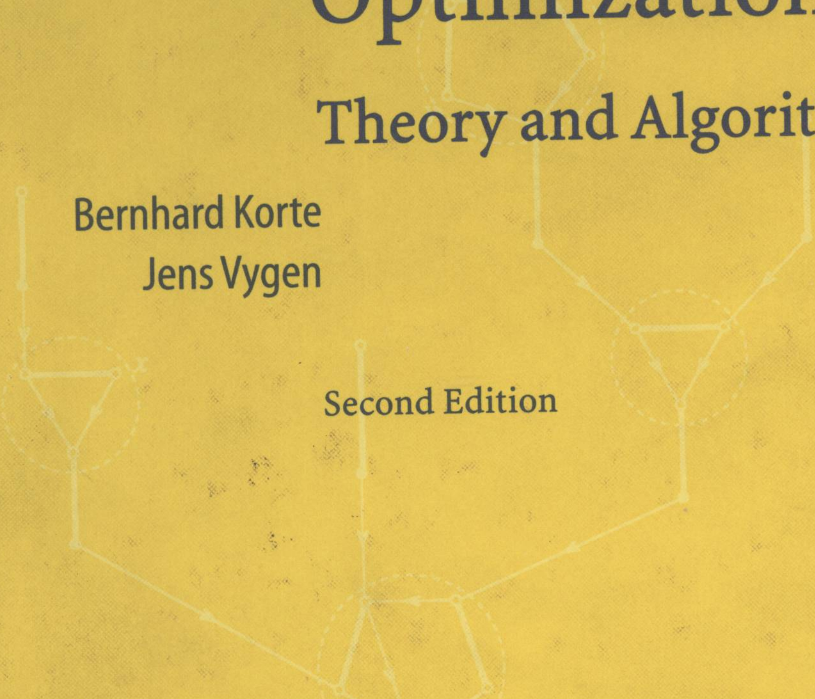
21 Algorithms and Combinatorics

Combinatorial Optimization

Theory and Algorithms

Bernhard Korte

Jens Vygen



Second Edition

Springer

O224
K85
E.2

Bernhard Korte
Jens Vygen

Combinatorial Optimization

Theory and Algorithms

Second Edition

Springer

Bernhard Korte
Jens Vygen
Research Institute for Discrete Mathematics
University of Bonn
Lennéstraße 2
53113 Bonn, Germany
e-mail: dm@or.uni-bonn.de
vygen@or.uni-bonn.de

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Korte, Bernhard:
Combinatorial optimization: theory and algorithms / Bernhard Korte; Jens Vygen. - 2. ed. -
Berlin; Heidelberg; New York; Barcelona; Hong Kong; London; Milan; Paris; Tokyo:
Springer, 2002
(Algorithms and combinatorics; 21)
ISBN 3-540-43154-3

Mathematics Subject Classification (2000):
90C27, 68R10, 05C85, 68Q25

ISSN 0937-5511
ISBN 3-540-43154-3 Springer-Verlag Berlin Heidelberg New York
ISBN 3-540-67226-5 1. ed. Springer-Verlag Berlin Heidelberg New York

Typeset in LaTeX by the authors. Edited and reformatted by Kurt Mattes, Heidelberg, using the MathTime fonts and a Springer LaTeX macro package.
Printed on acid-free paper SPIN 10859590 46/3142LK - 5 4 3 2 1 0

Algorithms and Combinatorics 21

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Preface to the Second Edition

It was more than a surprise to us that the first edition of this book already went out of print about a year after its first appearance. We were flattered by the many positive and even enthusiastic comments and letters from colleagues and the general readership. Several of our colleagues helped us in finding typographical and other errors. In particular, we thank Ulrich Brenner, András Frank, Bernd Gärtner and Rolf Möhring. Of course, all errors detected so far have been corrected in this second edition, and references have been updated.

Moreover, the first preface had a flaw. We listed all individuals who helped us in preparing this book. But we forgot to mention the institutional support, for which we make amends here.

It is evident that a book project which took seven years benefited from many different grants. We would like to mention explicitly the bilateral Hungarian-German Research Project, sponsored by the Hungarian Academy of Sciences and the Deutsche Forschungsgemeinschaft, two Sonderforschungsbereiche (special research units) of the Deutsche Forschungsgemeinschaft, the Ministère Français de la Récherche et de la Technologie and the Alexander von Humboldt Foundation for support via the Prix Alexandre de Humboldt, and the Commission of the European Communities for participation in two projects DONET. Our most sincere thanks go to the Union of the German Academies of Sciences and Humanities and to the Northrhine-Westphalian Academy of Sciences. Their long-term project "Discrete Mathematics and Its Applications" supported by the German Ministry of Education and Research (BMBF) and the State of Northrhine-Westphalia was of decisive importance for this book.

Bonn, October 2001 *Bernhard Korte and Jens Vygen*

Preface to the First Edition

Combinatorial optimization is one of the youngest and most active areas of discrete mathematics, and is probably its driving force today. It became a subject in its own right about 50 years ago.

This book describes the most important ideas, theoretical results, and algorithms in combinatorial optimization. We have conceived it as an advanced graduate text which can also be used as an up-to-date reference work for current research. The book includes the essential fundamentals of graph theory, linear and integer programming, and complexity theory. It covers classical topics in combinatorial optimization as well as very recent ones. The emphasis is on theoretical results and algorithms with provably good performance. Applications and heuristics are mentioned only occasionally.

Combinatorial optimization has its roots in combinatorics, operations research, and theoretical computer science. A main motivation is that thousands of real-life problems can be formulated as abstract combinatorial optimization problems. We focus on the detailed study of classical problems which occur in many different contexts, together with the underlying theory.

Most combinatorial optimization problems can be formulated naturally in terms of graphs and as (integer) linear programs. Therefore this book starts, after an introduction, by reviewing basic graph theory and proving those results in linear and integer programming which are most relevant for combinatorial optimization.

Next, the classical topics in combinatorial optimization are studied: minimum spanning trees, shortest paths, network flows, matchings and matroids. Most of the problems discussed in Chapters 6–14 have polynomial-time ("efficient") algorithms, while most of the problems studied in Chapters 15–21 are *NP*-hard, i.e. a polynomial-time algorithm is unlikely to exist. In many cases one can at least find approximation algorithms that have a certain performance guarantee. We also mention some other strategies for coping with such "hard" problems.

This book goes beyond the scope of a normal textbook on combinatorial optimization in various aspects. For example we cover the equivalence of optimization and separation (for full-dimensional polytopes), $O(n^3)$-implementations of matching algorithms based on ear-decompositions, Turing machines, the Perfect Graph Theorem, *MAXSNP*-hardness, the Karmarkar-Karp algorithm for bin packing, recent approximation algorithms for multicommodity flows, survivable network de-sign and the Euclidean traveling salesman problem. All results are accompanied by detailed proofs.

Of course, no book on combinatorial optimization can be absolutely comprehensive. Examples of topics which we mention only briefly or do not cover at all are tree-decompositions, separators, submodular flows, path-matchings, delta-matroids, the matroid parity problem, location and scheduling problems, nonlinear problems, semidefinite programming, average-case analysis of algorithms, advanced data structures, parallel and randomized algorithms, and the theory of probabilistically checkable proofs (we cite the *PCP* Theorem without proof).

At the end of each chapter there are a number of exercises containing additional results and applications of the material in that chapter. Some exercises which might be more difficult are marked with an asterisk. Each chapter ends with a list of references, including texts recommended for further reading.

This book arose from several courses on combinatorial optimization and from special classes on topics like polyhedral combinatorics or approximation algorithms. Thus, material for basic and advanced courses can be selected from this book.

We have benefited from discussions and suggestions of many colleagues and friends and – of course – from other texts on this subject. Especially we owe sincere thanks to András Frank, László Lovász, András Recski, Alexander Schrijver and Zoltán Szigeti. Our colleagues and students in Bonn, Christoph Albrecht, Ursula Bünnagel, Thomas Emden-Weinert, Mathias Hauptmann, Sven Peyer, Rabe von Randow, André Rohe, Martin Thimm and Jürgen Werber, have carefully read several versions of the manuscript and helped to improve it. Last, but not least we thank Springer Verlag for the most efficient cooperation.

Bonn, January 2000 *Bernhard Korte and Jens Vygen*

Table of Contents

1. Introduction

Let us start with two examples.

A company has a machine which drills holes into printed circuit boards. Since it produces many of these boards it wants the machine to complete one board as fast as possible. We cannot optimize the drilling time but we can try to minimize the time the machine needs to move from one point to another. Usually drilling machines can move in two directions: the table moves horizontally while the drilling arm moves vertically. Since both movements can be done simultaneously, the time needed to adjust the machine from one position to another is proportional to the maximum of the horizontal and the vertical distance. This is often called the L_∞-distance. (Older machines can only move either horizontally or vertically at a time; in this case the adjusting time is proportional to the L_1-distance, the sum of the horizontal and the vertical distance.)

An optimum drilling path is given by an ordering of the hole positions $p_1, \dots, p_n$ such that $\sum_{i=1}^{n-1} d(p_i, p_{i+1})$ is minimum, where d is the L_∞-distance: for two points $p = (x, y)$ and $p' = (x', y')$ in the plane we write $d(p, p') := \max\{|x - x'|, |y - y'|\}$. An order of the holes can be represented by a permutation, i.e. a bijection $\pi : \{1, \dots, n\} \to \{1, \dots, n\}$.

Which permutation is best of course depends on the hole positions; for each list of hole positions we have a different problem instance. We say that one instance of our problem is a list of points in the plane, i.e. the coordinates of the holes to be drilled. Then the problem can be stated formally as follows:

DRILLING PROBLEM	
Instance:	A set of points $p_1, \dots, p_n \in \mathbb{R}^2$.
Task:	Find a permutation $\pi : \{1, \dots, n\} \to \{1, \dots, n\}$ such that $\sum_{i=1}^{n-1} d(p_{\pi(i)}, p_{\pi(i+1)})$ is minimum.

We now explain our second example. We have a set of jobs to be done, each having a specified processing time. Each job can be done by a subset of the employees, and we assume that all employees who can do a job are equally efficient. Several employees can contribute to the same job at the same time, and one employee can contribute to several jobs (but not at the same time). The objective is to get all jobs done as early as possible.

In this model it suffices to prescribe for each employee how long he or she should work on which job. The order in which the employees carry out their jobs is not important, since the time when all jobs are done obviously depends only on the maximum total working time we have assigned to one employee. Hence we have to solve the following problem:

JOB ASSIGNMENT PROBLEM

Instance:	A set of numbers $t_1, \dots, t_n \in \mathbb{R}_+$ (the processing times for n jobs), a number $m \in \mathbb{N}$ of employees, and a nonempty subset $S_i \subseteq \{1, \dots, m\}$ of employees for each job $i \in \{1, \dots, n\}$.
Task:	Find numbers $x_{ij} \in \mathbb{R}_+$ for all $i = 1, \dots, n$ and $j \in S_i$ such that $\sum_{j \in S_i} x_{ij} = t_i$ for $i = 1, \dots, n$ and $\max_{j \in \{1,\dots,m\}} \sum_{i:j \in S_i} x_{ij}$ is minimum.

These are two typical problems arising in combinatorial optimization. How to model a practical problem as an abstract combinatorial optimization problem is not described in this book; indeed there is no general recipe for this task. Besides giving a precise formulation of the input and the desired output it is often important to ignore irrelevant components (e.g. the drilling time which cannot be optimized or the order in which the employees carry out their jobs).

Of course we are not interested in a solution to a particular drilling problem or job assignment problem in some company, but rather we are looking for a way how to solve all problems of these types. We first consider the DRILLING PROBLEM.

1.1 Enumeration

How can a solution to the DRILLING PROBLEM look like? There are infinitely many instances (finite sets of points in the plane), so we cannot list an optimum permutation for each instance. Instead, what we look for is an algorithm which, given an instance, computes an optimum solution. Such an algorithm exists: Given a set of n points, just try all possible $n!$ orders, and for each compute the L_∞-length of the corresponding path.

There are different ways of formulating an algorithm, differing mostly in the level of detail and the formal language they use. We certainly would not accept the following as an algorithm: "Given a set of n points, find an optimum path and output it." It is not specified at all how to find the optimum solution. The above suggestion to enumerate all possible $n!$ orders is more useful, but still it is not clear how to enumerate all the orders. Here is one possible way:

We enumerate all n-tuples of numbers $1, \dots, n$, i.e. all n^n vectors of $\{1, \dots, n\}^n$. This can be done similarly to counting: we start with $(1, \dots, 1, 1)$, $(1, \dots, 1, 2)$ up to $(1, \dots, 1, n)$ then switch to $(1, \dots, 1, 2, 1)$, and so on. At each step we increment the last entry unless it is already n, in which case we go back to the last entry that is smaller than n, increment it and set all subsequent entries to 1.

This technique is sometimes called backtracking. The order in which the vectors of $\{1, \ldots, n\}^n$ are enumerated is called the lexicographical order:

Definition 1.1. *Let $x, y \in \mathbb{R}^n$ be two vectors. We say that a vector x is* **lexicographically smaller** *than y if there exists an index $j \in \{1, \ldots, n\}$ such that $x_i = y_i$ for $i = 1, \ldots, j-1$ and $x_j < y_j$.*

Knowing how to enumerate all vectors of $\{1, \ldots, n\}^n$ we can simply check for each vector whether its entries are pairwise distinct and, if so, whether the path represented by this vector is shorter than the best path encountered so far.

Since this algorithm enumerates n^n vectors it will take at least n^n steps (in fact, even more). This is not best possible. There are only $n!$ permutations of $\{1, \ldots, n\}$, and $n!$ is significantly smaller than n^n. (By Stirling's formula $n! \approx \sqrt{2\pi n}\frac{n^n}{e^n}$.) We shall show how to enumerate all paths in approximately $n^2 \cdot n!$ steps. Consider the following algorithm which enumerates all permutations in lexicographical order:

```
PATH ENUMERATION ALGORITHM

Input:   A natural number $n \geq 3$. A set $\{p_1, \ldots, p_n\}$ of points in the plane.

Output:  A permutation $\pi^* : \{1, \ldots, n\} \to \{1, \ldots, n\}$ with
         $cost(\pi^*) := \sum_{i=1}^{n-1} d(p_{\pi^*(i)}, p_{\pi^*(i+1)})$ minimum.

①  Set $\pi(i) := i$ and $\pi^*(i) := i$ for $i = 1, \ldots, n$. Set $i := n - 1$.

②  Let $k := \min(\{\pi(i) + 1, \ldots, n + 1\} \setminus \{\pi(1), \ldots, \pi(i - 1)\})$.

③  If $k \leq n$ then:
       Set $\pi(i) := k$.
       If $i = n$ and $cost(\pi) < cost(\pi^*)$ then set $\pi^* := \pi$.
       If $i < n$ then set $\pi(i + 1) := 0$ and $i := i + 1$.
    If $k = n + 1$ then set $i := i - 1$.
    If $i \geq 1$ then go to ②.
```

Starting with $(\pi(i))_{i=1,\ldots,n} = (1, 2, 3, \ldots, n-1, n)$ and $i = n-1$, the algorithm finds at each step the next possible value of $\pi(i)$ (not using $\pi(1), \ldots, \pi(i-1)$). If there is no more possibility for $\pi(i)$ (i.e. $k = n + 1$), then the algorithm decrements i (backtracking). Otherwise it sets $\pi(i)$ to the new value. If $i = n$, the new permutation is evaluated, otherwise the algorithm will try all possible values for $\pi(i + 1), \ldots, \pi(n)$ and starts by setting $\pi(i + 1) := 0$ and incrementing i.

So all permutation vectors $(\pi(1), \ldots, \pi(n))$ are generated in lexicographical order. For example, the first iterations in the case $n = 6$ are shown below:

$$\begin{array}{llll}
 & \pi := (1,2,3,4,5,6), & i := 5 & \\
k := 6, & \pi := (1,2,3,4,6,0), & i := 6 & \\
k := 5, & \pi := (1,2,3,4,6,5), & & cost(\pi) < cost(\pi^*)? \\
k := 7, & & i := 5 & \\
k := 7, & & i := 4 & \\
k := 5, & \pi := (1,2,3,5,0,5), & i := 5 & \\
k := 4, & \pi := (1,2,3,5,4,0), & i := 6 & \\
k := 6, & \pi := (1,2,3,5,4,6), & & cost(\pi) < cost(\pi^*)?
\end{array}$$

Since the algorithm compares the cost of each path to π^*, the best path encountered so far, it indeed outputs the optimum path. But how many steps will this algorithm perform? Of course, the answer depends on what we call a single step. Since we do not want the number of steps to depend on the actual implementation we ignore constant factors. In any reasonable computer, ① will take at least $2n+1$ steps (this many variable assignments are done) and at most cn steps for some constant c. The following common notation is useful for ignoring constant factors:

Definition 1.2. *Let $f, g : D \to \mathbb{R}_+$ be two functions. We say that f is $O(g)$ (and sometimes write $f = O(g)$) if there exist constants $\alpha, \beta > 0$ such that $f(x) \le \alpha g(x) + \beta$ for all $x \in D$. If $f = O(g)$ and $g = O(f)$ we also say that $f = \Theta(g)$ (and of course $g = \Theta(f)$). In this case, f and g have the same* **rate of growth**.

Note that the use of the equation sign in the O-notation is not symmetric. To illustrate this definition, let $D = \mathbb{N}$, and let $f(n)$ be the number of elementary steps in ① and $g(n) = n$ ($n \in \mathbb{N}$). Clearly we have $f = O(g)$ (in fact $f = \Theta(g)$) in this case; we say that ① takes $O(n)$ time (or linear time). A single execution of ③ takes a constant number of steps (we speak of $O(1)$ time or constant time) except in the case $k \le n$ and $i = n$; in this case the cost of two paths have to be compared, which takes $O(n)$ time.

What about ②? A naive implementation, checking for each $j \in \{\pi(i)+1, \dots, n\}$ and each $h \in \{1, \dots, i-1\}$ whether $j = \pi(h)$, takes $O((n-\pi(i))i)$ steps, which can be as big as $\Theta(n^2)$. A better implementation of ② uses an auxiliary array indexed by $1, \dots, n$:

```
②   For $j := 1$ to $n$ do $aux(j) := 0$.
    For $j := 1$ to $i-1$ do $aux(\pi(j)) := 1$.
    Set $k := \pi(i) + 1$.
    While $k \le n$ and $aux(k) = 1$ do $k := k + 1$.
```

Obviously with this implementation a single execution of ② takes only $O(n)$ time. Simple techniques like this are usually not elaborated in this book; we assume that the reader can find such implementations himself.

Having computed the running time for each single step we now estimate the total amount of work. Since the number of permutations is $n!$ we only have to estimate the amount of work which is done between two permutations. The counter i might move back from n to some index i' where a new value $\pi(i') \le n$ is found. Then it moves forward again up to $i = n$. While the counter i is constant each of ② and ③ is performed once. So the total amount of work between two permutations