Elementary Introduction to Graph Theory

(01EIG 2025/2026)

Francesco Dolce
dolcefra@cvut.cz

PDF available at the address: dolcefra.pages.fit/ens/2526/eig.html

updated: December 16, 2025

Contents

Chapter 1

Introduction

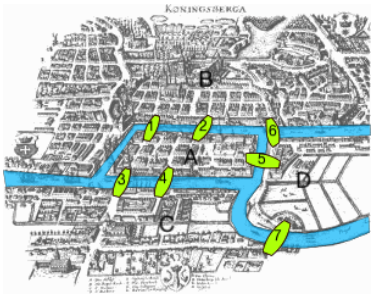
This booklet contains the material for the course `01EIG` (*Elementary Introduction to Graph Theory*) taught during the Winter Semester 2025/2026 at the Faculty of Nuclear Science and Physical Engineering (FJFI) of the Czech Technical University in Prague (ČVUT v Praze).

If you find any typo or mistake, please write me an email at the address `dolcefra@cvut.cz`.

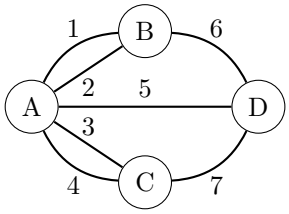
A Swiss mathematician and seven Prussian bridges

In 1736 the mayor of the Prussian city of Königsberg asked the mathematician Leonhard Euler whether it was possible to take a walk around the city crossing each of the seven bridges exactly once.

Euler's intuition was to describe the problem by means of a diagram consisting of a set of points and lines instead of regions and bridges.



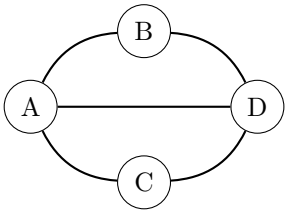
(a) Königsberg in Euler's time.



(b) Diagram of Königsberg.

History passed on Königsberg, now known as Kaliningrad (or Královec), and some of the bridges fell down.

(a) Kaliningrad nowadays.



(b) Diagram of Kaliningrad.

Was the walk possible before? Is it now?

Chapter 2

Basic definitions

2.1 Vertices and edges

Let V be a set and $r \in \mathbb{N}_0$. By $\binom{V}{r}$ we denote the set of all r-element subsets of V. Note that $\#\binom{V}{r} = \binom{\#V}{r}$, where $\#A$ is the cardinality of A.

Example 2.1 If $V = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, then

$$\binom{V}{0} = \{\emptyset\}, \qquad \binom{V}{1} = \{\{\mathtt{a}\}, \{\mathtt{b}\}, \{\mathtt{c}\}\},$$

$$\binom{V}{2} = \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{c}\}\}, \qquad \binom{V}{3} = \{V\}$$

A *graph* is a pair $G = (V, E)$, where V is a set of *vertices* and E is a multiset of *edges* of the form $e = \{u, v\}$, with $u, v \in V$. The numbers $\#V$ and $\#E$ are respectively the *order* and the *size* of G. A graph G is *finite* if both $\#V$ and $\#E$ are finite.

The usual way to illustrate a graph is to draw a dot for each $v \in V$ (or a circle with the label v), and to join $u, v \in V$ by a line if $e = \{u, v\} \in E$.

Example 2.2 The order of the graph $G = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\}\})$ is 3, while its size is 2 (see left of Figure 2.1).

The order of the graph $G' = (\{\mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}\}, \{\{\mathtt{u}, \mathtt{w}\}, \{\mathtt{v}, \mathtt{x}\}, \{\mathtt{v}, \mathtt{x}\}, \{\mathtt{y}, \mathtt{y}\}\})$ is 5, while its size is 4 (see right of Figure 2.1).

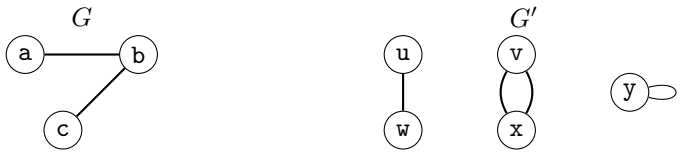


Figure 2.1: Two graphs.

Graphs can be represented graphically, but there is not a unique way to do so: relative position of points, shape/length of lines, etc. We are only interested in the incidence relation between vertices and edges.

Two vertices connected by an edge are said to be *adjacent*. They are both *incident* with the edge. Two distinct adjacent vertices are *neighbours*. Given a vertex $v \in V$ we define its neighbourhood in G

$$\mathrm{N}_G(v) = \{u \mid u \text{ neighbour of } v\} = \{u \mid \{u, v\} \in E\}$$

An edge of the form $\{u, u\}$ is called a *loop*. Two edges connecting the same vertices are called *multiedges* (or *parallel edges*) A graph $G = (V, E)$ is *simple* if it does not contain any loop or multiedge, i.e., if $E \subseteq \binom{V}{2}$.

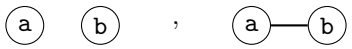
The graph $(\emptyset, \emptyset)$ is called the *null graph*.

We will mostly consider finite non-empty simple graphs.

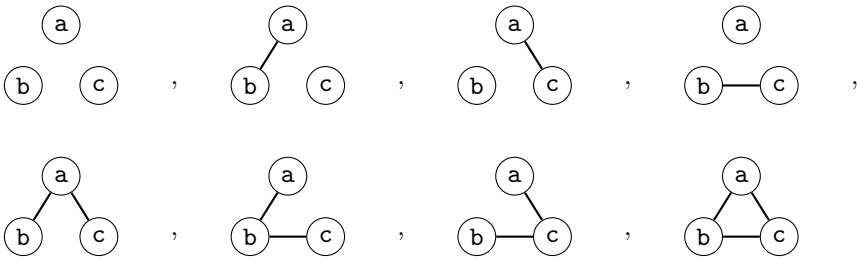
Example 2.3 How many simple graphs are there on n vertices? $2^{\binom{n}{2}}$.

- The only graph of 0 vertices is the null graph: $2^{\binom{0}{2}} = 2^0 = 1$.
- If $V = \{\mathtt{a}\}$, there is only one possible graph: $2^{\binom{1}{2}} = 2^0 = 1$

- If $V = \{\mathtt{a}, \mathtt{b}\}$, there are two possible graphs: $2^{\binom{2}{2}} = 2^1 = 2$



- If $V = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, there are eight possible graphs: $2^{\binom{3}{2}} = 2^3 = 8$



2.2 Some important graphs

Any graph of the form $(\{v\}, \emptyset)$, i.e., with only one vertex and no edges, is called a *trivial graph*. Every other graph is called *non-trivial*.

A simple graph of the form $K_n = \left(V, E = \binom{V}{2}\right)$, with $\#V = n$, is called a *complete graph* (see Figure 2.2).

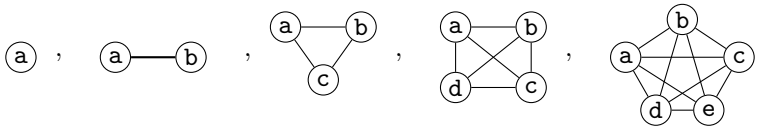


Figure 2.2: Complete graphs K_1, K_2, K_3, K_4 and K_5.

A graph $G = (V, E)$ is *bipartite* if the set of vertices is a disjoint union $V = X \sqcup Y$ and every edge $e \in E$ is of the form $e = \{x, y\}$ with $x \in X$ and $y \in Y$. When , $\#X = m$ and $\#Y = n$, and we are connecting every edge of X with every edge of Y, we call such a graph *complete bipartite* and denote it $K_{n,m}$ (see Figure 2.3). A graph of the form $K_{1,k}$ is called *k-star*.

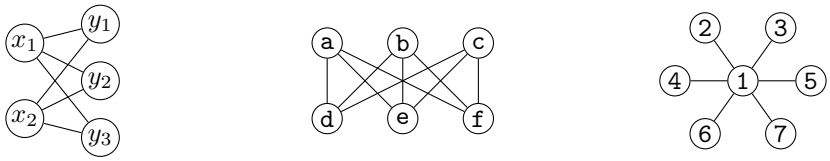


Figure 2.3: Three bipartite graphs.

2.3 Graph operations

If we have two graphs $G = (V, E)$ and $H = (U, F)$ we can combine them in several ways to obtain new graphs.

Their *union* is defined as $G \cup H = (V \cup U, E \cup F)$. Note that if $V \cap U = \emptyset$ we just obtain a "juxtaposition" of the two graphs.

Their *intersection* is defined as $G \cap H = (V \cap U, E \cap F)$.

Example 2.4 Let us consider $G = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}, \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{d}\}, \{\mathtt{b}, \mathtt{c}\}, \{\mathtt{c}, \mathtt{d}\}\})$ and $H = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}, \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{e}\}, \{\mathtt{c}, \mathtt{e}\}\})$ (see Figure 2.4). Then

$$G \cup H = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}, \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{d}\}, \{\mathtt{b}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{e}\}, \{\mathtt{c}, \mathtt{d}\}, \{\mathtt{c}, \mathtt{e}\}\})$$

and

$$G \cap H = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\{\mathtt{a}, \mathtt{b}\}\}) .$$

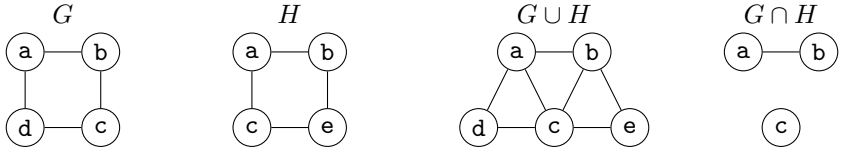


Figure 2.4: Union and intersection of two graphs.

The *cartesian product* $G \,\square\, H$ of the two graphs is defined as the graph having set of vertices $V \times U$ and edges defined by

$$\{(a,u),(b,v)\} \text{ edge of } G \,\square\, H \Leftrightarrow ((\{a,b\} \in E \wedge u = v) \ \vee \ (a = b \wedge \{u,v\} \in F)) \,.$$

Example 2.5 Let $G = (\{\mathtt{a},\mathtt{b},\mathtt{c}\}, \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{a},\mathtt{c}\},\{\mathtt{b},\mathtt{c}\}\})$ and $H = (\{1,2\},\{\{1,2\}\})$. The cartesian product $G \,\square\, H$ is represented in Figure 2.5.

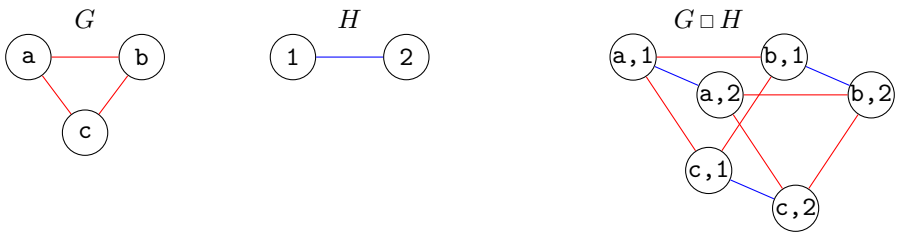


Figure 2.5: Cartesian product of two graphs.

If two graphs $G = (V, E_1)$ and $H = (V, E_2)$ are defined over the same set of vertices, we can define their *symmetric difference* $G \,\triangle\, H$ as

$$G \,\triangle\, H = (V, E_1 \,\triangle\, E_2), \qquad \text{where} \quad E_1 \,\triangle\, E_2 = (E_1 \setminus E_2) \cup (E_2 \setminus E_1) \,.$$

Example 2.6 Let $G = (V, E_1)$ and $H = (V, E_2)$,where $V = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$, $E_1 = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{a},\mathtt{d}\},\{\mathtt{b},\mathtt{c}\},\{\mathtt{c},\mathtt{d}\}\}$, and $E_2 = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{b},\mathtt{d}\},\{\mathtt{c},\mathtt{d}\}\}$. Then $G \,\triangle\, H = (V, \{\{\mathtt{a},\mathtt{d}\},\{\mathtt{b},\mathtt{d}\},\{\mathtt{b},\mathtt{c}\}\})$ (see Figure 2.6).

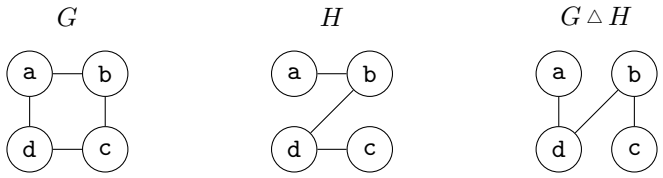


Figure 2.6: Symmetric difference of two graphs.

2.4 Degree of vertices

Let $G = (V, E)$ be a simple graph. The *degree* of a vertex $v \in V$ in G is

$$d_G(v) = \#\mathrm{N}_G(v) = \#\{u \in V \mid \{v,u\} \in E\} \,,$$

i.e., the number of edges in G incident with v. When G is clear from the context, we just write $d(v)$. The *minimum degree* of G is $\delta(G) = \min\{d_G(v) \mid v \in V\}$.

The *maximum degree* of G is $\Delta(G) = \max\{d_G(v) \mid v \in V\}$. The *average degree* of G is $d(G) = \dfrac{1}{\#V}\sum_{v\in V} d_G(v)$. If $d_G(v) = 0$, we say that v is an *isolated vertex* in G.

Example 2.7 Let G be the graph in Figure 2.7. We have

$$\delta(G) = d(\mathtt{f}) = 0, \qquad d(\mathtt{a}) = d(\mathtt{b}) = d(\mathtt{c}) = d(\mathtt{d}) = 1, \qquad \Delta(G) = d(\mathtt{e}) = 4,$$

$$d(G) = \frac{1+1+1+1+4+0}{6} = \frac{4}{3}.$$

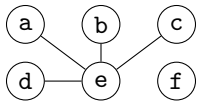


Figure 2.7: A graph with minimal degree 0 and maximal degree 4.

The definition can be extended to non simple graphs by counting parallel edges separately. In this case each loop counts twice.

Lemma 1 (Handshaking Lemma) *Let $G = (V, E)$ be a graph. Then,*

$$\sum_{v\in V} d_G(v) = 2\#E.$$

Proof. We sum up all vertices multiplied by their degree. We count every edge exactly twice, one for each of its ends. ∎

Corollary 1 *Let $G = (V, E)$ be a graph. Then, $\#\{v \mid d_G(v)$ is odd$\}$ is even.*

Proof. For each $v \in V$ we have

$$d_G(v) \equiv \begin{cases} 1 \pmod 2 & \text{if } d_G(v) \text{ is odd,} \\ 0 \pmod 2 & \text{if } d_G(v) \text{ is even.} \end{cases}.$$

Thus, $\sum_{v\in V} d_G(v) \equiv \#$ vertices of odd degree $\pmod 2$.

By the Handshaking Lemma 1, $\sum_{v\in V} d_G(v) \equiv 0 \pmod 2$. So, the number of vertices with odd degree is also even. ∎

Example 2.8 Let G be the graph in Example 2.7. The sum of the degrees is 8 and we have 4 vertices of odd degree: `a,b,c,d`.

A graph $G = (V, E)$ is called *k-regular* if $d_G(v) = k$ for every $v \in V$.

Example 2.9 K_n is $(n-1)$-regular for every $n \in \mathbb{N}_0$. A 3-regular graph is called a *cubic graph* (see Figure 2.8).

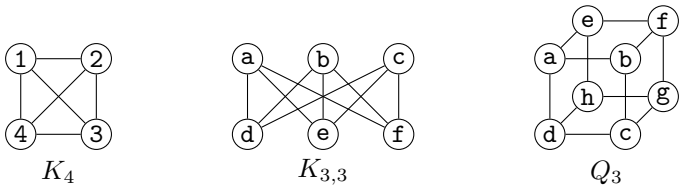


Figure 2.8: Three cubic graphs.

A graph in which each vertex has even degree is called an *even graph*.

Example 2.10 The graph K_2 (Figure 2.2) is even, while the graph $K_{1,6}$ (Figure 2.3) is not.

2.5 Graphic sequences

A sequence $(d_i)_{i=1}^n$ is called *graphic* if there exists a simple graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$ such that for every $1 \leq i \leq n$ we have $d_G(v_i) = d_i$. In this case we say that G *realises* the sequence $(d_i)_{i=1}^n$.

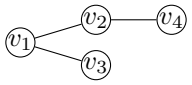
We usually order the sequence s.t. $d_1 \geq d_2 \geq \ldots \geq d_n \geq 0$.

Clearly not every sequence of non-negative integers is graphic. The Handshaking Lemma 1 gives us a necessary condition: $\sum_{i=1}^n d_i$ must be even. Also, in this case we must have $d_1 \leq n-1$.

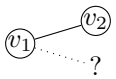
Example 2.11

- The sequence $(2, 2, 1, 1)$ is graphic.

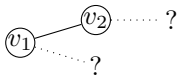
  A graph realising it is $G = (\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_4\}\})$.



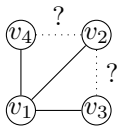
- The sequence $(2, 1)$ is not graphic. Indeed $2 + 1 = 3$ is not even.



- The sequence $(2, 2)$ is not graphic. Indeed $d_1 = 2 > 2 - 1 = 1$.



- The sequence $(3, 3, 1, 1)$ is not graphic (why?).



Theorem 2.1 (Havel (1955), Hakimi (1961)) *A non-increasing non-negative sequence $(d_i)_{i=1}^n$ is graphic if and only if*

$$(d_2 - 1, d_3 - 1, \dots d_{d_1+1} - 1, d_{d_1+2}, d_{d_1+3}, \dots, d_n)$$

is graphic.

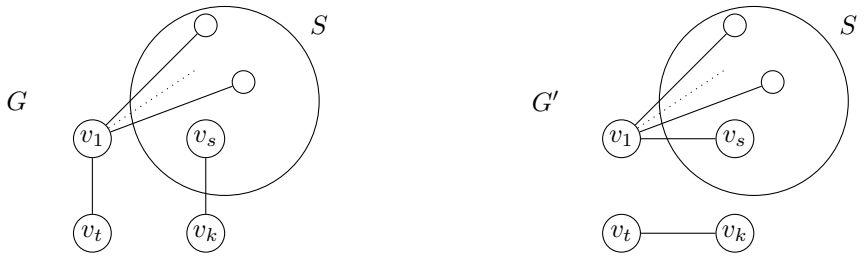
To prove Havel-Hakimi Theorem we need the following result.

Lemma 2 *Let $(d_i)_{i=1}^n$ be a graphic sequence with $d_1 \geq d_2 \geq \ldots \geq d_n \geq 0$. Then, there is a simple graph $G = (V, E)$ realising $(d_i)_{i=1}^n$ such that $V = \{v_1, \dots, v_n\}$, $d_G(v_i) = d_i$ for every i, and $\mathrm{N}_G(v_1) = \{v_2, \dots, v_{d_1+1}\}$.*

Proof. Let $S = \{v_2, \dots, v_{d_1+1}\}$. From all graphs realising $(d_i)_{i=1}^n$ let us select G s.t. $r = |\mathrm{N}_G(v_1) \cap S|$ is maximum.

If $r = d_1$, then G is the graph we are looking for and we are done.

If, by contradiction, $r < d_1$, then there exist s, t with $1 \leq s \leq d_1 + 1 < t \leq n$ such that $\{v_1, v_t\} \in E$ and $\{v_1, v_s\} \notin E$. Since $d_G(v_s) \geq d_G(v_t)$ and $\{v_1, v_t\} \in E$, there exists v_k such that $\{v_s, v_k\} \in E$ and $\{v_t, v_k\} \notin E$. Let $G' = (G \setminus \{\{v_1, v_t\}, \{v_s, v_k\}\}) + \{\{v_1, v_s\}, \{v_t, v_k\}\}$.



We have $|\mathrm{N}_{G'}(v_1) \cap S| = r + 1 > r$, which contradict the maximality of r. ∎

Proof of of Havel-Hakimi Theorem.

($\Rightarrow$) If $(d_i)_i$ is graphical, by Lemma 2 there exists a graph G realising it with $\mathrm{N}_G(v_1) = \{v_2, \ldots, v_{d_1+1}\}$. Thus $G \setminus \{v_1\}$ has degree sequence $(d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$.

($\Leftarrow$) If $(d_2 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$ is graphical, there exists a graph $G' = (V', E')$ with $\#V' = n - 1$, $V' = \{v_2, \ldots, v_n\}$ such that

$$d_{G'}(v_i) = \begin{cases} d_i - 1 & \text{for } 2 \leq i \leq d_{d_1+1} \\ d_i & \text{for } d_{d_1+2} \leq i \leq n \end{cases}.$$

Thus the graph $G = \left(V' \cup \{v_1\},\ E' + \bigcup_{i=2}^{d_1+1} \{v_1, v_i\} \right)$ realises $(d_i)_{i=1}^n$.

■

Havel-Hakimi Theorem gives us a recursive algorithm to check whether a sequence is graphic.

Algorithm 1: `GraphicSequence`$(d_1, d_2, \ldots, d_n)$

Input: non-increasing sequence $(d_1, d_2, \ldots, d_n)$
Output: `TRUE` if $(d_i)_{i=1}^n$ is graphic; `FALSE` if not

```
1 if (d_1 > n − 1) or (d_n < 0) then
2     return FALSE
3 else if d_1 = 0 then
4     return TRUE
5 else
6     Let (a_1, a_2, ..., a_{n−1}) non-increasing permutation of
        (d_2 − 1, ..., d_{d_1+1} − 1, d_{d_1+2}, ..., d_n)
7     GraphicSequence(a_1, a_2, ..., a_{n−1})
```

Example 2.12

- $(2, 1)$ is not a graphic sequence, since $2 > 1$.
- $(\mathbf{3}, \underline{2, 1, 1}) \to (\mathbf{1}, \underline{0}, 0) \to (-1, 0) \sim (0, -1)$ is not graphic since $-1 < 0$.
- $(\mathbf{2}, \underline{2, 1}, 1) \to (1, 0, 1) \sim (\mathbf{1}, \underline{1}, 0) \to (0, 0)$ is graphic (see Figure 2.9).

We can also use the algorithm backwards to realise a graphic sequence: we start with as many vertices are there are 0s in the last sequence and then, at each step, add a vertex with edges according to its degree (see Figure 2.9).

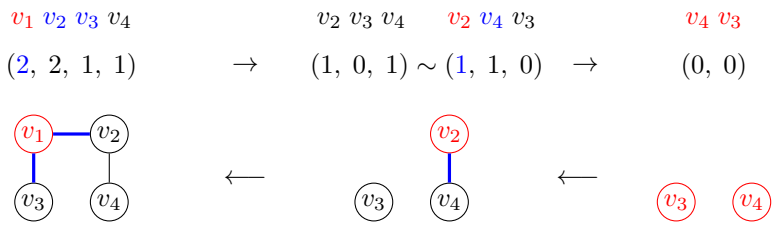


Figure 2.9: Realising a graph from a graphic sequence.

Exercise 1 Determine which of the following sequences are graphic, and in the positive case find a graph realising them: $(7,6,5,4,3,3,2)$, $(3,3,2,2,1,1)$.

Solution of Exercise 1 Determine which of the following sequences are graphic, and in the positive case find a graph realising them: $(7,6,5,4,3,3,2)$, $(3,3,2,2,1,1)$.

- $(7,6,5,4,3,3,2)$ is not a graphic sequence, since $7 > 6$.
- $(\mathbf{3},\underline{3,2,2},1,1) \rightarrow (\mathbf{2},\underline{1,1},1,1) \rightarrow (0,0,1,1) \sim (\mathbf{1},\underline{1},0,0) \rightarrow (0,0,0)$ is graphic (see Figure 2.10).

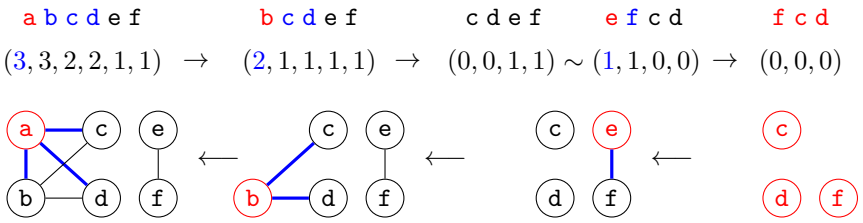


Figure 2.10: Realising a graph from a graphic sequence.

We also have the following non recursive result.

Theorem 2.2 (Erdős, Gallai (1960)) *A sequence $(d_i)_{i=1}^n$ with $d_1 \geq d_2 \geq \ldots \geq d_n \geq 0$ is graphic if and only if*

$$\sum_{i=1}^{n} d_i \text{ is even} \qquad \text{and} \qquad \sum_{i=1}^{k} d_i < k(k-1) + \sum_{i=k+1}^{n} \min\{k, d_i\} \quad \forall\, 1 \leq k \leq n.$$

Chapter 3

Subgraphs, connected graphs and trees

3.1 Subgraphs

Let $G = (V, E)$ be a graph. A graph $H = (U, F)$ is a *subgraph* of G, written $H \subseteq G$ (or, G is a *supergraph* of H, written $G \supseteq H$) if $U \subseteq V$ and $F \subseteq E$. We say that H is *contained in* G (or, G *contains* H).

Example 3.1 The graphs H_1, H_2, H_3 in Figure 3.1 are subgraphs of G. On the other hand, H_4 is not a subgraph of G.

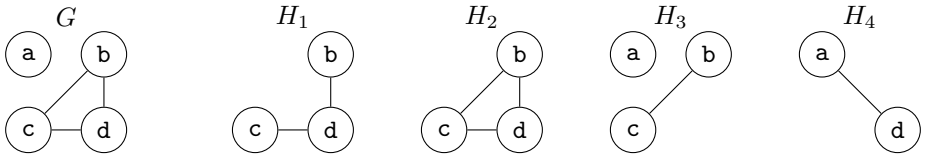


Figure 3.1: Five graphs with vertices (subsets of) $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$.

A subgraph $H = (U, F)$ of $G = (V, E)$ is called an *induced subgraph* of G by U, and we write $H = G[U]$, if for every two vertices $x, y \in U$ we have $\{x, y\} \in F \Leftrightarrow \{x, y\} \in E$. That is, the vertices in H are connected by exactly the same edges as in G. When H is simple we have $H = G[U] = \left(U, E \cap \binom{U}{2}\right)$.

Example 3.2 Let G, H_1, H_2 and H_3 as in Example 3.1. We have $H_2 = G[\{\mathtt{b}, \mathtt{c}, \mathtt{d}\}]$ and $H_3 = G[\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}]$. Note that H_1 is not an induced subgraph of G, since $\{\mathtt{b}, \mathtt{c}\}$ is an edge of G but not of H_1.

If $U \subset V$ and $F \subset \binom{V}{2}$, we write $G - U = G[V \setminus U]$. That is, $G - U$ is obtained from G by deleting all vertices in U and their incident edges.

Given a graph $G = (V, E)$ and a set of edges F, the graph $G \setminus F = (V, E \setminus F)$ is obtained by deleting some edges, the ones in $E \cap F$, but keeping all vertices. We call $G \setminus F$ a *spanning subgraph* of G.

In a similar way, given a set of vertices U and a set of edges F, we define $G + U = (V \cup U, E)$ and $G + F = (V, E \cup F)$.

When $U = \{u\}$ we just write $G + u$ instead of $G + \{u\}$ and $G - u$ instead of $G - \{u\}$. Similarly when $E = \{e\}$ we write $G + e$ and $G \setminus e$.

Example 3.3 Let $G, H_1 H_2, H_3$ and H_4 as in Example 3.1. Then $H_2 = G - \mathtt{a}$ and $H_3 = G - \mathtt{d}$. The graph H_1 is a spanning subgraph of H_2, indeed $H_1 = H_2 \setminus \{\mathtt{b}, \mathtt{c}\}$. Obviously we also have $H_2 = H_1 + \{\mathtt{b}, \mathtt{c}\}$. Moreover $G = H_2 + \mathtt{a}$.

3.2 Paths and cycles

A *path* is a non-empty graph $P = (V, E)$ with $V = \{v_0, v_1, \ldots, v_k\}$ and $E = \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}\}$ The vertices v_0, v_k are *linked* by P and are called its *endpoints*. The vertices $v_1, v_2, \ldots, v_{k-1}$ are the *inner vertices* of P.

If $k \geq 3$, we call *cycle* a graph of the form $C = (P - v_k) + \{v_{k-1}, v_0\}$.

The *length* of P (resp., of C) is k. A path (resp., cycle) is *odd* or *even* according to the parity of k. When a path (resp., cycle) appears as a subgraph G, we say that the path (resp., cycle) is *in* G.

We can represent a path (resp., a cycle) as a sequence of vertices

$$(v_0, v_1, \ldots, v_k)$$

or as a sequence of edges

$$(e_1, e_2, \ldots, e_k)$$

or, when interested in both vertices and edges, using the notation

$$(v_0 \overset{e_1}{\to} v_1 \overset{e_2}{\to} \cdots \overset{e_{k-1}}{\to} v_{k-1} \overset{e_k}{\to} v_k)$$

where $e_i = \{v_{i-1}, v_i\}$.

The *distance* between two vertices $u, v \in V$ in a graph $G = (V, E)$, denoted by $d_G(u, v)$, is the length of the shortest path in G with endpoints u and v. The *eccentricity* of v is defined as $\mathrm{ecc}_G(v) = \max\{d_G(v, u) \mid u \in V\}$.

The *center* of G is the set of its vertices with minimal eccentricity, i.e.,

$$\mathcal{C}(G) = \mathrm{argmin}_{v \in V} \mathrm{ecc}_G(v) = \left\{ v \in V \mid \mathrm{ecc}_G(v) = \min_{u \in V} \mathrm{ecc}_G(u) \right\}.$$

Example 3.4 The path P_4 and the cycle C_3 are represented on the left of Figure 3.2. On the right of the same figure you can see a path of length 5, $(\mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f})$, and a cycle of length 4, $(\mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e})$, in a graph G.

Note that we have $d_G(\mathtt{b}, \mathtt{f}) = 2$ since a shortest path in G between the two vertices is $(\mathtt{b}, \mathtt{e}, \mathtt{f})$. One can check that, e.g., $\mathrm{ecc}_{P_4}(v_0) = 4$, $\mathrm{ecc}_{P_4}(v_3) = 3$; $\mathrm{ecc}_{C_3}(u_2) = 1$; $\mathrm{ecc}_G(\mathtt{a}) = 3$, and $\mathrm{ecc}_G(\mathtt{c}) = 2$. Moreover $\mathcal{C}(P_4) = \{v_2\}$, $\mathcal{C}(C_3) = \{u_0, u_1, u_2\}$, and $\mathcal{C}(G) = \{\mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$.

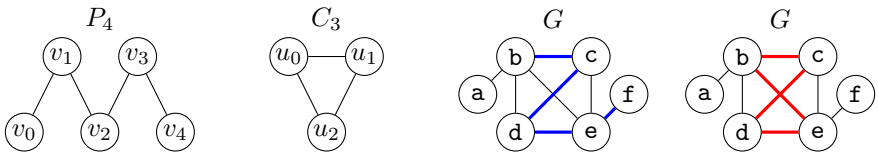


Figure 3.2: Paths and cycles.

3.3 Connected graphs

A graph is *connected* if it is non-empty and every two of its vertices are linked by a path in it. A maximal connected subgraph is a *component* of the graph. A graph with more than one component (i.e., that is not connected) is called *disconnected*.

Example 3.5 The graph G in Figure 3.3 has three components:

$$H_1 = G[\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}], \quad H_2 = G[\{\mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}\}] \quad \text{and} \quad H_3 = G[\{\mathtt{j}\}].$$

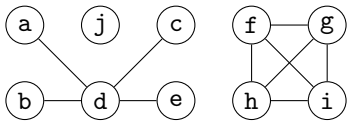
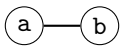


Figure 3.3: A graph with three components.

Let S_n be the number of different connected graphs on n vertices.

- If $V = \{\mathtt{a}\}$, there is only one possible graph, so $S_1 = 1$.

- If $V = \{\mathtt{a}, \mathtt{b}\}$, the two vertices are connected by an edge, so $S_2 = 1$.



- If $V = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, there are four possible connected graphs, so $S_3 = 4$.

Theorem 3.1 *For every* $n \in \mathbb{N}$

$$n \cdot 2^{\binom{n}{2}} = \sum_{k=1}^{n} \binom{n}{k} S_k \cdot k \cdot 2^{\binom{n-k}{2}}.$$

Proof. Let us count, in two different ways, all "rooted" graphs on n vertices, i.e., graphs with one particular vertex emphasised.

We know that the number of labelled graphs (connected or not) is $2^{\binom{n}{2}}$. So, the total number of rooted graphs is $n \cdot 2^{\binom{n}{2}}$.

On the other hand, each "root" will appear in a connected component of size k, with $1 \leq k \leq n$. For a fixed k, we have $\binom{n}{k}$ possibilites to select the k vertices, S_k ways of having the component connected, and k ways of selecting the "root" in the connected component; we do not know whether the remaining $n-k$ vertices are connected or not, so we have $2^{\binom{n-k}{2}}$ possibilites for them. ∎

Theorem 3.1 is quite unsatisfactory since it is a recursive formula. To compute, e.g., S_{20} one needs to successively compute $S_1, S_2, S_3, \ldots, S_{19}$.

Example 3.6 We can compute S_4 using Theorem 3.1, since

$$4 \cdot 2^{\binom{4}{2}} = \binom{4}{1} S_1 \cdot 1 \cdot 2^{\binom{3}{2}} + \binom{4}{2} S_2 \cdot 2 \cdot 2^{\binom{2}{2}} + \binom{4}{4} S_3 \cdot 3 \cdot 2^{\binom{1}{2}} + \binom{4}{4} S_4 \cdot 4 \cdot 2^{\binom{0}{2}}$$

we have, using the known values of S_1, S_2 and S_3,

$$4 \cdot 64 \;=\; 4 \cdot 1 \cdot 1 \cdot 8 \;+\; 6 \cdot 1 \cdot 2 \cdot 2 \;+\; 4 \cdot 4 \cdot 3 \cdot 1 \;+\; 1 \cdot S_4 \cdot 4 \cdot 1.$$

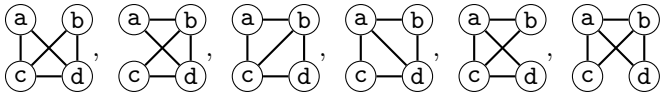
Hence $S_4 = 64 - 8 - 12 = 38$.

Exercise 2 Find all connected simple graphs of order 4.

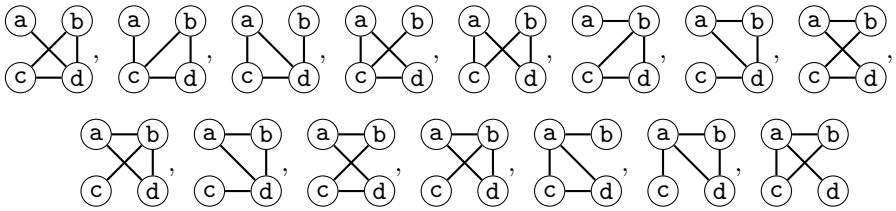
Solution of Exercise 2 The number of connected graphs on 4 vertices, S_4, is 38. These 38 graphs are:

a) 1 complete graph;

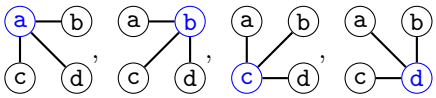
b) $6 = \binom{6}{1}$ graphs with one edge missing;



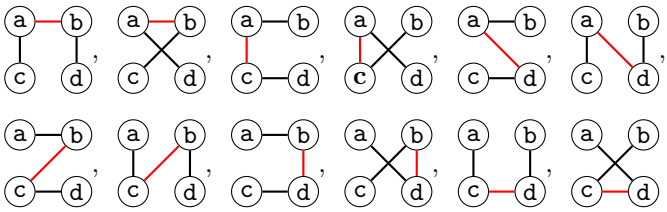
c) $15 = \binom{6}{2}$ graphs with two edges missing;



d) 4 graphs with one vertex with degree 3, connected to the other 3 vertices of degree 1;



e) $12 = 6 \cdot 2$ graphs that are paths of length 3, since we have 6 possibilities for the middle segment and 2 possibilities for the end segments;



3.4 Walks

Given a simple graph $G = (V, E)$, a sequence $(v_i)_{i=0}^{k}$ of vertices $v_i \in V$ is called a *walk* (or *k-walk*) in G if for every $1 \leq j \leq k$ we have $\{v_{j-1}, v_j\} \in E$.

A walk is *closed* if its initial and terminal vertices are the same. A walk where all its edges are distinct is called a *trail*.

The notions of *length*, *endpoints*, *inner vertices* in a walk are defined analogously as in a path. Nevertheless, note that, contrary to paths (resp., cycles), vertices in a walk (resp., closed walk) can be repeated.

Example 3.7 Let G be the graph in Figure 3.4. The sequence $(\mathtt{d}, \mathtt{a}, \mathtt{c}, \mathtt{a}, \mathtt{b})$ is a walk in G but not a path. The sequence $(\mathtt{a}, \mathtt{c}, \mathtt{b}, \mathtt{c}, \mathtt{a})$ is a closed walk in G but not a cycle. The sequence $(\mathtt{d}, \mathtt{a}, \mathtt{b}, \mathtt{c})$ is both a walk and a path. The sequence $(\mathtt{d}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{a})$ is a trail (but not a path nor a cycle).

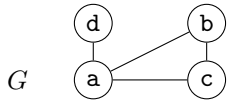


Figure 3.4: A graph G.

Connectedeness of a pair of vertices in a graph through a walk is an equivalence relation. Indeed:

- every vertex is connected by a 0-walk to itself;
- if there is a walk $W = (v_i)_{i=0}^{k}$ from a to b, then the reverse walk $\widetilde{W} = (v_{k-i})_{i=0}^{k}$ connects b to a;
- if a, b are connected by a walk $W_1 = (v_i)_{i=0}^{k}$ and v, w are connected by a walk $W_2 = (u_i)_{i=0}^{h}$, then there is a walk from u to w obtained $W_1W_2 = (w_i)_{i=0}^{k+h}$, where $w_i = v_i$ for $0 \leq i \leq k$ and $w_i = u_{i+k}$ for $k+1 \leq k \leq k+h$.

The equivalence class of a vertex v determined by the connectedness relation gives the component of the graph containing the vertex v.

We can also consider walks in non simple graphs.

Example 3.8 Let G be the graph in Figure 3.5. A walk in G is given by

$$(h, j, j, i, e, f, f) = (\mathtt{b} \xrightarrow{h} \mathtt{c} \xrightarrow{j} \mathtt{c} \xrightarrow{j} \mathtt{c} \xrightarrow{i} \mathtt{b} \xrightarrow{e} \mathtt{a} \xrightarrow{f} \mathtt{d} \xrightarrow{f} \mathtt{a}).$$

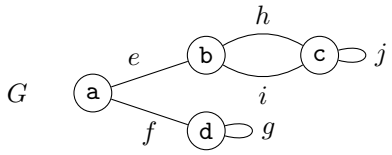


Figure 3.5: A non simple graph G.

3.5 Trees

A graph that does not have any cycle in it is called *acyclic*. An acyclic graph is also called a *forest*. A connected forest, i.e., a graph that is both connected and acyclic, is called a *tree*. Vertices of degree 1 in a tree are called *leaves*.

Example 3.9 The graph G on the left of Figure 3.6 is not acyclic. The graph H on the centre is a forest. The graph $K_{1,5}$, on the right, is a tree.

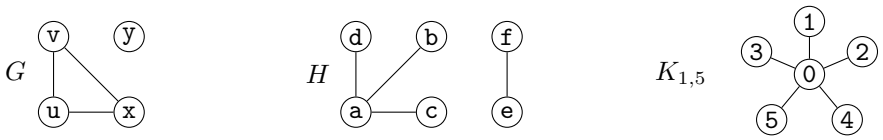


Figure 3.6: A graph with a cycle (left) a forest (centre) and a tree (right).

The trivial graph has exactly one leaf. Every tree having more than one vertex has at least 2 leaves. Indeed, it is enough to consider the endpoints of a longest path.

Theorem 3.2 *Let $G = (V, E)$ be a graph s.t. $d_G(v) \geq 2$ for every $v \in V$. Then G contains a cycle.*

Proof. The statement is clear if G has a loop or two parallel edges. Thus, WLOG, let us suppose that G is simple.

Let $P = (v_0, v_1, \ldots, v_{k-1}, v_k)$ be a longest path in G. Since $d_G(v_k) \geq 2$, then there exists $v \in \mathrm{N}_G(v_k)$ with $v \neq v_{k-1}$. If $v \notin P$, then $P + \{v\}$ would be a path longer than P, which is a contradiction. Thus, $v = v_i \in P$, which implies that $(v_i, \ldots, v_{k-1}, v_k)$ is a cycle in G. ∎

A graph $G = (V, E)$ is *minimally connected* if it is connected but $G \setminus e$ is disconnected for every $e \in E$. It is *maximally acyclic* if it is acyclic but for any two vertices $u, v \in V$ s.t. $\{u, v\} \notin E$ we have that $G + \{u, v\}$ contains a cycle.

Theorem 3.3 *The following are equivalent for a graph $G = (V, E)$.*

1. *G is a tree.*
2. *Any two vertices in V are linked by a unique path in G.*
3. *G is minimally connected.*
4. *G is maximally acyclic.*

Proof. Exercise. ∎

Corollary 2 (Euler's formula for trees) *A connected graph of order n is a tree if and only if it has size $n - 1$.*

The hypothesis of connectedness is necessary.

Example 3.10 Let us consider the graphs G, H and $K_{1,5}$ in Example 3.9. The graph H, a forest but not a tree, has order 6 and size 4. The tree $K_{1,5}$ has order 5 and size 4. The graph G, which is not a forest, has order 4 and size 3.

Theorem 3.4 (Cayley's formula) *Let $\#V = n$. The number of trees with set of vertices V is n^{n-2}.*

Example 3.11 The only tree with one vertex is the trivial one. The only tree with two vertices $\mathtt{a}, \mathtt{b}$ is the one with one edge $\{\mathtt{a}, \mathtt{b}\}$. If the set of vertices is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ we have three possible trees (see Figure 3.7). According to Theorem 3.4 there are 16 labelled trees on $V = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ (which ones?).

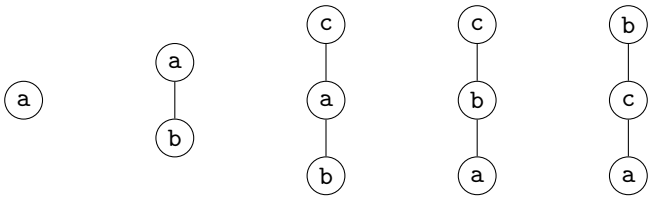


Figure 3.7: Possible trees on 1, 2 and 3 vertices.

A *rooted tree* $T(r)$ is a tree $T = (V, E)$ with a specified vertex $r \in V$ called the *root* of T. Given a vertex v in a rooted tree $T(r)$ different than r, we say that a vertex in $\mathrm{N}_T(v)$ on the unique path between r and v is a *parent* of v; all vertices in this path are called *ancestors* of v. Similarly, a vertex is a *child* of its parent, and u is a *descendant* of v if v is an ancestor of u.

We call *level* $\ell_T(v)$ of a vertex v in a rooted tree $T(r)$ the distance $d_T(r, v)$ (i.e., the length of the path from r to v in the tree). The root is the only vertex with level 0. The *height* of a rooted tree is the maximal level.

Note that when dealing with rooted trees we call *leaf* a vertex of degree 1 that has no children. Hence, except for the trivial tree with one vertex, the root is (in this context) never a leaf.

Example 3.12 The graph in Figure 3.8 is a rooted tree with root `r`. The vertices `a,b,c` are children of `r`. The vertex `h` is a child of `e` and a descendant of `a` (hence `e` is a parent of `h` and `a` an ancestor of `h`). The vertices `d,e,f,g` have level 2, while the height of the tree is 3 (`h` is a vertex of maximal level).

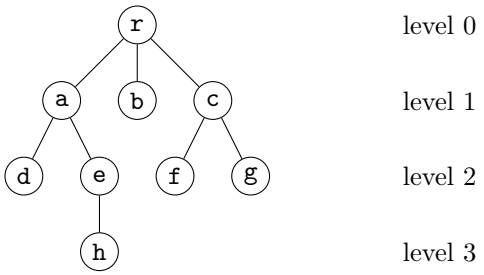


Figure 3.8: A rooted tree.

An *ordered tree* $T(r, \preceq)$ is a rooted tree $T(r) = (V, E)$ with root r and a partial order $\preceq$ on V giving the order of children of each vertex. We usually

represent it by drawing the children of a node from left to right starting from the smallest.

Example 3.13 Let T be the rooted tree in Example 3.12 and represented in Figure 3.8. Then, $T(\mathtt{r}, \preceq)$, with the partial order $\preceq$ defined by

$$\mathtt{a} \prec \mathtt{b} \prec \mathtt{c}, \qquad \mathtt{d} \prec \mathtt{e} \qquad \text{and} \qquad \mathtt{f} \prec \mathtt{g}$$

is a ordered tree.

3.6 Adjacency matrix

The *adjacency matrix* A_G of a simple graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the $n \times n$ matrix defined by

$$[A_G]_{i,j} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise} \end{cases}.$$

Clearly A_G is a non-negative symmetric matrix.

Example 3.14 Let $G = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}, \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{d}\}, \{\mathtt{b}, \mathtt{c}\}\})$. The adjacency matrix of G is

$$A_G = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

Note that the definition of A_G can be generalised to non-simple graphs by replacing 1 with the number of parallel edges and counting each loop twice.

Theorem 3.5 *Let $k \in \mathbb{N}_0$ and let A_G be the adjacency matrix of a graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$. Then $\left[A_G^k\right]_{i,j}$ is the number of walks of length k with endpoints v_i and v_j in G.*

Proof. The case $k = 0$ is trivial since we have $A_G^0 = I_n$, and the only walks of length 0 are the ones connecting a vertex to itself.

Let us prove it by induction on $k \geq 1$.

($k = 1$) It follows directly from the definition of adjacency matrix, since a walk of length 1 is exactly an edge.

($k - 1 \to k$) The (i, j)-element of A_G^k is given by

$$\left[A_G^k\right]_{i,j} = \left[A_G^{k-1} \cdot A_G\right]_{i,j} = \sum_{\ell=1}^{n} \left[A_G^{k-1}\right]_{i,\ell} \cdot [A_G]_{\ell,j} = \sum_{\substack{\ell=1 \\ \{v_\ell, v_j\} \in E}}^{n} \left[A_G^{k-1}\right]_{i,\ell}$$

but $\left[A_G^{k-1}\right]_{i,\ell}$ is, by induction hypothesis, the number of walks of length $k-1$ with endpoints v_i and v_ℓ. Since we are summing only the walks that can be prolonged from v_ℓ to v_j (for every possibile choice of v_ℓ) we can conclude (see Figure 3.9).

■

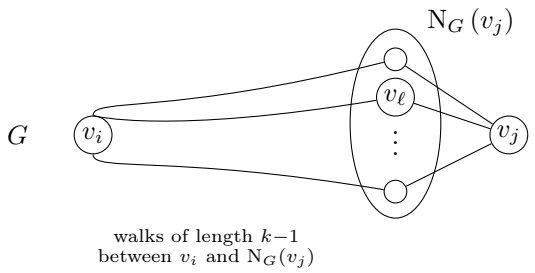


Figure 3.9: Illustration of the proof of Theorem 3.5.

Corollary 3 *A graph $G = (V, E)$ is connected if and only if $\sum_{k=0}^{\#V-1} A_G^k$ is a positive matrix.*

Proof.

(⇒) Let $v_i, v_j \in V$. Since G is connected the two vertices are linked in G. Thus, there is a walk of length $k \leq \#V - 1$ in G with endpoints v_i and v_j. Hence, $\left[A_G^k\right]_{i,j} \geq 1$.

(⇐) Let i, j with $0 < i, j < n$. Since $\left[\sum_{k=0}^{n-1} A_G^k\right]_{i,j} > 0$, there exists ℓ, with $0 \leq \ell \leq n-1$ such that $\left[A_G^\ell\right]_{i,j} \geq 1$. Thus, there is a walk of length ℓ in G with endpoints v_i and v_j.

■

Example 3.15 Let A_G be the adjacency matrix seen in Example 3.14. We have

$$A_G^2 = \begin{pmatrix} 3 & 1 & 1 & 0 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \quad \text{and} \quad A_G^3 = \begin{pmatrix} 2 & 4 & 4 & 3 \\ 4 & 3 & 3 & 2 \\ 4 & 3 & 2 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix}.$$

Indeed, we have, e.g., three walks of length 2 from `a` to `a`, namely `(a,b,a)`, `(a,c,a)` and `(a,d,a)`; and four walks of length 3 from `a` to `c`, namely: `(a,b,a,c)`, `(a,c,a,c)`, `(a,c,b,c)` and `(a,d,a,c)`.

Notice also that

$$\sum_{k=0}^{3} A_G^k \;=\; I_4 + A_G + A_G^2 + A_G^3 \;=\; \begin{pmatrix} 6 & 6 & 6 & 4 \\ 6 & 6 & 5 & 3 \\ 6 & 5 & 5 & 2 \\ 4 & 3 & 2 & 2 \end{pmatrix}.$$

is a positive matrix.

Chapter 4

Isomorphism of graphs

Two graphs $G = (V, E)$ and $H = (U, F)$ are *identical* if $V = U$ and $E = F$.

They are *isomorphic*, denoted $G \sim H$ (or $G \cong H$), if there exists a bijection $\varphi : V \to U$, called an *isomorphism*, such that for every $u, v \in V$

$$\{u, v\} \in E \quad \Leftrightarrow \quad \{\varphi(u), \varphi(v)\} \in F.$$

Clearly when $\varphi = \mathrm{id}$, the two graphs are identical.

Example 4.1 The graphs $G = \left(V, \binom{V}{2}\right)$ and $H = \left(U, \binom{U}{2}\right)$ with $V = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and $U = \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}\}$ are isomorphic (see Figure 4.1), with bijection, e.g.,

$$\varphi : \begin{cases} \mathtt{a} \mapsto \mathtt{x} \\ \mathtt{b} \mapsto \mathtt{y} \\ \mathtt{c} \mapsto \mathtt{z} \\ \mathtt{d} \mapsto \mathtt{t} \end{cases}$$

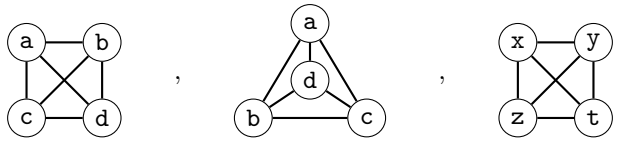


Figure 4.1: Three isomorphic graphs (the first two are identical).

Thus, two graphs are isomorphic if they are "the same" up to relabelling the vertices. Clearly, degree and distance are preserved under isomorphism.

Proposition 1 *Let $G = (V, E)$ and $H = (U, F)$ be two isomorphic graphs with isomorphism φ. Then, for every $u, v \in V$ we have*

1. $d_G(v) = d_H(\varphi(v))$,

2. $d_G(u, v) = d_H(\varphi(u), \varphi(v))$.

Proof.

1. From the definition of isomorphism it follows that $u \in \mathrm{N}_G(v)$ if and only if $\varphi(u) \in \mathrm{N}_H(\varphi(v))$. Thus the cardinality of the two sets, i.e., the two degrees $d_G(v)$ and $d_H(\varphi(v))$, coincide.

2. Exercise.

■

Example 4.2 The graphs represented in Figure 4.2 are isomorphic. A possible bijection is

$$\varphi : \begin{cases} \mathtt{a} \mapsto 1 \\ \mathtt{b} \mapsto 4 \\ \mathtt{c} \mapsto 2 \\ \mathtt{d} \mapsto 5 \\ \mathtt{e} \mapsto 3 \\ \mathtt{f} \mapsto 6 \end{cases} .$$

One can check, e.g., that $d_G(\mathtt{a}) = d_H(1) = 3$, and that $d_G(\mathtt{a}, \mathtt{e}) = d_H(1, 3) = 2$.

Figure 4.2: Two isomorphic graphs.

We do not normally distinguish between isomorphic graphs. When dealing with a graph up to isomorphism we can forget the labels in the drawing.

If two graphs are isomorphic, then clearly they have the same order and the same size. The opposite is not true.

Exercise 3 Show that the two unlabeled graphs in Figure 4.3, which have both order 4 and size 3, are not isomorphic.

Figure 4.3: Two non-isomorphic graphs with the same order and size.

Solution of Exercise 3 Let $G = (V, E)$ and $H = (U, F)$ be two isomorphic graph with isomorphism φ. Each vertex $v \in V$ is sent to a vertex $\varphi(v) \in U$ having the same degree. Indeed, φ is, in particular, a bijection between $\mathrm{N}_G(v)$ and $\mathrm{N}_H(\varphi(v))$. Thus,

$$d_G(v) = \#\mathrm{N}_G(v) = \#\mathrm{N}_H(\varphi(v)) = d_H(\varphi(v)).$$

In the first graph all vertices have degree 2, while in the second one we have a vertex of degree 3, two vertices of degree 2 and one vertex of degree 1. Thus G and H cannot be isomorphic.

The graph isomorphism problem (deciding whether two graphs are isomorpic) is $\mathcal{NP}$, but it is not known to belong to either $\mathcal{P}$ or $\mathcal{NP}$-complete.

4.1 Tree isomorphism problem

We will see a polynomial-time algorithm for isomorphism of trees. To do it we proceed in three steps, giving three different algorithms:

1. algorithm for isomorphism of ordered trees;
2. algorithm for isomorphism of rooted trees;
3. general algorithm for isomorphism of trees.

Two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are isomorphic, denoted

$$T_1 \sim T_2,$$

if the two trees are isomorphic as graphs, i.e., there exists a bijective mapping $\varphi : V_1 \to V_2$ such that for every $u, v \in V_1$

$$\{u, v\} \in E_1 \quad \Leftrightarrow \quad \{\varphi(u), \varphi(v)\} \in E_2.$$

Two rooted trees $T_1(r_1)$ and $T_2(r_2)$ are isomorphic, denoted

$$T_1(r_1) \sim_R T_2(r_2),$$

if $T_1 \sim T_2$ with isomorphism φ, and the morphism is sending root to root, i.e., $\varphi(r_1) = r_2$.

Two ordered trees $T_1(r_1, \preceq_1)$ and $T_2(r_2, \preceq_2)$ are isomorphic, denoted

$$T_1(r_1, \preceq_1) \sim_O T_2(r_2, \preceq_2),$$

if $T_1(r_1) \sim_R T_2(r_2)$ with isomorphism φ, and the mapping φ preserves the partial order of children of each vertex, i.e., for every $u, v \in V_1$

$$u \preceq_1 v \quad \Leftrightarrow \quad \varphi(u) \preceq_2 \varphi(v).$$

Example 4.3 The two trees T_1 and T_2 in Figure 4.4 are isomorphic. Indeed an isomorphism is given by

$$\varphi : \begin{cases} \mathtt{a} \mapsto \mathtt{E} \\ \mathtt{b} \mapsto \mathtt{D} \\ \mathtt{c} \mapsto \mathtt{C} \\ \mathtt{D} \mapsto \mathtt{B} \\ \mathtt{e} \mapsto \mathtt{A} \end{cases} .$$

As rooted trees, though, $T_1(\mathtt{d})$ and $T_2(\mathtt{C})$ are not isomorphic since there is no isomorphism sending $\mathtt{d}$ to $\mathtt{C}$ (why?)

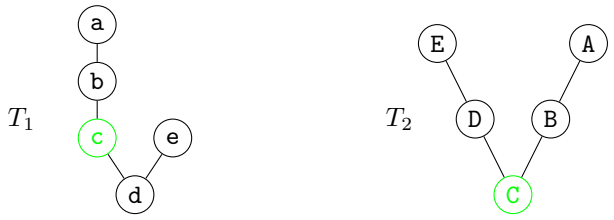


Figure 4.4: $T_1 \sim T_2$ but $T_1(\mathtt{d}) \not\sim_R T_2(\mathtt{C})$.

Example 4.4 The two rooted trees $T_3(\mathtt{r})$ and $T_4(\mathtt{R})$ in Figure 4.5 are isomorphic. Indeed it is enough to consider the bijection

$$\varphi : \begin{cases} \mathtt{r} \mapsto \mathtt{R} \\ \mathtt{a} \mapsto \mathtt{C} \\ \mathtt{b} \mapsto \mathtt{B} \\ \mathtt{c} \mapsto \mathtt{D} \\ \mathtt{d} \mapsto \mathtt{A} \end{cases} .$$

They are not isomorphic as ordered trees (why?)

The two ordered trees $T_4(\mathtt{R}, \preceq_4)$ and $T_5(\mathtt{s}, \preceq_5)$ in the same figure are isomorphic (find the isomorphism!).

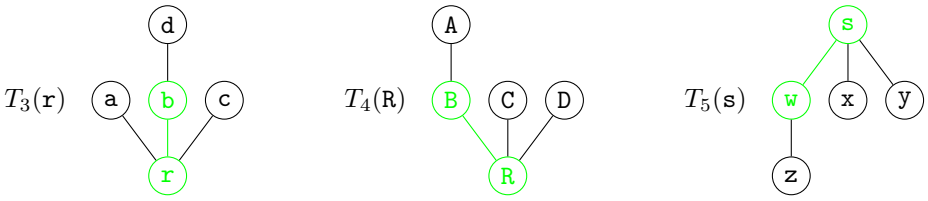


Figure 4.5: Three ordered trees.

4.2 Isomorphism of ordered trees

We will code a ordered tree by a sequence of 0s and 1s such that:

- the encoding uses only properties fixed by the mapping φ;
- the code is uniquely (up to isomorphism) decodable.

These two properties of the encoding imply that two ordered trees are isomorphic if and only if their codes are the same.

We code an ordered tree $T(r, \preceq)$ in the following way:

- each leaf gets the code $\mathtt{01}$;
- given a node v with children $v_1 \prec v_2 \prec \ldots \prec v_k$ having encoding $C(v_1), C(v_2), \ldots, C(v_k)$, we define $C(v) = \mathtt{0}\, C(v_1)C(v_2)\cdots C(v_k)\, \mathtt{1}$;
- $C(T) = C(r)$, i.e., the code of a tree coincides with the code of its root.

Example 4.5 Let $T_3(\mathtt{r}, \preceq_3)$ be the ordered tree in Example 4.4. Then

$$C(\mathtt{a}) = C(\mathtt{c}) = C(\mathtt{d}) = \mathtt{01}; \qquad C(\mathtt{b}) = \mathtt{0}.C(\mathtt{d}).\mathtt{1} = \mathtt{0011};$$

and

$$C(T_3) = C(\mathtt{r}) = \mathtt{0}.C(\mathtt{a}).C(\mathtt{b}).C(\mathtt{c}).\mathtt{1} = \mathtt{0010011011}.$$

One can check that $C(T_4) = C(T_5) = \mathtt{0001101011} \neq C(T_3)$.

We can also decode a code C of an ordered tree $T(r, \prec)$. If $C(T) = \mathtt{0}w\mathtt{1}$, then w corresponds to the concatenations of codes of ordered trees with roots corresponding to the children of r.

In particular, if $w = C(v_1)C(v_2)\cdots C(v_k)$, then $C(v_1)$ is the shortest prefix of w with the same number of 0s and 1s. We can cut $C(v_1)$ and continue analogously.

Example 4.6 Let $C(T_3) = \mathtt{0}.\mathtt{01001101}.\mathtt{1}$ as in Example **??**. The shortest prefix of $w = \mathtt{01\,0011\,01}$ having the same number of 0s and 1s is $C(v_1) = \mathtt{01}$. Similarly $C(v_2) = \mathtt{0011}$ and $C(v_3) = \mathtt{01}$. The vertex v_2 has a child v_4 with encoding $C(v_4) = \mathtt{01}$. Thus v_1, v_3, v_4 are leaves and the T_3 is isomorphic to the ordered tree $T = (\{r, v_1, v_2, v_3, v_4\}, \{\{r, v_1\}, \{r, v_2\}, \{r, v_3\}, \{v_2, v_4\}\})$.

4.3 Isomorphism of rooted trees

We would like to modify the encoding of ordered trees so that it can be used for deciding the isomorphism of rooted trees.

Which properties used in the previous case can we also use in the case of rooted trees?

1) a vertex is a root, ✓

2) a vertex is a leaf, ✓

3) a vertex is a child of a vertex, ✓

4) children of the same vertex are ordered. ✘

So we have to compensate for the fact that children of a vertex do not have a fixed (under isomorphism) order. To do that we:

- choose an ordering $\leq$ on strings over $\{\mathtt{0},\mathtt{1}\}$ (e.g., lexicographical or radix),
- if v has children $v_1, v_2, \ldots, v_k$ such that $C(v_1) \leq C(v_2) \leq \ldots \leq C(v_k)$, then we define $C(v) = \mathtt{0}\, C(v_1)C(v_2)\cdots C(v_k)\, \mathtt{1}$.

Example 4.7 Let $T_1(\mathtt{d})$ and $T_2(\mathtt{C})$ be the rooted tree in Example 4.3. We have

$$C(\mathtt{a}) = C(\mathtt{e}) = \mathtt{01}, \quad C(\mathtt{b}) = \mathtt{0011} \quad \text{and} \quad C(\mathtt{C}) = \mathtt{000110}.$$

If we use the lexicographic order $\leq_{lex}$ we have

$$C(\mathtt{c}) = \mathtt{000111} <_{lex} \mathtt{01} = C(\mathtt{e}),$$

so

$$C(T_1) = C(\mathtt{d}) = \mathtt{0}.C(\mathtt{c}).C(\mathtt{e}).\mathtt{1} = \mathtt{0000111011}$$

Similarly, we have

$$C(\mathtt{A}) = C(\mathtt{E}) = \mathtt{01}, \quad C(\mathtt{B}) = C(\mathtt{D}) = \mathtt{0011}, \quad \text{and} C(\mathtt{D}) = \mathtt{0011} = C(\mathtt{B}).$$

Hence, $C(T_2) = C(\mathtt{C}) = \mathtt{0001100111} \neq C(T_1)$.

On the other hand, one can check that the three trees in Example 4.4 have all the same encoding (using the lexicographic order) $\mathtt{0001101011}$.

Exercise 4 Prove, using the radix order, that:

a) $T_1(\mathtt{d}) \not\sim_R T_2(\mathtt{c})$,

b) $T_3(\mathtt{r}) \sim_R T_4(\mathtt{R}) \sim_R T_5(\mathtt{s})$.

Solution of Exercise 4 Let us use the radix order to compare coding of children of the same node.

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

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

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

The nodes of the rooted tree $T_2(\mathtt{C})$ have coding

$$C(\mathtt{E}) = C(\mathtt{A}) = \mathtt{01}, \qquad C(\mathtt{D}) = C(\mathtt{B}) = \mathtt{0.01.1}$$

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

Since $C(\mathtt{d}) \neq C(\mathtt{C})$ we can conclude that $T_1(\mathtt{d}) \not\sim_R T_2(\mathtt{c})$,

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

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

and

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

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

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

which implies $T_3(\mathtt{r}) \sim_R T_4(\mathtt{R}) \sim_R T_5(\mathtt{s})$.

4.4 Isomorphism of general trees

Again, we would like to adapt the encoding of rooted trees to decide isomorphism of general trees. Problems in this case, i.e., in switching from rooted to general trees, are much more serious. Indeed, we don't have the root, thus we cannot establish the child/parent relationship between nodes.

Therefore, we have to find a suitable (i.e., preserved under isomorphism) substitute for the root. To do that we use the centre of a tree.

While for a general graph $G = (V, E)$ the centre of G can be any subset of V (see, e.g., $\mathcal{C}(C_3)$ where the centre is the entire set of vertices), in the case of trees the situation is simpler.

Theorem 4.1 *Let $T = (V, E)$ be a tree. Then either $\mathcal{C}(T) = \{x\}$ or $\mathcal{C}(T) = \{x, y\}$, with $x, y \in V$. Moreover, in the latter case we have $\{x, y\} \in E$.*

Example 4.8 Let T_1 and T_2 be the trees in Example 4.3, and T_3, T_4, T_5 the ones in Example 4.4. One can check that

$$\mathcal{C}(T_1) = \{\mathtt{c}\},\ \ \mathcal{C}(T_2) = \{\mathtt{C}\},\ \ \mathcal{C}(T_3) = \{\mathtt{b}, \mathtt{r}\},\ \ \mathcal{C}(T_4) = \{\mathtt{B}, \mathtt{R}\},\ \ \mathcal{C}(T_5) = \{\mathtt{w}, \mathtt{s}\}.$$

Since the only graph property used in the definition of centre of a tree is the distance, and since distance of vertices is preserved under the isomorphism, we can use the centre of a tree instead of the root. In particular, when the centre is a singleton we consider it as the root of the tree. When the centre is a set of cardinality two, we use these two vertices as roots of two new trees obtained as

subgraphs of the original one.

Algorithm 2: `IsoTrees`(T_1, T_2)

Input: Two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$
Output: `TRUE` if $T_1 \sim T_2$; `FALSE` if not

Find centres $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$
if $\#\mathcal{C}(T_1) \neq \#\mathcal{C}(T_2)$ **then**
  return `FALSE` // $T_1 \not\sim T_2$
else if $\Big(\mathcal{C}(T_1) = \{x\}$ *and* $\mathcal{C}(T_2) = \{y\}\Big)$ **then**
  return `IsoRootTrees`$(T_1(x), T_2(y))$ // $T_1 \sim T_2 \Leftrightarrow T_1(x) \sim_R T_2(y)$
else
  Let $\mathcal{C}(T_1) = \{x_1, x_2\}$ and $\mathcal{C}(T_2) = \{y_1, y_2\}$.
  We use 4 rooted trees:
  $T_1^{(1)}(x_1)$: the component of $T_1 \setminus \{x_1, x_2\}$ containing x_1;
  $T_1^{(2)}(x_2)$: the component of $T_1 \setminus \{x_1, x_2\}$ containing x_2;
  $T_2^{(1)}(y_1)$: the component of $T_2 \setminus \{y_1, y_2\}$ containing y_1.
  $T_2^{(2)}(y_2)$: the component of $T_2 \setminus \{y_1, y_2\}$ containing y_2.
  return $\Big($ $\big($`IsoRootTrees`$(T_1^{(1)}(x_1), T_2^{(1)}(y_1))$ `and`
  `IsoRootTrees`$(T_1^{(2)}(x_2), T_2^{(2)}(y_2))\big)$ `or`
  $\big($`IsoRootTrees`$(T_1^{(1)}(x_1), T_2^{(2)}(y_2))$ `and`
  `IsoRootTrees`$(T_1^{(2)}(x_2), T_2^{(1)}(y_1))\big)$ $\Big)$
  // $T_1 \sim T_2 \Leftrightarrow \Big(T_1^{(1)}(x_1) \sim_R T_2^{(1)}(y_1) \wedge T_1^{(2)}(x_2) \sim_R T_2^{(2)}(y_2)\Big)$
  $\vee \Big(T_1^{(1)}(x_1) \sim_R T_2^{(2)}(y_2) \wedge T_1^{(2)}(x_2) \sim_R T_2^{(1)}(y_1)\Big)$

Example 4.9 Let T_1, T_2, T_3, and T_4 as in Examples 4.3 and 4.4.

- Clearly $T_1 \not\sim T_3$ since $\#\mathcal{C}(T_1) \neq \#\mathcal{C}(T_3)$.
- We have $T_1 \sim T_2$ since one can easily check that $T_1(\mathtt{c}) \sim_R T_2(\mathtt{C})$.
- We have $T_3 \sim T_4$ since one can build the rooted trees $T_3^{(1)}(\mathtt{b}), T_3^{(2)}(\mathtt{r}), T_4^{(1)}(\mathtt{B})$ and $T_4^{(2)}(\mathtt{R})$ (see Figure 4.6) and easily check that $T_3^{(1)}(\mathtt{b}) \sim_R T_4^{(1)}(\mathtt{B})$ and $T_3^{(2)}(\mathtt{r}) \sim_R T_4^{(2)}(\mathtt{R})$.

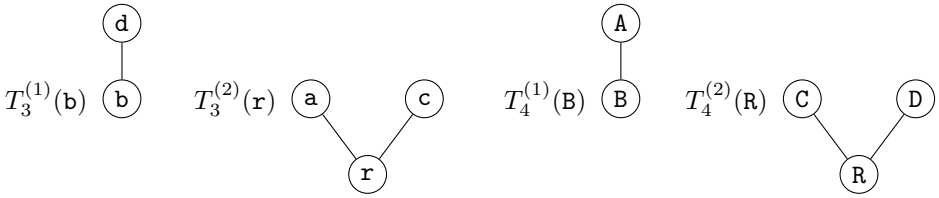


Figure 4.6: The trees obtained by deleting the central edges in T_3 and T_4.

Chapter 5

Directed graphs and graph traversal

5.1 Directed graphs

A *directed graph* or *digraph* (or *oriented graph*) is a pair $D(V, E)$, where V is a (finite) set, and $a \in V \times V$ for every $a \in E$.

Elements of V are called *vertices*, while elements of E are called *arcs* (or *directed edges*, or *oriented edges*).

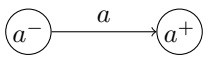
Given an arc $a = (u, v) \in E$, we call u the *tail* of a and v the *head* of a. Both u, v are the *ends* of a. The vertex u is called an *in-neighbour* of v, and the vertex v a *out-neighbour* of u. It is thus natural to define

$$\mathrm{N}_G^-(v) = \{u \mid (u, v) \in E\} \quad \text{and} \quad \mathrm{N}_G^+(v) = \{u \mid (v, u) \in E\}\,.$$

The *in-degree* and the *out-degree* of a vertex v are defined respectively as

$$d_G^-(v) = \#\mathrm{N}_G^-(v) \quad \text{and} \quad d_G^+(v) = \#\mathrm{N}_G^+(v).$$

Given an arc $a = (u, v)$, we also denote by $a^- = u$ its starting vertex and by $a^+ = v$ its ending vertex.



A graph $G = (V', E')$ is the *underlying graph* of a digraph D if $V' = V$ and $\{u, v\} \in E'$ if $(u, v) \in E$.

Example 5.1 The digraph $D = (V, E)$, with vertices $V = \{\mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}\}$ and arcs $E = \{(\mathtt{u}, \mathtt{v}), (\mathtt{u}, \mathtt{w}), (\mathtt{v}, \mathtt{x}), (\mathtt{x}, \mathtt{v}), (\mathtt{y}, \mathtt{v}), (\mathtt{y}, \mathtt{y})\}$ is shown in Figure 5.1 on the left. One can check that $d_G^-(\mathtt{v}) = 3$ and $d_G^+(\mathtt{v}) = 1$.

The underlying graph G of D is shown on the same figure on the right.

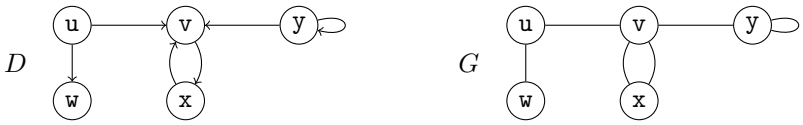


Figure 5.1: A digraph D (left) and its underlying graph G (right).

An *orientation* of a graph $G = (V, E)$ is a digraph obtained by choosing a direction for every edge in E. The *converse* of a digraph $D = (V, E)$ is the digraph $D' = (V', E')$ with set of vertices $V = V'$ and seet of arcs given by $(u, v) \in E \Leftrightarrow (v, u) \in E'$.

Clearly, both a digraph and its converse are orientations of the same graph.

Example 5.2 An orientation of the graph $G = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\}\})$ is the digraph $D = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{(\mathtt{a}, \mathtt{b}), (\mathtt{c}, \mathtt{b})\})$. The converse of D is the digraph $D' = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{(\mathtt{b}, \mathtt{a}), (\mathtt{b}, \mathtt{c})\})$ (see Figure 5.2).

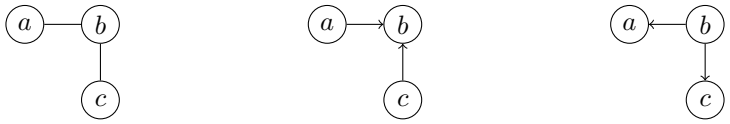


Figure 5.2: A graph (left) and two of its possible orientations (centre and right).

5.2 Branching

An orientation of a rooted tree such that every vertex but the root has in-degree 1 is called a *branching*.

It is clear that (u, v) is an arc in a branching if and only if v is a child of u in the correspective tree.

Example 5.3 Let $T(\mathtt{r})$ be the rooted tree shown on the left of Figure 5.3. A branching D of $T(\mathtt{r})$ is shown on the right of the same Figure 5.3.

Figure 5.3: A rooted tree (left) and its branching (right).

5.3 Graph Traversal Algorithms (GTS)

A *graph traversal* is the process of visiting each vertex of a (connected) graph. It is a key ingredient of many graph algorithms.

Tarry's algorithm. The oldest known GTS algorithm, Tarry's algorithm, uses two rules while traversing the graph:

(R1) every edge can be used at most once in every direction;

(R2) the edge of the first arrival to a vertex can be used (in the opposite direction) only if there is no other possibility.

Theorem 5.1 *Tarry's traversal is finite.*

Moreover, if there is no way to continue with the traversal, we have returned to the starting vertex, and, moreover, every edge of G has been used exactly twice.

Example 5.4 Let G be the graph in Figure 5.4. If the starting point is the vertex `a`, a possible traversal is the walk (`a,d,b,c,f,e,d,g,h,i`).

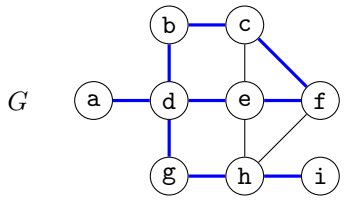


Figure 5.4: The walk (`a,d,b,c,f,e,d,g,h,i`) is a traversal.

Note that in a traversal we do not necessarily use all edges, and we may visit the same vertex more than once.

Today, the two primary graph traversal (or searching) algorithms are:

- *Breadth-First Search* (using queues).
- *Depth-First Search* (using stacks),

The following algorithm can be used with D either a queue, for BFS, or a stack, for DFS.

Algorithm 3: `Traversal(`G`)`

Input: graph G
Init: choose a vertex and put it in D
1 **repeat**
2 | pop a vertex from D
3 | process it
4 | put all its unprocessed neighbours in D
5 **until** D *is empty*

Example 5.5 A possible traversal of the graph G in Figure 5.4 starting from a is given by the queue (using BFS):

$$\begin{aligned}\emptyset &\to (\mathtt{a}) \to (\mathtt{a},\mathtt{d}) \to (\mathtt{d},\mathtt{b},\mathtt{e},\mathtt{g}) \to (\mathtt{b},\mathtt{e},\mathtt{g},\mathtt{c}) \to (\mathtt{e},\mathtt{g},\mathtt{c},\mathtt{f},\mathtt{h})\\ &\to (\mathtt{g},\mathtt{c},\mathtt{f},\mathtt{h}) \to (\mathtt{c},\mathtt{f},\mathtt{h}) \to (\mathtt{f},\mathtt{h}) \to (\mathtt{h},\mathtt{i}) \to (\mathtt{i}) \to \emptyset.\end{aligned}$$

Example 5.6 A possible traversal of the graph G in Figure 5.4 starting from a is given by the stack (using DFS):

$$\begin{aligned}\emptyset &\to [\mathtt{a}] \to [\mathtt{a},\mathtt{d}] \to [\mathtt{a},\mathtt{d},\mathtt{e}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c}]\\ &\to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c},\mathtt{b}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}]\\ &\to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h},\mathtt{g}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h},\mathtt{i}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}]\\ &\to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \to [\mathtt{a},\mathtt{d},\mathtt{e}] \to [\mathtt{a},\mathtt{d}] \to [\mathtt{a}] \to \emptyset.\end{aligned}$$

We can use a GTS algorithm, e.g., to find the components of a (not necessarily connected) graph G:

1. use DFS from a starting point (i.e., any vertex) $v \in V$;
2. if the algorithm stops, then all vertices are processed: we have the component of G containing v.

5.4 Tree-Search Algorithms: BFS vs DFS

Let $G = (V, E)$ be a graph. A *spanning tree* of G is a spanning graph of G that is a tree, i.e., a tree $T = (V, E')$ with $E' \subseteq E$.

A graph that is not connected cannot contain any spanning tree.

On the other hand, any connected graph contains at least one spanning tree.

Example 5.7 Let G be the graph in Figure 5.5 on the left. Two possible spanning trees are represented on the right of the same figure.

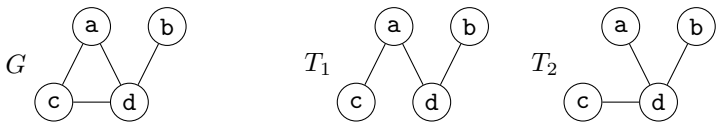


Figure 5.5: A graph and two of its possible spanning trees.

One can modify Algorithm 3 to construct the spanning tree of a graph. For instance, the following algorithm uses BFS, and a queue, to construct a spanning

tree of a given graph.

Algorithm 4: `Breadth-FirstSearch(`G`)`

Input: graph G
Output: a spanning tree $T = (V, E)$ of G
Init: choose a vertex v and append it to Q

```
1  V = {v}, E = ∅
2  repeat
3  |  let x be the head of Q
4  |  foreach y ∈ N_G(x) do
5  |  |  if y ∉ V then
6  |  |  |  append y to Q
7  |  |  |  V ← V ∪ {y}
8  |  |  |  E ← E ∪ {x, y}                    // y is a child of x
9  |  remove x from Q
10 until Q is empty
```

Example 5.8 Let G be the graph in Figure 5.4. A spanning tree is given by the following queue (see also Figure 5.6):

$$\emptyset \to (\mathtt{a}) \overset{\{\mathtt{a},\mathtt{d}\}}{\to} (\mathtt{a},\mathtt{d}) \to (\mathtt{d}) \overset{\{\mathtt{d},\mathtt{b}\}}{\to} (\mathtt{d},\mathtt{b}) \overset{\{\mathtt{d},\mathtt{e}\}}{\to} (\mathtt{d},\mathtt{b},\mathtt{e}) \overset{\{\mathtt{d},\mathtt{g}\}}{\to} (\mathtt{d},\mathtt{b},\mathtt{e},\mathtt{g})$$
$$\to (\mathtt{b},\mathtt{e},\mathtt{g}) \overset{\{\mathtt{b},\mathtt{c}\}}{\to} (\mathtt{b},\mathtt{e},\mathtt{g},\mathtt{c}) \to (\mathtt{e},\mathtt{g},\mathtt{c}) \overset{\{\mathtt{e},\mathtt{f}\}}{\to} (\mathtt{e},\mathtt{g},\mathtt{c},\mathtt{f})$$
$$\overset{\{\mathtt{e},\mathtt{h}\}}{\to} (\mathtt{e},\mathtt{g},\mathtt{c},\mathtt{f},\mathtt{h}) \to (\mathtt{g},\mathtt{c},\mathtt{f},\mathtt{h}) \to (\mathtt{c},\mathtt{f},\mathtt{h}) \to (\mathtt{f},\mathtt{h}) \to (\mathtt{h})$$
$$\overset{\{\mathtt{h},\mathtt{i}\}}{\to} (\mathtt{h},\mathtt{i}) \to (\mathtt{i}) \to \emptyset.$$

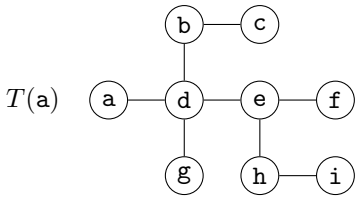


Figure 5.6: A spanning tree obtained by BFS.

Remember that, for a rooted tree $T(r)$ the level of a vertex v in T is defined as $\ell_T(v) = d_T(r, v)$.

Let $T(r)$ be a spanning tree of a connected graph G obtained by BFS . Then it is easy to show that for every vertex v of G we have $\ell_T(v) = d_G(r, v)$, i.e., the distance between r and v in the original graph is the same as the distance in the spanning tree.

Example 5.9 Let G be the graph in Example 5.4 and $T(\mathtt{a})$ be the rooted spanning tree obtained from it in Example 5.8. One can check that we have, e.g., $d_G(\mathtt{a},\mathtt{h}) = d_T(\mathtt{a},\mathtt{h}) = 3$.

Similarly we can construct a spanning tree of a graph using DFS, with a stack as data structure.

Algorithm 5: `Depth-FirstSearch(`G`)`

Input: graph G
Output: a spanning tree $T = (V, E)$ of G
Init: choose a vertex v and push it to the top of S

```
1  V = {v}, E = ∅
2  repeat
3      let x be the top of S
4      if N_G(x) \ V is not empty then
5          choose y ∈ N_G(x) \ V
6          push y to the top of S
7          V ← V ∪ {y}
8          E ← E ∪ {x, y}                    // y is a child of x
9      else
10         remove x from S
11 until S is empty
```

Example 5.10 Let G be the graph in Figure 5.4. A spanning tree is given by the following stack (see also Figure 5.7):

$$\emptyset \to [\mathtt{a}] \overset{\{\mathtt{a},\mathtt{d}\}}{\to} [\mathtt{a},\mathtt{d}] \overset{\{\mathtt{d},\mathtt{e}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e}] \overset{\{\mathtt{e},\mathtt{f}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \overset{\{\mathtt{f},\mathtt{c}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c}]$$
$$\overset{\{\mathtt{c},\mathtt{b}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c},\mathtt{b}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{c}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \overset{\{\mathtt{f},\mathtt{h}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}]$$
$$\overset{\{\mathtt{h},\mathtt{g}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h},\mathtt{g}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}] \overset{\{\mathtt{h},\mathtt{i}\}}{\to} [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h},\mathtt{i}]$$
$$\to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{h}] \to [\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f}] \to [\mathtt{a},\mathtt{d},\mathtt{e}] \to [\mathtt{a},\mathtt{d}] \to [\mathtt{a}] \to \emptyset.$$

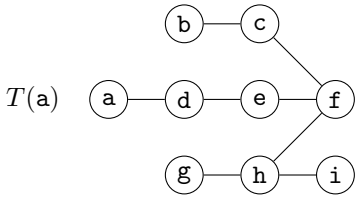


Figure 5.7: A spanning tree obtained by DFS.

5.5 Tree-Search Algorithms: ordered edges

A different algorithm to obtain a spanning tree T from a graph $G = (V, E)$ is given by adding a vertex at a time without creating any cycles.

1. Let us assume that the edges of E are ordered: $e_1 < e_2 < \ldots < e_m$.

2. Construct a sequence of subgraphs

$$G_0 \subseteq G_1 \subseteq \cdots \subseteq G_k$$

such that

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

$$E(G_{i+1}) = \begin{cases} E(G_i) \cup \{e_i\} & \text{if by adding } e_i \text{ no cycle is created in } E(G_i), \\ E(G_i) & \text{otherwise.} \end{cases}$$

3. We stop when $\#E_i = \#V - 1$.

In the last step, we rely on Euler's formula for trees.
The algorithm can be written in pseudo-code as follows.

Algorithm 6: `TreeSearchEdges(`G`)`

Input: graph $G = (V, E)$ with $E = \{e_1 < e_2 < \ldots < e_m\}$
Output: spanning tree $T = (V, E')$ of G

```
1 E' = ∅, i = 1
2 repeat
3 |   if (T + e_i is acyclic) then
4 |   |   E' ← E' ∪ {e_i}
5 |   i → i + 1
6 until #E' = #V − 1
```

Example 5.11 Let us consider the graph of Example 5.4 with edges ordered as on the left of Figure 5.8: $e_1 < e_2 < \ldots < e_{12}$. The spanning tree obtained using the algorithm above is shown on the right of the same figure. For instance, we do not add e_6 since this would add a cycle $(\mathtt{b}, \mathtt{c}, \mathtt{e}, \mathtt{d})$.

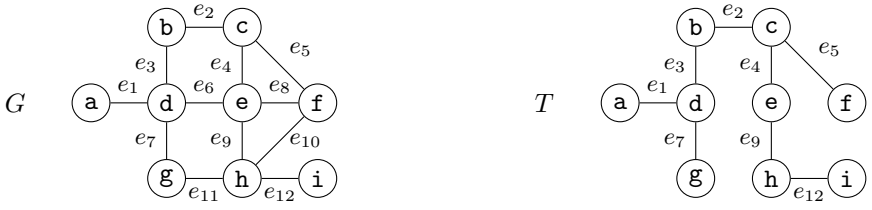


Figure 5.8: A graph and one of its spanning tree.

5.6 Weighted graphs

In some applications it is important to consider edges differently and to associate to each of them a different number.

A *weighted graph* (G, w) is a graph $G = (V, E)$ together with a function $w : E \to \mathbb{R}$, called *weight function*, associating to each edge e its *weight* $w(e)$. Given a subgraph $H = (U, F)$ of G, we call $w(H) = \sum_{e \in F} w(e)$ the *weight* of H.

Example 5.12 Let G be the simple graph in Figure 5.9 where the weight of each edge is represented above it. Let P be the path $(\mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g})$ in G (edges are in blue in Figure 5.9). Then $w(G) = 22$ and $w(P) = 14$.

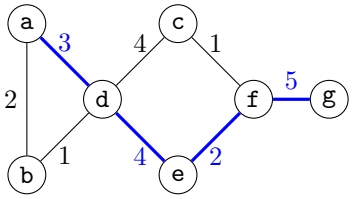


Figure 5.9: A weighted graph.

5.7 Minimal spanning trees

Let (G, w) be a weighted graph, with $G = (V, E)$. A *minimum spanning tree* (or *optimal tree*) of G is a spanning tree $T = (V, E')$ of G such that the cost $w(T) = \sum_{e \in E'} w(e)$ of T is minimal.

Several known algorithms exist for finding a minimum spanning tree. The most used is the so-called Kruskal's algorithm

Algorithm 7: `Kruskal`$((G, w))$

Input: weighted graph (G, w) with $G = (V, E)$
Output: minimal spanning tree $T = (V, E')$ of G
1 sort E s.t. $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
2 return `TreeSearchEdges`(G, E)

Example 5.13 ČEZ wants to connect some major cities (**P**rague, **B**rno, **L**iberec, **Ú**stí nad Labem, **O**lomouc, **H**radec Králove, **Z**lín, **Č**eské Buděovice) to the electricity network. The network has to be connected, otherwise some towns will not get the electricity. On the other hand, ČEZ wants to keep the building costs as low as possible. Thus, they want to build a connected graph with the lowest possible total cost of links. Assuming that the cost of building a link between two cities is proportional to the distance between these cities (see Table 5.1), find an optimal subgraph.

km	**P**	**B**	**L**	**Ú**	**O**	**H**	**Z**	**Č**
P	–	185	88	69	209	101	252	124
B		–	207	246	65	126	77	157
L			–	73	204	83	254	204
Ú				–	258	137	305	191
O					–	122	51	213
H						–	171	169
Z							–	234
Č								–

Table 5.1: Distance between cities in Czechia.

Using the distance between the cities we can construct a weighted complete graph K_8 on 8 vertices $\{\mathbf{P}, \mathbf{B}, \mathbf{L}, \mathbf{Ú}, \mathbf{O}, \mathbf{H}, \mathbf{Z}, \mathbf{Č}\}$. This complete graph has $\frac{8\cdot 7}{2} = 24$ edges. Let's order them according to their weight:

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

The minimal spanning tree of the complete graph using Kruskal's algorithm is shown in Figure 5.10

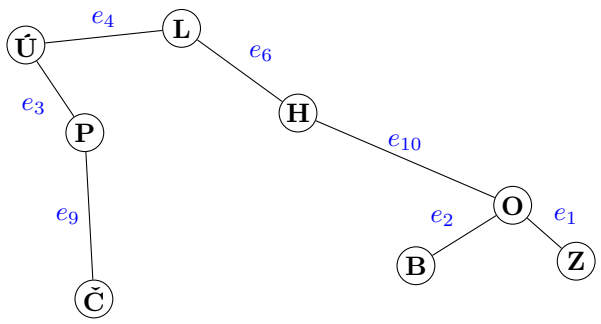


Figure 5.10: A minimal spanning tree of K_8.

Exercise 5 Let (G, w) be the weighted graph represented in Figure 5.11, where each edge is represented with its weight. Find a minimal spanning tree of G.

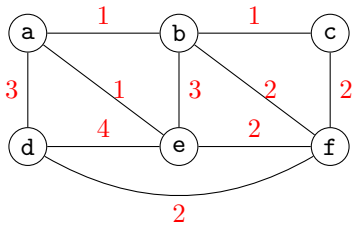


Figure 5.11: A weighted graph.

Solution of Exercise 5 Let us order the edges of G according to their weights: $e_1 < e_2 < \cdots < e_{10}$ (this order is not unique). The label ad weight of each vertex is shown on the left of Figure 5.12. The minimal spanning tree obtained using Kruskal's algorithm is shown on the right of the same figure.

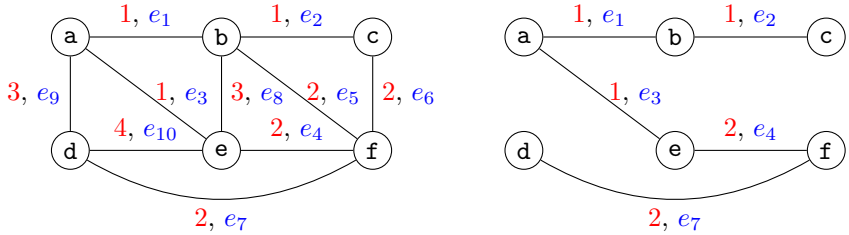


Figure 5.12: A weighted graph and one of its minimal spanning tree.

5.8 Time complexity, Union-Find operation

The time complexity of the algorithms seen in the previous lecture depends on the used data structures. For instance, in Kruskal's Algorithm, time complexity heavily depends on the choice of data structure used in the step where the algorithm has to test whether by addition of an edge we create a cycle in G_i.

To test whether we have a cycle is quite simple: it is enough to remember for each vertex v the component the vertex belong to: a new edge $\{x, y\}$ does create a cycle if and only if x and y already belong to the same component.

A naive implementation could be to use an array to store the label of its component for each vertex.

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

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

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

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

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

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

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

- We represent individual components by trees.
- The root of a tree is the label of the component, or a representative of the component.
- FIND (a): find out the root of the tree representing the component containing the vertex a. The complexity is proportional to the length of the tree, i.e., $\mathcal{O}(\ln n)$.
- UNION (a, b): we need to merge two components. We merge the trees representing both components so that the height of the resulting tree is as small as possible. To achieve this goal we connect the trees so that the root of the smaller tree is a child of the root of the longer tree. In this way the height increases at most by 1. The complexity is $\mathcal{O}(1)$ – if we remember the length of trees and we ran FIND $(\cdot)$ before UNION $(\cdot)$ – or $\mathcal{O}(\ln n)$.
- The total time complexity is $\mathcal{O}(m \cdot \ln n)$, where $m = \#E$ and $n = \#V$.

Example 5.14 Let G be a graph having two components A and X as in the left of Figure 5.13. One can check that FIND $(\texttt{b}) = A$ and FIND $(\texttt{z}) = B$. Two possible spanning trees of the components are shown in center of the same figure (we select `a` as representative of A and `x` as representative of X). The union of the two components containing `b` and `z` is shown on the right of the figure.

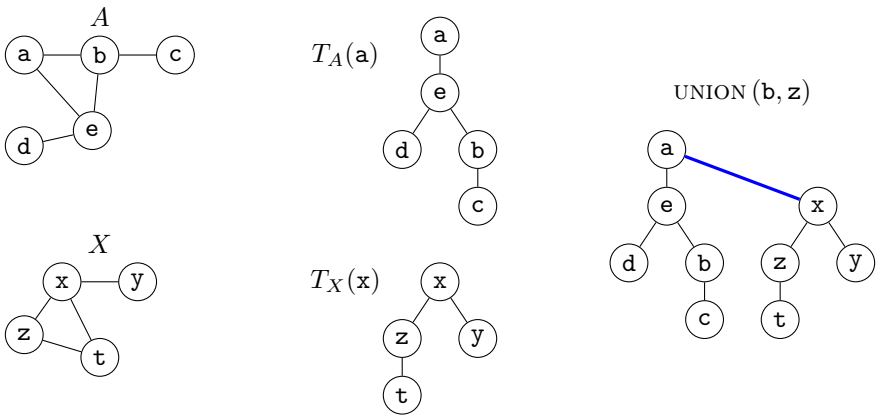


Figure 5.13: UNION-FIND scheme using trees.

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

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

Chapter 6

Euler tours and Hamiltonian cycles

6.1 Euler tours

Let $G = (V, E)$ be a (non necessarily simple) graph of order n and size m. A walk $(v_0 \overset{e_1}{\to} v_1 \overset{e_2}{\to} \cdots \overset{e_m}{\to} v_m)$ is called an *Eulerian walk* if for every $1 \leq i, j \leq m$ with $i \neq j$ we have $e_i \neq e_j$. Thus, an Eulerian walk is a trail that traverses every edge, i.e., every edge in E is used exactly once.

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

Example 6.1 Let G be the simple graph in Figure 6.1. An Eulerian walk in the graph is given by the path $(e_5, e_1, e_2, e_3, e_4, e_8, e_7, e_6)$. There is no Euler tour in G, so the graph is not Eulerian.

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

Figure 6.1: Two non-Eulerian graphs.

A necessary condition for having an Eulerian walk or an Euler tour in a graph is connectedness. This condition is not sufficient, though, as shown in Example 6.1.

Theorem 6.1 (Euler (1736)) *A connected graph is Eulerian if and only if it is even.*

Proof.

($\Rightarrow$) Every vertex appearing k times in an Euler tour must have degree $2k$.

($\Leftarrow$) Let G be a connected even graph. We construct an Euler tour in the following way.

0) • $i \leftarrow 1$.
   • Let us choose a random starting vertex v_i and a colour c_i.

1) Walk, as long as possible, along the edges (at random) of G. When leaving a vertex choose an edge with no colour. While traversing an edge, colour it using c_i.

2) If there is no way how to continue, we have returned to the starting vertex v_i and, moreover, all edges incident with v_i have already been coloured.

   a) If there are no colourless edges go to Step 3) (RECONSTRUCTION).

   b) If there are some colourless edges left:
      • $i \leftarrow i + 1$.
      • Choose a new colour c_i that has not been used so far.
      • Choose a new starting vertex v_i such that there are some colourless as well as some coloured edges incident with it (it is possible to find such a vertex v_i due to the assumption that G is connected).
      • Go to Step 1).

3) (RECONSTRUCTION) All edges of G have been coloured, i.e., the set E is partitioned by a set of disjoint closed walks $\{W_i\}_i$. We have to join these walks into one Euler tour.

   i) Start at v_1, follow the edges of the first closed walk.

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

■

By applying rule *ii*) in Theorem 6.1, the walks can "nest" several times and they can be proessed in an order different to the order in which they have been created.

Example 6.2 Let H be the graph in Figure 6.2.

(Step 0) We start by choosing the vertex `b` and the colour red.

(Step 1) We consider the closed walk $W_r = (\texttt{b} \xrightarrow{e_1} \texttt{d} \xrightarrow{e_2} \texttt{c} \xrightarrow{e_3} \texttt{a} \xrightarrow{e_4} \texttt{b} \xrightarrow{e_5} \texttt{e} \xrightarrow{e_6} \texttt{c} \xrightarrow{e_7} \texttt{b})$.

(Step 2) We choose the vertex d and the colour green.

(Step 1) We consider the closed walk $W_g = (\mathtt{d} \overset{e_8}{\to} \mathtt{h} \overset{e_9}{\to} \mathtt{g} \overset{e_{10}}{\to} \mathtt{d})$.

(Step 2) We choose the vertex g and the colour blue.

(Step 1) We consider the closed walk $W_b = (\mathtt{g} \overset{e_{11}}{\to} \mathtt{f} \overset{e_{12}}{\to} \mathtt{e} \overset{e_{13}}{\to} \mathtt{g})$.

(Step 2) All edges are coloured.

(Step 3) The Euler tour is $(e_1, e_8, e_9, e_{11}, e_{12}, e_{13}, e_{10}, e_2, e_3, e_4, e_5, e_6, e_7)$.

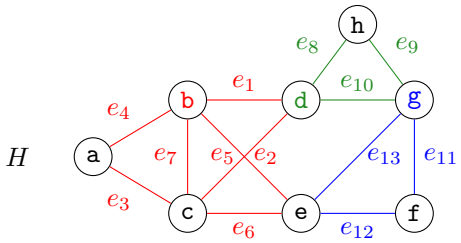


Figure 6.2: Finding an Eulerian tour in a graph.

The proof of Theorem 6.1 gives an algorithm for finding Euler tours in an Eulerian graph. There is, however, a more efficient algorithm based on Tarry's traversal algorithm with complexity $\mathcal{O}(m+n)$, where m and n are respectively the size and the order of the graph.

- Choose a starting vertex v_0.
- Traverse G using as long as possible the two rules:

  (R1) no edge can be used twice in the same direction;

  (R2) the order in which we choose an edge when leaving a vertex u is:

  a) an edge which has not yet been used,

  b) an edge used to arrive to u with the exception of the edge of first arrival,

  c) the edge of first arrival to u.

  During the traversal we record the edge of first arrival for every vertex v (except at the beginning for v_0), and add the edges in a sequence called *return sequence*, according to the order in which they are used for the second time.

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

Example 6.3 Let H be the graph in Example 6.2. Let us show an application of the algorithm described above on H, starting from the vertex `b` (see Figure 6.3).

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

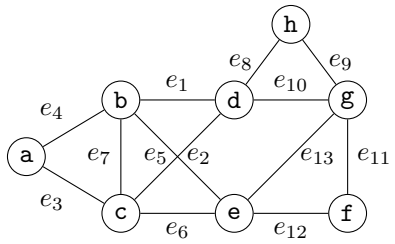
$$(\mathtt{b} \xrightarrow{e_1} \mathtt{d} \xrightarrow{e_2} \mathtt{c} \xrightarrow{e_3} \mathtt{a} \xrightarrow{e_4} \mathtt{b} \xrightarrow{e_5} \mathtt{e} \xrightarrow{e_6} \mathtt{c} \xrightarrow{e_7} \mathtt{b} \xrightarrow{e_7} \mathtt{c} \xrightarrow{e_6} \mathtt{e} \xrightarrow{e_{12}} \mathtt{f} \xrightarrow{e_{11}} \mathtt{g} \xrightarrow{e_{13}} \mathtt{e} \xrightarrow{e_{13}} \mathtt{g}$$
$$\xrightarrow{e_9} \mathtt{h} \xrightarrow{e_8} \mathtt{d} \xrightarrow{e_{10}} \mathtt{g} \xrightarrow{e_{10}} \mathtt{d} \xrightarrow{e_8} \mathtt{h} \xrightarrow{e_9} \mathtt{g} \xrightarrow{e_{11}} \mathtt{f} \xrightarrow{e_{12}} \mathtt{e} \xrightarrow{e_5} \mathtt{b} \xrightarrow{e_4} \mathtt{a} \xrightarrow{e_3} \mathtt{c} \xrightarrow{e_2} \mathtt{d} \xrightarrow{e_1} \mathtt{b}),$$

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

Thus, an Euler tour (closed Eulerian walk) over H is given by

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

H



vertex	first arrival
a	e_3
b	e_4
c	e_2
d	e_1
e	e_5
f	e_{12}
g	e_{11}
h	e_9

Figure 6.3: Finding an Eulerian tour in a graph.

Remember that the number of vertices with odd degrees in a graph is even. Therefore, in particular, it cannot be 1.

Theorem 6.2 *A connected graph has an Eulerian trail if and only if the number of vertices with odd degree is at most* 2 *(i.e., either* 0 *or* 2*).*

Example 6.4 The graphs G and K in Example 6.1 are not Eulerian. Indeed they are connected but not even, since, e.g., $d_G(\mathtt{e}) = 3$ and $d_K(A) = 5$.

The graph G has an Eulerian walk, since only `d` and `e` have odd degrees.

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

6.2 Hamilton cycles

A dual problem of the one of finding an Euler tour in a graph is the one of finding a cycle visiting every vertex exactly once.

We call *Hamilton path* and *Hamilton cycle* respectively a spanning path and a spanning cycle in a graph G.

Thus, every Hamilton path (resp., Hamilton cycle) in $G = (V, E)$ has length $\#V - 1$.

A graph containing a Hamilton path is called *traceable*. A graph containing a Hamilton cycle is called *Hamiltonian*. Clearly, every Hamiltonian graph is also traceable.

Example 6.5 The G graph in Example 6.1 is Hamiltonian. A Hamilton path and a Hamilton cycle of the graph are shown in Figure 6.4.

Figure 6.4: A Hamilton path (on the left) and a Hamilton cycle (on the right).

There is a major difference in comparison to the problem of the existence of Euler tours. To determine whether or not a given graph has a Hamilton cycle is much harder. No good characterisation is known or even expected to exist (the problem is a $\mathcal{NP}$-complete problem).

Path exchanges. A natural way of looking for a Hamilton path is by extending a path as much as possible. Let suppose that we found a path $P = (v_1, \ldots, v_k)$ in a graph G and let $v \in \mathrm{N}_G(v_k) \setminus \{v_k\}$ (we want to avoid loops).

- If v is not in P, i.e., if $v \neq v_i$ for all $1 \leq i \leq k$, then we can extend the longer path to $P' = P + \{(v_k, v)\}$.
- If v is in the path but $v \neq v_{k-1}$, i.e., $v = v_i$ for some $1 \leq i \leq k-2$, then we can obtain a path of the same length by exchanging $\{v_i, v_{i+1}\}$ with $\{v_k, v_i\}$, i.e., considering the path $P'' = (P \setminus \{v_i, v_{i+1}\}) + \{(v_k, v_i)\}$ (see Figure 6.5).

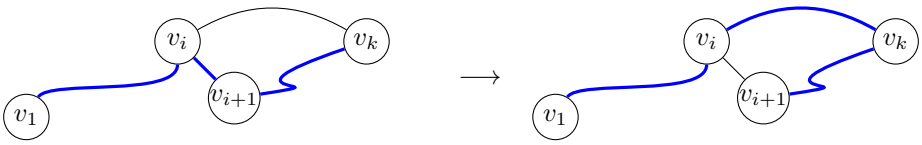


Figure 6.5: A path exchange.

Cycle exchange. In a similar way, let us suppose that we have a cycle $C = (v_1, \ldots, v_k)$ having two vertices v_i, v_j non consecutive in C, i.e., with $|i-j| > 1$ and such that both $\{v_i, v_j\}$ and $\{v_{i+1}, v_{j+1}\}$ are edges of the graph. Then we can obtain a new cycle

$$C' = (C \setminus \{\{v_i, v_{i+1}\}, \{v_j, v_{j+1}\}\}) + \{\{v_i, v_j\}, \{v_{i+1}, v_{j+1}\}\}$$

having the same length as C (see Figure 6.6).

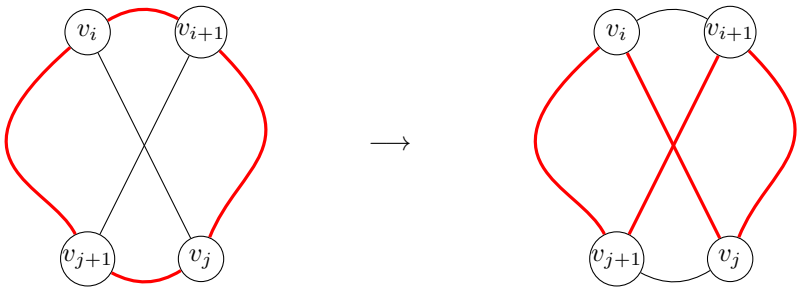


Figure 6.6: A path exchange.

There are several known sufficient conditions for the existence of a Hamilton cycle in a graph.

Every complete graph of order at least 3 is Hamiltonian, since a Hamilton cycle can be obtained by selecting all vertices one by one in an arbitrary order.

Example 6.6 The complete graph K_5 over set of vertices $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ has, e.g., the Hamilton cycle: $(\mathtt{b}, \mathtt{c}, \mathtt{a}, \mathtt{e}, \mathtt{d})$ (see Figure 6.7).

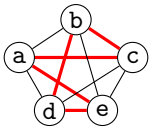


Figure 6.7: A Hamilton cycle in K_5.

When we have fewer edges, finding a Hamilton cycle is harder. Dirac proved what is the minimum degree that a graph must have to guarantee the existence of a Hamilton cycle.

Theorem 6.3 (Dirac (1952)) *Every simple graph G of order $n \geq 3$ such that $\delta(G) \geq \frac{n}{2}$ is Hamiltonian.*

Dirac's Theorem has the best possible bound on $\delta(G)$ to ensure the existence of a Hamilton cycle. We cannot, e.g., replace $\frac{n}{2}$ with $\lfloor \frac{n}{2} \rfloor$.

Example 6.7 Let n be an odd number and G be the union of two complete graphs $K_{\lceil \frac{n}{2} \rceil}$ meeting in one vertex w (see Figure 6.8, where $n = 7$). Then $\delta(G) = \lfloor \frac{n}{2} \rfloor$ but G cannot have a Hamilton cycle as it would have to go through w twice.

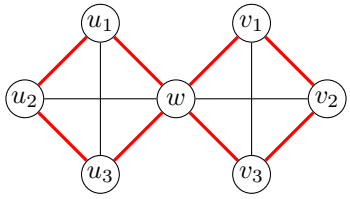


Figure 6.8: A closed walk visiting all vertices passing through w twice.

We can find other sufficient conditions.

Theorem 6.4 (Ore (1960)) *Let $G = (V, E)$ be a simple graph and let $u, v \in V$ such that $d_G(u) + d_G(v) \geq \#V$. Then G is Hamiltonian if and only if $G + \{u, v\}$ is Hamiltonian.*

Proof.

($\Rightarrow$) If G has a Hamilton cycle, then clearly the same cycle is also in $G + \{u, v\}$.

($\Leftarrow$) If $G + \{u, v\}$ has a Hamilton cycle C, then we can find a Hamilton cycle C' by using a cycle exchange (Exercise).

■

The *closure* of a (simple) graph $G = (V, E)$ is the graph obtained from G by recursively adding edges $\{u, v\}$ if $\{u, v\}$ is not an edge in the graph and the sum of the two degrees is at least $\#V$, until all vertices of such form are adjacent. One can prove that the order in which the edges are added does not change the final result (why?).

Example 6.8 Let G be the graph of Example 6.1. The closure of G is the complete graph K_5 (see Figure 6.9).

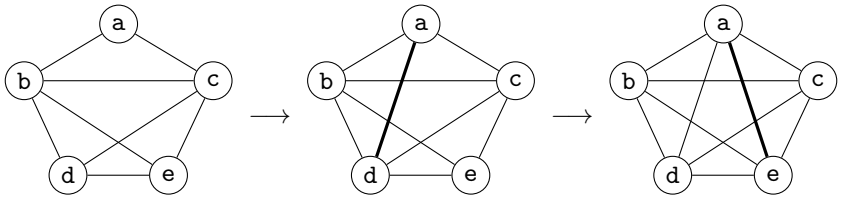


Figure 6.9: Closure of a graph.

A consequence of Theorem 6.4 is the following.

Corollary 4 *A simple graph is Hamiltonian if and only if its closure is Hamiltonian.*

There exist other sufficient condition for a graph to have a Hamilton cycle. Most of them requires the graph to have "enough" edges.

Theorem 6.5 (Pósa (1962)) *Let $G = (V, E)$ be a simple graph of order n realised by a graphic sequence $(d_i)_{i=1}^n$ with $d_1 \leq d_2 \leq \ldots \leq d_n$. If for every $k < \frac{n}{2}$ we have $d_k > k$, then G is Hamiltonian.*

Example 6.9 The Hamiltonian graph G of Example 6.1 is realised by the graphic sequence $(d_i)_{i=1}^5 = (2, 3, 3, 4, 4)$. For $k < \frac{5}{2}$, i.e., for $k = 1, 2$ we have $d_1 = 2 > 1$ and $d_2 = 3 > 2$.

Theorem 6.6 (Chvátal (1972)) *Let $G = (V, E)$ be a simple graph of order $n \geq 3$ realised by the graphic sequence $(d_i)_{i=1}^n$ with $d_1 \leq d_2 \leq \ldots \leq d_n$. If for every $k < \frac{n}{2}$ either $d_k > k$ or $d_{n-k} \geq n - k$, then H is Hamiltonian.*

Example 6.10 Let G be the graph shown in Figure 6.10. This graph has order 7 and it is realised by the graphic sequence $(d_i)_{i=1}^7 = (2, 3, 3, 5, 5, 5, 5)$. One has $d_3 = 3 \leq 3$ but $d_4 = 5 \geq 4$. Moreover, $d_1 = 2 > 1$ and $d_2 = 3 > 2$. So G satisfies Chvátal's condition but not Pósa's condition. A Hamilton cycle is shown in red in the same figure.

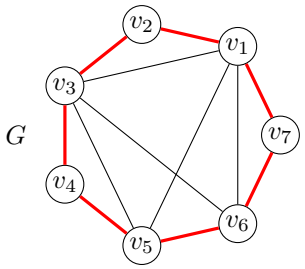


Figure 6.10: A Hamilton cycle in a graph.

Exercise 6 Let us consider the two graphs in Figure 6.11 (the one on the right is called *Petersen graph*).

1. Do the graphs have Eulerian trails?
2. Do the graphs have Eulerian tours?
3. Do the graphs have Hamilton paths?
4. Do they have Hamilton cycles?
5. Do they satisfy Dirac/Pósa/Chvátal conditions?

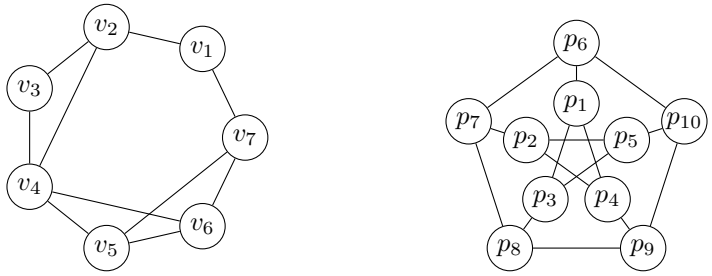


Figure 6.11: Two graphs (on the right the Petersen graph).

Solution of Exercise 6

- The degree of the vertices of the first graph are:

$$d(v_1) = d(v_3) = 2, \quad d(v_2) = d(v_5) = d(v_6) = d(v_7) = 3, \quad \text{and} \quad d(v_4) = 4.$$

Since there are more than 2 vertices with odd degree, the graph **cannot have any Eulerian tour** (**nor Eulerian trail**).

Since the minimum degree is $2 < \frac{7}{2}$, **Dirac's condition is not satisfied**.

The graphic sequence realising the graph is $(d_i)_{i=1}^7 = (2, 2, 2, 3, 3, 3, 4)$. One can check that $d_2 = 2 \not> 2$, so **Pósa's condition is not satisfied**, **nor is Chvátal's**, since $d_{7-2} = d_5 = 3 \not\geq 5$.

Nevertheless, the graph has a Hamilton cycle (and thus a Hamilton path): $(v_1, v_2, v_3, v_4, v_5, v_6)$ (see right of Figure 6.12).

- The second graph, the Petersen graph, is a cubic graph, so it has more than 2 vertices with odd degrees and **cannot have any Euler trail or Euler tour**.

The graphic sequence realising it is $(d_i)_{i=1}^{10} = (3, 3, 3, 3, 3, 3, 3, 3, 3, 3)$ and it is easy to check that the minimal degree is $3 < \frac{10}{2}$, that $d_3 \not> 3$ and that $d_{10-3} = d_7 = 3 \not\geq 3$. So **Dirac's condition, Pósa's condition and Chvátal's condition are not satisfied**.

A Hamilton path is: $(p_1, p_3, p_5, p_2, p_4, p_9, p_{10}, p_6, p_7, p_8)$. The graph does not admit any Hamilton cycle.

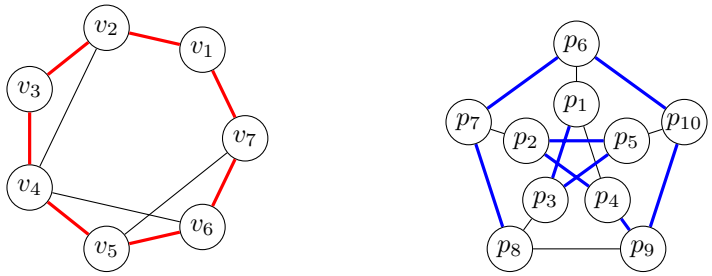


Figure 6.12: Two graphs (on the right the Petersen graph).

The *Travelling Salesperson Problem* (TSP) is a classical $\mathcal{NP}$-hard problem: given a weighted graph (G, w) find a minimum-weight Hamilton cycle in G.

Exercise 7 (Trick-or-Treat Spooky Problem (TSP)) It's Halloween Night and you decide to go around in your neighbourhood to collect as many candies as you can. The distance between the houses are represented in Table 6.1 and Figure 6.13.

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

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

Table 6.1: Distance between houses in the neigbourhood.

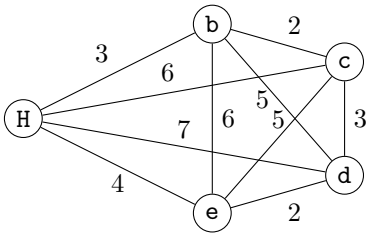


Figure 6.13: A plan of the neighbourhood.

Solution of Exercise 7 An optimal route for the graph is giving by the cycle $(\mathtt{H}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e})$ of weight 14 (see Figure 6.14).

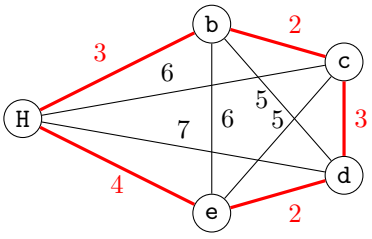


Figure 6.14: A plan of the neighbourhood.

Chapter 7

Networks and Flows

Consider the task of modelling a transportation network with one source s – i.e., one point from where all merchandise leaves – and one sink t – i.e., one point to where all merchandise arrives – in which the amount of flow through a given link between two nodes is subject to a certain capacity of that link. Our aim is to determine the maximum amount of flow through the network from s to t.

Formally, let $D = (V, E)$ be a directed graph.

A *network* with *topology* D is defined as $N = (D, s, t, c)$, where $s, t \in V$ are two particular vertices called respectively the *source* (producer) and the *sink* (consumer) of the network; and $c : E \to \mathbb{R}_0^+$ is a real-valued function called the *capacity function* of the network N. The value $c(a)$ of an arc $a \in E$ will also be called the *capacity* of a.

In our examples we will always consider, for simplicity, $c : E \to \mathbb{N}_0$. Also, note that in some contexts it may be useful to allow arcs of infinite capacity (we will not do that, though!).

A vertex $v \in V \setminus \{s, t\}$ is called an *intermediate vertex*, and we sometimes denote the set of all such vertices as I_N.

Example 7.1 Let $N = (D, \mathtt{s}, \mathtt{t}, c)$ be the network shown in Figure 7.1. The vertex `s` is the source of N, while `t` is its sink. The set of internal vertices is $I_N = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$. Each arc a is labeled with its capacity $c(a)$.

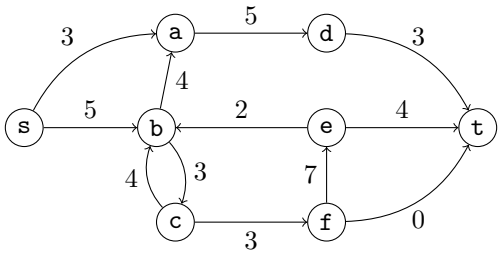


Figure 7.1: A network.

Let $N = (D, s, t, c)$ be a network and let $S \subseteq V$ be such that $s \in S$ and $t \notin S$. Let $\bar{S} = V \setminus S$. The pair $(S, \bar{S})$ is called a *cut* in N.

The *capacity* of a cut $(S, \bar{S})$ is defined as

$$c(S, \bar{S}) = \sum_{\substack{a^- \in S \\ a^+ \in \bar{S}}} c(a).$$

Note that we consider only edges "leaving" S.

A cut in a network N is called a *minimum cut* if no cut in N has a smaller capacity.

Example 7.2 Let N be the network in Example 7.1 and $S = \{\mathtt{s}, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Hence, $\bar{S} = \{\mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{t}\}$. The cut $(S, \bar{S})$ in the network is shown in Figure 7.2. Its capacity is $8 = 5 + 3$. Indeed we consider the sum of the capacities of arcs $(\mathtt{a}, \mathtt{d})$ and $(\mathtt{c}, \mathtt{f})$. Note that we are not considering the arc $(\mathtt{e}, \mathtt{b})$ since it is not leaving from S and arriving to $\bar{S}$.

Similarly, let $T = \{\mathtt{s}, \mathtt{a}, \mathtt{b}, \mathtt{d}\}$. The cut $(T, \bar{T})$ has capacity 6, and one can check that it is a minimum cut.

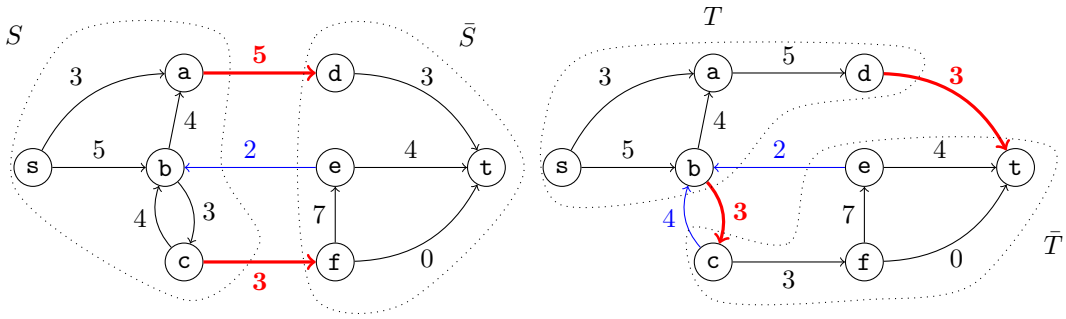


Figure 7.2: A network with two cuts.

7.1 Flows

Let $N = (D, s, t, c)$ be a network with topology $D = (V, E)$.

A function $f : E \to \mathbb{R}_0^+$ is a called a *flow* in N if it satisfies the two following conditions:

(F1) for every $a \in E$, $0 \leq f(a) \leq c(a)$, (*capacity constraint*)

(F2) for every $v \in I_N$, $\sum_{a^-=v} f(a) = \sum_{a^+=v} f(a)$. (*conservation condition*)

The second condition requires that the *net flow* out of any intermediate vertex must be zero. Every network has at least one flow. Indeed, it is enough to consider the *zero flow* defined as $f(a) = 0$ for every $a \in E$.

An arc a is called f-positive if $f(a) > 0$. It is f-*unsaturated* if $f(a) < c(a)$ and f-*saturated* if $f(a) = c(a)$.

Example 7.3 Let N be the network in Example 7.1. A flow f in N is shown in Figure 7.3, where we label each arc a with $f(a)/c(a)$. The path $(\mathtt{b}, \mathtt{c})$ is f-saturated, while the path $(\mathtt{c}, \mathtt{b})$ is f-unsaturated.

One can check, e.g., that for $a = (\mathtt{a}, \mathtt{d})$ one has $f(a) = 3 < 5 = c(a)$.

Moreover, one has $\sum_{a^+=\mathtt{b}} f(a) = 3 + 1 + 0 = 1 + 3 = \sum_{a^-=\mathtt{b}} f(a)$.

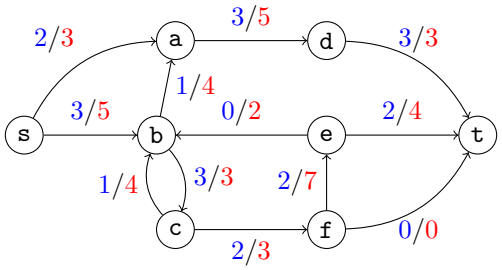


Figure 7.3: A flow in a network.

7.2 Maximum flow

We want to determine the maximum amount of flow through a network. How is this measured?

Let f be a flow in a network $N = (D, s, t, c)$. The *value* of f is

$$\mathrm{val}(f) = \sum_{a^-=s} f(a) - \sum_{a^+=s} f(a).$$

The value of a flow is basically the net amount of flow leaving the source s. Note that one also has

$$\mathrm{val}(f) = \sum_{a^-=t} f(a) - \sum_{a^+=t} f(a)$$

(Prove it!).

A flow f in N is called *maximum* if $\mathrm{val}(f) \geq \mathrm{val}(f')$ for every flow f' in N.

The following theorem states that the value of a flow can be determined not only at the source (or at the sink), but also "on the cut".

Theorem 7.1 *Let $N = (D, s, t, c)$ be a network, f be a flow in N and $(S, \bar{S})$ be a cut in N. Then*

$$val(f) = \sum_{\substack{a^- \in S \\ a^+ \in \bar{S}}} f(a) - \sum_{\substack{a^- \in \bar{S} \\ a^+ \in S}} f(a) \leq c(S, \bar{S}).$$

Hence, the value of a flow in N is never larger than the smallest capacity of a cut in N. In 1956, Ford & Felkurson, and Elias et al. proved that this upper bound is always attained by some flow.

Theorem 7.2 (Max-Flow Min-Cut) *In every network the value of a maximum flow equals the minimum capacity of a cut.*

Example 7.4 Let N be the network in Example 7.1. The flow f seen in Example 7.3 is not maximum since its value is $5 = 2 + 3$, while the capacity of a minimum cut in the network is 6 (see Example 7.2).

On the other hand, the flow $\tilde{f}$ shown in Figure 7.4 has value 6 and is maximum.

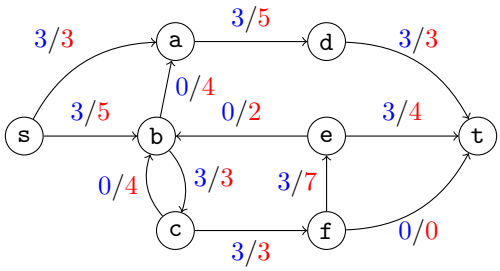


Figure 7.4: A flow in a network.

Theorem 7.2 is nowadays the cornerstone of a standard proof technique in graph theory (in areas not directly connected to flows in networks, e.g., matching or connectivity). It is, incidentally, a corollary of a theorem which characterises maximum flows in a network and that we will state later.

7.3 Semi-paths

Let $D = (V, E)$ be a directed graph. A sequence $(v_i)_{i=0}^k$ is called a *semi-path* in D if for each $1 \leq i \leq k$ either $(v_{i-1}, v_1) \in E$ or $(v_i, v_{i-1}) \in E$. In other words, P

is a semi-path in D if it is a walk in the underlying graph of D. So, a semi-path in a digraph concerns only the existence of an edge, not its actual direction.

Let $N = (D, s, t, c)$ be a network, f a flow in N, and P a semi-path in D with one endpoint s. Given an arc $a \in P$ we define its *reserve*, denoted $r(a)$ as

$$r(a) = \begin{cases} c(a) - f(a) & \text{if } a = (v_i, v_{i+1}) \in P \text{ for some } 0 \leq i < k \\ f(a) & \text{if } a = (v_{i+1}, v_i) \in P \text{ for some } 0 \leq i < k \end{cases}.$$

We call an arc of the form (v_i, v_{i+1}) a *forward arc* of P, and an arc of the form (v_{i+1}, v_i) a *reverse arc* of P.

The *reserve* of the semi-path P is defined as $r(P) = \min_{a \in P} r(a)$.

A semi-path in N with one endpoint in s is called *f-unsaturated* if $r(P) > 0$, i.e., if all forward arcs in P are f-unsaturated and all reverse arcs of P are f-positive. An f-unsaturated semi-path starting in s and ending in t is called *f-incrementing*.

Intuitively, an f-incrementing semi-path P is a semi-path that is not being used to its full capacity. Its reserve is the largest amount by which the flow f can be increased along P without violating the conservation condition.

Example 7.5 Let N and f be as in Example 7.3. Let us consider the two semi-paths $P_1 = (\mathtt{s}, \mathtt{b}, \mathtt{e}, \mathtt{t})$ and $P_2 = (\mathtt{s}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{f}, \mathtt{e}, \mathtt{t})$ in N, with $(\mathtt{b}, \mathtt{c})$ taken as a reverse arc in P_2 (see Figure 7.5). The reserve of P_1 is

$$r(P_1) = \min\{5 - 3,\ 0,\ 4 - 2\} = 0.$$

The reserve of P_2 is

$$r(P_2) = \min\{3 - 2,\ 1,\ 1,\ 3 - 2,\ 7 - 2,\ 4 - 2\} = 1 > 0.$$

P_1 is f-saturated, while P_2 is f-unsaturated and f-augmenting.

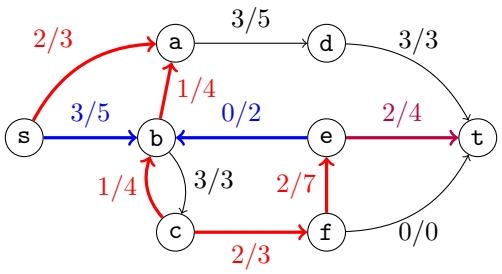


Figure 7.5: Two semi-paths in a network.

7.4 Ford-Fulkerson Theorem

If in a network N having a flow f we have an f-unsaturated semi-path, then f is not maximal.

Proposition 2 *Let $N = (D, s, t, c)$ be a network and f a flow in N. If there is an f-unsaturated semi-path P in N ending in t, then f is not a maximum flow.*

More precisely, the function $\tilde{f}$ defined as

$$\tilde{f}(a) = \begin{cases} f(a) + r(P) & \text{if } a \text{ is a forward arc of } P, \\ f(a) - r(P) & \text{if } a \text{ is a reverse arc of } P, \\ f(a) & \text{if } a \text{ is not on } P \end{cases}$$

is a flow in N such that $val(\tilde{f}) = val(f) + r(P)$.

Proof. First, let us check that $\tilde{f}$ fulfils conditions (F1) and (F2).

(F1) From the construction it follows that for every arc $a \in E$ we have

$$0 \leq \tilde{f}(a) \leq c(a).$$

Indeed:

- for every forward arc a we have
  - $0 \leq f(a) \leq f(a) + r(P) = \tilde{f}(a)$; and
  - $r(P) \leq c(a) - f(a)$, hence $\tilde{f}(a) = f(a) + r(P) \leq c(a)$;
- for every reverse arc a we have
  - $r(P) \leq f(a)$, hence $\tilde{f}(a) = f(a) - r(P) \geq 0$; and
  - $\tilde{f}(a) = f(a) - r(P) \leq f(a) \leq c(a)$.

(F2) Let us prove that for every intermediate vertex v we have

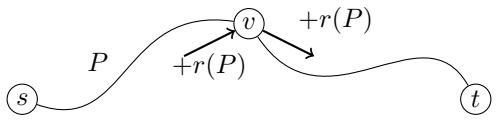
$$\sum_{a^- = v} \tilde{f}(a) = \sum_{a^+ = v} \tilde{f}(a).$$

There are four possible situations, i.e., four possible orientations of edges with respect to a vertex v and a semi-path P.

- If $v = a_1^+ = a_2^-$ for two arcs $a_1, a_2 \in P$, then

$$\tilde{f}(a_1) = f(a_1) + r(P) = f(a_2) + r(P) = \tilde{f}(a_2)$$

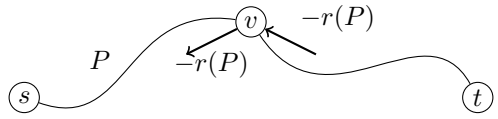
and the equation is satisfied since we added the same quantity, $r(P)$, to both sides.



– If $v = a_1^- = a_2^+$ for two arcs $a_1, a_2 \in P$, then

$$\tilde{f}(a_1) = f(a_1) - r(P) = f(a_2) - r(P) = \tilde{f}(a_2)$$

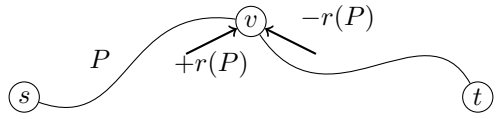
and the equation is satisfied since we removed the same quantity, $r(P)$, to both sides.



– If $v = a_1^+ = a_2^+$ for two arcs $a_1, a_2 \in P$, then

$$\tilde{f}(a_1) + \tilde{f}(a_2) = f(a_1) + r(P) + f(a_2) - r(P) = f(a_1) + f(a_2)$$

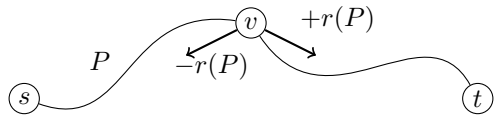
and the equation is satisfied since f is a flow and satisfies condition (F2).



– If $v = a_1^- = a_2^-$ for two arcs $a_1, a_2 \in P$, then

$$\tilde{f}(a_1) + \tilde{f}(a_2) = f(a_1) - r(P) + f(a_2) + r(P) = f(a_1) + f(a_2)$$

and the equation is satisfied since f is a flow and satisfies condition (F2).



Finally, since a semi-path is leaving s exactly one time more it is entering it, we have $\text{val}(\tilde{f}) = \text{val}(f) + r(P)$. ∎

We call the flow $\tilde{f}$ of Proposition 2 the *incremented flow* of f based on P.

Example 7.6 The flow $\tilde{f}$ in Figure 7.4 is the incremented flow of the flow f in Example 7.3 based on the semi-path P_2 in Example 7.5.

Exercise 8 Starting from the zero flow, find an incremenented flow for the network N described in Figure 7.6

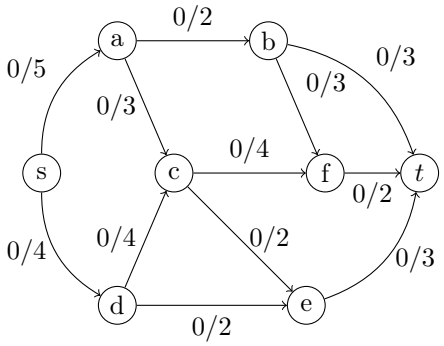
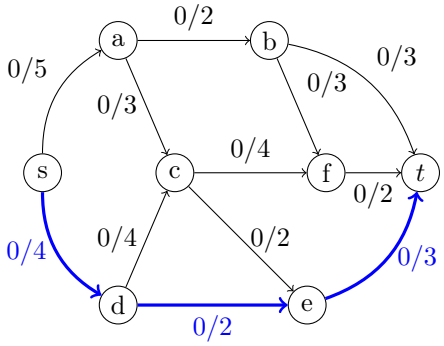


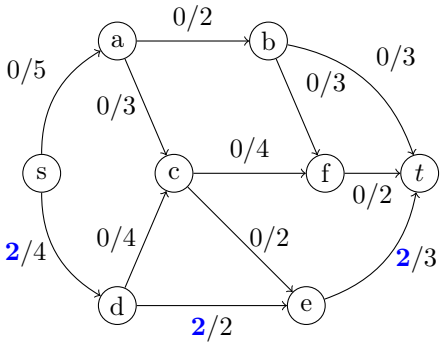
Figure 7.6: A network with zero flow f.

Solution of Exercise 8 Let N be the network in Figure 7.7a with zero flow f_0. A possible incremented flow f_1 is given by incrementing f_0 based on the semi-path $P_1 = (\mathtt{s}, \mathtt{d}, \mathtt{e}, \mathtt{t})$ having reserve $r(P_1) = 2$.

The flow f_1 is shown in Figure 7.7b.



(a) A network with zero flow f_0 with reserve 2.



(b) A network with flow f_1 incrementing f_0 based on the semi-path P_1.

7.5 Maximum flows

How to decide whether a flow is already maximum?

We need some systematic way of searching for f-incrementing semi-paths.

Proposition 3 *Let N be a network and f a flow in it. If there is no f-incrementing semi-path in N, then f is maximal in N.*

Proof. Let us consider the set

$$S = \{v \in V \mid \exists\, f\text{-unsaturated semi-path in } N \text{ ending in } v\}\,.$$

Obviously $s \in S$ and, by assumption, $t \notin S$. Thus $(S, \bar{S})$ is a cut in N. Let us inspect arcs of the type:

1. (x, y), with $x \in S$ and $y \in \bar{S}$, and
2. (z, w), with $z \in \bar{S}$ and $w \in S$.

We claim that for both types of arcs, their reserve is zero.

1. Let $a = (x, y) \in E$, with $x \in S$ and $y \in \bar{S}$.

   By defintion of S, there is a f-unsaturated semi-path P_x from s to x. Since there is no f-unsaturated semi-path from s to y, we have

   $$r(P_x \cup a) = 0 \;\;\Rightarrow\;\; r(a) = 0 \;\;\Rightarrow\;\; f(a) = c(a), \tag{7.1}$$

   where $r(a) = 0$ since a is a forward arc in $P_x \cup a$.

2. Let $a = (z, w) \in E$, with $z \in \bar{S}$ and $w \in S$.

   Let us denote $\bar{a} = (z, w)$. Again, $w \in S$ implies that there exists an f-unsaturated semi-path P_w from s to w. Since there is no f-unsaturated semi-path from s to z, we have

   $$r(P_w \cup \bar{a}) = 0 \;\;\Rightarrow\;\; r(a) = 0 \;\;\Rightarrow\;\; f(a) = 0, \tag{7.2}$$

   where $r(a) = 0$ since a is a reverse arc in $P_w \cup \bar{a}$.

Since we proved that the value of f can be computed not only at the source but also "on the cut", finally we have

$$\begin{aligned} \mathrm{val}(f) \;&= \sum_{\substack{a^- \in S \\ a^+ \in \bar{S}}} f(a) - \sum_{\substack{a^- \in \bar{S} \\ a^+ \in S}} f(a) \overset{(7.1)}{=} \sum_{\substack{a^- \in S \\ a^+ \in \bar{S}}} c(a) - \sum_{\substack{a^- \in \bar{S} \\ a^+ \in S}} f(a) \\ &\overset{(7.2)}{=} \sum_{\substack{a^- \in S \\ a^+ \in \bar{S}}} c(a) = c((S, \bar{S})). \end{aligned}$$

Therefore, we have a flow f and a cut $(S, \bar{S})$ such that $\mathrm{val}(f) = c((S, \bar{S}))$. It follows from the Max-Flow Min-Cut Theorem that f is a maximum flow in N. ∎

Note that from the proof of Proposition 3 we are also able to find a minimum cut.

Combining the results seen above (from this Lecture and from the previous one) we have the following result.

Theorem 7.3 (Ford-Fulkerson) *Let $N = (D, s, t, c)$ be a network and f a flow in N. Then, f is maximal in N if and only if there is no f-incrementing semi-path in N.*

7.6 Ford-Fulkerson Algorithm

Note that the semi-path in the Ford-Fulkerson Theorem 7.3 has necessarily endpoints s – from the definition of f-unsaturated semi-path – and t – by assumption of the theorem.

The method of construction of $\tilde{f}$ gives us an algorithm for finding a maximum flow. We can start with a known flow f, e.g., the zero flow, and search for an f-incrementing semi-path. We stop when there we find a flow $\tilde{f}$ such that the network does not have any $\tilde{f}$-incrementing semi-paths.

Example 7.7 Let us consider again the network N in Example 8 (see also Figure 7.8a) with zero-flow f.

- The semi-path $P_1 = (\mathtt{s}, \mathtt{d}, \mathtt{e}, \mathtt{t})$ is f-incrementing since its reserve is $r(P_1) = \min\{4-0,\ 2-0,\ 3-0\} = 2 > 0$. The incremented flow f_1 of f based on P_1 is shown in Figure 7.8b.

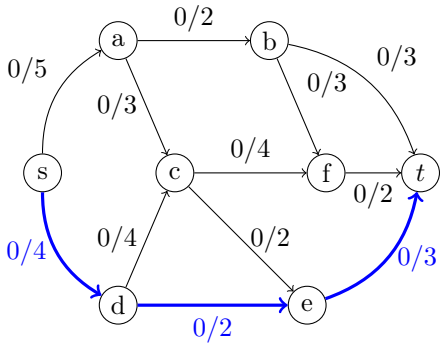
- The semi-path $P_2 = (\mathtt{s}, \mathtt{a}, \mathtt{b}, \mathtt{f}, \mathtt{t})$ is f_1-incrementing since its reserve is $r(P_2) = \min\{5-0,\ 3-0,\ 4-0,\ 3-0\} = 2 > 0$. The incremented flow f_2 of f_1 based on P_2 is shown in Figure 7.8c.

- The semi-path $P_3 = (\mathtt{s}, \mathtt{d}, \mathtt{c}, \mathtt{a}, \mathtt{b}, \mathtt{f}, \mathtt{c}, \mathtt{e}, \mathtt{t})$ is f_2-incrementing since its reserve is $r(P_3) = \min\{4-2,\ 4-0,\ 2,\ 2-0,\ 3-0,\ 2,\ 2-0,\ 3-2\} = 1 > 0$. The incremented flow f_3 of f_2 based on P_3 is shown in Figure 7.8d.

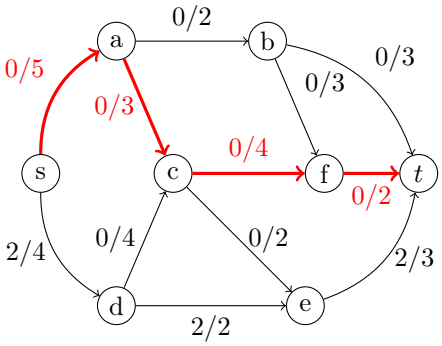
- The semi-path $P_4 = (\mathtt{s}, \mathtt{a}, \mathtt{c}, \mathtt{f}, \mathtt{b}, \mathtt{t})$ is f_3-incrementing since its reserve is $r(P_4) = \min\{5-2,\ 3-1,\ 4-1,\ 1,\ 3-0\} = 1 > 0$. The incremented flow f_4 of f_3 based on P_4 is shown in Figure 7.8e.

- The semi-path $P_5 = (\mathtt{s}, \mathtt{a}, \mathtt{b}, \mathtt{t})$ is f_4-incrementing since its reserve is $r(P_5) = \min\{5-3,\ 2-1,\ 3-1\} = 1 > 0$. The incremented flow f_5 of f_4 based on P_5 is shown in Figure 7.8f.

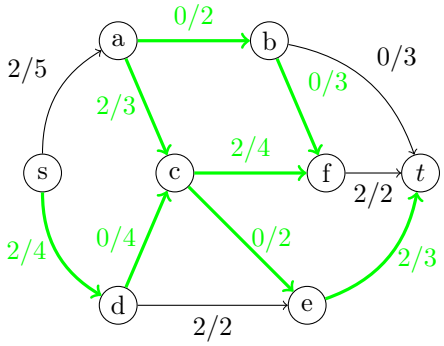
  The flow f_5 is a maximum flow. Its value is 7. A minimum cut of the network is given by $(\{\mathtt{s}, \mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}, \{\mathtt{b}, \mathtt{t}\})$.



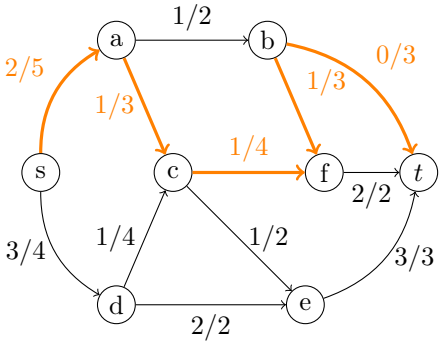
(a) A network with zero flow f.



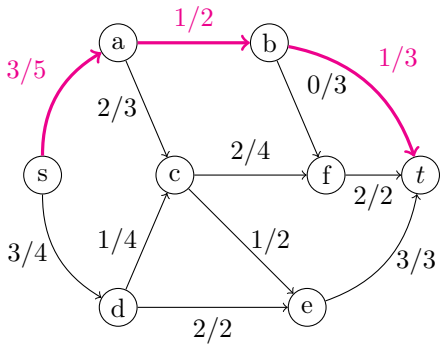
(b) A network with flow f_1.



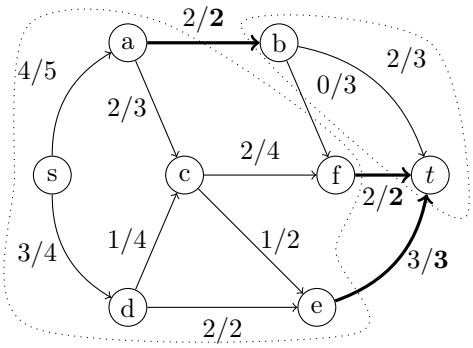
(c) A network with flow f_2.



(d) A network with flow f_3.



(e) A network with flow f_4.



(f) A network with flow f_5.

7.7 Incrementing Path Search

The choice of which semi-path to use at each step in Example 7.7 was somehow arbitrary.

A systematic way to implement Ford-Fulkerson Algorithm in a network $N = (D, s, t, c)$ is to look for f-incrementing semi-paths using a Tree-Search

Algorithm.

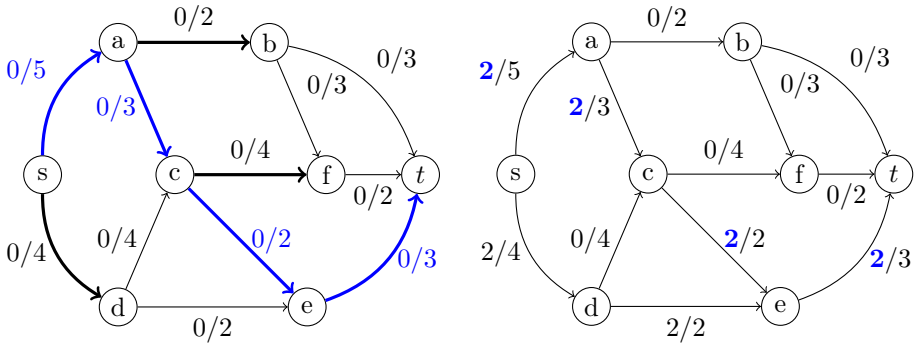
We start, as before, with the zero flow.

At each step we construct a *f-unsaturated rooted tree* $T(s)$, i.e., a rooted tree with root the source and such that for every vertex v in it, the semi-path from s to v is f-unsaturated. If the tree reaches the sink t, then the unique semi-path in the tree from s to t is an f-incrementing semi-path and we can replace f by the flow $\tilde{f}$ as seen in the previous Lecture.

This Tree-Search Algorithm is known as *Incrementing Path Search* (IPS).

Example 7.8 Let us consider the network N of Example 7.7 with zero flow f_0. A possible f_0-unsaturated rooted tree obtained by DFS is shown in Figure 7.9a on the right (arcs in bold). The semi-path $P_1 = (\mathtt{s}, \mathtt{a}, \mathtt{c}, \mathtt{e}, \mathtt{t})$ is f_0-incrementing with reserve 2.

The incremented flow f_1 of f_0 based on P_1 is shown in Figure 7.10a.



(a) An IPS in a network with zero flow.

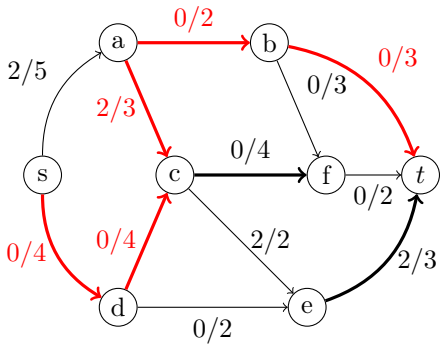
(b) Network with incremented flow.

Exercise 9 Use the Incrementing Path Search to find a maximum flow in the network in Example 7.8.

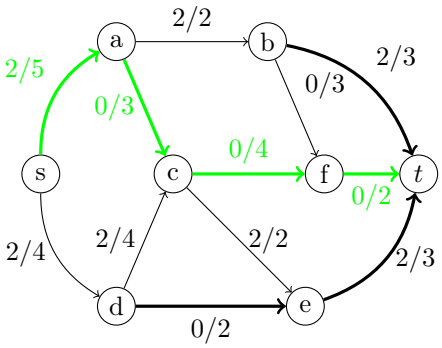
Solution of Exercise 9

- We already saw that a first step to increase the flow from the zero flow f_0 to f_1 may be done by considering the f_0-incrementing semi-path $P_1 = (\mathtt{s}, \mathtt{a}, \mathtt{c}, \mathtt{e}, \mathtt{t})$ with reserve 2 (Figure 7.10a).
- A possible way to continue is considering the f_1-unsaturated tree in Figure 7.10a (in bold) and the semi-path $P_2 = (\mathtt{s}, \mathtt{d}, \mathtt{c}, \mathtt{a}, \mathtt{b}, \mathtt{t})$ in it, which is f_1-incrementing with reserve 2. The incremented flow f_2 of f_1 based on P_2 is shown in Figure 7.10b.
- Then, we could consider the f_2-unsaturated tree in Figure 7.10b (in bold) and the semi-path $P_3 = (\mathtt{s}, \mathtt{a}, \mathtt{c}, \mathtt{f}, \mathtt{t})$ in it, which is f_2-incrementing with reserve 2. The incrementing flow f_3 of f_2 based on P_3 is shown in Figure 7.10c.

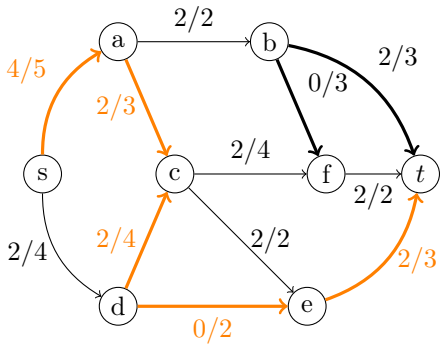
- An f_3-unsaturated tree (in bold), with the f_3-incrementing semi-path $P_4 = (\mathtt{s}, \mathtt{a}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{t})$ is shown in Figure 7.10c. The reserve of P_4 is equal to 1, and the incrementing flow flow f_4 based on P_4 is shown in figure 7.10d.
- The only possible f_4-unsaturated tree is shown in Figure 7.10d (in bold). Since there are no f_4-incrementing semi-paths in it, the flow f_4 is maximal.



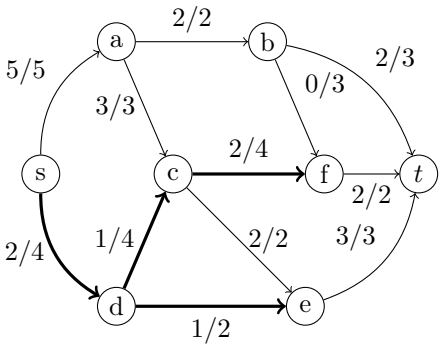
(a) Network with incremented flow f_1.



(b) Network with incremented flow f_2.



(c) Network with incremented flow f_3.



(d) Network with incremented flow f_4.

7.8 Time complexity of Ford-Fulkerson Algorithm

The time complexity of the demonstrated algorithm, called *Ford-Fulkerson Algorithm*, is, for a network with integer capacities $\mathcal{O}(\#E \cdot \mathrm{val}(f))$, where f is a maximum flow.

Indeed, if the algorithm chooses f-unsaturated semi-paths poorly, it can do a lot of unnecessary steps.

Example 7.9 Let us consider the network in Figure 7.11.

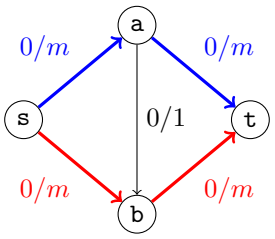


Figure 7.11: A network with zero flow.

Obviously, the value of a maximum flow is $2m$, and it can be easily found in 2 steps: using semi-paths $(\mathtt{s}, \mathtt{a}, \mathtt{t})$ and $(\mathtt{s}, \mathtt{b}, \mathtt{t})$.

However, let us assume that the algorithm will alternatively use semi-paths $(\mathtt{s}, \mathtt{a}, \mathtt{b}, \mathtt{t})$ and $(\mathtt{s}, \mathtt{b}, \mathtt{a}, \mathtt{t})$: The first three steps are shown in Figure 7.12.

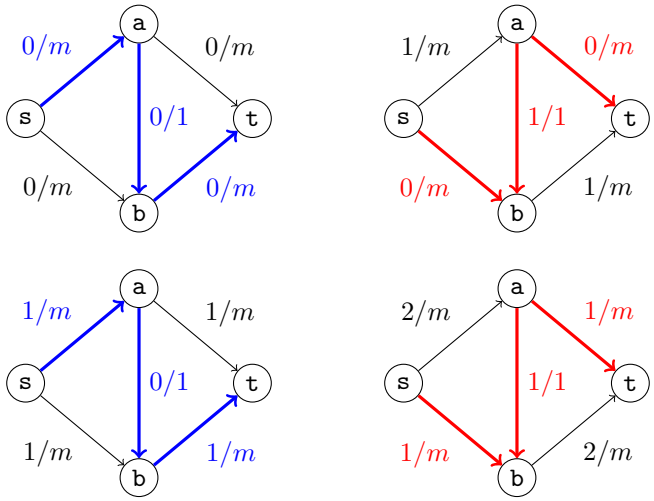


Figure 7.12: Initial configuration (top-left), first step (top-right), second step (bottom-left) and third step (bottom-right) in the Ford-Fulkerson algorithm.

The algorithm will eventually find the right value of maximum flow, $2m$, but it will take $2m$ steps.

This is one of many cases showing that Ford-Fulkerson algorithm is sensitive to the way the f-unsaturated paths are chosen.

Edmonds and Karp proved that most of such problems can be avoided if we choose the shortest f-unsaturated path in each step. The time complexity of this version of the algorithm is $\mathcal{O}(\#V \cdot \#E^2)$.

Chapter 8

Matchings

8.1 Bipartite graphs

Let us recall that a graph $G = (V, E)$ is *bipartite* if there is a decomposition of the set of vertices V into two subsets V_1, V_2, with $V_1 \cap V_2 = \emptyset$ and $V_1 \cup V_2 = V$, such that

$$\binom{V_1}{2} \cap E = \binom{V_2}{2} \cap E = \emptyset.$$

That is, there is no edge within V_1 and no edge within V_2, all the edges of G have one endpoint in V_1 and the second in V_2.

Bipartite graphs are quite common, e.g., in applications where one wants to inspect relations between two groups of different objects. For example, employees and tasks, patients and drugs, processors and jobs, etc.

Even though the definition of a bipartite graph is quite easy to understand, it would be quite unsuitable and difficult to use the definition to decide whether a graph is bipartite.

The following theorem gives us a simple characterisation of bipartite graphs, which is, moreover, easy to check: it can be done in linear time with respect to the number of vertices in the graph.

Theorem 8.1 *A graph G is bipartite if and only if there is no cycle of odd length in G.*

Example 8.1 The complete graph K_3 is not bipartite, since it has a cycle of length 3.

On the other hand, every tree is bipartite since it has no cycle, and thus in particular no odd cycles.

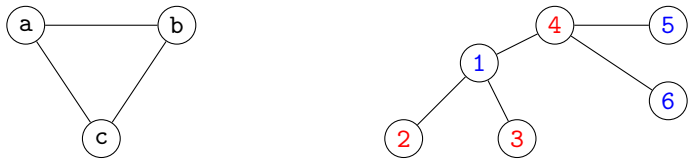


Figure 8.1: The complete graph K_3 (left) and a tree (right).

Let G be a bipartite graph. Let A_G be the adjacency matrix of G. One can easily see that there is a simulaneous permutation of rows and columns of A_G which transform A_G into the following form:

$$A_G = \begin{pmatrix} O & B \\ B^T & O \end{pmatrix}.$$

Indeed, this simultaneous permutation of rows and columns corresponds to the permutation of vertices.

Example 8.2 Let $G = (V, E)$ be the graph on the left of Figure 8.2. The graph is bipartite with $V = V_1 \cup V_2$, where $V_1 = \{1, 4, 5\}$ and $V_2 = \{2, 3\}$. Its adjacency matrix is

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

If we permute the names of vertices $2 \leftrightarrow 4$ and $3 \leftrightarrow 5$, which corresponds to a simultanous switch of the 2nd and 4th row and column and 3rd and 5th row and colomn, we get the adjacency matrix

$$\left(\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{array}\right) = \begin{pmatrix} O_3 & B \\ B^T & O_2 \end{pmatrix} \qquad \text{with} \qquad B = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix},$$

corresponding to the graph on the right of Figure 8.2.

The zero square submatrices on the diagonal of A_G after the permutation correspond to the fact that there are no edges within V_1 and V_2.

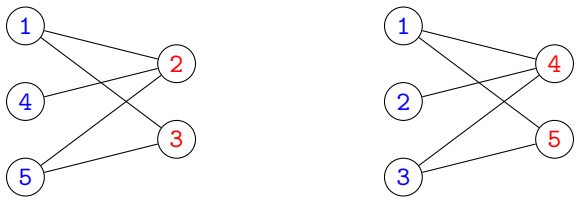


Figure 8.2: Permutation of vertices in a bipartite graph.

8.2 Matching

Let $G = (V, E)$ be a graph. A *matching* in G is a set $M \subset E$ such that for every $e, f \in M$ with $e \neq f$ we have $e \cap f = \emptyset$. In other words, a matching is a set of edges without common vertices.

A vertex v is called *matched* in M (or *saturated* by M) if v is an endopoint of an edge $e \in M$. A matching M is called *perfect* if every vertex in V is matched in M.

Example 8.3 There are a applicants to j different jobs. Each applicant has a subset of jobs they are interested in. Each job opening can accept only one applicant and each applicant can be hired for only one (at most one) job.

To find an assignment of jobs to applicants in such a way that as many applicants as possible gets a job corresponds to find a maximal matching in a bipartite graph $G = (A \cup J, E)$, where

- A is the set of applicants,
- J is the set of jobs,

and for each $a \in A$, $j \in J$ we have $\{a, j\} \in E$ if the applicant a is interested in the job j (see Figure 8.3).

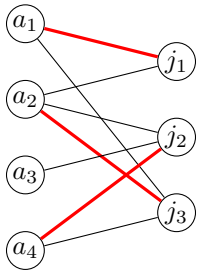


Figure 8.3: A bipartite graph with vertices represeting applicants (left) and jobs (right) with a possible match (in red).

Example 8.4 We have a computer with 2 processors and a set of q jobs (computational). Some of these jobs can run simultaneously (for simplicity let us assume that each job takes the same amount of time). To find the order of computation of jobs such that the total time is the smallest possible we can proceed as follows.

Let us construct a graph with

- set of vertices = jobs,
- edges of the form $\{u, v\} \in E$ if the jobs u and v can run simultanously.

Then a maximal matching in G gives pairs of jobs that can be run simultanously, so that the total time is the smallest possible.

A matching that cannot be extended to a larger matching is called *maximal*. A matching M in G is called *maximum* if for each matching M' in G we have $\#M \geq \#M'$. Note that a maximum matching is maximal, but not every maximal matching is a maximum.

Example 8.5 Let G be the graph represented in Figure 8.4. A maximal matching is represented on the left of the figure. No edge can be added to the matching to obtain a larger one.

A maximum matching is shown on the right of the same figure. Since all vertices are covered by it, the matching is perfect.

Figure 8.4: A maximal matching (left) and a maximal (and perfect) matching (right) for a graph.

8.3 Perfect matching

Obviously, not every graph contains a perfect matching. There are simple necessary conditions.

Proposition 4 *If G has a perfect matching, then $\#V$ is even.*

Proposition 5 *If $G = (V_1 \cup V_2, E)$ is a bipartite graph with a perfect matching, then $\#V_1 = \#V_2$.*

There are also some sufficient conditions, but most of them lead to high-time complexity of the decision procedure.

Recall that the neighbourhood of a vertex v in a graph $G = (V, E)$ is the set $\mathrm{N}_G(v) = \{u \in V \mid \{u, v\} \in E\}$. The *neighbourhood* of a set $S \subset V$ is defined as $\mathrm{N}_G(S) = \bigcup_{v \in S} \mathrm{N}_G(v)$.

Theorem 8.2 (Hall (1953)) *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph. There is a matching M in G such that every $v \in V_1$ is matched in M if and only if for every $S \subset V_1$ we have $\#S \leq \#\mathrm{N}_G(S)$.*

Hall's Theorem 8.2 is also known as the *Marriage Theorem*. Imagine a dating app for heterosexual people used by a a certain number of girls and a certain number of boys. If every group of she-users collectively like at least as many he-users as there are girls in the group, then each girl can marry a boy she likes.

The condition in theorem works for all bipartite graphs but it is not fast to check: since it requires the inequality to be checked for all subsets of V_1, the time complexity is exponential in $\#V_1$.

Recall that a graph G is called regular if there exists a non-negative integer $k \in \mathbb{N}_0$ such that $\delta(G) = \Delta(G) = k$, i.e., every vertex is incident to exactly k edges.

Corollary 5 *Let G be a regular bipartite graph. Then G contains a perfect matching.*

The condition $\delta(G) = \Delta(G)$ in Corollary 5 is quite easy to check, but it holds only for a special (quite small) class of graphs.

8.4 Augmenting paths

Let $G = (V, E)$ be a graph and M a matching in G. A path $P = (v_i)_{i=0}^{k}$ in G is called M-*alternating* if for every $i \in \{1, \dots k-1\}$ we have

$$\{v_{i-1}, v_i\} \in M \quad \Leftrightarrow \quad \{v_i, v_{i+1}\} \notin M,$$

i.e., it is a path in which edges alternate between those in M and those not in M. An M-alternating path is called M-*augmenting* if it starts and ends with a vertex that is not matched in M.

Example 8.6 Let G be the graph in Figure 8.5 and $M = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{d},\mathtt{e}\},\{\mathtt{g},\mathtt{h}\}\}$ a matching in G. The path $P_1 = (\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{d},\mathtt{g},\mathtt{h})$ is M-alternating. The path $P_2 = (\mathtt{c},\mathtt{b},\mathtt{a},\mathtt{d},\mathtt{e},\mathtt{f})$ is both M-alternating and M-augmenting.

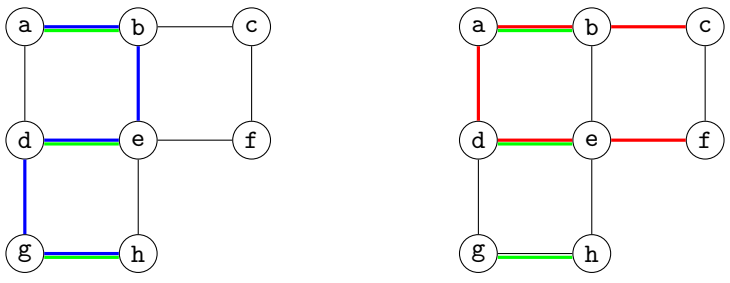


Figure 8.5: An M-alternating path (left) and an M-augmenting path (right).

An M-augmenting path P in a graph G can be used to find a larger matching in G. Indeed, we can consider the matching $M' = M \triangle P = (M \setminus P) \cup (P \setminus M)$. That is, M' is constructed from M in the following way:

- the edges both in P and M are discarded from M;
- the edges in P but not in M are added to M;
- the edges in M but not in P are left in M.

Obviously $\#M' = \#M + 1$. This implies that absence of a M-augmenting path in G is a necessary condition for M to be maximal.

Example 8.7 Let G, M and P_2 as in Example 8.6. The match $M' = M \triangle P_2 = \{\{\mathtt{a},\mathtt{d}\},\{\mathtt{b},\mathtt{c}\},\{\mathtt{e},\mathtt{f}\},\{\mathtt{g},\mathtt{h}\}\}$ is also a matching in the graph (see Figure 8.6)

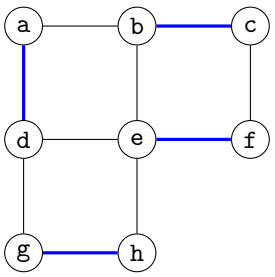


Figure 8.6: A matching in a graph.

The following theorem states that this condition is also sufficient.

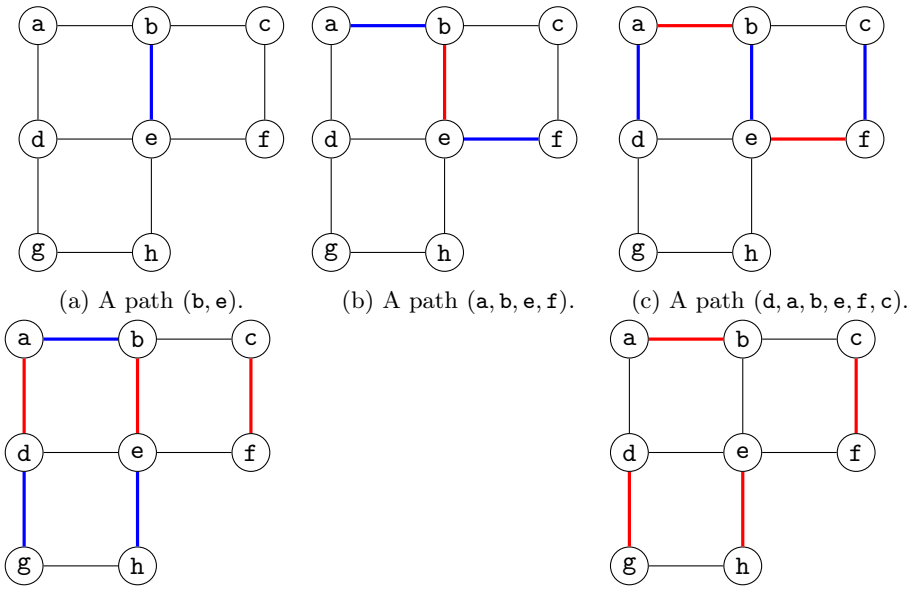
Theorem 8.3 (Berge (1957)) *Let $G = (V, E)$ be a graph, and M a matching in G. M is maximal in G if and only if there is no M-augmenting path in G.*

Thus, given a graph, we have a way to construct a maximum matching: we start with some matching M, e.g., the empty matching, and then search for an M-augmenting path P. If such a path exists, we replace M by $M \triangle P$. We repeat the process until no augmenting path can be found.

Example 8.8 Let G be again the graph in Example 8.6. A possible sequence of matches and corresponding augmenting paths, starting from the empty matching, is shown in Figure 8.7.

- We start with the empty match $M_0 = \emptyset$. An M_0-augmenting path is given by the path $P_1 = (\mathtt{b},\mathtt{e})$ (see Figure 8.7a).
- A larger matching is given by $M_1 = M_0 \triangle P_1 = \{\{\mathtt{b},\mathtt{e}\}\}$. An M_1-augmenting path is given by $P_2 = (\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{f})$ (see Figure 8.7b).

- A larger matching is obtained as $M_2 = M_1 \triangle P_2 = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{e},\mathtt{f}\}\}$. An M_2-augmenting path is given by $P_3 = (\mathtt{d},\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{f},\mathtt{c})$ (see Figure 8.7c).
- A larger matching is given by $M_3 = M_2 \triangle P_3 = \{\{\mathtt{a},\mathtt{d}\},\{\mathtt{b},\mathtt{e}\},\{\mathtt{c},\mathtt{f}\}\}$. An M_3-augmenting path is given by $P_4 = (\mathtt{g},\mathtt{d},\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{h})$ (see Figure 8.7d).
- The matching $M_4 = M_3 \triangle P_4 = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{c},\mathtt{f}\},\{\mathtt{d},\mathtt{g}\},\{\mathtt{e},\mathtt{h}\}\}$ is maximal. One can check that it is also a maximum matching and a perfect one.



(a) A path $(\mathtt{b},\mathtt{e})$. (b) A path $(\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{f})$. (c) A path $(\mathtt{d},\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{f},\mathtt{c})$.

(d) A path $(\mathtt{g},\mathtt{d},\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{h})$. (e) A maximal matching.

Figure 8.7: A sequence of augmenting paths in a graph.

8.5 Hungarian method

Let us show an algorithm for finding a maximum matching M in a given graph G in a systematic way, if such a matching exists.

To do so, let us start with a matching M in G. An *M-alternating tree* is a rooted tree whose root is not matched in M and all root-leaf paths in it are M-alternating (see Example 8.10).

If the M-alternating path not only starts with the unmatched root, but also end with an unmatched vertex, different than the root, then we have found a M-augmenting path.

Example 8.9 Let G and $M = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{d},\mathtt{e}\},\{\mathtt{g},\mathtt{h}\}\}$ be the graph and the matching on the left of Figure 8.8 (see also previous lecture).

The tree $T(\mathtt{c})$ shown on the right Figure 8.8 is an M-alternating tree. The path $(\mathtt{c}, \mathtt{b}, \mathtt{a}, \mathtt{d}, \mathtt{e}, \mathtt{f})$ is M-augmenting in the tree .

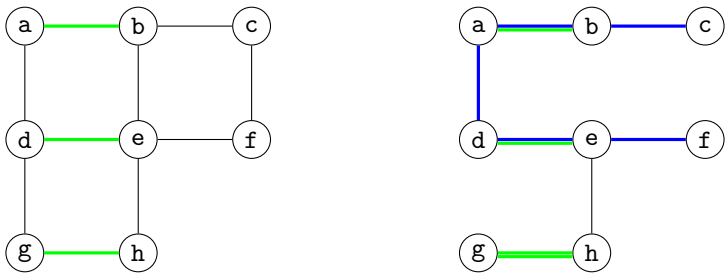


Figure 8.8: A graph G with a matching M (left) and an M-alternating tree and an M-augmenting path (right).

The following algorithm was developed by the Hungarian mathematician Egerváry (1931) and it is also known as the *Hungarian Algorithm.*

Hungarian Algorithm. Let us divide the vertices in the tree according to the parity of their level: even levels of an alternating tree are labeled S (*sudý*), while vertices of the odd leveles are labeled L (*lichý*). We keep unlabeled the vertices that are currently in no alternating tree.

Initially, all vertices are unlabeled, and all edges are unexplored.

At each step we have two options.

(S1) Choose a unlabeled and unmatched vertex r, label it S, and make it the root of a new tree.

(S2) Choose an unexplored edge $\{u, v\}$, where u is labeled S, and explore it as follows (5 options, based on v):

a) if v is unlabeled and unmatched, we label v as L, and we have found an M-augmenting path composed from the path from the root of u's tree to u plus the edge $\{u, v\}$;

b) if v is unlabeled and matched, let w be the vertex matched to v, e.g., the vertex such that $\{v, w\} \in M$; we label v as L and w as S and we add the edges $\{u, v\}$ and $\{v, w\}$ to the tree;

c) if v is labeled L, we do nothing;

d) if v is labeled S and does not belong to the same tree as u, we have discovered an M-augmenting path composed of the path from the root of u's tree to u, the edge $\{u, v\}$, and then the path from v to the root of v's tree;

e) if v is labeled S and belongs to the same tree as u, we have discovered an odd cycle in the graph with alternating edges; such an odd

cycle is called a *blossom*; there is a special procedure how to treat blossoms, but the basic version of the Hungarian algorithm is used only on bipartite graphs in which the blossom (odd cycle) cannot occur.

The labelling process stops either when an M-augmenting path – which can be used to construct larger matching - is found, i.e., on Step 2a) or Step 2d); or when there are no more unlabeled vertices, unmatched edges nor unexplored edges incident with at least one S vertex.

Proposition 6 *Let G be a bipartite graph of size m. If there is a M-augmenting path in G, the Hungarian algorithm finds it in $\mathcal{O}(m)$ steps.*

The previous proposition immediately gives as an $\mathcal{O}(m \cdot n)$ time algorithm for finding a maximal matching in a bipartite graph of order n and size m.

Example 8.10 Let G be the bipartite graph in Figure 8.9 with set of vertices $V = V_1 \cup V_2$, where $V_1 = \{a_1, a_2, a_3, a_4, a_5\}$ and $V_2 = \{b_1, b_2, b_3, b_4\}$. Let us consider the matching $M = \{\{a_2, b_1\}, \{a_3, b_3\}, \{a_4, b_2\}\}$ in G (left of Figure 8.9).

Figure 8.9: Two matchings in a bipartite graph.

The alternating forest can be constructed using the Hungarian algorithm, e.g., as follows (see also Figures 8.9a to 8.9g).

Step	Type	vertices/edges	matching/path	Figure
			M	8.9 left
1st	S1	$r = a_5$		8.9a
2nd	S2b)	$u = a_5,\ v = b_3,\ w = a_3$		8.9b
3rd	S1	$r = a_1$		8.9c
4th	S2b)	$u = a_1,\ v = b_1,\ w = a_2$		8.9d
5th	S2c)	$u = a_2,\ v = b_3$		8.9e
6th	S2b)	$u = a_1,\ v = b_2,\ w = a_4$		8.9f
7th	S2a)	$u = a_4,\ v = b_4$	$P = (a_1, b_2, a_4, b_4)$	8.9g
			$M' = M \triangle P$	8.9 right

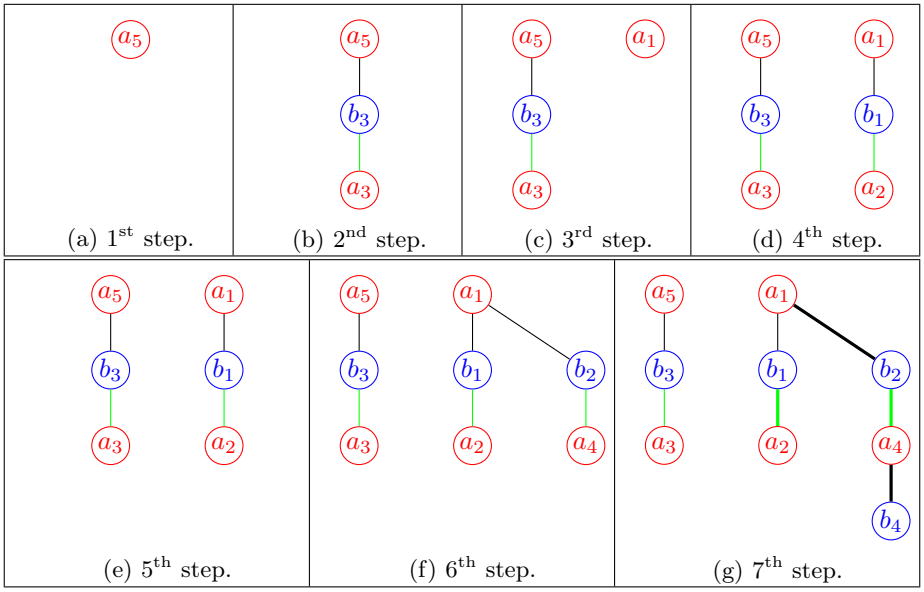


Figure 8.9: Steps of the Hungarian algorithm.

In the 7^{th} step we have discovered an M-augmenting path $P = (a_1, b_2, a_4, b_4)$. We can obtain a larger matching by considering

$$M' = M \triangle P = \{\{a_1, b_2\}, \{a_2, b_1\}, \{a_3, b_3\}, \{a_4, b_4\}\}$$

(see right of Figure 8.9).

Note that M' is maximal in G since all vertices in V_2 are matched in M'.

Let us modify the Hungarian algorithm in order to find a maximal matching in a graph that is not bipartite, by adding the procedure used whenever a blossom is found. This new algorithm, used for finding maximal/perfect matching in a general graph, is called *Edmond's Algorithm*.

Edmond's Algorithm. Similarly to the Hungarian algorithm, we explore the graph and construct an M-alternating forest.

There is only one difference: since we consider a general graph, we can have steps of the type S2e). This corresponds to an occurence of an odd M-alternating cycle in G, i.e., a blossom. When we detect such a blossom, we contract it into a single pseudovertex, label it S, and then we continue with the traversal. After we expand the blossom to its original vertices.

Example 8.11 Consider the graph $G = (\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}, \{\{\mathtt{A}, \mathtt{B}\}, \{\mathtt{A}, \mathtt{C}\}, \{\mathtt{B}, \mathtt{C}\}, \{\mathtt{B}, \mathtt{D}\}\})$ with empty matching $M = \emptyset$. (see Figure 8.10).

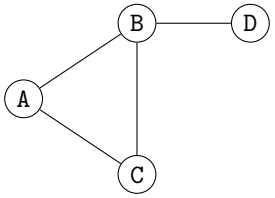
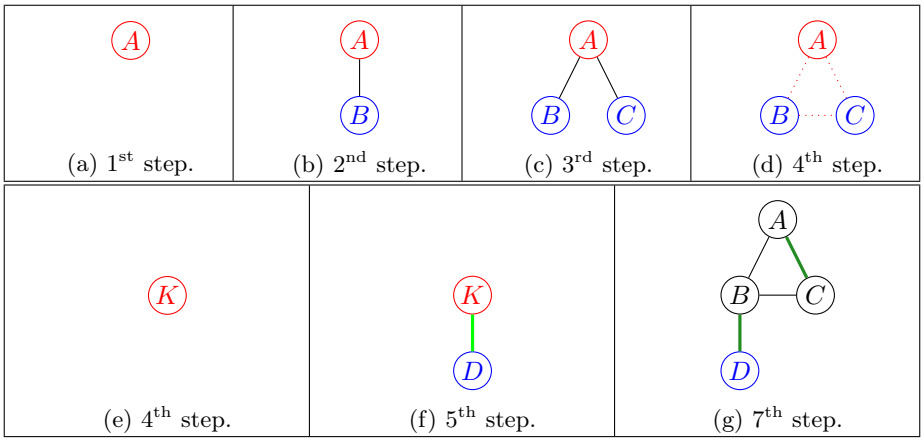


Figure 8.10: A graph G with an odd cycle.

The steps of Edmond's Algorithm, starting from the empty matching, are as follows.

Step	Type	vertices/edges	matching/path/blossom	Figure
			$M_0 = \emptyset$	8.10
1^{st}	S1	$r = A$		8.10a
2^{nd}	S2b)	$u = A, v = B$		8.10b
3^{rd}	S2b)	$u = A, v = C$		8.10c
4^{th}	S2e)	$u = B, v = C$	contract $\{A, B, C\}$ to K	8.10d, 8.10e
5^{th}	S2a)	$u = K, v = D$	$P_1 = (K, D)$	8.10f
			$M_1 = M_0 \triangle P_1 = \{\{K, D\}\}$	
	Lift blossom		expand K to $\{A, B, C\}$ $P_2 = (A, C, B, D)$ $M_2 = \{\{A, C\}, \{B, D\}\}$	8.10g



(a) 1^{st} step. (b) 2^{nd} step. (c) 3^{rd} step. (d) 4^{th} step.

(e) 4^{th} step. (f) 5^{th} step. (g) 7^{th} step.

Figure 8.10: Steps of Edmond's Algorithm.

Exercise 10 Using the Hungarian algorithm and starting from the empty matching $M_0 = \emptyset$, find a maximum matching M for the graph G in Figure 8.11

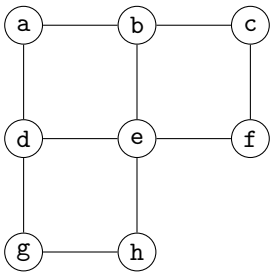


Figure 8.11: A graph with empty matching $M_0 = \emptyset$.

Solution of Exercise 10 We apply the Hungarian algorithm, starting from the matching $M_0 = \emptyset$.

1. Consider the empty matching $M_0 = \emptyset$.

   (a) Let us apply Step 1 by choosing the vertex a and labelling it S.

   (b) Let us apply Step 2a) by choosing the edge $\{\mathtt{a}, \mathtt{b}\}$ and labelling the vertex b as L.

   We find a new M_0-incrementing path $P_1 = (\mathtt{a}, \mathtt{b})$ (see Figure 8.12).

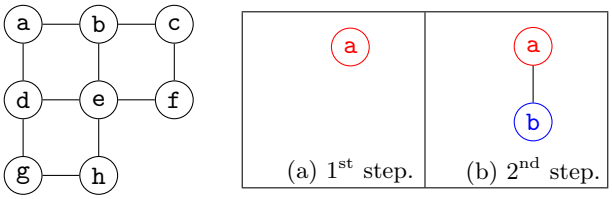


Figure 8.12: First run of the Hungarian algorithm on G.

   We can use this path to find a new matching

$$M_1 = M_0 \triangle P_1 = \{\{\mathtt{a}, \mathtt{b}\}\}.$$

2. Consider the matching $M_1 = \{\{\mathtt{a}, \mathtt{b}\}\}$.

   (a) Let us apply Step 1 by choosing the vertex c and labelling it S.

   (b) Let us apply Step 2b) by choosing the edge $\{\mathtt{c}, \mathtt{b}\}$, with b matched via the edge $\{\mathtt{b}, \mathtt{a}\} \in M_1$; we label the vertex b as L and the vertex a as S.

   (c) Let us apply Step 2a) starting from c, by choosing the edge $\{\mathtt{c}, \mathtt{f}\}$; we label the vertex f as L.

   We find a new M_1-incrementing path $P_2 = (\mathtt{c}, \mathtt{f})$ (see Figure 8.13).

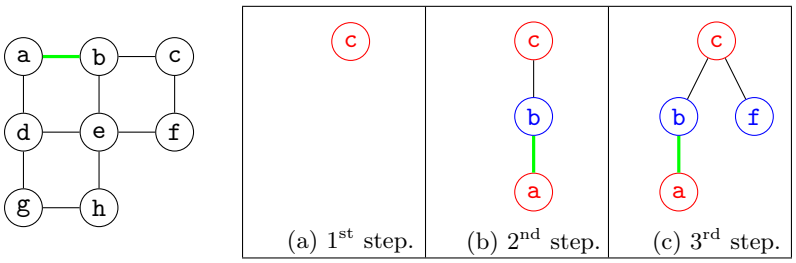


Figure 8.13: Second run of the Hungarian algorithm on G.

We can use this path to find a new matching

$$M_2 = M_1 \triangle P_2 = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{c},\mathtt{f}\}\}.$$

3. Consider the matching $M_2 = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{c},\mathtt{f}\}\}$.
   (a) Let us apply Step 1 by choosing the vertex d and labelling it S.
   (b) Let us apply Step 2b) by choosing the edge $\{\mathtt{d},\mathtt{a}\}$, with a matched via the edge $\{\mathtt{a},\mathtt{b}\} \in M_2$; we label the vertex a as L and the vertex b as S.
   (c) Let us apply Step 2a) starting from b, by choosing the edge $\{\mathtt{b},\mathtt{e}\}$; we label the vertex e as L.

   We find a new M_2-incrementing path $P_3 = (\mathtt{d},\mathtt{a},\mathtt{b},\mathtt{e})$ (see Figure 8.14).

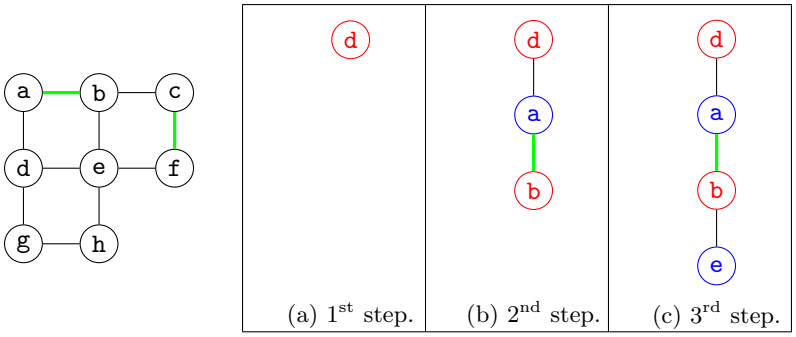


Figure 8.14: Third run of the Hungarian algorithm on G.

We can use this path to find a new matching

$$M_3 = M_2 \triangle P_3 = \{\{\mathtt{a},\mathtt{d}\},\{\mathtt{b},\mathtt{e}\},\{\mathtt{c},\mathtt{f}\}\}.$$

4. Consider the matching $M_3 = \{\{\mathtt{a},\mathtt{d}\},\{\mathtt{b},\mathtt{e}\},\{\mathtt{c},\mathtt{f}\}\}$.
   (a) Let us apply Step 1 by choosing the vertex g and labelling it S.

(b) Let us apply Step 2b) by choosing the edge $\{\mathtt{g},\mathtt{d}\}$, with $\mathtt{d}$ matched via the edge $\{\mathtt{d},\mathtt{a}\} \in M_3$; we label the vertex $\mathtt{d}$ as L and the vertex $\mathtt{a}$ as S.

(c) Let us apply Step 2b) by choosing the edge $\{\mathtt{a},\mathtt{b}\}$, with $\mathtt{b}$ matched via the edge $\{\mathtt{b},\mathtt{e}\} \in M_3$; we label the vertex $\mathtt{b}$ as L and the vertex $\mathtt{e}$ as S.

(d) Let us apply Step 2a) starting from $\mathtt{e}$, by choosing the edge $\{\mathtt{e},\mathtt{h}\}$; we label the vertex $\mathtt{h}$ as L.

We find a new M_3-incrementing path $P_4 = (\mathtt{g},\mathtt{d},\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{h})$ (see Figure 8.15).

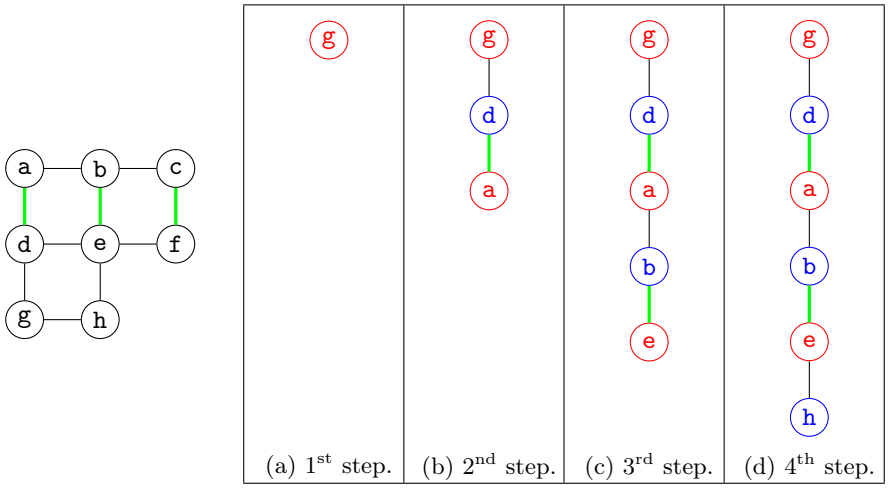


Figure 8.15: Fourth run of the Hungarian algorithm on G.

We can use this path to find a new matching

$$M_4 = M_3 \triangle P_4 = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{c},\mathtt{f}\},\{\mathtt{d},\mathtt{g}\},\{\mathtt{e},\mathtt{h}\}\}.$$

The matching M_4 is maximal (and perfect) in G (see Figure 8.16).

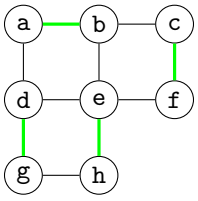


Figure 8.16: A maximal matching in G.

A perfect matching is also called a 1-*factor*, since it correspond to a spanning subgraph that is 1-regular. When the set of edges of a graph can be partitioned into k different 1-factors, we say that the graph has a 1-factorisation.

Theorem 8.4 *Let $n \geq 1$. The complete graph K_{2n} is 1-factorizable in a partition of $2n-1$ disjoint 1-factors.*

Example 8.12 The edges of the graph K_4 can be partitioned in three perfect matching $M_1 = \{\{\mathtt{a},\mathtt{b}\},\{\mathtt{c},\mathtt{d}\}\}$, $M_2 = \{\{\mathtt{a},\mathtt{c}\},\{\mathtt{b},\mathtt{d}\}\}$ and $M_3 = \{\{\mathtt{a},\mathtt{d}\},\{\mathtt{b},\mathtt{c}\}\}$ (see Figure 8.17).

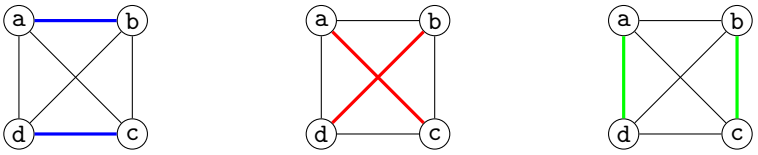


Figure 8.17: A 1-factorisation of K_4.

Chapter 9

Edge colouring

Let $G = (V, E)$ be a graph. An *edge colouring* by k colours of G, with k a positive integer, is a mapping $\varphi : E \to \hat{k}$, where $\hat{k} = \{1, 2, \ldots, k\}$.

An edge colouring is called *proper* if for every two distinct edges $e, f \in E$ we have

$$e \cap f \neq \emptyset \quad \Longrightarrow \quad \varphi(e) \neq \varphi(t),$$

i.e., edges sharing a common vertex have different colours.

Example 9.1 Let $G = (V, E)$ be the graph in Figure 9.1 and $\varphi_1, \varphi_2 : E \to \{1, 2, 3\}$ and $\varphi_3 : E \to \{1, 2, 3, 4\}$ be the three edge-colouring of G defined by

$$\varphi_1 : \begin{cases} \{\mathtt{a}, \mathtt{b}\} \mapsto 1 \\ \{\mathtt{a}, \mathtt{e}\} \mapsto 2 \\ \{\mathtt{b}, \mathtt{c}\} \mapsto 2 \\ \{\mathtt{b}, \mathtt{e}\} \mapsto 3 \\ \{\mathtt{c}, \mathtt{d}\} \mapsto 3 \\ \{\mathtt{d}, \mathtt{e}\} \mapsto 1 \end{cases}, \qquad \varphi_2 : \begin{cases} \{\mathtt{a}, \mathtt{b}\} \mapsto 1 \\ \{\mathtt{a}, \mathtt{e}\} \mapsto 1 \\ \{\mathtt{b}, \mathtt{c}\} \mapsto 2 \\ \{\mathtt{b}, \mathtt{e}\} \mapsto 3 \\ \{\mathtt{c}, \mathtt{d}\} \mapsto 2 \\ \{\mathtt{d}, \mathtt{e}\} \mapsto 3 \end{cases} \quad \text{and} \quad \varphi_3 : \begin{cases} \{\mathtt{a}, \mathtt{b}\} \mapsto 1 \\ \{\mathtt{a}, \mathtt{e}\} \mapsto 2 \\ \{\mathtt{b}, \mathtt{c}\} \mapsto 3 \\ \{\mathtt{b}, \mathtt{e}\} \mapsto 4 \\ \{\mathtt{c}, \mathtt{d}\} \mapsto 1 \\ \{\mathtt{d}, \mathtt{e}\} \mapsto 3 \end{cases}.$$

The first and the last edge-colourings are proper, while the second one is not, since, e.g., $\varphi_2(\{\mathtt{a}, \mathtt{b}\}) = \varphi_2(\{\mathtt{a}, \mathtt{c}\}) = 2$.

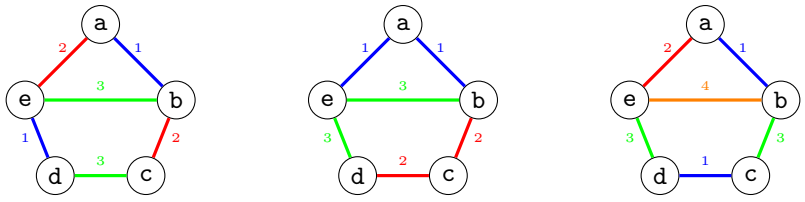


Figure 9.1: A graph with three different edge-colouring.

The *chromatic index* (or *edge chromatic number*) of G, denoted $\chi'(G)$, is the smallest positive integer $k \in \mathbb{N}$ such that G has a proper edge colouring using k colours.

Given a graph $G = (V, E)$ with a proper k-edge colouring, since all the edges incident with a vertex have different colours, we get that $d_G(v) \leq k$ for every $v \in V$. Hence $\Delta(G) \leq k$.

Thus, we have the following lower estimate on the chromatic index of G:

$$\chi'(G) \geq \Delta(G).$$

Let φ be a proper k-edge colouring of $G = (V, E)$, and let us consider the set $\varphi^{-1}(i)$ for $i \in \hat{k}$. Since this is the set of all edges of G having colour i in the proper edge colouring φ, no two edges of this set are adjacent. Consequently for every $i \in \hat{k}$, $\varphi^{-1}(i)$ is a matching on G.

Example 9.2 (Timetabling Problem) In a school there are m teachers u_1, u_2, ..., u_m, and n classes (groups of students) $t_1, t_2, \ldots, t_n$.

Given that a teacher u_i is required to each class t_j for $p_{i,j}$ periods, we want to schedule a complete timetable in the minimum possible number of periods.

We can represent teaching requirements by a bipartite graph $G = (U \cup T, E)$, where $U = \{u_1, u_2, \ldots, u_m\}$, $T = \{t_1, t_2, \ldots, t_n\}$, and vertices u_i and t_j are joined by $p_{i,j}$ edges (i.e., we allow multi-edges in this application and G is not necessarily a simple graph).

Since in any period a teacher can teach at most one class, and each class can be taught by at most one teacher, a teaching schedule for one period corresponds to a matching in G, and, conversely, each matching in G corresponds to a possible assignment of teachers to classes for one period.

Our problem, therefore, is to partition the edges of G into as least matchings as possible or, equivalently, to find proper edge-colouring of G with as least colours as possible.

By estimate $\Delta(G) \leq \chi'(G)$, we know that at least p periods are necessary, where p is a number such that no teacher teaches for more than p periods and no class is taught for more than p periods.

For bipartite graphs we can even be more precise.

Theorem 9.1 (König (1931)) *Let G be a bipartite graph. Then $\Delta(G) = \chi'(G)$.*

That is, for a bipartite graph the lower bound on $\chi'(G)$ in terms of the maximal degree in G is the exact value of $\chi'(G)$.

Example 9.3 (continuation of Example 9.2) Let us suppose that we have 4 teachers, 3 classes, with $p_{i,j}$ described by the matrix

$$P = (p_{i,j})_{i,j} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 3 \end{pmatrix}.$$

The problem could be solved by considering the bipartite graph in Figure 9.2. Since the maximum degree of the graph is 4 we have that the chromatic index is also 4. An example of 4-colouring, using the colours $\{1, 2, 3, 4\}$ is shown of Figure 9.2.

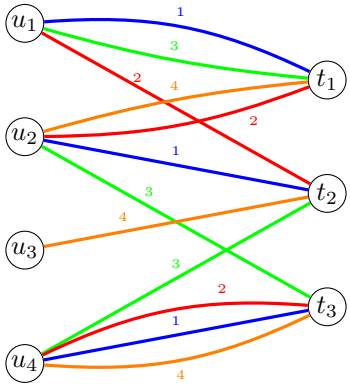


Figure 9.2: A 4-colouring of a bipartite graph.

If G is not bipartite, we cannot necessarily conclude that $\chi' = \Delta$.

Example 9.4 Let G be the graph in Figure 9.3. The maximal degree of the graph is $\Delta(G) = 3$. A proper edge 4-colouring is shown in Figure 9.3.

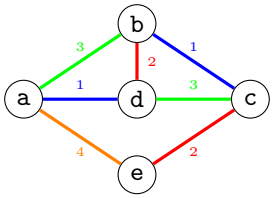


Figure 9.3: A graph.with a 4-edge colouring

One can check that no proper 3-edge-colouring exists for the graph G.

For regular graphs we have the following interesting corollary of Theorem 8.4 (remember that the trivial graph K_1 has no edges).

Theorem 9.2 *For every $n \geq 1$ we have $\chi'(K_{2n}) = 2n - 1$ and $\chi'(K_{2n+1}) = 2n + 1$.*

Example 9.5 Some possible edge-colourings of the graphs K_2, K_3, K_4, K_5 and K_6 are shown in Figure 9.4.

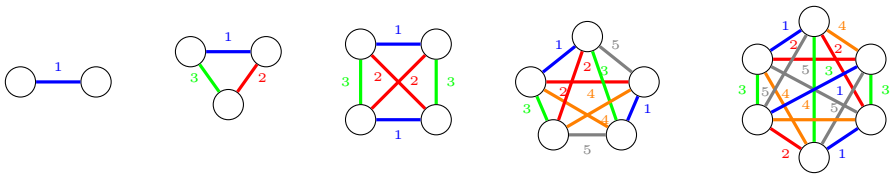


Figure 9.4: Edge colourings of the graphs K_2, K_3, K_4, K_5 and K_6.

Exercise 11 (Schoolgirl problem) Eight schoolgirls go for a walk in pairs every day. Show how they can arrange their outings so that each girl has different companions on different days of the week.

Solution of Exercise 11 The problem consists in finding a 7-edge-colouring of the graph K_8. A possible solution is given in Figure 9.5. The edge-colourings $\varphi_1, \varphi_2, \ldots, \varphi_8$ are such that for every i we have

$$\varphi_1^{-1}(E) = \{\{1,2\},\{3,4\},\{5,6\},\{7,8\}\},$$

$$\varphi_2^{-1}(E) = \{\{1,3\},\{2,4\},\{5,7\},\{6,8\}\},$$
$$\varphi_3^{-1}(E) = \{\{1,4\},\{2,3\},\{5,8\},\{6,7\}\},$$
$$\varphi_4^{-1}(E) = \{\{1,5\},\{2,6\},\{3,7\},\{4,8\}\},$$
$$\varphi_5^{-1}(E) = \{\{1,6\},\{2,5\},\{3,8\},\{4,7\}\},$$
$$\varphi_6^{-1}(E) = \{\{1,7\},\{2,8\},\{3,5\},\{4,6\}\},$$
$$\varphi_7^{-1}(E) = \{\{1,8\},\{2,7\},\{3,6\},\{4,5\}\}.$$

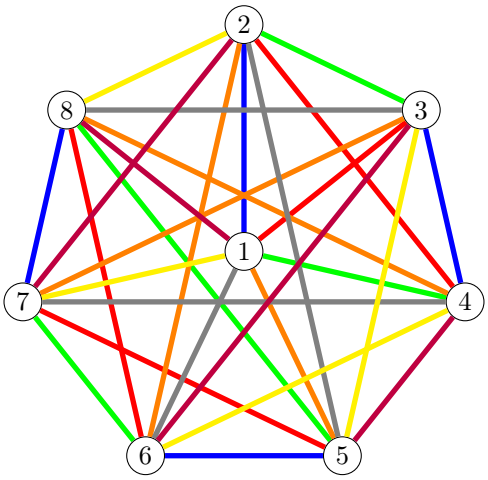


Figure 9.5: A 7-edge colourings of the graphs K_8.

For non-bipartite and non-regular graphs we do not have an exact formula. However, an important theorem by Vizing asserts that for a simple graph G the actual value of $\chi'(G)$ is always close to $\Delta(G)$.

Theorem 9.3 (Vizing (1964)) *Let G be a simple graph. Then*

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1.$$

Chapter 10

Vertex colouring

Let $G = (V, E)$ be a graph. A *vertex colouring* by k colours, or *vertex k-colouring*, of G is a mapping $\varphi : V \to \hat{k}$, with k a positive integer. Similarly to edge-colouring, a vertex-colouring φ is called *proper* if for every two distinct vertices $u, v \in V$ we have

$$\{u, v\} \in E \quad \Rightarrow \quad \varphi(u) \neq \varphi(v),$$

i.e., vertices connected by an edge have different colours.

A graph admitting a vertex k-colouring is called *k-colourable*.

Example 10.1 Let $G = (V, E)$ be the graph in Figure 10.1, and $\varphi_1, \varphi_2 : V \to \{1, 2, 3\}$ and $\varphi_3 : V \to \{1, 2, 3, 4, 5\}$ be the three vertex-colourings of G defined by

$$\varphi_1(\mathtt{a}) = \varphi_1(\mathtt{d}) = 1, \quad \varphi_1(\mathtt{b}) = 2 \quad \text{and} \quad \varphi_1(\mathtt{c}) = \varphi_1(\mathtt{e}) = 3;$$
$$\varphi_2(\mathtt{a}) = 1, \quad \varphi_2(\mathtt{b}) = \varphi_2(\mathtt{e}) = 2 \quad \text{and} \quad \varphi_2(\mathtt{c}) = \varphi_2(\mathtt{d}) = 3;$$
$$\varphi_3(\mathtt{a}) = 1, \quad \varphi_3(\mathtt{b}) = 2, \quad \varphi_3(\mathtt{c}) = 3, \quad \varphi_3(\mathtt{d}) = 4, \quad \text{and} \quad \varphi_3(\mathtt{e}) = 5.$$

The first and last vertex-colourings are proper, while the second one is not, since, e.g., $\varphi_2(\mathtt{b}) = \varphi_2(\mathtt{e}) = 2$.

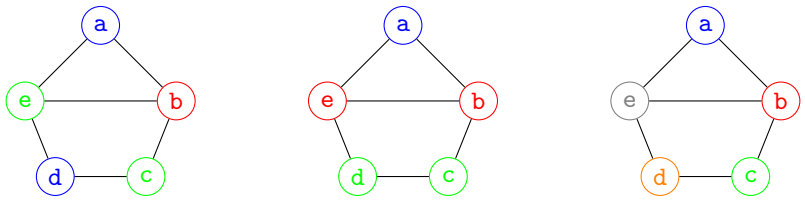


Figure 10.1: A graph with three different vertex-colouring.

The *chromatic number* of G, denoted by $\chi(G)$, is the smallest $k \in \mathbb{N}$ such that G has a proper vertex colouring using k colours (also called proper vertex k-colouring). A graph having chromatic number k is called *k-chromatic*.

Example 10.2 The graph G in Figure 10.2, known as the *Hajós graph*, has chromatic number $\chi(G) = 4$. Indeed, it is possible to show that no proper vertex 3-colouring exists, while a proper vertex 4-colouring is given by

$$\varphi(\mathtt{a}) = \varphi(\mathtt{d}) = 1, \quad \varphi(\mathtt{b}) = \varphi(\mathtt{e}) = 2, \quad \varphi(\mathtt{c}) = \varphi(\mathtt{f}) = 3 \quad \text{and} \quad \varphi(\mathtt{g}) = 4.$$

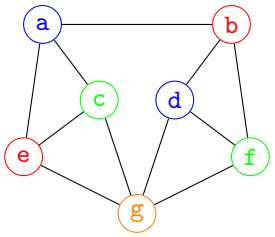


Figure 10.2: The Hajós graph.

Example 10.3 (Examination Scheduling) Let us assume that we have m students who are to take n exams during an examining period. Lecturers aim to schedule the exams in such a way that there are are no conflicts, i.e., they seek a conflict-free schedule with the minimum number of time slots.

We can construct a graph $G = (V, E)$ with

- set of vertices V given by the exams, i.e., $V = \{z_1, z_2, \ldots, z_n\}$,
- set of edges E given by connecting exams z_i and z_j if there is a student who is supposed to take both z_i and z_j.

The chromatic number of such a graph gives the minimal number of time slots needed for a conflict-free scheduling of exams.

An example is given in Figure 10.3, where the set of vertices (exams) is partitioned by a 4-colouring φ as

$$\varphi^{-1}(1) = \{z_1, z_3\}, \quad \varphi^{-1}(2) = \{z_2\} \quad \text{and} \quad \varphi^{-1}(3) = \{z_4, z_5\}.$$

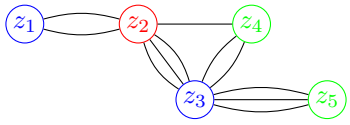


Figure 10.3: A graph representing the scheduling of exams.

Example 10.4 (Chemical Storage) A company manufactures n chemicals c_1, c_2, …, c_n. Certain pairs of these chemicals are incompatible - they would cause an explosion if brought into contact with each other.

The company wishes to partition its warehouse into compartments so that incompatible chemicals are stored separately. What is the least number of compartments into which the warehouse should be partitioned?

The solution is very similar to the exam scheduling in Example 10.3.

We construct a graph $G = (V, E)$, with vertices $V = \{c_1, c_2, \ldots, c_n\}$ by joining c_i and c_j with an edge in E if and only if the chemicals c_i and c_j are incompatible.

It is easy to see that the required least number of compartments is equal to the chromatic number of G.

An example is given in Figure 10.4 where the set of vertices is given by the chemicals **Na** (Sodium), **Cl** (Chlorine), **K** (Potassium), **F** (Fluorine), **Li** (Lithium) and $\mathbf{H_2O}$ (Water) and a possible vertex-colouring is given by

$$\varphi(\mathbf{Na}) = \varphi(\mathbf{K}) = \varphi(\mathbf{Li}) = 1, \quad \varphi(\mathbf{Cl}) = \varphi(\mathbf{H_2O}) = 2 \quad \text{and} \quad \varphi(\mathbf{F}) = 3.$$

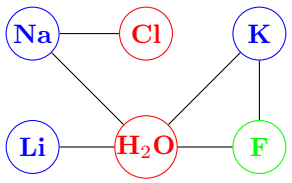


Figure 10.4: A graph representing chemicals not to be stored together.

Clearly only loopless graphs admit proper vertex-colourings. Moreover, a graph is k-colourable if and only if its underlying simple graph if k-colourable. Thus, in the rest of this section we can focus only on simple graphs.

Unfortunately, no good algorithm is known for determining the chromatic number of a generic graph. Only for a few kind of graphs we know their chromatic numbers.

The following Proposition is immediate, since a trivial graph has only one vertex and no edges.

Proposition 7 $\chi((V, \emptyset)) = 1$.

Figure 10.5: A colouring of the trivial graph.

Similarly, we have the following trivial result for the regular graph K_n, where each pair of vertices is connected by an edge.

Proposition 8 $\chi(K_n) = n$.

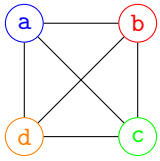


Figure 10.6: A colouring of the graph K_4.

This is actually a characterisation, since for every graph $G = (V, E)$ of order n that is not the complete graph, there exists at least a pair of vertices $u, v \in V$, such that $\{u, v\} \notin E$. So we can colour u and v with the same colour.

Bipartite graphs are exactly the ones who are 2-colourable. That is, we have the following result

Proposition 9 *$\chi(G) = 2$ if and only if G is a bipartite graph.*

Thus, it is quite easy to check whether a graph G on n vertices has

- $\chi(G) = 1 \quad \Leftrightarrow \quad$ is there an edge in G?
- $\chi(G) = n \quad \Leftrightarrow \quad$ is $G = K_n$?
- $\chi(G) = 2 \quad \Leftrightarrow \quad$ is G bipartite? $\quad \Leftrightarrow \quad$ Does G contain an odd cycle? (There is a linear-time algorithm to check the existence of an odd cycle).

On the other hand, determining whether $\chi(G) = 3$ for a given graph G is a $\mathcal{NP}$-complete problem (the class of most "difficult" decision problems).

There is a quite simple *greedy algorithm* which can be used to find a proper colouring of G. It is a linear-time algorithm, but it does not – in general – give the minimum number of colours possible.

The algorithm processes the vertices in the given order, assigning colours successively: each vertex is given the colour (from the set $\{1, 2, 3, \ldots\}$), with the smallest number that is not already used by one of its neighbours.

The resulting colouring depends on the ordering of vertices.

Example 10.5 Consider the following colouring of the same graph G, where the vertices are relabeled and thus reordered (equivalently, we consider two isomorphic graphs).

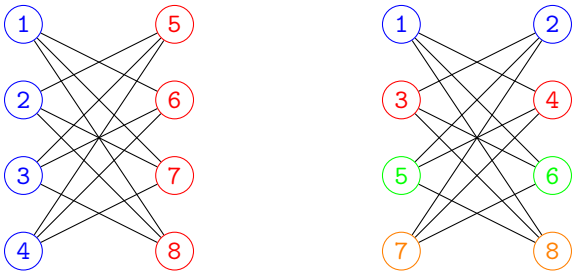


Figure 10.7: A graph with two different vertex-colouring.

With the first ordering, we obtain the 2-colouring $\{\{1,2,3,4\},\{5,6,7,8\}\}$, while with the second we obtain the 4-colouring $\{\{1,2\},\{3,4\},\{5,6\},\{7,8\}\}$.

Even though there always exists an ordering of vertices that produces a proper colouring, such ordering is (in general) hard to find.

A commonly used strategy is to place high-degree vertices earlier than low-degree ones.

The greedy algorithm at least gives us an upper bound of the chromatic number of G: in every step the considered vertex has at most $\Delta(G)$ neighbours. Thus at most $\Delta(G)$ colours are "blocked", there is always a proper vertex colouring of G using $\Delta(G)+1$ colours.

Theorem 10.1 *Let $G=(V,E)$ be a graph. Then $\chi(G) \leq \Delta(G)+1$.*

On the one hand, this upper bound is tight, since $\chi(K_n)=n$ and $\Delta(K_n)=n-1$. On the other hand, there are graphs for which the difference $\Delta(G)+1-\chi(G)$ can be arbitrary large.

Example 10.6 Let $K_{1,n}$ be the n-star. Since $K_{1,n}$ is bipartite, its chromatic number is $\chi(K_{1,n})=2$, but $\Delta(K_{1,n})=n$.

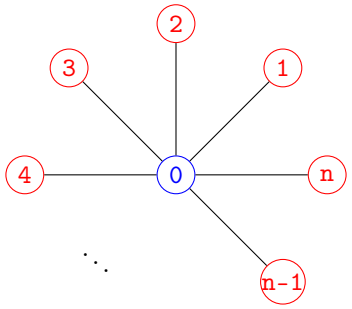


Figure 10.8: The n-star $K_{1,n}$.

Exercise 12 The graph in Figure 10.9 is called *Chvátal graph*. It is a 4-regular graph with 12 vertices. Show that the graph is 4-chromatic.

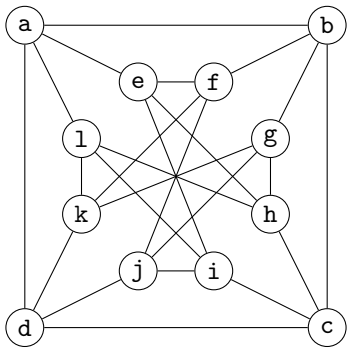


Figure 10.9: The Chvátal graph.

Solution of Exercise 12 The Chvátal graph does not have any vertex 3-colouring since it is 4-regular. A possible vertex 4-colouring is shown in Figure 10.10 and is given by

$$\varphi^{-1}(1) = \{\mathtt{a}, \mathtt{c}, \mathtt{f}, \mathtt{g}\},\ \varphi^{-1}(2) = \{\mathtt{b}, \mathtt{d}, \mathtt{e}, \mathtt{h}\},\ \varphi^{-1}(3) = \{\mathtt{i}, \mathtt{k}\},\ \varphi^{-1}(4) = \{\mathtt{j}, \mathtt{l}\}.$$

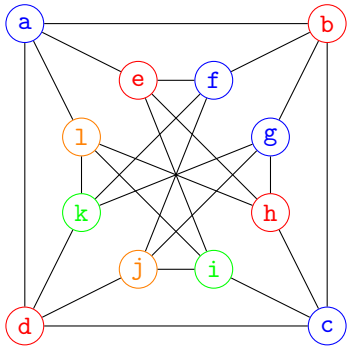


Figure 10.10: The Chvátal graph with a vertex 4-colouring.

10.1 Clique and stability number

There are other known bound on the chromatic number of a graph. To state them, let us give a couple of more definitions.

A *clique* in a graph $G = (V, E)$ is a set $S \subseteq V$ such that $\binom{S}{2} \subseteq E$, i.e., every pair of distinct vertices in S is adjacent (connected by an edge).

Clearly every vertex $v \in V$ forms a clique $\{v\}$. Similarly, for ever edge $\{u, v\} \in E$ we have that $\{u, v\}$ is a clique.

The *clique number* of G, denoted $\omega(G)$, is the size of the largest clique in G.

Example 10.7 Consider the complete graph K_4. The sets $\{\mathtt{a}\}$, $\{\mathtt{b},\mathtt{c},\mathtt{d}\}$ (see left of Figure 10.11) are cliques in K_4 (see rleft of Figure 10.11). Since the graph is regular, also the set $V = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$ is a clique (see right of Figure 10.11), so the clique number of the graph is $\omega(K_4) = 4$.

Figure 10.11: Three cliques of the complete graph K_4.

In general, the clique number of K_n is $\omega(K_n) = n$ for every $n \in \mathbb{N}$.

Example 10.8 Let G be the Hajós graph seen in Example 10.2. The sets $\{\mathtt{b}\}$, $\{\mathtt{d},\mathtt{f}\}$, $\{\mathtt{a},\mathtt{c},\mathtt{e}\}$ are cliques in G (see Figure 10.12). One can check that we do not have any clique of size 4, so the clique number of the graph is $\omega(G) = 3$.

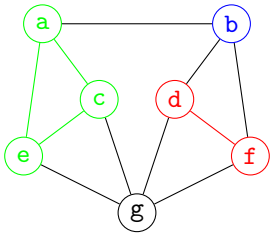


Figure 10.12: Three cliques of the Hajós graph.

A *stable set* (also called *independent set*) in $G = (V, E)$ is a set $S \subseteq V$ such that $\binom{S}{2} \cap E = \emptyset$, that is, no two vertices in S are adjacent.

The *stability number* (or *independence number*) of G, denoted $\alpha(G)$, is the number of vertices in a largest stable set in G.

Example 10.9 Consider the complete graph K_4. The set $\{\mathtt{a}\}$ is a stable set of K_4. Since every vertex is adjacent to every other vertex, the largest stable set has size 1, i.e., $\alpha(K_4) = 1$.

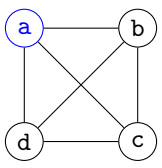


Figure 10.13: A stable set in the complete graph K_4.

In general, the stability number of K_n is $\omega(K_n) = 1$ for every $n \in \mathbb{N}$.

Example 10.10 Let G be the Hajós graph seen in Example 10.2. The sets {b}, {a, d}, {c, f} (Figure 10.14) are stable sets in G. One can check that we do not have any stable set of size bigger or equal to 3, so the stability number of the graph is $\alpha(G) = 2$.

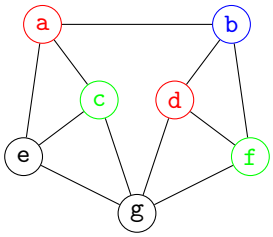


Figure 10.14: Three stable sets of the Hajós graph.

There is a quite obvious relation between proper colourings and independent sets.

Let φ be a proper k-vertex colouring of G. Then, the set $\varphi^{-1}(i)$, for $i \in \hat{k}$ (i.e., the set of vertices coloured by the colour i) is an independent set in G. Thus for every $i \in \hat{k}$ we have $\#\varphi^{-1}(i) \leq \alpha(G)$, which gives the following result.

Theorem 10.2 *Let $G = (V, E)$ be a graph. Then, $\#V(G) \leq \alpha(G) \cdot \chi(G)$.*

Proof. Consider a minimal proper colouring φ of G. Then,

$$\#V = \sum_{i=1}^{\chi(G)} \#\varphi^{-1}(i) \leq \sum_{i=1}^{\chi(G)} \alpha(G) = \alpha(G) \cdot \chi(G).$$

■

The set $\varphi^{-1}(i)$ is usually called a *colour class*.

The connection between colourings and cliques is even simpler. Since any clique induces a subgraph of G which is complete, any proper colouring of G need to use k colours to colour vertices of a clique of size k. The following bound easily follows.

Theorem 10.3 *Let G be a graph. Then, $\chi(G) \geq \omega(G)$.*

10.2 Brooks Theorem and critical graphs

We have seen before that the upper bound $\chi(G) \leq \Delta(G) + 1$ is tight, i.e., there are graphs for which $\chi(G) = \Delta(G) + 1$.

The example used to demonstrate the tightness was K_n.

There is another class of graphs for which the upper bound is reached: the class of odd cycles C_{2n+1}, where C_n is the cycle graph on n vertices, i.e., n vertices connected in a cycle.

Obviously, for every $n \in \mathbb{N}$, we have $\chi(C_{2n+1}) = 3 = \Delta(C_{2n+1}) + 1$.

Example 10.11 Consider the cycle C_7 (see Figure 10.15). Each vertex has degree 2, so $\Delta(C_7) = 2$. On the other hand we need three colours to properly colour all vertices in the graph, e.g.,

$$\varphi(1) = \varphi(3) = \varphi(5) = 1, \quad \varphi(2) = \varphi(4) = \varphi(6) = 2 \quad \text{and} \quad \varphi(7) = 3.$$

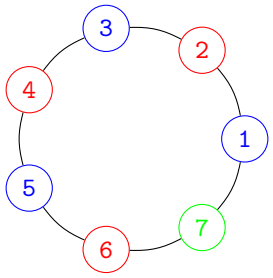


Figure 10.15: A colouring of the graph C_7.

The following theorem states that complete graphs and odd cycles are the only graphs which require $\Delta + 1$ colours.

Theorem 10.4 (Brooks (1941)) *Let $G = (V, E)$ be a connected graph which is neither a complete graph nor an odd cycle. Then $\chi(G) \leq \Delta(G)$.*

We will not inspect the proof of Brooks' Theorem as it is quite long and (at least a part of it) quite technical. Instead, we will only mention a class of graphs heavily used in the proof, since they are quite interesting and useful on their own.

A graph G is called *k-critical* if its chromatic number is $\chi(G) = k$, and for any proper subgraph H of G we have $\chi(H) \leq k - 1$.

The only 1-critical graph is K_1. Similarly, the only 2-critical graph is K_2.

Since a proper subgraph of a cycle is a union of paths, we have that C_{2n+1} is 3-critical for any $n \in \mathbb{N}$. Odd cycles are actually the only possible simple graphs which are 3-critical.

Proposition 10 *The only 3-critical simple graphs are the odd cycles C_{2n+1}.*

No equivalent characterisation of k-critical graphs is known for $k = 4$.

There are methods to construct a k–critical graph for a given $k \in \mathbb{N}$.

Example 10.12 Consider the two graphs in Figure 10.16 (the one on the right is known as the *Grötzsch graph*). One can check that both graphs are 4-critical.

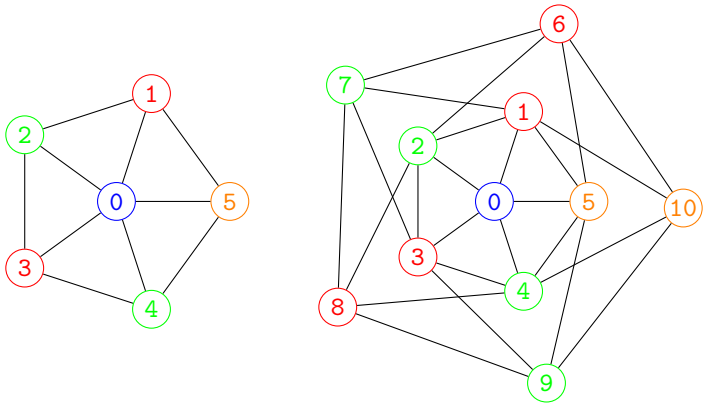


Figure 10.16: Two 4-critical graphs

Exercise 13 Show that the Hajós graph (defined in Example 10.2) is 4-critical.

Solution of Exercise 13 The possible subgraphs of the Hajós graph with one vertex removed are shown in Figure 10.17. One can check that each of them admits a 3-colouring. All other proper subgraphs of the Hajós graph are subgraphs of one of these seven graphs, so they also admit a 3-colouring.

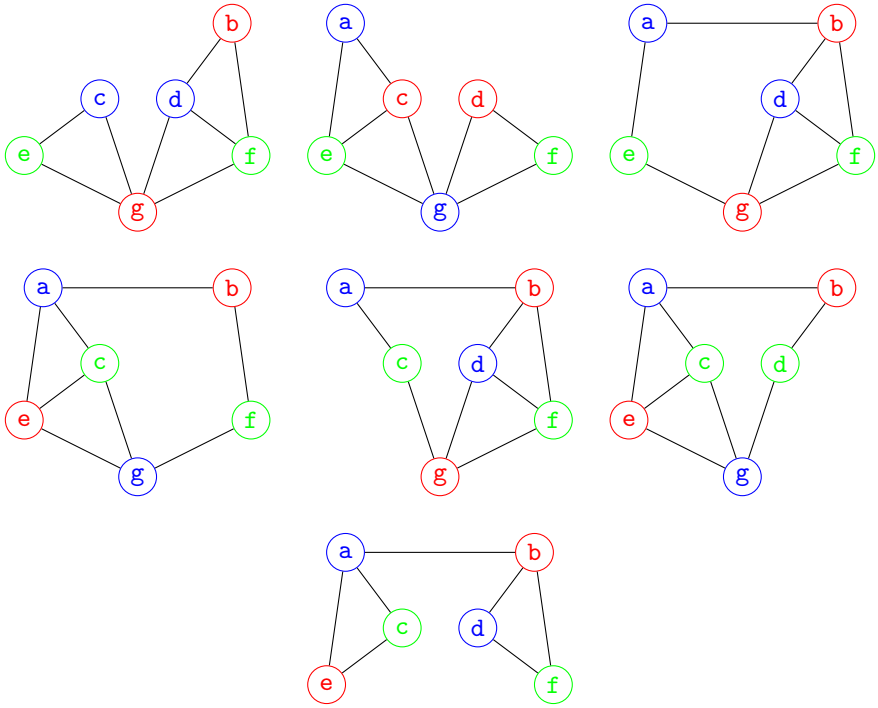


Figure 10.17: Maximal subgraphs of the Hajós graph.

Proposition 11 *Let G be a k-critical graph. Then G is connected and $\delta(G) \geq k-1$.*

Chapter 11

Planar graphs

When we draw a graph on a piece of paper, we naturally try to do this as transparently as possible. One obvious way to limit the mess created by all the lines is to avoid intersections, that is, we try to draw the graph in such a way that two edges do not meet in a point other than a common vertex.

A graph is said to be *planar* if it can be drawn (formally, it is identical or isomorphic to a graph drawn) in the plane without crossing of the edges. That means that two edges meet only at their common vertex.

More formally, a graph $G = (V, E)$ is called *planar* if there are two bijective mappings $\varphi : V \to \mathbb{R}^2$ and $\psi : E \to \mathcal{C}$, where $\mathcal{C}$ is the set of simple, continuous curves in $\mathbb{R}^2$, such that

(P1) for every $e = \{u, v\} \in E$, we have that $\varphi(u)$ and $\varphi(v)$ are endpoints of $\psi(e)$;

(P2) for every distinct $e, f \in E$, we have that $\psi(e) \cap \psi(f) = \varphi(e \cap f)$.

Example 11.1 The graph K_4 is planar. A representation where vertices and edges satisfy the two conditions (P1) and (P2) is shown in Figure 11.1

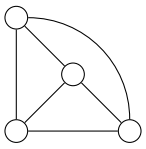


Figure 11.1: The graph K_4 is planar.

To study planar graphs we need some reminder of topology.

Recall that a *curve* in the real plane is just a continuous image of a segment, while a *closed curve* is a continuous image of a circle. A curve in $\mathbb{R}^2$ is called *simple* if it does not cross itself.

Example 11.2 The curve on the left and the closed curve in the centre of Figure 11.2 are simple, while the closed curve on the right of the same figure is not.

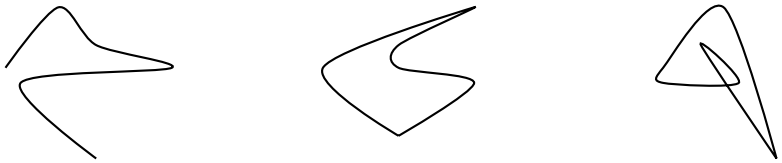


Figure 11.2: A curve (left) and two closed curves (centre and right) in $\mathbb{R}^2$.

Open sets in the real plane can be seen as a generalisation of open intervals in the real line. A simple definition of an *open set* in $\mathbb{R}^2$ is: a set $M \subseteq \mathbb{R}^2$ such that every point in it is the centre of an open circle (circle without its boundary) contained in M.

A set $M \subseteq \mathbb{R}^2$ is *arcwise-connected* if for every $x, y \in M$ there exists a continuous function $f : [0,1] \to M$ such that $f(0) = x$ and $f(1) = y$.

Example 11.3 Let us consider the set M defined as union of two disjoint open circles, i.e., we are considering only the points "inside" the circles and not on the borders. The set M is open but not arcwise-connected, since there is no arc in M connecting a point inside the first circle to a point inside the second one (see, e.g., Figure 11.3, where no curve inside M can have as endpoints x and z).

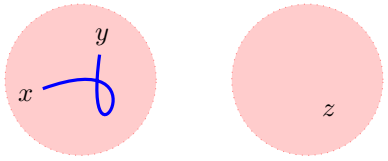
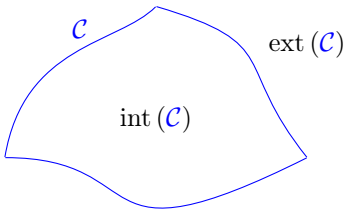


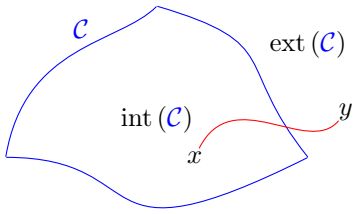
Figure 11.3: An open but not arcwise-connected set.

An essential result of topology that we are using when talking about planar graphs is the following.

Theorem 11.1 (Jordan Curve Theorem) *Any simple closed continuous curve $\mathcal{C}$ in $\mathbb{R}^2$ partitions the rest of $\mathbb{R}^2$ into two disjoint arcwise-connected open sets, denoted $int(C)$, or* interior *of $\mathcal{C}$, and $ext(\mathcal{C})$, or* exterior *of $\mathcal{C}$.*



Corollary 6 *Let $\mathcal{C} \in \mathbb{R}^2$ be a a simple, closed, continuous curve (called* Jordan curve*). Every continuous curve connecting a point $x \in int(\mathcal{C})$ with a point $y \in ext(\mathcal{C})$ intersect $\mathcal{C}$ somewhere.*

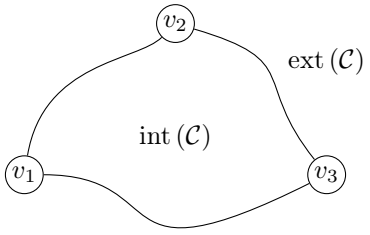


Using the previous corollary it is possible to prove that the regular graph K_5 is not planar.

Proposition 12 *The graph K_5 is not planar.*

Proof. Let $K_5 = (V, E)$ with $V = \{v_1, v_2, v_3, v_4, v_5\}$.

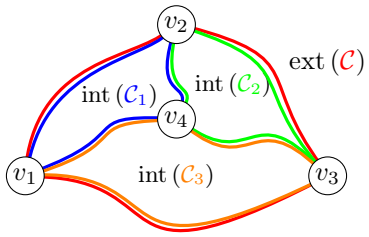
Assume, by contradiction, that K_5 is planar. Then, there is a planar drawing of K_5. First, let us consider the cycle (v_1, v_2, v_3). It corresponds to a Jordan curve $\mathcal{C}$ in our plane drawing of K_5.



Now, there are two possibilities for the placement of v_4: either $v_4 \in \text{int}(\mathcal{C})$, or $v_4 \in \text{ext}(\mathcal{C})$.

If we place v_4 into $\text{int}\,(\mathcal{C})$ and connect it by curves representing the edges $\{\{v_4, v_1\}, \{v_4, v_2\}, \{v_4, v_3\}\}$ we have the following closed curves:

- $\mathcal{C}$ given by the cycle (v_1, v_2, v_3),
- $\mathcal{C}_1$ given by the cycle (v_1, v_2, v_4),
- $\mathcal{C}_2$ given by the cycle (v_2, v_3, v_4),
- $\mathcal{C}_3$ given by the cycle (v_3, v_1, v_4).



We now have 4 possible placements of v_5: in $\text{int}\,(\mathcal{C}_1)$, in $\text{int}\,(\mathcal{C}_2)$, in $\text{int}\,(\mathcal{C}_3)$, or in $\text{ext}\,(\mathcal{C})$. None of them is possible in a planar drawing. For example, if $v_5 \in \text{int}\,(\mathcal{C}_1)$, since $v_3 \in \text{ext}\,(\mathcal{C}_1)$, the curve representing the edge $\{v_1, v_5\}$ would intersect (by the Jordan Curve Theorem 11.1) the curve $\mathcal{C}_1$, and thus the drawing would not be planar.

All other options – $v_5 \in \text{int}\,(\mathcal{C}_2)$, or in $\text{int}\,(\mathcal{C}_3)$, or in $\text{ext}\,(\mathcal{C})$; as well as v_4 in $\text{ext}\,(\mathcal{C})$ – lead to similar contradictions. ∎

When discussing about plane drawnings of a planar graph, we have to specify whether or not we are considering (only) planar drawings, since planar graphs (can) admit non-planar drawings as well.

Example 11.4 The graph on the left of Figure 11.4 is planar, but it has also non-planar drawings.

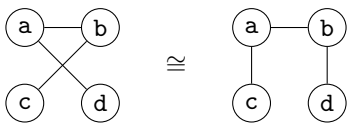


Figure 11.4: A planar graph.

A planar drawing of a (planar) graph is also called a *plane graph*.

The following classic results of Euler – here stated in its simplest form, for the plane $\mathbb{R}^2$ – marks one of the origins of connections between graph theory

and topology. The theorem relates the number of vertices, edges and faces, i.e., regions of $\mathbb{R}^2 \setminus G$, of a planar graph G. There is a general version of the formula which asserts a similar equality for graphs embedded in other surfaces (e.g., on a sphere).

Given a plane graph G, a *face* of G is an arcwise-connected region of $\mathbb{R}^2$ which borders are given by edges in G. Every planar graph has one unbounded face, called the *outer face*.

Theorem 11.2 (Euler's formula) *Let $G = (V, E)$ be a connected planar graph. Then*

$$\#V - \#E + \Psi(G) = 2,$$

where $\Psi(G)$ is the number of faces of a planar drawing of G.

As a consequence of the previous theorem we have that every planar drawing of a given planar graph has the same number of faces.

Example 11.5 A planar drawing of K_4 with four faces, denoted $\Omega_1, \Omega_2, \Omega_3$ and Ω_4 (this last being the outer face) is given in Figure 11.5. We have

$$\#V - \#E + \Psi(G) = 4 - 6 + 4 = 2.$$

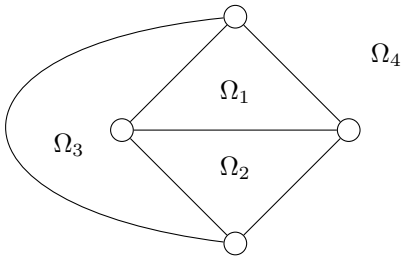


Figure 11.5: A planar graph with four faces.

We can use Euler's formula to derive two simple necessary (but not sufficient) conditions for the planarity of a graph G.

Proposition 13 *Let $G = (V, E)$ be a planar graph with $\#V = n \geq 3$.*

1) $\#E \leq 3n - 6$.

2) If G is bipartite, then $\#E \leq 2n - 4$.

Example 11.6 The complete graph K_4 has 4 vertices and $6 \leq 3 \cdot 4 - 6$ edges, according to point 1) in Proposition 13. The bipartite graph in Figure 11.6 has 6 vertices and $8 \leq 2 \cdot 6 - 4$ edges, according to point 2) of the same proposition.

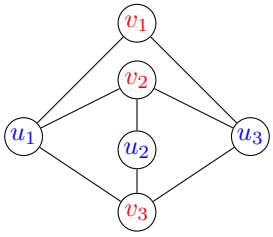


Figure 11.6: A bipartite graph.

Corollary 7 *Every simple planar graph has a vertex of degree at most five.*

Proof. Let $G = (V, E)$ be a simple planar graph, with $\#V = n$ and $\#E = m$. Then, using Proposition 13 and the results seen in previous lectures we have,

$$\delta(G) \cdot n \;\leq\; \sum_{v_i \in V} d_G(v) \;=\; 2m \;\leq\; 2 \cdot (3n - 6) \;=\; 6n - 12.$$

Thus $\delta(G) \leq 6 - \frac{12}{n} < 6$, which implies $\delta(G) \leq 5$. ∎

We can also use Proposition 13 to demonstrate non-planarity of two important graphs.

Theorem 11.3 *The graphs K_5 and $K_{3,3}$ are not planar.*

Proof. The graph K_5 has 10 edges and 5 vertices, and $10 > 3 \cdot 5 - 6 = 9$. So, by point 1) of Proposition 13 the graph cannot be planar.

The bipartite graph $K_{3,3}$ has 9 edges and 6 vertices, and $9 > 2 \cdot 6 - 4 = 8$. So, by point 2) of Proposition 13 the graph cannot be planar. ∎

Both K_5 and $K_{3,3}$ play a prominent role in the theory of planar graphs. They are, as shown in the next section, the epitomes of non-planarity.

11.1 Subdivisions

Let $G = (V, E)$ be a graph. A *subdivision* of G is, informally, any graph obtained from G by "subdividing" some of its edges by drawing new vertices on those edges. In other words, we replace some edges in E with new paths between their ends, so that none of these new paths has an inner vertex in V or on another new path.

When H is a subdivision of G, we also write that $H = T(G)$. The original vertices of G are called *branch vertices* of $T(G)$; its new vertices are called *subdividing vertices*.

If a graph G' contains a $T(G)$ as a subgraph, then G is called a *topological minor* of G'.

Example 11.7 Let G, $H = T(G)$ and G' be as in Figure 11.7. Since $H \subseteq G'$, we have that G is a topological minor of G'.

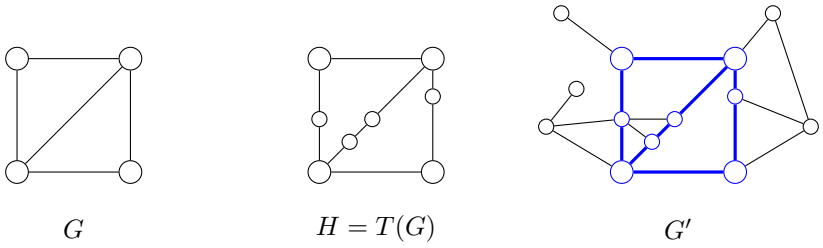


Figure 11.7: Topological minors.

Clearly, if G is a planar graph, then every subgraph H of G is also planar.

Similarly, if G is a non-planar graph, then any subdivision $T(G)$ is also non-planar.

Theorem 11.4 (Kuratowski (1930)) *Let G be a graph. Then G is planar if and only if G has neither K_5 nor $K_{3,3}$ as a topological minor.*

The "only if" part of the theorem follows easily from the previous observation and from the already established non-planarity of K_5 and $K_{3,3}$.

The "if" part of the proof of the theorem is quite long and technical.

It may be quite non-intuitive how to use Theorem 11.4 do demonstrate non-planarity of a particular graph.

Example 11.8 Let us consider the graph G on the left of Figure 11.8. The subgraph H of G shown in the centre of the same figure is a topological minor of $K_{3,3}$ (right of the same figure). Therefore H is not planar, and hence G is not planar neither.

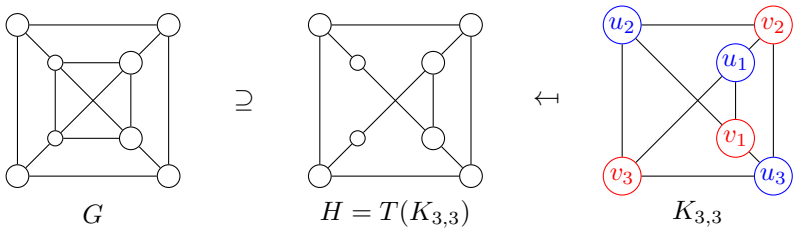


Figure 11.8: Topological minors.

11.2 Planarity algorithm

Graph planarity plays a crucial role in many real-world applications where edge crossings can complicate design and functionality. For example:

- **Printed circuit board** (**PCB**) design, where minimizing crossings reduces complexity and improves manufacturability.
- **Subway and transportation network planning**, where clear, non-overlapping routes enhance usability and efficiency.

Determining whether a graph is planar – and, if so, constructing a planar drawing – is a fundamental challenge with broad practical implications.

We present a human-readable explanation of the classical *path addition method* introduced by Hopcroft and Tarjan in 1974. This method was groundbreaking since it was the first linear-time algorithm (with respect to the number of vertices) for testing graph planarity.

Let H be a planar subgraph of G, and let $\hat{H}$ be a planar embedding (i.e., a planar drawing) of H.

We say that $\hat{H}$ is *G-admissible* if G is planar and there exists a planar embedding $\hat{G}$ of G such that $\hat{H}$ is a subgraph of $\hat{G}$.

Example 11.9 Let us consider the graph G on the left of Figure 11.9 and its subgraph $H = G \setminus \{\mathtt{a}, \mathtt{f}\}$. The planar embedding $\hat{H}_1$ of H, shown in the centre of the figure, is G-admissible, while the planar embedding $\hat{H}_2$ of H, shown on the right, is not.

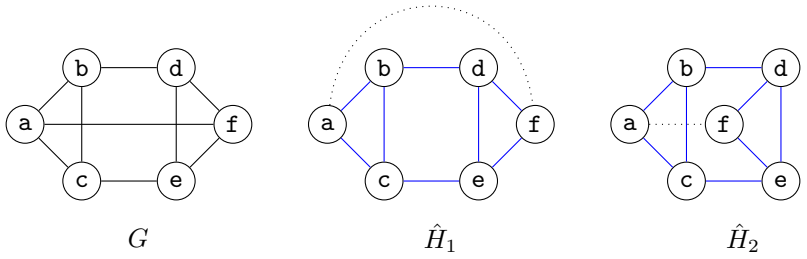


Figure 11.9: A planar graph G and two embeddings of a subgraph H.

Suppose $H = (V', E') \subset G = (V, E)$. We define an equivalence relation $\sim_B$ on the edges of G that are not in H, i.e., on $E \setminus E'$, as follows: two edges e_1 and e_2 are equivalent, denoted $e_1 \sim_B e_2$, if there exists a walk W in G such that

- the first and last edges of W are e_1 and e_2, respectively;
- W is internally disjoint from H, i.e., no internal vertex of W belongs to H.

The relation $\sim_B$ is an equivalence relation. A subgraph of $G \setminus E'$ induced by an equivalence class under $\sim_B$ is called a *bridge* of H in G.

Example 11.10 Let us consider the graph G in Figure 11.10 and its subgraph H given by the cycle $(1, 2, 3, 4, 5, 6, 7, 8, 9)$. Five possible bridges of H in G are shown in the picture.

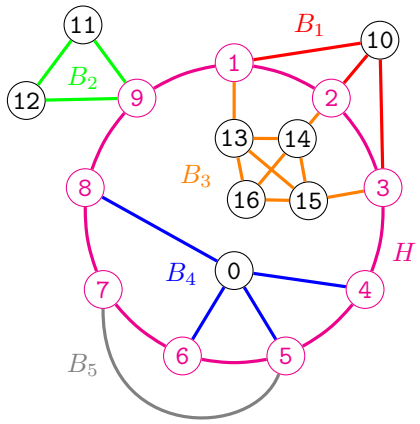


Figure 11.10: A graph G with a subgraph H and 5 bridges.

Given a planar drawing $\hat{H}$ of H, a bridge B of H is *drawable* in a face f of $\hat{H}$ if the vertices of attachment of B to H lie in the boundary of f.

We denote by $F(B, \hat{H})$ the set of faces of $\hat{H}$ in which B is drawable.

Example 11.11 Let G be the graph in Figure 11.11. Let H be the subgraph of G, whose planar drawing $\hat{H}$ is represented in the figure in blue. $\hat{H}$ has tree faces: Ω_1 (the outer face), Ω_2 and Ω_3. The bridge B, with edges represented in red, of H is drawable in Ω_1 and Ω_2, but not in Ω_3, i.e., $F(B, \hat{H}) = \{\Omega_1, \Omega_2\}$.

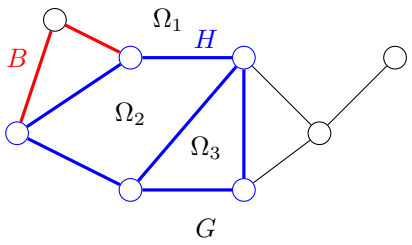


Figure 11.11: A graph G with a subgraph H and a bridge B.

Theorem 11.5 *If $\hat{H}$ is G-admissible, then for every bridge B of H in G we have $F(B, \hat{H}) \neq \emptyset$.*

Algorithm

The algorithm constructs an **increasing sequence** of planar subgraphs of $G = (V, E)$:

$$G_1 \subseteq G_2 \subseteq G_3 \subseteq \cdots,$$

with each $G_i = (V_i, E_i)$, along with their corresponding planar embeddings:

$$\hat{G}_1 \subseteq \hat{G}_2 \subseteq \hat{G}_3 \subseteq \cdots.$$

If G is planar, each $\hat{G}_i$ is G-admissible, and the sequence $(\hat{G}_i)_{i \geq 1}$ terminates in a planar embedding of G.

Steps.

1. Initialise G_1 as a cycle in G, find its planar embedding $\hat{G}_1$, and set $i = 1$.
2. If $E \setminus E_i = \emptyset$, then `STOP`: $\hat{G}_i$ is a planar embedding of G.

   Otherwise, determine all bridges of G_i in G. For each such bridge B find the set of faces $F(B, \hat{G}_i)$ in which B is drawable.
3. If there exists a bridge B such that $F(B, \hat{G}_i) = \emptyset$, then `STOP`: G is not planar.

   If there exists a bridge B such that $\#F(B, \hat{G}_i) = 1$, let $\{f\} = F(B, \hat{G}_i)$.

   Otherwise, choose any bridge B and any face $f \in F(B, \hat{G}_i)$.
4. Select a path $P_i \subseteq B$ connecting two vertices of attachment of B to G_i.

   Set $G_{i+1} = G_i \cup P_i$ and obtain $\hat{G}_{i+1}$ by drawing P_i in the face f of $\hat{G}_i$.

   Increment i by 1 and return to Step 2.

Exercise 14 Let G be the graph shown in Figure 11.12. Apply the previous algorithm to find a planar embedding of G.

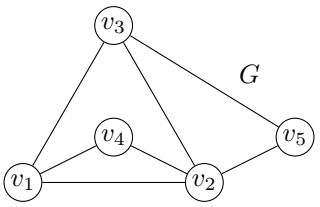
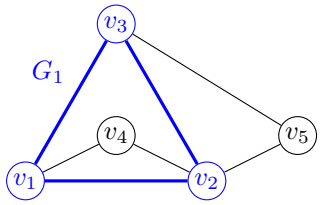


Figure 11.12: A planar graph.

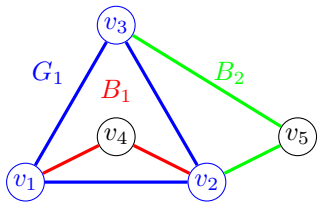
Solution of Exercise 14 Consider the subgraph G_1 of G given by the cycle (v_1, v_2, v_3).

- **Step 1:** G_1 is the cycle (v_1, v_2, v_3), and $\hat{G}_1$ is its planar embedding.



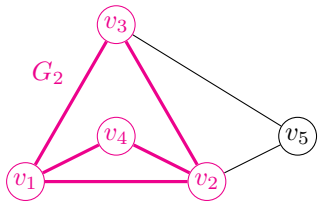
G

- **Step 2:** The edges (v_1, v_4), (v_4, v_2), (v_2, v_5), and (v_5, v_3) are not in G_1. The bridges of G_1 in G are:
  - B_1 formed by $\{v_1, v_4\}$ and $\{v_4, v_2\}$,
  - B_2 formed by $\{v_2, v_5\}$ and $\{v_5, v_3\}$.



G

- **Step 3:** Both B_1 and B_2 are drawable. Let us pick the bridge B_1 and the inner face of $\hat{G}_1$.
- **Step 4:** Select the path $P_1 = (v_1, v_4, v_2)$ in B_1. Add P_1 to G_1 to form G_2, and draw P_1 in the inner face of $\hat{G}_1$.



G

- **Iteration:** Repeat the process for G_2 and the remaining edges until all edges are embedded or non-planarity is detected (in this example the graph is planar).

11.3 Four Colours Theorem

If any result in graph theory has a claim to be known to the world outside, it is the following Four Colour Theorem.

Theorem 11.6 (Four Colour Theorem) *Every planar graph has chromatic number at most* 4*.*

The above theorem (proved in 1976 by Appel and Haken) proves the long standing Four Colour Conjecture that every planar graph is 4-colourable. This problem has an easy to understand reformulation "outside" of graph theory:

"*Any map can be coloured using at most four colours in such a way that adjacent regions (i.e., those sharing a common boundary segment, not just a point) receive different colours*"

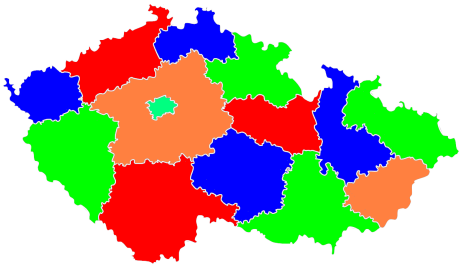


Figure 11.13: Map of Czechia's regions using only four colours.

Indeed, given a planar graph G one can construct a dual graph G' putting in bijection the set of faces of G with the set of vertices of G', with two faces adjacent in G if and only if the two correspective vertices in G' are connected by an edge.

Example 11.12 The graph G on the left of Figure 11.14 is the dual of G' represented on the right of the same figure.

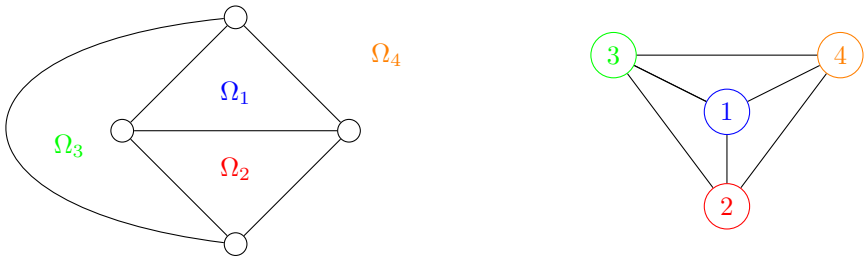


Figure 11.14: Two dual graphs.

History. The Four Colour Conjecture was first formulated by Francis Guthrie in 1852 while attempting to colour a map of the counties of England. Francis shared the problem with his brother Frederick Guthrie, then an undergraduate at Cambridge, who passed it on to Augustus De Morgan. The conjecture gained broader attention when Arthur Cayley presented it to the London Mathematical Society in 1878.

In 1879, Alfred Kempe published what was initially believed to be a proof, but it was later revealed to be incorrect. However, in 1890, Percy Heawood adapted Kempe's work to prove the Five Colour Theorem, which states that every planar graph G satisfies $\chi(G) \leq 5$.

The first widely accepted proof of the Four Colour Theorem (FCT) was published by Kenneth Appel and Wolfgang Haken in 1977. Their proof followed a two-step approach:

1. **Unavoidable Configurations**: They demonstrated that every planar triangulation must contain at least one of 1,482 specific "unavoidable configurations."

   Reducibility: Using a computer, they showed that each of these configurations is "reducible" – meaning any planar triangulation containing such a configuration can be 4-coloured by combining 4-colourings of smaller triangulations.

Together, these steps formed an inductive proof that all planar graphs can be 4-coloured.

The proof sparked significant controversy, not only due to its reliance on computer assistance but also because of its complexity. In response to criticism, Appel and Haken published a 741-pages algorithmic version in 1989, addressing errors and clarifying the proof. A more concise proof, though still computer-assisted, was later provided by Neil Robertson and colleagues in 1997.

To this day, no "simple" or purely theoretical proof of the Four Colour Theorem has been discovered.