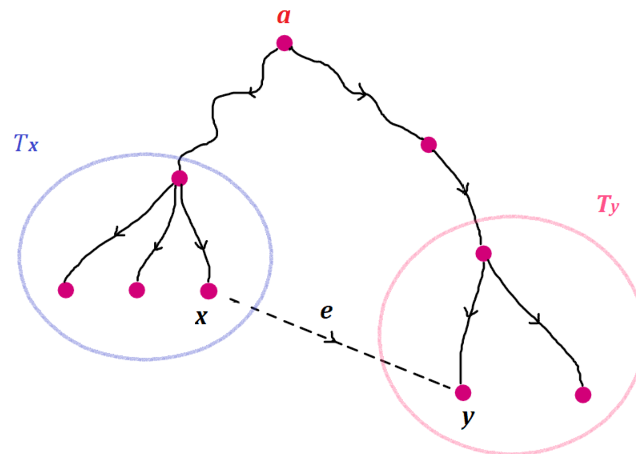


Fig. 8.4. Subtree of T with its root a and both T_x and T_y

(i) For any $x \in V$, the value low(x) corresponds to the smallest discovery number of all vertices which are known to be in the same strongly connected component as x,

(ii) For any $x \in V$, the binary function placed(x) takes the value TRUE if the vertex x has been placed in a strongly connected component, while it takes the value FALSE otherwise,

(iii) If x belongs to a tree (and is not the root), parent(x) refers to the parent of x in the tree,

(iv) For any $e \in E$, the binary function examined(e) takes the value TRUE if the edge e has already been examined (treated), while it takes the value FALSE otherwise,

(v) For any $e \in E$, the value head(e) denotes the target of e, which had been denoted by t(e) in Section 1.1.

Let us also recall that the structure of a stack has been introduced in Section 3.4.4 and illustrated in Figure 3.20b. In the following algorithm, we denote by *holdstack* a stack which keeps the vertices that have been processed but which are not placed yet in any strong component. Another stack will be denoted by *vertexstack* and will take care of vertices. Before running the algorithm, the initialization of its values is necessary. This process is described in Figure 8.5. Note that the strongly connected components are simply called *strong components*, that vertices are denoted by v, w, and that edges are also called arcs. For coherence, we shall follow these conventions in the rest of this section.

Before providing the algorithm, let us still describe how the function low is going to be computed. For that purpose, we shall say that a vertex v is *completely processed* if all arcs connected to v have been examined. On the other hand, we shall say that v is

For each vertex $w \in V$
  $dfnumber(w) := -1$
  $low(w) := -1$
  $placed(w) :=$ FALSE
For each arc $e \in E$
  $examined(e) :=$ FALSE
Initialize $vertexstack$ to contain vertex v
Initialize tree T as vertex v.
$dfnumber(v) := 0$
Initialize $dfnumber$ counter $dfnum := 1$
Initialize strong component counter $k := 0$
Initialize $holdstack$ as empty.

Fig. 8.5. Initialization of the algorithm, from Algorithm 9.5.2 of [GYA]

active if an arc having v as its origin is currently examined. The values of the function low will be updated as the algorithm proceeds. Once a vertex is completely processed, one has $\mathrm{low}(v) = \mathrm{dfnumber}(v)$ if and only if the vertex v is at the root of a subtree whose vertices form a strongly connected component. During the process, the value of $\mathrm{low}(v)$ will be updated according to the following rule:

1) If v has been completely processed and if $\mathrm{low}(v) < \mathrm{dfnumber}(v)$, then one sets

$$\mathrm{low}\big(\mathrm{parent}(v)\big) := \min\big\{\mathrm{low}\big(\mathrm{parent}(v)\big), \mathrm{low}(v)\big\}.$$

2) If v is active and if the arc being examined is a back-arc from v to an ancestor w of v, then one sets

$$\mathrm{low}(v) := \min\big\{\mathrm{low}(v), \mathrm{dfnumber}(w)\big\}.$$

3) If v is active and if the arc being examined is a cross-arc from v to w, then one sets

$$\mathrm{low}(v) := \min\big\{\mathrm{low}(v), \mathrm{dfnumber}(w)\big\}$$

if and only if w has not already been assigned to a strongly connected component, namely if and only if $\mathrm{placed}(w) = FALSE$.

An algorithm for exhibiting the strongly connected components of a directed graph is provided in Figure 8.6. Observe that this algorithm does not always produce a tree containing all vertices of the initial graph. This was already observed when the algorithm about the growth of a tree was proposed, without looking at the strongly connected components. However, once the above algorithm has stopped, one can initiate it again on $G - V_T$, with V_T the set of vertices of the tree. By performing again this algorithm, new strongly connected components of the remaining part of G can be found.

Input: a digraph D and a starting vertex v.
Output: a dfs-tree T with root v;
vertex-sets $S_1, S_2, \ldots, S_k$ of the strong components of tree T

```
Initialize variables as prescribed above.
[Begin processing]
While vertexstack is not empty
    Let vertex t be top(vertexstack).
    While there are unexamined arcs incident from vertex t
        Let e be an unexamined arc incident from vertex t.
        examined(e) := TRUE
        Let vertex w be head(e).
        If dfnumber(w) = −1 [vertex w is not yet in tree T]
            dfnumber(w) := dfnum
            dfnum := dfnum + 1
            low(w) := dfnumber(w)
            Push vertex w onto vertexstack.
            t := w
        Else If placed(w) = FALSE
            [vertex w has not yet been placed in a strong component]
            low(t) := min{low(t), dfnumber(w)}
    [processing of vertex t has been completed]
    If low(t) = dfnumber(t)
        [vertex t is the root of a subtree of T whose vertices form a
          strong component]
        k := k + 1
        placed(t) := TRUE
        Initialize set S_k := {t}. [place vertex t in strong component S_k]
        While holdstack ≠ ∅ AND low(top(holdstack)) ≥ dfnumber(t)
            z := top(holdstack)
            placed(z) := TRUE [z and t are mutually reachable]
                S_k ::= S_k ∪ {z}
                Pop holdstack.
        Pop vertex t from vertexstack.
    Else
        Push vertex t onto holdstack
        Pop vertexstack.
        If parent(t) exists (i.e., t is not the root of T)
            low(parent(t)) := min{low(parent(t)), low(t)}
    [depth-first search backs up to parent(t)]
Return tree T vertex-sets S_1, S_2, ..., S_k.
```

Fig. 8.6. Strongly connected component's algorithm, from Algorithm 9.5.2 of [GYA]

8.2 Tournaments

In this section and in the following one, we discuss some applications of directed graphs, and introduce a few more concepts related to them. Recall that a *round-robin tournament* or *all-play-all tournament* is a competition in which each contestant meets all other contestants in turn. The mathematical counterpart is provided by the following definition.

Definition 8.4. *A* tournament *is a simple directed graph whose underlying graph is complete.*

One can immediately observe that any tournament is obtained by adding an orientation on all edges of a complete graph K_n with n vertices. In applications, whenever there is an edge from x to y one says that x *dominates* (or *beats*) y. It is easy to remember this since $x \to y$ can also be seen as $x > y$. According to this convention, the outdegree of a vertex x corresponds to the *score* of the vertex x. Note that since the complete graph K_n has $n(n-1)/2$ edges and if we attribute 1 point for each contest between two vertices, then there exists a total number of $n(n-1)/2$ points to be distributed inside the graph.

A special example of a tournament corresponds to a *transitive tournament.* Before considering them, we introduce the notion of transitivity for arbitrary directed graphs.

Definition 8.5 (Transitive digraph)**.** *A directed graph is* transitive *if whenever there exists an edge from x to y and an edge from y to z, then there exists an edge from x to z.*

For tournament, transitivity can be expressed by several equivalent properties. We list some of them in the following statement.

Lemma 8.6. *Consider a tournament with n vertices. Then the following statements are equivalent:*

(i) G is transitive,

(ii) G is acyclic,

(iii) G has exactly one Hamiltonian path,

(iv) G admits a strict total ordering,

(v) The sequence of scores attributed to the vertices is $(0, 1, 2, \ldots, n-1)$,

(vi) G does not contain a cycle of length 3.

Transitive tournaments are convenient because they naturally offer a unique ranking. What about tournaments which are not transitive ? How can one choose a Hamiltonian path which would correspond to a ranking ? The next result shows that there always exists a least one Hamiltonian path in any tournament. The proof (by contradiction) is left as an exercise.

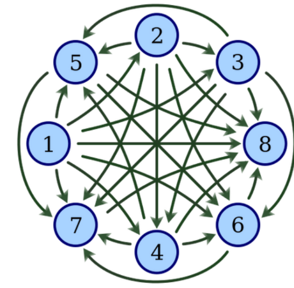


Fig. 8.7. A transitive tournament

Lemma 8.7 (Rédei, 1934)**.** *Any tournament contains at least one Hamiltonian path.*

If more than one Hamiltonian path exists, additional work is necessary in order to provide a ranking of the vertices. One easy solution is to provide a ranking by considering the strongly connected components of the graph. Recall that the condensation of a directed graph has been introduced in Definition 8.2. Then one has:

Lemma 8.8. *The condensation of any tournament is a transitive tournament.*

As a consequence, even for tournaments that are not transitive, the strongly connected components of the tournament are totally ordered. The theory of ranking in tournaments is very well developed and not trivial. A nice account for this theory is provided in [Mo]. We mention below just a few results, starting with the definition of a king !

Definition 8.9 (A king)**.** *In a tournament, a* king *is a vertex which can reach any other vertex with a path of length at most* 2*.*

In the next statement, we observe that kings always exist.

Proposition 8.10. *Any tournament possesses at least one king.*

Proof. The proof is by induction. Clearly, the statement holds if the tournament consists only in two vertices. So, let us assume that the statement is true for any tournament of n vertices, and let us consider a tournament $G = (V, E)$ of $n + 1$ vertices. Let y be any vertex in this tournament, and consider the n vertices tournament obtained by $G - \{y\}$. By assumption, this tournament has a king, which we denote by x. Let D be the set of vertices of $G - \{y\}$ containing x and all the vertices dominated by x. Clearly, x reaches all elements of D by a path of length at most 1 and all the element of $V \setminus D$ by a path of length 2 (passing through an element of D). If there exists in D one vertex which dominates y, then x is a king for G. Otherwise, y dominates all elements of D, and therefore can reach all elements of $G - \{y\}$ by a path of length at most 2. In this case, y is a king. □

Several results of this type can be found is the seminal paper [Ma] with beautiful illustrations as represented in Figure 8.8. For example, it is shown that a king is unique if and only if it is an emperor (it dominates all other vertices). Also it is shown that two kings can not coexist, but more than two can coexist.

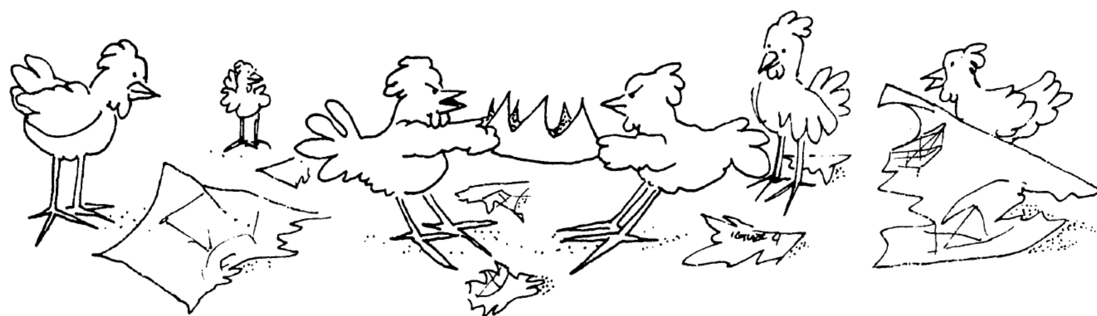


Fig. 8.8. About the King chicken theorems, picture from [Ma]

Another application of tournaments leads to a rather famous paradox in social choice theory. Very briefly, this so-called Condorcet paradox occurs because a collective preference can be cyclic, even if the preferences of individual voters are not cyclic. Let's be more explicit. A tournament with n vertices can be used to indicate the preferences between n candidates. This is often realized when candidates are evaluated by pair, with all possible pairs represented in a tournament. If the tournament is transitive, it means that the candidates are strictly ordered, but more complicated patterns can also appear. In this framework, each tournament represents the voting preferences provided by one person. What about the voting preferences of a group of persons ?

Definition 8.11 (Majority digraph). *Let $\{G_i\}_{i=1}^N$ be a family of tournaments of n vertices. The* majority digraph *based on this family is a tournament in which an edge is oriented from x to y if x dominates y in a majority of graphs G_i.*

Note that for simplicity we have assumed that a majority always exist. This will always be the case if the family $\{G_i\}_{i=1}^N$ consists in a odd number of tournaments, namely if N is odd. The majority digraph represents the preferences obtained over the N individual preferences. If the majority digraph is transitive, then the outcome of such a selection procedure is rather clear. However, more surprising situation can take place. For example, assume that $N = 3$ and that each individual voting preferences are organized as a transitive tournament. In this case, one can simply represent the graph by an ordered set, with the emperor on the top. When the majority digraph is drawn, the complexity of the tournament can be much higher. An example of this situation is represented in Figure 8.9. In this situation, one speaks about the Condorcet paradox because of the appearance of cycles in the majority graph.

The paradox can be explained by looking at the cycle generated by the three vertices A, B and C on Figure 8.9. Indeed, a majority of voters prefer A to B, a majority prefer B to C, and a majority prefer C to A. Thus, if candidate A wins, then a majority of the voters would have been happier if C had won, if candidate C wins, then a majority of the voters would have been happier if B had won, and if candidate B wins, then a majority of the voters would have been happier if A had won.

We refer to [29] for other references about social choice theory and for various additional links to applications of graph theory in social sciences.

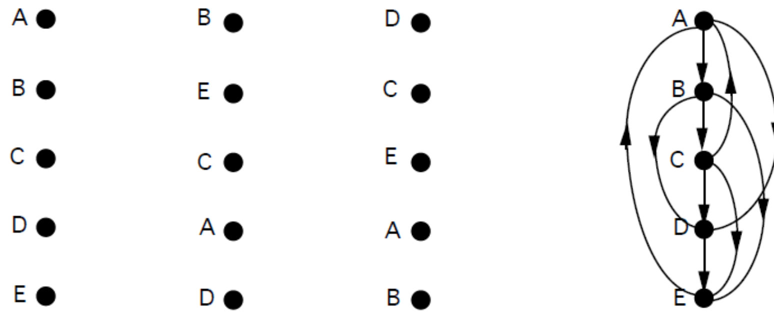


Fig. 8.9. 3 individual preferences, and the majority digraph, see Figure 9.3.3 of [GYA]

8.3 Project scheduling

In this section, digraphs are used for scheduling several activities which are interrelated: some activities have to be finished before others can start, while other activities are completely independent. Establishing such a graph helps for a better planing, for minimizing the necessary time for the completion of the project, and for identifying the key activities which can delay the entire project.

One way to represent such complex activities is to use an *activity-on-arc network*, or in short *AOA network*. In this representation, each directed edge of the graph represents one activity, with the head of the arrow indicating the direction of progress of the project. A weight on the edge represents the duration of the activity corresponding to this edge. Each vertex in the AOA network represents an event that coincides with the completion of one or more activities and with the beginning of new activities. Equivalently, the final vertex of an edge represents the completion of the activity, while the origin of an edge corresponds to the start of the activity. If the end vertex of an edge A coincides with the origin of an edge B, we say that the activity A is a predecessor of the activity B. In particular, it means that B can not start before A is completed. In Figure 8.10 different activities are organized. In (a) the completion of A is necessary before the start of B; in (b) A and B have to be completed before C can start, but A and B are independent; in (c) A has to be completed before the two independent activities B and C can start.

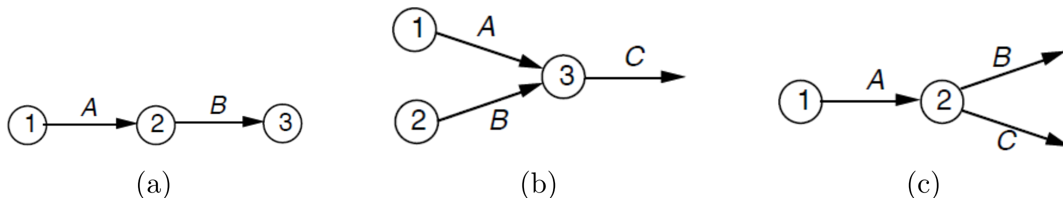


Fig. 8.10. Three AOA-networks

Let us now fix some rules when establishing an AOA network. Clearly, the graph corresponding to the network should be acyclic, otherwise none of the activities will

ever start. In addition, we shall require that there is a unique start (with indegree 0) and a unique end (with outdegree 0). Additional requirements are:

(i) Vertex 1 corresponds to the start of the project, while vertex N corresponds to its end,

(ii) Each activity with no predecessor is represented by an edge starting at the vertex 1,

(iii) Vertices are labeled with elements of $\mathbb{N}$ chosen such that for each edge, the initial vertex has a label which is smaller than the label of its end vertex,

(iv) Each activity is represented only by one edge in the network,

(v) The graph is simple and finite.

Note that in order to satisfy the rules imposed above, it is sometimes necessary to add a *dummy activity* which does not take any time. For example, if A and B can be performed simultaneously, with the same initial vertex and the same final vertex, then it is necessary to introduce an activity with a zero weight, as shown in Figure 8.11. Also, since the graph is simple, edges can be indexed without ambiguity by the labels of their endpoints. Thus, any edge will be denoted by (i, j) with $i \in \mathbb{N}$ corresponding to the label of the vertex at the origin of the edge, and $j \in \mathbb{N}$ corresponding to the label of the vertex at the end of the edge. By the convention imposed in the above rules, one has $i < j$.

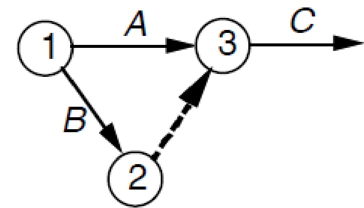


Fig. 8.11. Dummy activity

Let us now set a few notations: the indices $i, j \in \mathbb{N}$ correspond to labels on vertices.

(i) $\omega(i, j)$ denotes the weight on the edge (i, j) (the duration of the corresponding activity),

(ii) $\mathrm{ET}(i)$ denotes the earliest time at which the event corresponding to vertex i can occur,

(iii) $\mathrm{LT}(i)$ denotes the latest time at which the event corresponding to vertex i can occur without delaying the completion of the project,

(iv) $\mathrm{pred}(j)$ denotes the vertices preceding j, which means the set of vertices i for which there exists an edge (i, j). These vertices are called *the immediate predecessors* of the vertex j,

(v) $\mathrm{succ}(i)$ denotes the vertices following i, which means the set of vertices j for which there exists an edge (i, j). These vertices are called *the immediate successors* of the vertex i.

By convention, one sets $\mathrm{ET}(1) = 0$. In addition, the following recursive formula always holds:

$$\mathrm{ET}(j) := \max_{i \in \mathrm{pred}(j)} \{\mathrm{ET}(i) + \omega(i,j)\} \tag{8.1}$$

which means simply that the activities starting at j can take place only when all preceding activities are finished. Clearly, the following result holds:

Lemma 8.12. *In an AOA network, the earliest time* $\mathrm{ET}(i)$ *is given by the length of the longest path from vertex* 1 *to vertex* i.

In the rest of this section we provide some simple algorithms for computing quantities in an AOA network. The setting will always be the same: an AOA network with N vertices. Note that these algorithms use implicitly the fact that any acyclic and finite directed graph possesses a least one vertex with indegree 0 and one vertex with outdegree 0. The first algorithm provides the earliest time for every vertex, with the convention that $\mathrm{ET}(1) = 0$.

Algorithm 8.13 (Earliest event time). *Let* $G = (V,E)$ *be an AOA network.*

(i) *Set* $\mathrm{ET}(j) = 0$ *for any* $j \in \{1,\dots,N\}$.

(ii) *For* $i \in G$ *with* $\deg_{\mathrm{in}}(i) = 0$, *and for each* $j \in G$ *with* $(i,j) \in E$, *set*

$$\mathrm{ET}(j) := \max\{\mathrm{ET}(j), \mathrm{ET}(i) + \omega(i,j)\}.$$

Set $G := G - \{i\}$.

(iii) *Repeat (ii) until* $G = \emptyset$.

(iv) *Return* $\mathrm{ET}(1), \mathrm{ET}(2), \dots, \mathrm{ET}(N)$.

The next algorithm provides the latest event time. Its computation is rather similar to the computation of the earliest time, but it goes backward. The necessary convention is that $\mathrm{LT}(N) = \mathrm{ET}(N)$. In order to understand the algorithm, consider a vertex j which is a immediate successor of a vertex i, see Figure 8.12. If the event corresponding to vertex i occurs after $\mathrm{LT}(j) - \omega(i,j)$, then event j will occur after $\mathrm{LT}(j)$, thereby delaying the completion of the project. Since this is true for any immediate successor of i, $\mathrm{LT}(i)$ is the minimum of these differences taken over all immediate successors of i. This leads to a relation similar to (8.1) and which reads

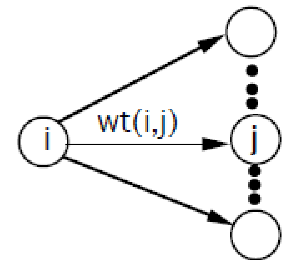


Fig. 8.12. Immediate successor

$$\mathrm{LT}(i) = \min_{j \in \mathrm{succ}(i)} \{\mathrm{LT}(j) - \omega(i,j)\}. \tag{8.2}$$

Based on this relation, one infers the following algorithm:

Algorithm 8.14 (Latest event time)**.** *Let $G = (V, E)$ be an AOA network, and let T be the earliest completion time.*

(i) *Set* $\mathrm{LT}(j) = T$ *for any* $j \in \{1, \dots, N\}$.

(ii) *For* $j \in G$ *with* $\deg_{\mathrm{out}}(j) = 0$*, and for each* $i \in G$ *with* $(i, j) \in E$*, set*

$$\mathrm{LT}(i) := \min\{\mathrm{LT}(i), \mathrm{LT}(j) - \omega(i, j)\}.$$

Set $G := G - \{j\}$.

(iii) *Repeat (ii) until* $G = \emptyset$.

(iv) *Return* $\mathrm{LT}(1), \mathrm{LT}(2), \dots, \mathrm{LT}(N)$.

By using the previous two concepts, namely the earliest time ET and the latest time LT, one can introduce more more useful concept: the *total float*. For the activity on the edge (i, j), the total float $\mathrm{TF}(i, j)$ corresponds to the amount by which this activity can be increased without delaying the full project. Equivalently, it corresponds to the amount of time the start of the activity (i, j) can be delayed without having an effect on the project. Knowing this information is an important issue since it might allow to start this activity at an optimal time during a certain interval of time. Once the previous two algorithms have been performed, one can directly compute the total float. The proof of the following statement is easy, see also [GYA, Prop. 9.4.3].

Lemma 8.15. *The total float for the activity* (i, j) *is given by*

$$\mathrm{TF}(i, j) = \mathrm{LT}(j) - \mathrm{ET}(i) - \omega(i, j).$$

Let us conclude this section with one more remark. Clearly, a critical activity is one activity for which the total float is 0. Accordingly, any path in the graph from the vertex 1 to the vertex N made of critical activities is called a critical path. Such paths are the longest ones in the graph, and dictate the duration of the entire project. Usually, such paths are not unique.

Chapter 9

Flows

In this chapter we continue the investigations on directed graphs.

9.1 Capacity, flows and cuts

For shortness, a finite directed and loopless graph $G = (V, E)$ will simply be called a *network*. We start by introducing a few definitions related to arbitrary networks. First of all and in relation with the indegree and the outdegree functions, let us define two natural notions.

Definition 9.1 (In and out sets)**.** *For any vertex x of a network $G = (V, E)$ we set* $\mathrm{in}(x) := \{e \in E \mid \mathrm{t}(e) = x\}$ *and* $\mathrm{out}(x) = \{e \in E \mid \mathrm{o}(e) = x\}$*.*

In other terms, $\mathrm{in}(x)$ corresponds to the set of edges targeting x, while $\mathrm{out}(x)$ corresponds to the set of edges leaving x. Clearly, the cardinality of the first set is $\deg_{\mathrm{in}}(x)$ while the cardinality of the second set is $\deg_{\mathrm{out}}(x)$.

Let us also introduce a rather convenient notation which is somehow related to the A-B-path introduced in Definition 5.5. Let $G = (V, E)$ be a network and let A, B be two subsets of V. For any $e \in E$ we set

$$e \in (A, B) \Longleftrightarrow \mathrm{o}(e) \in A \text{ and } \mathrm{t}(e) \in B. \tag{9.1}$$

In other words, $e \in (A, B)$ means that the edge e starts in A and ends in B. Observe that for this definition, it is not necessary that A and B are disjoint, of even different. In the sequel, special pairs of subsets of V will play an important role. For that reason, for any $U \subset V$ we write U^c for $V \setminus U$. Clearly, U and U^c define a partition of V. In this case relation (9.1) reads

$$e \in (U, U^c) \Longleftrightarrow \mathrm{o}(e) \in U \text{ and } \mathrm{t}(e) \notin U.$$

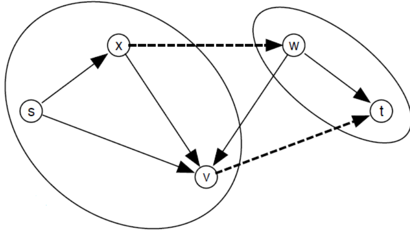


Fig. 9.1. st-network with cut

From now on we shall consider networks with two distinguished vertices s and t satisfying $\deg_{\text{out}}(s) \neq 0$ and $\deg_{\text{in}}(t) \neq 0$ (and impose that $s \neq t$). Such networks are called st-networks[1]. Note that the AOA networks introduced in Section 8.3 are special instances of st-network. The two distinguished vertices are usually called the *source* and the *sink* (or *target*). If G is a st-network, any pair (U, U^c) with $s \in U$ and $t \in U^c$ is called a *cut of* G. Special examples of cuts are $\big(\{s\}, \{s\}^c\big) \equiv \big(\{s\}, V \setminus \{s\}\big)$ and $\big(\{t\}^c, \{t\}\big) \equiv \big(V \setminus \{t\}, \{t\}\big)$.

From now on, we shall endow the edges of the network with weights. In the present context, an edge weight $\omega : E \to [0, \infty)$ is called *capacity function*, or simply a *capacity*, and is denoted by the letter c. We also introduce a second type of functions on edges. Note that these functions have some relations with the capacity function.

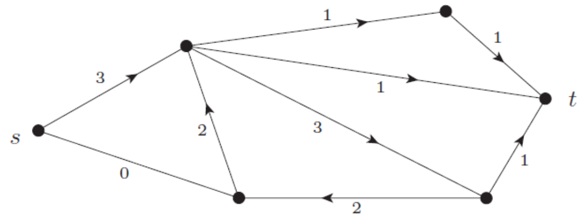


Fig. 9.2. st-network with capacity

Definition 9.2 (Flow). *Let $G = (V, E)$ be a st-network endowed with a capacity c. A* flow *on G is a function $f : E \to \mathbb{R}_+$ satisfying the following conditions:*

(i) $f(e) \leq c(e)$ for any $e \in E$,

(ii) For any $x \in V$ with $x \notin \{s, t\}$ one has the conservation constraint

$$\sum_{e \in \text{in}(x)} f(e) = \sum_{e \in \text{out}(x)} f(e). \tag{9.2}$$

The equality (9.2) is also called *the conservation of flow*. It is a requirement which naturally appears in several applications. It prevents the accumulation of the quantity represented by f at any vertex.

Let us now come back to two subsets A, B of V. For any capacity or any flow on G we set

$$c(A, B) := \sum_{e \in (A,B)} c(e) \qquad \text{and} \qquad f(A, B) := \sum_{e \in (A,B)} f(e).$$

In particular, it follows from the conservation constraint that for any $x \in V$ with

[1] It is sometimes required that $\deg_{\text{in}}(s) = 0$ and that $\deg_{\text{out}}(t) = 0$, but this is not strictly necessary. However, by adding new vertices, one can always come back to this situation. Also, networks with multiple sources and multiple sinks can easily be transformed into a network with a single source and a single sink.

$x \notin \{s,t\}$ one has

$$f\big(\{x\},\{x\}^c\big) = f\big(\{x\},V\big) = \sum_{e\in\text{out}(x)} f(e) = \sum_{e\in\text{in}(x)} f(e) = f\big(V,\{x\}\big) = f\big(\{x\}^c,\{x\}\big). \tag{9.3}$$

In the special case of a cut (U,U^c), the quantity $c(U,U^c)$ is also called *the capacity of the cut* (U,U^c).

So far, the source s and the sink t of the st-network have not played a special role. The next definition is related to them:

Definition 9.3 (Value of a flow)**.** *Let $G=(V,E)$ be a st-network endowed with a capacity c, and let f be a flow on G. We define the* value $\text{val}(f)$ *of the flow f, also denoted by $|f|$, by*

$$\text{val}(f) := \sum_{e\in\text{out}(s)} f(e) - \sum_{e\in\text{in}(s)} f(e).$$

Clearly, this definition put more emphasize on the source s than on the sink t. We shall see later on that this asymmetry is not a real one. The next statement is of central importance in our framework.

Proposition 9.4. *Let $G=(V,E)$ be a st-network endowed with a capacity c, let f be a flow on G, and let (U,U^c) be any cut of G. Then, one has*

$$\text{val}(f) = f(U,U^c) - f(U^c,U) \leq c(U,U^c). \tag{9.4}$$

Before the proof, observe that (9.4) contains two pieces of information. The first one is about the invariance of the expression $f(U,U^c) - f(U^c,U)$ for any cut (U,U^c). The second one is that $\text{val}(f)$ is dominated by the capacity of any cut (U,U^c).

Proof. From (9.3) one infers that for any $x \in U\setminus\{s\}$ one has $f\big(\{x\},V\big) - f\big(V,\{x\}\big) = 0$. Thus,

$$\begin{aligned}
\text{val}(f) &= \sum_{e\in\text{out}(s)} f(e) - \sum_{e\in\text{in}(s)} f(e) \\
&= f\big(\{s\},V\big) - f\big(V,\{s\}\big) \\
&= \sum_{x\in U} \Big\{ f\big(\{x\},V\big) - f\big(V,\{x\}\big)\Big\} \\
&= f(U,V) - f(V,U) \\
&= f(U,U) + f(U,U^c) - f(U,U) - f(U^c,U) \\
&= f(U,U^c) - f(U^c,U)
\end{aligned}$$

which provides the equality in the statement. For the inequality, observe that

$$f(U,U^c) - f(U^c,U) \leq c(U,U^c) - f(U^c,U) \leq c(U,U^c) \tag{9.5}$$

since $f(U^c,U) \geq 0$. This inequality corresponds to the one of the statement. □

By choosing $U = \{t\}^c$, one directly gets

$$\begin{aligned}\mathrm{val}(f) &= f\big(\{t\}^c, \{t\}\big) - f\big(\{t\}, \{t\}^c\big) \\ &= f\big(V, \{t\}\big) - f\big(\{t\}, V\big) = \sum_{e\in \mathrm{in}(t)} f(e) - \sum_{e\in \mathrm{out}(t)} f(e).\end{aligned}$$

Thus, the following statement corrects the apparent asymmetry of Definition 9.3 :

Corollary 9.5. *In the previous setting, one has* $\mathrm{val}(f) = \sum_{e\in \mathrm{in}(t)} f(e) - \sum_{e\in \mathrm{out}(t)} f(e)$.

There is an other important consequence of Proposition 9.4 which has to be emphasized. The value $\mathrm{val}(f)$ is always smaller than or equal to the capacity of the any cut (U, U^c). In particular, it means that

$$\mathrm{val}(f) \leq \min\big\{c(U, U^c) \mid U \subset V \text{ with } s \in U \text{ and } t \in U^c\big\}.$$

For that reason, it is natural to call a subset $U \subset V$ realizing this inequality a *minimum cut* of the network. Such a cut will be denoted by $\big(U^*, (U^*)^c\big)$.

In the sequel we shall look for a maximal flow f^*, which means a flow on G which satisfies $\mathrm{val}(f) \leq \mathrm{val}(f^*)$ for any flow f on G. It follows from the previous observations that the inequality

$$\mathrm{val}(f^*) \leq c\big(U^*, (U^*)^c\big)$$

always holds. In addition, if for some flow f and some cut (U, U^c) one has $\mathrm{val}(f) = c(U, U^c)$, then it turns out that f is a maximal flow, and that (U, U^c) is a minimal cut. Observe finally that (9.5) provides a condition for a flow to be maximal. Indeed, suppose that there exist a flow and a cut (U, U^c) satisfying $f(e) = c(e)$ for any $e \in (U, U^c)$ and $f(e) = 0$ for any $e \in (U^c, U)$, then the inequalities in (9.5) are saturated, and one deduces that f is a maximal flow and that (U, U^c) is a minimum cut.

9.2 Maximum flow problem

In this section, we look for the maximal flow on a given st-network endowed with a capacity c. The simplest solution for increasing a flow f is through a single st-path in the network. Indeed, suppose that there exists a path from s to t satisfying the condition $\epsilon := \min\{c(e) - f(e)\} > 0$, where the minimum is taken over all edges of the path. Then the flow f_ϵ defined by $f_\epsilon(e) = f(e) + \epsilon$ if e belongs to the st-path, and $f_\epsilon(e) = f(e)$ otherwise, is a new flow on G. Indeed, it is easy to check that the conservation constraint is still satisfied for the new flow. A representation of this situation is provided in Figure 9.3. In the first picture, the values of the flow and of the capacity are represented with the notation flow/capacity[2]. In the second picture, a st-path with $\epsilon = 1$ is presented. Note however that this simplest solution is rarely the best one. Its weakness is that a single path is considered, while a network usually admits several st-paths.

[2]Note that different authors use different notations for representing the flow and the capacity. One easily finds out who is who since the inequality $f(e) \leq c(e)$ always holds.

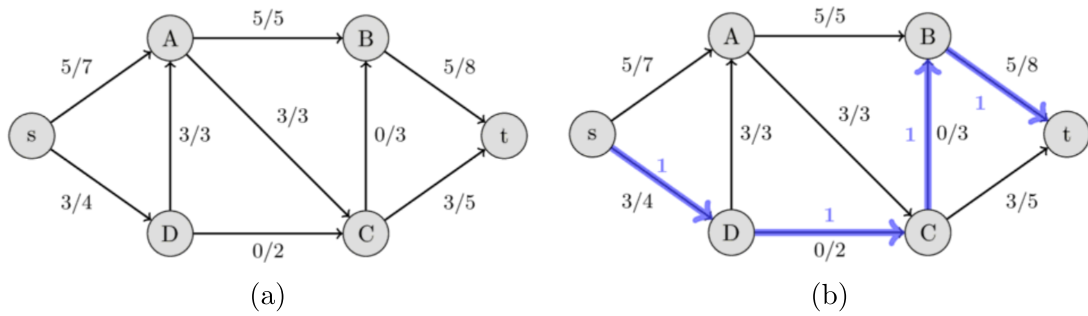


Fig. 9.3. Flow increasing through one path, from [30]

Before developing some tools for the search of a maximal flow, it is natural to wonder if such a flow exists ? The answer is yes as long as the capacity takes integer values. More precisely, the following statement holds:

Theorem 9.6 (Max-flow Min-cut theorem)**.** *In every st-network with integer-valued capacity function there exists a maximal flow which is also integer-valued.*

This theorem is due to Ford and Fulkerson and a proof can be found in [Die, Thm. 6.2.2]. An extension to capacities with values in the set of rational numbers is easy (just multiply by a sufficiently large integer to be back to the integer-valued setting). On the other hand, for irrational capacities, the convergence of the algorithm to a maximal flow might fail.

In order to extend the construction mentioned above with one st-path, let us introduce a generalization of a path.

Definition 9.7 (st-quasi path)**.** *In a st-network a* st-quasi path *or* st-semi path *is a path on the underlying undirected graph which starts at s and ends at t.*

When considering such a quasi path, the edges can be divided into two families: the ones going in the direction of the path, and the ones going backward. Accordingly, they are called *forward edges* or *forward arcs*, and *backward edges* or *backward arcs*.

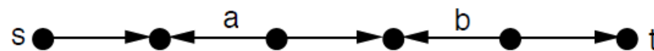


Fig. 9.4. 3 forward edges, 2 backward edges

Consider now a flow on a st-network endowed with a capacity. We shall say that a st-quasi path is f-augmenting if $f(e) < c(e)$ for any forward edge e on the path, and $f(e) > 0$ on any backward edge e on the path. In particular, if one sets

$$\Delta_e = \begin{cases} c(e) - f(e) & \text{if } e \text{ is a forward edge} \\ f(e) & \text{if } e \text{ is a backward edge} \end{cases}$$

then $\Delta_e > 0$ for any f-augmenting st-quasi path. The quantity Δ_e is called the *slack* on edge e. So far, this quantity has been computed on each edge individually. However, for the conservation of the flow, one can not change the flow on edges independently, and one has to consider a path as a whole. For that reason, for any f-augmenting st-quasi path P let us set

$$\Delta_P := \min\{\Delta_e \mid e \in P\}.$$

Note that this notion corresponds to the value ϵ introduced at the beginning of this section for an oriented st-path, and corresponds to the value 1 on the blue path in Figure 9.3. By two minutes of thought one easily infers the following result:

Lemma 9.8. *Let $G = (V, E)$ be a st-network endowed with a capacity c, let f be a flow on G, and let P be a f-augmenting st-quasi path. Let us set*

$$f_P(e) := \begin{cases} f(e) + \Delta_P & \text{if } e \text{ is a forward edge of } P \\ f(e) - \Delta_P & \text{if } e \text{ is a backward edge of } P \\ f(e) & \text{otherwise} \end{cases}.$$

Then f_P is a new flow on G, and $\operatorname{val}(f_P) = \operatorname{val}(f) + \Delta_P$.

In addition to the previous result, one can also infer that a flow is maximal if and only if there does not exist any f-augmenting st-quasi path. A proof of this statement is presented in [GYA, Thm. 10.2.3]. Putting the information obtained so far, one has essentially proved Theorem 9.6. However, the missing information so far is *how do we find f-augmenting st-quasi path ?* Indeed, once such a quasi path has been identified, Lemma 9.8 provides us with an increased flow.

There exist several algorithms for identifying f-augmenting st-quasi paths, which are more or less complicated, but also less or more time consuming. We provide only the one of Edmond and Karp, based on the early solution provided by Ford and Fulkerson. The main idea for is to construct a tree starting at s. Once the sink t is reached, one uses the path between s and t as a st-quasi path.

Recall firstly that the algorithm for constructing a tree has been provided in Algorithm 4.4. When growing the tree, the elements of $\operatorname{Front}(G, T_i)$ play a crucial role. For unoriented graphs, $\operatorname{Front}(G, T_i)$ consists of edges with one endpoint in T_i and one endpoint outside of T_i. In the current application, we shall consider $\operatorname{Front}(G, T_i)$ consisting of two types of edges e:

(i) $\mathrm{o}(e) \in T_i$, $\mathrm{t}(e) \notin T_i$, and $c(e) - f(e) > 0$,

(ii) $\mathrm{o}(e) \notin T_i$, $t(e) \in T_i$, and $f(e) > 0$.

Such edges are usually called *usable*. They are clearly related to the forward and backward edges in the final st-quasi path. The choice of an edge inside $\operatorname{Front}(G, T_i)$ will then be performed following the breadth-first search, namely the frontier edge $e_{i+1} \in \operatorname{Front}(G, T_i)$ is chosen with a tree endpoint at x_j with the minimal number j (starting from $j = 0$ and then upward).

The following algorithm puts these ideas together. Note that a function backpoint : $V \to V$ is used during the implementation.

Algorithm 9.9 (Finding a f-augmenting st-quasi path)**.**

(i) *Set* $T_0 := \{s\}$ *and fix* $i := 0$*,*

(ii) *In* $\mathrm{Front}(G, T_i)$*, choose the edge* e_{i+1} *with tree endpoint* x_j *with the minimal number* j*, set* x_{i+1} *as the non-tree endpoint of* e_{i+1}*, define* $\mathrm{backpoint}(x_{i+1}) := x_i$*, set* $T_{i+1} = T_i \sqcup \{e_{i+1}\}$*, and set* $i := i + 1$*,*

(iii) *Repeat (ii) until* $t \in T_i$ *or until* $\mathrm{Front}(G, T_i) = \emptyset$*,*

(iv) *If* $t \in T_i$*, reconstruct the* f*-augmenting* st*-quasi path starting from* t *and using the information contained in* backpoint*, while if* $\mathrm{Front}(G, T_i) = \emptyset$*, then* (T_i, T_i^c) *is a minimum cut.*

Note that one has to be slightly careful in the above algorithm since it provides a st-quasi path, and not a st-path. This implies that x_{i+1} corresponds sometimes to $\mathrm{o}(e_{i+1})$ and sometimes to $\mathrm{t}(e_{i+1})$. This depends on the type of usable edge e_{i+1}. Also, if the tree does not reach t, it means that the flow f is already a maximal flow, and therefore the algorithm can be used for exhibiting a minimal cut.

As a final remark, by combining Lemma 9.8 and the previous algorithm one easily gets a maximal flow on any st-network endowed with an integer-valued capacity. It is enough to start with a flow f identically equal to 0, find a f-augmenting st-quasi path with the algorithm, update the flow, and iterate the process again. We illustrate the construction in Figure 9.5. In these figure, A is the source, G is the sink, and the notation flow/capacity is used. The final value $\mathrm{val}(f^*)$ of the maximal flow is 5. Note that the increase of the length of the paths is due to the choice of the bfs algorithm. At every step one looks for the shortest possible path.

9.3 Applications

In this section we briefly sketch a few applications of the results obtained in the previous section.

9.3.1 Flow and Menger's theorem

The first application of the Max-flow Min-cut theorem provided in Theorem 9.6 is usually dedicated to the proof of Menger's theorem, both for oriented and unoriented graphs, see Section 5.2. We shall not present the construction here, but refer for [GYA, Sec. 10.3]. We only provide two statements which can be easily proved and which might be useful later on. Recall that the notion of internally disjoint paths has been introduced in Definition 5.3 and was related to the absence of common vertices in different paths. Similarly, edge-disjoint paths correspond to paths which do not share any common edge.

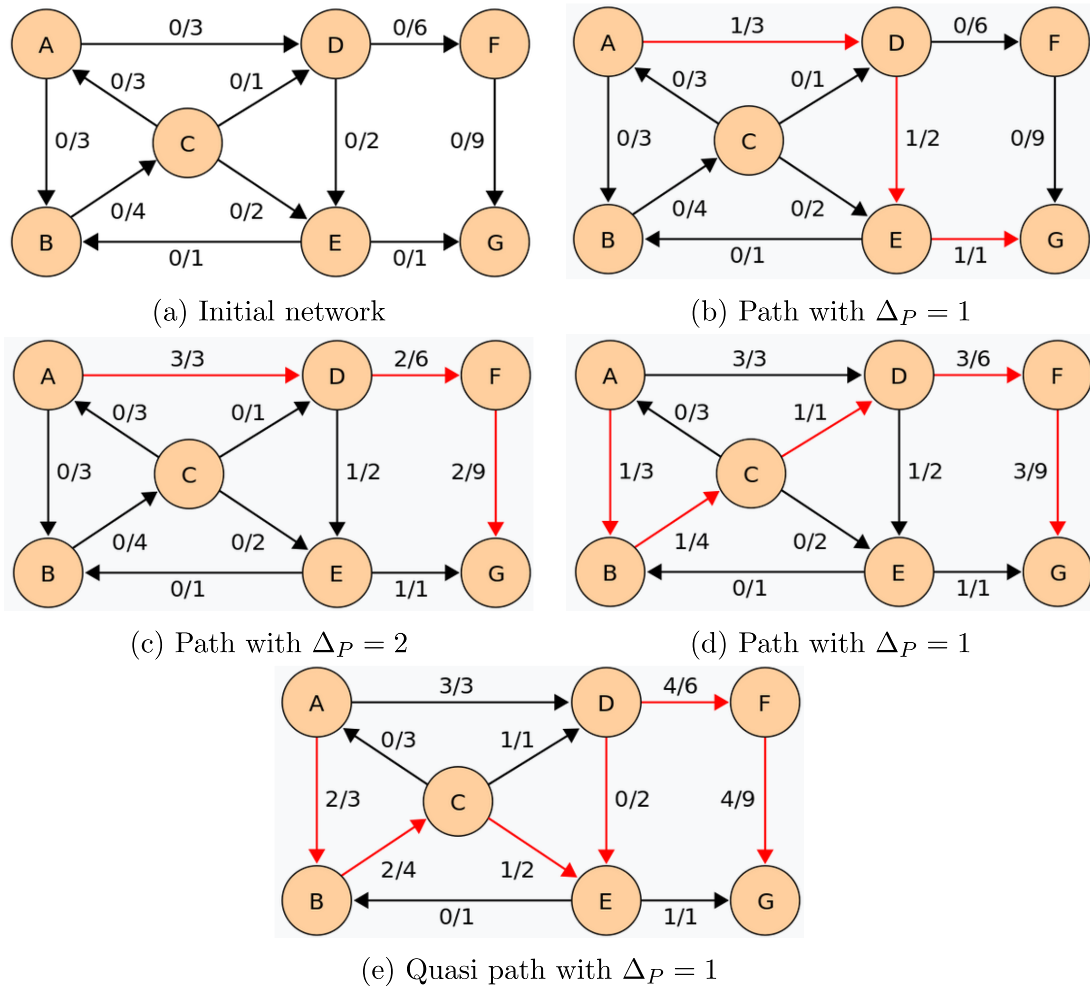


Fig. 9.5. Construction of a maximal flow, figures from [31]

Lemma 9.10. *Let $G = (V, E)$ be a st-network satisfying the three conditions:*

$$\deg_{\text{out}}(s) - \deg_{\text{in}}(s) = m = \deg_{\text{in}}(t) - \deg_{\text{out}}(t)$$

for some $m \in \mathbb{N}^*$, and $\deg_{\text{in}}(x) = \deg_{\text{out}}(x)$ for all $x \in V \setminus \{s, t\}$. Then there exist m edge-disjoint st-paths in G.

Proposition 9.11. *Let $G = (V, E)$ be a st-network endowed with a constant capacity $c = 1$. Then $\text{val}(f^*)$ for a maximal flow in G is equal to the number of edge-disjoint st-paths in G*

9.3.2 Matching

Recall that the notion of a matching has been introduced in Definition 6.4 and corresponds to a set of edges having no common endpoints. If e belongs to a matching M

and has endpoints x and y, we also say that x is matched with y by M. For a given graph, we speaks about a *maximum matching* if the matching contains the greatest possible number of edges. A problem which appears quite frequently is to look for a maximum matching in a bipartite graph, see Figure 9.6. How many edges are contained a maximum matching ?

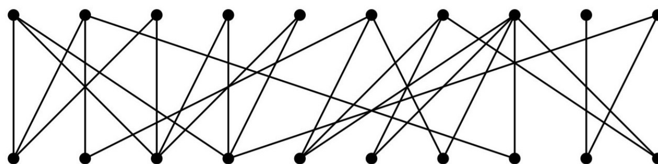


Fig. 9.6. Looking for a maximum matching in a bipartite graph

Let us transform this problem into a maximal flow problem, as solved in the previous section. Let $G = (V, E)$ be the initial bipartite graph with bipartition subsets V_s and V_t, and let us construct a st-network $G' = (V', E')$ based on G as follows:

(i) $V' := V \sqcup \{s, t\}$,

(ii) The set E' consists of three types of oriented edges: one oriented edge from s to each vertex of V_s, one oriented edge from V_s to V_t for each edge of E, one oriented edge from each vertex of V_t to t,

In addition, we consider G' endowed with the constant capacity $c = 1$. A representation of this construction is provided in Figure 9.7.

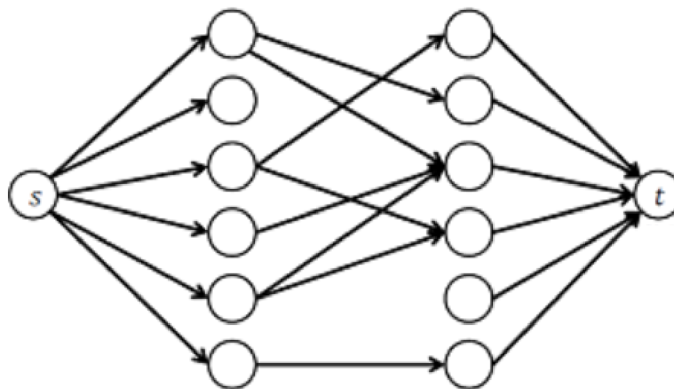


Fig. 9.7. The construction of a st-network

Once this st-network with capacity $c = 1$ is available, tools from the previous section can be used. The relation between the maximal flow problem, and the current maximum matching is established in the following statement, see [GYA, Prop. 10.4.1] for a proof.

Proposition 9.12. *Let G be a finite bipartite graph, and let G' be the st-network constructed from G as mentioned above. Then there is a bijective relation between integer-valued flows on G' and matching in G. In particular, f^* is a maximal flow if and only if* $\mathrm{val}(f^*)$ *corresponds to the number of elements of a maximum matching.*

With this result at hand, one can now apply the Algorithm 9.9 for constructing a maximal flow f^*. Once done, the edges in a maximum matching if obtained by keeping all $e \in E \subset E'$ with $f^*(e) = 1$.

9.3.3 Transversals

Another application of the maximal flow problem is related to the transversal problem.

Definition 9.13 (Transversal)**.** *Let A be a finite set, and let $\mathcal{F} = \{S_1, \dots, S_r\}$ with $S_j \subset A$ be a finite family of subsets of A. A* transversal *for $\mathcal{F}$ is a sequence $T = (a_1, \dots, a_r)$ with $a_j \in S_j$ and $a_j \neq a_k$ for any $j, k \in \{1, \dots, r\}$ and $j \neq k$.*

In other terms, a transversal consists in choosing one element in each subset S_j such that the r chosen elements are different. Fortunately, the problem of finding a transversal can be reformulated in terms of a bipartite graph and a matching problem. Indeed, let us define the bipartite graph $G = (V, E)$ with bipartition subsets $V_\mathcal{F}$ and V_A defined by $V_\mathcal{F} = \{S_1, \dots, S_r\}$ and $V_A = A$. Note that in this definition, S_j is just considered as a vertex, it is not considered as a set containing other elements. We also define $e \in E$ with endpoints $S_j \in V_\mathcal{F}$ and $a \in V_A$ whenever $a \in S_j$. Then, finding a transversal corresponds to finding a matching in this bipartite graph. Figure 9.8 corresponds to the transposition of the initial problem (5 persons interested in 6 gifts, but with some preferences) into a bipartite graph. Obviously, one is interested in the situation when all elements of $V_\mathcal{F}$ are enpoints of the edges in the matching. For that purpose, the next definition is natural:

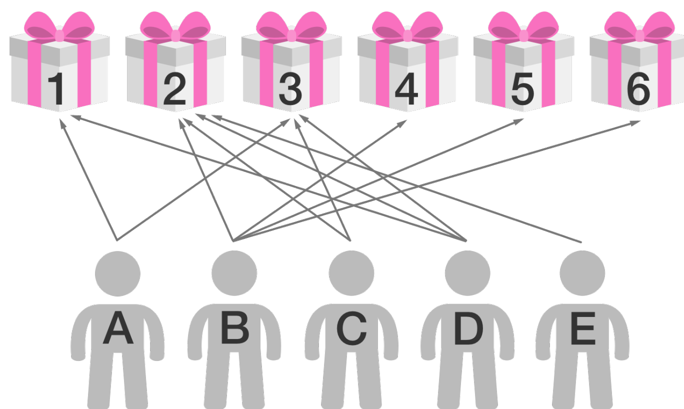


Fig. 9.8. Looking desperately for a transversal, from [32]

Definition 9.14. *Let G be a bipartite graph with bipartition subsets V_1 and V_2. A matching M in G is* V_1-saturated *if each vertex of V_1 are endpoints of the edges in M.*

It follows clearly from this definition that if the matching M is V_1-saturated, then the cardinality of M and of V_1 should be equal. An example of a $V_\mathcal{F}$-saturated matching is provided in Figure 9.9.

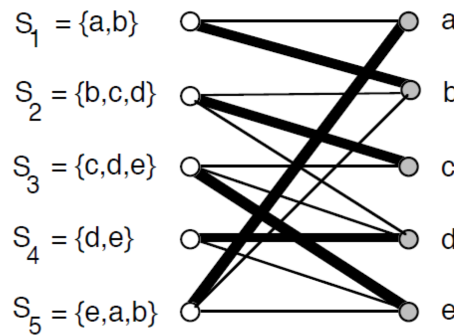


Fig. 9.9. A $V_{\mathcal{F}}$ saturated bipartite graph

Once the initial problem has been recast in the framework of a bipartite graph, the tools developed before can once again be used. However, there is one important issue: when is it possible to find a $V_{\mathcal{F}}$-saturated matching. Or equivalently, when is it possible to find a transversal ? The following theorem provides a necessary and sufficient condition. It is stated in the framework of bipartite graph, but its transposition to the initial problem is straightforward. For its statement, recall that the set $N(x)$ of neighbours of the vertex x has been introduced in Definition 1.4 and corresponds to the set of all vertices connected to x. For a subset $U \subset V$ of vertices, we set $N(U)$ for the set of all vertices which are connected to at least one element of U.

Theorem 9.15 (Hall's theorem for bipartite graphs)**.** *Let G be a finite bipartite graph with bipartition subsets V_1 and V_2. Then G has a V_1-saturated matching if and only if for any subset U of V_1 one has $|U| \leq |N(U)|$, where $|U|$ and $|N(U)|$ denote the cardinality of these sets.*

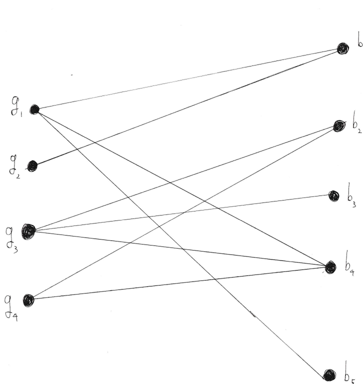
A proof of this theorem is provided in Section 9.4.1. It uses the theory of maximal flow developed in the previous section. For our initial transversal problem, it means that a transversal exists if the union of any k different subsets S_j contains at least k distinct elements of A.

9.4 Appendix

9.4.1 Hall's marriage theorem

This section has been studied and written by Dam Truyen Duc and Atsuya Watanabe.

The marriage theorem, proved in 1935 by Philip Hall, answers the following question, known as the marriage problem: if there is a finite set of girls, each of whom knows several boys. Under what conditions can all the girls marry the boys in such a way that each girl marries a boy she knows? For example, if there are four girls $\{g_1, g_2, g_3, g_4\}$ and five boys $\{b_1, b_2, b_3, b_4, b_5\}$, and the friendship are shown below, then a possible solution is for g_1 to marry b_4, g_2 to marry b_1, g_3 to marry b_3, and g_4 to marry b_2.



Theorem: A necessary and sufficient condition for a solution of the marrige problem is that each set of k girls collectively knows at least k boys, for $1 \leq k \leq m$.

Proof. The necessity is clear, so we can concentrate on the sufficiency.

Let the girls be $g_1, g_2, g_3, \ldots, g_n$, and the boys be $b_1, b_2, b_3, \ldots, b_m$, with $m \geq n$. The relation between a girl and a boy and whether they know each other constructs a bipartite graph $V = V_1 \cup V_2$ where V_1 is the set of vertices $g_1, g_2, \ldots, g_n$, V_2 is the set of vertices $b_1, b_2, \ldots, b_m$, and there exists an edge of G between g_i and b_j if the girl g_i knows the boy b_j. In this way, we construct a simple bipartite graph.

Let A is a subset of V_1. Denote $P(A)$ for the subset of V_2 that all the edges from A to V_2 have an endpoint in $P(A)$, and each vertex in $P(A)$ has an edge connected to a vertex in A . Let $|A|$ represent the number of elements contained in the subset A. Then, proving the necessity is to proving that if any A subset of V_1 satisfies $|A| \leq |P(A)|$, then there is a complete matching from V_1 to V_2, namely any g_i is connected with a different b_j. Thus, assume that

$$|A| \leq |P(A)|$$

for any subset A of V_1.

Let us add to G a vertex of v adjacent to (and only to) every vertex in V_1 and a vertex w adjacent to (and only to) every vertex in V_2. Menger's theorem says that if S is a vw-separating set, then $|S| \geq \#$ internalydisjoint path form v to w. (here, # means "the number of"). Clearly, $|V_1| \geq$ #internally disjoint path from v to w because V_1 is a vw-separating set.

Let $S = A \cup B$ be a v-w separator in which A is a subset of V_1 and B is a subset of V_2. Then $|S| \geq$ #internally disjoint path from v to w by Menger's theorem. Clearly, $(V_1 - A)$ and $(V_2 - B)$ are not connected with each other by any edge since if they are connected by some edges to each other, then $A \cup B = S$ would not be a separator of v-w. Thus, $P(V_1 - A)$ is a subset of B since $V_1 - A$ is connected with vertices in V_2 but not in $V_2 - B$ as we argued above.

With our assumption, we obtain that

$$|V_1 - A| \leq |P(V_1 - A)| \leq |B.|$$

As $|S| = |A| + |B|$ (A subset of V_1 and B subset of V_2 are disjoint),

$$|S| \geq |A| + |V_1 - A| = |V_1|.$$

From the above relations, we have

$$\begin{aligned}
|V_1| &\geq \text{\#internally disjoint path from } v \text{ to } w \\
|S| &\geq \text{\#internaly disjoint path from } v \text{ to } w \text{ by Menger's theorem.} \\
|S| &\geq |V_1|
\end{aligned}$$

Thus, we get the total relation:

$$|S| \geq |V_1| \geq \text{ \#internally disjoint path from } v \text{ to } w.$$

But Menger's theorem says that $|S|_{min}$ = max(#internally disjoint path from v to w). Thus, $|S|_{min} = |V_1|$ = max(# internally disjoint path from v to w). Therefore, $|V_1|$ = max(#internally disjoint path from v to w. This means there exists a set of $|V_1|$ internally disjoint paths from v to w in which each path includes a different vertex in V_1 and a different vertex in V_2 from any other path in order to satisfy the internally disjoint condition. Thus, we have the perfect matching from this set of internally disjoint paths. □

Chapter 10

Random graphs: the $\mathcal{G}(n,p)$ model

In this chapter we touch the surface of random graphs, much more could and should be said. We also do it without requiring any real prerequisite in probability theory, and for that reason several arguments will only be sketched.

Let us start by recalling some notations. For any $p \in \mathbb{N}$ we set

$$p! := p \cdot (p-1) \cdot (p-2) \dots 2 \cdot 1$$

for the factorial of p. Also, for any two positive integers p and q with $p \geq q$ we set $\binom{p}{q} := \frac{p!}{q!(p-q!)}$ for the binomial coefficients. This number represents the number of ways to choose an (unordered) subset of q elements from a fixed set of p elements. This number can also be written

$$\binom{p}{q} = \frac{p \cdot (p-1) \cdot (p-2) \dots (p-q+1)}{q \cdot (q-1) \cdot (q-2) \dots 2 \cdot 1}.$$

In particular, the number $\binom{p}{2}$ will often appear, and is equal to $\frac{1}{2}p(p-1)$. One can also observe that it corresponds to the number of elements in the upper (of lower) half of a square $p \times p$ matrix once the diagonal has been eliminated.

10.1 Basic results

The Gilbert-Erdős-Rényi model for random graphs is certainly the simplest, most natural and most studied model. It is often simply called *random graphs*, but also *Poisson random graphs* or *Bernoulli random graphs*[1]. Since other models of random graphs exist, we shall use the name *Gilbert-Erdős-Rényi model* in honor of the authors who introduced and popularized the model in late 1950s and early 1960s. Note that for this section we shall mainly follow the approach proposed in Chapter 11 of [Ne], and borrow several pictures from this reference.

[1] The name Bernoulli comes from the fact that the existence of an edge follows a Bernoulli distribution. The reason for Poisson will appear later on.

The Gilbert-Erdős-Rényi model is often denoted by $\mathcal{G}(n,p)$[2]. The idea behind this notation is the following: for any $n \in \mathbb{N}$ we consider n fixed vertices. Then, the number $p \in [0,1]$ represents the probability that an edge between two vertices exists. It is assumed that the graphs are simple (no loop, no multiple edge) and unoriented, and that the existence of any edge is independent of the existence (or non-existence) of any other edge. Thus, given two distinct vertices x_j and x_k, the probability that there exists an edge between them is p, no matter if other edges have x_j or x_k as endpoints (but not both) and no matter if other edges exist between the other vertices, see Figure 10.1. For the time being, we shall consider a fixed p, but this parameter could also depend on other quantities, and could depend for example on n, see Remark 10.1.

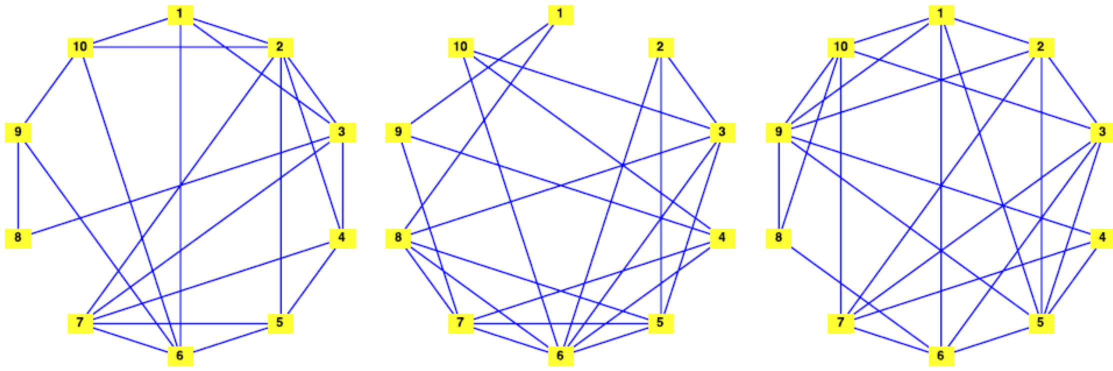


Fig. 10.1. Three elements of $\mathcal{G}\big(10, \frac{1}{2}\big)$

Observe firstly that there exists $\binom{n}{2}$ distinct pairs of vertices between n vertices, when (x_j, x_k) and (x_k, x_j) with $j \neq k$ are identified, and when (x_j, x_j) are disregarded. It thus follows that in $\mathcal{G}(n,p)$, any graph G with m edges has a probability to appear given by

$$\mathbb{P}(G) = p^m (1-p)^{\binom{n}{2}-m}. \tag{10.1}$$

Indeed, it is necessary that m edges are present and $\binom{n}{2} - m$ edges are absent. Note that many of these graphs could be isomorphic, as introduced in Definition 2.8. On the other hand, if vertices are considered with labels, or equivalently endowed with different weights, then none of these graphs are isomorphic as labeled or weighted graphs.

One information which can be directly deduced from (10.1) is the distribution of graphs having m edges, namely:

$$\mathbb{P}(m) = \binom{\binom{n}{2}}{m} p^m (1-p)^{\binom{n}{2}-m}. \tag{10.2}$$

For information, this function $m \to \mathbb{P}(m)$ corresponds to the binomial distribution $B\big(\binom{n}{2}, p\big)$. A representation of $B(n,p)$ is provided in Figure 10.2.

[2]There exists a variant of this model, denoted by $\mathcal{G}(n,m)$ where the number m of edges is fixed. We shall not consider it here, see [33].

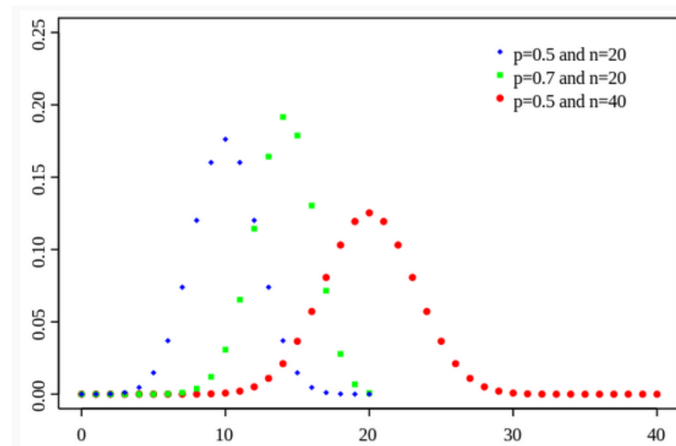


Fig. 10.2. The Binomial distributions $B(n,p)$

Let us now compute some quantities related to the family of graphs $\mathcal{G}(n,p)$. In that respect, we should not think anymore about the realization of a single graph, but about generic properties of all graphs in $\mathcal{G}(n,p)$, which means all graphs of n vertices, with an arbitrary number m of edges. The only condition for each edge is its probability p of existence. As an example of such quantity, what is the average number of edges among all these graphs ? This average will be denoted $\mathbb{E}(m)$ and can be easily computed. Indeed, for a single pair of vertices, this average is p, which implies that it is $\binom{n}{2}p$ for any graph (recall that each graph contains $\binom{n}{2}$ distinct pair of vertices). As a consequence,

$$\mathbb{E}(m) = \binom{n}{2} p = \frac{1}{2}n(n-1)p. \tag{10.3}$$

From this number, one can directly deduce the average degree (or mean degree) for each vertex. Indeed, this mean degree is given by $2\mathbb{E}(m)/n$, the factor 2 coming from the fact that each edge has two endpoints. In summary, for any vertex x of the graph,

$$c := \mathbb{E}\big(\deg(x)\big) = 2\frac{1}{2}n(n-1)p/n = (n-1)p. \tag{10.4}$$

Note that we shall simply denote this quantity by c, and that it will play an important role in the sequel.

Remark 10.1. *By looking carefully at* (10.4) *one observes that a fixed p can not always be a good idea. Indeed, if p is fixed and if we consider a sequence of graph in $\mathcal{G}(n,p)$ with n growing, then the average number of vertices at each vertex will grow with n. This is not a really natural situation, and a constant c is much more preferable. For a constant average degree c it is thus necessary that $p = \frac{c}{n-1}$, or roughly $p = \frac{c}{n}$. Note however that one could be more general by considering p (or c) having a more complicated dependence on n, and one is naturally led to the notion of* sparse *or* dense *graphs.*

The above information is an average over all individual degrees. It is also interesting to know the distribution of these degrees, namely the ratio of the number of vertices

having a degree k among all possible degrees. For any vertex x, the probability that this vertex is connected to k other vertices and not to the other $n-k-1$ remaining ones is given by $p^k(1-p)^{n-p-1}$. In addition, since there are $\binom{n-1}{k}$ ways to choose these k vertices, one obtains the probability that $\deg(x)=k$:

$$\mathbb{P}\big(\deg(x)=k\big) = \binom{n-1}{k} p^k (1-p)^{n-k-1}. \tag{10.5}$$

Observe that $\deg(x)$ follows also a binomial distribution, see Figure 10.2.

It is natural to wonder if a simpler expression for $\mathbb{P}\big(\deg(x)=k\big)$ can be obtained in the limit $n\to\infty$. This is indeed possible if the average degree c defined in (10.4) remains constant, see Remark 10.1. Indeed, if we set $p=\frac{c}{(n-1)}$, it then turns out that

$$\mathbb{P}\big(\deg(x)=k\big) \quad \overset{n\to\infty}{\longrightarrow} \quad \mathrm{e}^{-c}\frac{c^k}{k!}. \tag{10.6}$$

We shall not prove it but the main idea is the following. By setting $p=\frac{c}{(n-1)}$ on the r.h.s. of (10.5) and by considering some approximations for n large, then ne infers the above expression as $n\to\infty$. It should be mentioned that this expression corresponds to a Poisson distribution, and this is why this model is also called the Poisson random graphs: it refers to the Poisson distribution taken by the degree function, see also Figure 10.3.

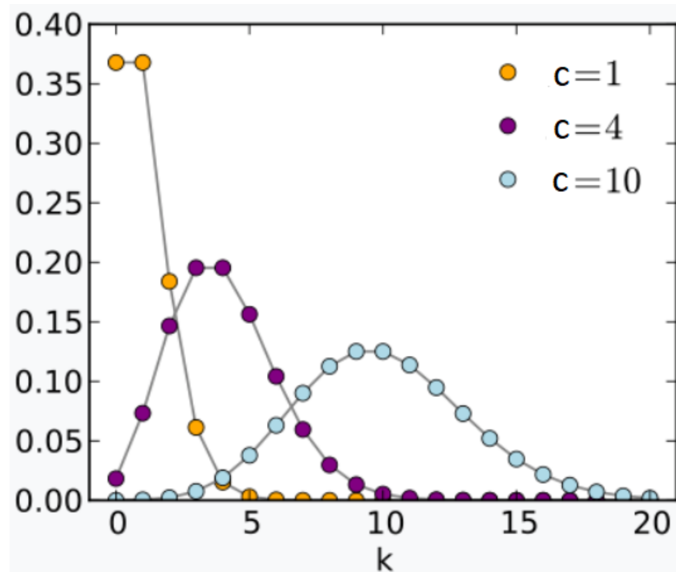


Fig. 10.3. The Poisson distribution, for different value of c.

Let us provide two additional results which can be proved with similar arguments. Proofs are given in [Die, Sec. 11.1]. For the first one we recall that the notion of independent vertices has been introduced in Definition 2.17 and corresponds to vertices which are not related by any edge. In particular, The independence number $\alpha(G)$ of a graph G corresponds to the number of vertices of a largest independent set in G.

Proposition 10.2. *For any integers $n \geq k \geq 2$ and for any $G \in \mathcal{G}(n,p)$, the probability that $\alpha(G) \geq k$ is upper estimated by*

$$\mathbb{P}\big(\alpha(G) \geq k\big) \leq \binom{n}{k} (1-p)^{\binom{k}{2}}.$$

The next result is about k-cycles, namely closed paths of length k. How many of them can one expect in any $G \in \mathcal{G}(n,p)$? The next statement is about this expectation, or in other words about the average number of k-cycles in G. For the statement, we introduce the *Pochhammer symbols* or *Pochhammer functions*: for $x \in \mathbb{R}$ and $n \in \mathbb{N}$ one sets

$$(x)_n = x(x-1)(x-2)\ldots(x-n+1) \tag{10.7}$$

with the convention that $(x)_0 = 1$.

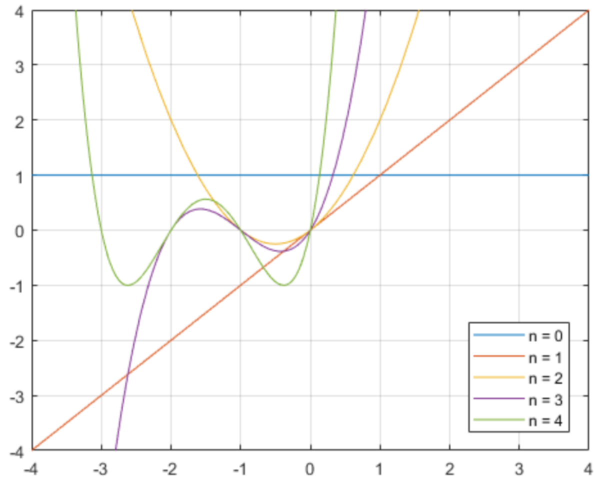


Fig. 10.4. Pochhammer's functions $(x)_n$

Proposition 10.3. *For any integers $n \geq k \geq 3$ one has*

$$\mathbb{E}(k\text{-}cycles) = \frac{(n)_k}{2k} p^k.$$

10.2 Components

So far in these notes, most graphs were considered connected, and if not the analysis was performed independently on each connected component. The notion of connectivity was also discussed in Chapter 5. For random graphs the existence of one or more connected components takes a different interest, and this is related to the average degree c. Two extreme situations are presented in Figure 10.5 which consists in a totally disconnected graph of 16 vertices, and the graph K_{16}. The first one is obtained for $p = 0$ while the second one is obtained for $p = 1$.

One central question is about the existence of a giant component. This notion is not defined for a single graph but for a family of graphs whose number of vertices is going to infinity. Recall from Definition 1.3 that the number of vertices of a graph G corresponds to its order and is denoted by $|G|$. A working definition of such a giant component is provided in:

Definition 10.4 (Giant component)**.** *A family of graphs $\{G_n\}_{n\in\mathbb{N}}$, satisfying $|G_n| \to \infty$ as $n \to \infty$, possesses a* giant component *if there exist some connected subgraphs $\Lambda_n \subset G_n$ and $\epsilon > 0$ with $|\Lambda_n| > \epsilon |G_n|$ for all $n \in \mathbb{N}$.*

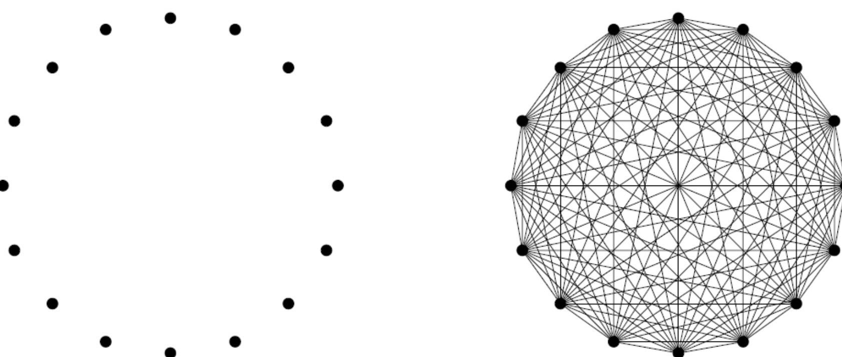


Fig. 10.5. One graph with 16 components, and K_{16}

By looking at the two examples provided in Figure 10.5 one guesses that the existence of a giant component is linked to the parameter c. Indeed, in the special case $c = 0$ (which corresponds to $p = 0$), no giant component exists, while in the case $c = n - 1$ (which corresponds to $p = 1$) it is clear that a giant component exists, since all graphs K_n are connected. What about $c \in (0, n-1)$? Our next aim is to answer this question.

In the setting of the previous definition, and if one assumes that there exists a giant component, let us set u for the probability that a vertex does not belong to the giant component. For simplicity, we assume that this probability is independent of n, which is correct if $|\Lambda_n|/|G_n|$ is a constant independent of n. Consider now one vertex x of G_n which is not in the giant component, and let y be any other vertex of G_n. The relation between x and y is either they are not connected (which takes place with probability $1-p$) or they are connected (with a probability p) but it is then necessary that y is not in the giant component (with happens with the probability u), since otherwise x would also belong to it. Thus, the probability for x not to be in the giant component through y is given by $(1-p) + pu$[3]. Since y can be any of the $n - 1$ vertices of the graph, it follows that the probability u of not being in the giant component satisfies the relation

$$u = \big((1-p) + pu\big)^{n-1} = \Big(1 - \frac{c(1-u)}{n-1}\Big)^{n-1}, \tag{10.8}$$

where the average degree c defined in (10.4) has been introduced. By assuming c fixed, as explained in Remark 10.1, and by considering the limit $n \to \infty$, one gets from (10.8) that $u = \mathrm{e}^{-c(1-u)}$, or alternatively if one sets $S = 1 - u$

$$S = 1 - \mathrm{e}^{-cS}. \tag{10.9}$$

Equation (10.9) can not be solved explicitly, but its solutions can be easily visualized, see Figure 10.6. Indeed, one can plot the function $S \mapsto 1 - \mathrm{e}^{-cS}$ for various values of c, and also the function $S \mapsto S$. The intersections of these curves give the solutions

[3] We use here the fact that the probability of a union of two mutually exclusive events is given by the sum of their probability.

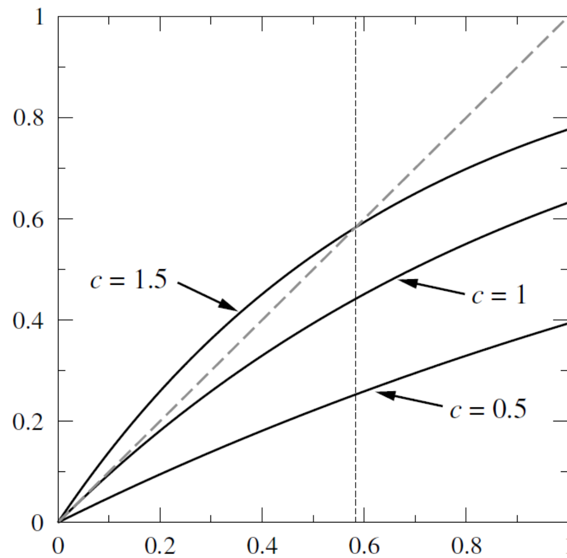


Fig. 10.6. Function $S \mapsto 1 - \mathrm{e}^{-cS}$

of (10.9), and the value $S = 0$ is always a solution. If $c = 1$, the two curves are tangent at $S = 0$, which implies that only one solution exists, and the same happens for $c < 1$. On the other hand, for if $c > 1$ a second solution always exists, even if its explicit expression can not be obtained. Numerically, this second solution can easily be obtained, as a function of c, and the graph of this second solution as a function of c is reported in Figure 10.7. Clearly, this second solution is monotically increasing from the value 0 to the value 1 as c goes from 0 to ∞. Observe finally that since u was the probability of not being in the giant component, the variable S corresponds to the probability of being in the giant component. As visible in Figure 10.7, this probability is 0 as long as $c \leq 1$ and then is strictly positive. Thus, a giant component can exist whenever $c > 1$, and in this case its relative size is given by the function $c \mapsto S$. For this model, the value $c = 1$ corresponds to a *phase transition*, since a giant component does not exist for $c \leq 1$ while it exists for $c > 1$.

By a separate argument, as presented in [Ne, Sec. 11.5.1] one can show that if a giant component exists, then it is unique. In other words, it is not possible in this model that two giant components coexist, although it is not prohibited by the definition of a giant component. The argument goes roughly as follows: if they were two giant components, the probability that they would not be connected can be computed and is exponentially small. In the limit $n \to \infty$, this exponentially small term vanishes, and therefore the probability that the two components are not connected is 0.

Now, what about the *small components* ? Indeed, the giant component does not cover the entire graph, as a consequence that $S < 1$. What can one say about the remaining parts of the graph ? Since the size of the small components grow with a rate smaller then $|G_n|$, their number has to increase as $n \to \infty$. For this model, the behavior of the small components is well understood and can be studied analytically.

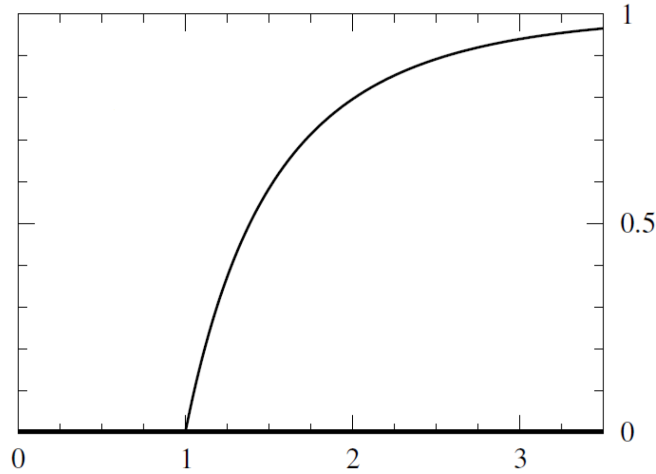


Fig. 10.7. The solutions of (10.9) as a function of c

We only summarize the outcomes, and refer to [Ne, Sec. 11.5] for more information and additional references.

Let us consider a small component G_s of fixed $s \in \mathbb{N}$ vertices. The first observation is that this component will be a tree with probability 1 as $n \to \infty$. The argument is the following: first of all, observe that a tree contains $s-1$ edges, which is the minimal number of edges for any connected graph of s vertices. Then, there exists $\binom{s}{2} - (s-1) = \frac{1}{2}(s-1)(s-2)$ possibilities for adding one edge in this graph, and the probability of adding one edge is $p = \frac{c}{n-1}$. Thus, the average total number of additional edges to this component is

$$\frac{1}{2}(s-1)(s-2)\frac{c}{n-1} = \frac{c(s-1)(s-2)}{2(n-1)}$$

and this number goes to 0 as $n \to \infty$. As a consequence, each small component is likely to be a tree in the limit $n \to \infty$.

Let us now denote by π_s the probability that a randomly chosen vertex belongs to a small component of size s. Clearly, $\sum_{s\in\mathbb{N}^*} \pi_s = 1 - S$, which is the probability of not being in the giant component, if this one exists. It is possible but rather long to deduce the explicit expression for π_s, we only provide the final result in the limit $n \to \infty$, namely

$$\pi_s = \frac{\mathrm{e}^{-sc}(sc)^{s-1}}{s!}.$$

It is also possible to get the average size of the small component to which a randomly chosen vertex belongs. More precisely one has

$$\mathbb{E}(s) := \frac{\sum_{s\in\mathbb{N}^*} s\pi_s}{\sum_{s\in\mathbb{N}^*} \pi_s} = \frac{\sum_{s\in\mathbb{N}^*} s\pi_s}{1-S} = \frac{1}{1-c+cS}, \tag{10.10}$$

where S is the value provided by Figure 10.7. A representation of $\mathbb{E}(s)$ as a function of c is provided in Figure 10.8. There is clearly a singularity as $c = 1$. In fact, when one

approaches $c = 1$ either from the left or from the right, the small components tend to become bigger and bigger. For understanding what happens precisely at $c = 1$, further refined analysis is needed.

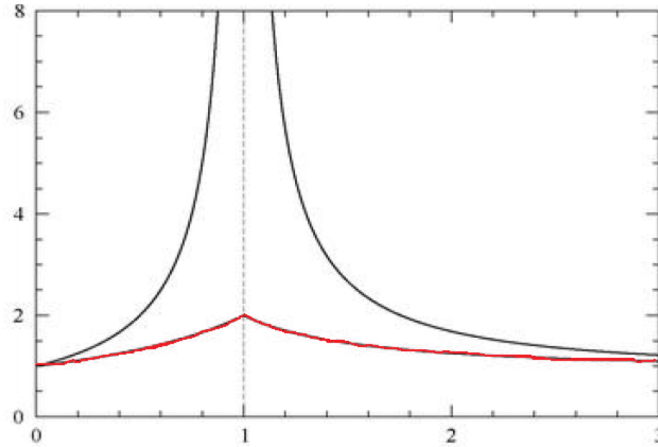


Fig. 10.8. In black, the function $c \mapsto \mathbb{E}(s)$ given in (10.10); in red, the function $c \mapsto \mathbb{E}(|G_s|)$ given in (10.11)

Let us finish this section with one tricky but interesting point. Does (10.10) give us the mean size $\mathbb{E}(|G_s|)$ of the small components ? The answer is no because the computation is biased. Indeed, recall that π_s is the probability that a randomly selected vertex is in a component of size s. It is therefore not the probability of existence of a component of size s (which leads to the average size of the small components). Thus, in order to correct the computation, let us observe that if n_s denotes the number of components of size s, then the following equality holds:

$$\pi_s = \frac{sn_s}{n}.$$

For the computation of the mean size $\mathbb{E}(|G_s|)$ of the small components one has

$$\mathbb{E}(|G_s|) := \frac{\sum_{s\in\mathbb{N}^*} sn_s}{\sum_{s\in\mathbb{N}^*} n_s} = \frac{n\sum_{s\in\mathbb{N}^*} \pi_s}{n\sum_{s\in\mathbb{N}^*} \frac{\pi_s}{s}} = \frac{1-S}{\sum_{s\in\mathbb{N}^*} \frac{\pi_s}{s}} = \frac{2}{2-c+cS}. \tag{10.11}$$

It is interesting to see that this function does not possess a singularity at $c = 1$, as shown in Figure 10.8.

10.3 Clustering coefficient and path lengths

Let us start by introducing the *clustering coefficient* which plays an important role in the analysis of real graphs. This coefficient is a measure of the degree to which nodes in a graph tend to cluster together. It is also related to the notion of transitivity which has been introduced in Definition 8.5. More precisely, the clustering coefficient is going

to estimate the lack of transitivity of a graph. Two versions of this measure exist: the global one and the local one. The global version was designed to give an overall indication of the clustering in the network, whereas the local gives an indication of the embeddedness of single nodes.

For the following definition, we call *triplet* three vertices which are connected either by 2 edges (*open triplet*) or by 3 edges (*closed triplet*). Since the graphs considered are simple, it means that open triplets consist in one vertex connected to two distinct vertices, while closed triplets correspond to 3-cycles, also called *triangles*.

Definition 10.5 (Global clustering coefficient)**.** *For a finite simple graph, its* global clustering coefficient C *is defined by*

$$C := \frac{\sharp(\textit{closed triplets})}{\sharp(\textit{triplets})}.\,[4] \tag{10.12}$$

For example, if a graph is transitive, then $C = 1$. On the other hands, trees have a global clustering coefficient equal to 0 since they do not possess a single closed triplet. For the local clustering coefficient, recall from Definition 1.4 that the neighbours $N(x)$ of x consist in the vertices connected to x. Clearly, the cardinality of $N(x)$ is equal to the degree $\deg(x)$ of x.

Definition 10.6 (Local clustering coefficient)**.** *For a finite simple graph* $G = (V,E)$ *and for any* $x \in V$*, the* local clustering coefficient C_x *is defined by*

$$C_x := 2\frac{\big|\big\{(y,z) \in E\big\} \mid y,z \in N(x)\big\}\big|}{\deg(x)\big(\deg(x)-1\big)}. \tag{10.13}$$

More explicitly, the numerator consists in the number of edges with both endpoints in $N(x)$ while $\frac{1}{2}\deg(x)\big(\deg(x)-1\big)$ consists in the total number of possible edges with two endpoints in $N(x)$. This coefficient measures again the lack of transitivity around x, since it considers if the two vertices y and z, which are connected to x, are also connected to each others. Based on this local notion, it is also possible to compute its average, namely

$$\overline{C} := \frac{1}{|G|}\sum_{x\in G} C_x$$

but in general this value is not equal to the global clustering coefficient.

What about these concepts applied to the random model $\mathcal{G}(n,p)$? Since in this model the probability of any two vertices to be connected is always the same, namely $p = \frac{c}{(n-1)}$, and since this probability is independent of all the other existing or missing edges one infers that

$$\mathbb{E}(C) = p = \frac{c}{n-1}, \qquad \mathbb{E}(C_x) = p = \frac{c}{n-1} \tag{10.14}$$

[4] Note that on has to be consistent when computing these numbers: if the triplet (x,y,z) is considered different from the triplet (y,z,x), then this rule should be taken into account both for the numerator and for the denominator. For that reason, slightly different formulations also appear in the litterature: for example it is sometimes written $C = \frac{3\sharp(\text{triangles})}{\sharp(\text{triplets})}$.

which vanish in the limit $n \to \infty$. We shall see later on that this constant value of the clustering coefficients is one of the factors which make random graphs quite different from real-world graphs.

There is a second quantity which provides some important information about a graph, namely the so-called *length path*. In fact, this name is slightly misleading since the correct notion is the diameter introduced in Definition 1.15. More precisely, one looks for the shortest path between two arbitrary vertices, and if one considers the longest such path, it is precisely the diameter of the graph. Note that one could also consider the mean of the distance between two arbitrary vertices.

The computation of the diameter of random graphs is heuristically very simple, but precisely rather delicate, and there exist several research papers dealing only with this question. Here, we shall consider only the heuristic argument, and refer [Ne, Sec. 11.7] and references therein for more precise considerations. Note that a precise computation has to take into account the precise dependence on n of the average degree c.

The heuristic argument goes as follows: starting from an arbitrary vertex x, c vertices are at a distance 1, $c(c-1)$ new vertices are at a distance 2, $c(c-1)^3$ new vertices are at a distance 3, and $c(c-1)^{j-1}$new vertices are at a distance j. It means that at a distance at most j one can reach

$$c + c(c-1) + c(c-1)^2 + \cdots + c(c-1)^j - 1 = c\sum_{i=0}^{j-1}(c-1)^i = (c-1)^j - 1 \cong c^j$$

new vertices, if we assume that c is large enough. Of course, this estimate is really a rough estimate since for a fixed i there may exist several paths of length i between x and a given y. This observation means that in the above summation, there are several redundancies. We should clearly think about this estimate as a first order approximation. Thus, whenever $c^j = n$, with n the number of vertices of the graph, it means that we have "roughly" been able to visit all vertices of the graph. Based on this equality, one deduces an estimate on the diameter of the graph, namely

$$c^j = n \Leftrightarrow j\ln(c) = \ln(n) \Leftrightarrow j = \frac{\ln(n)}{\ln(c)}.$$

In fact, a more precise approach leads to the estimate

$$\mathbb{E}\big(\mathrm{diam}(G)\big) = A + \frac{\ln(n)}{\ln(c)} \tag{10.15}$$

for some constant A which can be computed. As for the clustering coefficient, this result will have to be compared with the diameter of real-world graphs.

10.4 Weaknesses

The random graph model $\mathcal{G}(n,p)$ has been one of the first ones introduced, and is considered as a simple model. However, when compared to real networks or real-world

	Network	Type	n	m	c	S	ℓ	α	C
Social	Film actors	Undirected	449 913	25 516 482	113.43	0.980	3.48	2.3	0.20
	Company directors	Undirected	7 673	55 392	14.44	0.876	4.60	–	0.59
	Math coauthorship	Undirected	253 339	496 489	3.92	0.822	7.57	–	0.15
	Physics coauthorship	Undirected	52 909	245 300	9.27	0.838	6.19	–	0.45
	Biology coauthorship	Undirected	1 520 251	11 803 064	15.53	0.918	4.92	–	0.088
	Telephone call graph	Undirected	47 000 000	80 000 000	3.16			2.1	
	Email messages	Directed	59 812	86 300	1.44	0.952	4.95	1.5/2.0	
	Email address books	Directed	16 881	57 029	3.38	0.590	5.22	–	0.17
	Student dating	Undirected	573	477	1.66	0.503	16.01	–	0.005
	Sexual contacts	Undirected	2 810					3.2	
Information	WWW `nd.edu`	Directed	269 504	1 497 135	5.55	1.000	11.27	2.1/2.4	0.11
	WWW AltaVista	Directed	203 549 046	1 466 000 000	7.20	0.914	16.18	2.1/2.7	
	Citation network	Directed	783 339	6 716 198	8.57			3.0/–	
	Roget's Thesaurus	Directed	1 022	5 103	4.99	0.977	4.87	–	0.13
	Word co-occurrence	Undirected	460 902	16 100 000	66.96	1.000		2.7	
Technological	Internet	Undirected	10 697	31 992	5.98	1.000	3.31	2.5	0.035
	Power grid	Undirected	4 941	6 594	2.67	1.000	18.99	–	0.10
	Train routes	Undirected	587	19 603	66.79	1.000	2.16	–	
	Software packages	Directed	1 439	1 723	1.20	0.998	2.42	1.6/1.4	0.070
	Software classes	Directed	1 376	2 213	1.61	1.000	5.40	–	0.033
	Electronic circuits	Undirected	24 097	53 248	4.34	1.000	11.05	3.0	0.010
	Peer-to-peer network	Undirected	880	1 296	1.47	0.805	4.28	2.1	0.012
Biological	Metabolic network	Undirected	765	3 686	9.64	0.996	2.56	2.2	0.090
	Protein interactions	Undirected	2 115	2 240	2.12	0.689	6.80	2.4	0.072
	Marine food web	Directed	134	598	4.46	1.000	2.05	–	0.16
	Freshwater food web	Directed	92	997	10.84	1.000	1.90	–	0.20
	Neural network	Directed	307	2 359	7.68	0.967	3.97	–	0.18

Fig. 10.9. Basic properties of several networks, from Table 10.1 of [Ne]

graphs, some of its properties do not match with the observations. There exist numerous analysis of real and social networks, and we shall not go in this direction. However, let us just mention a few weaknesses of the Gilbert-Erdős-Rényi model, and refer to [Ne, Chap. 10] for more information.

A table containing several example of networks and their basic properties is provided in Figure 10.9. In this table, the following information are mentioned:

(i) The type of the graph: directed or undirected,

(ii) The total number of vertices n,

(iii) The total number of edges m,

(iv) The mean degree c,

(v) The probability S of being in the giant component,

(vi) The mean distance ℓ between connected vertices,

(vii) The exponent α of the power law distribution of the degrees, if it follows a power law as introduced in (10.16),

(viii) The clustering coefficient, as defined in (10.12).

One important difference between the content of this table and the outcome of the random model $\mathcal{G}(n,p)$ is the clustering coefficient. Indeed, all values for C is this table range between 0.005 and 0.2, with a majority close to 0.1. On the other hand, the result obtained in (10.14) for $\mathcal{G}(n,p)$ reads $C = \frac{c}{n-1}$, which leads to values which are much smaller than the ones measured.

Another striking difference is about the degree distribution of vertices. It was shown in (10.6) that the distribution of degrees in the $\mathcal{G}(n,p)$ model follows a Poisson distribution, in the limit $n \to \infty$. On the other hand, several real networks present a power law distribution for $\deg(x)$. A typical power law distribution over $\mathbb{R}_+$ is shown in Figure 10.10, and it corresponds to the graph of a function of the form $x \mapsto x^{-\alpha}$. Note that some regularization near 0 should be considered, or one should only consider $x \geq x_{\min} > 0$. The discrete version of a power law reads

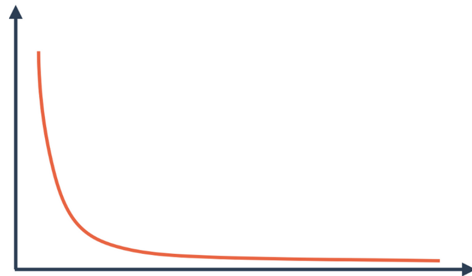


Fig. 10.10. Power law distribution

$$\mathbb{N}^* \ni k \mapsto k^{-\alpha} \in \mathbb{R}_+. \tag{10.16}$$

The coefficient α provided in Figure 10.9 corresponds to the exponent of this function, whenever the degrees follow such a lower law. The clear difference between the Poisson distribution provided by the $\mathcal{G}(n,p)$ model, and the degree distribution of a network like internet is presented in Figure 10.11, despite the fact that these two distributions have the same average.

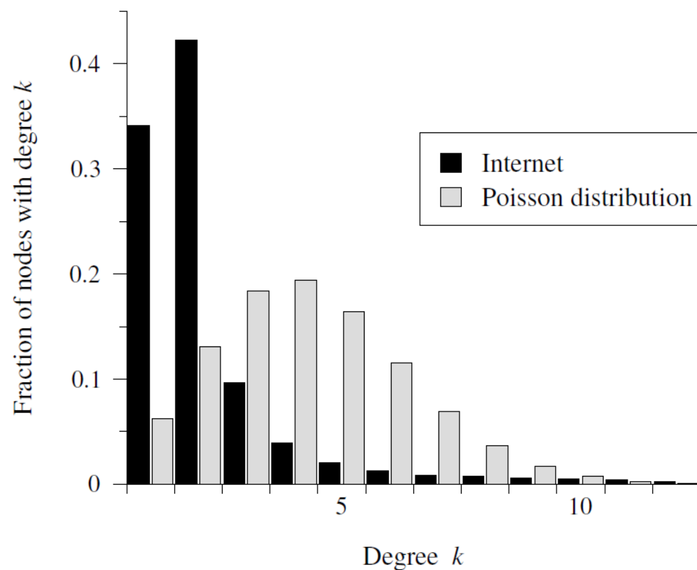


Fig. 10.11. Comparison of two degree distributions, from Figure 11.8 of [Ne]

Chapter 11

The configuration model

In this chapter we introduce another model of random graphs which is one of the most important theoretical models for the study of networks. It is often used as a first model before turning to more specialized ones. Let us mention that we shall consider only undirected graphs, but alternative solutions for directed graphs exist. One of the important feature of the configuration model is that the degree distribution can be easily tuned. The notion of modularity is also introduced in the last section of this chapter.

11.1 Construction, and basic properties

In the configuration model, the degree of each vertex is pre-defined, which means that any degree distribution can be implemented. In other words and as opposed to the $\mathcal{G}(n,p)$ model, the degree distribution is not restricted to have a Poisson-distribution, the model allows the user to give the network any desired degree distribution.

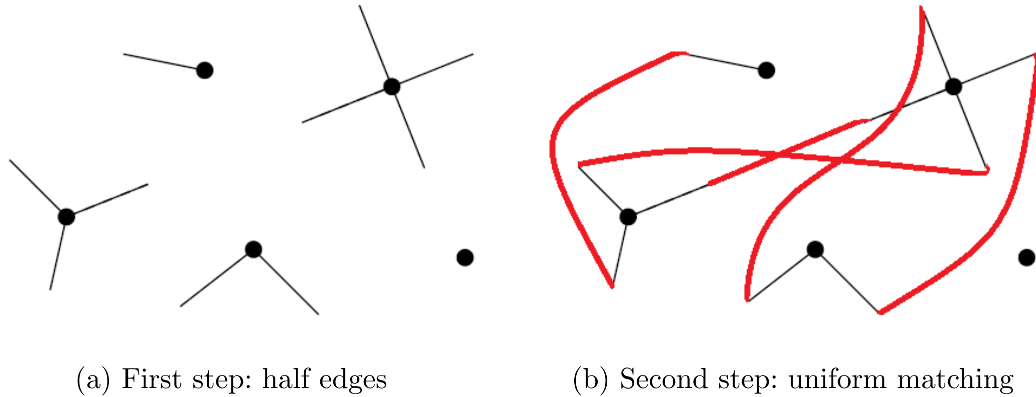
Let us start by dealing with a sequence of positive integers. For a fixed $n \in \mathbb{N}$, consider a sequence $\{k_i\}_{i=1}^n$ with $k_i \in \mathbb{N}$ and with the property that $\sum_{i=1}^n k_i = 2m$ for some $m \in \mathbb{N}$. In other words, the positive integers k_i sum up to an even number. We call such a sequence $\{k_i\}_{i=1}^n$ a *nm-degree sequence*. Note that the number n will correspond to the number of vertices of the future graph, the number m will correspond to its number of edges, while the numbers k_i will correspond to the degree of the vertex x_i.

For any nm-degree sequence, let us also define n_k as the number of elements in the sequence satisfying $k_i = k$. In other words, n_k is going to be the number of vertices with degree k. Clearly, $\sum_{k\in\mathbb{N}} n_k = n$. Let us set $p_k := \frac{n_k}{n}$. The distribution $\{p_k\}_{k\in\mathbb{N}}$ is going to provide the degree distribution of the future graph. Thus, by first choosing a distribution $\{p_k\}$, and then a suitable nm-degree sequence having this degree distribution, one can construct graphs with any prescribed degree distribution.

Let us now concentrate on the construction of the graph, with the notation introduced in the previous paragraphs. The graph is constructed for any given nm-degree sequence:

(i) For any $i \in \{1, \dots, n\}$ endow x_i with k_i half edges, also called *stubs*, see Figure 11.1a,

(ii) Connect the half edges uniformly at random by creating proper edges, see Figure 11.1b.



(a) First step: half edges (b) Second step: uniform matching

Fig. 11.1. Construction of the configuration model

With this construction, it is clear that one ends up with a graph containing n vertices and m edges. Note that multiple edges can appear, and loops as well. It is also clear why the condition $\sum_{i=1}^{n} k_i$ is even is necessary: if not, one half edge would not match with any other. The resulting graph has a prescribed degree distribution, given by the distribution $\{p_k\}_{k\in\mathbb{N}}$ introduced above. On the other hand, any half edge is equally likely to be connected to any other half edge.

There are two standard choices for the distribution $\{p_k\}_{k\in\mathbb{N}}$: a power law distribution or a Poisson distribution. In the latter case, observe that the resulting graph is very close to a graph obtained by the $\mathcal{G}(n, p)$ model. However, graphs obtained by the configuration model can have multiple edges and loops, while no such graphs exist in the $\mathcal{G}(n, p)$ approach. On the other hand, since many real-world graphs are observed to have a power law degree distribution, the configuration model with power law degree distribution allows us to study them theoretically and to understand some of their properties.

Let us now study some simple properties of the configuration model. For the time being we do not impose any condition on the nm-degree sequence $\{k_i\}_{i=1}^{n}$, or equivalently on the distribution $\{p_k\}_{k\in\mathbb{N}}$. Note that we shall always consider large n and m, or even the limits $n \to \infty$ and $m \to \infty$.

First of all, given two vertices x_i and x_j, what is the probability of having an edge from x_i to x_j ? One stub at x_i can be connected to $2m - 1$ other stubs, among them k_j belong to x_j. Thus the probability of this stub to be connected to x_j is $\frac{k_j}{2m-1}$. If x_i has k_i stubs, then the probability that x_i is connected to x_j is $\frac{k_i k_j}{2m-1}$. For m large enough, one usually sets: the probability p_{ij} of having an edge between x_i and x_j is given by

$$p_{ij} = \frac{k_i k_j}{2m}. \tag{11.1}$$

Based on this expression, one can easily compute the probability of having at least two edges between x_i and x_j, namely $\frac{k_i k_k}{2m}\frac{(k_i-1)(k_j-1)}{2m} = \frac{k_i k_j (k_i-1)(k_j-1)}{(2m)^2}$. Note that the factors $k_i - 1$ and $k_j - 1$ are due to the fact that one stub at x_i and one stub at x_j are already used by the first edge. What is now the expected number of multiple edges in the graph ? One has to sum over all indices i and j, and divide by two in order to avoid counting twice every pairs of vertices. Thus, by using the notations

$$\langle k\rangle = \frac{1}{n}\sum_i k_i \qquad \text{and} \qquad \langle k^2\rangle = \frac{1}{n}\sum_i k_i^2. \tag{11.2}$$

one gets

$$\begin{aligned}\frac{1}{2(2m)^2}\sum_{i,j} k_i k_j (k_i-1)(k_j-1) &= \frac{1}{2n^2\langle k\rangle^2}\Big(\sum_i k_i(k_i-1)\Big)\Big(\sum_j k_j(k_j-1)\Big) \\ &= \frac{1}{2}\left(\frac{\langle k^2\rangle - \langle k\rangle}{\langle k\rangle}\right)^2. \end{aligned} \tag{11.3}$$

Note that we have used the equality $2m = n\langle k\rangle$ coming from the equality $\sum_i k_i = 2m$. The expression $\langle k\rangle$ and $\langle k^2\rangle$ are called the first and second moments of the degree distribution[1]. Indeed the following equalities hold:

$$\langle k\rangle = \frac{1}{n}\sum_i k_i = \frac{1}{n}\sum_k n_k k = \sum_k \frac{n_k}{n} k = \sum_k k p_k = \mathbb{E}(k)$$

and

$$\langle k^2\rangle = \frac{1}{n}\sum_i k_i^2 = \frac{1}{n}\sum_k n_k k^2 = \sum_k k^2 p_k = \mathbb{E}(k^2).$$

Observe that (11.3) does not depend on n, which means that the expected number of multiple edges remains constant as the graph grows (when $n \to \infty$). Accordingly, the density of multiple edges by vertex goes like $\frac{1}{n}$ which means that multiple edges are rare in the configuration model, when n is large enough.

A similar derivation for loops holds: one easily find that the probability of a loop at x_i is given for m large enough by[2]

$$p_{ii} = \frac{k_i(k_i-1)}{2(2m)}.$$

[1] If these quantities are computed with a power law distribution one should be a little bit more careful. Indeed, as long as these quantities are computed for a finite degree sequence, all sums are finite, but in the limit $n \to \infty$ these sums might contain an infinite number of contributions, and accordingly their convergence might not hold. In fact, one easily observes that if $p_k = k^{-\alpha}$ for all $k \in \mathbb{N}^*$, then only a finite number of moments exist, if any, and this number depends on α. As a rule, whenever we write $\langle k^s\rangle$ for some $s > 0$ we assume that this quantity exists and is not infinite.

[2] The factor 2 is due to the factor $\binom{k_i}{2}$ for the possible choices of the two stubs.

The expected number of loops in the graph is then given by

$$\sum_i p_{ii} = \sum_i \frac{k_i(k_i-1)}{2(2m)} = \frac{\langle k^2\rangle - \langle k\rangle}{2\langle k\rangle},$$

which means that the density of loops by vertex goes again like $\frac{1}{n}$. In the limit $n \to \infty$, loops become very rare in the configuration model.

Let us end this section with one more information which can be computed similarly. Given the edges x_i and x_j what it the expectation n_{ij} of having a common neighbours x_l (which means that x_l is connected to x_i and to x_j). By one minute of deep thought one infers that

$$n_{ij} = \sum_l \frac{k_i k_l}{2m}\frac{k_j(k_l-1)}{2m} = \frac{k_i k_j}{2m}\sum_l \frac{k_l(k_l-1)}{n\langle k\rangle} = p_{ij}\frac{\langle k^2\rangle - \langle k\rangle}{\langle k\rangle}.$$

Thus, this expected value depends only on p_{ij} and on some factors due only to the degree distribution.

11.2 Additional properties

In this section we study a few additional properties of the configuration model, and one of them turns out to be rather surprising. As mentioned in the previous section, the model is either described by a nm-degree sequence, or by a degree distribution $\{p_k\}_{k\in\mathbb{N}}$. We opt for the latter setting. In this context, p_k is the probability that a vertex chosen uniformly at random has a degree k.

Suppose firstly that we start at one vertex, and follow one of its edges. What is the probability that the second endpoint of this edge has a degree k ? The naive answer p_k can not be correct, since for example one would never be able to reach a vertex with degree 0, while such vertices exist with probability p_0. Less naively, our current edge can end at any of the $2m-1$ other stubs, among which knp_k belong to vertices with degree k. Thus, the probability of ending at a vertex of degree k for m large is given by

$$\frac{knp_k}{2m} = \frac{kp_k}{\langle k\rangle}. \tag{11.4}$$

This result is in fact rather natural, since one has a bigger chance of reaching a vertex with degree k than a vertex of degree 1 (assuming that $k \geq 1$) even if $p_k = p_1$.

Let us infer a rather surprising result from (11.4). By starting again at an arbitrary vertex x, what is the average degree of its neighbours ? This quantity is obtained by averaging k over the probability of having degree k given by (11.4), namely

$$\mathbb{E}\big(\{\deg(y) \mid y \in N(x)\}\big) = \sum_k k\frac{kp_k}{\langle k\rangle} = \frac{\langle k^2\rangle}{\langle k\rangle}.$$

Thus, the average degree of the neighbours of x is different from the average degree in the graph, which is $\langle k\rangle \equiv \mathbb{E}(k)$! In addition, one observes that

$$\frac{\langle k^2\rangle}{\langle k\rangle} - \langle k\rangle = \frac{1}{\langle k\rangle}\big(\langle k^2\rangle - \langle k\rangle^2\big) = \frac{\sigma_k^2}{\langle k\rangle}$$

where $\sigma_k^2 = \langle k^2\rangle - \langle k\rangle^2 > 0$ corresponds to the variance of the degree distribution. In other words, the average degree of the neighbours of x is bigger than the average degree of an arbitrary vertex in the graph. In terms of a grumpy person ☹ : *your friends have more friends than you do.* Let us emphasize that this phenomenon is not a special feature of the configuration model, it can be measured on various networks, as shown in Figure 11.2. Note also that this surprising property can be fully explained. In short, any vertex with degree k will appear as neighbour of exactly k other vertices, and hence will appear in k averages. At the same time, all vertices with 0 edge won't play any role in this computation (simply because they are never reached), while they are counted in the computation of $\langle k\rangle$.

Network	n	Average degree	Average neighbor degree	$\frac{\langle k^2\rangle}{\langle k\rangle}$
Biologists	1 520 252	15.5	68.4	130.2
Mathematicians	253 339	3.9	9.5	13.2
Internet	22 963	4.2	224.3	261.5

Fig. 11.2. The degree of neighbours is always higher, from Sec. 12.2 of [Ne]

Still based on (11.4) let us introduce one more quantity, the *excess degree* of a vertex. Again, starting at x and arriving at a neighbour y, its excess degree is simply the number of its edges minus the one used for reaching it. Thus the probability that the excess degree is k simply given by

$$q_k = \frac{(k+1)p_{k+1}}{\langle k\rangle} \tag{11.5}$$

which is obtained by considering (11.4) for $k+1$. The distribution defined by $\{q_k\}_{k\in\mathbb{N}}$ is called *the excess degree distribution.* Its average can be easily computed, namely

$$\mathbb{E}(q) = \sum_k kq_k = \frac{1}{\langle k\rangle}\sum_k k(k+1)p_{k+1} = \frac{1}{\langle k\rangle}\sum_k k(k-1)p_k = \frac{\langle k^2\rangle - \langle k\rangle}{\langle k\rangle}.$$

Let us now turn our attention to the clustering property of the configuration model, as defined in Definition 10.5. Consider a vertex x with at least two edges, and let x_i and x_j the two distinct vertices connected to x. Their remaining number of edges is denoted by k_i and k_j respectively, and are distributed according to the excess degree distribution q_{k_i} and q_{k_j}. In addition, the probability that they share an edge is given

by (11.1), namely $\frac{k_ik_j}{2m}$. The clustering coefficient is obtained by summing all these contributions:

$$\begin{aligned}\mathbb{E}(C) &= \sum_{k_i,k_j} q_{k_i}q_{k_j}\frac{k_ik_j}{2m} = \frac{1}{2m}\Big(\sum_k kq_k\Big)^2 \\ &= \frac{1}{2m\langle k\rangle^2}\Big(\sum_{k=0}^{\infty} k(k+1)p_{k+1}\Big)^2 = \frac{1}{2m\langle k\rangle^2}\Big(\sum_{k=0}^{\infty} k(k-1)p_k\Big)^2 \\ &= \frac{1}{n}\frac{\big(\langle k^2\rangle - \langle k\rangle\big)^2}{\langle k\rangle^3}. \end{aligned} \tag{11.6}$$

This result can be compared to the one obtained in (10.14) for the random model $\mathcal{G}(n,p)$. Their similar feature is the decay in $\frac{1}{n}$, which does not really take place for real-world graphs. However, the factor $\langle k^2\rangle$ appearing in (11.6) can take very large values, depending on the degree distribution considered, and therefore lead to a clustering coefficient more in line with the observations.

What about a giant component and about the small components, as studied in Section 10.2 for the random model $\mathcal{G}(n,p)$. It turns out that a similar analysis can be performed for the configuration model, and that similar results hold. More precisely, it can be shown that a giant component exists if the following condition is satisfied:

$$\langle k^2\rangle - 2\langle k\rangle > 0. \tag{11.7}$$

This condition can be obtained from different approaches, and we refer to [Ne, Sec. 12.6] for the details. Let us just mention that one approach is to set u for the probability that a vertex does not belong to the giant component, and by a clever reasoning one infers that u has to satisfy the equation

$$u = g_1(u) \tag{11.8}$$

with

$$g_1(u) = \sum_k q_k u^k. \tag{11.9}$$

In general (it depends on the initial degree distribution $\{p_k\}$) this equation can not be solved explicitly, but some arguments leads to the condition (11.7).

If a giant component exists, then one can also look at the small components. As in the previous model, these small components are trees, in the limit of an infinite graph. As we already did in (10.10), let us provide the average size of the small component to which a randomly chosen vertex belongs. If s denotes the size of a small component to which a randomly chosen vertex belongs to, then for the configuration model one has

$$\mathbb{E}(s) = 1 + \frac{u^2\langle k\rangle}{g_0(u)\big(1-g_1'(u)\big)},$$

with $g_0(u) = \sum_k p_k u^k$. Clearly, this expression is complicated. However, if there is no giant component, namely when $u = 1$, it can be simplified. In this situation one gets

$$\mathbb{E}(s) = 1 + \frac{\langle k\rangle^2}{2\langle k\rangle - \langle k^2\rangle}.$$

Another quantity which has been mentioned for the $\mathcal{G}(n,p)$ model is the diameter of a graph containing n vertices. We only provide the result, and refer to [Ne, Sec. 12.9] for its precise computation. Note however that the approach is rather standard for any type of graphs, and often lead to similar results. For the current model of graphs with n vertices, it turns out that

$$\mathbb{E}\big(\mathrm{diam}(G)\big) = A + \frac{\ln(n)}{\ln\big(\frac{\langle k^2\rangle - \langle k\rangle}{\langle k\rangle}\big)} \tag{11.10}$$

for some constant A which can be computed. Note that this result is quite similar to the one already obtained in the previous model.

As already mentioned at the beginning of this chapter, one interest in the configuration model is the ability of choosing the degree distribution. For example, it is possible to implement that only a few p_k are not 0, meaning that one vertex can only have a prescribed number of edges attached to it. In such a case, the computations are usually quite simple, as shown in an explicit example provided in [Ne, Sec. 16.1]. Another example is to consider exact power law provided for $\alpha > 0$ by the formula

$$p_k = \frac{1}{\zeta(\alpha)}\begin{cases} 0 & \text{if } k = 0 \\ k^{-\alpha} & \text{if } k \geq 1 \end{cases},$$

where $\zeta(\alpha) = \sum_{k=1}^{\infty} k^{-\alpha}$ is the Riemann zeta function. In such a situation, some additional computations can be performed, as for example

$$\langle k\rangle = \sum_{k=0}^{\infty} k p_k = \frac{1}{\zeta(\alpha)}\sum_{k=1}^{\infty} k^{-\alpha+1} = \frac{\zeta(\alpha-1)}{\zeta(\alpha)},$$

or

$$\langle k^2\rangle = \sum_{k=0}^{\infty} k^2 p_k = \frac{1}{\zeta(\alpha)}\sum_{k=1}^{\infty} k^{-\alpha+2} = \frac{\zeta(\alpha-2)}{\zeta(\alpha)}.$$

With this explicit expression, condition (11.7) for the existence of a giant component reads $\zeta(\alpha-2) > 2\zeta(\alpha-1)$ which can be solved numerically: one gets this existence if $\alpha < 3.4788\ldots$.

As already mentioned in the footnote on page 181, the power law distribution suffers from the non-existence of most of its moments. Indeed, it is easily seen that $\sum_{k=1}^{\infty} k^s p_k = \frac{1}{\zeta(\alpha)}\sum_{k=1}^{\infty} k^{s-\alpha}$ exists if and only if $s < \alpha - 1$. Thus, the expression $\langle k\rangle$ is finite if $\alpha > 2$ and $\langle k^2\rangle$ is finite if $\alpha > 3$. Nevertheless, it is sometimes possible to cut the tail of this distribution (put $p_k = 0$ for k large enough) and get some meaningful result.

For example, one infers from this approach that (11.7) is always satisfied for $\alpha \in (2, 3]$. If $\alpha \in (3, 3.4788...)$ there is still a giant component, but not for larger α. Note that for $\alpha \leq 2$, a giant component also exists, but other tools are necessary for proving it.

Let us end this section with one key word: *generating functions*. These functions are very useful in probability and should have been introduced. Some results of this section can be easily obtained with them. However, these functions have to be properly introduced in a course on probability. Don't miss to attend such a course, very powerful techniques will then be available.

11.3 Community structure, or modularity

In this section, we introduce one more concept which is useful for arbitrary graphs. This notion, called *community structure* or *modularity*, provides one measure of the structure of a graph. It was designed to measure the strength of division of a network into modules (also called groups, clusters or communities). Networks with high modularity have dense connections between the vertices within modules but sparse connections between vertices in different modules. Modularity is often used in optimization methods for detecting community structure in networks. However, it has been shown that modularity suffers a resolution limit and, therefore, it is unable to detect small communities. A graph with two clear communities is presented in Figure 11.3. Note that there is a third community (grey dots) which is spread on the other two communities.

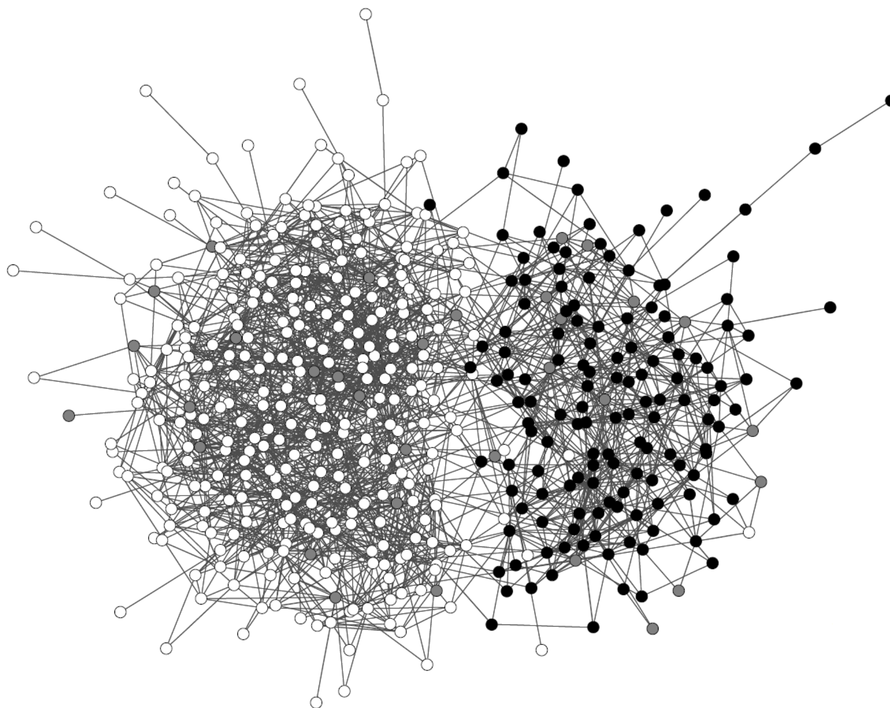


Fig. 11.3. A graph with 3 communities, from Sec. 7.7 of [Ne]

The relation between the search of community structure and the configuration model is the following: the number of edges inside a suspected community will be compared to the configuration model, in which edges are placed uniformly at random. In other words, the configuration model is used as a test model. If there exist more edges than the one provided by the configuration model, then the community really exist. But note that it might not be the tighter community (or the one with the biggest number of edges between its members). The computation is based on the adjacency matrix $A_G = \{a_{ij}\}$ introduced in Definition 2.1, and will also use the probability p_{ij} of having an edge between x_i and x_j computed in (11.1) for the configuration model.

We consider a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, both numbers being large but finite. Let $U \subset V$ be a subset of vertices, and let us simply write i for the vertex x_i. The number of edges between vertices in U is then given by $\frac{1}{2}\sum_{i,j\in U} a_{ij}$, where the factor $\frac{1}{2}$ compensates the fact that we count each edge twice. Note that even for loops, it gives the right answer due to our convention in the definition of the adjacency matrix for undirected graphs. On the other hand, for the configuration model the expected number of edges between vertices in U is given by $\frac{1}{2}\sum_{i,j\in U}\frac{k_i k_j}{2m}$, as a consequence of (11.1). Thus, the important quantity is the difference between the existing edges and the expected ones, namely

$$\frac{1}{2}\sum_{i,j\in U}\Big(a_{ij} - \frac{k_i k_j}{2m}\Big).$$

So far, we have used only one set $U \subset V$ which could correspond to one community. What about a situation where we would like to consider several communities ? For that purpose, we can complicate a little bit the above expression. Assume that the vertices are distributed within N communities, or equivalently that they are labeled with N different labels (or weights). Without loss of generality we can consider the set $\{1, 2, \dots, N\}$ as this set of labels, and write $\ell_i \in \{1, 2, \dots, N\}$ for the label of the vertex x_i. With the usual notation of the Kronecker delta function we shall write $\delta_{\ell_i\ell_j} = 1$ if $\ell_i = \ell_j$, and $\delta_{\ell_i\ell_j} = 0$ if $\ell_i \neq \ell_j$. Then one defines the *modularity*

$$Q := \frac{1}{2m}\sum_{i,j}\Big(a_{ij} - \frac{k_i k_j}{2m}\Big)\delta_{\ell_i\ell_j}. \tag{11.11}$$

Note that the preliminary factor $\frac{1}{2m}$ provides a kind of normalisation: we do not count edges anymore, but fraction of the total number of edges.

Note that so far, the modularity has been computed once the labels are given. In that sense, it is possible to check if a partition into some communities is valuable or not. However, the correct question is how to detect communities, and check that they are real ? Such investigations are often called *community detecting* or *modularity maximization*, and it is considered as a complicated problem. If falls into the general framework of *discrete optimization problems.* Indeed, first of all the problem is not really well posed, since the number and the size of the communities have not been specified. In addition, what makes a good partition, and is it possible to find a better one ? It

is certainly not possible to look for all possible partitions of the vertices in different subsets and to compute Q for all of them.

Let us just sketch a few ideas, and refer to the specialized literature for further information. Note that we provide information only in the very special case of two communities.

First of all, let us define

$$B_{ij} := a_{ij} - \frac{k_i k_j}{2m}$$

and observe that $\sum_i B_{ij} = k_j - \frac{k_j}{2m}\sum_i k_i = k_j - k_j = 0$, and similarly $\sum_j B_{ij} = 0$. Let us also use the labels $\{-1, 1\}$ instead of $\{1, 2\}$. With these labels, namely with $\ell_i \in \{-1, 1\}$ one gets that $\delta_{\ell_i \ell_j} = \frac{\ell_i \ell_j + 1}{2}$. Thus, by rewriting (11.11) and by using the special property mentioned above one infers that

$$Q = \frac{1}{4m}\sum_{ij} B_{ij}(\ell_i \ell_j + 1) = \frac{1}{4m}\sum_{i,j} B_{ij}\ell_i \ell_j = \frac{1}{4m}{}^t\ell B \ell \tag{11.12}$$

where we have used the notation B for the matrix $\{B_{ij}\}$ and ℓ for the vector with the n components $\ell_j \in \{-1, 1\}$. One has also used ${}^t\ell$ for the transpose vector. Observe that $\|\ell\|^2 = n$, where $\|\cdot\|$ denotes the Euclidean norm in $\mathbb{R}^n$. Once recast in this framework, we are looking at a maximization problem: Find the extremum of (11.12) under the constraint that $\|\ell\| = \sqrt{n}$. There is just one problem: the solution we are looking for should take place in $\{-1, 1\}^n$, but we shall reformulate the probem in $\mathbb{R}^n$ (which is necessary for using tools from calculus) and therefore obtain a solution in $\mathbb{R}^n$. Nevertheless, it will be possible to look at the closest solution inside our framework, and hope that the error by imposing $\ell_j \in \{-1, 1\}$ will not change the result drastically. Let us mention that in such a problem, one rarely looks for the best solution, but for a solution quite close to it.

Our new problem can now be solved with the technique of Lagrange multiplier. Namely, one looks for a solution of the system of equations

$$\frac{\partial}{\partial_{\ell_i}}\Big[\sum_{i,j} B_{ij}\ell_i\ell_j + \lambda\Big(n - \sum_i \ell_i^2\Big)\Big] = 0 \Longleftrightarrow \sum_j B_{ij}\ell_j = \lambda \ell_i.$$

By writting this system with matrices, we look for a solution of

$$B\ell = \lambda \ell.$$

which corresponds to an eigenvalue / eigenvector problem. But which eigenvalue ? By inserting this solution into (11.12) one infers that

$$Q = \frac{1}{4m}{}^t\ell B \ell = \frac{1}{4m}{}^t\ell \lambda \ell = \frac{n}{4m}\lambda.$$

Since we want to maximize Q, the corresponding eigenvalue should be maximum, or in other terms we look for the maximal eigenvalue of B, and for the corresponding eigenvector ℓ.

The solution of this problem will certainly not have its solution in $\{-1,1\}^n$. However, if $u \in \mathbb{R}^n$ denotes the eigenvector corresponding to the highest eigenvalue, then we can always set $\ell_j = +1$ if $u_j \geq 0$ and $\ell_j = -1$ if $u_j < 0$. The corresponding vector ℓ is not an eigenvector of the matrix B, but it turns out that the partition of the graph according to the label given by ℓ is often quite good. In fact, this method provides a surprisingly good solution in many situations, and has the advantage of being easily implementable. More sophisticated methods exist, but as a first and simple approach, this spectral method works well.

Chapter 12

Epidemics on graphs

In this final chapter we look at applications of graphs for the modelization of epidemic spread. It is certainly a hot topic, but clearly we can only touch its surface. Further investigations are encouraged. Note that in the first section, no graph is involved.

12.1 Basic models

In this section we introduce the simplest models for the spread of infections. These models always consist in a certain number of *compartments* which contain a population in a homogeneous state. For example, the compartment **S** contains the population *susceptible* of getting a disease, while the compartment **I** contains the population which has been *infected*. Now, these populations can consist of individuals (which means that the compartments contain an integer number of elements) or can consist in a percentage of the total population. In the first picture, the sum of the different compartments provide the total number of individuals in the population, while in the second picture the number in each compartment sum up to 1. Note that for simplicity, we shall continue speaking about individuals even in the second picture. In both pictures, the content of each compartment is time dependent. The evolution of the system is usually given by a system of differential equations relating the content of the different compartments. The number of compartments and the relations between their content determine the complexity of the model. Several parameters are often involved, and determining these parameters is often part of the problem.

12.1.1 The SI-model

This model is the simplest one and consist only of two compartments: **S** and **I**, which means that once an individual has been infected, it remains infected and can forever infect susceptible individuals. Let us introduce two variables describing the content of **S** and **I**, namely s and ι (pronounced *iota*). We shall assume that $s + \iota = 1$, which means that $s = 1 - \iota$. In fact, we should write $\iota(t)$ and $s(t) = 1 - \iota(t)$ with the variable t representing the time, but the notation ι and s is commonly admitted.

For this system, the flow between the two compartments corresponds to the individuals which get infected. In other words, some elements of **S** with leave this compartment, and join **I**. On the other hand, **I** will simply receive this flow of individuals from **S**, but nothing will escape from this compartment. The corresponding system of equations is

$$\begin{cases} \frac{\mathrm{d}s}{\mathrm{d}t} = -\beta s\iota \\ \frac{\mathrm{d}\iota}{\mathrm{d}t} = \beta s\iota \end{cases} \tag{12.1}$$

where $\beta > 0$ corresponds to a *transmission coefficient* or *contact average*. More precisely, β provides the contact rate with random other individuals per unit time. Observe that in this model, it is considered that the probability of getting infected is propositional to the number of infected persons and to the number of susceptible individuals. Note that since $s + \iota = 1$ we don't need both equations, and in addition the second one can be rewritten as

$$\frac{\mathrm{d}\iota}{\mathrm{d}t} = \beta\iota(1-\iota). \tag{12.2}$$

The equation (12.2) appears at many places, and is called *the logistic equation*. Its solution is also known, namely

$$\iota(t) = \frac{\iota_0 \mathrm{e}^{\beta t}}{1 - \iota_0 + \iota_0 \mathrm{e}^{\beta t}},$$

where ι_0 corresponds to the value of ι at $t = 0$. Note that an *initial condition* is always necessary for such an equation, but the choice of $t = 0$ for the initial condition is rather arbitrary. A representation of the function $t \mapsto \iota(t)$ for $t \geq 0$ is provided in Figure 12.1. It is easy to observe that for this model $\lim_{t\to\infty} \iota(t) = 1$, whenever $\iota_0 > 0$. As a consequence, $s(t)$ will converge to 0 while $\iota(t)$ will converge to 1. In other terms, for this model, whenever a tiny part of the population is infected, then the entire population will become infected, even if it takes a long time. This model is useful for some diseases, but its outcome is clearly not the only possible one.

12.1.2 The SIR-model

Another common model consists in three compartments, which are commonly called **S**, **I** and **R**, where the new compartment corresponds to individuals who have *recovered* from the infection. Typically, an infected individual stays a couple of time unit in the compartment **I** before leaving for the compartment **R**. Note that there are different ways to modelize the time spent in **I** before moving to **R**. Here we consider only the simplest situation, see Remark 12.1. The variables for this system are s, ι, and r, and they satisfy $s + \iota + r = 1$. The corresponding system of equation reads

$$\begin{cases} \frac{\mathrm{d}s}{\mathrm{d}t} = -\beta s\iota \\ \frac{\mathrm{d}\iota}{\mathrm{d}t} = \beta s\iota - \gamma\iota \\ \frac{\mathrm{d}r}{\mathrm{d}t} = \gamma\iota \end{cases} \tag{12.3}$$

where the additional parameter $\gamma > 0$ is interprated as $1/($*mean infectious time*$)$.

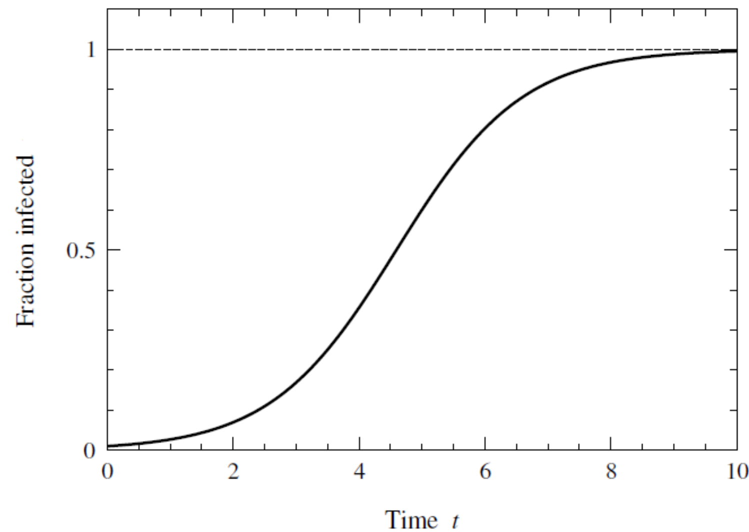


Fig. 12.1. The solution of the logistic equation, from Sec. 16.1 of [Ne]

Remark 12.1. *With the value γ, one can calculate the probability that an individual is still infected after a time t. Since the probability of recovering in any time interval Δt is equal to $\gamma\Delta t$, and the probability of not recovering is $1-\gamma\Delta t$, one gets that the probability of still being infected after a total time t is given by*

$$\lim_{\Delta t\to 0}(1-\gamma\Delta t)^{t/\Delta t} = \mathrm{e}^{-\gamma t}. \tag{12.4}$$

As a consequence, the probability that an individual is still infected after a total time t and recovers during the following interval Δt is given by $\gamma\mathrm{e}^{-\gamma t}\Delta t$, which corresponds to an exponential distribution. This behavior is certainly not very realistic, since usually an individual has a disease for about a fixed duration (1 week, 2 weeks, 1 month,...), while with the current model the individual is most likely to recover immediately, but might also keep the disease for an exponentially long time. Nevertheless, this model is often kept for its simplicity.

As for the **SI**-model, an initial condition has to be given. Here the natural choice is $\iota_0 > 0$ and $r_0 = 0$, meaning that no individual has recovered at time $t = 0$. Obviously, other choices are possible. Then, by eliminating the variable ι between the first and the third equation of (12.3) one obtains

$$\frac{1}{s}\frac{\mathrm{d}s}{\mathrm{d}t} = -\frac{\beta}{\gamma}\frac{\mathrm{d}r}{\mathrm{d}t}$$

leading to the solution

$$s = s_0\mathrm{e}^{-\beta r/\gamma}$$

with $s_0 = 1-\iota_0$. By inserting into the third equation of (12.3) the relation $\iota = 1-s-r$ and the previous result for s one then infers that

$$\frac{\mathrm{d}r}{\mathrm{d}t} = \gamma\big(1-r-s_0\mathrm{e}^{-\beta r/\gamma}\big). \tag{12.5}$$

Unfortunately, an explicit solution of this equation can be not obtained, but numerical evaluations are at hand. Typical outcomes for the system (12.3) are shown in Figure 12.2. In this picture, the value $\beta = 1$, $\gamma = 0.4$, $\iota_0 = 0.01$, $s_0 = 0.99$, and $r_0 = 0$ have been chosen.

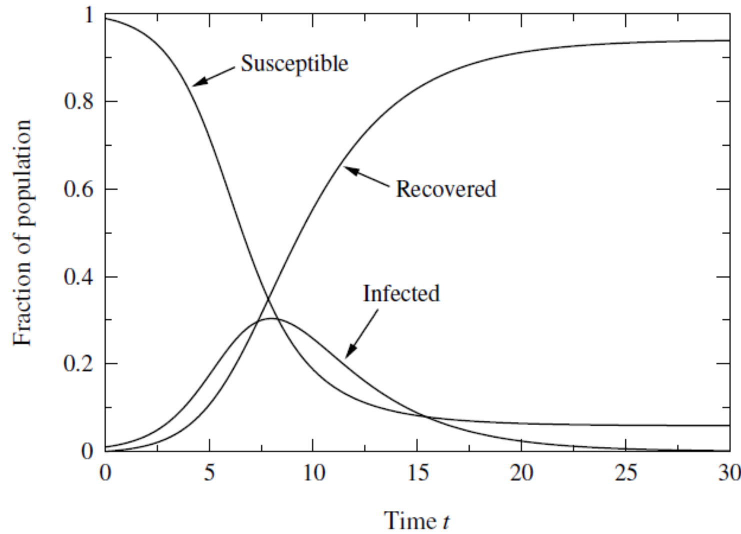


Fig. 12.2. The solution of the system (12.3), from Sec. 16.1 of [Ne]

Let us emphasize that the final outcome here is different from the one of the **SI**-model. Indeed, $s(t)$ does not converge to 0 as $t \to \infty$, which means that part of the population will never be infected. More precisely, when $t \to \infty$, there is no more evolution, and therefore $\frac{\mathrm{d}r}{\mathrm{d}t} = 0$. Thus, by using (12.5), one infers that $\lim_{t\to\infty}(1-r-s_0\mathrm{e}^{-\beta r/\gamma}) = 0$. In other words, it means that $r(\infty)$ satisfies the equation

$$r(\infty) = 1 - s_0\mathrm{e}^{-\frac{\beta}{\gamma}r(\infty)}.$$

Usually, the value s_0 is very close to 1, and therefore one looks for a solution of

$$r(\infty) = 1 - \mathrm{e}^{-\frac{\beta}{\gamma}r(\infty)}. \tag{12.6}$$

Quite surprisingly, this equation is similar to (10.9) which appears in relation with the existence of the giant component for the $\mathcal{G}(n,p)$ model. This means that we can borrow the results obtained in that context, namely: when $\frac{\beta}{\gamma} > 1$ there exists a solution $r(\infty)$ of (12.6) satisfying $r(\infty) < 1$, and accordingly that $s(\infty) > 0$. When $\frac{\beta}{\gamma} \leq 1$, no epidemic is taking place. The initial infected population recovers faster than the susceptible individuals become infected. The variable $\iota(t)$ will simply decrease, and no local maximum like in Figure 12.2 will take place. Note that some tools for visualizing the behavior of the evolution as a function of β and γ are easily available on internet, see for example [34]. Note that the transition $\frac{\beta}{\gamma} = 1$, which means $\beta = \gamma$ is called a *epidemic threshold*. Because of its importance, the ration $\frac{\beta}{\gamma}$ is called *basic reproduction number* and is often denoted by R_0.

12.1.3 Other models

There exist plenty of models based on the same ideas but with more compartments and a more complicated differential systems. Some key ideas are for example that

(i) Some individuals might come back from **R** to **S** if they have not got any immunity during the infected period,

(ii) The total population is not constant, due to births, deaths, or other factors,

(iii) Several diseases could interact and either facilitate each others, or prevent each others,

(iv) Some diseases need complex contagions, which means that the exposition to one infected person is not sufficient, but additional contacts with other infected persons are necessary.

This list could be continued endlessly, see [35] for further information. We shall not complicate any further our models, but let us mention that current models used for the Covid-19 can take up to 21 compartments into account, see Figure 12.3, and involve a system of 21 related equations.

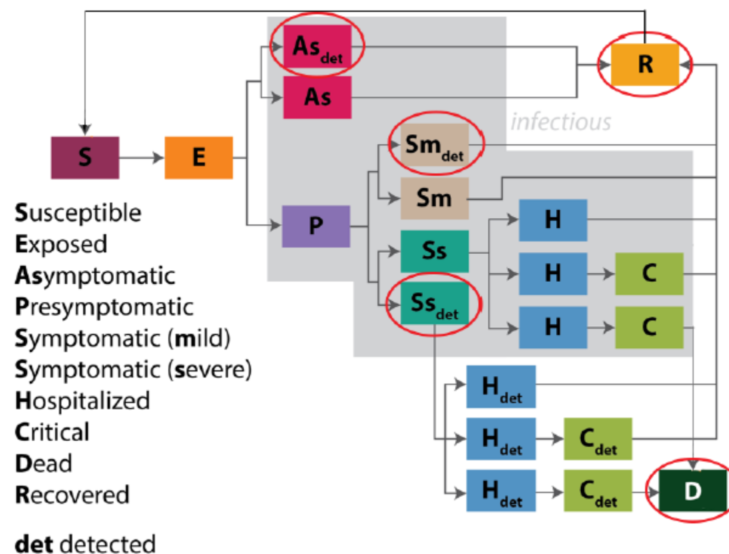


Fig. 12.3. One model used for the modelization of Covid-19, from [36]

12.2 Percolation

In the previous section, it was considered that any individual could be in contact with any other. In reality, individuals have often a set of acquaintances, and do not interact so

often with other individuals uniformly at random. In the sequel, the set of acquaintances will be described by a graph, with individuals represented by vertices, and with edges representing the relations between the individuals.

In this framework, a concept which is going to play an important role is the one of *percolation* or more precisely of *site percolation* or *bond percolation*. The idea of site percolation is the following: consider a connected undirected graph, and let us start removing some vertices uniformly at random. What is the proportion of vertices which have to be removed such that the graph becomes disconnected ? The central point in percolation theory is the existence of a *threshold value* such that the properties of the graphs below or above this value are very different. This threshold value is also called a *phase transition*, and it appeared already in the setting of random graphs in Chapters 10 and 11.

There exist different strategies for removing vertices of a graph. For example, one can remove some of them independently of their degree, or choose only the ones with the minimum or with the maximum degree. Depending on the purpose, a particular strategy can be more useful than others. Let us look at the situation in which the vertices are removed independently of their degree, they are chosen uniformly at random. We shall use the parameter ϕ for quantifying the removal process: ϕ is called *the occupation probability*, and $\phi = 1$ means that all vertices are present, and none has been removed. On the other hand, $\phi = 0$ means all vertices have been removed, there is no more any graph.

Let us now consider again the configuration model with a degree distribution $\{p_k\}_{k\in\mathbb{N}}$, and set u for the average probability that a vertex is not connected to the giant component via a particular neighbour. Suppose also that part of the vertices have been removed uniformly at random, and that ϕ provides the fraction of the remaining ones. Then, the relation that u has to satisfy is not more provided by (11.8) but has to be adapted if $\phi \neq 1$. There are two ways to not be connected to the giant component via a neighbour: either this neighbour has been removed, which happens with a probability $1-\phi$, or the neighbour is present (with a probability ϕ) but this one is not connected to the giant component. As a consequence, the new relation that u has to satisfy is

$$u = 1 - \phi + \phi g_1(u) \tag{12.7}$$

with g_1 defined in (11.9), see [Ne, Sec. 15.2.1] for the details. There is no way to solve (12.7), but a graphical approach is proposed in Figure 12.4.

From the graphical approach, it is quite clear that the threshold value for ϕ takes place when the r.h.s. of (12.7) satisfies

$$\frac{\mathrm{d}}{\mathrm{d}u}\big(1 - \phi + \phi g_1(u)\big)\big|_{u=1} = 1$$

which means that the two functions represented on Figure 12.4 are tangent at $u = 1$. Thus, this threshold ϕ_c satisfies $\phi_c = \frac{1}{g_1'(1)}$, which can be computed explicitly. By using (11.9) for the explicit expression for g_1 one finds

$$\phi_c = \frac{\langle k\rangle}{\langle k^2\rangle - \langle k\rangle}. \tag{12.8}$$

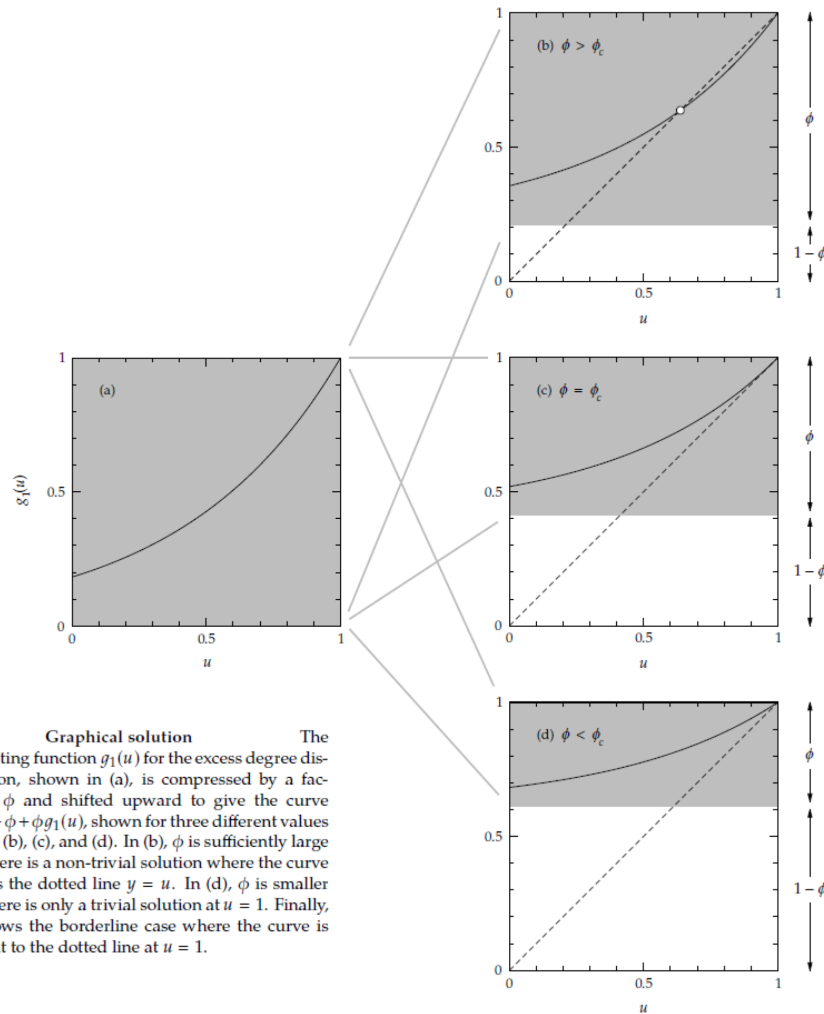


Fig. 12.4. A graphical resolution of (12.7), from Section 15.2.1 of [Ne]

Interestingly, if the degree distribution follows a Poisson distribution, as for the $\mathcal{G}(n,p)$ model, then one gets $\phi_c = \frac{1}{c}$, with $c = \langle k \rangle$ the mean degree of the graph. If the degree distribution follows a power law with $\alpha \in (2,3)$, it has already been mentioned that $\langle k^2 \rangle = \infty$. In such a situation, $\phi_c = 0$ meaning that no matter how many vertices will be removed from the graph, a giant component will persist ! This is the sign of the strong robustness of graphs with a degree distribution provided by a power law.

The previous construction is based on a uniform removal of nodes, but as already mentioned, other strategies might be useful. We refer to Section 15.3 of [Ne] for alternative constructions.

12.3 Epidemic on graphs and percolation

From now on, let us consider a fixed graph, with vertices representing individuals and edges representing the relation between these individuals. Clearly, graphs are loopless and unoriented, but an orientation could be added for some applications. Let us also introduce a *transmission rate* or *infection rate* β which is the probability per unit time that an infection will be transmitted between two individuals through an edge between the corresponding two vertices. Let us emphasize that this coefficient is slightly different form the one already introduced in (12.1) since in the present situation it corresponds to the rate of contact with just one individual connected through one edge. Observe that the transmission rate β depends on the disease itself, but also on the social and behavioural parameters of the population.

A precise evolution equation for a disease on a graph will only be introduced in the next section. We provide here only some heuristic considerations, together with a trick. We start by emphasizing a major difference between the previous continuous models, and any models on graphs. Indeed, if the transmission of a disease takes place trough edges and if at the initial time there exists only one single infected individual, then only the component of the corresponding vertex might expect an epidemic. The other components will remain uninfected. As a consequence, if a graph contains a giant component and several small components, with a fraction S of vertices in the giant component, the maximum fraction of individuals that one single initial individual can infect is precisely S, if this initial individual is in the giant component. If the corresponding vertex is in a small component, only a very small number of individuals might get the disease. In summary, the connected components play now a role which simply do not exist in the continuous models.

In a graph's version of the **SI**-model, one expects that the entire connected components of the initial infected individual will become infected, as soon as $\beta > 0$. This outcome is due to the fact that once infected, one individual will remain infected and can infect others forever.

Let us now present an heuristic argument for an analog of the **SIR**-model on graphs. Consider firstly one infected vertex which is connected to a susceptible vertex. By a computation similar to the one performed in (12.4), one infers that the probability that the disease is not transmitted during an interval of time τ is given by $\mathrm{e}^{-\beta\tau}$, where β is the transmission rate. Thus, the probability that the disease is transmitted during this interval is

$$\phi = 1 - \mathrm{e}^{-\beta\tau}. \tag{12.9}$$

If we now consider that the infected individual recovers precisely after a period of time equal to τ, the probability that he would have infected any connected vertex before recovering is precisely given by (12.9). We call this quantity *the transmission probability*. Let us emphasize that in this argument, the paradigm used in Remark 12.1 is changed: now the infected person is no more infectious after a time τ.

In this framework, the transmission probability is a constant over the graph, and any susceptible individual has an equal probability ϕ of getting the disease through

an infected neighbour. A trick, already introduced decades ago, is to use some ideas coming from the theory of percolation. Assume that an edge in the graph is present with a probability ϕ, and absent with a probability $1-\phi$. Equivalently, one can assume that the fraction $1-\phi$ of the edges has been removed. As a consequence, the remaining edges correspond to the ones along which the disease can propagate. However, if too many edges has been removed, an initially connected part of the graph might now be no more connected. The threshold and phase transition mentioned in the previous section are now going to play a role. Note however that there is a small difference between the current situation and the one of the previous section: here one should speak about *bond percolation* or *edge percolation* while *site percolation* was discussed before. Fortunately, this difference will not play any role for our purpose.

Let us again consider the configuration model, and let ϕ be the transmission probability given in (12.9). Even though the transmission probability is associated with edges, if we set u for the average probability that a vertex is not connected to the giant component via a particular neighbour, one gets again the self-consistent equation for u obtained in (12.7), namely $u = 1 - \phi + \phi g_1(u)$ with g_1 defined in (11.9). It follows that the result obtained in the previous section for site percolation can be used again, and one infers a transition for ϕ given by (12.8). This result together with (12.9) leads to the relation

$$\beta\tau = -\ln\big(1-\phi_c\big) = \ln\left(\frac{\langle k^2\rangle - \langle k\rangle}{\langle k^2\rangle - 2\langle k\rangle}\right). \tag{12.10}$$

Let us recall that the denominator on the r.h.s. is positive precisely when a giant component exists, see (11.7). Thus, if $\beta\tau$ is bigger that the value on the r.h.s. and if the initial infected individual belongs to the giant component, an epidemic is expected. On the other hand, if $\beta\tau$ is smaller than the r.h.s. then an epidemic will not take place, no matter where the initial infected individual is located. In the former case, note however that not all the individuals might get infected, only the one in the giant component, which does not represent the entire population in general.

As a consequence of (12.10), a certain control on the propagation of the disease is possible through β. This parameter is partially due to the inherent propagation properties of the disease itself, but also to the behavior of the population (*wear a mask !*). The r.h.s. is a property of the social relations, it can also be adjusted (less interactions between individuals) even if the effect is less clear through this formula.

12.4 Time dependent evolution

The link between epidemic on graphs and percolation can only lead to some asymptotic results. For a more precise picture, an evolution equation is necessary, and has not been introduced so far. We shall use the shorter notation introduced in Section 11.3, namely a vertex is simply denoted by i. Then, we write $s_i \equiv s_i(t)$ for the probability that the vertex i is susceptible, $\iota_i \equiv \iota_i(t)$ for the probability that the vertex i is infected, and $r_i \equiv r_i(t)$ for the probability that the vertex i has recovered. For the graph's version of

the **SI**-model, the relation $s_i + \iota_i = 1$ holds, while for the analog of the **SIR**-model the condition is $s_i + \iota_i + r_i = 1$ for any i.

Let us recall that the adjacency matrix has been introduced in Definition 2.1. Here, we shall simply denote it by $A = \{a_{ij}\}$. We first consider the **SI**-system. In this case, the natural analog of the differential system (12.1) takes the form

$$\begin{cases} \frac{\mathrm{d}s_i}{\mathrm{d}t} = -\beta s_i \sum_j a_{ij}\iota_j \\ \frac{\mathrm{d}\iota_i}{\mathrm{d}t} = \beta s_i \sum_j a_{ij}\iota_j \end{cases} \iff \begin{cases} \frac{\mathrm{d}s_i}{\mathrm{d}t} = -\beta s_i \sum_j a_{ij}(1-s_j) \\ \frac{\mathrm{d}\iota_i}{\mathrm{d}t} = \beta(1-\iota_i) \sum_j a_{ij}\iota_j \end{cases} \tag{12.11}$$

for any i. Again, one of these equations is sufficient since $s_i + \iota_i = 1$. For **SIR**-system, the natural analog of the differential system (12.3) takes the form

$$\begin{cases} \frac{\mathrm{d}s_i}{\mathrm{d}t} = -\beta s_i \sum_j a_{ij}\iota_j \\ \frac{\mathrm{d}\iota_i}{\mathrm{d}t} = \beta s_i \sum_j a_{ij}\iota_j - \gamma\iota_i \\ \frac{\mathrm{d}r_i}{\mathrm{d}t} = \gamma\iota_i, \end{cases} \tag{12.12}$$

where γ is *the recovery rate*, or more precisely the probability per unit time that an infected individual will recover. Note that the content of the last equation is closer to the approach taken in Section 12.1.2 than to the approach mentioned in Section 12.3.

An initial condition has to be associated to these systems. One possibility if to consider $\iota_i = \frac{c}{n}$ for any i, where c is a small positive integer and n represent the number of vertices of the graph or of the giant component. The idea is that c vertices are initially infected, and that the probability of being infected is distributed uniformly at random over all vertices. Clearly, other choices are possible.

The systems of equations (12.11) and (12.12) are usually not solvable explicitly, and several approaches have been developed. One can either look at numerical simulations, or get some analytical results after imposing some simplifications to these systems. Let us just mention a few key ideas. One approach is to consider the initial condition as mentioned above, which makes all initial ι_i small, and then approximate the second term in (12.11) as $\frac{\mathrm{d}\iota_i}{\mathrm{d}t} \cong \beta \sum_j a_{ij}\iota_j$. In other terms, it means ignoring the second order terms. This can be done for a short time approximation of the evolution, but the result is usually not so good. A better approach is called *pair approximation* and consider the product $s_i\iota_j$ as a new variable. More precisely, one sets p_{ij} for the probability that j is infected given that i is not infected. Then, it is possible to rewrite a system of equation for the variable p_{ij} and solve it rather explicitly (once a suitable assumption is taken on the relations between triplet of vertices). The outcome is usually much better than in the first approach, but whenever the graph is highly transitive, the approximation is no more suitable. Finally, a heuristic approach is to suppose that all vertices with the same degree have the same probability of getting infected at any time. With this assumption, one ends up with new variables which depend only on the degree k and no more on any specific vertex i. Surprisingly, this approach leads to rather accurate prediction. We refer to Sections 16.5 and 16.6 of [Ne] for more information on these approaches. For finite graphs of different types, simulations are quite enlightening and can be easily performed on the website [37].

Let us end this section with the so-called *mean field approach*. The main idea of mean field theory is the study the behavior of high-dimensional random models by studying a simpler model that approximates the original by averaging over degrees of freedom. Here, the simpler variables will the average number of vertices in a prescribed state. For that purpose, let us first denote by $X_i = X_i(t)$ the function (random variable) taking the three values S, I, or R depending if the vertex i is susceptible, infected or has recovered at time t. We then define for $A \in \{S, I, R\}$ the expected value

$$[A](t) := \sum_i \mathbb{P}\big(X_i(t) = A\big)$$

there $\mathbb{P}$ is the probability, and where we have assumed that the summation is taking place on the vertices of a finite simple graph. Then, $[S](t)$ represents the expected value for the number of vertices which are susceptible, $[I](t)$ the one for the number of infected vertices, and $[R](t)$ the one for the number of vertices which have recovered. Clearly, for the **SI**-model one has $[S](t) + [I](t) = n$, if n denotes the number of vertices of the graph, while for the **SIR**-model one has $[S](t) + [I](t) + [R](t) = n$

Let us now define for $A, B, C \in \{S, I, R\}$ the new quantity

$$[AB](t) := \sum_{i,j} a_{ij}\mathbb{P}\big(X_i(t) = A, X_j(t) = B\big)$$

and

$$[ABC](t) := \sum_{i,j,k} a_{ij}a_{jk}\mathbb{P}\big(X_i(t) = A, X_j(t) = B, X_k(t) = C\big),$$

where $\{a_{ij}\}$ denote the adjacency matrix for the graph. These quantities correspond to number of expected connected vertices which are in a prescribed states. Note that the role of the adjacency matrix is precisely to keep track of the vertices which are connected. Note also that $[AB](t) = [BA](t)$, but that such relations with three sets do not hold in general. Note also that for the **SI**-model, the relation

$$[SS](t) + [SI](t) + [IS](t) + [II](t) = nc \equiv n\langle k\rangle$$

where c or $\langle k\rangle$ denote the mean degree or (average degree) in the graph. Clearly, a similar relation holds for the **SIR**-model as well.

With these notations, the individual dynamics introduced in (12.11) and (12.12) can be transformed into collective dynamics, namely

$$\begin{cases} [\dot{S}](t) \equiv \frac{\mathrm{d}[S](t)}{\mathrm{d}t} = -\beta[SI](t) \\ [\dot{I}](t) \equiv \frac{\mathrm{d}[I](t)}{\mathrm{d}t} = \beta[SI] \end{cases} \tag{12.13}$$

and

$$\begin{cases} [\dot{S}](t) \equiv \frac{\mathrm{d}[S](t)}{\mathrm{d}t} = -\beta[SI](t) \\ [\dot{I}](t) \equiv \frac{\mathrm{d}[I](t)}{\mathrm{d}t} = \beta[SI](t) - \gamma[I](t) \\ [\dot{R}](t) \equiv \frac{\mathrm{d}[R](t)}{\mathrm{d}t} = \gamma[I](t). \end{cases} \tag{12.14}$$

Observe that the r.h.s. of (12.14) depends on the dynamics of $[SI](t)$. Thus, in order to solve this system, we could look for an additional equation. In fact, all quantities $[AB](t)$ with $A, B \in \{S, I, R\}$ are related, as shown in Figure 12.5. Note that in this figure, τ should be replaced by β. Thus, we can look for additional relation and find that

$$\frac{\mathrm{d}[SI](t)}{\mathrm{d}t} = -\gamma[SI](t) + \beta\big([SSI](t) - [ISI](t) - [SI](t)\big)$$

and

$$\frac{\mathrm{d}[SS](t)}{\mathrm{d}t} = -2\beta[SSI](t).$$

Similar relation can also be found for $[II](t)$, $[SR](t)$, $[IR](t)$ and $[RR](t)$. Clearly, one could go one, and get an infinite set of differential equations. However, the trick is do do some approximations for some products $[A \dots Z](t)$. For example, a common approximation takes the form

$$[ABC](t) \cong \frac{[AB](t)\ [BC](t)}{[B](t)}$$

and such a formula will "close" the system of equations. Note that such approximations can be justified according to the structure of the graphs, but they are always approximations. Clearly, by keeping more terms, one gets a better approximation of the true solution, but the price is an increase of complexity. Life is all about balance ☺. For further investigations about epidemics on networks, we refer to the monograph [KMS].

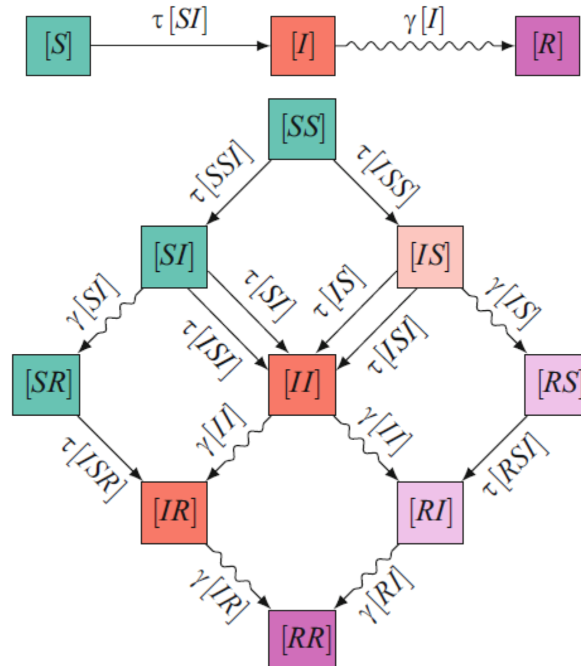


Fig. 12.5. The first two flow diagrams for the **SIR**-model, from Fig. 4.3 of [KMS]

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_problem

[12] https://www.cs.princeton.edu/∼wayne/kleinberg-tardos/

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

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

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

[16] https://en.wikipedia.org/wiki/Seven_Bridges_of_Königsberg

[17] https://www.geeksforgeeks.org/chinese-postman-route-inspection-set-1-introduction/

[18] https://en.wikipedia.org/wiki/Double_factorial

[19] https://en.wikipedia.org/wiki/Petersen_graph

[20] https://en.wikipedia.org/wiki/Heuristic_(computer_science)

[21] https://www.codingalpha.com/christofides-algorithm-c-program/

[22] O. Svensson, J. Tarnawski, L. Végh, *A Constant-Factor Approximation Algorithm for the Asymmetric Traveling Salesman Problem*, Preprint arXiv:1708.04215, 2017.

[23] https://en.wikipedia.org/wiki/Floor_and_ceiling_functions

[24] https://www.ics.uci.edu/∼eppstein/junkyard/euler/

[25] https://en.wikipedia.org/wiki/Four_color_theorem

[26] https://n.freemap.jp/tw/20150403_23572634232

[27] https://en.wikipedia.org/wiki/Five_color_theorem

[28] https://en.wikipedia.org/wiki/Strongly_connected_component

[29] https://en.wikipedia.org/wiki/Social_choice_theory

[30] https://cp-algorithms.com/graph/edmonds_karp.html

[31] https://en.wikipedia.org/wiki/Edmonds-Karp_algorithm

[32] https://brilliant.org/wiki/hall-marriage-theorem/

[33] https://en.wikipedia.org/wiki/Erdős-Rény_model

[34] http://math.colgate.edu/∼wweckesser/solver/DiseaseSIR.shtml

[35] https://en.wikipedia.org/wiki/Compartmental_models_in_epidemiology

[36] E, Amstrong, M. Runge, J. Gerardin, *Identifying the measurements required to estimate rates of COVID-19 transmission, infection, and detection, using variational data assimilation*, medRxiv preprint doi: https://doi.org/10.1101/2020.05.27.20112987

[37] http://systems-sciences.uni-graz.at/etextbook/networks/sirnetwork.html