Schirmer, Andreas

A guide to complexity theory in operations research

Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, No. 381

Provided in Cooperation with:
Christian-Albrechts-University of Kiel, Institute of Business Administration

Suggested Citation: Schirmer, Andreas (1995) : A guide to complexity theory in operations research, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, No. 381, Universität Kiel, Institut für Betriebswirtschaftslehre, Kiel

This Version is available at:
https://hdl.handle.net/10419/149837

Manuskripte
aus den
Instituten für Betriebswirtschaftslehre
der Universität Kiel

No. 381

A Guide to Complexity Theory in Operations Research

Schirmer

No. 381

A Guide to Complexity Theory in Operations Research

Schirmer

December 1995

Andreas Schirmer, Institut für Betriebswirtschaftslehre,
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, D-24118 Kiel, Germany

Contents

Figures

Tables

Abstract: It is a well-known fact that there exists an ever increasing number of problems for which, despite the efforts of many inventive and persistent researchers, it seems virtually impossible to find efficient algorithms. In this situation, the theory of computational complexity may provide helpful insight into how probable the existence of such algorithms is at all. Unluckily, some of its concepts can still be found to be used erroneously, if at all. For instance, it is a common misunderstanding that any problem that generalizes an **NP**-complete problem is **NP**-complete or **NP**-hard itself; indeed any such generalization could as well be exponential in the worst case, i.e. solvable with effort exponentially increasing in the size of the instances attempted. In this work we develop the basic concepts of complexity theory. While doing so, we aim at presenting the material in a way that emphasizes the correspondences between the kind of problems considered in operations research and the formal problem classes which are studied in complexity theory.

Keywords: COMPLEXITY THEORY; OPTIMIZATION PROBLEMS; **NP**-COMPLETE; **NP**-HARD; **NP**-EASY; **NP**-EQUIVALENT

1. Introduction

It is a well-known fact that there exists an ever increasing number of problems (just think of the general integer programming problem, the satisfiability problem, or the travelling salesman problem) for which, despite the efforts of many inventive and persistent researchers, it seems virtually impossible to find really efficient, i.e. fast, exact algorithms. In this situation, the theory of computational complexity may provide helpful insight into how probable the existence of such algorithms is at all. Unluckily, when dealing with computationally intractable problems, the terminology of complexity theory can still be found to be used erroneously. Jeffcoat, Bulfin 1993 state in the context of resource-constrained scheduling that the problem of "finding a feasible solution is **NP**-hard" while Laursen 1993 claims that "many types of **NP**-complete problems can [...] be solved to proven optimality" (cf. also Brinkmann, Neumann 1994). Other authors (e.g. Russell 1986; Fadlalla et al. 1994) just assume the intractability of certain problems but refrain from backing up their claim with complexity results. Cooper 1976 observes for scheduling problems that exact methods

> "become computationally impracticable for problems of a realistic size, either because the model grows too large, or because the solution procedure is too lengthy, or both, and heuristics provide the only viable scheduling techniques for large projects."

Badiru 1988 reports certain scheduling problems to be

> "cumbersome because of the combinatorial nature of activity scheduling and the resulting high [...] CPU time requirements"

Formulations like these indicate some insecurity with respect to questions like: What does the term X stand for? What problems can be X? What are the implications of a problem being X? for X $\in$ {**NP**-complete, **NP**-hard, **NP**-easy, **NP**-equivalent}. The reason for this insecurity may lie in the fact that often fundamental concepts are presented in an informal way only; while this approach promotes an intuitive grasp, it is left to the reader to pin down important details. In addition, complexity proofs are often presented in a simplified or shortened manner

where crucial aspects are omitted. In this way, even missing links in the chain of reasoning may go undetected, rendering a seemingly correct proof erroneous (cp. the findings on this topic in Brüggemann 1995). Simplified proof techniques like the prevalent "proof by restriction" have their merit as they facilitate the tedious task of formulating a thorough and complete complexity proof. Nevertheless, their simplicity is sometimes misleading and has to be taken with a grain of salt. For instance, it is a common misunderstanding that any problem generalizing (cf. Definition 14) an **NP**-complete problem is **NP**-complete itself; indeed any such generalization could as well be exponential in the worst case, i.e. solvable with effort exponentially increasing in the size of the instances attempted.

We thus believe it is not superficial to reconsider the basic concepts of complexity theory and to back them up with rigorous mathematical definitions, even though we try to convey the important ideas as well in an informal manner in order to facilitate understanding for the practitioner. While doing so, we also aim at presenting the material in a way that emphasizes the correspondences between the kind of problems considered in operations research and the formal problem classes which are studied in complexity theory. Since most texts on complexity are written from a background in mathematics or computer science, the relationship to practical applications is sometimes lost on the reader. Most of the following material is known, but some of it are usually presented in a different framework such that it is not always clear that it also pertains to the world of operations research.

The remainder of this work is organized along the following lines. Section 2 covers the classical issues of complexity theory. Section 3 addresses several less popular extensions of the theory. The concluding Section 4 summarizes the most important ideas presented.

2. Classical Complexity Theory

2.1. Background and Motivation

The inherent computational tractability of problems constitutes the central subject matter of complexity theory. It is long-known from the fields of mathematics, informatics, and formal logic that there exist problems which are impossible to solve (Church 1936, Turing 1936, Post 1946). It is noteworthy, however, that even among those problems which are guaranteed to be solvable in principle there exist problems which may be regarded as practically unsolvable. Hermes (1978, p. 5) points out that even for a simple problem like calculating the value of n^m for $n \in \mathbb{N}$ and $m \in \mathbb{N}$ there may exist instances whose solving may be in contrast to the laws of nature, e.g. because there is not enough matter in our universe to write down the result or because the human race may not last long enough to complete the computation. Similarly, Brüggemann (1995, p. 78) states that solving large instances of certain problems may take longer than the known universe exists.

In the following, the focus is on solvable problems: Rather than asking *whether* a problem is solvable at all, the question of *how efficient* a problem may be solved is at the heart of complexity analyses. However, before the complexity of a problem and thus the efficiency of problem solving can be defined in a way allowing to build a theory of computational complexity upon it, several abstractions need to be established. We will describe these abstractions in the sequel.

Numerous criteria for measuring the efficiency of algorithms have been proposed (cf. e.g. Hopcroft, Ullman 1979, Chapters 12 and 13), but while "in its broadest sense the notion of efficiency involves all [...] resources needed for executing an algorithm [...], by the 'most efficient' algorithm one normally means the fastest" (Garey, Johnson 1979, p. 5). Indeed, the time required to find a solution is still the most often used criterion when rating the performance of algorithms (Papadimitriou, Steiglitz 1982; Johnson 1983; Schrijver 1986; Nemhauser, Wolsey 1988).

Accordingly, one aims at expressing the relation between an instance of a problem and the time required by an algorithm to solve it. In our understanding, any algorithm will perform one or several arithmetic operations; hence, first of all one has to specify how much time it takes to perform such operations. Here, two different approaches can be adopted: Either each operation is assumed to require an amount of time proportional to the length of the involved operands (*logarithmic time measure*) or, more simple, it is assumed that each operation can be executed in the same amount of time (*unit time measure*). In this contribution, we will stick with the more common unit time measure (similarly e.g. Papadimitriou, Steiglitz 1982, p. 162). Due to its simplicity, it has the merit that the number of operations which an algorithm has to perform also directly specifies the time it will take (*running time*) - up to a constant factor expressing the duration of one operation. Therefore, we will not differentiate between the number of operations an algorithm performs on a given instance and its running time.

With respect to complete algorithms, the time they take is usually a function of the particular instance attempted. Since, however, consideration of the individual instances would obviously render any relation between instance and required time a quite involved measure, one abstracts from the specific instance and considers only a numerical value to describe the instance, viz. its length. This abstraction allows to express the time complexity or *complexity of an algorithm* in terms of solution time as a function of the instance size. Again two possible paths may be followed by considering, for each instance size, either the *worst-case complexity* or the expected or *average-case complexity*. The worst-case time complexity of an algorithm states the time required in the worst case to solve an instance of that size. In contrast, the average-case complexity gives the expected time needed for solving an instance of that size. It may be argued that a time complexity tailored to the average case might be more appropriate for practical purposes because the worst case may occur only rarely in practice. However,

usually it will be all but impossible to identify a probability distribution for the different instances of a given problem; hence, it will be very difficult as well to isolate typical average instances as to weigh the results of several different instances appropriately in order to determine an average complexity. Therefore, we will use the worst-case version and, as to define the *complexity of an algorithm*, we will call an algorithm *polynomial-time* iff its time complexity can be bounded by a polynomial in the length of the instance, and *exponential-time* otherwise.

Note that this definition includes certain nonpolynomial functions like $n^{\log n}$ or n!, which are normally not regarded as exponential and hence are sometimes referred to as "subexponential" (Papadimitriou, Steiglitz 1982, p. 164) or "superpolynomial" (Aarts, Korst 1989, p. 4). We will, however, stick with the above terminology since the exponential function "2^n is the paradigm of nonpolynomial rates of growth" (Papadimitriou, Steiglitz 1982, p. 164) which intends to capture not only algorithms whose time complexity can be expressed by a polynomial in $LNG_e(I)$ but also those algorithms whose time complexity can only be bounded from above by such a polynomial. This includes subexponential functions, but also functions like $n^{2.5}$.

By extension, the *complexity of a problem* is regarded as the complexity of a best possible algorithm solving it, where best possible refers to the lowest order time complexity. Accordingly, a problem is called polynomial if and only if it can be solved by a polynomial algorithm and exponential otherwise, i.e. if no such algorithm can possibly exist. Thus, for any given problem, the complexity of *any* algorithm known to solve it provides an upper bound for its complexity (Brüggemann 1995, p. 80).

Since there exists an abundant number of problems for which consideration of their computational complexity is of interest to researchers and practitioners, the next natural step is to establish a taxonomy of problems classifying them with respect to their complexity. The starting point of this classification is provided by deciding which algorithms are regarded as efficient, i.e. fast, and which are not. In this regard, the distinction between *polynomial-time* and *exponential-time* algorithms (to be introduced in Definition 8) is commonly accepted. By extension, problems which are *only* solvable by exponential-time algorithms are regarded as *intractable*, i.e. not efficiently solvable, while problems which can be solved by a polynomial-time algorithm are considered as *tractable*. (This consideration was first proposed by Cobham 1965 and Edmonds 1965. Cf. also Ullman 1976; Papadimitriou, Steiglitz 1982; Schrijver 1986; Nemhauser, Wolsey 1988.) We will refer to this kind of results which establish whether the time complexity of a problem is polynomial or exponential as *categorial results*. We should emphasize here that in order to establish a problem as polynomial it is not required to exhibit a specific polynomial; it suffices to prove that the complexity could be bounded by some polynomial without specifying it explicitly.

Let us for a moment contemplate why polynomial algorithms are usually considered as efficient and exponential algorithms are not. The first, elementary point we want to make by Table 1 (taken from Papadimitriou, Steiglitz 1982, p. 164), namely that the rate of growth of polynomial functions is substantially higher than that of exponential ones.

Table 1

The Growth of Polynomial and Exponential Functions

Function	Approximate Values		
n	10	100	1000
n log n	33	664	9966
n^3	1000	1,000,000	10^9
$10^6 n^8$	10^{14}	10^{22}	10^{30}
2^n	1024	1.27×10^{30}	1.05×10^{301}
$n^{\log n}$	2099	1.93×10^{13}	7.89×10^{29}
n!	3,628,800	10^{158}	4×10^{2567}

Table 2

Polynomial-Time Algorithms Take Better Advantage of Computation Time

Time Complexity	n = 10	n = 20	n = 30	n = 40	n = 50	n = 60
n	0.00001 second	0.00002 second	0.00003 second	0.0000 second	0.00005 second	0.00006 second
n^2	0.0001 second	0.0004 second	0.0009 second	0.0016 second	0.0025 second	0.0036 second
n^3	0.001 second	0.008 second	0.027 second	0.064 second	0.125 second	0.216 second
n^5	0.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	0.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	0.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Table 2 (taken from Garey, Johnson 1979, p. 7) illustrates that in most cases polynomial algorithms make better use of given computer time because they are - at least up from a certain instance size - faster than exponential ones.

Even more illuminating is the effect that a technological breakthrough improving computer speed would have. Table 3 (taken from Papadimitriou, Steiglitz 1982, p. 165) demonstrates how the size of the largest instance solvable increases when a computer (or an algorithm) with the tenfold speed becomes available: The most striking insight from such a comparison is that for a polynomial function this size multiplies by some factor while for an exponential function increases only by some additive term.

Table 3

Polynomial-Time Algorithms Take Better Advantage of Technology

Function	Size of Instance Solved in One Day	Size of Instance Solved in One Day on a Computer 10 Times Faster
n	10^{12}	10^{13}
$n \log n$	0.948×10^{11}	0.87×10^{12}
n^2	10^6	3.16×10^6
n^3	10^4	2.15×10^4
$10^8 n^4$	10	18
2^n	40	43
10^n	12	13
$n^{\log n}$	79	95
$n!$	14	15

In practice, however, the distinction between polynomial- and exponential-time problems is of only limited value since for a vast number of problems neither a polynomial-time algorithm nor a proof of their intractability has ever been found. Therefore, in classifying problems according to tractability, today one usually starts off by distinguishing the polynomial ones from the *NP-complete* ones. (This idea was published first in Cook 1971 and Karp 1972 while the terminology was first propagated in Knuth 1974a and 1974b.) The significance of the latter problems stems from the fact that there is convincing evidence for their intractability: First, one can show that either *all* **NP**-complete problems or *none* of them are solvable in polynomial time. Second, for not a single **NP**-complete problem has ever a polynomial-time algorithm been discovered, although various extensively studied problems like those mentioned above belong to this class. Hence, it is widely conjectured that **NP**-complete

problems indeed cannot be solved in polynomial time. (However, as yet no proof or disproof for this conjecture is in sight; some results even indicate that such a proof or disproof is impossible before the advent of a whole new part of formal mathematics (cf. Hartmanis, Hopcroft 1976).)

In the following we will develop the basic definitions and concepts needed to introduce these two problem classes, starting with an informal look at algorithms and problems. An important extension of this classification which relates to the so-called *pseudo-polynomial* algorithms will be discussed later on.

2.2. *Problems, Encoding Schemes, and Algorithms*

If not stated otherwise, within this work all algorithms are understood to be deterministic in nature. For our purposes it suffices to imagine a *deterministic algorithm* as performing the steps of a computation in an exactly prescribed sequence, one step at a time. In a strictly theoretical sense this concept is not well-defined until a specific abstract model of computation has been fixed (A detailed introduction to the concept of an algorithm is given in Hermes 1978, while formalizations of deterministic algorithms, using different models of computation, are discussed in Aho, Hopcroft, Ullman 1974; Hermes 1978; Ullman 1976; Garey, Johnson 1979; Hopcroft, Ullman 1979.). However, the tractability of a problem is essentially independent of the particular choice made in this respect: Changing the model of computation within certain reasonable ranges will lead to the same complexity results (for more details cf. Garey, Johnson 1979, pp. 6, 9-11, 19-23; Schrijver 1986, pp. 16-17).

All *problems* have in common that they possess one or more free variables or *parameters*; associated with each parameter is a set, called *domain*, of finite objects. A particular *instance* of a problem is derived by specifying for each parameter an object from the corresponding domain. A problem may thus be seen as specifying a set of instances. In the sequel, four kinds of problems will be treated in detail.

Definition 1 A *search problem* is a pair $\Sigma = (D, S)$ where $D = \{I_1,..., I_n\}$ is a set of finite objects called *instances* and $S = \{S_1,..., S_n\}$ is a set of sets where, for each instance $I_i \in D$, S_i is a set of finite objects called *solutions* for I_i.

An algorithm that returns for each instance $I_i \in D$ "no" iff S_i is empty and one $s \in S_i$ otherwise is said to *solve* Σ. ■

As an example consider the *prime numbers problem* (PNP) of determining the primes in a set of integers, which can be couched in the following way:

(**PNP**) Given an instance I, i.e. a set $J \subseteq \mathbb{N}$, find the set of all primes in J.

It is easy to see that the PNP provides an example of a search problem since $D = \{I_1,..., I_n\}$ where $I_i = J_i \subseteq \mathbb{N}$ $(1 \leq i \leq n)$ and $S = \{S_1,..., S_n\}$ where $S_i = \{j \in J_i \mid j \text{ is prime}\}$ $(1 \leq i \leq n)$. Note that an algorithm solving a search problem is only required to return *one* solution because sometimes several objects may qualify as solutions for a given instance. If e.g. two different tours for an instance of the travelling salesman problem (TSP) share the property of being the shortest, then any one of them would be adequate as solution.

Definition 2 An *optimization problem* (either a minimization or a maximization problem) $\Omega = (D, S)$ is a search problem where each instance $I_i \in D$ is a pair $I_i = (F_i, f_i)$, F_i is a set of finite objects called *feasible solutions*, and f_i is a mapping $f_i: F_i \rightarrow \mathbb{Z}$ called *objective function*. $S_i \subseteq F_i$ consists of those solutions $s \in F_i$ which for all $x \in F_i$ satisfy the inequality $f_i(s) \leq f_i(x)$ in the case of minimization or $f_i(s) \geq f_i(x)$ in the case of maximization.

An algorithm that solves Ω is called *optimization algorithm.* ■

In what follows, optimization problems are understood to be combinatorial, i.e. all numerical values are integral, which is equivalent to allowing arbitrary rational numbers. In order to clarify the relationship between these problem classes and the world of operations research, let us point out that in optimization problems the set F_i of feasible solutions is usually specified by formulating a number of constraints: F_i is then understood to consist of those solutions which satisfy all the constraints. E.g., the well-known *single-mode project scheduling problem* (SMPSP) constitutes an optimization problem since it may be formulated as:

(**SMPSP**) Given an instance I, i.e. $J \in \mathbb{N}$, $R \in \mathbb{N}_0$, $d_j \in \mathbb{N}_0$ $(1 \leq j \leq J)$, $k_{jr} \in \mathbb{N}_0$ $(1 \leq j \leq J;$ $1 \leq r \leq R)$, $K_r \in \mathbb{N}_0$ $(1 \leq r \leq R)$, and $\angle$ partial order on $\{1,...,J\}$, find a feasible schedule for I that has minimal length, i.e. values for the variables x_{jt} $(1 \leq j \leq J;$ $EF_j \leq t \leq LF_j)$ that satisfy the respective constraints and minimize the respective objective function.

Even though many interesting problems belong to the optimization variety, in complexity theory problems are usually phrased as *decision problems.* (Cf. Garey, Johnson 1979. Other authors use the term *recognition problem* to emphasize the relationship between formal languages and decision problems; cf. e.g. Papadimitriou, Steiglitz 1982.) Informally, decision problems may be characterized as asking for a yes-no answer. Their significance lies in the fact that, historically, the first abstract model of an algorithm was the *one-tape Turing machine*, proposed by Turing 1936. This model may only accept or reject an input; hence problems to be solved by it must be couched as decision problems.

Definition 3 A *decision problem* (or short *problem* where no confusion can arise) is a pair $\Pi = (D, Y)$ where D is a set of finite objects called *instances* and $Y \subseteq D$ is a subset of objects called *yes-instances*.

An algorithm that determines for each $I_i \in D$ whether $I_i \in Y_i$ is said to *solve* Π. ■

Often Y is specified by a yes-no-question; Y is then understood to consist of all instances I where the question, applied to I, evaluates to "*yes*".

Closely related to optimization problems are the *feasibility problems* (cp. Nemhauser, Wolsey 1988, p. 127), also referred to as *constraint satisfaction problems* (cp. Schmidt 1989). While the former ask for specific feasible solutions, viz. those which are also optimal, the latter ask whether there exists a feasible solution at all. Formally, the feasibility problems are a subclass of the decision problems, since they require a yes-no answer on a specific question.

Definition 4 A *feasibility problem* $\Phi = (D, Y)$ is a decision problem where each instance $I_i \in D$ is a set F_i of finite objects called *feasible solutions* and the set of yes-instances $Y \subseteq D$ consists of all instances I_i where F_i is nonempty. ■

Again, the correspondence between feasibility problems as defined above and operations research is straightforward, since the *feasibility version* of each optimization problem can easily be constructed by omitting the objective function and - given a particular instance - asking whether there exists a feasible schedule for it. E.g. the feasibility problem (SMPSP^{feas}) associated with the SMPSP may be couched as:

(SMPSP^{feas}) Given an instance I of the SMPSP, does there exist a feasible schedule for I?

The mutual relationships between the four kinds of problems defined above are summarized in Figure 1.

Figure 1

Different Kinds of Problems in Complexity Theory and Operations Research

Optimization Problems	$\subset$	Search Problems
$\cup$		$\cup$
Feasibility Problems	$\subset$	Decision Problems
("OR / MS" Terminology)		("Complexity Theory" Terminology)

Before an algorithm can actually work on an instance, the instance must be translated into a comprehensible and workable form. If e.g. the algorithm is to be run on a computer the instance has to be brought into a computer-readable form; if the algorithm is to be executed

with pencil and paper the instance must be written down in a prescribed manner by representing numbers in some arithmetic system such as the binary or the decimal system. This process is referred to as *encoding*. It is assumed that for any encoding function there exists a polynomial algorithm able to decode the original instance from its encoded version since otherwise the encoding would fail to translate all relevant information contained in the instance.

Definition 5 Let Λ be an arbitrary problem and Γ a finite alphabet. Then an *encoding scheme* (short: *encoding*) is a function $e: D \rightarrow \Gamma^*; I \rightarrow e(I)$ which maps any instance I to a finite string $e(I)$, called *input*, over the alphabet Γ, and for which the inverse function e^{-1} can be computed by a polynomial algorithm.

The *size* (or *length*) of an instance I under an encoding scheme e is a function $LNG: D \rightarrow \mathbb{N}; I \rightarrow LNG_e(I)$ which maps any instance I to an integer reflecting the length of the encoded instance, i.e. the number of symbols necessary to represent I under encoding e.

The *magnitude* of an instance I under an encoding scheme e is a function $MAX_e: D \rightarrow \mathbb{N}; I \rightarrow MAX_e(I)$ which maps any instance I to an integer reflecting the magnitude of the largest numerical value occurring in I under encoding e, and to 0 if no such value occurs in I. ■

For the sake of simplicity we assume that all numerical values are encoded into the same number of digits. This approach can be motivated by computers' internal representation of numbers where each number is stored in one register and all registers have the same size or length. Consequently, for each arithmetic system with base B the number of digits required to represent a parameter is at most $\lceil \log_B MAX_e(I) + 1 \rceil$ because if the largest number is B^k then $k + 1$ digits are necessary to encode it (Brüggemann 1995, p. 72. Note that the term "+ 1" is not superficial even though some authors omit it (cf. Papadimitriou, Steiglitz 1982, p. 160) since in a B-nary arithmetic system k digits can at most encode $B^k - 1$).

We will refer to any encoding as *standard encoding* as long as it obeys the following restrictions: Values of numerical parameters are encoded into their B-nary representation where $B > 1$. (Doing so, all integers as well as certain rationals can be represented exactly while all other numbers can be approximated with arbitrary exactness.) Vectors and matrices are encoded by sequentially listing their encoded elements. Objects (such as cities in the TSP or activities in the SMPSP) are numbered and their identifying numbers encoded as described above. Sets are encoded by sequentially listing their encoded elements. Binary relations on a set of objects (such as precedence orders) are encoded into lists of adjacent pairs of objects where the objects are encoded as above. (Moderately different encodings can be found in Garey, Johnson 1979, pp. 19-23; Papadimitriou, Steiglitz 1982, pp. 159-161.) Even though this prescription is not specific enough to describe one particular encoding, it is comprehensive enough to convey an idea of what a suitable choice of an encoding scheme

might look like. In addition, it is simple enough to be easily modified or extended to cover other needs. The reader should note that w.l.o.g. all separating symbols (so-called *delimiters*) such as {, }, [, or] can be omitted once the exact sequence of the parameter values as well as the number of digits into which numerical values are encoded are fixed.

From this sketch of encoding schemes it becomes apparent that in the above definition of magnitude the largest numerical value does not refer to those parameters which simply state the number of certain objects since these numbers are only introduced by the encoding of these objects.

We will also specify standard size and magnitude functions. For each encoding scheme e, we take $LNG_e(I)$ to equal the number of symbols necessary to represent I under encoding e and $MAX_e(I)$ to equal the largest number occurring in I under encoding e and 0 if no number occurs in I. In the sequel we will assume that an encoding scheme as well as a length and a magnitude function are associated with each problem.

We are now able to capture in a formal way what we mean when saying that a problem is solvable in polynomial time.

Definition 6 The *time complexity* (short: *complexity*) of an algorithm is a function that specifies, for each possible input size, the number of arithmetic operations the algorithm needs in the worst case to solve an instance of that size. ■

Since each operation is assumed to take the same time, the number of operations needs only to be multiplied by the operations' duration to determine the actual running time of the algorithm.

Definition 7 A function g(n) is called *O(h(n))* iff there exists a nonnegative constant c such that the following inequality holds:

$$|g(n)| \leq c\,|h(n)| \qquad (\forall\, n \geq 0) \quad \blacksquare$$

This so-called "big-oh" notation focusses on the highest degree term of a function; for example the function $g(n) = 10\,x^3 + 2\,x^2 - 17\,x - 9$ is $O(x^3)$.

Definition 8 An *algorithm* is called *polynomial-time* (short: *polynomial*) iff its time complexity is $O(p(LNG_e(I)))$ for some polynomial p in the length $LNG_e(I)$ of the instance, and *exponential-time* (short: *exponential*) otherwise. A *problem* is said to be *solvable in polynomial time* (short: *polynomially solvable* or *polynomial*) iff it can be solved by a polynomial-time algorithm, and *solvable in exponential time* (short: *exponentially solvable* or *exponential*) otherwise. ■

Based upon this concept, the complexity of a problem is regarded as the complexity of the best possible algorithm solving it.

Definition 9 A decision problem is *polynomial* iff it can be solved by a polynomial algorithm, and *exponential* otherwise.

Let ***P*** denote the class of all polynomial problems and ***EXP*** the class of all exponential problems. ■

Now that we have defined two of the problem classes mentioned in the beginning of this section, we will conclude this section by laying the grounds for defining the last one. To do so, we need to introduce another problem class, viz. **NP**, and another concept, viz. the *polynomial transformation*, but before providing the formal definitions, we think it worthwhile to outline the idea behind **NP**.

Being unable to prove polynomial solvability for a problem being studied, one may decide to settle for a slightly less stringent property, viz. *polynomial verifiability*. This means that for each yes-instance of the problem there exists at least one structure, called *certificate*, which allows to prove it being a yes-instance. As an example consider the following variant of the TSP: Given a finite set of cities, a positive distance between each pair of cities, and a positive bound, is there a tour of all cities having total length equal to or less than the bound? No polynomial algorithm for this problem is known. However, assume that we claim a specific instance of the TSP to be a yes-instance: If we were provided somehow with a tour for that instance, we could easily check in polynomial time whether that tour possesses the desired properties; if so, the instance indeed has the solution "yes". Here any such tour is a certificate; obviously, for each yes-instance of the TSP there exists at least one certificate which could be verified in polynomial time. Note, however, that a tour not having the desired properties is no certificate since checking it will not help us in deciding whether or not the solution is "yes".

In order to capture this idea, **NP** is usually defined in terms of a *nondeterministic algorithm* which may be viewed as being composed of two stages, the first one a *guessing stage* and the second one a *checking stage*. Given an instance I of a decision problem, the first stage "guesses" a structure C, the certificate. Both I and C are then passed on to the second stage which checks deterministically in polynomial time whether C is a certificate for I. If so, the algorithm terminates, returning as solution the answer "yes". (For more details and a formal definition cf. Garey, Johnson 1979, pp. 28-32.) For feasibility problems modelled as above, any feasible solution provides a certificate; thus, the proof that any certificate can be verified in polynomial time amounts to showing that reading the certificate (in order to instantiate the variables with these values) as well as evaluating the constraints (in order to check feasibility) can be done in polynomial time. Note that the length of C must be polynomially bounded in the length of the instance since otherwise even reading C would take exponential time, to say

nothing of verifying it. Thus, the class **NP** can be seen as the set of polynomially verifiable problems. It should be evident that this concept is merely "a definitional device [...] rather than a realistic method for solving problems" (Garey, Johnson 1979, p. 29).

Definition 10 Let *NP* denote the class of all decision problems solvable by a nondeterministic polynomial algorithm. ■

It is interesting to note that many apparently intractable problems, i.e. problems for which no deterministic polynomial algorithm is known, belong to **NP**; hence, **NP** provides a good starting point for our attempt to capture apparent intractability in a formal way.

In order to show that a problem belongs to **NP** one has to show that verifying any certificate has a time complexity which is polynomial in the input length. This involves two prerequisites: first, that the certificate can be read in polynomial time (in order to instantiate the decision variables with the values of the certificate), and second, that all constraints can be evaluated in polynomial time (in order to verify the certificate as indeed describing a solution of the considered problem). Let us examine both prerequisites. First, any instantiation of the variables establishes a potential certificate. If we assume that the number of decision variables is polynomially bounded in the length of the instance, i.e. is $O(p_1(LNG_e(I))$, then the effort for certificate reading and instantiating is also polynomial, viz. $O(p_1(LNG_e(I)) \cdot \log MAX_e(I))$. If all variables are binary, this figure even reduces to $O(p_1(LNG_e(I)) \cdot \lceil \log\ 1 + 1 \rceil) = O(p_1(LNG_e(I))$. Second, under the assumption that as well the number of constraints as the number of arithmetic operations, viz. additions and multiplications, within these constraints are polynomially bounded in the input length, i.e. are $O(p_2(LNG_e(I))$, also the evaluation of the constraints can be accomplished in polynomial time, viz. $O(p_2(LNG_e(I))$, because performing any of these operations is assumed to require a constant amount of time. Neither of these prerequisites is a trivial one (cf. Schirmer 1996a and b).

2.3. NP-Complete and NP-Hard Problems

Having defined **NP**, we need just one more concept, namely that of a polynomial transformation, to formalize **NP**-completeness.

Definition 11 Let $\Pi = (D, Y)$ and $\Pi' = (D', Y')$ be decision problems. Then Π *polynomially transforms* to Π' ($\Pi \propto \Pi'$) iff there exists a function f: $D \rightarrow D'$; $I \rightarrow I'$ such that

(i) $I \in Y$ iff $I' \in Y'$ (short: *I and I' are equivalent)*, and

(ii) f can be computed by a deterministic polynomial algorithm (short: *f is polynomial*).

f is called a *polynomial-time* (short: *polynomial transformation*) from *D* to *D'*. ■

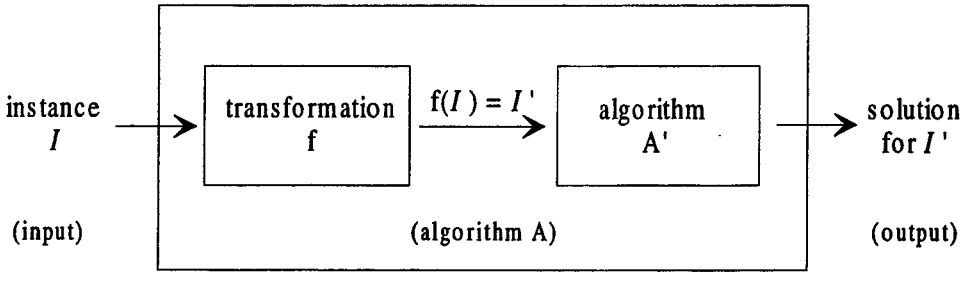
The usefulness of polynomial transformations is demonstrated by the following lemma:

Lemma 1 If $\Pi \propto \Pi'$, then $\Pi' \in P$ implies $\Pi \in P$ (and, equivalently, $\Pi \notin P$ implies $\Pi' \notin P$).

Proof: Let denote I an instance of Π, A' an (hypothetical) algorithm solving Π', and f: $D \rightarrow D'$; $I \rightarrow I'$ a polynomial transformation from D to D'. Then one can construct an algorithm A solving Π by transforming I into an instance $f(I) = I'$ of Π', using A' to solve I', and returning the solution for I' (cf. Figure 2). A solves Π since f preserves the equivalence of instances and thus the solution for I' is also correct for I. Finally, due to the polynomiality of f, A is polynomial iff A' is. ■

Figure 2

How Polynomial Transformations Work



Alternatively, the concept of polynomial transformation can be defined in the following more intuitive way. While Definition 11 emphasises that f constructs equivalent instances, the emphasis in the following definition is on how to exploit this equivalence solving Π. It is easy to see from Figure 2 that indeed both definitions are equivalent.

Definition 12 Let Π and Π' be decision problems. Then Π *polynomially transforms* to Π' ($\Pi \propto \Pi'$) iff Π could be solved by a deterministic polynomial algorithm A that, for any instance I of Π, once

(i) constructs from I an equivalent instance I' of Π' and

(ii) uses a (hypothetical) algorithm A' for Π' to solve I',

and then returns the solution produced by A' for I' as solution for I.

In this case the complete algorithm A is called a *polynomial-time* (short: *polynomial*) *transformation* of Π to Π'. ■

The underlying idea of this definition is to transform each I of Π into an instance I' of another problem Π' which can (hypothetically) be solved by A'. Then trivially any solution of I' would also be the solution of I since by construction both instances are equivalent. Notice that it is

not required that such an algorithm be known or even existent because we do not intend to construct an actually working algorithm; rather than that we are interested in how difficult to solve a given problem is. Therefore it suffices to know that a problem is at least as difficult (and in many cases just as difficult) to solve as another problem. Polynomial transformations allow to establish exactly this kind of results: Except of the time spent by A', A is polynomial; hence, if A' were polynomial, then also the algorithm A as a whole would be polynomial. Thus, using polynomial transformations one can define classes of problems which are of equivalent computational complexity. The most prominent example of such classes is the class of **NP**-complete problems.

Definition 13 A decision problem Π is *NP-complete* iff both $\Pi \in$ **NP** and all other problems in **NP** polynomially transform to Π.

Let *NPC* denote the class of all **NP**-complete problems. ■

It is far from obvious how to prove **NP**-completeness by showing that all other problems in **NP** polynomially transform to a problem at hand; in fact it is questionable whether complexity theory would have developed in the way it did if the above definition would indicate the only way of proving **NP**-completeness. Luckily, the following lemma provides a simpler approach (for a proof cf. Garey, Johnson 1979, p. 38). It states that for proving that a problem $\Pi \in$ **NP** is **NP**-complete, it suffices to show that some known **NP**-complete problem polynomially transforms to Π. Hence, it is enough to identify *one* **NP**-complete problem for which a polynomial transformation to Π can be found, rather than constructing one for *every* **NP**-complete problems.

Lemma 2 If $\Pi' \propto \Pi$ and $\Pi \in$ **NP**, then Π' **NP**-complete implies Π **NP**-complete (and, equivalently, Π not **NP**-complete implies Π' not **NP**-complete).

Often the easiest way to do this it to show that $\Pi \in$ **NP** generalizes an **NP**-complete problem $\Pi_{\mathbf{NPC}}$, i.e. contains it as a subproblem; a subproblem simply comprises a subset of the original instances and its yes-instances are exactly those of the original yes-instances which are also contained in this subset.

Definition 14 A *subproblem* (or *special case*) Π' of a decision problem $\Pi = (D, Y)$ is defined as $\Pi' = (D', Y')$ where $D' \subseteq D$ and $Y' = Y \cap D'$.

Π is then said to *generalize* Π'. ■

This can be done by restricting the problem Π at hand to a special case Π', i.e. by placing additional restraints on D. The difficulty lies in finding a restriction such that the restricted problem Π' is either identical to an **NP**-complete problem $\Pi_{\mathbf{NPC}}$ (that would be the best case we could hope for) or (more probably) that there is an apparent one-to-one correspondence

between the instances of Π' and Π_{NPC} which preserves the equivalence of solutions. In the first case, the desired transformation is the identity mapping; in the latter case, the transformation from Π_{NPC} to Π' is provided by the above correspondence between Π' and Π_{NPC}. Clearly, computing such a one-to-one transformation takes time essentially linear in the size of the respective instance; thus, any such restriction provides a polynomial transformation. We should emphasize that, even though the restriction is from the general to the special case (problem), the corresponding polynomial transformation proceeds in the opposite direction, viz. from the special to the general case. Thus, in order to prove Π **NP**-complete one may (in the best case) identify a polynomial transformation from Π_{NPC} to Π simply by restricting Π to Π_{NPC}.

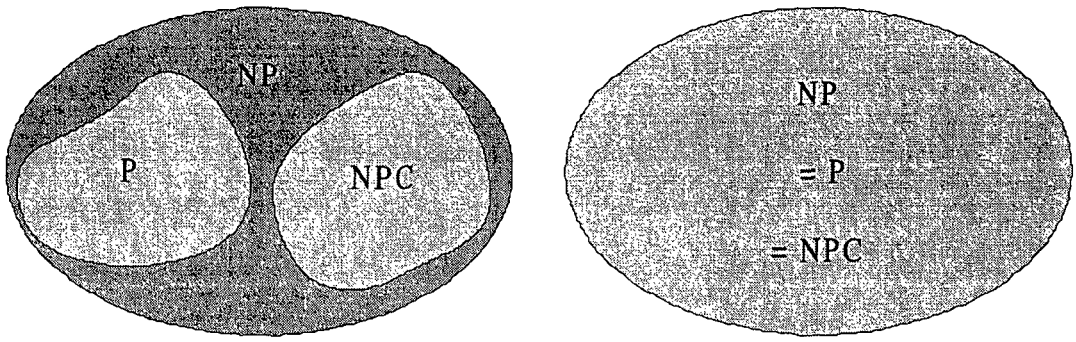
Returning to the distinction between **NP**-complete and polynomially solvable problems from the beginning of this section, clearly the problems in **P** can be considered the "easiest problems in **NP**" since all of them are solvable in polynomial time. In contrast, Lemmata 1 and 2 justify to regard the **NP**-complete problems as the "hardest problems in **NP**": If any **NP**-complete problem can be solved in polynomial time, then - by virtue of the polynomial transformations - so can all problems in **NP**. Conversely, if any problem in **NP** is intractable, then so are all **NP**-complete problems. In other words, for each **NP**-complete problem Π the property "$\Pi \in$ **P** iff **P** = **NP**" and thus the following corollary hold (cp. Garey, Johnson 1979, p. 37):

Corollary 1 If a problem Π is **NP**-complete, then Π can be solved by a polynomial algorithm iff **P** = **NP**. ■

It is also interesting to note that, even though the exact outlook of **NP** is unknown, there are only two possible topographies of **NP** with respect to **P** and **NPC** (cf. Figure 3). Either (a) **P** and **NPC** are disjoint subsets of **NP**, but there exist problems in **NP** that belong neither to **P** nor to **NPC**. Or (b) **NP** equals **P** and thus also **NPC**, in which case all three classes collapse into one (for a proof cf. e.g. Garey, Johnson 1979, pp. 154-155; Papadimitriou 1994, pp. 330-332).

Figure 3

What May NP Look Like?



(a) (b)

Although by definition **NP**-completeness applies only to problems in **NP**, and thus only to decision problems, this restriction does not limit the applicability of **NP**-completeness results to search problems. Focussing for a moment on the operations research world, clearly the following relationship holds true: On one hand, if an optimization problem is polynomially solvable, then trivially the corresponding feasibility problem is also polynomially solvable since any optimal solution is also a feasible one. On the other hand, if the feasibility problem (finding a feasible solution) is already **NP**-complete, then clearly the optimization problem (finding a not only feasible but even optimal solution) will be at least as difficult to deal with. In this way, the conjecture of intractability generally associated with **NP**-completeness can be extended straightforwardly from **NP**-complete feasibility problems to optimization problems which then are called *NP-hard.* In general, many search problems (especially optimization ones) are related in this way with their decision counterparts (especially feasibility ones). (In Section 3.6 we will address an even closer correspondence between decision and search problems in the context of *NP-equivalence.*)

To formalize this insight, the above concept of a polynomial transformation, which pertains to decision problems only, is replaced by that of a polynomial reduction which allows to cover the more general class of search problems. Recall Section 3.3 where we argued that a polynomial transformation A is - except of the time spent by A' - a polynomial algorithm and that hence, if A' were polynomial, then also the algorithm A as a whole would be polynomial. Obviously, this relationship continues to hold true even if A calls the subroutine for Π' several (though no more than a polynomially bounded number of) times and even if the hypothetical subroutine solves a search rather than a decision problem. This insight provides the basis for the following generalization.

Definition 15 Let Λ and Λ' be arbitrary problems. Then Λ *polynomially reduces* to Λ' ($\Lambda \propto_R \Lambda'$) iff Λ could be solved by a deterministic polynomial algorithm A that, for any instance I of Λ and for some polynomial p in $LNG_e(I)$, performs no more than $O(p(LNG_e(I)))$ iterations i, in each of which it

(i) constructs from I an instance I_i of Λ' and

(ii) uses a (hypothetical) algorithm A' for Λ' to solve I_i,

and then returns a solution for I.

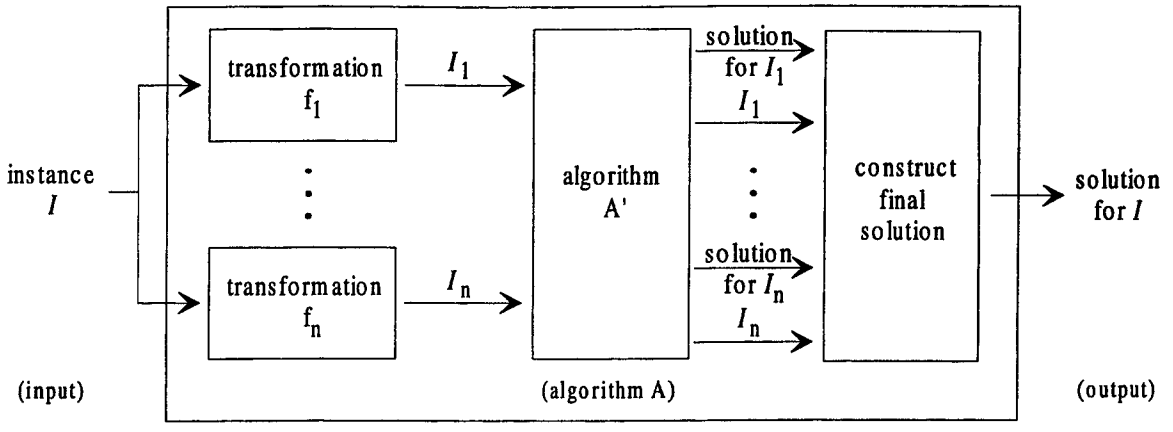
A is called a *polynomial-time* (short: *polynomial*) *reduction* of Λ to Λ'. ■

Polynomial transformations and reductions differ from each other in that the latter are allowed to iterate *more than once*, although - to maintain polynomiality - *no more than a polynomially bounded number of times*. Accordingly, it is common practice to call the algorithm A' a *subroutine* for algorithm A and to refer to step (ii) as *calling the subroutine A'*. Note that each polynomial transformation can also be regarded as a polynomial reduction since it calls its subroutine just once. (The mutual relationships between the above defined polynomial transformations and reductions as well as other forms of reductions are investigated in detail in Ladner, Lynch et al. 1975). Moreover, being defined on arbitrary problems, polynomial reductions can be applied to decision as well as search problems; this includes also the cases of reducing a search problem to a decision one or vice versa.

The idea of a polynomial reduction is to construct in each iteration i from I a different (and not necessarily equivalent) instance I_i of a second problem Λ' which could (hypothetically) be solved by A' (cf. Figure 4). Notice that each such construction step may (and often will) depend on information generated in the previous iteration (cp. the exemplary proof of Theorem 1 below); in other words, it may be defined in terms of - besides the original instance I - the instance I_{i-1} considered in the previous iteration and the corresponding solution. Then, having compiled enough information in terms of intermediate instances and solutions, the final solution can be constructed from them. Loosely speaking, the instances provide the information about the input to the individual iterations while the solutions provide the respective results, which are thus depicted in Figure 4 as being "passed through" by A'. Since this last construction step "translates" the information produced in the iterations into a solution for I, the equivalence of instances, which is part of Definition 12 for polynomial transformations, is no requirement for polynomial reductions and can be waived here.

Figure 4

How Polynomial Reductions Work



We can now extend our definition of **NP**-completeness by replacing polynomial transformability with polynomial reducibility, an extension allowing to embrace also problems which are more general than decision problems and thus do not belong to **NP**. Such problems which are at least as hard to solve as **NP**-complete ones but do not belong to **NP** are called **NP**-hard; they have been characterized as "at least as hard" (Garey, Johnson 1979, p. 109) as the **NP**-complete ones because, if any **NP**-hard problem is polynomially solvable, then so are all **NP**-complete problems and thus all problems in **NP**. In other words, for each **NP**-hard problem Π the property "**P** = **NP** if $\Pi \in$ **P**" and thus the following corollary hold:

Corollary 2 If a problem Π is **NP**-hard, then Π cannot be solved by a polynomial algorithm unless **P** = **NP**. ■

Definition 16 A search problem Σ is *NP-hard* iff some **NP**-complete problem polynomially reduces to Σ.

Let *NPH* denote the class of all **NP**-hard problems. ■

With respect to the operations research world this definition implies that any optimization problem can be characterized as **NP**-hard if the corresponding feasibility problem is **NP**-complete. This is due to the fact that for any optimization problem trivially exists a polynomial reduction from its feasibility version.

Lemma 3 If Ω is an optimization problem and Φ the corresponding feasibility version, then $\Phi \propto_R \Omega$.

Proof: Assuming some hypothetical algorithm A' solving Ω, each instance I of Φ can be solved by an algorithm A that proceeds in the following way: Construct from I an instance I' of Ω and solve it by A'. If A' returns some optimal solution for I', then by Definition 2 this solution is also feasible; accordingly the solution for I is "yes". If A' returns "no", then no optimal - and thus no feasible solution for I' exists; hence the solution for I is "no" as well. This construction can be done in linear, thus in polynomial time. Then by Definition 15 the algorithm A is a polynomial reduction from Φ to Ω. ■

While this application of **NP**-hardness to the operations research world does not use polynomial reductions to its full extent since it requires only one iteration, Section 3.6 will provide an example illustrating the greater power of polynomial reductions (cf. the proof of Theorem 1). (For another example cf. Garey, Johnson 1979, pp. 114-117: Even though the *K^{th} largest subset problem* can be shown to be **NP**-hard using a polynomial reduction, no such proof has been discovered yet which relies only on a polynomial transformation.)

Note finally that a problem which generalizes another, **NP**-complete problem is not necessarily **NP**-complete itself. A generalization of an **NP**-complete problem need not be member of **NP**, and may thus be exponential (cf. Schirmer 1996a).

3. Extensions of the Classical Theory

3.1. Strongly NP-Complete and Strongly NP-Hard Problems

In Section 3.1 we argued that there is good reason to conjecture that **NP**-complete problems cannot be solved in polynomial time, and so we reported that they are generally considered intractable. There are, however, certain problems which - despite their **NP**-completeness - can be solved rather efficiently. Consider e.g. the PARTITION problem which may be couched in the following way:

Given $J \in \mathbb{N}$, $J = \{1,...,J\}$, and $\mu_j \in \mathbb{N}$ $(1 \leq j \leq J)$, is there a set $J' \subseteq J$ such that

$$\sum_{j \in J'} \mu_j = \sum_{j \in J - J'} \mu_j \qquad (1)$$

holds?

The PARTITION problem was one of the first problems for which membership in the class of **NP**-complete problems has been shown (Karp 1972). Still, it can be solved by a simple dynamic programming procedure where - letting B denote half the sum of all μ_j $(1 \leq j \leq J)$ - $J \cdot B$ entries have to be made in a table for any given instance (Garey, Johnson 1979, pp. 90-

91). Since any such entry can be made in at most constant time, the time complexity of the algorithm can be bounded by a polynomial in the number of entries and thus is O(J·B).

At first glance, this result is rather surprising since it seems to provide a polynomial algorithm for an **NP**-complete problem. That being the case would imply **P = NP = NPC**, a result which would completely obviate the need for complexity theory since all apparently intractable problems in **NP** could all be solved in polynomial time by constructing polynomial algorithms from the cited algorithm and the respective polynomial transformation from PARTITION to the respective problem (cf. Figure 2). But in fact, O(J·B) is *not* polynomial, i.e. polynomial in the input length, but exponential, since $LNG_{STD}(I_{PARTITION})$ is $O(J \cdot \log_2 B)$. However, since $MAX_{STD}(I_{PARTITION}) = O(B)$, a bound for O(J·B) could be formulated in terms of some bivariate polynomial in the variables length and magnitude of the instance. In other words, the cited algorithm would be polynomial-time with respect to such a bivariate polynomial. This kind of algorithms is called *pseudo-polynomial* (Garey, Johnson 1978).

Definition 17 An exponential *algorithm* is called *pseudo-polynomial-time* (short: *pseudo-polynomial*) iff its time complexity is $O(p(MAX_e(I), LNG_e(I)))$ for some bivariate polynomial p in the variables length $LNG_e(I)$ and magnitude $MAX_e(I)$ of the instance. A *problem* is said to be *solvable in pseudo-polynomial time* (short: *pseudo-polynomially solvable*) iff it can be solved by a pseudo-polynomial algorithm. ■

Consider a **NP**-complete problem Π. If Π, like the PARTITION problem, can be solved by a pseudo-polynomial algorithm A, then those instances of Π where the magnitude of all numbers is polynomially bounded in the length of the instance can be solved polynomially (more exactly, the pseudo-polynomial time complexity of A becomes polynomial for all such instances). This in turn implies that only those instances of pseudo-polynomially solvable problems will be intractable whose magnitude grows exponentially with their length. Consequently, **NP**-complete problems being pseudo-polynomially solvable are often regarded as tractable, since in many cases typical, practically relevant instances will not contain such large numbers and thus will be polynomially solvable. Conversely, problems where even those instances satisfying the above condition cannot be solved polynomially unless **P = NP**, i.e. are **NP**-complete, are considered as being **NP**-complete in an even stronger than the usual sense.

Definition 18 Given a decision problem $\Pi = (D, Y)$ and a polynomial p (over the integers), let denote Π_p the subproblem obtained from Π by restricting D to those instances which satisfy

$$MAX_e(I) \leq p(LNG_e(I)) \tag{2}$$

Then Π is *NP-complete in the strong sense* (short: *strongly NP-complete*) iff both $\Pi \in$ **NP** and there exists a polynomial p (over the integers) such that Π_p is **NP**-complete.

Let *sNPC* denote the class of all strongly **NP**-complete problems. ■

Now, the following observations about pseudo-polynomial algorithms are straightforward (for proofs cf. Garey, Johnson 1979, p. 95):

Corollary 3 If a decision problem Π can be solved by a pseudo-polynomial algorithm, then Π_p can be solved by polynomial algorithm. ■

Corollary 4 If a problem Π is strongly **NP**-complete, then Π can be solved by a pseudo-polynomial algorithm iff **P** = **NP**. ■

In order to prove a decision problem Π to be strongly **NP**-complete one may identify a specific polynomial p and show that the restricted problem Π_p is **NP**-complete. However, similar to Lemma 2 which - using the concept of polynomial transformations - provided an easier way of proving **NP**-completeness, there exists a lemma which - using the concept of *pseudo-polynomial transformations* - facilitates the task of proving strong **NP**-completeness.

Definition 19 Let $\Pi = (D, Y)$ and $\Pi' = (D', Y')$ be decision problems. Then Π *pseudo-polynomially transforms* to Π' ($\Pi \propto_p \Pi'$) iff there exists a function f: $D \rightarrow D'$; $I \rightarrow I'$ such that

(i) $I \in Y$ iff $I' \in Y'$,

(ii) f can be computed by a deterministic pseudo-polynomial algorithm,

(iii) there is a polynomial q_1 such that $q_1(LNG'_{e'}(f(I))) \geq LNG_e(f(I))$ holds for all instances $I \in D$,

(iv) there is a bivariate polynomial q_2 such that $MAX'_{e'}(f(I)) \leq q_2(LNG_e(I), MAX_e(I))$ holds for all instances $I \in D$.

f is called a *pseudo-polynomial-time* (short: *pseudo-polynomial*) *transformation* from D to D'.■

The first two conditions merely translate the definition of polynomial transformations (cp. Definition 11) to the context of pseudo-polynomiality; the two additional conditions serve to keep length and magnitude of the constructed instance within a reasonable, viz. polynomial, corridor around length and magnitude of the given instance such that pseudo-polynomiality results for one instance will continue to hold true for the other one constructed from it. Conditions (i) and (ii) require that the given instance I and the constructed instance I' be equivalent, and that the construction function f be pseudo-polynomial. Condition (iii) ensures that the construction does not cause a substantial, i.e. logarithmic, decrease in the instance length. Note that the opposite case, namely an exponential increase in the instance length, is impossible because f is pseudo-polynomial, implying that computing, let alone writing down, I' can at most require pseudo-polynomial time such that trivially the instance length of I' is pseudo-polynomially bounded from above. Finally, condition (iv) guarantees that the magnitude of the largest number occurring in the constructed instance I' will not be exponentially

larger than the magnitude and length of the given instance *I*. The opposite case where the magnitude of the constructed instance *I'* is logarithmically smaller than the magnitude and length of the given instance *I* is uncritical since we are only interested in the magnitude of the largest - and not the smallest - occurring number. It is interesting to notice that not every polynomial transformation will be pseudo-polynomial, because condition (iv) will not be met by all of them; otherwise all **NP**-complete problems would also be **NP**-complete in the strong sense. Conversely, due to condition (ii) not every pseudo-polynomial transformation will be polynomial, because its time complexity may also be polynomially bounded in the magnitude of the instance (cf. Brüggemann 1995, p. 102).

We can now formulate the following lemma (for a proof cf. Garey, Johnson 1979, p. 101-102):

Lemma 4 If $\Pi' \propto_p \Pi$ and $\Pi \in$ **NP**, then Π' strongly **NP**-complete implies Π strongly **NP**-complete (and, equivalently, Π not strongly **NP**-complete implies Π' not strongly **NP**-complete).

Hence, for proving that a problem $\Pi \in$ **NP** is strongly **NP**-complete, it suffices to show that some known strongly **NP**-complete problem pseudo-polynomially transforms to Π. As for the **NP**-complete problems, often the easiest way to do this is to show that Π contains a strongly **NP**-complete problem Π_{sNPC} as a subproblem by restricting Π to a special case Π'. Again, one has to find a restriction such that Π' is either identical to a strongly **NP**-complete problem Π_{sNPC} or that there is an apparent one-to-one correspondence between the instances of Π' and Π_{sNPC}. In the first case, the desired transformation is the identity mapping; in the latter case, the transformation from Π_{sNPC} to Π' is provided by the above correspondence between Π' and Π_{sNPC}. Note that this transformation is not only polynomial but also pseudo-polynomial since all numbers occurring in any constructed instance *I'* also occur in the original instance *I* such that the magnitude of *I'* can even be bounded by the identical polynomial in the magnitude of *I*.

We will now bring together in a class of their own all decision problems which do *not* satisfy (2), i.e. which possess at least one numerical parameter whose domain cannot be bounded from above by a polynomial in the length of any possible instance. We will see in the following that only for these the distinction between being "merely" **NP**-complete and being strongly **NP**-complete is meaningful because for any other problems both concepts immediately collapse into one.

Definition 20 A problem is called a *number problem* if there exists no polynomial p such that for all instances

$$MAX_e(I) \leq p(LNG_e(I)) \tag{3}$$

holds, a *non-number problem* otherwise. ■

Corollary 5 If Π is a non-number problem, then an algorithm for Π is pseudo-polynomial iff it is polynomial. ■

Corollary 6 If Π is a non-number problem, then Π is strongly **NP**-complete iff it is **NP**-complete. ■

Accordingly, with respect to strong **NP**-completeness, **NPC** naturally decomposes into three distinct subclasses.

First, all non-number problems, i.e. those problems where the magnitude $MAX_e(I)$ of any instance I is naturally limited because no instance may have arbitrarily large numbers. Examples are problems dealing with graphs where the only numbers occurring are indices denoting the different vertices of a graph. Since the number of vertices is part of the input of the instance, there can be no index which cannot be bounded polynomially in the length of the input. For these problems, **NP**-completeness immediately implies strong **NP**-completeness.

Second, number problems, i.e. problems without any natural limitations on the magnitude of their instances, where the unlimited numbers have no influence on the complexity of the problems. Even placing the additional restriction (2) on their instances does not make them polynomially solvable. For instance, the TSP allows for arbitrarily large distances between the different cities; however, even after restricting it to distances of only 1 or 2 the problem remains **NP**-complete (Garey, Johnson 1979, pp. 35-36). These number problems are strongly **NP**-complete, and thus they can be solved neither by a polynomial nor by a pseudo-polynomial algorithm.

Third, number problems where the magnitude of the instances indeed plays a vital role with respect to their complexity. Such problems like PARTITION can only be shown to be **NP**-complete by polynomial transformations which produce exponentially large numbers (cf. Garey, Johnson 1979, p. 61). These problems are **NP**-complete but not strongly **NP**-complete; in other words, even though they are not solvable by a polynomial algorithm, they still can be solved by an algorithm taking pseudo-polynomial time.

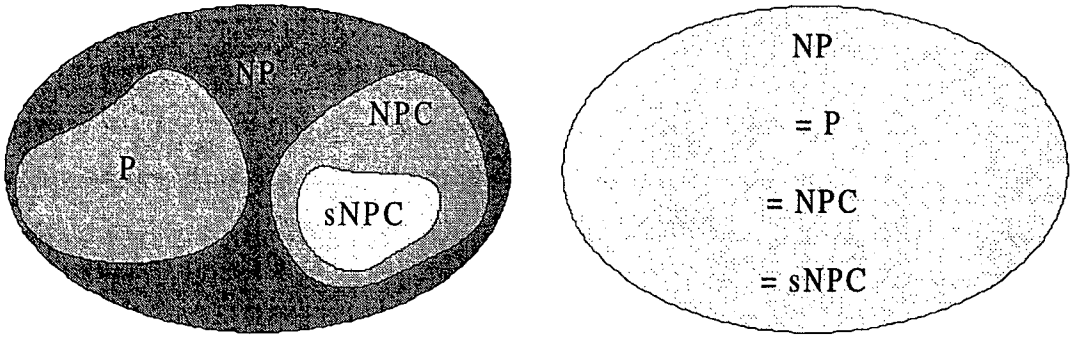
Unfortunately, it is not clear a priori whether a certain number problem belongs to the second or the third class. The fundamental implication of this fact is that a mere **NP**-completeness result allows to regard a problem as intractable if and only if the problem is a non-number

one. In other words, for a number problem only a proof of its strong **NP**-completeness will provide sufficient grounds for advocating to use heuristics for solving them.

We may now update our possible topographies of **NP** with the knowledge that **sNPC** is a proper subset of **NPC** (cf. Figure 5). Either (a) **P** and **NPC** are disjoint subsets of **NP**, but there exist problems in **NP** that belong neither to **P** nor to **NPC**; **sNPC** in turn is proper subset of **NPC**. Or (b) **NP** equals **P**, thus also **NPC**, thus also **sNPC** such that all four classes collapse into one.

Figure 5

What May NP Look Like? (Extended)



(a) (b)

Finally, similar to the above extension of the concept of **NP**-completeness (pertaining to decision problems) to **NP**-hardness (pertaining to search problems), the conjecture associated with strongly **NP**-complete (decision) problems, namely that even those of their subproblems which satisfy (2) are intractable, can be extended straightforwardly to search problems which then are called strongly **NP**-hard. In order to do so, the above definition of a polynomial reduction, which earlier allowed the same extension from the **NP**-complete to the **NP**-hard problems, needs to be generalized to a pseudo-polynomial reduction.

Definition 21 Let Λ and Λ' be arbitrary problems. Then Λ *pseudo-polynomially reduces* to Λ' ($\Lambda \propto_{pR} \Lambda'$) iff Λ could be solved by a deterministic pseudo-polynomial algorithm A that, for any instance I of Λ and for some bivariate polynomial p in $LNG_e(I)$ and $MAX_e(I)$, performs no more than $O(p(LNG_e(I)))$ iterations i, in each of which it

(i) constructs from I an instance I_i of Λ' which satisfies conditions (ii) and (iii) of Definition 19 and

(ii) uses a (hypothetical) algorithm A' for Λ' to solve I_i,

and then returns a solution for I.

A is called a *pseudo-polynomial-time* (short: *pseudo-polynomial*) *reduction* of Λ to Λ'. ■

Definition 22 A search problem Σ is ***NP-hard in the strong sense*** (short: ***strongly NP-hard***) iff some strongly **NP**-complete problem pseudo-polynomially reduces to Σ.

Let ***sNPH*** denote the class of all strongly **NP**-hard problems. ■

Now, any optimization problem may be characterized as strongly **NP**-hard if its feasibility variant can be shown to be strongly **NP**-complete. This is due to the fact that for any optimization problem the obvious polynomial reduction from its feasibility version is also pseudo-polynomial.

Lemma 5 If Ω is an optimization problem and Φ the corresponding feasibility version, then $\Phi \propto_{pR} \Omega$.

Proof: By Lemma 3, there exists a polynomial reduction A from Φ to Ω; hence it remains to show that A is also pseudo-polynomial. This is the case since all numbers occurring in any constructed instance I' of Ω also occur in the original instance I of Φ such that the magnitude of I' can even be bounded by the identical polynomial in the magnitude of I. ■

Finally, we may formulate the following insights:

Corollary 7 If a problem Π is strongly **NP**-hard, then Π cannot be solved by a pseudo-polynomial algorithm unless **P** = **NP**. ■

Corollary 8 If Σ is a non-number search problem, then Σ is strongly **NP**-hard iff it is **NP**-hard. ■

To summarize the above insights, Table 4 illustrates which complexity results one might possibly expect for a given kind of problem. E.g., since optimization problem belongs to the search problems (recall Figure 1), it may be polynomially solvable or **NP**-hard (possibly in the strong sense). In contrast, any feasibility problem constitutes a decision problem, which may be polynomially solvable or **NP**-complete (possibly in the strong sense).

Table 4

Possible Complexity Results

A ...	may be...
decision problem...	- polynomially solvable
	- **NP**-complete
	- strongly **NP**-complete
	- ...
search problem...	- polynomially solvable
	- **NP**-hard
	- strongly **NP**-hard
	- ...

Note that Table 4 refers explicitly only to those complexity classes introduced above. While the entries "..." indicate that this list is not exhaustive - in fact, there exist many other complexity classes (For examples such as **PBAND** and **co-NP** cp. Garey, Johnson 1979, pp. 153-186; Papadimitriou 1994, pp. 409-499; Aho, Hopcroft, and Ullman, Chapters 13, 14; cf. also Figure 4) - the above classes have been found to be the most influential ones with respect to the design of algorithms for practical applications.

3.2. Encoding Schemes, Length and Magnitude Functions Revisited

The definition given above for encoding schemes (cf. Definition 5) is general enough to allow a wide range of encoding schemes for a given problem. For instance, a graph being part of a problem could be encoded in terms of an adjacency matrix as well as an adjacency list or a list of its nodes followed by a list of its arcs, represented as pairs of incident nodes. Therefore, having defined several complexity classes such as **P** or **NPC**, the question arises: Could our choice of a specific encoding be influential on the complexity results we obtain for a problem? This question is not a trivial one since our notion of polynomial and pseudo-polynomial algorithms is directly linked to the length of the input which depends on the respective encoding: for instance, the greater the cardinality of the alphabet used to encode an instance, the smaller the resulting input.

To be able to answer this question, one has to recall the kind of complexity results we are interested in: Actually, we do not look for the exact running times of algorithms but rather for categorial results establishing whether their time complexities are polynomial, probably not polynomial (or pseudo-polynomial), or exponential. Since the combination of two polyno-

mials is again a polynomial, different time complexities will be categorially the same as long as they differ at most by a polynomial factor. It turns out that under encodings which are concise, i.e. do not expand the input artificially, and use non-unary alphabets, i.e. alphabets Γ with $|\Gamma| \geq 2$, the resulting input lengths are of categorially the same size, i.e. they are in a fixed ratio (Note that for any base $B \geq 2$ of an arithmetic system $\log_B n = \log n / \log B$ holds and $\log B$ is constant once B is fixed. Hence $\log_B n$ is $O(\log n)$ for any B such that no matter what arithmetic system, i.e. alphabet is used, the length of the representation of an integer n is $O(\log n)$) and thus differ at most by a polynomial factor (for more details cf. Garey, Johnson 1979, pp. 19-23; Brüggemann 1995, pp. 118-119). Under this assumption, the running times of an algorithm working on two differently encoded versions of the same instance will differ at most by a polynomial factor, consequentially, the algorithm will be polynomial either in none or in both cases. In other words, the existence of a polynomial (or pseudo-polynomial) algorithm does not depend on the particular choices made in this respect as long as they meet the above restrictions.

This insight led to the widely accepted assumption that all complexity results hold independent of the respective encoding scheme used. However, recently it was shown that the above restrictions are not as trivial as they might seem. Brüggemann (1995) demonstrated that a commonly used encoding which is widely accepted as being concise may cease to remain concise in some circumstances. Some encodings may - if only for certain, pathological instances - produce inputs which contain redundant, superfluous information such that the encodings are no longer concise. Hence, different encodings will not always lead to categorially the same input sizes such that complexity results can no longer be safely assumed to be encoding-independent (cf. Schirmer 1996a and b).

Let us now turn to the LNG(.) and MAX(.) functions. Since they are required only to *correspond* to the length of the input, it is clear that also different length and magnitude functions could be defined even for the same problem and encoding. For example, for the TSP under standard encoding, length functions such as

$$\mathrm{LNG}_{\mathrm{STD}}(I_{\mathrm{TSP}}) = C + \max\{\lceil \log d_{ij} \rceil \mid (i, j) \in C \times C\}$$
$$\mathrm{LNG}_{\mathrm{STD}}(I_{\mathrm{TSP}}) = C + \lceil \log \max\{d_{ij} \mid (i, j) \in C \times C\} \rceil$$
$$\mathrm{LNG}_{\mathrm{STD}}(I_{\mathrm{TSP}}) = C \cdot \lceil \log \max\{d_{ij} \mid (i, j) \in C \times C\} \rceil$$
$$\mathrm{LNG}_{\mathrm{STD}}(I_{\mathrm{TSP}}) = C + \Sigma_{(i,j) \in C \times C} \lceil \log d_{ij} \rceil$$

and, analogously, magnitude functions like

$$\mathrm{MAX}_{\mathrm{STD}}(I_{\mathrm{TSP}}) = \max\{d_{ij} \mid (i, j) \in C \times C\}$$
$$\mathrm{MAX}_{\mathrm{STD}}(I_{\mathrm{TSP}}) = \Sigma_{(i,j) \in C \times C} d_{ij}$$

might be useful and appropriate. Again, the question arises whether different choices might possibly bring about categorially different complexity results. Here it turns out (cf. Garey, Johnson 1979, pp. 92-93) that complexity results hold for all length and magnitude functions as long as they are *polynomially related* in the following sense:

Definition 23 Let Λ be an arbitrary problem and e an encoding scheme for Λ. Then two *length functions* $LNG_e(I)$ and $LNG'_e(I)$ are *polynomially related* iff there exist two polynomials p and p' such that, for all $I \in D_\Lambda$,

(i) $LNG_e(I) \leq p'(LNG'_e(I))$ and

(ii) $LNG'_e(I) \leq p(LNG_e(I))$.

Two pairs of *length and magnitude functions* $(LNG_e(I), MAX_e(I))$ and $(LNG'_e(I), MAX'_e(I))$ are *polynomially related* iff $LNG_e(I)$ and $LNG'_e(I)$ are polynomially related and there exist two bivariate polynomials q and q' such that, for all $I \in D_\Lambda$,

(iii) $MAX_e(I) \leq q'(MAX'_e(I), LNG'_e(I))$ and

(iv) $MAX'_e(I) \leq q(MAX_e(I), LNG_e(I))$. ■

To illustrate this concept, consider the following example. Let $a \in \mathbb{N}$ and $b \in \mathbb{N}$; then $LNG_e(I) = a + b$ and $LNG'_e(I) = a \cdot b$ are polynomially related. In order to prove this, let $p(x) = x + 1$ and $p'(x) = x^2$. Then $a + b \leq p'(a \cdot b) = a \cdot b + 1$ and $a \cdot b \leq p(a + b) = (a + b)^2 = a^2 + 2ab + b^2$ hold by induction.

In other words, two length or magnitude functions are called polynomially related if they mutually differ at most by a polynomial factor. Suitable choices of these functions will only in the most unusual cases fail to meet these conditions (cf. Garey, Johnson 1979, pp. 92-93).

Therefore, the categorial results we are trying to establish are essentially independent of the particular choices of length and magnitude functions because exchanging one length or magnitude function for another will neither change a polynomial algorithm into an exponential one nor vice versa, as long as both functions are polynomially related.

3.3. NP-Easy and NP-Equivalent Problems

In Section 2.3 we argued that a search problem qualifies as **NP**-hard if some **NP**-complete decision problem reduces to it. We provided a general class of examples by observing that any optimization problem is **NP**-hard if its feasibility variant is **NP**-complete (recall Lemma 3).

However, for many optimization problems also the opposite relation holds: the optimization problem reduces to its feasibility variant. The implication of this fact being that the optimization problem can be no harder to solve than the corresponding feasibility problem. More generally speaking, there are search problems which can be polynomially reduced to some

decision problem. In accordance with the above terminology, calling problems **NP**-hard if they are at least as hard as the **NP**-complete ones but do not belong to **NP**, one may call problems **NP**-easy if they are no harder than the **NP**-complete ones but do not belong to **NP**. Consequentially, for each **NP**-easy problem Π the property "$\Pi \in \mathbf{P}$ if $\mathbf{P} = \mathbf{NP}$" and thus the following corollary hold:

Corollary 9 If a problem Σ is **NP**-easy, then $\mathbf{P} = \mathbf{NP}$ cannot hold unless Σ can be solved by a polynomial algorithm. ■

Definition 24 A search problem Σ is *NP-easy* iff Σ polynomially reduces to some **NP**-complete problem.

Let ***NPE*** denote the class of all **NP**-easy problems. ■

Further, a search problem being as well **NP**-hard as **NP**-easy is referred to as *NP-equivalent*, meaning that itself and the **NP**-complete problems are of equivalent computational complexity.

Definition 25 A search problem Σ is *NP-equivalent* iff Σ is **NP**-hard and **NP**-easy.

Let ***NPQ*** denote the class of all **NP**-equivalent problems. ■

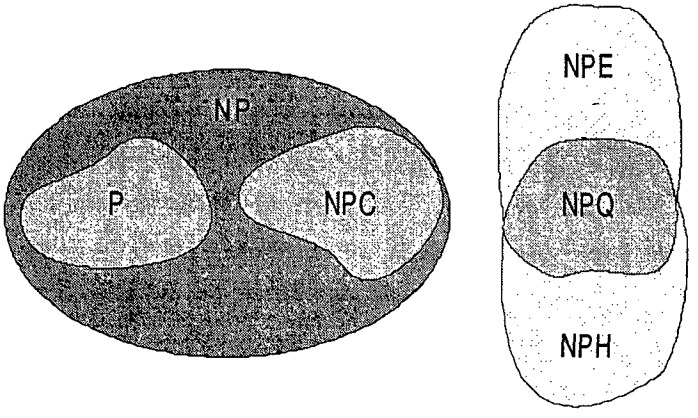
As is true for the **NP**-complete problems, as well for each **NP**-equivalent problem Σ the property "$\Sigma \in \mathbf{P}$ iff $\mathbf{P} = \mathbf{NP}$" holds.

Corollary 10 If a problem Σ is **NP**-equivalent, then Σ can be solved by a polynomial algorithm iff $\mathbf{P} = \mathbf{NP}$. ■

Therefore, if someone would eventually be able to answer the fundamental question "Is $\mathbf{P} = \mathbf{NP}$?" ("yes" or "no"), this answer would also immediately and finally establish the complexity of all **NP**-equivalent problems (as polynomial or exponential). Note that this relation does not hold for mere **NP**-hardness results. If indeed $\mathbf{P} = \mathbf{NP}$ holds then an **NP**-hard problem Σ could still be as well polynomial as exponential for all we know is that it is not easier than the - then polynomially solvable - **NP**-complete problems; still, it could be much harder to solve. In other words, a generalization of an **NP**-complete or **NP**-equivalent problem Σ will at least be as hard to solve as Σ; still, they may in the same time harder, i.e. neither **NP**-easy nor **NP**-equivalent. Figure 6 provides a sketch of the relationship between these problem classes.

Figure 6

*What May The World Beyond **NP** Look Like?*



no harder than **NPC** since
$\mathbf{NPE \subseteq P \Leftarrow P = NP}$

as hard as **NPC** since
$\mathbf{NPQ \subseteq P \Leftrightarrow P = NP}$

at least as hard as **NPC** since
$\mathbf{NPH \subseteq P \Rightarrow P = NP}$

Applying these definitions, an **NP**-equivalence proof for a search problem Σ has to proceed in the following four steps: First, provide some **NP**-complete decision problem Π; second, construct a polynomial reduction from Π to Σ; third, provide some **NP**-complete decision problem Π'; and fourth, construct a polynomial reduction from Σ to Π'. The first two steps serve to prove Σ is **NP**-hard while the last two ones show it is **NP**-easy. A schematic overview of the sequence of polynomial transformations and reductions involved in proving **NP**-equivalence of optimization problems is given in Figure 7 where "⇨" is to be read as "$\propto_R$". Generally, Π and Π' may be two different problems (case (a)); still, in many cases Π and Π' will actually denote the same problem (case (b)).

Figure 7

Sequence of Polynomial Reductions
Involved in Proving NP-Equivalence in General

This procedure can be greatly simplified when dealing with an optimization problem Ω. In order to do so, one has to define two new but straightforward problems, namely the *threshold* and the *extension variant* of an optimization problem.

Definition 26 The *threshold variant* $\Omega^{thr} = (D', Y)$ of an optimization problem $\Omega = (D, S)$ is a decision problem where each instance $I'_i \in D'$ is a triple $I'_i = (F_i, f_i, B)$, $B \in \mathbb{N}$, and the set of yes-instances $Y \subseteq D'$ consists of those instances I'_i where at least one feasible solution $s \in F_i$ satisfies the inequality $f_i(s) \leq B$ in the case of minimization or $f_i(s) \geq B$ in the case of maximization. ■

Loosely speaking, the threshold variant replaces the optimality criterion by a threshold value that the objective function value has to stay below in the case of minimization (to exceed in the case of maximization). Instead of searching for a feasible solution with minimum objective function value, one asks "Is there a feasible solution having an objective function value of B or less (more)?". As an example consider the threshold variant TSP^{thr} of the TSP (which, even though only informally, was already introduced in Section 3.2):

Given $C \in \mathbb{N}$, $C = \{1,...,C\}$ (its elements called *cities*), $d_{ij} \in \mathbb{N}$ $(\forall (i, j) \in C \times C)$ (called a *distance* between each pair of cities), and $B \in \mathbb{N}$ (called a *bound*), is there an ordering $<\pi(1),...,\pi(K),...,\pi(C)>$ (called a *full tour*) of C such that the expression (4) (called the *length* of the tour)

$$\left(\sum_{i=1}^{m-1} d_{\pi(i),\pi(i+1)} \right) + d_{\pi(m),\pi(1)} \tag{4}$$

is equal to or less than B?

Definition 27 The *extension variant* $\Omega^{ext} = (D', Y)$ of an optimization problem $\Omega = (D, S)$ is a decision problem where each instance $I'_i \in D'$ is a quadruple $I'_i = (F_i, f_i, B, s')$, $B \in \mathbb{N}$, s' a partial solution for Ω and the set of yes-instances $Y \subseteq D'$ consists of those instances I'_i where the partial solution s' can be extended to a full solution s having an objective function value of B or less in the case of minimization or of B or more in the case of maximization. ■

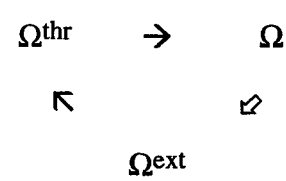
The extension variant of a minimization (maximization) problem can be characterized as augmenting the problem by a positive bound and a partial solution and asking "Can the partial solution be extended to a full solution with an objective function value of B or less (more)?" In most occasions it is quite straightforward to see what a partial solution for a given optimization problem will be like. Still, we refrain from rigging up a general, formal definition because a partial solution clearly will need to be defined in terms of the problem at hand. As an example consider the extension variant TSP^{ext} of the TSP:

Given $C \in \mathbb{N}$, $C = \{1,...,C\}$ (its elements called *cities*), $d_{ij} \in \mathbb{N}$ $(\forall (i, j) \in C \times C)$ (called a *distance* between each pair of cities), $B \in \mathbb{N}$ (called a *bound*), $K \in C$, and an ordering $\Xi = <\pi(1),...,\pi(K)>$ (called a *partial tour*) of $\{1,...,K\}$, can Ξ be extended to an ordering $<\pi(1),...,\pi(K),...,\pi(C)>$ (called a *full tour*) of C such that the expression (4) (called the *length* of the tour) is equal to or less than B? Note that for $K = C$ the above notion of a partial tour also includes a full tour as a special case.

Now, given an optimization problem Ω, the proof idea is to show that the threshold variant Ω^{thr} is **NP**-complete and reduces to Ω, and that conversely Ω reduces to Ω^{ext} which in turn reduces to Ω^{thr}; due to the transitivity of reductions this implies that Ω reduces to Ω^{thr}. Again, the first two steps serve to prove the considered problem is **NP**-hard while the last two ones show it is **NP**-easy. A schematic overview of the sequence of polynomial transformations and reductions involved in proving **NP**-equivalence of optimization problems is given in Figure 8 where "→" and "⇨" are to be read as "$\propto$" and "$\propto_R$", respectively. Although this work plan appears to be even more complicated, it is actually simpler: The required polynomial reduction from Ω^{thr} to Ω trivially exists since the optimization problem can be restricted to its threshold variant, inducing a polynomial reduction. In the same way, the polynomial reduction from the extension to the threshold variant can be established by restricting the latter to the former. Accordingly, only two steps of the proof must be exhibited explicitly: First, prove the **NP**-completeness of the threshold variant; second, provide a polynomial reduction from the optimization problem to its extension variant. In order to clarify matters, we will now give an exemplary proof of the **NP**-equivalence of the TSP.

Figure 8

Sequence of Polynomial Transformations and Reductions Involved in Proving NP-Equivalence of Optimization Problems



Theorem 1 TSP is **NP**-equivalent.

Proof:

(i) (TSP is **NP**-hard)

(TSP^{thr} is **NP**-complete) Follows from Garey, Johnson 1979, pp. 35-36 and Theorem 3.4, pp. 56-60.

(TSP^{thr} $\propto$ TSP) Holds trivially since TSP can be restricted to TSP^{thr} (cp. Lemma 3).

Now, since some **NP**-complete problem polynomially transforms to TSP, TSP is **NP**-hard.

(ii) (TSP is **NP**-easy)

(TSP^{thr} is **NP**-complete) See above.

(TSP^{ext} $\propto$ TSP^{thr}) Holds trivially since TSP^{thr} can be restricted to TSP^{ext}.

(TSP $\propto_R$ TSP^{ext}) Suppose an algorithm A that solves TSP^{ext} when provided with an instance [C, d, B, Ξ] of TSP^{ext}. (To specify the particular instance which A has to solve, we will refer to the application of A as *calling* A[C, d, B, Ξ].) We now have to show that any instance I of TSP could be solved by calling A on several different instances of TSP^{ext} while calling A no more than a number of times polynomially bounded in $LNG_{STD}(I_{TSP})$. We will do this in two steps: First, determine the optimal objective function value of I_{TSP}, i.e. the minimum tour length; second, construct an optimal solution, i.e. a full tour having minimum length.

As any optimal tour incides with all cities and any optimal tour may be cyclically permuted without changing its length, there are at least C optimal tours, each starting off at one of the C cities in C and having length B^*. From $d_{ij} \in \mathbb{N}$ $(\forall (i, j) \in C \times C)$ it is also clear that $C \leq B^*$, and - letting stand B_{MAX} for $C \cdot \max \{d_{ij} \mid (i, j) \in C \times C\}$ - that $B^* \leq B_{MAX}$. Accordingly, performing a binary search within this interval, one can determine B^* by a sequence of at most $\lceil \log B_{MAX} \rceil$ calls of A[C, d, B, <1>] with different values of B. The binary search procedure can be described in pseudo code as follows:

```
B_MIN ← C;
B_MAX ← C · max {d_ij | (i, j) ∈ C × C};
while not (B_MAX - B_MIN = 1)
{
    B ← ⌈(B_MIN + B_MAX) / 2⌉;
    call A[C, d, B, <1>];
    if (solution = "yes")
        B_MIN ← B;
    else
        B_MAX ← B;
}
B* ← B_MAX;
return (B*);
```

In order to build an optimal length tour, let a candidate partial solution (cps) be a partial tour that can be extended to an optimal tour, i.e. a full tour of minimum length. Clearly, <1> is a cps; hence, there must be at least one city $c \in C \setminus \{1\}$ such that <1, c> is a cps. c can be identified from checking all $c' \in C \setminus \{1\}$ by calling A[C, d, B^*, <1, c'>]; this will involve at most C-1 calls. In general, for each cps $<\pi(1),..., \pi(K)>$ with $K < C$, another cps $<\pi(1),..., \pi(K), \pi(K+1)>$ can be determined by a sequence of at most C-K-1 calls of A such that - given the optimal tour length B^* - an optimal tour can be identified by $1/2 \cdot (C-1)(C-2)$ calls of A.

Having specified the above algorithm, it remains to verify that it is indeed a polynomial reduction. Adding the above numbers yields $1/2 \cdot (C-1)(C-2) + \log B_{MAX}$ as the total number of calls which is $O(C^2 + \log B_{MAX})$. Under the standard encoding scheme the length $LNG_{STD}(I)$ of a TSP instance is $O(C^2 \cdot \log B_{MAX})$. Now we have to show that there is some polynomial p in $LNG_{STD}(I)$ such that the total number of calls of A is $O(p(LNG_{STD}(I)))$, or in other words that there exist such a polynomial p and a constant e such that for all instances $|1/2 \cdot (C-1)(C-2) + \log B_{MAX}| \leq e \cdot |p(C^2 \cdot \log B_{MAX})|$. Choosing $p(x) = x + 1$ and $e = 1$, $|1/2 \cdot (C-1)(C-2) + \log B_{MAX}| \leq |C^2 + \log B_{MAX}| \leq |C^2 \cdot \log B_{MAX} + 1)|$ holds by induction. As all the numbers that occur in a TSP-instance I also appear in the corresponding TSP^{ext}-instance I' and as $MAX_{STD}(I) = MAX_{STD}(I')$, the remaining conditions on a pseudo-polynomial reduction are met, as well.

Now, since TSP polynomially reduces to some **NP**-complete problem, TSP is **NP**-hard.

In total, this shows TSP to be **NP**-equivalent. ■

This *binary search* proof technique allows to show that many (but not all) **NP**-hard optimization problems are actually **NP**-equivalent (cf. Garey, Johnson 1979, p. 117), lending justification to the fact that the main focus of complexity analyses even today is still directed towards the decision problems:

> "In fact, we now observe that the restriction of the basic theory to decision problems has caused no substantial loss of generality, since most often the search problems whose decision problem counterparts have been proved to be **NP**-complete are themselves **NP**-easy and hence of equivalent complexity." (Garey, Johnson 1979, p. 117).

To clarify the binary search technique, consider the following example:

Example Let an instance of the TSP be given by $m = 4$ and $d = d_{ij} = \begin{pmatrix} 3 & 2 & 2 \\ & 2 & 2 \\ & & 4 \end{pmatrix}$.

Then obviously $B_{MIN} = 4$ and $B_{MAX} = m \cdot \max\{d_{ij} \mid (i, j) \in C \times C\} = 16$. So, the proceeding of the algorithm used in the above proof can be illustrated by the trace shown in Table 5.

Table 5

Exemplary Trace of Binary Search Technique

i	$I_i = [m, d, B, \Xi]$	S_i	Remarks
1	4, d, <1>, $\lceil(4+16)/2\rceil = 10$	yes	B_{MAX} decreases
2	4, d, <1>, $\lceil(4+10)/2\rceil = 7$	no	B_{MIN} increases
3	4, d, <1>, $\lceil(7+10)/2\rceil = 9$	yes	B_{MAX} decreases
4	4, d, <1>, $\lceil(7+9)/2\rceil = 8$	yes	B_{MAX}-$B_{MIN} = 1$, $B^* = 8$
5	4, d, <1,2>, 8	no	
6	4, d, <1,3>, 8	yes	
7	4, d, <1,3,2>, 8	yes	
8	4, d, <1,3,2,4>, 8	yes	full tour established

Note that actually the last call of algorithm A is redundant since only one city remains to be added to the tour in iteration $i = 7$. ■

Finally, since problems which are no harder to solve than the **NP**-complete ones but do not belong to **NP** are termed **NP**-easy, one may call problems not belonging to **NP** which are no harder to solve than the strongly **NP**-complete ones strongly **NP**-easy. In the same way, the notion of **NP**-equivalence can be generalized.

Definition 28 A search problem Σ is *NP-easy in the strong sense* (short: *strongly NP-easy*) iff Σ pseudo-polynomially reduces to some strongly **NP**-complete problem.

Let ***sNPE*** denote the class of all strongly **NP**-equivalent problems. ■

Definition 29 A search problem Σ is *NP-equivalent in the strong sense* (short: *strongly NP-hard*) iff Σ is strongly **NP**-hard and strongly **NP**-easy.

Let ***sNPQ*** denote the class of all strongly **NP**-equivalent problems. ■

To summarize the above insights, Table 6 extends the overview of which complexity results one might possibly expect for a given kind of problem. Note that the table remains unchanged with respect to decision problems since the concepts introduced in this section only pertain to search problems.

Table 6

Possible Complexity Results (Extended)

A ...	**may be...**
decision problem...	- polynomially solvable
	- (strongly) **NP**-complete
	- ...
search problem...	- polynomially solvable
	- (strongly) **NP**-hard
	- (strongly) **NP**-easy
	- (strongly) **NP**-equivalent
	- ...

3.4. Approximation Algorithms

Above (cf. Section 3.1) we argued that strong **NP**-completeness or -hardness results preclude - modulo the $\mathbf{P} \neq \mathbf{NP}$ conjecture - the existence not only of polynomial but even of pseudo-polynomial algorithms. For optimization problems this means that no algorithm will be able to solve every instance to optimality in reasonable time. In this situation, one might decide to settle with a less ambitious goal, contenting oneself with algorithms which produce feasible but not necessarily optimal solutions, in the hope that they will work in reasonable time for every instance. Hence, in constructing algorithms for strongly **NP**-hard optimization problems one may either go for optimality, at the risk of long computation times, or for short computation times, at the risk of sub-optimality. The former option constitutes the optimization algorithms; well-known examples are enumeration methods using cutting plane, branch-and-bound, or dynamic programming techniques. The latter option defines the *approximation algorithms*; examples are heuristic construction methods and simulated annealing (cf. Garey, Johnson 1978, pp. 505-506; Papadimitriou, Steiglitz 1982, Chapter 17; Papadimitriou 1994, Chapter 13.1).

Definition 30 Let Ω be an optimization problem. An algorithm that returns for each instance $I_i \in D$ "no" iff F_i is empty and one feasible solution $x \in F_i$ otherwise is called an *approximation algorithm*. A is said to solve Ω. ■

For approximation algorithms, it is usually difficult or even impossible to prove anything about their worst-case performance in the sense of how far away the solutions returned are from optimality. Still, in some cases it is actually possible to derive a bound on the worst-case *relative error*. Let us first formalize this concept.

Definition 31 Let I be an instance of an optimization problem Ω where the objective function f_i is a mapping $f_i: F_i \rightarrow \mathbb{Z}\backslash\{0\}$, x a feasible solution for I, and s an optimal solution for I. Then the *relative error* of x is defined as

$$\Delta_I(x) = \frac{|f(x) - f(s)|}{f(s)} \tag{5}$$

■

Note that the relative error is well-defined only as long as $f(s)$ is nonzero; this assumption will be met by all but the most unusual objective functions, though.

Now, consider an approximation scheme that produces for all instances a solution whose relative error can be bounded by some fixed nonnegative number ε; such an algorithm is called an *ε-approximation algorithm.*

Definition 32 Let denote $\Omega = (D, S)$ an optimization problem and A an approximation algorithm for Ω. Then A is an *ε-approximation algorithm* iff, given an instance $I \in D$ and a fixed number $\varepsilon \in \mathbb{R}_{\geq 0}$, it returns a feasible solution x for I which satisfies

$$\Delta_I(x) \leq \varepsilon \qquad (\forall I \in D) \quad (6)$$

ε is also referred to as *error ratio.*

An ε-approximation algorithm A is called *polynomial-time* (short: *polynomial*) iff for every fixed error ratio ε the time complexity of A is $O(p(LNG_e(I)))$ for some polynomial p in the length $LNG_e(I)$ of the instance and *fully polynomial-time* (short: fully *polynomial*) iff its time complexity is $O(p(LNG_e(I), 1/\varepsilon))$ for some bivariate polynomial p in the variables length $LNG_e(I)$ and $1/\varepsilon$ of the instance. ■

Under certain circumstances it is possible to turn pseudo-polynomial algorithms into fully polynomial ε-approximation algorithms. (The first such results are due to Ibarra, Kim 1975; Horowitz, Sahni 1976; Sahni 1976; Lawler 1977; cf. also the references in Papadimitriou 1994, pp. 323-326) The main idea is to round the numerical values of the given instance appropriately, allowing to speed up the computation, but without letting the cumulative error exceed the error ratio ε. Without demonstrating how to do this, we will now show that a strong **NP**-hardness result has similar implications for the possibility of fully polynomial ε-approximation algorithms as of pseudo-polynomial optimization algorithms (cf. Garey, Johnson 1978, pp. 505-506).

Theorem 2 Let Ω be an optimization problem where for each instance I_i the following properties hold:

(i) The objective function f_i is a mapping $f_i: F_i \rightarrow \mathbb{N}$.

(ii) The optimal solution value $f_i(s)$ is strictly bounded from above by some bivariate polynomial $q(MAX_e(I), LNG_e(I))$.

If there exists a fully polynomial ε-approximation algorithm solving Ω, then there is also a pseudo-polynomial algorithm solving Ω.

Proof: Let A be a fully polynomial ε-approximation algorithm solving Ω, then we can construct a pseudo-polynomial algorithm A' which solves Ω as well. For any instance I, A' calls A with $\varepsilon = 1/(q(LNG_e(I), 1/\varepsilon))$. By (ii), $|f_i(x) - f_i(s)| \leq f_i(s) \cdot \varepsilon = f_i(s) / q(LNG_e(I), 1/\varepsilon) < q(LNG_e(I), 1/\varepsilon) / q(LNG_e(I), 1/\varepsilon) = 1$ holds. By (i), $f_i(x)$ and $f_i(s)$ are integers such that $|f_i(x) - f_i(s)| = 0$ such that A' solves Ω exactly. To see that A' is also pseudo-polynomial, recall that by Definition 32 the time complexity of A is $O(p(LNG_e(I), 1/\varepsilon))$, hence $O(p(LNG_e(I), q(MAX_e(I), LNG_e(I))))$ which is clearly a polynomial in $MAX_e(I)$ and $LNG_e(I)$). ■

Even if the above properties may seem quite restrictive, they are not too much so. The first condition entails no loss of generality as it can always be met by adding some large positive number, thereby changing only the final solution value but not the ratios between different solution values. Most instances of practical relevance will also naturally meet the second condition; even failing to do so they can often be made compliant by appropriate scaling of the numerical values which can be done in polynomial time (Garey, Johnson 1978, p. 506).

Since a strong **NP**-hardness results excludes the possibility for any pseudo-polynomial optimization algorithm, unless **P** = **NP**, Theorem 2 implies that in addition no **NP**-hard optimization can be solved by a fully polynomial ε-approximation algorithm if not **P** = **NP**.

Corollary 11 If an optimization problem Ω is strongly **NP**-hard, then Ω cannot be solved by a fully polynomial ε-approximation algorithm unless **P** = **NP**. ■

One final word on the implication of strong **NP**-hardness for approximation algorithms seems in place. Recall the implication of strong **NP**-hardness for optimization algorithms: Such a result - even under the assumption that **P** ≠ **NP** - does not entail the existence of some fixed bound in terms of input length and magnitude that will be exceeded by the time complexity of any algorithm. It does say, though, that the larger length and magnitude of the attempted instances become, the longer any algorithm will need to solve them. In much the same way, such a result does not imply some fixed error ratio ε which cannot be respected by any polynomial approximation algorithm. Rather than that, it indicates that the smaller ε becomes, the slower any such algorithm must become (Garey, Johnson 1978, p. 506).

4. Summary and Conclusions

In this paper, we developed the central concepts from complexity theory. Do these concepts correspond to anything of interest in the world of operations research? Does complexity theory have interesting, nontrivial, and convincing applications? We believe so. By relating concepts such as (strong) **NP**-completeness and (strong) **NP**-hardness to optimization and feasibility problems, we attempted to demonstrate some useful applications of the former to the latter and thus to reinforce the bridge between computational complexity theory and operations research.

In addition, we pointed out that for number problems, a class to which most problems of interest in operations research belong, a proof of **NP**-completeness or -hardness alone does not allow to regard them as intractable: Under a moderate additional assumption pseudo-polynomial algorithms, whose existence is not ruled out by such a result, become polynomial and will thus be regarded as perfectly practical. Only if a number problem is **NP**-complete or

NP-hard in the strong sense, this possibility is excluded. We also indicated that such results preclude the existence of fully polynomial ε-approximation algorithms.

Finally, we advocated establishing (strong) **NP**-equivalence results for optimization problems because only such a result allows to immediately and finally establish the complexity of an optimization problem as soon as someone is able to answer the notorious question of whether **P** = **NP**. To facilitate the formulation of such results, we addressed a simplified proof technique allowing to show (strong) **NP**-equivalence of optimization problems once their feasibility version is known to be (strongly) **NP**-complete.

Acknowledgement

We would like to thank Wolfgang Brüggemann for several helpful and inspiring discussions. Also we are indebted to Andreas Drexl for his encouraging advice and helpful comments. Finally, we benefited from the valuable suggestions of Wolfgang Thomas whose comments are gratefully acknowledged.

References

AARTS, E. AND J. KORST (1989), *Simulated annealing and Boltzmann machines*, Wiley, Chichester.

AHO, A.V., J.E. HOPCROFT, AND J.D. ULLMAN (1974), *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass.

BADIRU, A.B. (1988), "Towards the standardization of performance measures for project scheduling heuristics", IEEE Transactions on Engineering Management 35, pp. 82-89.

BRINKMANN, K. AND K. NEUMANN (1994), "Heuristic Procedures for Resource-Constrained Project Scheduling with Minimal and Maximal Time Lags: The Minimum Project-Duration and Resource-Levelling Problems", Report WIOR-443, Universität Karlsruhe.

BRÜGGEMANN, W. (1995), *Ausgewählte Probleme der Produktionsplanung - Modellierung, Komplexität und neuere Lösungsmöglichkeiten*, Physica, Heidelberg.

BRÜGGEMANN, W. AND H. JAHNKE (1994), "Remarks on 'Some Extensions of the Discrete Lotsizing and Scheduling Problem' by M. Salomon, L.G. Kroon, R. Kuik, and L.N. van Wassenhove in Management Science 37 (1991), 801-812", Working Paper 32, Institut für Logistik und Transport, Universität Hamburg.

CHURCH, A. (1936), "An unsolvable problem of elementary number theory", American Journal of Mathematics 58, pp. 345-363.

COBHAM, A. (1965), "The intrinsic computational difficulty of functions", in: *Logic, methodology and philosophy of science*, Proceedings of the International Congress 1964, Y. Bar-Hillel (ed.), North-Holland, Amsterdam, pp. 24-30.

COOK, S.A. (1971), "The complexity of theorem proving procedures", Proceedings of the Third Annual ACM Symposium on Theory of Computing (Shaker Heights, Ohio), ACM, New York, pp. 151-158.

COOPER, D.F. (1976), "Heuristics for scheduling resource-constrained projects: An experimental investigation", Management Science 22, pp. 1186-1194.

EDMONDS, J. (1965), "Paths, trees and flowers", Canadian Journal of Mathematics 17, pp. 449-467.

FADLALLA, A., J.R. EVANS, AND M.S. LEVY (1994), "A greedy heuristic for the mean tardiness sequencing problem", Computers in Operations Research 21, pp. 329-336.

GAREY, M.R. AND D.S. JOHNSON (1978), " 'Strong' NP-completeness results; Motivation, Examples, and implications", Journal of the ACM 25, pp. 499-508.

GAREY, M.R. AND D.S. JOHNSON (1979), *Computers and intractability - A guide to the theory of NP-completeness*, W.H. Freeman, San Francisco.

GRAHAM, R.L., E.L. LAWLER, J.K. LENSTRA, AND A.H.G. RINNOOY KAN (1979), "Optimization and approximation in deterministic sequencing and scheduling: A survey", Annals of Discrete Mathematics 5, pp. 287-326.

HALL, N.G., S.P. SETHI, AND C. SRISKANDARAJAH (1991), "On the complexity of generalized due date scheduling problems", European Journal of Operational Research 51, pp. 100-109.

HARTMANIS, J. AND J.E. HOPCROFT (1976), "Independence results in computer science", SIGACT News 8:4, pp. 13-24.

HERMES, H. (1978), *Aufzählbarkeit, Berechenbarkeit, Entscheidbarkeit*, third edition, Springer, Berlin.

HOPCROFT, J.E. AND J.D. ULLMAN (1979), *Introduction to automata theory, languages and computation*, Addison-Wesley, Reading, Mass.

HOROWITZ, E. AND S. SAHNI (1976), "Exact and approximate algorithms for scheduling nonidentical processors", Journal of the ACM 23, pp. 317-327.

IBARRA, O.H. AND C.E. KIM (1975), "Fast approximation algorithms for the knapsack and sum of subset problems", Journal of the ACM 22, pp. 463-468.

JANSEN, K. (1993), "Scheduling with constrained processor allocation for interval orders", Computers in Operations Research 20, pp. 587-595.

JEFFCOAT, D.E. AND R.L. BULFIN (1993), "Simulated Annealing for resource-constrained scheduling", European Journal of Operational Research 70, pp. 43-51.

JOHNSON, D.S. (1983), "The NP-completeness column - An ongoing guide", Journal of Algorithms 4, pp. 189-203.

KARP, R.M. (1972), "Reducibility among combinatorial problems", in: *Complexity of computer computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, New York, pp. 85-103.

KNUTH, D.E. (1974a), "A terminological proposal", SIGACT News 6:1, pp. 12-18.

KNUTH, D.E. (1974b), "Postscript about NP-hard problems", SIGACT News 6:2, pp. 15-16.

LADNER, R.E., N.A. LYNCH, AND A.L. SELMAN (1975), "A comparison of polynomial time reducibilities", Theoretical Computer Science 1, pp. 103-123.

LAURSEN, P.J. (1993), "Simulated annealing for the QAP - Optimal tradeoff between simulation time and solution quality", European Journal of Operational Research 69, pp. 238-243.

LAWLER, E.L. (1977), "Fast approximation schemes for knapsack problems", Proceedings of the 18th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, pp. 206-213.

NEMHAUSER, G.L. AND L.A. WOLSEY (1988), *Integer and combinatorial optimization*, Wiley, New York.

PAPADIMITRIOU, C.H. (1994), *Computational complexity*, Addison-Wesley, Reading, Mass.

PAPADIMITRIOU, C.H. AND K. STEIGLITZ (1982), *Combinatorial optimization: Algorithms and complexity*, Prentice-Hall, Englewood Cliffs (N.J.).

POST, E. (1946), "A variant of a recursively unsolvable problem", Bulletin of the American Mathematical Society 52, pp. 264-268.

CHURCH, A. (1936), "An unsolvable problem of elementary number theory", American Journal of Mathematics 58, pp. 345-363.

COBHAM, A. (1965), "The intrinsic computational difficulty of functions", in: *Logic, methodology and philosophy of science*, Proceedings of the International Congress 1964, Y. Bar-Hillel (ed.), North-Holland, Amsterdam, pp. 24-30.

COOK, S.A. (1971), "The complexity of theorem proving procedures", Proceedings of the Third Annual ACM Symposium on Theory of Computing (Shaker Heights, Ohio), ACM, New York, pp. 151-158.

COOPER, D.F. (1976), "Heuristics for scheduling resource-constrained projects: An experimental investigation", Management Science 22, pp. 1186-1194.

EDMONDS, J. (1965), "Paths, trees and flowers", Canadian Journal of Mathematics 17, pp. 449-467.

FADLALLA, A., J.R. EVANS, AND M.S. LEVY (1994), "A greedy heuristic for the mean tardiness sequencing problem", Computers in Operations Research 21, pp. 329-336.

GAREY, M.R. AND D.S. JOHNSON (1978), " 'Strong' NP-completeness results; Motivation, Examples, and implications", Journal of the ACM 25, pp. 499-508.

GAREY, M.R. AND D.S. JOHNSON (1979), *Computers and intractability - A guide to the theory of NP-completeness*, W.H. Freeman, San Francisco.

GRAHAM, R.L., E.L. LAWLER, J.K. LENSTRA, AND A.H.G. RINNOOY KAN (1979), "Optimization and approximation in deterministic sequencing and scheduling: A survey", Annals of Discrete Mathematics 5, pp. 287-326.

HALL, N.G., S.P. SETHI, AND C. SRISKANDARAJAH (1991), "On the complexity of generalized due date scheduling problems", European Journal of Operational Research 51, pp. 100-109.

HARTMANIS, J. AND J.E. HOPCROFT (1976), "Independence results in computer science", SIGACT News 8:4, pp. 13-24.

HERMES, H. (1978), *Aufzählbarkeit, Berechenbarkeit, Entscheidbarkeit*, third edition, Springer, Berlin.

HOPCROFT, J.E. AND J.D. ULLMAN (1979), *Introduction to automata theory, languages and computation*, Addison-Wesley, Reading, Mass.

HOROWITZ, E. AND S. SAHNI (1976), "Exact and approximate algorithms for scheduling nonidentical processors", Journal of the ACM 23, pp. 317-327.

IBARRA, O.H. AND C.E. KIM (1975), "Fast approximation algorithms for the knapsack and sum of subset problems", Journal of the ACM 22, pp. 463-468.

JANSEN, K. (1993), "Scheduling with constrained processor allocation for interval orders", Computers in Operations Research 20, pp. 587-595.

JEFFCOAT, D.E. AND R.L. BULFIN (1993), "Simulated Annealing for resource-constrained scheduling", European Journal of Operational Research 70, pp. 43-51.

JOHNSON, D.S. (1983), "The NP-completeness column - An ongoing guide", Journal of Algorithms 4, pp. 189-203.

KARP, R.M. (1972), "Reducibility among combinatorial problems", in: *Complexity of computer computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, New York, pp. 85-103.

KNUTH, D.E. (1974a), "A terminological proposal", SIGACT News 6:1, pp. 12-18.

KNUTH, D.E. (1974b), "Postscript about NP-hard problems", SIGACT News 6:2, pp. 15-16.

LADNER, R.E., N.A. LYNCH, AND A.L. SELMAN (1975), "A comparison of polynomial time reducibilities", Theoretical Computer Science 1, pp. 103-123.

LAURSEN, P.J. (1993), "Simulated annealing for the QAP - Optimal tradeoff between simulation time and solution quality", European Journal of Operational Research 69, pp. 238-243.

LAWLER, E.L. (1977), "Fast approximation schemes for knapsack problems", Proceedings of the 18th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, pp. 206-213.

NEMHAUSER, G.L. AND L.A. WOLSEY (1988), *Integer and combinatorial optimization*, Wiley, New York.

PAPADIMITRIOU, C.H. (1994), *Computational complexity*, Addison-Wesley, Reading, Mass.

PAPADIMITRIOU, C.H. AND K. STEIGLITZ (1982), *Combinatorial optimization: Algorithms and complexity*, Prentice-Hall, Englewood Cliffs (N.J.).

POST, E. (1946), "A variant of a recursively unsolvable problem", Bulletin of the American Mathematical Society 52, pp. 264-268.

RUSSELL, R.A. (1986), "A comparison of heuristics for scheduling projects with cash flows and resource restrictions", Management Science 32, pp. 1291-1300.

SAHNI, S. (1976), "Algorithms for scheduling independent tasks", Journal of the ACM 23, pp.116-127.

SCHIRMER, A. (1996a), "New insights on the complexity of resource-constrained project scheduling - A case of single-mode scheduling", Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

SCHIRMER, A. (1996b), "New insights on the complexity of resource-constrained project scheduling - Two cases of multi-mode scheduling", Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

SCHMIDT, G. (1989), "Constraint satisfaction problems in project scheduling", in: *Advances in project scheduling*, R. Slowinski and J. Weglarz (eds.), Elsevier, Amsterdam.

SCHRIJVER, A. (1986), *Theory of integer and linear programming*, Wiley, Chichester.

TURING, A. (1936), "On computable numbers, with an application to the Entscheidungs-problem", Proceedings of the London Mathematical Society Series 2 42, pp. 230-265 and 43, pp. 544-546.

ULLMAN, J.D. (1976), "Complexity of sequencing problems", in: *Computer and job-shop scheduling theory*, Coffman, E.G., Jr. (ed.), Wiley, New York, pp. 139-164.