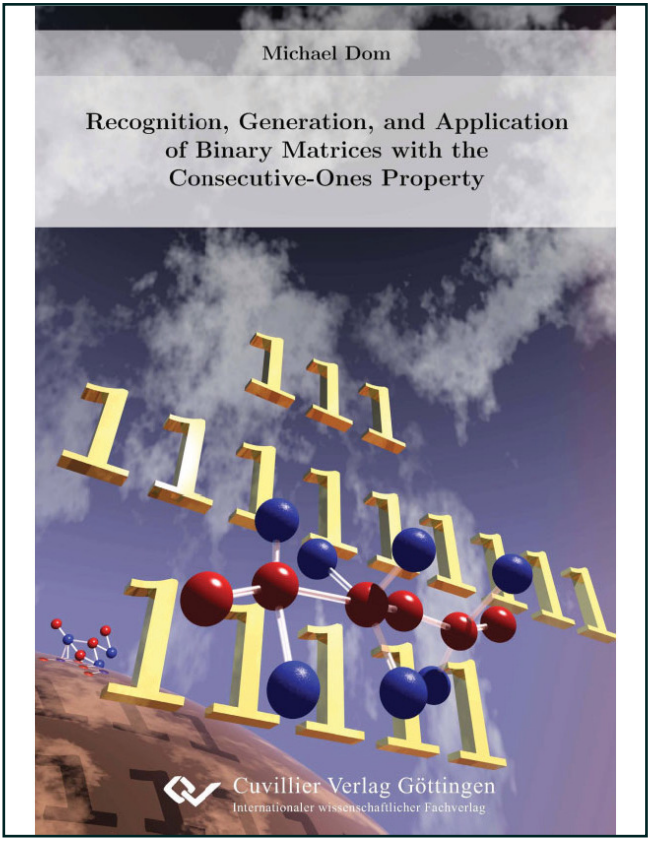
Recognition, Generation, and Application of Binary Matrices with the Consecutive-Ones Property



Chapter 1

Introduction

This thesis deals with combinatorial problems that are all closely related to the consecutive-ones property of binary matrices. Herein, the *consecutive-ones property (C1P)* means that the columns of a binary matrix can be ordered in such a way that the 1s appear consecutively in every row, that is, every row contains at most one block of 1s.

In this first chapter, we start with giving three short examples for the occurrence of the C1P in practical applications. In the remainder of the chapter, we introduce the notation used throughout the thesis and provide a short overview of the basic concepts of algorithmics and computational complexity theory.

1.1 Introductory Examples

This section presents, as a motivation and warm-up, three short examples that illustrate how the C1P can play a role in practical applications. The examples shall also demonstrate how problems from practical applications can be formulated in a compact and precise mathematical form that abstracts from all information that is unnecessary for solving the problem. Since our goal here is to give a rather intuitive understanding, we do not prove the correctness of the approaches described in the three examples.

Physical mapping of DNA. Our first example application has its background in computational biology, where the construction of physical maps for the human DNA was a central issue in the past years [ABH98, AM96, GGKS95, LH03, WR00]. A *physical map* is a map that describes the relative order of *markers* on a *chromosome*. A chromosome is basically a long sequence of DNA, and a marker is a short DNA sequence that appears only once on the chromosome and, therefore, is of special interest. To create a physical map, the chromosome is cut into shorter pieces, which are duplicated and called *clones*. Thereafter, one tests for each of the clones which of the markers appears on it. These tests, however, can only find out whether a marker appears on a clone, but it is not possible to

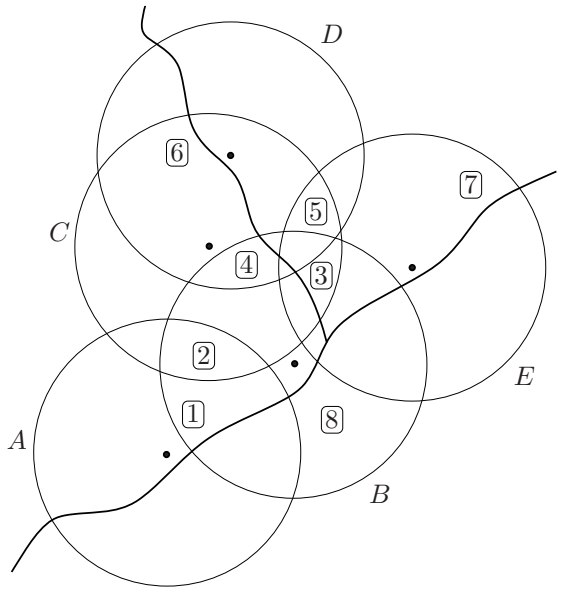
	A	*B*	*C*	*D*	*E*	*F*	*G*
1	0	**1**	0	0	0	**1**	0
2	0	**1**	**1**	**1**	0	0	**1**
3	**1**	**1**	**1**	**1**	0	0	0
4	0	**1**	**1**	**1**	0	0	**1**
5	0	0	**1**	0	**1**	0	0
6	**1**	0	0	**1**	**1**	**1**	0
7	0	**1**	0	**1**	0	**1**	**1**
8	**1**	**1**	0	0	0	**1**	0

	A	*F*	*B*	*G*	*D*	*C*	*E*
1	0	**1**	**1**	0	0	0	0
2	0	0	**1**	**1**	**1**	**1**	0
~~3~~	~~**1**~~	~~0~~	~~**1**~~	~~0~~	~~**1**~~	~~**1**~~	~~0~~
4	0	0	**1**	**1**	**1**	**1**	0
5	0	0	0	0	0	**1**	**1**
~~6~~	~~**1**~~	~~**1**~~	~~0~~	~~0~~	~~**1**~~	~~0~~	~~**1**~~
7	0	**1**	**1**	**1**	**1**	0	0
8	**1**	**1**	**1**	0	0	0	0

Figure 1.1: An application from physical mapping.

determine the order of the markers on the clone. The result is a binary matrix as shown in the left part of Figure 1.1: Every row corresponds to a clone, and every column corresponds to a marker. If a marker appears on a clone, then the corresponding entry of the matrix is 1, otherwise it is 0. Now, the crucial observation for finding the correct order of the markers is that if two markers A and B appear on a clone x, but another marker C does not appear on x, then C cannot lie between A and B on the chromosome. Therefore, to figure out the order of the markers on the chromosome, all one has to do is to order the columns of the matrix in such a way that in every row the 1s appear consecutively. In concrete practical applications, however, the biochemical methods always produce errors such that it is often impossible to order the columns in the resulting matrices as described. One way to deal with these errors is to discard a smallest possible number of clones such that the remaining clones lead to a consistent order of the markers. On the level of binary matrices, this approach means that one has to delete a minimum number of rows such that in the resulting matrix the 1s can be placed consecutively by reordering the columns. This matrix problem is the subject of Chapter 4 of this thesis. The right part of Figure 1.1 shows that in our example we do not have to delete more than two rows.

Placing sender stations in cellular networks. Cellular networks are networks that consist of two types of participants: base stations and client stations. For example, cell phone towers and cell phones form a cellular network. The operator of a such a network can be confronted with the following problem (see also [KRW+05]): To guarantee the network coverage of an area with several settlements, new base stations have to be built. Taking into account the landscape, and, since the stations should be accessible by car, there exist only a certain number of locations that are suitable for planting new base stations. Once a base station is built, it has a certain transmission range. The left part of Figure 1.2 demonstrates the situation: there are eight settlements 1–8 and five suitable locations A–E; each location is drawn as a point together with a cycle denoting the



	A	B	C	D	E
1	**1**	**1**	0	0	0
2	**1**	**1**	**1**	0	0
3	0	**1**	**1**	0	**1**
4	0	**1**	**1**	**1**	0
5	0	0	**1**	**1**	**1**
6	0	0	**1**	**1**	0
7	0	0	0	0	**1**
8	0	**1**	0	0	0
3	0	**1**	**1**	0	**1**
4	0	**1**	**1**	**1**	0

Figure 1.2: The problem of placing base stations in cellular networks.

transmission range of a base station that could be placed at this point. The task of the network operator now is to select a sufficient large number of the locations for building his new base stations there, such that an optimal network coverage of all settlements is obtained. Thereby, two constraints have to be regarded: First, every settlement should lie within the transmission range of at least one base station. Second, there are some client stations that are sensitive to interferences. In our example, let us assume that such client stations exist in the settlements 3 and 4. Therefore, each of the settlements 3 and 4 should lie within the transmission range of at most two base stations—receiving signals from three or more base stations would disturb the client stations there.

The right part of Figure 1.2 shows how to translate this problem into a matrix problem. The corresponding matrix consists of an upper part and a lower part. The upper part has one row for every settlement 1–8, and the lower part has one row for each of the settlements 3 and 4. Furthermore, the matrix has one column for every location A–E. If a settlement lies within the transmission range of a potential base station, then the corresponding entry of the matrix is 1, otherwise it is 0. The problem that now has to be solved on this matrix is: Find some columns that contain at least one 1 from every row of the upper part and at most two 1s from every row of the lower part. In our example, the columns B, D, and E would form a solution for this problem. One can easily verify that building base stations at the corresponding locations yields a network coverage for the settlements as desired.

Note that the matrix resulting from the base station problem typically has a very special structure (see also [MSW05, MW04, RS04]): it is "close" to having

the C1P—in our example, there are only two blocks of 1s in every row. This is due to the facts that the transmission ranges of the base stations are cycles and that the locations A–E of the base stations all are close to some street and, therefore, are arranged in a special way.

Sensor selection in multi-sensor fusion applications. Our third example is adapted from Koushanfar et al. [KSPS02] and deals with minimizing the number of sensors in a multi-sensor fusion application. In this application, one has to classify objects by using a set of sensors. Herein, classifying means to determine for every object that is detected to which of six given object types A–F the object belongs. Every object has two properties—we will call them size and wavelength here—, which can be expressed as a number each. By considering these two properties, every object can uniquely be assigned to one of the six object types. In particular, no two object types have the same combination of these two properties. See parts a) and c) of Figure 1.3 for two examples; the object types are displayed as points in a two-dimensional coordinate system where one coordinate stands for the size and the other for the wavelength of the corresponding object type. The object type C in part a) of Figure 1.3, for example, consists of objects of size 2.7 μm and a wavelength of 420 nm.

Now assume that there are a huge number of different sensors available; each of these sensors has a certain threshold value t and can either detect whether the size of an object is at least t, or whether the wavelength is at least t (in particular, a sensor cannot measure both size *and* wavelength). For several reasons (costs, simplicity,...), as few as possible sensors shall be bought and installed, such that with these few sensors it is possible to identify the type of every object that passes the sensors. For the object types displayed in part a) of Figure 1.3 one needs five sensors; part b) of Figure 1.3 shows one (of several) possibilities how these five sensors can be selected—every vertical or horizontal line in this illustration corresponds to one sensor. These five sensors indeed have the ability to classify every object: If, for example, the sensors report that an object has a size between 200 and 400 μm and a wavelength of at least 500 nm, then this object must be of type D. Part d) of Figure 1.3 shows that three sensors are sufficient for the object types shown in part c) of the figure.

Given a set of object types, how can one find a suitable set of as few as possible sensors? Parts b) and d) of Figure 1.3 illustrate a way how to interpret this task as a geometric problem: The problem of selecting sensors is equivalent to the problem of finding a minimum-size set of axis-parallel lines in the coordinate systems of parts a) and c) of Figure 1.3, such that every pair of points is divided by these lines. In other words, one has to find a minimum-size set of axis-parallel lines that divide the coordinate system into several areas, such that no two points lie within the same area. In order to solve the problem of dividing points with lines, we transform this problem into another geometric problem. To this end, for every pair of points in the coordinate system, we insert a rectangle such that

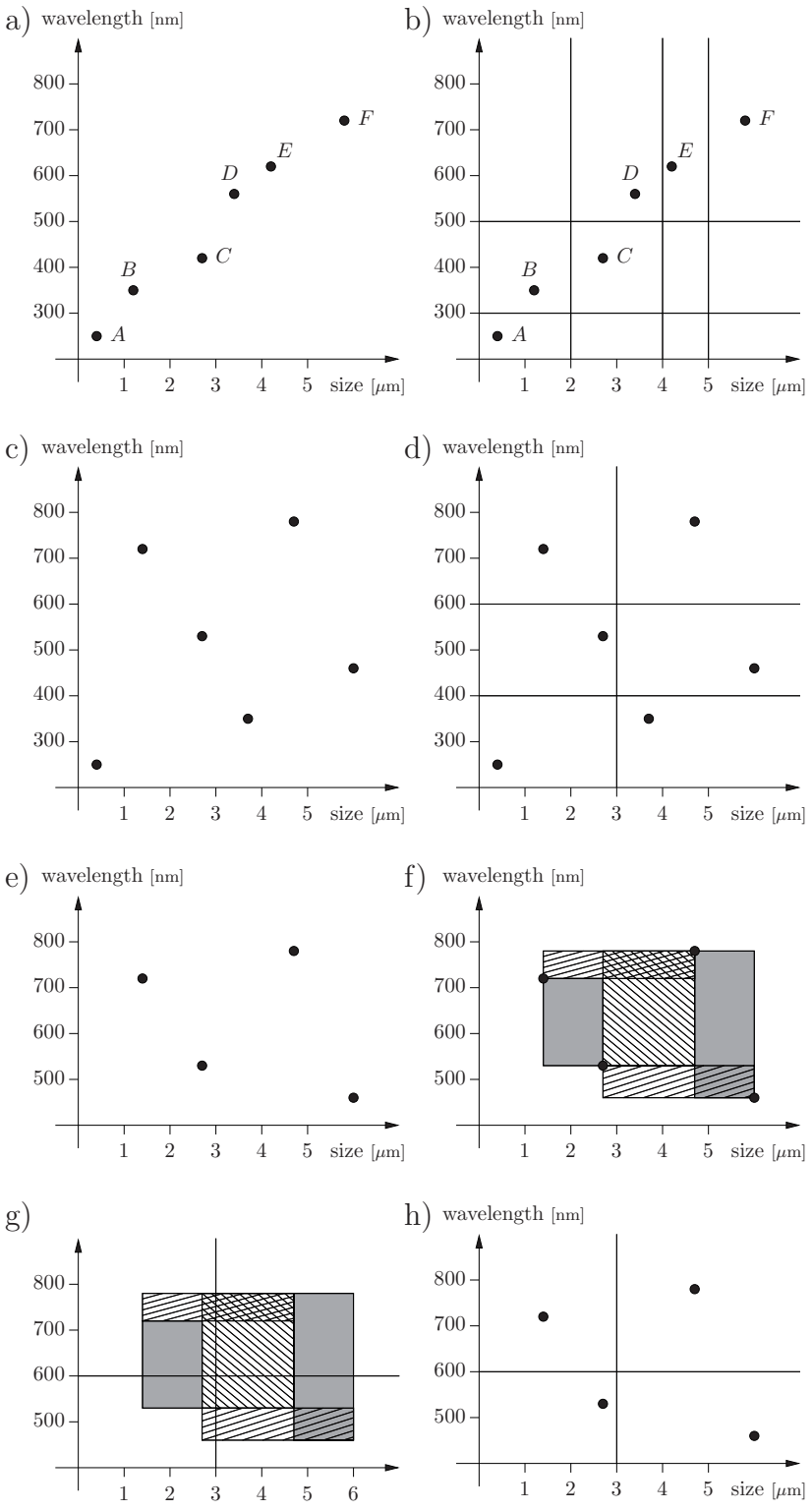


Figure 1.3: Selecting a minimum number of sensors.

the two points lie on two opposite edges of the rectangle [CDKW05].

Part e) of Figure 1.3 shows an example which consists, for the ease of presentation, of only four object types. Part f) of Figure 1.3 shows how to insert the rectangles. Since rectangles that contain other rectangles can be omitted, there are five rectangles. (In the figure, two rectangles are grey-colored, two rectangles are drawn with a diagonal top right to bottom left hatching, and one is drawn with a diagonal top left to bottom right hatching.) The problem that has to be solved now is the following: Find a set of axis-parallel lines such that every rectangle is intersected by at least one of these lines. This problem is known as (2-Dimensional) Rectangle Stabbing, and it is identical to a column selection problem on matrices that have at most two blocks of 1s per row. Part g) of Figure 1.3 shows that all rectangles in the example can be intersected with only two lines, and part h) shows the corresponding solution for dividing the four given points in the coordinate system. Hence, for classifying objects, in this example two sensors would suffice: one sensor with a size threshold of 3 μm and one sensor with a wavelength threshold of 600 nm.

All problems occurring in these three application examples are subject of this thesis. The problem of obtaining the C1P by row or column deletions is addressed in Chapter 4. For solving this problem, we also have to identify those parts of a matrix that conflict with the C1P; in Chapter 3 we provide our results in this direction. Selecting columns from a matrix in order to hit at least one 1-entry from some of the rows but not too many 1-entries from the other rows is the subject of Chapter 5, and in Chapter 6, finally, we consider the (d-Dimensional) Rectangle Stabbing problem.

1.2 Basic Definitions

A set S *properly* contains a set S' if $S' \subseteq S$ and $S \setminus S' \neq \emptyset$; we also say that S' is a *proper subset* of S. A set S is called *minimal* (*maximal*) with respect to a property if no proper subset (no proper superset) of S also has this property. In contrast, a set is called *minimum* (*maximum*) with respect to a property if there exists no set of smaller (greater) cardinality that has the property.

As usual, we often write *iff* instead of "if and only if." With $\log(x)$ we denote the logarithm of x to the base 2. By $\mathbb{N}$, we refer to the set of positive integers. For two integers i, j with $j > 0$, the remainder of the division i by j is denoted by $i \bmod j$; for example, $17 \bmod 5 = 2$. We define $i \operatorname{\underline{mod}} j$ as

$$i \operatorname{\underline{mod}} j := ((i-1) \bmod j) + 1,$$

that is,

$$i \operatorname{\underline{mod}} j = \begin{cases} i \bmod j & \text{if } i \bmod j > 0 \\ j & \text{if } i \bmod j = 0 \end{cases}$$

Moreover, for an integer $n > 0$ we define

$$\mathrm{pred}_n,\ \mathrm{succ}_n : \{1,\ldots,n\} \to \{1,\ldots,n\}$$

as the two functions given by

$$\mathrm{pred}_n(x) := (x-1) \bmod n,\ \mathrm{succ}_n(x) := (x+1) \bmod n,$$

that is,

$$\mathrm{pred}_n(x) = \begin{cases} x-1 & \text{if } x > 1 \\ n & \text{if } x = 1 \end{cases}$$

and

$$\mathrm{succ}_n(x) = \begin{cases} x+1 & \text{if } x < n \\ 1 & \text{if } x = n. \end{cases}$$

The *Big-O-Notation* allows to ignore constants when describing functions: Given two functions $f, g : \mathbb{N} \to \mathbb{R}$, then $f \in O(g)$ if $\exists c > 0\ \exists n_0\ \forall n > n_0 : |f(n)| \leq c \cdot |g(n)|$. One often writes $f = O(g)$ instead of $f \in O(g)$, and if $f \in O(g)$, one often writes $O(g(x))$ to denote the value $f(x)$.

The *O^*-Notation* is a notation similar to the Big-O-Notation; it was introduced by Woeginger [Woe03] for describing running times of algorithms and thereby omitting all polynomials in the input size. We use the following definition: Given two functions $f, g : \mathbb{N}^d \to \mathbb{R}$, then $f \in O^*(g)$ if there exists a polynomial $p : \mathbb{N}^d \to \mathbb{R}$ such that $\exists n_0\ \forall n_1,\ldots,n_d > n_0 : |f(n_1,\ldots,n_d)| \leq |p(n_1,\ldots,n_d) \cdot g(n_1,\ldots,n_d)|$.

For basic introductions to discrete mathematics and algorithmics, we refer to [Ros06, CLRS01].

Graphs. An *(undirected) graph* is a tuple (V, E), where V is a finite set and E is a set of size-two subsets from V. An element from V is called a *vertex*, and an element from E is called an *edge*. For a graph G, we denote with $V(G)$ the set of G's vertices and with $E(G)$ the set of G's edges. In a graph $G = (V, E)$, two vertices v and w are *adjacent* (or *connected by an edge*) if E contains the edge $\{v, w\}$; in this case v and w are *neighbors* of each other. The edge $\{v, w\}$ is *incident* to v and w, and the vertices v, w are the *endpoints* of $\{v, w\}$. The *degree* $\deg(v)$ of a vertex v denotes the number of its neighbors. The *(open) neighborhood* of a vertex v is the set of all neighbors of v and is denoted by $N(v)$. The *closed neighborhood* of v, denoted by $N[v]$, is defined as $N[v] := N(v) \cup \{v\}$. A subgraph of G is a graph $G' := (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. For $V' \subseteq V$, the subgraph of G that is *induced* by V' is the graph $G' = (V', E')$ with $E' = \{\{v, w\} \in E \mid v \in V' \wedge w \in V'\}$; this subgraph is denoted by $G[V']$. *Deleting* a vertex $v \in V$ from a graph $G = (V, E)$ means deleting v from V and deleting every edge from E where v is one of the endpoints.

A *path* is a graph $P = (V, E)$ with vertex set $V = \{v_1,\ldots,v_n\}$ and edge set $E = \{\{v_1, v_2\}, \{v_2, v_3\},\ldots,\{v_{n-2}, v_{n-1}\}, \{v_{n-1}, v_n\}\}$; the vertices v_1 and v_n are

called the *endpoints* of P. A *cycle* is a graph consisting of a path $v_1, \ldots, v_n$ and the additional edge $\{v_n, v_1\}$. Sometimes we describe a path or a cycle by giving only the sequence $v_1, \ldots, v_n$ of its vertices. The *length* of a path or cycle is the number of its edges, that is, the length of a path P equals the number of P's vertices minus 1 and the length of a cycle C equals the number of C's vertices. With P_n we denote a path with n vertices, and with C_n we denote a cycle with n vertices. A *chord* of a cycle is an edge e that is not part of the cycle but connects two vertices of the cycle. A *hole* is an induced cycle of length at least 5, that is, a cycle of length at least 5 where no chords exist. A graph is *chordal* if it contains no induced cycle (that is, no chordless cycle) of length greater than three. Two vertices v_i, v_j in a graph G are called *connected (by a path)* if G contains a path as subgraph whose endpoints are v_i and v_j. A graph is called a *connected graph* if all of its vertices are pairwisely connected by paths. A maximal connected subgraph of a graph G is called a *connected component* of G. A *tree* is a connected graph without cycles; a vertex of degree one in a tree is called a *leaf*. A *rooted* tree is a tree where one vertex is marked as the *root* of the tree. A *clique* is a complete graph, that is, a graph $G = (V, E)$ with $E = \{\{v, w\} \mid v, w \in V \ \wedge \ v \neq w\}$. With K_n we denote a clique with n vertices. An *independent set* is a set of vertices that are not connected by edges.

A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two vertex sets V_1, V_2 such that every edge in E has one endpoint in V_1 and one endpoint in V_2. A bipartite graph $G = (V, E)$ together with a partition of V into V_1 and V_2 is often denoted by a triple (V_1, V_2, E). A hole in a bipartite graph is an induced cycle of length at least 6.

A *directed graph* is a tuple (V, E), where the edges from E are ordered pairs (instead of size-2 sets) of vertices from V. Directed paths and cycles are defined analogously to the undirected case, that is, a *directed path* is a directed graph $P = (V, E)$ with vertex set $V = \{v_1, \ldots, v_n\}$ and edge set $E = \{(v_1, v_2), \ldots, (v_{n-1}, v_n)\}$, and a *directed cycle* is a graph consisting of a path $v_1, \ldots, v_n$ and the additional edge (v_n, v_1).

For a general introduction to graph theory, we refer to [Die05, Wes01]; for more about directed graphs, see [BG02].

Matrices. A *matrix* is a rectangular table of numbers, which are called the *entries* of the matrix. An $m \times n$ *matrix* contains $m \cdot n$ entries, which are arranged in m rows and n columns. The entry in the ith row and jth column of a matrix M is denoted by $m_{i,j}$; moreover, we usually use r_i and c_j to denote the ith row and the jth column, respectively, of a matrix. One can also regard a matrix as a set of columns (or rows) together with an order on this set; the order of the columns (rows) is called the *column ordering* (*row ordering*) of the matrix. Two matrices M and M' are called *isomorphic* if M' is a permutation of the rows and columns of M. In particular, if two matrices consist of the same set of columns (rows) but only differ in their column orderings (row orderings), these matrices

are isomorphic (however, the reverse is not always true, since two isomorphic matrices may consist of differing rows *and* columns). We use the term *line* of a matrix M to denote a row or column of M.

A matrix M' is usually called a *submatrix* of a matrix M if one can select a subset of the rows and columns of M in such a way that deleting all but the selected rows and columns from M results in M'. We extend this notion and call M' also a submatrix of M if we can select a subset of the rows and columns of M in such a way that deleting all but the selected rows and columns results in a matrix that is *isomorphic* to M'. If one can find a submatrix M' of M in this way, we say that M *contains* M' as a submatrix and that M' is *induced* by the selected rows and columns. A matrix M is *M'-free* if M' is not a submatrix of M. Let r_i denote the ith row and let c_j denote the jth column of M, and let M' be the submatrix of M that results from deleting all rows except for $r_{i_1}, \ldots, r_{i_p}$ and all columns except for $c_{j_1}, \ldots, c_{j_q}$ from M. Then, a row r_i of M *belongs* to M', denoted by $r_i \in M'$, if $i \in \{i_1, \ldots, i_p\}$. Analogously, a column c_j of M belongs to M' if $j \in \{j_1, \ldots, j_q\}$. A submatrix of a matrix M is called a *proper* submatrix of M if not all rows or not all columns of M belong to M'.

A matrix whose entries are all from $\{0, 1\}$ is called a *binary matrix* or *0/1-matrix*; a matrix whose entries are all from $\{0, 1, -1\}$ is called a $0/\pm 1$-matrix. A column of M that contains only 0-entries is called a *0-column*. *Complementing* a line ℓ of a matrix means that all 1-entries of ℓ are replaced by 0s and all 0-entries are replaced by 1s.

A *square matrix* is an $m \times n$ matrix with $m = n$; the *main diagonal* of an $n \times n$ square matrix M denotes the entries $m_{1,1}, m_{2,2}, \ldots, m_{n,n}$. A *unit matrix* is a square matrix where the entries of the main diagonal are 1 and all other entries are 0. The *transpose* of an $m \times n$ matrix M, denoted by M^{T}, is the $n \times m$ matrix M' with $m'_{j,i} = m_{i,j}$. A *vector* $\vec{x}$ is an $m \times 1$ matrix, its entries are usually denoted with $x_1, \ldots, x_m$.

The *half adjacency matrix* of a bipartite graph $G = (V_1, V_2, E)$ with $V_1 = \{v_1, \ldots, v_{n_1}\}$ and $V_2 = \{w_1, \ldots, w_{n_2}\}$ is the $n_1 \times n_2$ binary matrix M with $m_{i,j} = 1$ iff $\{v_i, w_j\} \in E$. Every 0/1-matrix M can be interpreted as the half adjacency matrix of a bipartite graph; this graph is called the *representing graph* G_M of M. In other words, for every row and every column of a matrix M, there is a vertex in its representing graph G_M, and for every 1-entry $m_{i,j}$ in M, there is an edge in G_M connecting the vertices corresponding to the ith row and the jth column of M.

1.3 Computational Complexity Theory

The main chapters of this work deal with the question whether there are efficient algorithms for certain combinatorial problems. Herein, "efficient" means that the running time of an algorithm is a slowly growing function in the size of the input. This question, or, more generally, the analysis of the amount of required re-

sources (not only time but also, for example, memory space or bits of information exchanged between several processors) for solving problems is one of the main issues in computational complexity theory and theoretical computer science. When considering a problem (we will define later what exactly we mean with the term "problem"), hence, the task is typically to find either an efficient algorithm for the problem, or, contrariwise, a proof for the non-existence of such an algorithm. Unfortunately, in most cases where we are not able to find an efficient algorithm, there are no methods known how to prove that an efficient algorithm cannot exist. Therefore, we are usually satisfied with comparing such a difficult problem with other problems. To this end, we sort problems into complexity classes, which allows to say that a problem is "at least as difficult as many other problems that are already assumed to be difficult." There are many "hardness predicates" of this kind (for example, NP-hardness, APX-hardness, W[1]-hardness, ...), and the ways they are defined are very similar in most cases: First, define a class $\mathcal{C}$ that contains many problems that are already assumed to be "difficult" (which means that after a long period of research there is still no efficient algorithm known for them). Second, show that if there was an efficient algorithm for one of the "difficult" problems in $\mathcal{C}$, then for *all* problems in $\mathcal{C}$ there would exist efficient algorithms (this affirms the "difficulty" of each of the "difficult" problems). Now, a problem X can be called "$\mathcal{C}$-hard" if one can show that the existence of an efficient algorithm for X would imply the existence of efficient algorithms for *all* problems in $\mathcal{C}$.

In this section, we give a more formal description of the types of problems we are dealing with, and we introduce some of the most important complexity classes and notions of hardness. We restrict ourselves to the resource "time", that is, we only consider the time that is needed by an algorithm; on the one hand, because this is the resource that is studied most extensively in literature, and, on the other hand, because time seems to be the resource most relevant in practice.[1]

1.3.1 "Classical" Complexity Theory

Here, we introduce the main concepts of complexity theory as described by Papadimitriou [Pap94]. We start with considering decision problems and the corresponding complexity classes and will then turn over to other problem types.

Decision problems. The first type of problems to describe are *decision problems*. A decision problem has a (usually infinitive) set of possible inputs, which are called *(problem) instances* and consist of mathematical objects. For every given instance of a particular problem, a question is posed, which asks if the instance has a certain property and which can be answered with *yes* or *no*. This question

[1] If an algorithm needs only a limited amount of time for its computation, then, of course, the memory space it needs is also bounded because for every memory access a certain amount of time has to be spent.

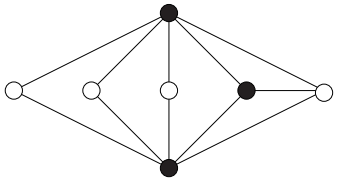


Figure 1.4: Example for VERTEX COVER. The black vertices form a vertex cover of size three: Every edge has at least one black endpoint.

is specific for each decision problem, which means that every decision problem is uniquely defined by the allowed inputs and the question that is asked for each of the inputs. Therefore, we consider a decision problem X as a pair (I_X, q_X) consisting of a set I_X of instances and a question q_X. A problem instance $x \in I_X$ is called a *yes-instance* of X if the answer to the question q_X is *yes* for x, and a *no-instance* otherwise.[2]

As an example for a decision problem, consider the problem VERTEX COVER, which is defined as follows and will be used as a running example throughout this section.

VERTEX COVER
Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
Question: Is there a subset $C \subseteq V$ of at most k vertices such that each edge in E has at least one endpoint in C?

In the case of VERTEX COVER, every problem instance consists of a pair $(G = (V, E), k)$; a vertex set $C \subseteq V$ with the property that every edge in E has at least one endpoint in C is called a *vertex cover* for the graph G (see Figure 1.4).

Running Times and the Turing Machine Model. Problems can be classified by considering the running times that are needed to solve them. To this end, an abstract computer model is used which is called *(deterministic) Turing machine* and whose computational power is identical to that of most relevant real-world computer programs—for example, each problem that can be solved by a Java program can also be solved within a "similar" running time by an adequate Turing machine and vice versa. The running time needed to solve a problem X is measured in terms of how many steps a Turing machine M has to perform to solve the problem. This number of steps is always given as a function $t_M(n)$ in the size n of the input. (For example, the running time of a Turing machine solving VERTEX COVER would be given as a function in the number of vertices

[2] An alternative point of view is the definition of a decision problem as a *language*: A language, in the sense of complexity theory, is a set of binary strings, the so-called *words*; instead of defining a decision problem X as a pair (I_X, q_X), one can define X by a language that contains as words all *yes*-instances of X, encoded as binary strings.

and edges of the input graph.) More exactly, the function $t_M(n)$ always expresses the so-called *worst-case running time* of the Turing machine M, that is, $t_M(n)$ is the maximum running time needed by M, taken over all inputs of size n.

The Complexity Class P**.** In order to constitute a classifying tool, problems that can be solved within similar running times are grouped together and sorted into *complexity classes*, with P and NP being two of the most important of these classes. The class P contains all decision problems that can be solved within a *polynomial running time*. That is, for every problem $X \in$ P there is a Turing machine M_X whose running time $t_{M_X}(n)$ is a polynomial in n. For most practical applications such a polynomial running time means that the problem is solvable within a reasonable amount of time. However, there are a lot of problems of practical relevance for which no polynomial-time algorithms are known.

Nondeterminism. For defining the complexity class NP, a modified computer model called *nondeterministic Turing machine* is introduced: In contrast to deterministic Turing machines, a nondeterministic Turing machine has the freedom to *guess* in every step. By definition, a nondeterministic Turing machine solves a problem if for every *no*-instance it always answers correctly with *no*, and if for every *yes*-instance it answers correctly with *yes* provided that it has made the right guess in every step. One can imagine a nondeterministic Turing machine as a randomized machine that answers correctly for every *no*-instance and that answers correctly with a probability greater than zero for every *yes*-instance. Typically, a nondeterministic Turing machine solves a problem with the following two-phase approach: In the first phase, it guesses a "witness" that proves the correctness of the answer *yes* (for example, in the case of Vertex Cover it guesses a vertex set C of size k). In the second phase, it checks—without guessing—whether the witness is correct (in our example, it checks whether C is indeed a vertex cover) and answers according to the result of the check. A correct witness is usually called a *certificate* or *solution*.

The Complexity Class NP**.** Like in the case of deterministic Turing machines, the running time of a nondeterministic Turing machine M for a problem X is expressed as a function $t_M(n)$ in the input size n; it gives the maximum number of steps the Turing machine can need for solving a problem instance of size n. The class NP is defined as the class of all decision problems X that can be solved nondeterministically within a polynomial running time. That is, for every problem $X \in$ NP there is a nondeterministic Turing machine M_X whose running time $t_{M_X}(n)$ is a polynomial in n. Therefore, the class NP contains exactly those problems where each *yes*-instance has a certificate whose size is polynomial in the input size and whose correctness can be checked deterministically in polynomial time. (Note that the class P contains those problems of NP where a certificate of

each *yes*-instance can not only be checked, but also constructed deterministically in polynomial time.)

Polynomial-Time Reductions, NP-Hardness, and NP-Completeness. Due to the power of guessing, the class NP contains an enormous number of problems (although there exist problems that are even too hard to be solved nondeterministically in polynomial time), many of them of substantial practical relevance. In particular, all problems that belong to P are also contained in NP (which directly follows from the definitions). However, there are a lot of important problems in NP that seem not to be solvable deterministically in polynomial time: Replacing the powerful guessing by deterministic steps—for example, by trying several possibilities—often results in exponential running times (the so-called "combinatorial explosion"), which often makes these problems intractable in practice. In order to unite such "difficult" problems into a class of their own, and to produce evidence for their intractability by showing that either all or none of these problems are solvable deterministically in polynomial time, the concept of *reductions* is introduced: A problem X is *(polynomial-time) reducible* to a problem Y, denoted by $X \leq_{\mathrm{P}} Y$, if there is a function Φ that maps every problem instance x of X to a problem instance $y = \Phi(x)$ of Y such that y is a *yes*-instance of Y iff x is a *yes*-instance of X. Moreover, there must be a polynomial t_Φ such that the time for computing $\Phi(x)$—and, hence, also the size of $\Phi(x)$—does not exceed $t_\Phi(|x|)$. Intuitively speaking, the problem Y is "as least as hard" as the problem X, because a problem instance x of X can be solved in polynomial time by using any polynomial-time algorithm for the problem Y: First, compute the problem instance $y = \Phi(x)$ of Y, and then solve y using the algorithm for Y—the output of this algorithm is the answer for y as well as for x.[3]

The concept of reducing one problem to another can be illustrated by the very simple reduction of the problem Independent Set to Vertex Cover: The problem Independent Set asks, given a graph $G = (V, E)$ and a nonnegative integer k, whether G has an independent set $V' \subseteq V$ of at least k vertices. (An independent set is a set of vertices that are pairwise not connected by edges.) Independent Set can be reduced to Vertex Cover by mapping each problem instance (G, k) of Independent Set to a problem instance $(G, |V| - k)$ of Vertex Cover. If $(G, |V| - k)$ is a *yes*-instance of Vertex Cover, then G has a vertex cover C of size at most $|V| - k$ and, therefore, (G, k) is a *yes*-instance of Independent Set: The vertices not belonging to C are not connected by

[3]Polynomial-time reductions as described here are also called *many-one reductions* or *Karp reductions*. There are also other types of reductions (see [LLS75]), of which the most common is called *Turing reduction*: A problem X is called *Turing reducible* to a problem Y if there is a deterministic Turing machine M that can solve X in polynomial time provided that M has a built-in subroutine—called *oracle*—that solves Y in constant time. For solving X in polynomial time, M can construct polynomially many polynomial-size instances of Y and call the oracle on these instances. The class that contains all problems that are Turing reducible to problems in NP is called P^{NP}.

edges and form an independent set of size at least k (to see this, consider the white vertices in Fig 1.4). If, however, the instance $(G, |V| - k)$ is a *no*-instance of Vertex Cover, then (G, k) is a *no*-instance of Independent Set, because if there was an independent set V' of size at least k, then the vertices in $V \setminus V'$ would form a vertex cover of size at most $|V| - k$.

The definition of polynomial-time reductions directly implies that if $X \leq_{\mathrm{P}} Y$ and $Y \in \mathrm{P}$, then also $X \in \mathrm{P}$ ("the class P is closed under polynomial-time reductions"). The other way round, if $X \leq_{\mathrm{P}} Y$ and if X is one of the "difficult" problems, it is unlikely that Y can be solved in polynomial time (because otherwise the reduction would constitute a polynomial-time algorithm for X). A problem Y is called NP-*hard* if every problem $X \in \mathrm{NP}$ can be reduced to Y. If, in addition, the problem Y itself belongs to NP, the problem Y is called NP-*complete*. For example, Vertex Cover is an NP-complete problem. Note that to show the NP-hardness of a problem Y it suffices to give a reduction from *one* NP-hard problem X to Y, because the composition of two polynomial-time reductions is again a polynomial-time reduction.

NP-complete problems are, by definition, the "most difficult" problems of the class NP, and no algorithms are known that solve these problems efficiently. In fact, it is very unlikely that a polynomial-time algorithm for any NP-hard problem can ever be found, because this would immediately imply that *all* problems in NP (in particular, all NP-complete problems) could be solved in polynomial time, meaning that NP = P. There are thousands of NP-complete problems, and they arise in all areas of life [GJ79, Pap97]; the question whether NP = P is one of the seven "Millennium Prize Problems" named by The Clay Mathematics Institute [Cla09].

Function Problems. So far, we have only considered decision problems. However, in practical applications one often does not only want to know whether a problem instance has a solution (that is, whether it is a *yes*-instance), but one is interested in finding such a solution. For example, in the case of Vertex Cover, the following problem definition could be more useful for many applications.

> **Vertex Cover II**
> **Input:** An undirected graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find a vertex cover C for G that consists of at most k vertices, or report that no such vertex cover exists.

Problems of this kind are called *function problems*. Since the definitions of P and NP do not apply to these problems, there are specific complexity classes for function problems: In particular, the class of problems where the task is to compute a certificate for an instance of a decision problem X from NP is called FNP (Vertex Cover II would be a typical representative for this class). If a problem in FNP can be solved deterministically in polynomial time, that is, a certificate for a given instance can be computed in polynomial time if existing, then it belongs to the class FP.