Chapter 3

Elements of Graph Theory

3.1 A basic vocabulary

Graph theory is a beautiful and amazing mathematical theory whose colorful and pictorial language allows us to elegantly formulate and efficiently solve many problems of applied mathematics. Notably in scheduling graph theory is an indispensable tool. It is the intention of this chapter to give you an introduction into this important field. Being confined to the most basic concepts our course will proceed in a pedestrian like manner. So practically no proofs are presented in this chapter, as this is not a book on graph theory. We refer the interested readers kindly to any of the many excellent textbooks available. At the end of this chapter there is a section with bibliographic notes supporting readers to get some orientation in literature.

3.1.1 Graphs as binary relations

Graph theory is a mathematical theory of *relations*. In mathematical terms, a *binary relation* E on a finite set[1] V is a subset of the set of all *ordered pairs* we can form by the elements of V, i.e., $E \subset V \times V$, the cartesian product of V with itself. The pairs comprising E will be denoted by $(x, y) \in E$, where $x, y \in V$ and we call them *edges*, the elements of V are called *nodes* or *vertices*. We agree that edges (x, y) and (y, x) are considered different objects and if there is no danger of confusion we abbreviate an edge (x, y) by simply writing xy. If there exists an edge xy then we also say that x is *adjacent* to y and y is adjacent from x. Adjacency is also considered a property of edges. Two edges $a, b \in E$ are called adjacent, if the have a common endpoint.

Thus a graph G is an *ordered pair* formed by the set of nodes V and the set of edges E, symbolically, $G = (V, E)$. If the number of elements in V equals n, i.e. $|V| = n$, then we say that G is of *order* n. The number of elements in E, $m = |E|$ is called the *size* of G.

Graphs as we have defined them are often called *directed graphs* or *digraphs* in literature. A word of caution here: although graph theory is nowadays a mature field of mathematics, its language is highly nonstandard, unfortunately. Richard Stanley once pointed out: "*The number of systems of terminology presently used in graph theory is equal, to a close approximation, to the number*

[1] Also infinite sets may be considered, but we do not need this generalization.

of graph theorists." (cited from Knuth (2011)). I have tried to avoid this somewhat babylonian confusion by defining and using terms as this is done in most classical textbooks on graph theory.

The concept of a binary relation is a very general one, thus we are often led to restrict the relation E making up a graph G to relations which may or may not have some of the following properties:

- *Symmetric property*: $xy \in E \implies yx \in E$ for $x, y \in V$.
- *Antisymmetric property*: $xy \in E \implies yx \notin E$ for $x, y \in V$.
- *Reflexive property*: $xx \in E$ for $x \in V$.
- *Irreflexive property*: $xx \notin E$ for $x \in V$.
- *Transitivity*: $xy \in E, yz \in E \implies xz \in E$, for $x, y, z \in V$.

If E happens to have the *symmetric property*, thus to each edge $xy \in E$ there is also $yx \in E$, then G is called an *undirected graph*. In this case it is convenient to consider its edges as *unordered pairs* of nodes which we denote e.g. by $\{x, y\}$, etc.

Being directed or undirected, we shall always assume that E has the *irreflexive property*, so G does not have loops. Loops are edges directing a node to itself.

It is often very helpful to draw a diagram of a graph. Indeed, the term *graph* has been used 1878 for the first time in this sense by J. J. Sylvester.

Drawing a graph is done in the following way:

- Nodes are represented by circles or dots in the plane, with or without labels.
- If there exists an edge $xy \in E$, then we draw an arrow from x to y.
- If G is an undirected graph, then it is customary instead of drawing two arrows, viz. xy and yx, to draw a single line without arrow tips connecting x and y.

3.1.2 Some examples

Flight connections

Consider 8 towns, any two of them, say u and v being related, if there exists a *direct flight connection* either in direction $u \to v$ or in direction $v \to u$. So we arrive at some sort of *traffic network* representable by a graph $G = (V, E)$ with nodes $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$, The direct connections (no changes required when traveling from u to v) are given in the following table. In this table we place a 1 in row u and column v if there is a direct flight from $u \to v$ and otherwise we place a 0 there.

from/to	1	2	3	4	5	6	7	8
1	0	1	0	0	0	1	0	0
2	0	0	0	0	0	1	0	0
3	0	1	0	0	0	1	0	0
4	0	0	1	0	0	0	0	0
5	0	0	0	1	0	0	0	0
6	0	0	0	0	1	0	1	0
7	0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	1	0

The row and column indices pointing to ones in the above table give us the edge set E:

$$E = \left\{ \begin{array}{ccccc} (1,2) & (1,6) & (2,6) & (3,2) & (3,6) \\ (4,3) & (5,4) & (6,7) & (8,1) & (8,7) \end{array} \right\}.$$

The sets V and E constitute a graph $G = (V, E)$, the corresponding diagram is displayed in Figure 3.1.

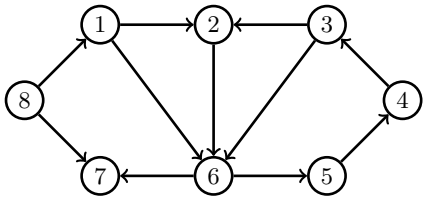


Figure 3.1: Flight connections

This graph is of order $n = 8$ and size $m = 10$.

Friendships

Suppose there are five persons P_1, P_2, P_3, P_4 and P_5, each represented by its numerical index. These identifying numbers form a set $V = \{1, 2, 3, 4, 5\}$. Suppose further:

- 1 has two friends among V, namely 2 and 3.
- 2 is a friend of $1, 3$ and 5.
- 3 is a friend of $1, 2$ and 5.
- 4 has no friend at all among V.

Let us interpret the pair xy as

$$x \text{ is a friend of } y.$$

Of course, friendship is a *symmetric* binary relation. If x is a friend of y then y is also a friend of x. So, in our example we have five unordered friendship pairs:

$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 5\}, \{3, 5\}\}.$$

Now $\{x, y\}$ denotes the undirected edge connecting nodes x and y and this is just a useful abbreviation for effectively two directed edges xy and yx.

These edges together with V define an *undirected graph* $G = (V, E)$. Its order is $n = |V| = 5$ and its size equals $m = 5 = |E|$. The corresponding diagram is shown in Figure 3.2.

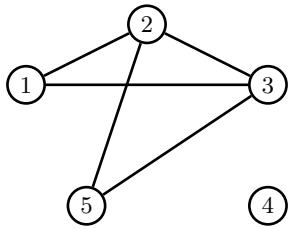
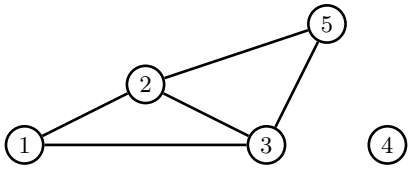


Figure 3.2

Node 4 is not connected to any other node, because person 4 has no friends. Such a node is called *isolated*.

Note that Figure 3.2 is just one of many different ways to draw this graph G. A diagram equivalent to Figure 3.2 is, for instance:



One final remark on this example: it may be the case that in a randomly selected group of people no one has a friend in his group. In this case the edge set is empty, $E = \varnothing$. As a result the graph $G = (V, \varnothing)$ consists of n isolated nodes, its size is zero. This graph G is also called *null-graph* and denoted by K_0.

Precedence relations

In many scheduling problems jobs to be scheduled are *dependent* in the sense that a particular job can be processed only if certain other jobs have been already completed[2]. Consider for instance the job of a male getting dressed in the morning. This job may consist of the tasks of putting on:

[2]Recall the bicycle example in Chapter 1!

task number	task
1	underwear
2	socks
3	shirt
4	trousers
5	tie
6	jacket
7	cap
8	shoes

Clearly, it seems to by very strange and unpractical to put on trousers first and then shoes, and many other weird combinations may come into your mind. A reasonable structure of this process can be represented as a *precedence graph* like that in Figure 3.3. Here nodes represent tasks and edges exhibit dependencies. Of course, these dependencies are not symmetric.

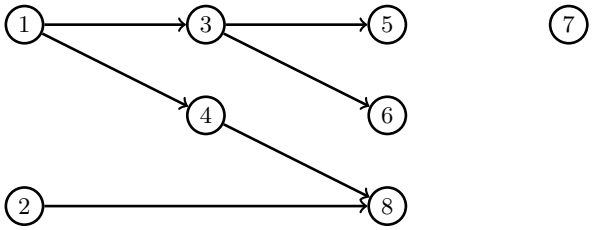


Figure 3.3

From this figure we may read off that putting on shoes requires first to put on trousers and socks. Putting on trousers in turn requires first to put on your underwear. Observe that there is no edge from/to node 7, so node 7 is isolated. This simply means that putting on your cap can be done at any time in the process of dressing[3].

Web pages and hyperlinks

An example of a very huge graph is the world wide web with nodes being web pages and an edge between two pages u and v, if page u has a link pointing to page v. Of course, this graph is much too big to be drawn, as there are billions of pages online, indeed, about 5 billion as of April 2015[4].

3.1.3 A useful multivalued mapping

It turns out to be very useful to have a device which provides us with information about the *neighbors* of a particular node x. Claude Berge (1966) seems to have

[3] Although this may give rise to rather funny configurations during the process of dressing. By the way, *topologically* it is possible to put on first trousers and then underwear. But nobody does this under normal circumstances since it very impractical.

[4] Source: `http://www.worldwidewebsize.com/`

been the first to use a multivalued mapping $\Gamma(x)$ which assigns to each node x all nodes y for which an edge xy exists. Thus $\Gamma(x)$ yields the set of all nodes y which are *adjacent from* x. For instance in the example on flight connections we find from Figure 3.1:

$$\Gamma(1) = \{2, 6\}, \quad \Gamma(2) = \{6\}, \quad \Gamma(7) = \varnothing, \qquad \text{etc.}$$

Γ may be extend easily to arguments which are subsets of nodes. Let $U = \{u_1, u_2, \ldots, u_k \in V\}$, then we define

$$\Gamma(U) = \Gamma(u_1) \cup \Gamma(u_2) \cup \cdots \cup \Gamma(u_k). \tag{3.1}$$

In case of the graph given in Figure 3.1 we have, for instance:

$$\begin{aligned}\Gamma(\{1, 2, 3\}) &= \Gamma(1) \cup \Gamma(2) \cup \Gamma(3) \\ &= \{2, 6\} \cup \{6\} \cup \{2, 6\} = \{2, 6\}.\end{aligned}$$

The *inverse* mapping $\Gamma^{-1}(x)$ gives us the set of all nodes which are *adjacent to* x, i.e., all nodes y for which there exists an edge $yx \in E$. For instance, from Figure 3.1 we find:

$$\Gamma^{-1}(7) = \{6, 8\}, \quad \Gamma^{-1}(8) = \varnothing.$$

Γ^{-1} may be extended to set arguments in the same way as we did with Γ. Also Γ^{-1} may be iterated to Γ^{-2}, etc. with an obvious meaning.

In case $G[V, E]$ is an *undirected graph* for each node x we have necessarily $\Gamma(x) = \Gamma^{-1}(x)$. For the undirected graph drawn in Figure 3.2 we verify

$$\Gamma(a) = \{b, c\}, \quad \Gamma(b) = \{a, c, e\}, \quad \Gamma(d) = \varnothing.$$

The *outdegree* d_x^+ and the *indegree* d_x^- of a node x are defined as

$$d_x^+ = |\Gamma(x)|, \quad d_x^- = |\Gamma^{-1}(x)|. \tag{3.2}$$

This definition carries over to undirected graphs. The degree d_x of a node x equals

$$d_x = |\Gamma(x)|. \tag{3.3}$$

For the graph drawn in Figure 3.1:

$$\begin{array}{llll} d_1^+ = 2 & d_1^- = 1 & d_2^+ = 1 & d_2^- = 2 \\ d_3^+ = 2 & d_3^- = 1 & d_4^+ = 1 & d_4^- = 1 \\ d_5^+ = 1 & d_5^- = 1 & d_6^+ = 2 & d_6^- = 3 \\ d_7^+ = 0 & d_7^- = 2 & d_8^+ = 2 & d_8^- = 0 \end{array},$$

and for the undirected graph in Figure 3.2 we have:

$$d_a = 2, \quad d_b = 3, \quad d_c = 3, \quad d_d = 0, \quad d_e = 2. \tag{A}$$

3.1.4 Exercises

1. The sequence of numbers $[2, 3, 3, 0, 2]$ in (A) is also known as *degree sequence* of an undirected graph. Is it possible for an undirected graph to have degree sequence $[2, 3, 3, 1, 2]$? If not, explain why. Otherwise, give an example.

2. Prove that for a graph $G = (V, E)$ always:

$$\sum_{x \in V} d_x^+ = \sum_{x \in V} d_x^- = |E|,$$

and for any *undirected* graph G:

$$\sum_{x \in V} d_x = 2|E|.$$

3. *More on friendships.* Suppose there is a group of $2n + 1$ people, $n = 1, 2, 3, \ldots$. Suppose further that every pair of persons in this group has one friend in common. Then there must be one person in this group who is friend of all other people. You may not try to prove this famous result which is known as *Friendship Theorem.* Just find a representation of its statement as an undirected graph. Draw this graph for $n = 1, 2, 3, 4$ and verify this statement *by experiment.*

4. *The eight-circles problem.* Consider the graph displayed in Figure 3.4.

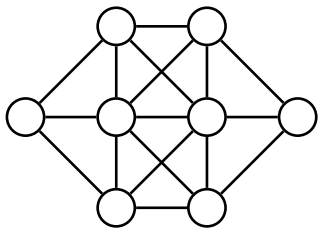


Figure 3.4

Find a way to label the nodes with the letters *A, B, C, D, E, F, G* and *H* in such a way that no letter is adjacent to a letter that is next to it in the alphabet.

3.2 Some important graphs

In this section we will present some directed and undirected graphs which play a special role in graph theory. This list is by no means complete and will be extended occasionally later when we proceed with our investigations.

3.2.1 Complete graphs

The complete graph of order n is an *undirected* graph denoted by K_n in which every node is adjacent to every other node. The first few complete graphs are presented in Figure 3.5.

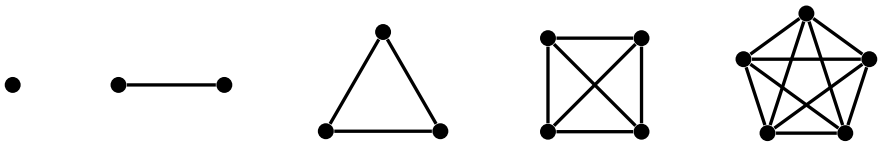


Figure 3.5: Complete graphs K_1, K_2, K_3, K_4, K_5

Assigning an *orientation* to each edge of K_n yields a directed graph called a *tournament*.

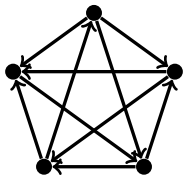


Figure 3.6: A tournament of order $n = 5$

3.2.2 Bipartite graphs

An undirected graph $G = (V, E)$ is called *bipartite*, if its node set can be decomposed into two disjoint sets S and T:

$$V = S \cup T, \quad S \cap T = \varnothing,$$

in such a way that each edge has one endpoint in S and the other in T, see Figure 3.7 for some examples.

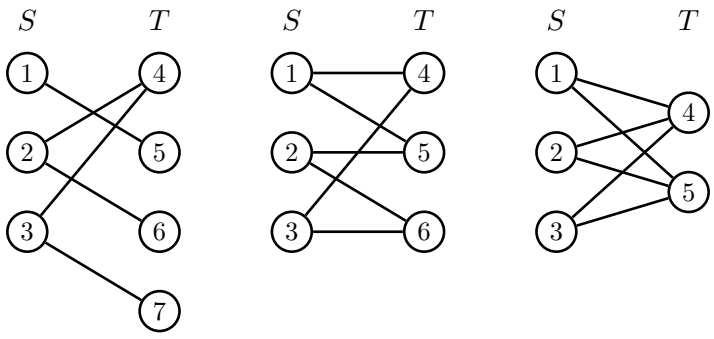


Figure 3.7: Three examples of bipartite graphs

It is customary to emphasize the decomposition of the node set in the special triplet notation $G = (S, T; E)$. If $|S| = m, |T| = n$ and each node in S is connected to a node in T, then G is called *complete bipartite graph* $K_{m,n}$. The rightmost example in Figure 3.7 shows a $K_{3,2}$.

Bipartite graphs arise quite naturally when dealing with assignment problems. For instance S may denote a set of jobs and T a set of machines. An edge $\{x, y\}, x \in S, y \in T$ means that job x should run on machine y.

It is not at all *a priori* clear whether a given undirected graph is bipartite. To decide that an *algorithm* is required, we shall discuss this issue in a few moments. The reader is invited to check that the graph given in Figure 3.8 is indeed bipartite.

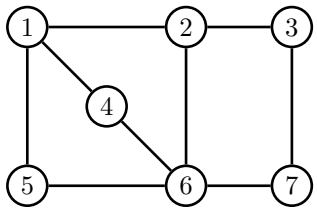


Figure 3.8

3.2.3 Interval graphs

Consider the following course scheduling problem: During a particular semester a university offers n courses to students. Let $T_i = (s_i, t_i)$ be the time interval during which course number i should take place. The management of the university wants to assign these courses to classrooms so that no two courses meet in the same room at the same time. How many classrooms are needed?

This planning problem has a natural representation as an undirected graph $G = (V, E)$. The node set $V = \{1, 2, \ldots .n\}$ is the set of offered courses. Two nodes u and v are connected by an edge, if their corresponding time slots intersect, i.e.,

$$\{u, v\} \in E \Leftrightarrow T_u \cap T_v \neq \varnothing.$$

The graph G is known as *interval graph*, a very important class of graphs with some very special properties.

▼ Example 3.1

To make affairs more concrete, suppose there are in total 9 courses and due to availability and preferences of teaching personnel the following time slots for courses are requested:

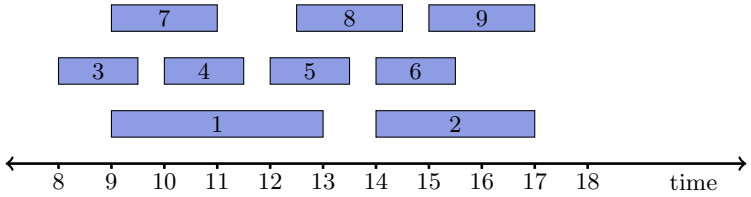


Figure 3.9: Interval graph for classroom scheduling

The reader may remember that we have already met a very similar situation in Example 2.7 on page 30.

The corresponding interval graph is:

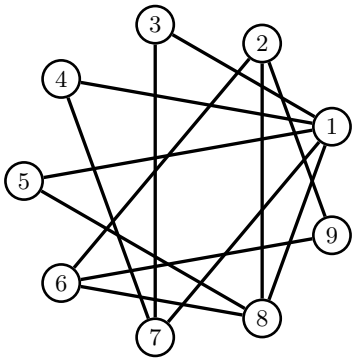


Figure 3.10: An interval graph

▲

3.2.4 Exercises

1. What is the number of edges of K_n, i.e. its *size*?

2. What is the size of $K_{m,n}$?

3. An undirected graph is called *k-regular*, if each node has degree k. A famous example is shown in Figure 3.11. Can you find *all* 3– and 4-

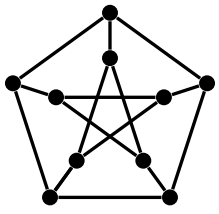
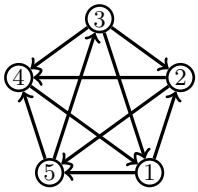


Figure 3.11: A 3-regular graph – the *Petersen graph*

regular graphs? If a graph of order n is k-regular, what is their size?

4. *Preferences.* The CEO of a company has ordered four proposals A, B, C and D for a marketing campaign. After having studied these carefully he ranks the proposals according to his personal preferences. The best proposal seems to be B, it gets rank one, then follow D, A and C in this order. Find a graph representation of this ranking. What is special about the resulting graph?

5. Why does the following graph *not* represent a ranking of five items that makes sense?



6. *An identification problem.* Define a *node coloring* of an undirected graph $G = (V, E)$ by the following process: from an unlimited set of available colors assign a color to each node of G in such a way that no two adjacent nodes get the same color. Let G be a graph. Show that it is bipartite if and only if it can be colored always using two colors only.

7. Show that the graph in Figure 3.8 is a bipartite graph $G = (S, T; E)$. Identify the node sets S and T.

8. Find the maximum size of a bipartite graph of order 10. More generally: determine the maximum size of a bipartite graph of order $n \geq 2$.

9. Develop an idea how the interval graph representation in Figure 3.10 may help to solve our scheduling problem in Example 3.1: so, what is the minimum number of classrooms required so that courses are not in conflict?

3.3 Basic operations on graphs

3.3.1 Deleting nodes and edges

Given a graph $G = (V, E)$ deletion of a node u results in a graph $H = (V', E')$ with

$$V' = V - \{u\}, \quad E' = E - \{uv, vu \in E\}$$

Symbolically:

$$H = G - \{u\}.$$

In other words, besides removing u from V we also remove all edges which are adjacent to or adjacent from u.

Similarly, deleting an edge $uv \in E$ results in a graph $H = G - \{uv\}$ with nodes $V' = V$ and edges $E' = E - \{uv\}$. Observe, that deleting an edge does not remove any node.

See Figure 3.12 for an illustration.

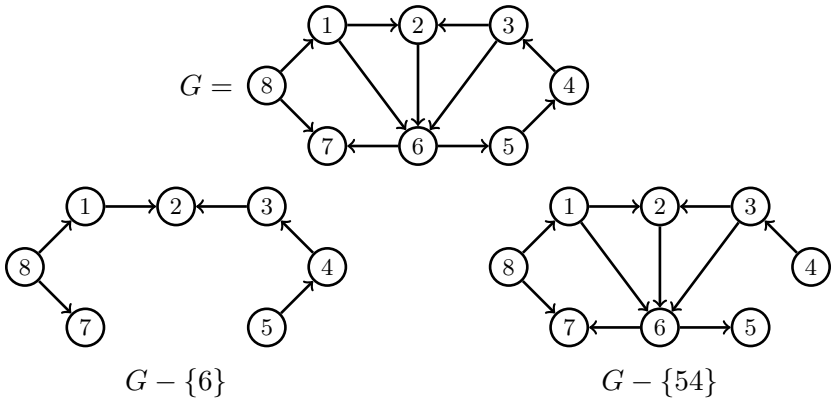


Figure 3.12

Adding a node u to $G = (V, E)$ gives rise to a new graph $H = (V', E')$. This does not create any new edges, it only augments the node set to $V' = V \cup \{u\}$. On the other hand, if we want to add an edge uv then we write symbolically $H = G \cup \{uv\}$. H has node set V and edge set $E' = E \cup \{uv\}$, so no new nodes are created.

3.3.2 Union, difference and complement

Both operations, addition and deletion of nodes and edges can be extended to sets of nodes and edges. More generally, if $G = (V, E)$ and $H = (W, F)$ with $V \cap W = \varnothing$, then the *union* of G and H is the graph $J = (X, Z)$ with

$$X = V \cup W, \quad Z = E \cup F.$$

The *difference* $G - H$ is the graph $J = (X, Z)$ with nodes and edges $X = V - W$. The edges Z are obtained from E by removing all edges adjacent to or from a node in W.

Another important concept is that of the *complement* of an undirected graph $G = (V, E)$. It is a graph $\overline{G} = (V, \overline{E})$ obtained from G by making each edge a non-edge in $\overline{G}$ and each non-edge becomes an edge. Formally, the set of edges in the complement is given by:

$$\overline{E} = \{xy \in V \times V : x \neq y \text{ and } xy \notin E\}.$$

An example is presented in Figure 3.13.

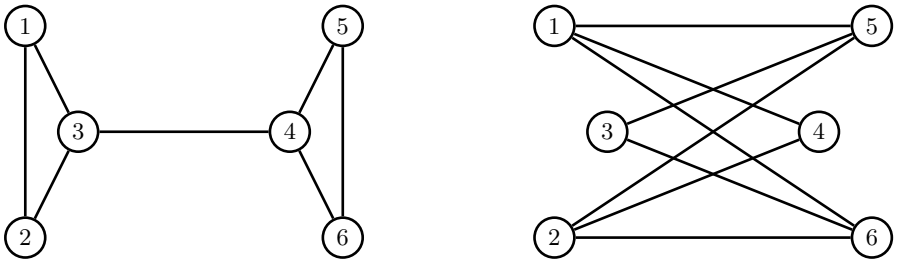


Figure 3.13: A graph G and its complement $\overline{G}$

3.4 Subgraphs, cliques and co

3.4.1 Subgraphs

Let $G = (V, E)$ be any graph. A *partial subgraph* of G is a graph $H = (U, F)$ with $U \subset V$ and $F \subset E$. An example is shown in Figure 3.14. The reader will

Figure 3.14: A graph G and one of its subgraphs H

notice that H does not have the edge $(6, 3)$, this is perfectly OK and conforms with our definition of a partial subgraph. But it is not always what we want. In several applications of graph theory discussed in this book subgraphs are constructed by specifying a subset of nodes U. The resulting subgraph H, however, should have all edges of G provided their endpoints lie in U. Its edge set is

$$F = \{xy \in E : x \in U \text{ and } y \in U\}$$

In this important case H is called the *subgraph induced* by U. It is denoted by $H = G|U$. As its construction is based on first selecting a subset of nodes, H is indeed a *node induced subgraph*, see Figure 3.15 for an illustration.

Figure 3.15: A graph G and its induced subgraph $H = G|\{1, 2, 3, 6\}$

3.4.2 Cliques

Let $G = (V, E)$ be an undirected graph and $H = G|U$ an induced subgraph of G. H is called a *clique*, if H is complete. i.e. each node in H is connected to each other node in H.

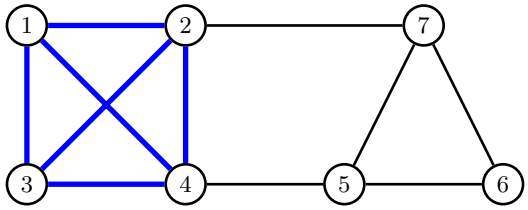


Figure 3.16: A graph G with a 4-clique

The order of the largest clique of an undirected graph G is called its *clique number* and denoted by $\omega(G)$. The graph G in Figure 3.16 has $\omega(G) = 4$. Finding the clique number of a graph is known to be a very hard problem, indeed, this problem is NP-hard.

3.4.3 Stable sets

The opposite of a clique is a *stable set* or *independent set*. It is a subset $W \in V$ of an undirected graph $G = (V, E)$, such that no two nodes in W are adjacent. In other words, there are no edges between the nodes in W. The number of nodes in a stable set of maximum cardinality is called stability number $\alpha(G)$.

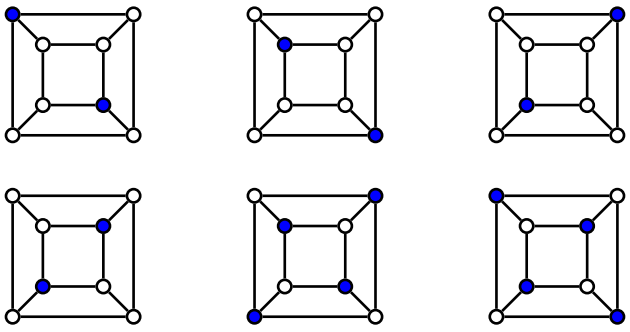


Figure 3.17: Stable sets of a graph G with $\alpha(G) = 4$

For a bipartite graph $G = (S, T; E)$ S and T are stable sets, of course.

3.4.4 Matchings

Let $G = (V, E)$ be an *undirected* graph. A *matching* M is a subset of E such that no two edges in M have an endpoint in common. M is called a *perfect*

matching, if each node in V is adjacent to an edge in M. See Figure 3.18 for an example.

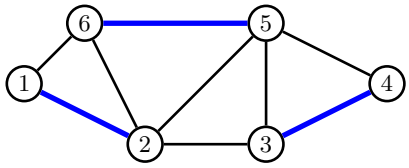


Figure 3.18: A graph an one of its matchings

Of particular importance due its applications in scheduling and other areas of optimization are *bipartite matchings*. If $G = (S, T; E)$ is a bipartite graph and $M \in E$ is a matching in G, then any edge $x \in E$ must connect a node in S with a node in T.

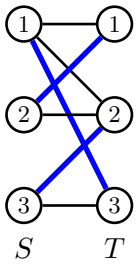


Figure 3.19: A perfect matching in a bipartite graph

Note that we have numbered nodes in S and T in the same way, as it is commonly done when dealing with dealing with bipartite matchings.

3.4.5 Modules

Let $G = (V, E)$ be an undirected graph. A *module* is a set M of nodes of G such that the elements of M have the same neighbors in $G - H$, where $H = G|M$ is the subgraph induced by M. Formally, for all nodes $x, y \in M$

$$\Gamma(x) - M = \Gamma(y) - M.$$

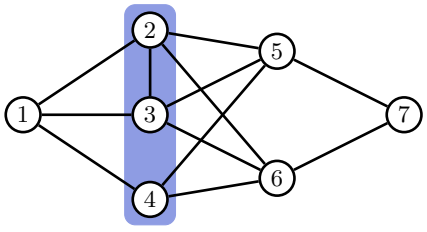


Figure 3.20: A module in a graph

Thus nodes in a module M are all related to nodes outside M in the same way: for each $y \notin M$ either there is an edge xy for each $x \in M$, or there is no such edge.

3.4.6 Exercises

1. The graph shown in Figure 3.16 has several 3-cliques. Find all of them. What about 2-cliques and 1-cliques? How many are there?

2. Find the clique number ω of the interval graph shown in Figure 3.10. How is it related to the minimum number of classrooms required?

3. What is the subgraph induced by $\{1, 2, 4, 7\}$ of G given in Figure 3.16?

4. Determine a stable set of maximum size for the Petersen-graph given in Figure 3.11. What is its size?

5. Does the graph in Figure 3.18 have a perfect matching? If so, find one.

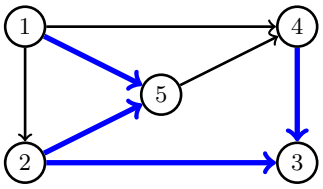
6. Verify that the set $M = \{2, 3, 4\}$ is a module of the graph in Figure 3.20. Are there other modules in this graph? If so, find them all.

3.5 Chains, paths, cycles, connectivity

Let $G = (V, E)$ be a graph. A *chain* is a sequence of nodes

$$[v_1, v_2, \ldots, v_r]$$

with $(v_{i-1}, v_i) \in E$ or $(v_i, v_{i-1}) \in E$ for $i = 1, 2, \ldots, r$. A chain is called *elementary*, if every node in the chain occurs only once.



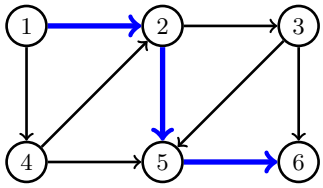
$C = [1, 5, 2, 3, 4]$

Figure 3.21: A graph G and an elementary chain in G

A *path* is a sequence of nodes

$$[v_1, v_2, \ldots, v_r]$$

with $(v_{i-1}, v_i) \in E$ for all $i = 1, 2, \ldots, r$. A path is *elementary* if each node on the path occurs only once.



$P = [1, 2, 5, 6]$

Figure 3.22: A graph G and an elementary path in G

The *cardinality* of a chain C and a path P will be denoted by $|C|$ and $|P|$, it is defined as the number of edges on the chain/path. Thus cardinality is some sort of *length*. In Chapter 4 we will give a rather natural definition of the length of a path or a chain in terms of *weights* assigned to nodes or edges.

The interested reader may ask now: *What is the difference between chains and paths*? To see the difference it is best to give the graph a physical interpretation, e. g., the graph may represent a water supply system, nodes are households, edges are pipes and water can only flow in direction of edges. A chain from a node v_1 to v_r is a physically existing connection, but it may not be possible to send water from v_1 to v_r because edges are not coherent with regard to orientation. On the other hand, a path from v_1 to v_r is a connection which is orientation-consistent. So it is technically possible to send water along the path from v_1 to v_r.

Of course, in case of an undirected graph there is no difference between paths and chains.

Still note, both concepts represent some sort of *connection* between two nodes.

A graph $G = (V, E)$ is *connected*, if there exists a *chain* between any pair of nodes. G is *strongly connected*, if there exists a *path* between any pair of nodes. A graph with only one node is always considered strongly connected.

A subgraph H of G is a *connected component* of G, if H is connected. H is called a *strong component*, if it is strongly connected.

If G is undirected then G is connected if there exists a path between any pair of nodes. Otherwise G is not connected or *disconnected* and has two or more components.

▼ **Example 3.2** Consider the graph G in Figure 3.23.

- It is connected, but not strongly connected.
- It has two strong components with node sets $U_1 = \{1, 2, 3, 4\}$ and $U_2 = \{5, 6, 7\}$.

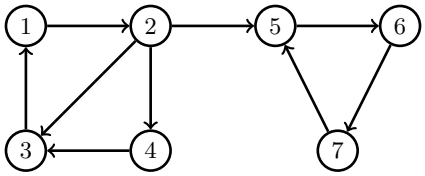


Figure 3.23: A graph G with two strong components

The graph G in Figure 3.24 is disconnected, it has two weak components and five strong components.

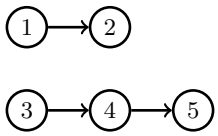


Figure 3.24

The undirected graph in Figure 3.25 is connected, indeed, it is an example of a *tree*.

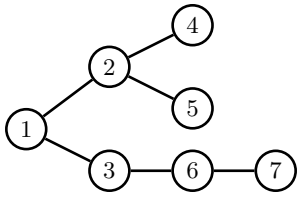


Figure 3.25

If we remove a *single edge* in this tree, it becomes disconnected. Such a critical edge is called a *bridge*. ▲

A *cycle* is a sequence of nodes

$$[v_0, v_1, v_2, \ldots, v_r, v_0]$$

such that $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, r$ and $(v_r, v_0) \in E$. The cycle C is called *elementary*, if each node except for v_0 on the cycle occurs only once. A cycle is *odd*, if it is made up by an odd number of edges, otherwise it is *even*. A cycle is *hamiltonian*, if it passes through all nodes of a graph.

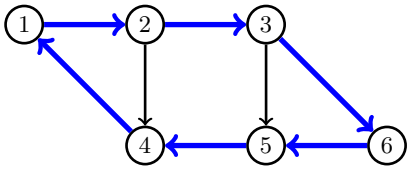


Figure 3.26: A graph with one hamiltonian and several other cycles

Absence or presence of cycles is a fundamental property of graphs. For instance, it can be shown that an undirected graph is *bipartite*, if and only if it does not contain cycles of odd length, see Exercise 3.6.1.3.

3.6 Acyclic graphs, trees

Being connected and/or being cyclic are the two most important properties a graph can have.

Acyclic graphs have no cycles. It is an interesting and nontrivial problem to find out whether a graph has this property. Directed acyclic graphs, in short *DAGs*, play a most important role in scheduling when they arise very naturally as *precedence graphs*. A typical example is shown in Figure 3.27.

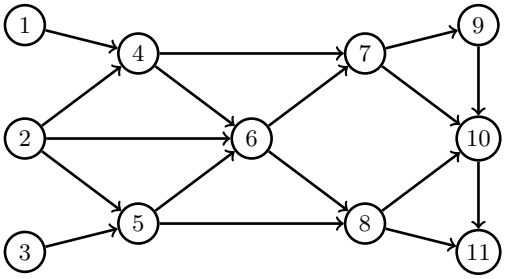


Figure 3.27: A DAG

A node k of a DAG is called a *source*, if its indegree $d_k^- = 0$, it is called a *sink*, it its outdegree $d_k^+ = 0$. The DAG in Figure 3.27 has three sources, the nodes 1,2 and 3, and one sink, the node 10.

A *tree* is a connected undirected graph without cycles. A disconnected undirected graph all whose components are trees is called a *forest*. For trees the following statements are always true:

- There exists exactly one path between any pair of nodes.
- Each edge is a *bridge*: if a bridge is removed the resulting graph becomes disconnected, the result is a *forest*.

Intrees and *outtrees* are the directed analogues of trees. They have a distinguished node v_0, the root, and:

- For intrees the *outdegree* d_i^+ of each node i is at most one.
- For outtrees the *indegree* d_i^- is at most one of all nodes.

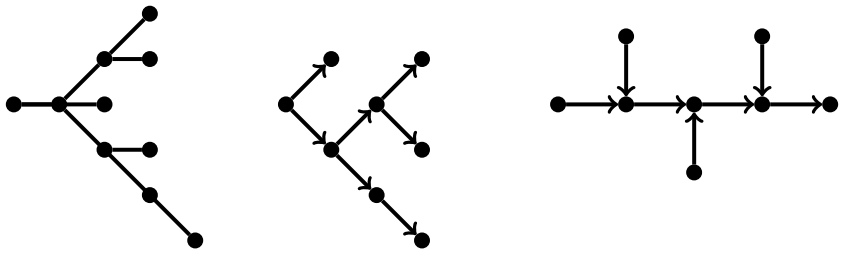


Figure 3.28: A tree, an outtree and an intree

For intrees and outtrees the following statements are always true:

- There exists exactly one path from the root to any other node.
- Each edge is a *bridge*: if an edge is removed the resulting graph becomes disconnected and is called a *in-forest* or *out-forest*.

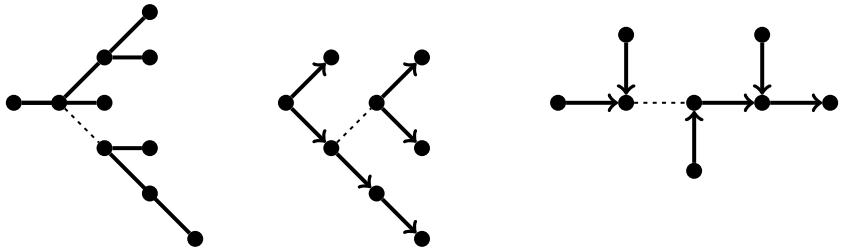
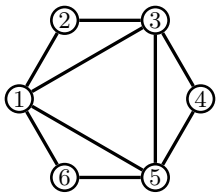


Figure 3.29: A forest, an out-forest and an in-forest

3.6.1 Exercises

1. *Euler Cycles (Leonhard Euler, 1735).* Given an undirected connected graph $G = (V, E)$, a cycle C is called an *Euler cycle*, if each edge appears exactly once on C. Prove that a graph G can have an Euler cycle only if the degrees of all of its nodes are *even numbers*.

2. Does the following graph have an Euler cycle? If so, find one. Does it have more than one Euler cycle? If so, determine their number.



3. Show that for a (undirected) graph to be *bipartite* it must not have cycles of *even* (≥ 4) length. That is to say: the absence of even cycles is a *necessary condition* for a graph to be bipartite.

4. Show that every undirected graph which does not have even cycles is bipartite. In other words, this condition is also *sufficient*. This exercise is a little bit more difficult than the former one.

5. Prove that in a tree there is exactly one path between any pair of nodes.

6. Show that if a tree has n nodes then its number of edges is necessarily $n-1$.

7. Prove that in an intree there is always exactly one path from a given node x to the root v_o. Prove also the analogous statement for outtrees.

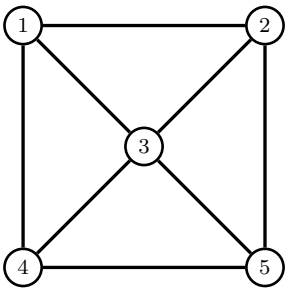
3.7 Data structures for graphs

3.7.1 Adjacency matrix and adjacency list

Human beings like pictures, computers do not, Eugen Lawler (1976, p. 20) one pointed out. So far, when introducing various concepts of graph theory we were able to *visualize* these by drawing diagrams. But this idea of *Anschaulichkeit*, a major concern of Felix Klein (1849-1925), has certainly its limitations when graphs become larger and large in order and size. Furthermore, important problems like deciding whether a given graph is acyclic or connected require an *algorithm*. It is impractical if not impossible to solve such problems by merely inspecting a diagram. Thus at the latest now we recognize the need of sophisticated data structures to represent graphs for computational purposes.

Recall, a graph $G = (V, E)$ is a binary relation on a finite set V. The most natural data structure to represent G is its *adjacency matrix*. This is a matrix $\mathbf{A}$ of order $n \times n$, $n = |V|$, with

$$\mathbf{A} = (a_{ij}), \quad a_{ij} = \begin{cases} 1 & \text{if } ij \in E \\ 0 & \text{if } ij \notin E \end{cases} .$$

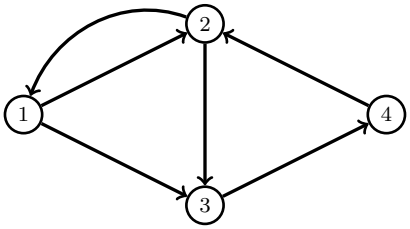


$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Figure 3.30: An undirected graph and its adjacency matrix

Figure 3.30 gives an example of an undirected graph G of order 5. Observe that its adjacency matrix is *symmetric*, because the binary relation underlying G has the symmetry property.

General graphs usually do not have this property. Figure 3.31 shows an example.



$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 3.31: A directed graph and its adjacency matrix

A major advantage of using the adjacency matrix to represent a graph is its simplicity: it is very easy to find out whether there exists an edge ij in G. Just check if $a_{ij} = 1$, and this can be done in constant time.

But this concept has also a downside, this is storage requirement. The minimum space requirement to store an adjacency matrix of order $n \times n$ is n^2 bits. For very large graphs this may be a problem. Think for example of the world wide web, where web pages are nodes and two nodes x and y are connected by an edge if page x has a hyperlink pointing to page y. The corresponding adjacency matrix requires about 10^{18} bits, thus roughly 1000 terabytes. Moreover, most of space allocated to store this matrix is wasted. While the order of this graph is enormous, its size compared to its order is moderate. Most entries of the adjacency matrix are zero. One also says that this graph is *sparse*.

A reasonable alternative is the *adjacency list*. Given a graph $G = (V, E)$ of order n, its adjacency list L is a list of length n. The list entry $L(i)$ at position i corresponds to node i and points itself to a list holding the elements $\Gamma(i)$, the nodes adjacent from i.

Consider for example the graph shown in Figure 3.32.

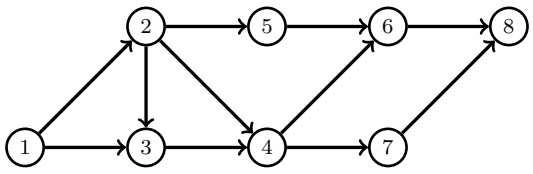
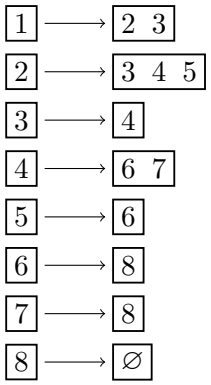


Figure 3.32

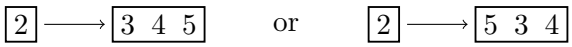
For this graph we determine:

$$\Gamma(1) = \{2, 3\}, \quad \Gamma(2) = \{3, 4, 5\}, \quad \Gamma(3) = \{4\}, \quad \Gamma(4) = \{6, 7\}$$
$$\Gamma(5) = \{6\}, \quad \Gamma(6) = \{8\}, \quad \Gamma(7) = \{8\}, \quad \Gamma(8) = \varnothing.$$

Using this information it is easy to form the adjacency list of G:



The entries pointed to by $L(i)$ are typically organized as *linked lists* which makes it easy to traverse them quickly. But, observe, normally there is no special order among the elements $L(i)$ points to, which is okay, since $\Gamma(i)$ is a *set*. So in the example above both would be valid:



Sometimes we may want to avoid this ambiguity, then we would sort the list entries in a particular way, e. g., by increasing values.

The major advantages of adjacency lists are:

- Compactness: The storage requirements of an adjacency list is proportional to $|V| + |E|$.
- Adding an edge can be done in constant time.
- When iterating through the neighbors of a node x this requires time proportional to the (out) degree of x. We shall see soon that systematically visiting all neighbors of a node is a very useful and important operation in graph algorithms

3.7.2 Numerical attributes

Assigning numerical attributes to nodes and/or edges of a graph greatly enhances their usefulness. Such attributes are also called *weights*.

Let us talk about attributes of edges first. Suppose that a graph models a network of roads with nodes representing cities. Typical attributes of its edges may be physical distances between cities connected by a road.

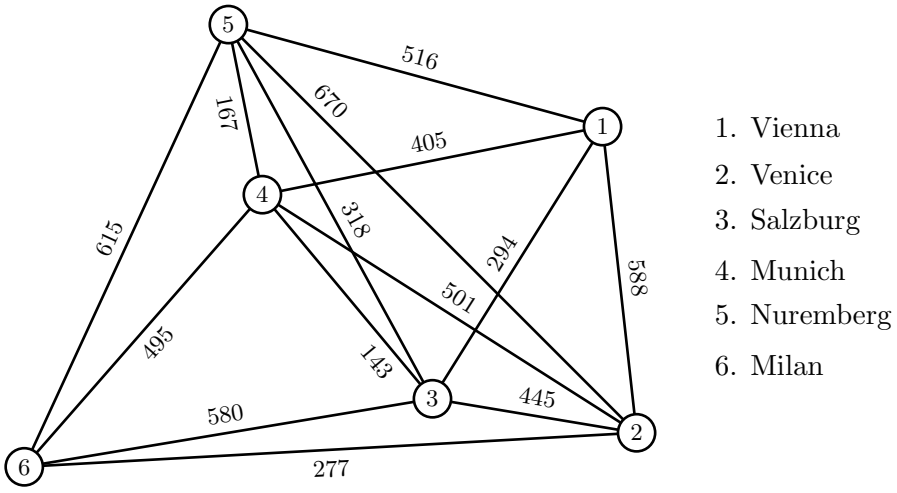


Figure 3.33: Distances between six European cities in km

Other weights of edges may be *probabilities*, e.g., p_{xy} may be the probability that the connection represented by edge xy is available. If the graph models a network of pipelines then a reasonable edge weight is transportation capacity of a pipe between two transmitter nodes.

Now the *length* of a path or a cycle becomes a somewhat different meaning. So far we have defined length as the number of edges on a path (cycle). Now length of a path may be:

- The *sum* of the weights of edges on a path or cycle, if weights represent distances.
- The *product* of the weights in case of probabilities.
- The *minimum* of the weights of edges on a path, e.g., when weights are transportation capacities.

For example in Figure 3.33, a path from Vienna to Milan may pass through Salzburg. Its length is obviously 874 km.

The most natural data structure to store this type of information is that of a *distance matrix* **D**. Its entry in row i and column j is just the distance, capacity, probability of availability of an edge. In a sense the concept of distance matrix is very similar to that of an adjacency matrix. The 1's on the adjacency matrix tell us that there *exists* an edge between two nodes. In the distance matrix we have in place of theses 1's some distance measure. But there is an important difference: if there is no edge between two nodes then the value of the corresponding entry in the distance matrix depends on the type of problem we want to solve.

▼ Example 3.3

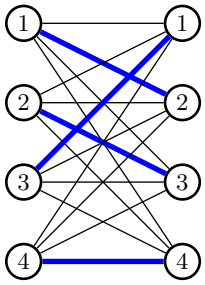
Consider the following scheduling problem: there are 4 jobs to be run on 4 machines. Each job must be assigned to exactly one machine, each machine

must be used, all jobs start at time zero and no preemption is allowed. The execution time of job i on machine j varies from machine to machine and is denoted by t_{ij}. In particular, we have the processing time data:

	Machine			
Job	1	2	3	4
1	7	9	8	9
2	2	8	5	7
3	1	6	6	9
4	3	6	2	2

Thus job 1 requires 9 minutes time on machine 2, and job 4 only 2 minutes on machine 3 or 4.

We are looking for an *assignment* jobs $\leftrightarrow$ machine which guarantees minimum total processing time. This scheduling problem is known as *Linear Sum Assignment Problem (LSAP)*. It can be formulated and solved as a minimum weight perfect matching on a complete bipartite graph $K_{4,4}$ with node sets $S =$ jobs and $T =$ machines. Any perfect matching M has a value which is the sum of weights of those edges which become part of M. A feasible solution of the *LSAP* is shown in Figure 3.34. It is indeed an optimal solution.



Assignments:
$1 \to 2,\ 2 \to 3,\ 3 \to 1,\ 4 \to 4$

Total processing time:
$T = t_{12} + t_{23} + t_{31} + t_{44} = 17$

Figure 3.34: A solution of the assignment problem

Later, in Chapter 4, we will outline algorithms for solving the *LSAP*.

▲

We may assign numerical attributes also to *nodes* of a graph. For example, if the graph models a transportation network, nodes represent facility locations and weights may be demands, supplies or production capacities. In a scheduling context we will often use graphs to represent *precedence relations*. In this case nodes are jobs to be processed and an edge uv means that job v cannot be started unless job u has been completed. The numerical weights assigned to nodes then are typically processing times.

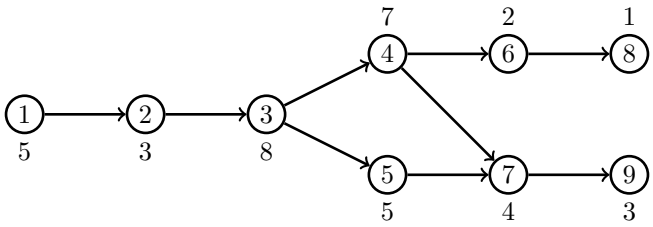


Figure 3.35: Precedence relations with processing times

The *length of a path* is now defined as the sum of the weights of all nodes lying on the path. Much more has to be said about precedence relations in sections 3.9 and 3.10. In Chapter 4 we will make the somewhat surprising observation that *paths of maximum length* are of particular importance in scheduling.

3.7.3 Exercises

1. Suppose we want to find a shortest route between two cities in a network of streets. If there is no road between cities x and y, what is an appropriate entry d_{xy} of the distance matrix?

2. Suppose we want to find a connection of maximum capacity between two transmitters in a network of pipelines for natural gas. If there is no link between transmitters x and y, what is an appropriate entry d_{xy} of the distance matrix?

3. Consider once more the precedence graph shown in Figure 3.35. Find a path from node $1 \rightarrow 9$ with maximum length. How would you interpret this number in a scheduling context?

3.8 Exploring a graph

Now, after having presented so many concepts its time to go one step further. So far our approach was a descriptive one. But now we will ask: *when does some graph have a particular property*?

Such questions are very important and to answer them will typically require an *algorithm*. Here are a few questions yet to be answered: Given a graph $G = (V, E)$,

- Does there exist a path between two nodes x and y?
- Is G acyclic?
- Is G bipartite?
- Is G connected, strongly connected?
- How can we identify its components, its strong components?
- What parts G are reachable from a given node?

All these questions are somehow related to *paths* in a graph. Therefore we need algorithms that allow us to *systematically walk* through a graph along a particular path and thereby visiting a subset or eventually all of its nodes.

One of the most effective ways to explore a graph is *depth-first search*. It has its origin in the work of Charles Pierre Trémaux (1859-1882) and his studies on *mazes*.

3.8.1 Depth-first Search

You will eventually have heard the ancient saga of Theseus and Ariadne. So you will know that two things are required to find a way out of a maze without getting trapped in an endless cycle: a chalk and a string. With the chalk we mark junctions when we are visiting them for the first time. The string helps us to find back to an already visited junction, if we got stuck at a dead end or return to a junction visited earlier. So the procedure of traversing a maze is essentially this:

- Take an unmarked passage unrolling the string behind you while walking along.
- Mark all intersections at the *first time* you are visiting them.
- Retrace steps if you arrive either at an already visited junction or a dead end.

The string guarantees that you will always find a way back, and the chalk marks help you to avoid visiting a passage twice.

These ideas carry over in a natural way to graphs, as there is a close correspondence between graphs and mazes, see Figure 3.36.

Both, chalk and string can be easily simulated in an algorithm. The chalk is just a variable whose value indicates that a certain node has been visited earlier. The string used for *backtracking* is implemented by *recursion*.

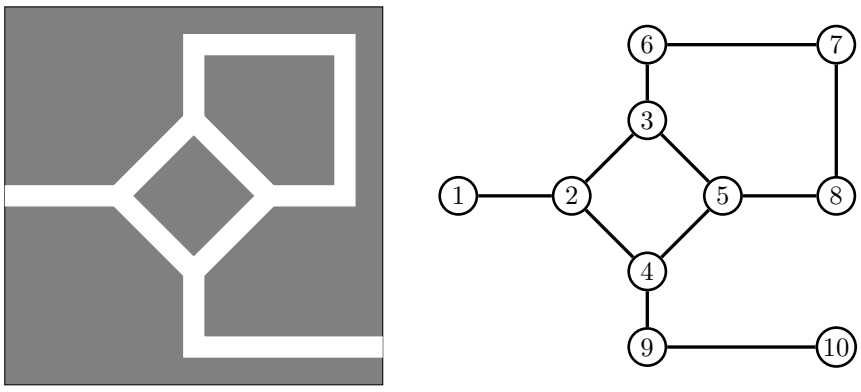


Figure 3.36: A very simple maze and its translation into a graph

Algorithm 3.1 to be presented below works an any graph G may it be directed or undirected. It starts at any node u and recursively explores all adjacent nodes until no more unexplored nodes can be found.

The chalk to mark a newly discovered node is a vector of dimension n, let us call it *visited*, where

$$visited[i] = \begin{cases} 1 & \text{if node } i \text{ has been discovered during search} \\ 0 & \text{otherwise} \end{cases}$$

Furthermore we will use a clock *time* to record the time when a new node is discovered and the time when exploration of a node is finished, i.e. it has no more unvisited nodes. Also we will keep track of the immediate *predecessor* of each newly discovered node. The reason for this overhead will become clear soon.

Algorithm 3.1

DFS
Input: a graph $G = (V, E)$
Output: three vectors
visited = vector indicating discovery of nodes
d = vector of discovery times
f = vector of finishing times
π = vector of predecessors

for $u \in V$ **do**
  begin
    $\pi[u] := 0$
    visited$[u] := 0$
  end

time $:= 0$

for $u \in V$ **do**
  if *visited*$[u] = 0$ **then** *explore(u)*

procedure *explore*
Input: a graph $G = (V, E)$ and a node $u \in V$

begin
  visited$[u] := 1$
  time $:=$ *time* $+ 1$
  $d[u] :=$ *time*
  for $v \in \Gamma(u)$ **do**
    begin

```
        if visited[v] = 0 then
            begin
                π[v] := u
                explore(v)
            end
    end

    time := time + 1
    f[u] := time
end
```

▼ **Example 3.4**

Our first example will be a demonstration of Algorithm 3.1 when run on an *undirected* graph, as it is shown in Figure 3.37.

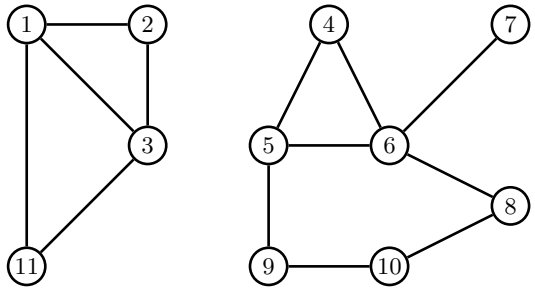


Figure 3.37

Starting in node 1 the first sequence of recursions results in the situation depicted in Figure 3.38. Visited nodes are shown in color gray, in the lower half of each visited node v we see the pair $d[v]/f[v]$.

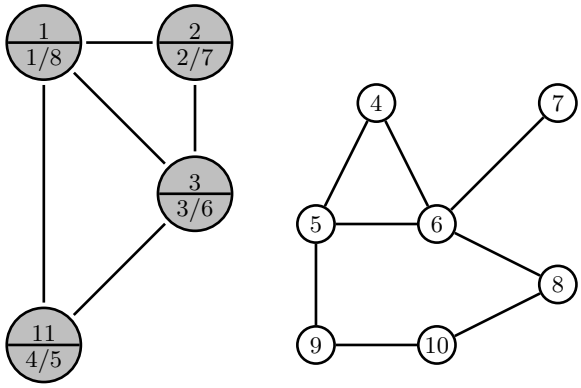


Figure 3.38

At this stage the predecessor vector π contains:

$$\pi = [0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 3]$$

Continuing with node 4 the exploration results in the graph shown in Figure 3.39.

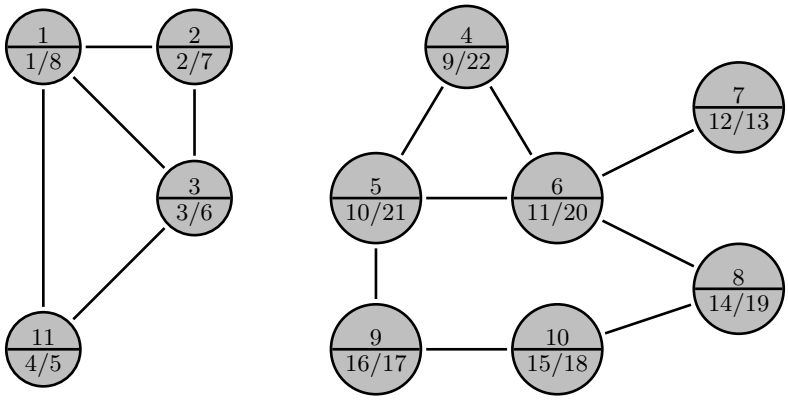


Figure 3.39

▲

The data generated by the *DFS*-search in Example 3.4 yield a lot of important information about the graph. Lets us look first hat the predecessor vector π which we display in this way:

$\pi[u]$	0	1	2	0	4	5	6	6	10	8	3
u	1	2	3	4	5	6	7	8	9	10	11

Each column in this table except for $u = 1$ and $u = 4$ represents an edge in a graph T with node set V. It is a *subgraph* of G and displayed in Figure 3.40.

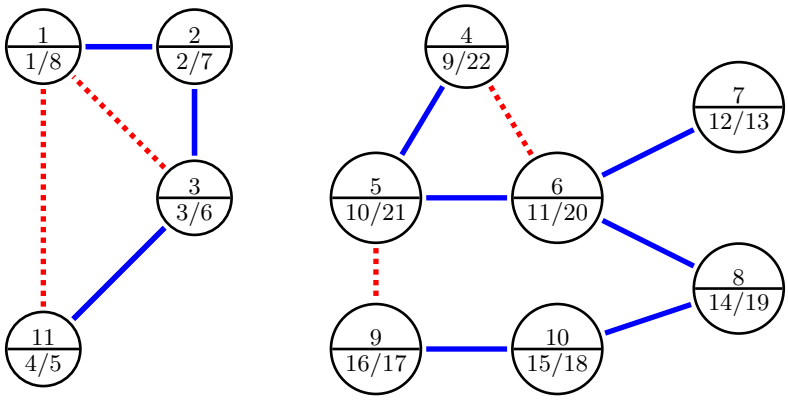


Figure 3.40

This figure shows the subgraph generated by the edges $(\pi(u), u)$ which are drawn in bold blue. The subgraph is a *forest*, thus a collection of trees, it

is known as *DFS-forest*. Each tree corresponds to a connected component of G. Once we start exploration in a node of a component, e.g. in node 1, *DFS* explores all nodes of this component. Then it is restarted in a node so far undiscovered, in the example node 4, and explores all nodes which can be reached by a path.

Note that it is a easy exercise to adapt Algorithm 3.1 in such a way that it enumerates and marks all connected components of an *undirected* graph.

But much more has been found out.

Discovery times d_i and finishing times f_i returned by Algorithm 3.1 yield additional valuable information. In our example graph:

d_i	1	2	3	9	10	11	12	14	15	16	4
f_i	8	7	6	22	21	20	13	19	17	18	5

Of particular interest are edges (u, v) such that

$$d_v < d_u < f_u < f_v \tag{3.4}$$

These *nesting inequalities* hold for instance for edges $(3, 1)$ and $(11, 1)$ in the first component of our example graph. Such edges of G are not part of the *DFS*-forest. They are called *back edges*. Recall, when we add an edge to a tree this always results in a *cycle*. Thus whenever we encounter a back edge in an undirected graph then it will be cyclic.

In Figure 3.40 back edges are drawn in red dotted lines. Indeed, they create (several) cycles.

DFS-search is very fast, it is indeed *linear* in size and order of a graph, its complexity is $O(|V|+|E|)$ when using adjacency lists to represent G numerically.

Algorithm 3.1 works without any changes also on *directed graphs*.

▼ Example 3.5

Let us consider the graph $G = (V, E)$ displayed in Figure 3.41.

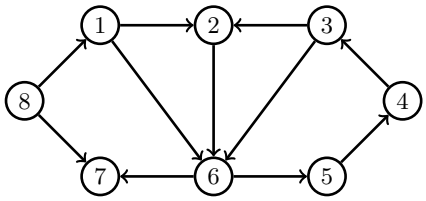
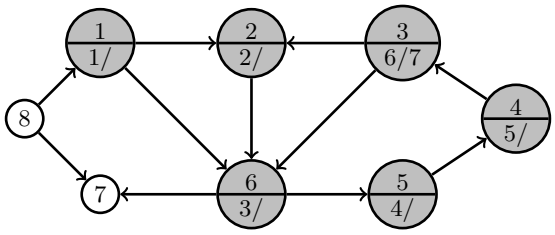
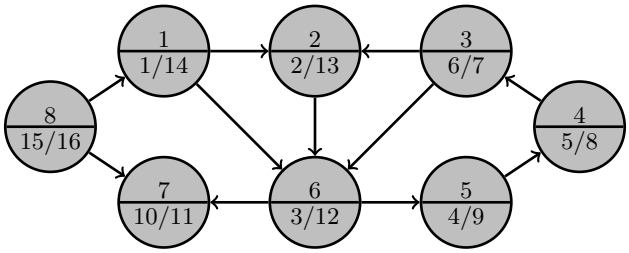


Figure 3.41

When we start *DFS* in node 1 it explores nodes 2, 6, 5, 4 and 3 in this order.



Node 3 has no undiscovered neighbors thus *DFS* tracks back to node 6 and discovers node 7.



As there are no undiscovered nodes adjacent to 7, *DFS* tracks back to node 1 and starts exploration of the last undiscovered node 8. Here *DFS* stops since node 8 has no undiscovered neighbors.

▲

Again it is instructive to have a look at the data generated by *DFS*. The predecessor vector π in Example 3.5 has been found to be

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

The pairs $F = \{(\pi(u), u), u \in V\}$ form the edge set of a subgraph $H = (V, F)$ of G. As for undirected graphs this is a forest, more precisely the *DFS out-forest* of G.

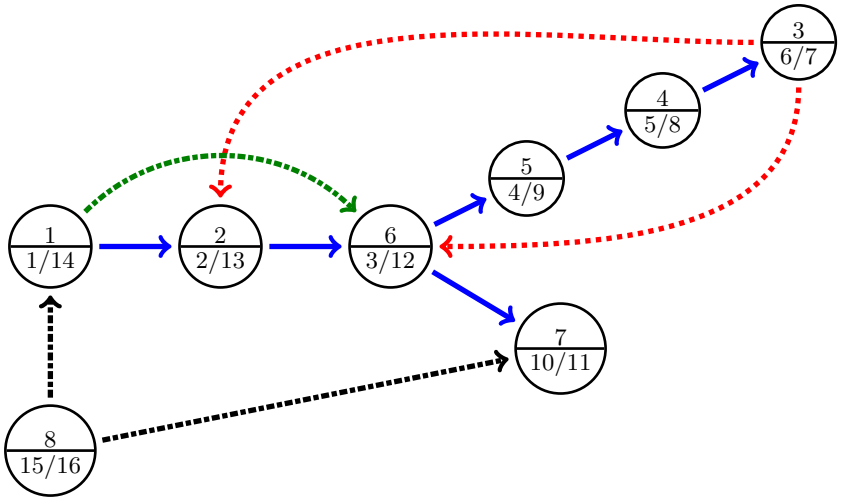


Figure 3.42: The *DFS*-outforest of Example 3.2

As was the case with undirected graphs there may be edges of $G = (V, E)$ which are not part of any *DFS*-outtree. Their classification is quite informative and based again on discovery and finishing times d_u and f_u. Consider an edge $uv \in E$ and let us represent the discovery time of node u by a left bracket $[_u$, its finishing time by a right bracket $]_u$. Based on the ordering of discovery and finishing times of two nodes u and v we have:

- If the ordering is

$$[_u \; [_v \;]_v \;]_u,$$

  then uv is either a tree edge or a *forward edge*. The latter lead from a node to a non-child descendant in the same a *DFS*-tree. In our example there is one forward edge (drawn as green dotted arrow) from node 1 to node 6.
- If the graph has an edge uv with ordering

$$[_v \; [_u \;]_u \;]_v,$$

  then uv is called a *back edge*. In Figure 3.42 there are two back edges, $(3, 2)$ and $(3, 6)$.
- If there is an edge with

$$[_v \;]_v \; [_u \;]_u,$$

  it is called a *cross edge*. Such edges lead neither to a descendant nor to a predecessor. They lead from a node whose exploration as been finished to another finished node. In our running example there are two cross edges, $(8, 1)$ and $(8, 7)$.

These relations are known as *Parenthesis Theorem*.

Back edges are the most interesting: whenever *DFS* identifies a back edge in a graph G, then it must have a cycle. In other words, a graph is acyclic if and only if it has no back edges.

Another important application of *DFS* is to study connectivity properties of a graph. In case of an undirected graph this is very easy:

- If procedure *DFS* calls *explore* with a node u, *explore* calls itself recursively until all nodes which can be reached from u have been discovered. Since these nodes are all connected to u by a path they form a connected component of G, that component which contains u.
- *DFS* then starts *explore* with a new yet undiscovered node provided there is one and discovers the next connected component of G, and so on.

Determining the *strong components* of a directed graph is more difficult, still it can be done very elegantly by means *DFS*. Here *DFS* is part of the famous *Kosaraju-Sharir* algorithm. The interested reader is referred to Sedgewick and Wayne (2011, pp. 590).

Remark: one final comment on depth-first search is in order. The *DFS*-forest is usually not uniquely defined. Observe that for a given node u *explore* processes the neighbors in $\Gamma(u)$ in *some* order. Normally this order is given by the way the entries in the adjacency list are organized, or it is the way nodes have been numbered, when we work with the adjacency matrix of a graph. Depending on the way the nodes in $\Gamma(u)$ are processed we obtain different *DFS*-forests. Sometimes it may be necessary to avoid this ambiguity and impose some ordering on the sets $\Gamma(u)$.

3.9 Precedence Graphs

3.9.1 Partial Orders

Probably the most important applications of graph theory in scheduling deal in one or the other way with *dependent jobs*. In many production and service environments jobs cannot be processed in arbitrary order. Then, when forming a schedule certain restrictions on order have to be obeyed. Typically such restriction are due to:

- *technological impossibilities*,
- *practicability*,
- *availability of resources*,
- *constraints imposed by transportation and facility location*, etc.

On a formal level precedence relations define a *partial order* P on a set of jobs J. P is defined as a pair $P = (J, \rightarrow)$ where $\rightarrow$ defines an *order relation* in the following way: for $a, b \in J$: $a \rightarrow b$ means, job b cannot be started unless job a has been completed.

The relation $\rightarrow$ satisfies:

- *Irreflexivity*: for all $a \in J$: $a \not\to a$.
- *Antisymmetry*: $a \to b$ implies $b \not\to a$ for all pairs of jobs for which $\to$ is specified.
- *Transitivity*: if $a \to b$ and $b \to c$, then $a \to c$.

We say that two jobs are *comparable*, if either $a \to b$ or $b \to a$, otherwise a and b are *incomparable*, symbolically: $a \| b$.

A set of pairwise comparable jobs is called a *chain*, and a set of pairwise incomparable jobs an *antichain*. The empty set $\varnothing$ and sets $\{a\}$ consisting of a single job are by definition both, chain and antichain.

The alert reader may recall that we have defined a graph $G = (V, E)$ as an irreflexive binary relation E on some finite set V. Augmenting E by asymmetry and transitivity makes it a partial order $P = (J, \to)$ and therefore creates a special type of graph, the *precedence graph* $G(P) = (J, E)$ corresponding to P. It is constructed as follows:

- The nodes of $G(P)$ are the elements of P, in our case jobs to be scheduled.
- $G(P)$ has an edge ab job if a precedes b in P, i.e. $a \to b$.

For notational convenience we will mostly write G instead of $G(P)$ when dealing with precedence graphs.

The precedence graph G is a always a *directed acyclic graph*. Such *DAGs* have been already introduced in Section 3.6. That G is a directed graph follows from the ordering relation $\to$, That it is *acyclic* is a consequence of *transitivity*. To see this, assume that G has a cycle

$$a_1 \to a_2 \to a_3 \to \ldots \to a_{k-1} \to a_k \to a_1$$

Then by transitivity $a_1 \to a_2 \to a_3 \implies a_1 \to a_3$ and therefore successively:

$$\begin{aligned}
&a_1 \to a_3 \to \ldots \to a_{k-1} \to a_k \to a_1\\
&a_1 \to a_4 \to \ldots \to a_{k-1} \to a_k \to a_1\\
&\quad\vdots\\
&a_1 \to a_k \to a_1\\
&a_1 \to a_1.
\end{aligned}$$

The last statement violates the postulate of irreflexivity. So G must be acyclic. If we would drop the transitivity requirement, then we would run into series logical troubles. Suppose we have a partial order P on three jobs with:

$$1 \to 2, \quad 2 \to 3, \quad 3 \to 1,$$

Job 2 cannot start unless job 1 has been completed, job 3 requires finishing job 2 first, but job 1 requires completion of 3? Of course, this logical inconsistency is due to the fact that the graph $G(P)$ displayed in Figure 3.45 is a *directed 3-cycle*. While it is an easy matter to detect such inconsistencies in small partial orders, the situation is different, if we deal with several hundred or thousands of

dependent jobs. But we know how to detect cycles: just perform a *DFS*-search and watch out for back edges.

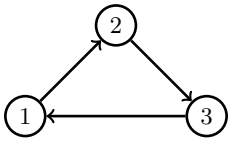


Figure 3.43: Precedence graphs must be acyclic

▼ Example 3.6

Suppose we have four jobs $1, 2, 3$ and 4 and it is known that production requires:

$$1 \to 3, \quad 1 \to 4, \quad 2 \to 3, \quad 3 \to 4$$

These precedence constraints define four edges in the precedence graph:

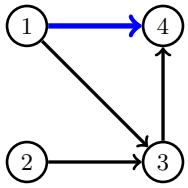


Figure 3.44: A precedence graph G

A closer look at this graph reveals that the edge $(1, 4)$ is *redundant*, it is *implied* by transitivity, because

$$1 \to 3 \to 4 \implies 1 \to 4$$

Thus the precedence graph without edge $(1, 4)$

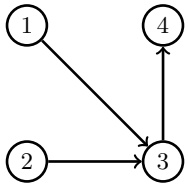


Figure 3.45: Transitive reduction of G

represents exactly the same precedence relations as the graph in Figure 3.44. Indeed, there are many ways to define a precedence graph given a partial order $P = (J, \to)$. However, among these there are two which are of particular importance.

The *transitive reduction* G_0 contains as edges only those which are not implied by transitivity, this is the *minimal* precedence graph resulting from a partial order P.

The *transitive closure* G_c of a precedence graph G has an edge $a \to b$ whenever there is a *path* from a to b in G.

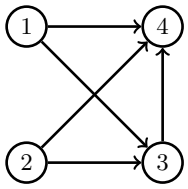


Figure 3.46: Transitive closure G_c of G

These observations lead us to two important definitions:

- If there is an edge (a, b) in the transitive reduction of a precedence graph, then a is an *immediate predecessor* of b and b an *immediate successor* of a. In this case there is no job c such that

  $$a \to c \to b$$

  We may also write $b \in \Gamma(a)$ and $a \in \Gamma^{-1}(b)$, using the mapping Γ defined earlier in this chapter.
- If there is an edge (a, b) in the transitive closure then there exists a *path* from a to b in G and, of course also in the transitive reduction. In this case a is called a *predecessor* of b and b a *successor* of a.

So two questions arise now:

- When should we use the transitive closure of a precedence graph, when its transitive reduction?
- How can we *determine* these given a precedence graph?

For the first question, this is a matter of efficiency. Many scheduling algorithms are iterative and contain statements like this one:

for all successors k of job $a \in J$ **do** *something*

This requires to determine whether there is a path from a to k over and over again which may be very time consuming. But having calculated the transitive closure of a precedence graph we can find out very quickly whether there is such a path: just look if there is an edge (a, k).

On the other hand, transitive closures are *very dense graphs*, thus the appropriate data structure to represent such a graph is its adjacency matrix which is typically dense and may require a lot of storage space, see Figure 3.47 for an example.

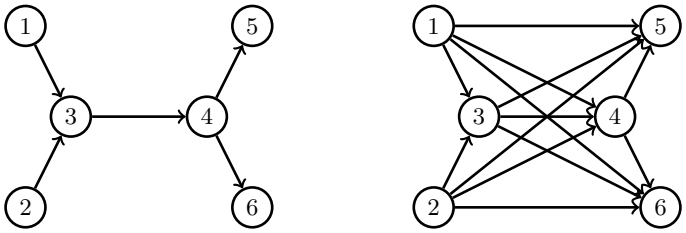


Figure 3.47: A precedence graph G and its transitive closure of G_c

The graph G in Figure 3.47 is the transitive reduction of graph shown in the right half.

Algorithm 3.2 given below determines the transitive closure of a precedence graph by the *Floyd-Warshall Algorithm* due to Floyd (1962) and Warshall (1981). Its idea is a very simple one: all pairs of edges are checked, and when a pair implies an edge by transitivity this edge is added to E.

Algorithm 3.2

TClose
Input: a DAG $G = (V, E)$
Output: the transitive closure G_c

$V(G_c) := V(G)$
$E(G_c) := E(G)$

for $x \in V$ **do**
  for $y \in V$ **do**
    for $z \in V$ **do**
      if $xy \in E$ **and** $yz \in E$ **then**
        $E := E \cup \{xz\}$

Algorithm 3.3 does the reverse of Algorithm 3.2. It successively deletes edges which are transitively redundant.

Algorithm 3.3

TReduce
Input: a DAG $G = (V, E)$
Output: the transitive reduction G_0

$V(G_0) := V(G)$
$E(G_0) := E(G)$

for $x \in V$ **do**
  for $y \in V$ **do**

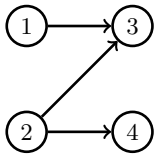
for $z \in V$ **do**
  if $xy \in E$ **and** $yz \in E$ **then**
    $E := E - \{xz\}$

3.9.2 Linear extension and topological ordering

Consider a set of four jobs $J = \{1, 2, 3, 4\}$ to be processed on a single server and assume that technology affords:

$$1 \to 3, \quad 2 \to 3, \quad 2 \to 4, \tag{A}$$

Thus (A) defines a partial order on J with precedence graph:



How can we find *some* schedule for these data? Note that we are not talking about finding an *optimal schedule* at this point.

Finding a schedule for a single-server problem without inserted idle time and preemption means finding a *job ordering*, as we have found out in Chapter 2. However, among the $24 = 4!$ schedules of 4 jobs, in the present situation we only accept permutations of $\{1, 2, 3, 4\}$ which are not in conflict with the precedence constraints (A). Such schedules are called *feasible*.

In our problem we find that there are *incomparable jobs*, in particular, $1 \| 2$ and $3 \| 4$. Thus we have a choice regarding the order in which 1 and 2 appear in the schedule and we have also a choice regarding the pair 3 and 4. But we have no choice when scheduling, for instance, 1 and 3. For a schedule to be feasible, 1 must always come before 3.

Now, it is easily verified that the only permutations of $\{1, 2, 3, 4\}$ that are not in conflict with (A) are:

$$[1, 2, 3, 4], \quad [1, 2, 4, 3], \quad [2, 1, 3, 4], \quad [2, 1, 4, 3], \quad [2, 4, 1, 3]$$

Thus our problem has only five feasible schedules.

Once we have chosen a particular schedule, say for instance $S = [2, 1, 3, 4]$ we have to decided to process jobs in this order:

$$2 \to 1 \to 3 \to 4 \tag{B}$$

But, (B) is also a partial order on J, indeed, it a partial order which has no incomparable pairs of jobs. Thus S is a chain!

Given a partial order $P = (J, \to)$ on n elements of J, a partial order S is called a *linear order*, if it has no incomparable elements. S is called a *linear extension*

of P, if it is linear and not in conflict with P. A fundamental theorem in the theory of partial orders, the *order-extension principle* by Edward Szpilrajn (1930) postulates that for any partial order a linear extension can be found. But in general, a linear extension is not uniquely defined.

Two questions come up immediately:

- How can we find a linear extension of a given partial order P? In other words: given a precedence graph G, how can we find a feasible schedule conforming to P?
- How many linear extensions does a given partial order P have?

Let's deal with the second question first: unfortunately there is no easy answer, there is no formula finding the number of linear extensions of a partial order P. Actually, it can be shown that counting the number of linear extensions is a very hard problem except when P has some very simple special structure.

Now to the first question: one way of finding a linear extension is to *renumber* nodes in such a way that if $xy \in E$ then necessarily $x < y$. In other words, an edge always leads from a node with lower number to a node with higher number. This renumbering, also known as *topological sorting*, not only yields a feasible schedule, it is also the first step of many algorithms, which rely on a *preordering* of nodes. This is very helpful if we have to enumerate cleverly all feasible schedules. Some of these algorithms will be discussed in the next chapter.

And now to the good news: topological sorting can be done easily and efficiently by *depth-first search*. Once we have determined the *finishing times* of the nodes, renumbering goes as follows: just sort the finishing times by *decreasing values* and renumber nodes accordingly. This is best explained by an example.

▼ Example 3.7

The reader may recall the Bicycle Problem discussed in Chapter 1. Production of a bicycle involves 10 jobs, their dependencies are represented by the precedence graph shown

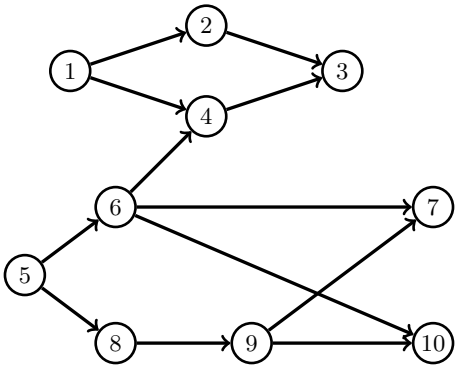


Figure 3.48: The precedence graph of Graham's Bicycle Problem

We want to sort this precedence graph topologically. Using Algorithm 3.1, *DFS* yields finishing times:

i	1	2	3	4	5	6	7	8	9	10
f	8	5	4	7	20	15	12	19	18	14

Sorting the f_i by decreasing values yields:

i	5	8	9	6	10	7	1	4	2	3
f	20	19	18	15	14	12	8	7	5	4

Thus one of a many of feasible schedules[5] is

$$S = [5, 8, 9, 6, 10, 7, 1, 4, 2, 3]$$

The reader may verify that this sequence is not in conflict with the precedence relation shown in Figure 3.48. Renumbering nodes according to S yields the topological ordering of the precedence graph:

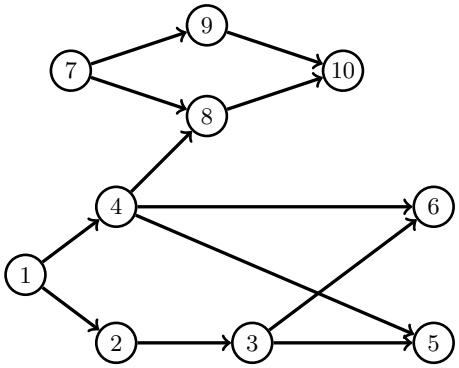


Figure 3.49: A topological ordering

▲

3.9.3 Exercises

1. Algorithms 3.2 and 3.3 are independent of the representation of the graph G. G may be represented by its adjacency matrix or by its adjacency list. Work out the details.

2. Determine the time complexity of algorithms 3.2 and 3.3. Is it better to use the adjacency list representation or the adjacency matrix of G?

3. Algorithms 3.2 and 3.3 both require the input graph to be acyclic. What would happen if this condition is not satisfied?

[5]Recall the remark on ambiguity of *DFS* on page 82.

4. Devise an adaptation of *DFS* by introducing appropriate variables to *count* the number of linear extensions of a partial order. Use this to show that for the precedence graph of Figure 3.48 there are 1490 feasible schedules. This is quite a small number compared to $n! = 3628800$ schedules if jobs were independent.

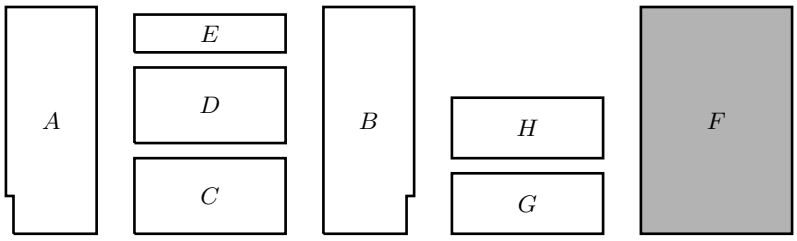
3.10 SP-graphs

3.10.1 Series and parallel composition

There is an important class of precedence graphs which have a very simple structure, simple enough so that some notoriously hard scheduling problems become tractable. This is the class of *series-parallel graphs* or *sp-graphs*. Before we give a formal definition of sp-graphs let us discuss first an example.

▼ Example 3.8

A home center offers a book case for do-it-yourself construction. It is cheap, just perfectly suited for student apartments. After unpacking you find a lot of screws and bolts and these pieces:

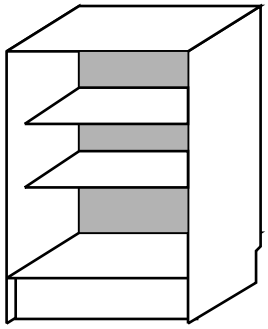


There is also a folder with assembly instructions, it tells you how to assemble pieces to get a wonderful bookcase in no more than 8 steps:

Job	Action
1	fix base board E to left side plate A
2	fix base board E to right side plate B
3	fix bottom board C to left side plate A
4	fix bottom board C to right side plate B
5	fix top board D to left and right side plates
6	fix back board F
7	install case board G
8	install case board H

These instructions will also contain a picture showing you how the book case should look like finally[6]:

[6] if things work out smoothly, of course...



So, how to assemble the book case? In the instructions you will find a lot of pictures telling you essentially this:

- You may first fix the base board E to the left side plate A followed by fixing the bottom board C to A. Thus: $1 \to 3$.
- Alternatively, you may start fixing E to the right side plate B followed by the bottom board C. Thus $2 \to 4$.
- Now put together left and right side plates with E and C already mounted and fix the top board D. This is job 5.
- Next fix back board F, job 6.
- Either install first case board G and then H or do it the other way round.

At this point the bookcase should be finished, hopefully.

Now having learned a lot about precedence relations you know that these assembly instructions define a partial order on the set of jobs $J = \{1, 2, 3, 4, 5, 6, 7, 8\}$. What is the corresponding precedence graph, more precisely, its *transitive reduction*? Here it is:

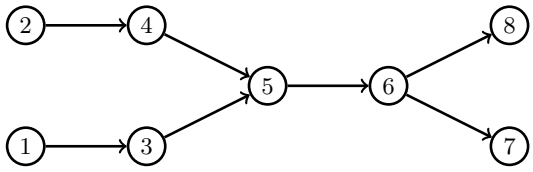


Figure 3.50: The precedence graph for the bookcase

This is a typical example of a *series-parallel graph.*

▲

Series-parallel graphs are constructed *recursively* following these rules:

- If V contains only a single node and E is empty, then $G = (V, E)$ is series-parallel.
- *Parallel composition*: given two sp-graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with no common nodes, i.e. $V_1 \cap V_2 = \varnothing$, the parallel composition is a sp-graph $G = G_1 + G_2$ with nodes V and edges E defined

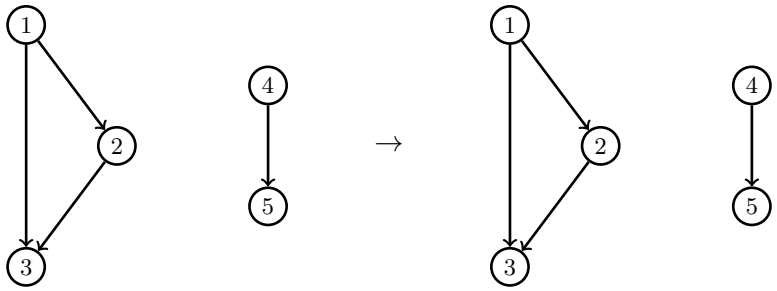
by:

$$V = V_1 \cup V_2, \quad E = E_1 \cup E_2$$

- *Series composition*: given two sp-graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $V_1 \cap V_2 = \varnothing$, the series composition is an sp-graph $G = G_1 \cdot G_2$ with nodes V and edges E defined by:

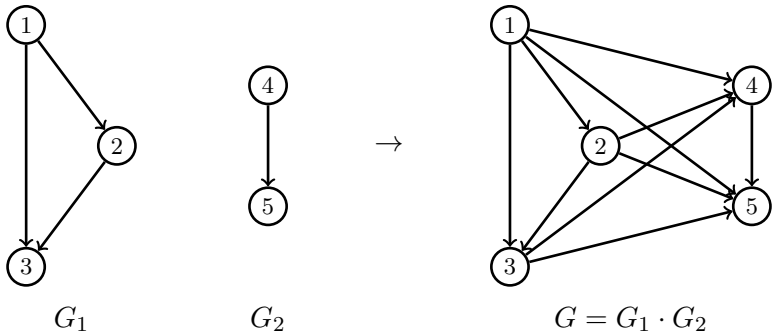
$$V = V_1 \cup V_2, \quad E = E_1 \cup E_2 \cup \{xy : x \in V_1, y \in V_2\}$$

This needs some explanation. Parallel composition is easy: just put two different sp-graphs G_1 and G_2 together to get a new sp-graph G which is disconnected and has (at least) components G_1 and G_2.



Note that parallel composition is *commutative*: $G_1 + G_2 = G_2 + G_1$.

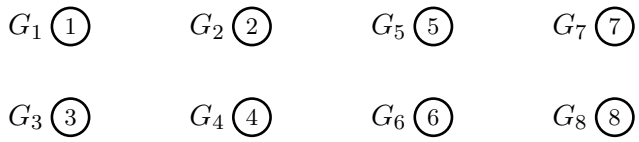
In case of a series composition $G = G_1 \cdot G_2$ each node x of G_1 is joined with each node y of G_2 by a directed edge xy.



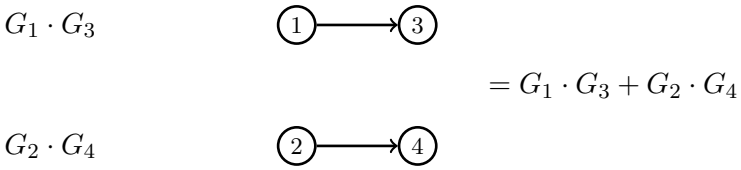
Series composition is *not commutative*, i.e. $G_1 \cdot G_1 \neq G_2 \cdot G_1$.

▼ **Example 3.8** *(continued)*

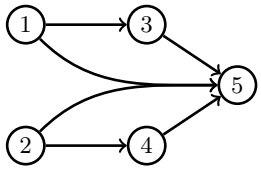
At the beginning we have 8 graphs $G_1, G_2, \ldots, G_8$ each having only one node, namely the jobs $1, 2, \ldots, 8$. These singletons are sp-graphs by definition.



Next we model the dependency of jobs 1 and 3 and 2 and 4. These are two series-compositions which are put in parallel:

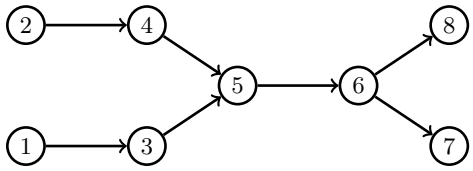


G_5 is joined by a series composition:



$$(G_1 \cdot G_3 + G_2 \cdot G_4) \cdot G_5$$

Next we join job G_6 and the parallel composition $G_7 + G_8$ by series composition (for clearness only the transitive reduction is drawn below):



$$(G_1 \cdot G_3 + G_2 \cdot G_4) \cdot G_5 \cdot G_6 \cdot (G_7 + G_8)$$

▲

3.10.2 The decomposition tree

One of the messages of Example 3.8 was that a series-parallel precedence graph can be expressed as an *algebraic expression*. In the example:

$$G = (G_1 \cdot G_3 + G_2 \cdot G_4) \cdot G_5 \cdot G_6 \cdot (G_7 + G_8) \tag{3.5}$$

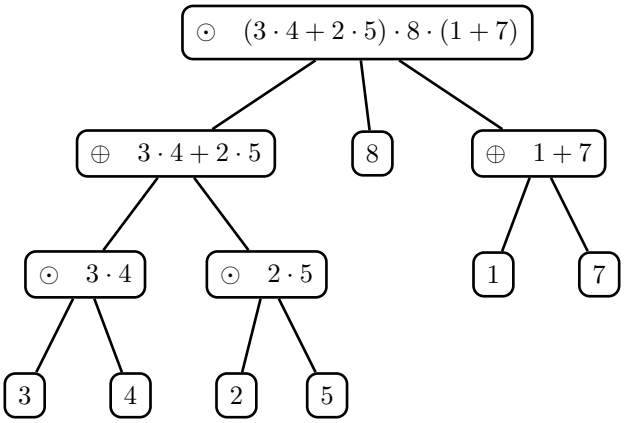
This expression is made up by constants and operators: the operators are a commutative *addition* $+$ and a non commutative multiplication $\cdot$, with multiplication having precedence over addition.

To gain a little bit more insight into (3.5) let us assume for the moment that the constants are natural numbers and addition and multiplication have now their usual meaning, so that multiplication is commutative: for instance (3.5) may now look like:

$$(3 \cdot 4 + 2 \cdot 5) \cdot 8 \cdot (1 + 7) \tag{3.6}$$

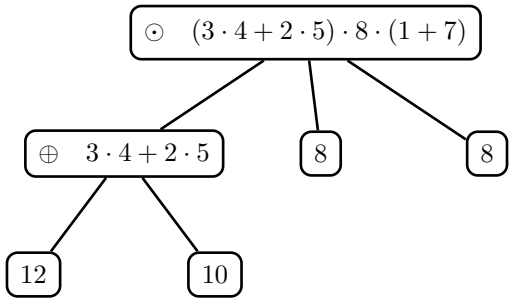
This expression *in numbers* is structurally fully equivalent to (3.5). But, how is it evaluated?

Computers, even simple pocket calculators, evaluate (3.6) by constructing a *parse tree*:



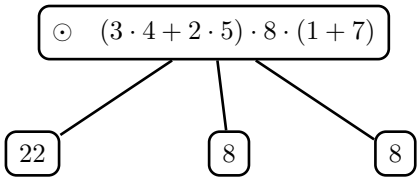
Observe that each inner node has a label indicating the way its children are to be composed, either by $\oplus =$ *addition* or by $\odot =$*multiplication*.

Evaluation is done by working through the tree in a *bottom-up manner*. First leaves are combined according to the label of their common father[7]:



In the next step the tree reduces to

[7] I have to apologize for this somewhat patriarchic nomenclature, obviously it is of biblic origin but still standard in graph theory. This apology is copywrited by Martin Golumbic.



A final reduction yields:

$$(3 \cdot 4 + 2 \cdot 5) \cdot 8 \cdot (1 + 7) = 22 \cdot 8 \cdot 8 = 1408$$

The nice thing about sp-graphs is: it is always possible to represent them by an algebraic expression and to expand this into a *parse tree* $\mathcal{T}$. This tree is called a *decomposition tree*. Once $\mathcal{T}$ has been determined we can evaluate the tree in a bottom-up fashion, though the process of evaluation now depends on the scheduling problem to be solved. This evaluation process will be discussed in greater detail in Chapter 5.

Here we will confine ourselves to the problem of *finding* the decomposition tree $\mathcal{T}$. And in passing we will answer also another important question which we have not touched so far:

Are all precedence graphs series-parallel? If not, how can we find out?

Here is a quick answer to the first question: *no*, not all precedence graphs are series-parallel. The following is a counter example.

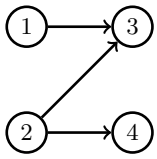


Figure 3.51: A non sp-precedence graph

It is impossible to construct this precedence graph by series and parallel compositions only. In fact, this graph is known as **N**-structure, and any precedence graph having this **N**-structure as subgraph cannot be series-parallel therefore **N** is also called a *forbidden substructure*.

The vehicle to detect such forbidden subgraphs and to find the decomposition tree of a given precedence graph is the concept of a *comparability graph*.

Given a precedence graph $G(J, E)$ its *comparability graph* $\widetilde{G}(J, E)$ is constructed as follows:

- Form the transitive closure G_c of G.
- Delete orientation of edges in G_c.

Let us illustrate this concept by two examples. The transitive closure of G shown in 3.52 is just G, deleting orientation of edges we obtain an interesting structure:

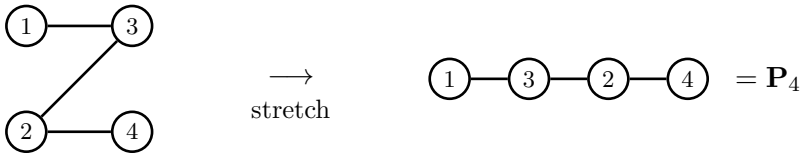


Figure 3.52: The comparability graph of G given in Figure

$\mathbf{P}_4$ is a *path* consisting of four nodes. Whenever it occurs in a comparability graph, then the underlying precedence graph is not series-parallel.

The comparability graph of the precedence graph in Example 3.8 (Figure 3.50) is a bit more complicated, it is displayed in Figure 3.53.

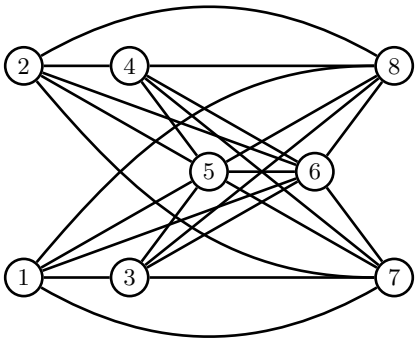


Figure 3.53: Comparability graph of Example 3.4

Comparability graphs have many interesting properties, among these for our purposes one of the most important ones is this: they *preserve* structural properties of the underlying precedence graph (Buer and Möhring, 1983). Thus if a precedence graph G is series-parallel, then this property is also present in a certain sense in the corresponding comparability graph $\widetilde{G}$.

The following Algorithm 3.4 is based on ideas due to (Gallai, 1967), but see also (Möhring, 1989). It produces the decomposition tree $\mathcal{T}$ of the comparability graph of a partial order or stops with the message that the partial order is not series-parallel.

Let $C := C(G)$ be the comparability graph of a precedence graph G. Basically, the algorithm does the following:

- Initially $\mathcal{T}$ consists of a single root node which is has a *data field* holding all nodes of C.
- If C is disconnected then it must be a parallel construction and the connected components are its parts. In this case the root node is marked by a $\oplus$ to indicate that its children are part of a parallel composition. These parts are attached as children to the root and each child has an own data field holding the nodes of each part.
- Otherwise: if C is connected, we form its *complement* $\overline{C}$. Recall, that

now each edge becomes a non-edge and each non-edge an edge. If $\overline{C}$ is disconnected, then the root node is marked by a $\odot$, thus showing us that its children are combined by a series-composition. For each part a new child is attached to the root node with corresponding data field.

- If both, C and $\overline{C}$ are connected, then G cannot be series-parallel. The algorithm stops at this point and outputs an appropriate message.
- Otherwise the algorithm is called recursively in a depth-first manner on the subgraphs of C induced by the nodes in the data field of children of the root, making each one to the root of a subtree until a child is encountered which contains a single node.

Algorithm 3.4

procedure *SPTree*

Input: the comparability graph C of a precedence graph $G = (V, E)$
Output: the decomposition tree $\mathcal{T}$ of G or
a message that G is not series-parallel

begin
$N := V$ [**comment:** the root of the tree to be constructed]
$\mathcal{T} := \{N\}$
Loop:
Compute the connected components of the induced subgraph $C|N$
if $C|N$ is disconnected **then**
begin
Label N with $\oplus$ as *parallel*
Assign the node sets of the connected components
of $C|N$ as children to N
go to *Stop*
end

Compute the connected components of the complement $\overline{C}|N$ of $C|N$

if $\overline{C}|N$ is disconnected **then**
begin
Label N with $\odot$ as *series*
Assign the node sets of the connected components
of $\overline{C}|N$ as children to N
go to *Stop*
end

if $C|N$ and $\overline{C}|N$ are both connected **then**
begin
Output message that C is not series-parallel
return
end

Stop:
if there is an unlabeled node N' with ≥ 2 nodes in its data field **then**

```
        begin
            N := N'
            go to Loop
        end
    else
        return T
```

▼ **Example 3.4** *(continued)*

We start with the comparability graph $C(G)$ shown in Figure 3.53. Initially the tree $\mathcal{T}$ has a single node N containing all nodes of C, so $N = \{1, 2, \ldots, 8\}$.

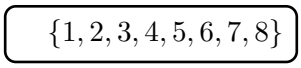
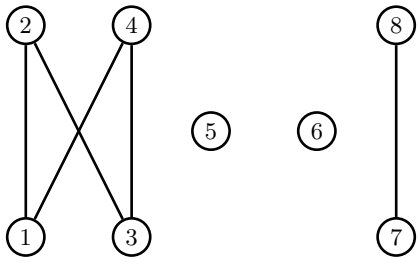


Figure 3.54: The root of the decomposition tree in Example 3.4

As C is connected, we construct the complement $\overline{C}$:



$\overline{C}$ has 4 connected components, with node sets

$$V_1 = \{1, 2, 3, 4\}, \quad V_2 = \{5\}, \quad V_3 = \{6\}, \quad V_4 = \{7, 8\}$$

We mark the root of the tree with $\odot$ and attach as children in this order new nodes containing the sets V_1, V_2, V_3 and V_4 as data fields:

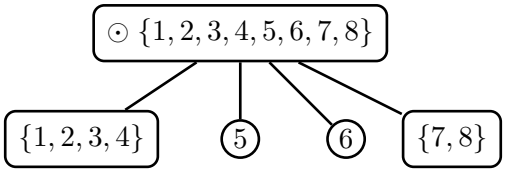
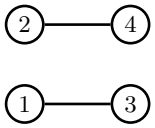


Figure 3.55: The tree after the first decomposition step

Now let us work on the children of $\mathcal{T}$. The leaves 5 and 6 are single nodes and cannot be decomposed further. The first leave contains four nodes, so we set $N = \{1, 2, 3, 4\}$ and look at the induced subgraph $C|N$. From Figure 3.53 we find that this subgraph is:



This subgraph has two connected components, thus we have a parallel compositions with parts having node sets $V_1 = \{1, 3\}$ and $V_2 = \{2, 4\}$. We add these to N and mark N by $\oplus$ as parallel:

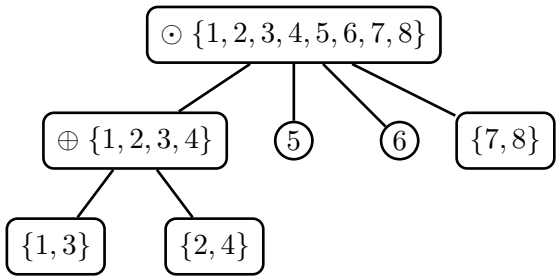


Figure 3.56: The tree after the second decomposition step

Next comes the node with $N = \{7, 8\}$. The subgraph $C|N$ induced by N is (look once more at Figure 3.53) is:

It is disconnected, therefore we mark this tree node with $\oplus$ as parallel composition and attach children consisting of single nodes 7 and 8:

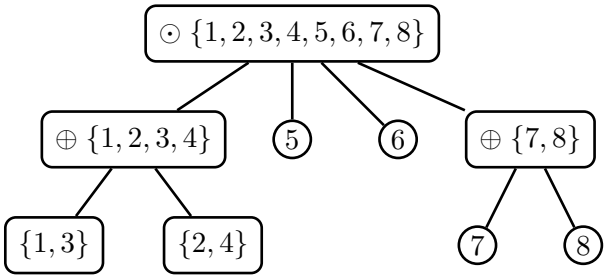
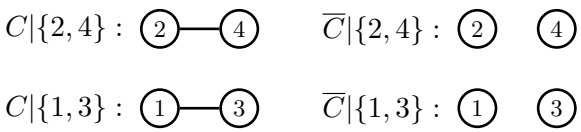


Figure 3.57: The tree after the third decomposition step

Two nodes are left in the tree containing more than one node of the original comparability graph C. Both are connected, so we form their complements:



The complements are disconnected, hence we mark the corresponding nodes of $\mathcal{T}$ as series composition and append the single nodes $1, 3$ and $2, 4$ are leaves. This yields the final decomposition tree $\mathcal{T}$, because all leaves of $\mathcal{T}$ consist of a single node of the original comparability graph C:

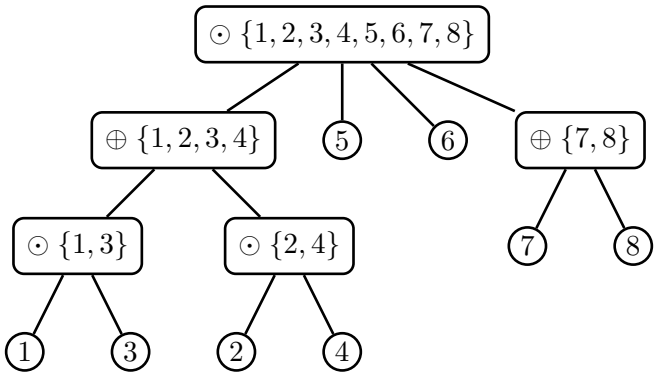


Figure 3.58: The tree after the last decomposition step

▼ Example 3.9

Let us see how the algorithm behaves when fed with a comparability graph which is not series-parallel. Take the forbidden **N**-structure from Figure 3.52. Figure 3.59 shows that both, $C(G)$ and its complement $\overline{C}(G)$ are connected:

Figure 3.59: The comparability graph of a **N**-structure and its complement

Therefore Algorithm 3.4 outputs the message that the **N**-structure is not series-parallel.

▲

3.10.3 Exercises

1. Show by an example that series composition is not commutative.

2. Let A and B denote two sp-graphs and let $L(A)$ and $L(B)$ denote the number of linear extensions of A and B, respectively. By an *elementary* combinatorial argument show that

$$L(A \cdot B) = L(A)L(B), \quad L(A + B) = \binom{|A| + |B|}{|A|} L(A)L(B),$$

where $\binom{n}{k}$ denotes the usual binomial coefficient and $|A|$ and $|B|$ denote the orders of A and B. Extend these formulas to cover also the cases:

$$L(A_1 \cdot A_2 \cdots A_k) \quad \text{and} \quad L(A_1 + A_2 + \ldots + A_k)$$

3. What is the number of linear extensions of the bookcase graph shown in Figure 3.50?

4. The following picture shows you the precedence graph of a scheduling problem with 12 jobs.

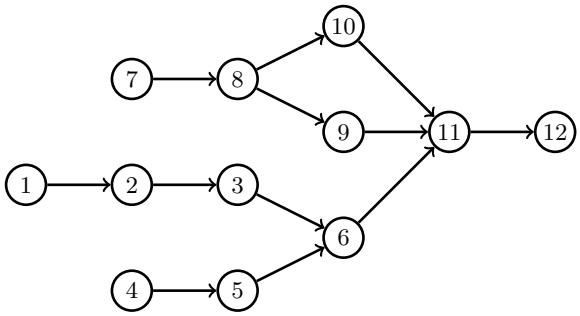


Figure 3.60: A precedence graph for 12 jobs

Is this graph series-parallel? If so, find its decomposition tree and determine the number of linear extensions.

5. Verify by *inspection* that the following precedence graph is not series parallel.

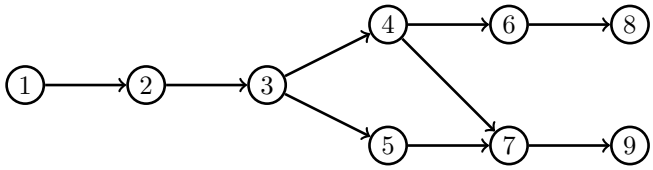


Figure 3.61

3.11 Vertex coloring

We conclude this chapter with another interesting and important concept: the assignment of colors to nodes of a graph.

Let $G = (V, E)$ be an *undirected* graph. By a *proper vertex coloring* we mean the assignment of one color to each node such that no two adjacent nodes are assigned the same color. If exactly k colors are used this assignment is called a k-coloring. The smallest number of colors needed is the *chromatic number* $\chi(G)$. A coloring using only $\chi(G)$ colors is considered optimal as uses a minimum number of colors.

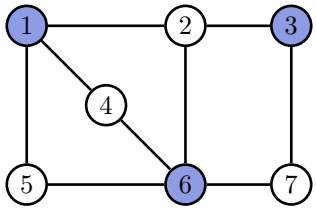


Figure 3.62: A graph G with $\chi(G) = 2$

The graph shown in Figure 3.62 is 2-colorable, moreover it is also bipartite (please verify!). This is no incidence: every bipartite graph can be colored using two colors only. Indeed, this property may be used to *define* bipartite graphs: a graph is bipartite if and only if it is 2-colorable.

Another observation from Figure 3.62: the coloring partitions V into two subsets, the white nodes $V_w = \{2, 4, 5, 7\}$ and the blue nodes $V_b = \{1, 3, 6\}$. By definition of a proper coloring no two nodes in V_b are adjacent, the same is true of V_w. It follows that V_b and V_w are *stable sets*.

More generally, any k-coloring of a graph partitions V into k stable sets, the *color classes* of a graph.

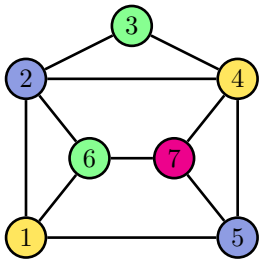
Unfortunately, finding a proper coloring with a minimum number of colors and thus finding the chromatic number of a graph is known to be NP-hard. However, for several special classes of graphs polynomial time algorithms for coloring are known.

No algorithm is required for complete graphs K_n of order n. Since in K_n every node is adjacent to every other node, we always need n colors, thus $\chi(K_n) = n$.

This is an important observation as it gives us a *lower bound* for the chromatic number of a graph. Let $\omega(G)$ be the *clique number* of G, i.e., the size of the largest clique being a subgraph of G (see section 3.3.2), then obviously $\chi(G) \geq \omega(G)$. On the other hand, it can be shown that $\chi(G) \leq 1 + \max_{i \in V}(d_i)$, so there always holds the bounding

$$\omega(G) \leq \chi(G) \leq 1 + \max_{i \in V} d_i \tag{3.7}$$

The upper bound above is not sharp in general, as the graph in Figure 3.63 shows. The upper bound equals $1 + \max d_i = 5$, but this Figure displays a 4-coloring, which is not optimal. Indeed, this graph is 3-colorable.



Can you find a 3-coloring?

Figure 3.63: A graph G with a 4-coloring

Algorithm 3.5 below produces a coloring of a given graph G. It takes as input a *linear ordering* of the nodes and a list of colors. It outputs a coloring.

Algorithm 3.5

Coloring

Input: an undirected graph $G = (V, E)$
a linear node ordering $I = [v_1, v_2, \ldots, v_n]$
a list of n colors $c := [c_1, c_2, \ldots, c_n]$
Output: a coloring C of nodes

for $v_i \in I$ **do**
  begin
    $c_\ell :=$ smallest color not assigned to lower indexed neighbors of v_i
    $C(v_i) := c_\ell$
  end

In general the coloring obtained by Algorithm 3.5 is not optimal, it depends on the linear ordering I. Different orderings will in general yield different colorings which may be far from being optimal. An extreme example are bipartite graphs, see Exercise 3.11.1.1.

However, for special families of graphs Algorithm 3.5 produces an optimal coloring. A particularly interesting family is the set of *interval graphs*.

Recall Section 3.2.3: Let $T_i = (s_i, t_i), i = 1, 2, \ldots, n$ be a set of n intervals on the real line. With each interval T_i we associate a node i. Any two nodes i and j are adjacent, if $T_i \cap T_j \neq \varnothing$.

The linear order yielding an optimal coloring is obtained by sorting the intervals T_i by increasing values of starting times s_i. The chromatic number $\chi(G)$ obtained by Algorithm 3.5 equals the minimum number of classrooms required for a conflict-free schedule of the courses.

Example 3.1 (continued)

Sorting intervals by increasing values of starting times yields the linear ordering:

$$I = [3, 1, 7, 4, 5, 8, 2, 6, 9]$$

For the interval graph displayed in Figure 3.9 Algorithm 3.5 produces the coloring displayed below.

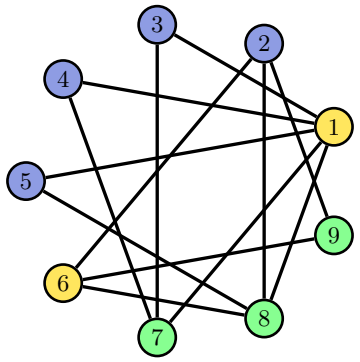


Figure 3.64: An interval graph

Here the first few steps of the algorithm:

$I(1) = 3$, this node has no lower indexed adjacent nodes, so it is assigned color $c_1 = blue$.

$I(2) = 1$. $\Gamma(1) = \{3, 4, 5, 7, 8\}$, but among these there is only node 3 which has lower index. Because this node already has color c_1, node 1 gets color $c_2 = yellow$.

$I(3) = 7$. $\Gamma(7) = \{1, 3, 4\}$, only nodes 3 and 1 have lower index than node 7, They use colors c_1 and c_2, thus node 7 gets color $c_3 = green$.

$I(4) = 4$. $\gamma(4) = \{1, 7\}$. These nodes both have lower index than node 4 and use colors c_2 and c_3. The smallest free color is c_1 which is assigned to node 4.

Continuing in this manner we finally obtain the coloring

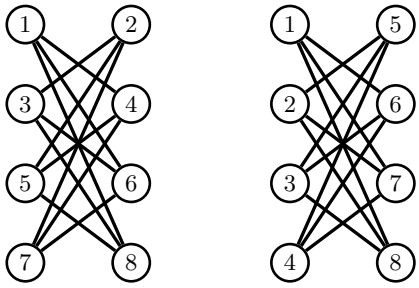
node	1	2	3	4	5	6	7	8	9
color	2	1	1	1	1	2	3	3	3

$c_1 = blue$
$c_2 = yellow$
$c_3 = green$

It follows that the chromatic number of this interval graph G equals 3 which in turn means that we need only 3 classrooms for a conflict-free schedule of the courses.

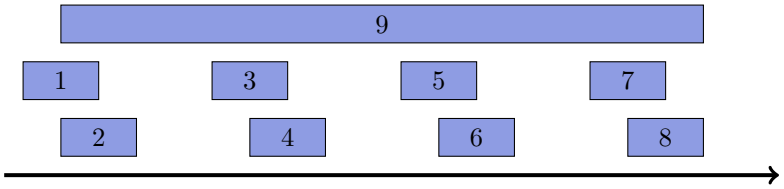
3.11.1 Exercises

1. As remarked earlier, Algorithm 3.5 is quite sensitive with respect to the ordering of nodes. The following Figure presents two bipartite graphs, they differ only in the numbering of nodes.



How many colors are used by Algorithm 3.5 for the left graph, how many for the right graph?

2. Nine courses have to be scheduled in time slots as shown below.



Find the corresponding interval graph. It has a very special shape known as *windmill graph* and is indeed a special case of a *friendship graph* mentioned in Exercise 3.1.4.3. How many classrooms are needed for a conflict-free schedule?

3.12 Bibliographic Notes

There are many excellent introductory textbooks on graph theory, probably one of the best is certainly Chartrand (1977). It gives a smooth introduction to basic concepts without assuming any preknowledge of readers about graphs. The focus of this book lies on modeling with graphs. Many interesting examples, sometimes famous puzzles are presented. Chartrand has coauthored another great book on graph theory, Chartrand and Ping (2012). This book goes much more into depth but it is still an easily accessible introduction and covers also newer developments of the theory. It offers also a lot of information to readers interested in the history of the subject. For beginners I also strongly recommend the nice textbook Hartsfield and Ringel (1994).

Christofides (1975) is one of *the* classical textbooks. Its emphasis is on algorithms and its provides are wide coverage of combinatorial optimization problems including routing problems, facility location and matching. A more recent textbook is Gibbons (1991). A thorough and fine exposition on *depth-first search* is found in Dasgupta, Papadimitriou, and Vazirani (2008).

One of the finest books on graph theory is Berge (1966). Although no more really up to date it introduces readers to most of the topics we have dealt with

in this chapter. Somewhat more theoretically in exposition is Harary (1994). In this book you will find carefully worked out proofs but no graph algorithms. it is also one of the classical textbooks. Algorithms are the strength of Gondran and Minoux (1995), which has also a chapter on algebraic methods, a topic covered only in few books. The books of Cormen et al. (2001) and Sedgewick and Wayne (2011) are written for computer scientists. But these people are also using intensively methods from graph theory. In both books readers find a thorough coverage of various algorithms including, of course, depth-first search and an analysis of their complexity.

Comparability graphs and interval graphs belong to the class of *perfect graphs*. These are those graphs for which the chromatic number χ of each induced subgraph equals its clique number ω. An excellent book on this subject is Golumbic (2004).

3.13 References

[1] C. Berge. *The Theory of Graphs and its Applications*. Methuen, 1966.

[2] Hermann Buer and Rolf H. Möhring. "A Fast Algorithm for the Decomposition of Graphs and Posets". In: *Mathematics of Operations Research* 8.2 (1983), pp. 170–184.

[3] Gary Chartrand. *Introductory Graph Theory*. New York: Dover Publications, 1977.

[4] Gary Chartrand and Zhang Ping. *A First Course in Graph Theory*. New York: Dover Publications, 2012.

[5] Nicos Christofides. *Graph Theory: An Algorithmic Approach (Computer Science and Applied Mathematics)*. Orlando, FL, USA: Academic Press, Inc., 1975.

[6] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001.

[7] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2008.

[8] Robert W. Floyd. "Algorithm 97 (SHORTEST PATH)". In: *Communications of the ACM* 5.6 (1962), p. 345.

[9] Tibor Gallai. "Transitiv orientierbare Graphen". In: *Acta Mathematica Academiae Scientiarum Hungarica* 18.1-2 (1967), pp. 25–66.

[10] Allan Gibbons. *Algorithmic Graph Theory*. Cambridge: Cambridge University Press, 1991.

[11] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Amsterdam: Elsevier, 2004.

[12] Michel Gondran and Michel Minoux. *Graphs and Algorithms*. New York, NY, USA: John Wiley & Sons, Inc., 1995.

[13] Frank Harary. *Graph Theory*. Perseus Books, 1994.

[14] Nora Hartsfield and Gerhard Ringel. *Pearls in Graph Theory*. New York: Dover Publications, 1994.

[15] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. New York, NY, USA: Holt, Rinehart and Winston, 1976.

[16] Rolf H. Möhring. "Computationally Tractable Classes of Ordered Sets". In: *Algorithms and Order*. Ed. by Ivan Rival. Vol. 255. NATO ASI Series. Springer Netherlands, 1989, pp. 105–193.

[17] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[18] Stephen Warshall. "A Theorem on Boolean Matrices". In: *Journal of the ACM* 9.1 (1981), pp. 11–12.