Introduction to Complexity Theory

Alexander K. Hartmann

hartmann@theorie.physik.uni-goettingen.de

Institut für theoretische Physik, Bunsenstr. 9

37073 Göttingen, Germany

Tel. +49-551-395995, Fax. +49-551-399631

October 26, 1999

Contents

1 Introduction to optimization

Optimization problems [7] are very common in everyday life. For example when driving to work one usually tries to take the *shortest* route. Sometimes additional *constraints* have to be fulfilled, e.g. a bakery should be located along the path or one tries to avoid busy roads when riding by bicycle.

An optimization problem can be described mathematically in the following way: let $\underline{\sigma} = (\sigma_1, \ldots, \sigma_n)$ be a vector with n elements which can take values from a domain X^n: $\sigma_i \in X$, and let H be a real valued function on X^n. The *minimization problem* is:

$$\textit{Find that } \underline{\sigma} \textit{ which minimizes } H(\underline{\sigma})$$

A maximization problem is defined in an analogous way. We will consider only minimization problems, since maximizing a function H is equivalent to minimizing $-H$. Here, only minimization problems are considered where the set X is *countable*. Then the problem is called *combinatorial*.
Constraints may be expressed by additional equations or inequalities which must hold for the solution. An arbitrary value of $\underline{\sigma}$ which fulfills all constraints is called *feasible*. Usually constraints can be expressed more conveniently without giving equations or inequalities. This is shown in the first example.

Example: Traveling Salesman Problem (TSP)

Consider n cities distributed randomly in a plane. Without loss of generality the plane is considered to be the unit square. The minimization task is to find the shortest round-tour through all cities which visits each city exactly once. The tour stops at the city where it started. The problem is described by

$$X = \{1, 2, \ldots, n\} \tag{1}$$

$$H(\underline{\sigma}) = \sum_{i=1}^{n} d(\sigma_i, \sigma_{i+1}) \tag{2}$$

where $d(\sigma_\alpha, \sigma_\beta)$ is the distance between cities σ_α and σ_β and $\sigma_{n+1} \equiv \sigma_1$. The constraint that every city is visited exactly once can be realized by constraining the vector $\underline{\sigma}$ to be a permutation of the sequence $[1, 2, \ldots, n]$.

For the general TSP the cities are not placed in a plane, but an arbitrary distance matrix d is given.

As an example 15 cities in a plane are given in Fig. 1. You can try to find the shortest tour. The solution is presented later on. □

The optimum order of the cities for a TSP depends on their exact positions, i.e. on the random values of the distance matrix d. It is an characteristic of all problems we will encounter here that they are characterized by some random parameters. Each random realization of the parameters is called an *instance* of the problem. In general, if we have a collection of optimization problems of the same (general) type, we will call each single problem an instance of the general problem.
Because the values of the random parameters are fixed for each instance of the TSP, one speaks of *frozen* or *quenched* disorder. For obtaining informations about

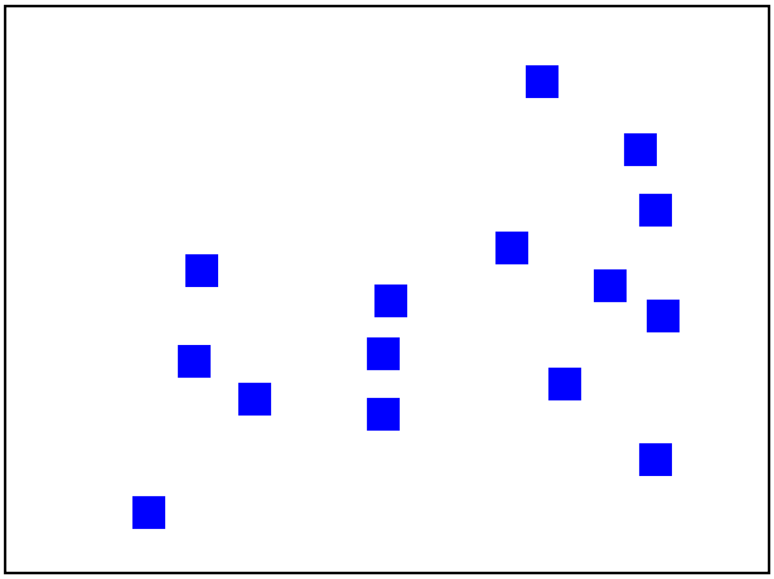


Figure 1: 15 cities in a plane.

the general structure of a problem one has to average measurable quantities, like the length of the shortest tour for the TSP, over the disorder. Later we will see that usually one has to consider many different instances to get reliable results. While the TSP originates from everyday life, in the following example from physics a simple model describing complex magnetic materials is presented.

Example: Ising Spin Glasses

An Ising spin σ_i is a small magnetic moment which can take only two orientations called *up* or *down*, e.g $\sigma_i = \pm 1$. For the simplest model of a magnetic material one assumes that spins are placed on the sites of a simple lattice and that a spin interacts only with its nearest neighbors. In a *ferromagnet* it is energetically favorable for a spin to be in the same orientation as its neighbors, i.e. parallel spins give a negative contribution to the total energy. On the other hand the thermal noise causes different spins to point randomly up or down. For low temperatures T the thermal noise is small, thus the system is *ordered*, i.e. ferromagnetic. For temperatures higher than a critical temperature T_c no long range order exists. One says that a *phase transitions* occurs at T_c. For an introduction to phase transitions, see e.g. [10]

A spin configuration which occurs at $T = 0$ is called a *ground state*. It is just the absolute minimum of the energy $H(\underline{\sigma})$ of the system since no thermal excitations are possible at $T = 0$. They are of high interest because the serve as basis for understanding the low temperature behavior a physical systems. From what was said above, it is clear that in the ground state of a ferromagnet all spins have the same orientation (if quantum mechanical effects are neglected).

A more complicated class of materials are *spin glasses* which exhibit not only ferromagnetic but also *antiferromagnetic* interactions. Neighbors of spins connected by an antiferromagnetic interaction like to be in different orientations. In a spin glass ferromagnetic and antiferromagnetic interactions are distributed randomly within the lattice. Consequently, it is not obvious how ground state configurations look like. I.e. finding the minimum energy is a non-trivial minimization problem. Formally the problem reads as follows:

$$X = \{-1, 1\} \tag{3}$$

$$H(\underline{\sigma}) = \sum_{<i,j>} J_{ij}\sigma_i\sigma_j \tag{4}$$

where J_{ij} denotes the interaction between spin on site i and site j and the sum $\langle i, j\rangle$ runs over all pairs of nearest neighbors. The values of the interactions are chosen according some probability distribution. Each random realization is given by the collection of all interactions $\{J_{ij}\}$. Even the simplest distribution, where $J_{ij} = 1$ or $J_{ij} = -1$ with the same probability, induces are highly non-trivial behavior of the system. Please note that the interaction parameters are frozen variables, while the spins σ_i are free variables which are to be adjusted in such a way that the energy becomes minimized.

Fig. 2 shows a small two-dimensional spin glass and one of its ground states. For this kind of system usually many different ground states for each realization of the disorder are feasible. One says the ground state is *degenerate*. □

These two example are just intended as motivation why dealing with optimization problems is an interesting and fruitful task. The aim of this course is to give an introduction to *methods* how to *solve* these problems, i.e. how to find the minimum. Interestingly, there is no single way to achieve this. For some problems it is very easy while for others it is rather hard. Additionally, within the class of hard or within the class of easy problems, there is no universal method. Mostly even for each kind of problem there are many different ways to obtain an optimum.

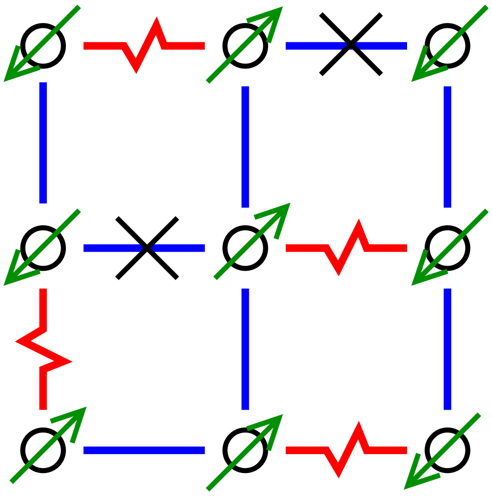


Figure 2: Two-dimensional spin glass. Solid lines represent ferromagnetic interactions while jagged lines represent antiferromagnetic interactions. The small arrows represent the spins, adjusted to a ground-state configuration. For all except two interactions (marked with a cross) the spins are oriented relative to each other in an energetically favorable way. It is not possible to find a state with lower energy (try it!).

Once a problem becomes large, i.e. when the number of variables n is large, it is impossible to find a minimum by hand. Then computers are used to obtain a solution. Only the rapid development in the field of computer science during the last two decades pushed forward the real world application of optimization methods.
Programming languages are used to instruct a computer what to do. Here, no specific language is chosen, since this is only a technical detail. We are more interested in the general way a method works, i.e. in the *algorithm*. In the following chapters we introduce a notation for algorithms, give some examples and explain the most important results about algorithms provided by theoretical computer sciences.

2 Algorithms

Here, we don't want to try to give a precise definition of what an *algorithm* is. We assume that an algorithm is a sequence of statements which is computer readable and has an unambiguous meaning. Each algorithm may have input and output (see Fig. 3) which are well defined objects like sequences of numbers or letters. No user-computer interaction or high-level output like graphic or sound is covered here. Please note that the communication between the main processing units and keyboards or graphic-/sound devices takes place via sequences of numbers as well. Thus, our notion of an algorithm is universal.

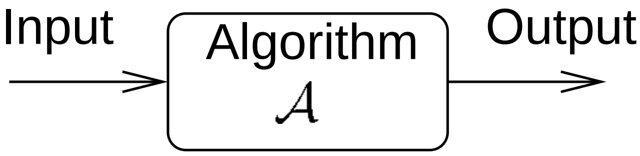


Figure 3: Graphical representation of an algorithm.

Later algorithms for several purposes will be presented. We will concentrate on the main ideas of each method and not on implementational details. Thus, the algorithms will not be presented using a specific programming language. Instead, we will use a *notation* for algorithms called *pidgin Algol*, which resembles modern high-level languages like Algol, Pascal or C. But unlike any conventional programming language, variables of arbitrary type are allowed, e.g. they can represent numbers, strings, lists, sets or graphs. It is not necessary to declare variables and there is no strict syntax.
For the definition of pidgin Algol, we assume that the reader is familiar with at least one high-level language and that the meaning of the terms *variable*, *expression*, *condition* and *label* is clear. A pidgin Algol program is *one* of the following *statements*, please note that by using the *compound* statement, programs can be of arbitrary length:

1. *Assignment*

   variable := expression

   A value is assigned to a variable. Examples: $a := 5 * b + c$, $A := \{a_1, \dots, a_n\}$

   Also more complex and informal structures like
   let z be the first element of the queue Q are allowed.

2. *Condition*

   if *condition* **then** *statement 1*
   else *statement 2*

   The **else** clause is optional. If the condition is true, statement 1 is executed, else statement 2, if it exists.

   Example: **if** money>100 **then** restaurant:=1 **else** restaurant:=0

3. *While-Loop*

   while *condition* **do** *statement*

   The statement is performed as long as the condition is true.

   Example: **while** counter< 200 **do** counter:=counter+1

4. *for-loop*

   for *list* **do** *statement*

   The statement is executed for all parameters in the list. Examples:

   for i:=1, 2, ... ,n **do** sum:=sum+i

   for all elements q of queue Q **do** waits[q]:=waits[q]+1

5. *goto statement*

   a) *label: statement*
   b) **goto** *label*

   When the execution of an algorithm reaches a goto statement the execution is continued at the statement which carries the corresponding label.

6. *compound statement*

   begin
   statement 1;
   statement 2;
   ...
   statement n;
   end

   The compound statement is used to convert a sequence of statements into one statement. It is useful e.g. if a for-loop should be executed for a body of several statements.

   Example:

   for i:=1, 2, ... ,n **do**
   begin
   a:=a+i;
   b:=b+i*i;
   c:=c+i*i*i;
   end

   For briefness, sometimes a compound statement is written as a list of statements in one line, without the **begin** and **end** keywords.

7. *procedures*

   procedure *procedure-name* (*list of parameters*): *statement*
   return *expression*

   A procedure is used to define a new name for one statement or, using a compound statement, for a collection of statements. A procedure can be invoked by writing: *procedure-name* (*arguments*)

   Example:

procedure minimum(x,y)
if x>y **then return** y
else return x

8. *comments*

   comment *text*

   Comments are used to explain parts of an algorithm, i.e. to aid its understanding.

9. *miscellaneous statements*: practically any text which is self-explanatory is allowed. Examples:

   Calculate determinant D of matrix M
   Calculate average waiting time for queue Q

As a first example we present a simple heuristic for the TSP. This method constructs a tour which is quite short, but it does not guarantee to find the optimum. The basic idea is to start at a randomly chosen city. Then iteratively the city which has the shortest distance from the present city, i.e. its *nearest neighbor*, is chosen from the set of cities which have not been visited yet. The array v will be used to indicate which cities belong already to the tour. Please remember that $d(i,j)$ denotes the distance between cities i and j and n is the number of cities.

algorithm TSP-nearest-neighbor($n, \{d(i,j)\}$)
begin
  for $i = 1, 2, \ldots, n$ **do**
    $v[i] := 0$;
  σ_1:= one arbitrarily chosen city;
  $v[\sigma_1] := 1$;
  for $i = 2, 3, \ldots, n$ **do**
  begin
    min:=∞;
    for all unvisited cities j **do**
      if $d(\sigma_{i-1}, j) <$ min **then**
        min:=$d(\sigma_{i-1}, j)$; $\sigma_i := j$;
    $v[\sigma_i] := 1$;
  end
end

Please note that the length of the tour constructed in this way depends on the city where the tour starts and that this city is randomly chosen. This algorithm and many other heuristics for the TSP can be found on the nice TSP web-pages of Stephan Mertens [9]. On these pages different TSP algorithms are implemented using Java-applets. It is possible to run the algorithms step by step and watch the construction of the tour on the screen. In figure 4 the results for one sample of 15 cities are shown. The left part presents a Java-applet which contains an result for the heuristic while in the right part the shortest tour is given.

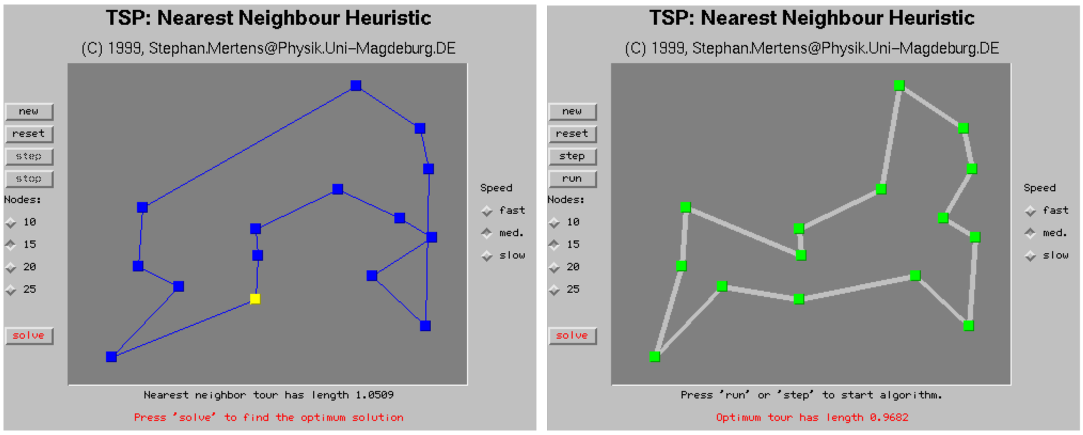


Figure 4: A sample TSP containing 15 cities. The results for the nearest-neighbor heuristic (left) and the exact optimum tour (right) are shown. The starting city for the heuristic is marked by a white square. The nearest neighbor of that city is located above it.

The basis tools and results for the analysis of algorithms were developed in the field of theoretical computer science. For a beginner many of the results may seem unimportant for practical programming purposes. But in fact, for the development of effective algorithms their knowledge is essential. Here, we give the reader just a short glimpse into the field by presenting the most fundamental definitions and results. As an example we will prove in the second part of this section that there are functions on natural number which cannot be programmed on a computer. For that purpose an important technique called *diagonalization* is used. Now we prepare the proof in several steps.

Pidgin Algol is sufficient to present and analyze algorithms. But for a theoretical treatment exact methods and tools are necessary. For this purpose, a precise definition of algorithms is needed. Formal models of computation like the *Turing machine* are used, where everything is stored on a tape via a read/write head. Also very common is the *Random access machine* which is a simple model of real computers consisting of a RAM memory and a central processing unit. It can be shown that all reasonable formal machine models are equivalent. This means that for any program on one model an equivalent program can be written for a different model. For more information the reader is referred e.g. to [3].

The observation that all reasonable machine models are equivalent has led to the *Church's thesis*: "For any algorithm a program can be written on all reasonable machine models." Since the term algorithm cannot be defined exactly it is impossible to prove Church's thesis. Nobody has come up with an algorithm that cannot be transfered to a computer. Hence, it seems reasonable that his thesis is true.

In the following we will concentrate on programs which have just one natural number as input and one natural number as output. This is no restriction because every input/output sequence can be regarded as one long list of bits, i.e. one (possibly large) natural number.

Every program of this kind realizes a *partial* function $f : \mathcal{N} \to \mathcal{N}$ from natural numbers to natural numbers. The term partial means that they may be not defined for every input value, the corresponding program for f will run forever for some input x. If f is not defined for the argument x we write $f(x) =$div.

As a next step towards the proof that there are functions which are not computable, we present a way how to *enumerate* all computable functions. This enumeration works by assigning a code-number to each program. For an precise definition of the assignment, one must utilize a precise machine model like the Touring machine or the random access machine. For our purpose here a simple treatment is sufficient. Thus, we can assume that the programs are written in a high level language like C, but restricted to the case where only one input and one output number (with arbitrary large precision) is allowed. The code-number is assigned to a program in the following way: when the program is stored in memory it is just a long sequence of bits. This is a quite a long natural number, representing the program in a unique way. Now, let f_n be the function which is defined through the text with number n, if the text is a valid program. If text n is not a valid program or if the program has more than one input our output number, then we define $f_n(x) =$div for all $x \in \mathcal{N}$. In total, this procedure assigns a function to *each* number.

All functions which can be programmed on a computer are called *computable*. Please note that for every computable function f there are multiple ways to write a program, thus there are many numbers n with $f_n = f$. No we want to show:

There are functions $f : \mathcal{N} \to \mathcal{N}$ which are not computable

Proof: We define the following function

$$f^*(x) = \begin{cases} 1 & \text{for} \quad f_x(x) = \text{div} \\ \text{div} & \text{else} \end{cases} \tag{5}$$

Evidently, this is a well defined partial function on the natural numbers. The point is that it is different from all computable functions f_n, i.e. f^* itself is not computable:

$$\forall n : f_n(n) \neq f^*(n), \quad \Rightarrow \quad \forall n : f_n \neq f \tag{6}$$

$\square$

The technique applied in the proof above is called *diagonalization*. The reason is that if one tabulates the infinite matrix consisting of the values $f_n(i)$ then the function f^* is different from each f_n on the diagonal $f_n(n)$. The principle used for the construction of f^* is visualized in Fig. 5. The technique of diagonalization is very useful for many proofs occurring not only in the area of theoretical computer science but also in many fields of mathematics. The method was probably introduced by Georg Cantor at the beginning of the century to show that there are more than a countable number of real numbers.

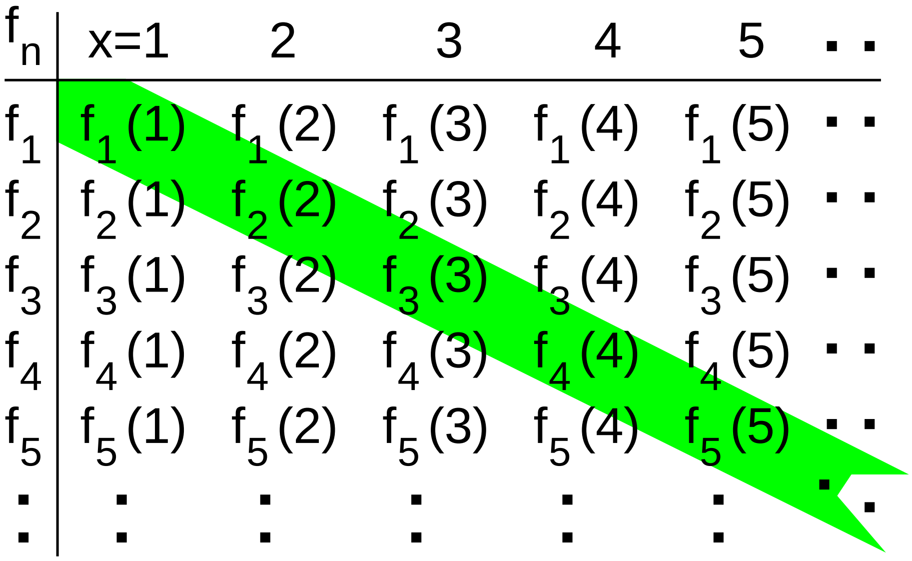


Figure 5: Principle of diagonalization: define a function which differs from all computable functions on the diagonal.

It should be pointed out that the existence of f^* is not a contradiction to Church's thesis since f^* is *not* defined through an algorithm. If someone tries to implement the function f^* from above, he/she must have available an algorithm or test which tells whether a given computer program will halt at some time or whether it will run forever ($f_n(x)$=div). The question whether a given program stops or not is called the *halting problem*. With a similar and related diagonalization argument as we have seen above, it can be shown that there is indeed no solution to this problem. It means that no universal algorithm exists which *decides* for *all* programs whether the program will halt with a given input or run forever. On the other hand, if a test for the halting problem was available it would be easy to implement the function f^* on a computer, i.e. f^* would be computable. Thus, the undecidability of the halting problem follows from the fact that f^* is not computable as well.

In principle, it is always possible to prove for a *given* program whether it will

halt on a given input or not by checking the code and deep thinking. The insolvability of the halting problem just means that there is no systematic way, i.e. no algorithm to construct a proof for *any* given program. Here, as for most proofs in mathematics, the person who carries it out must rely on his/her creativity. But with increasing length of the program usually the proof becomes extremely difficult. It is not surprising that for realistic programs like word processors or databases no such proofs are available. The same is true for the *correctness-problem*: There is no systematic way to prove that a given program works according a given specification. On the other hand, this is fortunate, since otherwise many computer scientists and programmers would be unemployed.
The halting problem is a so called *recognition-problem*: for the question "will Program P_n halt on input x" only the answers "yes" or "no" are possible. In general, we will call an instance (here a program) *yes-instance* if the answer is "yes" for it, otherwise *no-instance*. As we have seen, the halting-problem is not *decidable*, because it is not possible to prove the answer "no" systematically. But if the answer is "yes", i.e. if the program stops, this can always be proven: just take the program P_n, supply input x, run it and wait till it stops. This is the reason why the halting problem at least is *provable*.

3 Time complexity

After we have taken a glimpse at the theory of computability, we will proceed with defining the *time complexity* of an algorithm which describes its speed. We will define under what circumstances we call an algorithm *effective*. The speed of a program can only be determined if it halts on every input. For all optimization problems we will encounter, there are algorithms which stop on all inputs. Consequently, we will restrict ourself to this case.
Almost always the time for executing a program depends on the input. Here, we are interested in the dependence on the *size* $|x|$ of the input x. For example, finding a tour visiting 10 cities takes usually less time than finding a tour which passes through one million cities. The most straightforward way for defining the size of the input is counting the number of bits (without leading zeros). But for most problems a "natural" size is obvious, e.g. the number of cities for the TSP or the number of spins for the spin-glass problem. Sometimes there is more than one characteristic size. E.g. a general TSP is given through several distances between pairs of cities. Then the execution time of a program may depend also on the number of distances, i.e. the number of nonzero entries of the matrix $d(i, j)$.
Usually one is interested not in the actual running time $t(x)$ of an algorithm for a specific implementation on a given computer. Obviously, this depends on the input, the skills of the programmer, the quality of the compiler and the amount of money which is available for buying the computer. Instead, one would like to

have same kind of measure which characterizes the algorithm itself.
As a first step, one takes the longest running time over all inputs of a given length. This is called the *worst case running time* or worst case *time complexity* $T(n)$:

$$T(n) = \max_{x:|x|=n} t(x) \tag{7}$$

Here, the time is measured in some arbitrary units. Which unit is used is not relevant: on a computer B which has exactly twice the speed than computer A a program will consume only half the time. We want to characterize the algorithm itself. Therefore, a good measure must be independent of such constant factors like the speed of a computer. To get rid of these constant factors one tries to determine the *asymptotic* behavior of a program by giving *upper* bounds:
Definition: O/Θ-notation Let T, g be functions from natural numbers to reals.

- We write $T(n) \in O(g(n))$ if there exists a positive constant c with $T(n) \leq cg(n)$ for all n. We say: $T(n)$ is of order at most $g(n)$.
- $T(n) \in \Theta(g(n))$ if there exist two positive constants c_1, c_2 with $c_1 g(n) \leq T(n) \leq c_2 g(n) \quad \forall n$. We say: $T(n)$ is of order $g(n)$

□

Example: For $T(n) = pn^3 + qn^2 + rn$, the cubic term is the fastest growing part. Let $c \equiv p + q + r$. Then $T(n) \leq cn^3 \quad \forall n$, which means $T(n) \in O(n^3)$. Since e.g. $n^4, 2^n$ are growing faster than n^3, we have $T(n) \in O(n^4)$ and $T(n) \in O(2^n)$. Let $c' \equiv \min\{p, q, r\}$. Then $c'n^3 \leq T(n) \leq cn^3$. Hence, $T(n) \in \Theta(n^3)$. This smallest upper bond characterizes $T(n)$ most precisely.
We are interested in obtaining the time-complexity of a given algorithm without actually implementing and running it. The aim is to analyze the algorithm given in pidgin Algol. For this purpose we have to know how long basic operations like assignments, increments and multiplications take. Here, we assume that a machine is available where all basic operations take *one* time-step. This restricts our arithmetic operations to a fixed number of bits, i.e. numbers of arbitrary length can not be computed. If we encounter a problem where numbers of arbitrary precision can occur, we must include the time needed for the arithmetic operations explicitly into the analysis.
As an example, the time-complexity of the TSP heuristic is now investigated, which was presented in the last section. At the beginning of the algorithm a loop is performed which sets all variables $v[i]$ to zero. Each assignment costs a constant amount of time. The loop is performed n times. Thus, the loop is performed in $\Theta(n)$. The random choice of a city and the assignment of $v[\sigma_1]$ take constant time. The rest of the algorithm is a loop over all cities which is performed $n - 1$

times. The loop consists of assignments, comparisons and another loop which runs over all unvisited cities. Therefore, the inner loop is performed $n-1$ times for $i=2$, $n-2$ times for $i=3$, etc. Consequently, the **if**-statement inside that loop is performed $\sum_{i=2}^{n}(n+1-i) = \sum_{i=1}^{(n-1)}(n-i) = n(n-1)/2$ times. Asymptotically this pair of nested loops is the most time-consuming part of the algorithm. Thus, in total the algorithm has a time complexity of $\Theta(n^2)$.
Can the TSP heuristic be considered as being fast? The following table shows the growth of several functions as a function of input size n.

$T(n)$	$T(10)$	$T(100)$	$T(1000)$
n	10	100	1000
$n \log n$	10	200	3000
n^2	10^2	10^4	10^6
n^3	10^3	10^6	10^9
$n^{\log n}$	10	10^4	10^9
2^n	1024	1.3×10^{30}	1.1×10^{301}
$n!$	3.6×10^6	10^{158}	4×10^{2567}

For algorithms which have a time complexity which grows faster than all polynomials, even moderate increases of the system size make the problem impossible to treat. Therefore, we will call an algorithm *effective* if its time complexity is bounded by a polynomial: $T(n) \in O(n^k)$. In practice, values of the exponent up to $k=3$ are considered as suitable. For very large exponents and small system sizes algorithms with exponentially growing time complexity may be more useful. Compare for example an algorithm with $T_1(n) = n^{80}$ and another with $T_2(n) = 2^n$. The running-time of the first algorithm is even for $n=3$ astronomical, while the second one is able to treat at least small input sizes.
The application of the O/Θ-notation neglects constants or lower order terms for the time complexity. Again, in practice an algorithm with running time $T_3(n) = n^3$ may be faster for small input sizes than another with $T_4(n) = 100n^2$. But these kinds of examples are very rare and rather artificial.
In general, finding an algorithm which has a lower time complexity is always more effective than waiting for a computer to come which is ten times faster. Consider two algorithms with time complexities $T_5(n) = n \log n$ and $T_6(n) = n^3$. Let n_5 resp. n_6 be the maximum problem sizes which can be treated within one day of computer time. If a computer is available which is ten times faster, the first algorithm can treat approximately inputs of size $10n_5$ (if n_5 is large) within one day while for the second the maximum input size grows only like $\sqrt{10}n_6$.
To summarize, algorithms which run in polynomial time are considered as being fast. But there are many problems, especially optimization problems, where no polynomial-time algorithm is known. This holds e.g. for the TSP if the exact minimum tour is to be computed. The study of such problems led to the concept

of *NP-completeness*, which is introduced in the next section.

4 NP-completeness

For the moment, we will only consider recognition problems. Please remember that these are problems for which only the answers “yes” or “no” are possible. We already have introduced the halting and the correctness-problem which are not decidable. The following example of a recognition problem, called *SAT* is of more practical interest. In the field of theoretical computer science it is one of the most basic recognition problems. For SAT it was shown first that many other recognition problems can mapped onto it. This will be explained in detail later on. Recently SAT has attracted much attention within the physics community [5].

Example: k-satisfiability (k-SAT)

A *boolean variable* x_i may only take the vales 0 (false) and 1 (true). Here, we consider three *boolean operations*:

- NOT $\overline{}$ (negation): the *clause* $\overline{x_i}$ ("NOT x_i") is true ($\overline{x_i} = 1$), if and only if (iff) x_i is false: $x_i = 0$
- AND $\wedge$ (conjunction): the *clause* $x_i \wedge x_j$ ("x_i AND x_j") is true, iff both variables are true: $x_i = 1$ AND $x_j = 1$.
- OR $\vee$ (*disjunction*): the *clause* $x_i \vee x_j$ ("x_i OR x_j") is true, iff at least one of the variables is true: $x_i = 1$ OR $x_j = 1$

A variable x_i or its negation $\overline{x_i}$ is called a *literal*. Using braces different clauses may be combined to produce complex *boolean formulas*, e.g. $(x_1 \vee \overline{x}_2) \wedge (x_3 \vee x_2)$.

A formula is called *satisfiable*, if there is at least one assignment for the values of its variables such that the whole formula is true. E.g., the formula $(\overline{x_1} \vee x_2) \wedge \overline{x_2}$ is satisfiable, because for $x_1 = 0, x_2 = 0$ it is true. The formula $(\overline{x_1} \vee x_2) \wedge \overline{x_2} \wedge x_1$ is not satisfiable, because $\overline{x_2} \wedge x_1$ implies $x_1 = 1, x_2 = 0$, but then $(\overline{x_1} \vee x_2)$ is false.

For the k-SAT problem, formulas of the following type are considered, called k-CNF (*conjunctive normal form*) formulas: each formula F consists of m clauses C_j combined by the AND operator:

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m \quad . \tag{8}$$

Each clause C_p has k literals l_{pq} combined by the OR-operator:

$$C_p = l_{p1} \vee l_{p2} \vee \ldots \vee l_{pk} \quad . \tag{9}$$

For example $(x_1 \vee x_2) \wedge (\overline{x_4} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_3)$ is a 2-CNF formula, while $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_4})$ is a 3-CNF formula.

The class k-SAT consists of all problems of the form "is F satisfiable?" where F is a k-CNF formula. The question whether an arbitrary formula is satisfiable is an instance of the such defined SAT problem. Please note that every boolean formula can be rewritten as conjunction of clauses each containing only disjunctions and negations. This form is called CNF. □

We have already seen that some recognition problems are undecidable. For these problems it has been proven that no algorithm can be provided to solve it. The k-SAT problem is decidable, i.e. there is a so called *decision-algorithm* which gives for each instance of a k-SAT problem the answer "yes" or "no". The simplest algorithm uses the fact that each formula contains a finite number n of variables. Therefore, there are exactly 2^n different assignments for the values of all variables. To check whether a formula is satisfiable, one can scan through all possible assignments and check whether the formula evaluates to true or to false. If for one of them the formula is true, then it is satisfiable, otherwise not. In the following table all possible assignments for the variables of $(x_2 \vee x_3) \wedge (x_1 \vee \overline{x_3})$ and the results for both clauses and the whole formula is displayed. A table of this kind is called *truth table*.

x_1	x_2	x_3	$x_2 \vee x_3$	$x_1 \vee \overline{x_3}$	$(x_2 \vee x_3) \wedge (x_1 \vee \overline{x_3})$
0	0	0	0	1	0
0	0	1	1	0	0
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Since for each formula up to 2^n assignments have to be tested, this general algorithm has an exponential time complexity (in the number of variables). Since the number of variables is bounded by the number km (m=number of clauses), the algorithm is $O(2^{km})$. But there are special cases, where a faster algorithm exists. Consider for example the 1-SAT class. Here each formula has the form $l_1 \wedge l_2 \wedge \ldots \wedge l_m$, where l_i are literals, i.e. $l_i = x_k$ or $l_i = \overline{x_i}$ for some i. Since each literal has to be true so that the formula is true, the following simple algorithm tests whether a given 1-SAT formula is satisfiable. Its idea is to scan the formula from left to right. Variables are set such that each literal becomes true. If a literal cannot be satisfied because the corresponding variable is already fixed, then the formula is not satisfiable. If on the other hand the end of the formula is reached, it is satisfiable.

algorithm 1-SAT
begin
  initially all x_i are unset;
  for $i = 1, 2 \ldots, m$ **do**
  begin
    let k the number of the variable occurring in literal l_i: $l_i = x_k$ or $l_i = \overline{x_k}$;
    if x_k is unset **then**
      chose x_k such that l_i=true;
    else
      if literal l_i=false **then**
        return(no); **comment** not satisfiable
  end
  return(yes); **comment** satisfiable
end

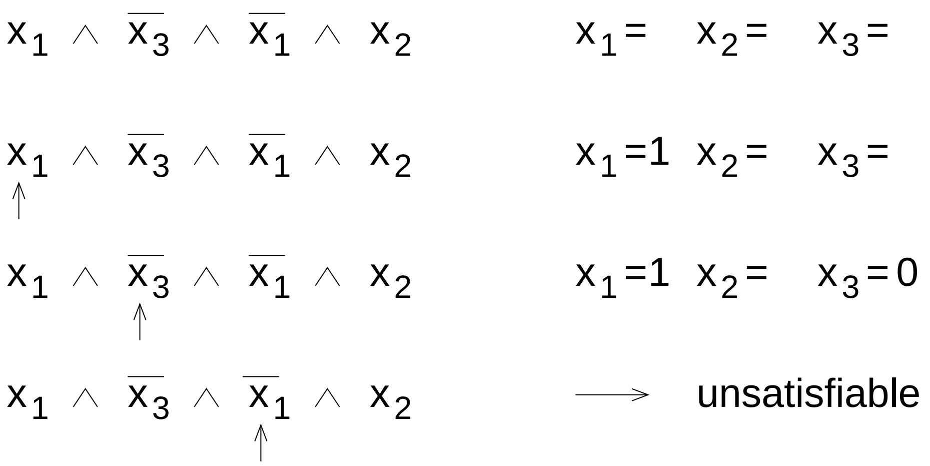
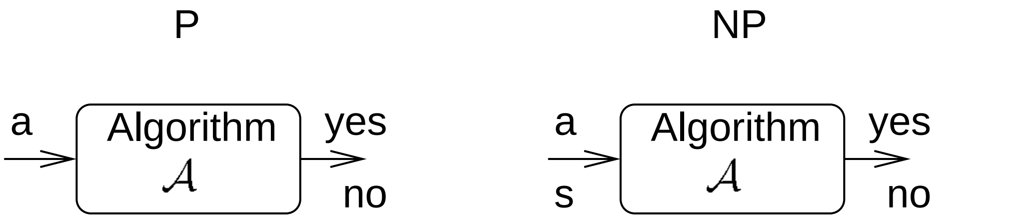


Figure 6: Sample run of algorithm 1-SAT for formula $x_1 \wedge \overline{x_3} \wedge \overline{x_1} \wedge x_2$.

Obviously, the algorithm tests whether a 1-SAT formula is satisfiable or not. Fig 6 shows as an example how the formula $x_1 \wedge \overline{x_3} \wedge \overline{x_1} \wedge x_2$ is processed. In the

left column the formula is displayed and an arrow indicates the literal which is treated. The right column shows the assignments of the variables. The first line shows the initial situation. The first literal ($l_1 = x_1 \to k = 1$) causes $x_1 = 1$ (second line). In the second round ($l_2 = \overline{x_3} \Rightarrow k = 3$) $x_3 = 0$ is set. The variable of the third literal ($l_3 = \overline{x_1} \Rightarrow k = 1$) is set already, but the literal is false. Consequently, the formula is not satisfiable.
The algorithm contains only one loop. The operations inside the loop take constant time. Therefore, the algorithm is $\Theta(m)$, which is clearly faster than $\Theta(2^{km})$. For 2-SAT also a polynomial-time algorithm is known, for more details see [2]. Both 1-SAT and 2-SAT belong to the following class of problems:
Definition: P (polynomial) The class P contains all recognition-problems for which there exists a polynomial-time decision algorithm. □
For 3-SAT problems no polynomial-time algorithm is known which checks satisfiability. On the other hand, up to now there is no proof that 3-SAT$\notin$P! But, since many clever people have failed in finding an effective algorithm, it is very likely that 3-SAT (and k-SAT for $k > 3$) is not decidable in polynomial time.
There is another class of recognition problems A, which now will be defined. For this purpose we use *certificate-checking* (CC) algorithms. These are algorithms $\mathcal{A}$ which get as input instances $a \in$A like decision algorithms and additionally strings $s = s_1 s_2 \ldots s_n$, called *certificates* (made from a suitable alphabet). Like decision algorithms they halt on all inputs (a, s) and return only "yes" or "no". The meaning of the certificate strings will become clear from the following. A new class, called NP, can be described as follows:



$a \in$A $\Leftrightarrow \mathcal{A}(a) =$"yes" $a \in$A $\Leftrightarrow \exists s : \mathcal{A}(a, s) =$"yes"

Figure 7: Classes P and NP.

The difference between P and NP is (see Fig. 7): for a yes-instance of a P problem the decision algorithm answers "yes". For a yes-instance of an NP problem there *exists* at least one certificate-string s such that the CC algorithm answers "yes". I.e. there may be many certificate strings s with $\mathcal{A}(a, s)$="no" even if a is a yes-instance. For a no-instance of a P problem the decision algorithm answers "no", while for a no-instance of a NP problem the CC algorithm answers "no" for *all* possible certificate strings s. As a consequence, P is a subset of NP, since

every decision algorithm can be extended to a certificate-checking algorithm by ignoring the certificate.
The formal definition of NP is as follows:
Definition: NP (nondeterministic polynomial) A recognition-problem A is in NP, if there is a polynomial-time (in $|a|$, $a \in$A) certificate-checking algorithm with the following property:
An instance $a \in$A is a yes-instance if there is at least one certificate s with $\mathcal{A}(a,s)$=yes, for which the length $|s|$ is polynomial in $|a|$ ($\exists z : |s| \leq |a|^z$). □
In fact, the requirement that the length of s is polynomial in $|a|$ is redundant, since the algorithm is allowed to run only a polynomial number of steps. During that time the algorithm can read only a certain number of symbols from s which cannot be larger than the number of steps itself. Nevertheless, the length-requirement on s is included for clarity in the definition.
The concept of certificate-checking seems rather strange at first. It becomes more clear if one takes a look at k-SAT. We show k-SAT $\in$ NP:
Proof: Let $F(x_1, \ldots, x_n)$ be a boolean formula. The suitable certificate s for the k-SAT problem represents just one assignment for all variables of the formula: $s = s_1 s_2 \ldots s_n$, $s_i \in \{0,1\}$. Clearly, the number of variables occurring in a formula is bounded by the length of the formula: $|s| \leq |F|^1$. The certificate-checking algorithm just assigns the values to the variables ($x_i := s_i$) and evaluates the formula. This can be done in linear time by scanning the formula from left to right, similar to the algorithm for 1-SAT. The algorithm answers "yes" if the formula is true and "no" otherwise. If a formula is satisfiable, then, by definition, there is an assignment of the variables, for which the formula F is true. Consequently, then there is a certificate s for which the algorithm answers $\mathcal{A}(F,s)$=yes. □
The name "nondeterministic polynomial" comes from the fact that one can show that a *nondeterministic algorithm* can decide NP problems in polynomial time. A usual algorithm is deterministic, i.e. from a given *state* of the algorithm, which consists of the value of all variables and the program line where the execution is at one moment, the next state follows in a deterministic way. Nondeterministic algorithms are able to choose the next state randomly. Thus, a machine executing nondeterministic algorithms is just a theoretical construct, but cannot be built yet in reality[1]. The definition of NP relies on certificate-checking algorithms. For each CC algorithm an equivalent nondeterministic algorithm can be formulated in the following way. The steps where a CC algorithm reads the certificate can be replaced by the nondeterministic changes of state. An instance is a yes-instance, if there is at least one run of the nondeterministic algorithm which answers "yes" with the instance as input. Thus, both models are equivalent.
As we have stated above, different recognition problems can be mapped onto each other. Since all algorithms are polynomial which we encounter in this context,

[1]Quantum computers can be seen as a realization of nondeterministic algorithms.

only transformations are of interest which can be carried through in polynomial time as well (as a function of the length of an instance). The precise definition of the transformation is as follows:
Definition: Polynomial-time reducible Let A, B two recognition problems. We say A is *polynomial-time* reducible to B (A$\leq_p$B), if there is a polynomial-time algorithm f such that

$$\text{x is yes-instance of A} \quad \Leftrightarrow \quad \text{f(x) is yes-instance of B}$$

□

Fig. 8 shows how a certificate-checking algorithm for B can be transformed to a certificate checking algorithm for A using the polynomial-time transformation f.

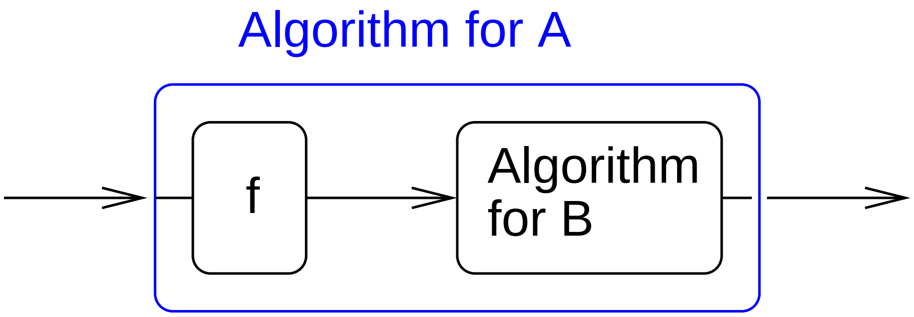


Figure 8: Polynomial-time reducibility: a certificate-checking algorithm for problem A consisting of the transformation f and the algorithm for B.

As example we prove SAT$\leq_p$3-SAT, i.e. every boolean formula F can be written as a 3-CNF formula F^3 such that F^3 is satisfiable iff F is satisfiable. The transformation runs in polynomial time in $|F|$.

Example: Transformation SAT $\rightarrow$ 3-SAT

Let $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ be a boolean formula in CNF, i.e. every clause C_p contains disjunctions of literals l_{pq}. We are now constructing a new formula F^3 by replacing each clause C_p by a sequence of clauses in the following way:

- If C_p has three literals, we do nothing.
- If C_p has more than three literals, say $C_p = l_1 \vee l_2 \vee \ldots \vee l_z$ $(z > 3)$, we introduce $z-3$ new variables $y_1, y_2, \ldots, y_z$ and replace C_p by $z-2$ clauses $(l_1 \vee l_2 \vee y_1) \wedge (\overline{y_1} \vee l_3 \vee y_2) \wedge \ldots (\overline{y_{z-3}} \vee l_{z-1} \vee l_z)$. Now assume that $C_p = 1$, then at least one $l_p = 1$. Now we chose $y_i = 1$ for all $i \leq p-2$ and $y_i = 0$ for all $i > p-2$.

Then all new $z-2$ clauses are true. On the other hand, if the conjunction of the $z-2$ clauses is true, there must be at least one $l_i = 1$. Consequently, if C_p is satisfiable, then the new clauses are satisfiable as well and vice versa.

- At last the case where C_p has less than three literals: If $C_p = l$ we replace it by $l \vee y_1 \vee y_2$ and if $C_p = l_1 \vee l_2$ we replace it by $l_1 \vee l_2 \vee y_1$. In order to keep (un)satisfiability we have to ensure that the new variables y_1, y_2 are always false. We cannot just add $\overline{y_1} \wedge \overline{y_2}$, because every clause has to contain three literals. Therefore, we have to add, with z_1, z_2 two additional new variables: $(\overline{y_1} \vee z_1 \vee z_2) \wedge (\overline{y_1} \vee \overline{z_1} \vee z_2) \wedge (\overline{y_1} \vee z_1 \vee \overline{z_2}) \wedge (\overline{y_1} \vee \overline{z_1} \vee \overline{z_2}) \wedge (\overline{y_2} \vee z_1 \vee z_2) \wedge (\overline{y_2} \vee \overline{z_1} \vee z_2) \wedge (\overline{y_2} \vee z_1 \vee \overline{z_2}) \wedge (\overline{y_2} \vee \overline{z_1} \vee \overline{z_2})$

At the end we have a 3-CNF formula F^3 which is (un)satisfiable iff F is (un)satisfiable. The construction of F^3 works obviously in polynomial time. Consequently, SAT $\leq_p$ 3-SAT. □

There is a special subset of NP problems which reflects in some sense the general attributes of all problems in NP: it is possible to reduce all problems of NP to them. This leads to the following definition:
Definition: NP-completeness The recognition problem A $\in$ NP is called *NP-complete* if all problems B$\in$NP are polynomial reducible to A:

$$\forall \text{ B} \in \text{NP: B} \leq_p \text{A}.$$

□

It can be shown that SAT is NP-complete. The proof is quite technical. It requires an exact machine model for certificate-checking algorithms. The basic idea is: Each problem in NP has a certificate-checking algorithm. For that algorithm a given instance and a given certificate, an equivalent boolean formula is constructed which is only satisfiable if the algorithm answers "yes" given the instance and the certificate. For more details see [2, 3, 7].
Above we have outlined a simple certificate-checking algorithm for SAT. Consequently, using the transformation from the proof that SAT is NP-complete, one can construct a certificate-checking algorithm for every problem in NP. In practice this is never done, since it is always easier to invent such an algorithm for each problem in NP directly.
Since SAT is NP-complete, A$\leq_p$SAT for every problem A$\in$NP. Above we have shown SAT$\leq_p$3-SAT. Since the $\leq_p$-relation is transitive, we obtain A$\leq_p$3-SAT. Consequently, 3-SAT is NP-complete as well. There are many other problems in NP which are NP-complete. For a proof it is sufficient to show A$\leq_p$B for any

other NP-complete problem A, e.g. 3-SAT$\leq_p$B. The list of NP-complete problems is growing permanently. Several of them can be found in [4].
As we have said above, P is a subset of NP. If for one NP-complete problem a polynomial-time decision algorithm will be found one day, then, using the polynomial time reducibility, this algorithm can decide every problem in NP in polynomial time. Consequently, P=NP would hold. But for no problem in NP a polynomial-time decision algorithm has been found so far. On the other hand for no problem in NP there is a proof that no such algorithm exists. Therefore the so called P-NP-problem, whether P$\neq$NP or P=NP, is still unsolved, but P=NP seems to be very unlikely. We can draw everything what we know about the different classes in a figure: NP is a subset of the set of decidable problems. The NP-complete problems are a subset of NP. P is a subset of NP. If we assume P$\neq$NP, then problems in P are not NP-complete (see Fig. 9).

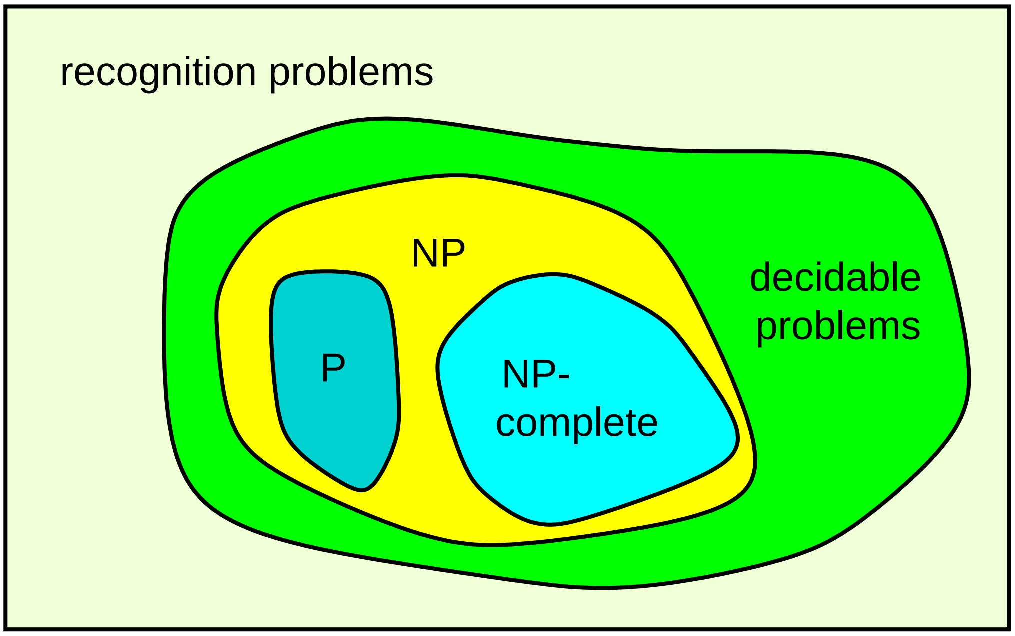


Figure 9: Relation between different classes of recognition problems.

In this section we have concentrated on recognition problems. Optimization problems are no recognition problems since one tries to find a minimum or maximum. This is not a question which can be answered by "yes" or "no". But, every problem $\min H(\underline{\sigma})$ can be transformed into a recognition problem of the form

$$\text{"given a value } K\text{, is there a } \underline{\sigma} \text{ with } H(\underline{\sigma}) \leq K\text{?"}$$

It is very easy to see that the recognition problems for the TSP and the spin glass ground state are in NP: Given an instance of the problem and given a tour/ a spin configuration (the certificates) the length of the tour/ energy of the configuration can be computed in polynomial time. Thus, the question "is $H(\underline{\sigma}) \leq K$" can be answered easily.

If the corresponding recognition problem for an optimization problem is NP-complete, then the optimization problem is called *NP-hard*. In general, these are problems which are harder than problems from NP or which are no recognition problems, but every problem in NP can be reduced to them. This leads to the definition:

Definition: NP-hard Let A be a problem such that every problem in NP is polynomial reducible to A. If A$\notin$NP then A is called *NP-hard*. □

From what we have learned in this section, it is clear that for a NP-hard problem no algorithm is known which finds the optimum in polynomial time. Otherwise the corresponding recognition problem could be solved in polynomial time as well, by just testing whether the so obtained optimum is lower than m or not.

The TSP and the search for a ground state of spin glasses in three dimensions are both NP-hard. Unfortunately this is true for most interesting optimization problems. Therefor, clever programming techniques are needed to implement fast algorithms. Here "fast" means slowly but still exponentially growing. In the next section three of the most basic programming techniques are presented. They are not only very useful for the implementation of optimization methods but for all kinds of algorithms as well.

5 Programming techniques

In this section useful standard programming techniques are presented: recursion, divide-and-conquer, dynamic programming and back-tracking. Since there are many specialized textbooks in this field [1, 8] we demonstrate these fundamental techniques only by presenting simple examples. Furthermore, for efficient data structures, which also play a key role for the development of fast programs, we have to refer to these textbooks. On the Internet the LEDA library is available [6] which contains lots of useful data types and algorithms written in C++.

If a program has to perform many similar tasks this can be expressed as a loop, e.g. with the **while**-statement from pidgin Algol. Sometimes it is more convenient to use the concept of *recursion*, especially if the quantity to be calculated has a recursive definition. One speaks of recursion if an algorithm calls itself. As a simple example we present an algorithm for the calculation of the factorial $n!$ of a natural number $n > 0$. It's recursive definition is given by:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{else} \end{cases} \tag{10}$$

This definition can be translated directly into an algorithm:

algorithm factorial(n)

```
begin
    if n ≤1 then
        return 1;
    else
        return n*factorial(n − 1);
end
```

In the first line it is tested whether $n \leq 1$ instead of testing for $n = 1$. Therefore, it is guaranteed that the algorithm halts on all inputs.
During the execution of factorial(n) a sequence of nested calls of the algorithm is created up to the point where the algorithm is called with argument 1. The call to factorial(n) begins before and is finished after all other calls to factorial(i) with i<n. The hierarchy in Fig. 10 shows the calls for the calculation of factorial(4).

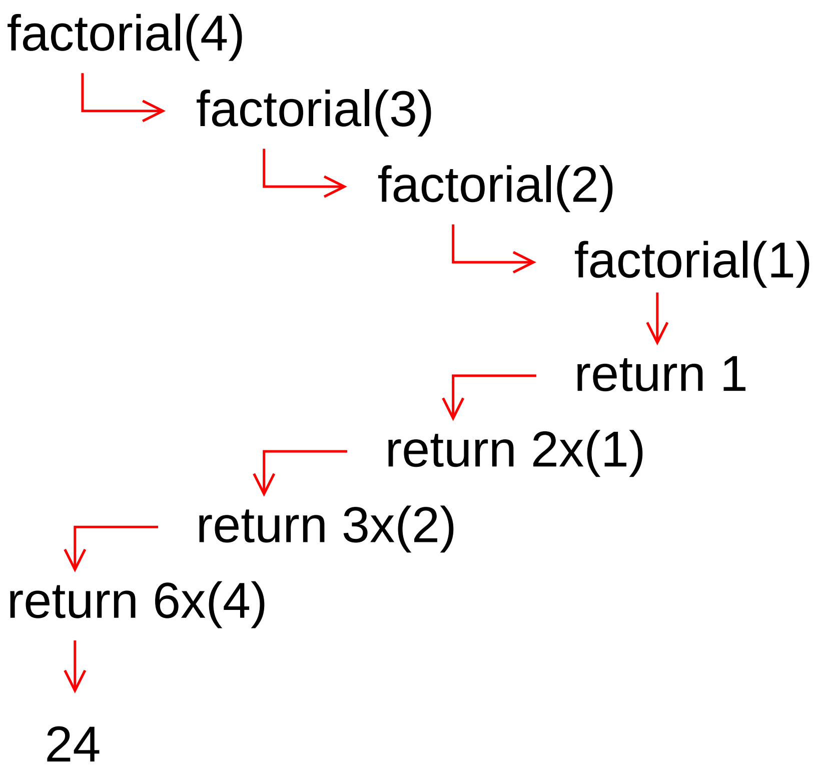


Figure 10: Hierarchy of recursive calls for calculation of factorial(4).

Every recursive algorithm can be rewritten as a sequential algorithm, containing no calls to itself. Instead loops are used. Usually, sequential versions are faster by some constant factor but harder to understand, at least if the algorithm is more complicated than in the present example. The sequential version for the calculation of the factorial reads as follows:

```
algorithm factorial2(n)
begin
    t := 1;   comment this is a counter
    f := 1;   comment here the result is stored
    while t ≤ n do
    begin
        f := f * t;
        t := t + 1;
    end
    return f;
end
```

The sequential factorial algorithm contains one loop which is executed n times. Thus, the algorithm runs in $\Theta(n)$ steps. For the recursive variant the time complexity is not so obvious. For the analysis of recursive algorithms, one has to write down a *recurrence* equation for the execution time. For n=1, the factorial algorithm takes constant time $T(1)$. For $n > 1$ the algorithm takes the time $T(n-1)$ for the execution of factorial($n-1$) plus another constant time for the multiplication. Here and in the following, let C the maximum of all occurring constants. Then, we obtain

$$T(n) = \begin{cases} C & \text{for } n = 1 \\ C + T(n-1) & \text{for } n > 1 \end{cases} \tag{11}$$

One can verify easily that $T(n) = Cn$ is the solution of the recurrence, i.e. both recursive and sequential algorithms have the same asymptotic time complexities. There are many examples, where a recursive algorithm is asymptotically faster than a straight-forward sequential solution, e.g. see [1].

An important area for the application of efficient algorithms are sorting problems: Given n numbers (or strings or complex data structures) A_i $(i = 1, 2, \ldots, n)$ we want to find an permutation B_i of them such that they are sorted in (say) increasing order: $B_i < B_{i+1}$ for all $i < n$. There is a simple recursive algorithm for sorting elements. Please note that the sorting is performed within in the array A_i they were provided in. That means, here not the *values* of the numbers are taken as arguments, i.e. there are no local variables which take the values. Instead the variables (or their memory positions) itself are passed to the following algorithm. Therefore, the algorithm can change the original data. The basic idea of the algorithm is to look for the largest element of the array, store it in the last position, and sort the first $n-1$ elements by a recursive call.

```
algorithm sort(n,{A_1, ..., A_n})
begin
```

```
    if n=1 then
        return;
    max:=1;   comment will contain maximum of all A_i
    pos:=1    comment will contain position of maximum
    t:=2;
    while t<n   comment look for maximum
        if A_t >max then
        begin
            max:=A_t;
            pos:=t;
        end
        exchange maximum and last element;
        sort(n-1,{A_1, ... , A_{n-1}})
end
```

In Fig. 11 it is shown how the algorithm runs with input $(6, \{5, 9, 3, 6, 2, 1\})$. On the left side the recursive sequence of calls is itemized. The maximum element for each call is marked. In the right column the actual state of the array before the next call is displayed.

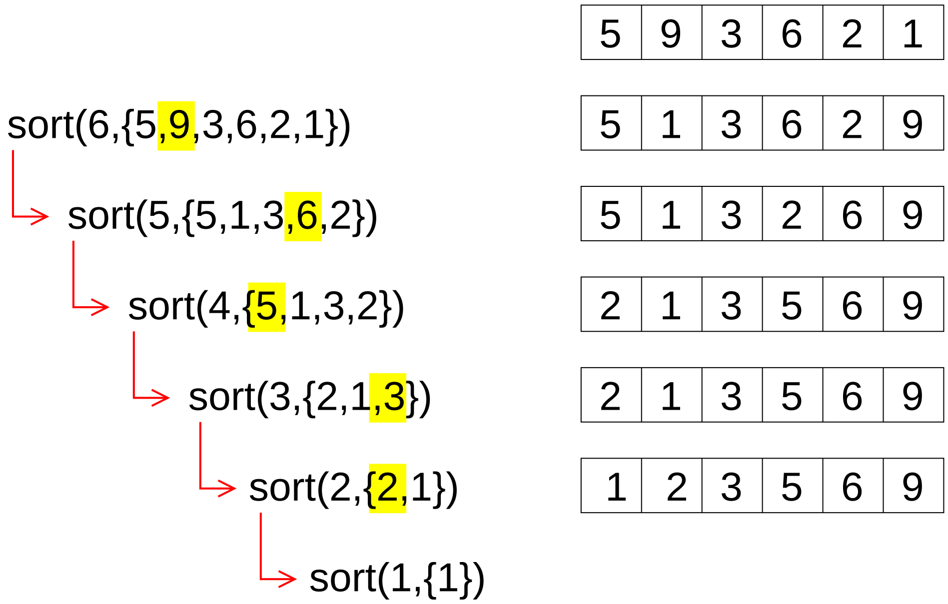


Figure 11: Run of the sorting algorithm with input $(6, \{5, 9, 3, 6, 2, 1\})$.

The algorithm takes linear time for finding the maximum element plus the time for sorting $n-1$ numbers. I.e. for the time complexity $T(n)$ one obtains the following recurrence:

$$T(n) = \begin{cases} C & (n = 1) \\ Cn + T(n-1) & (n > 1) \end{cases} \quad (12)$$

Obviously, the solution of the recurrence is $\Theta(n^2)$. Compared with algorithms for NP-hard problems this is very fast. But there are sorting-algorithms which can do even better. One of them is "mergesort" which relies on the principle of *divide-and-conquer*. The basic idea of this principle is to divide a problem into smaller subproblems, solve the subproblems and then combine the solutions of the subproblems to form the final solution. Usually, recursive calls of the algorithm are applied here as well.
The basic idea of mergesort is to part the set which is to be sorted into two subsets of roughly equal size, sort them recursively and finally *merge* the two sorted sequences into one sorted sequence. The merging is performed by iteratively removing the smallest element of both sequences. Without loss of generality we assume that the number n of items to be sorted is a power of 2. The algorithm reads as follows:

```
algorithm mergesort(n,{A_1,...,A_n})
begin
   if n = 1 then
      return
   for i= 1,2,...,n/2 do     comment divide set
   begin
      B_i := A_i;
      C_i := A_{i+n/2};
   end
   mergesort(n/2,{B_1,...,B_{n/2}});     comment sort subsets
   mergesort(n/2,{C_1,...,C_{n/2}});
   x := 1; y := 1;   comment largest elements of sequences
   for i = 1,...,n do   comment merge sorted subsets
      if x <= n/2 AND B_x < C_y then
         A_i := B_x; x := x + 1;
      else
         A_i := C_y; y := y + 1;
end
```

The hierarchy of recursive calls of mergesort$(4, \{5, 2, 3, 1\})$ is displayed in the upper part of Fig. 12. In the lower part the merging of the sorted subset is shown. For $n = 2^k$ one obtains $k + 1$ layers in the hierarchy of calls.

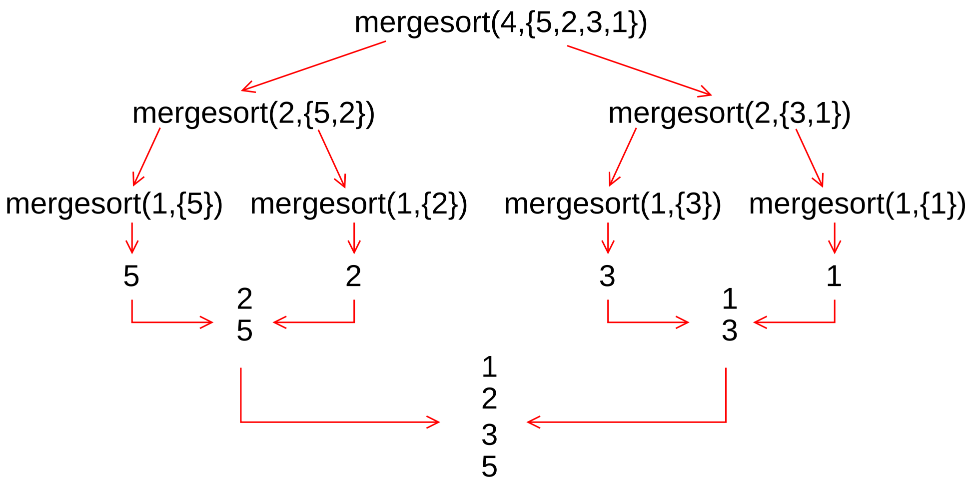


Figure 12: Call of mergesort$(4, \{5, 2, 3, 1\})$.

The division of the sets and the merge-operation takes $\Theta(n)$ time, while each recursive call takes $T(n/2)$. Hence, the recurrence for this algorithms reads:

$$T(n) = \begin{cases} C & (n = 1) \\ Cn + 2T(n/2) & (n > 1) \end{cases} \tag{13}$$

If n is large enough this recurrence can be solved by $T(n) = n \log n$. Consequently, the divide-and-conquer realization of sorting is asymptotically faster than the simple recursive sort-algorithm.

Another problem where the application of divide-and-conquer and recursion seems quite natural is the calculation of *Fibonachi-Numbers* $\text{fib}(n)$. Their definition is as follows:

$$\text{fib}(n) = \begin{cases} 1 & (n = 1) \\ 1 & (n = 2) \\ \text{fib}(n-1) + \text{fib}(n-2) & (n > 2) \end{cases} \tag{14}$$

Thus, e.g. fib(4)= fib(3)+fib(2)= (fib(2)+fib(1))+fib(2)=3, fib(5)= fib(4)+fib(3)= 3+2=5. The functions grows very rapidly: fib(10)= 55, fib(20)= 6765, fib(30)= 83204, fib(40)> 10^8. Let's assume that this definition is translated directly into a recursive algorithm. Then a call to $\text{fib}(n)$ would call $\text{fib}(n-1)$ and $\text{fib}(n-2)$. The recursive call of $\text{fib}(n-1)$ would call *again* $\text{fib}(n-2)$ and $\text{fib}(n-3)$ (which is called from the two calls of $\text{fib}(n-2)$, etc). The total number of calls increases rapidly, even more than $\text{fib}(n)$ increases itself. In Fig. 13 the top of a hierarchy of calls is shown. Obviously, every call to fib with a specific argument is performed very often which is definitely a waste of time.

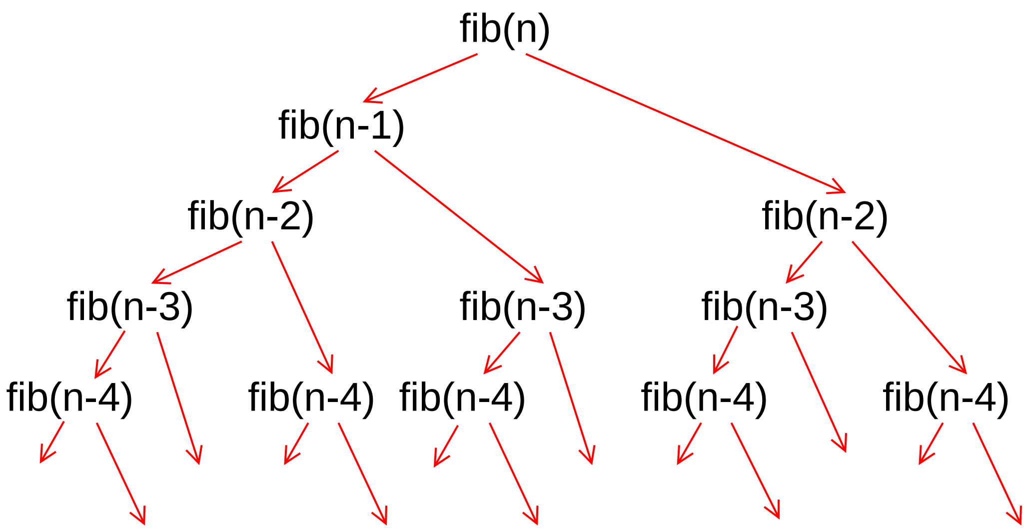


Figure 13: Hierarchy of calls for fib(n).

Instead, one can apply the principle of *dynamic programming*. The basic idea is to start with small problems, solve them and store them for later use. Then one proceeds with larger problems by using divide-and-conquer. If for the solution of a larger problem a smaller one is necessary, it is already available. Therefore, no direct recursive calls are needed. As a consequence, the performance increases drastically. The divide-and-conquer algorithm for the Fibonachi-numbers reads as follows, the array $f[]$ is used to store the results:

algorithm fib-dynamic(n)
begin
  if $n < 3$ **then**
    return 1;
  $f[1] := 1$;
  $f[2] := 1$;
  for $i = 3, 4, \ldots, n$ **do**
    $f[i] := f[i-1] + f[i-2]$
  return $f[n]$;
end

Since the algorithm contains just one loop it runs in $\Theta(n)$ time.
The last basic programming principle which is presented here is *backtracking*. This method is applied when there is no direct way to compute a solution. This is typical for many optimization problems. Remember the simple TSP algorithm which constructs a feasible tour, but which does not guarantee to find the optimum. In order to improve the method, one has to try some (sub-)solutions

and throw them away if they are not good enough. This is the basic principle of back-tracking.
In the following we will present a back-tracking algorithm for the solution of the N-queens problem.

Example: N queens problem

N queens are to be placed on a $N \times N$ chess board in such a way that no queen checks against any other queen.

This means that in each row, in each column and in each diagonal at most one queen is placed. □

A naive solution of the algorithms works by enumerating all possible configurations of N queens and checking for each configuration whether any queen checks against another queen. By restricting the algorithm to place at most one queen per column, there are N^N possible configurations.
The idea of backtracking is to place one queen after the other. One stops placing further queens if a non valid configuration is obtained already at intermediate stage. Then one goes one step back, removes the queen which was placed at the step before, places it elsewhere if possible and continues again.
In the algorithm, we use an array Q_i which stores the position of the queen in column i. If $Q_i = 0$, no queen has been placed in that column. For simplicity, it is assumed that N and the array Q_i are global variables. Initially all Q_i are zero. The algorithm starts in the last column and places a queen. Next a queen is placed in the second last column by a recursive call and so forth. If all columns are filled a valid configuration has been found. If any stage it is not possible to place any further queen in a given column then the backtracking-step is performed: The recursive call finishes, the queen which was set in the recursion-step before is removed. Then it is placed elsewhere and the algorithm proceeds again.

algorithm queens(n)
begin
    if n=0 **then**
        print array $Q_1, \dots, Q_N$; problem solved;
    for $i = 1, \dots, N$ **do**
    begin
        $Q_n := i$
        if Queen n does not check against any other **then**
            queens(n-1);
    end

$Q_i := 0$;
return
end

In Fig. 14 it is shown how the algorithm solves the problem for $N = 4$. It starts with a queen in column 4 and row 1. Then queens(3) is called. The positions where no queen is allowed are marked with a cross. For the third column no queens in row 1 and row 2 are allowed. Thus, a queen is placed in row 3 and queens(2) is called. In the second column it is not possible to place a queen. Hence, the call to queen(2) finishes. The queen in column 3 is placed on row deeper (second line in Fig. 14). Then, by calling queen(2), a queen is placed in row 2. Now, no queen can be placed in the first column. Since there was only one possible position in column 2, the queen is removed and the call finishes. Now, both possible positions for the queen in column 3 have been tried. Therefore, the call for queen(3) finishes as well and we are back at queen(1). Now, the queen in the last column is placed in the second row (third line in Fig. 14). From here it is straight forward to place queens in all columns and the algorithm succeeds.

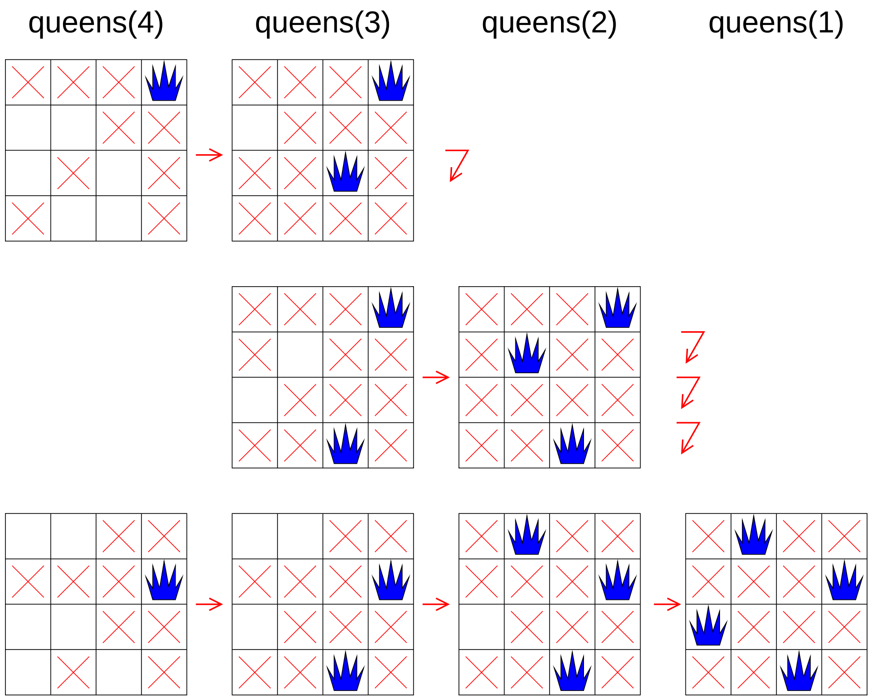


Figure 14: How the algorithm solves the 4-queens problem.

Although this algorithm avois many "dead ends" it has still an exponential time complexity as a function of N. This is quite common to many hard optimization problems. More sophisticated algorithms, which we will encounter later, try to reduce the number of configurations visited even more.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman: The design and analysis of computer algorithms, Addison-Wesley, Readin (MA) 1974

[2] S.A. Cook, J. Assoc. Comput. Mach. **18**, 4-18 (1971)

[3] M.D. Davis, R. Sigal and E.J. Wyuker: Computability, complexity and languages, Academic Press, San Diego 1994

[4] M.R. Garay and D.S. Johnson: Computer and Interactability A Guide to the Theory of NP-Completeness, W.H. Freeman & Company 1979

[5] B. Hayes, Americ. Sci. **85**, 108 (1997)

[6] K. Mehlhorn, S. Näher and C. Uhrig: LEDA, http://www.mpi-sb.mpg.de/LEDA/leda.html

[7] C.H. Papadimitriou and K. Steiglitz: Combinatorial Optimization, Dover Publications, Mineola (NY) 1998

[8] R. Sedgewick: Algorithms in C, Addison-Wesley, Readin (MA) 1990

[9] Stephan Mertens, http://itp.nat.uni-magdeburg.de/~mertens/TSP/index.html (1999)

[10] J.M. Yeomans: Statistical Mechanics of Phase Transitions, Clarendon Press, Oxford 1992