CHAPTER 28

Combinatorial Optimization

M. GRÖTSCHEL

Konrad-Zuse-Zentrum für Informationstechnik, Heilbronner Str. 10, D-10711 Berlin-Wilmersdorf, Germany

L. LOVÁSZ

Department of Computer Science, Yale University, New Haven, CT 06250, USA

Contents

HANDBOOK OF COMBINATORICS
Edited by R. Graham, M. Grötschel and L. Lovász

1. Introduction

Optimizing means finding the maximum or minimum of a certain function, called the objective function, defined on some domain. Classical theories of optimization (differential calculus, variational calculus, optimal control theory) deal with the case when this domain is infinite. From this angle, the subject of combinatorial optimization, where the domain is typically finite, might seem trivial: it is easy to say that "we choose the best from this finite number of possibilities". But, of course, these possibilities may include all trees on n nodes, or all Hamilton circuits of a complete graph, and then listing all possibilities to find the best among them is practically hopeless even for instances of very moderate size. In the framework of complexity theory (chapter 29), we want to find the optimum in polynomial time. For this (or indeed, to do better than listing all solutions) the special structure of the domain must be exploited.

Often, when the objective function is too wild, the constraints too complicated, or the problem size too large, it is impossible to find an optimum solution. This is quite frequently not just a practical experience; mathematics and computer science have developed theories to make intuitive assertions about the difficulty of certain problems precise. Foremost of these is the theory of NP-completeness (see chapter 29).

In cases when optimum solutions are too hard to find, algorithms (so-called *heuristics*) can often be designed that produce approximately optimal solutions. It is important that these suboptimal solutions have a guaranteed quality; e.g., for a given maximization problem, the value of the heuristic solution is at least 90% of the optimum value for every input.

While not so apparent on the surface, of equal importance are algorithms, called *dual heuristics*, that provide (say for a maximization problem again) upper bounds on the optimum value. Dual heuristics typically solve so-called *relaxations* of the original problem, i.e., optimization problems that are obtained by dropping or relaxing the constraints so as to make the problem easier to solve. Bounds computed this way are important in the analysis of heuristics, since (both theoretically and in practice) one compares the value obtained by a heuristic with the value obtained by the dual heuristic (instead of the true optimum, which is unknown). Relaxations also play an important role in general solution schemes for hard problems like branch-and-bound.

The historical roots of combinatorial optimization lie in problems in economics: the planning and management of operations and the efficient use of resources. Soon more technical applications came into focus and were modelled as combinatorial optimization problems, such as sequencing of machines, scheduling of production, design and layout of production facilities. Today we see that discrete optimization problems abound everywhere. We encounter them in areas such as portfolio selection, capital budgeting, design of marketing campaigns, investment planning and facility location, political districting, gene sequencing, classification of plants and animals, the design of new molecules, the determination of ground states, layout of survivable and cost-coefficient communication networks, positioning of satellites, design and production of VLSI circuits and printed circuit boards, sizing of truck fleets and transportation planning, layout of mass transportation systems and scheduling of buses and trains, assignment of workers to jobs such as drivers to buses and airline crew scheduling, design of unbreakable codes, etc. The list of applications seems endless; even in areas like sports, archeology or psychology, combinatorial optimization is used to answer important questions. We refer to chapter 35 for a detailed description of several real-world examples.

There are basically two ways of presenting "combinatorial optimization": by problems or by methods. Since combinatorial optimization problems abound in this Handbook and many chapters deal with particular problems, discuss their practical applications and algorithmic solvability, we organize our material according to the second approach. We will describe the fundamental algorithmic techniques in detail and illustrate them on problems to which these methods have been applied successfully.

Some important aspects of combinatorial optimization algorithms we can only touch in this chapter. One such aspect is *parallelism*. There is no doubt that the computers of the future will be parallel machines. A systematic treatment of parallel algorithms is difficult since there are many computer architectures, based on different principles, and each architecture leads to a different model of parallel computational complexity. One very general model of parallel computation is described in chapter 29.

Another important aspect is the *on-line* solution of combinatorial problems. We treat here "static" problems, where all data are known before the optimization algorithm is called. In many practical situations, however, data come in one by one, and decisions must be made before the next piece of data arrives. The theoretical modelling of such situations is difficult, and we refrain from discussing the many possibilities.

A third disclaimer of this type is that we focus on the *worst-case* analysis of algorithms. From a practical point of view, *average-case analysis*, i.e., the analysis of algorithms on random input, would be more important; but for a mathematical treatment of this, one has to make assumptions about the input distribution, and except for very simple cases, such assumptions are extremely difficult to justify and lead to unresolvable controversies.

Finally, we should mention the increasing significance of *randomization*. It is pointed out in chapter 29 that randomization should not be confused with the average case analysis of algorithms. Randomization is often used in conjunction with determinant methods, and it is an important tool in avoiding "traps" (degeneracies), in reaching tricky "corners" of the domain, and in many other situations. We do discuss several of these methods; other issues like *derandomization* (transforming randomized algorithms into deterministic ones), or the reliability of random number generators will, however, not be treated here.

Combinatorial optimization problems typically have as inputs methods based on determinant computation, combinatorial structures and numbers (e.g., a graph with weights on the edges). In the Turing machine model, both are encoded as

0–1 strings; but in the RAM machine model it is natural to consider the input as a sequence of integers. If the input also involves a combinatorial structure, then the combinatorial structure can be considered as a set of 0s and 1s. We denote by $\langle a \rangle$ the number of bits in the binary representation of the integer a; for a matrix $A = (a_{ij})$ of integers, we define $\langle A \rangle = \sum \langle a_{ij} \rangle$.

An algorithm (with an input sequence of integers $a_1, \ldots, a_n\rangle$ runs in *polynomial time* (short: is *polynomial*) if it can be implemented on the RAM machine so that the number of bit-operations performed is bounded by a polynomial in the number of input bits $\langle a_1 \rangle + \cdots + \langle a_n \rangle$. Considering the input as a set of numbers allows two important versions of this notion.

An algorithm (with an input sequence of integers $a_1, \ldots, a_n$) is *pseudopolynomial*, if it can be implemented on the RAM machine so that the number of bit-operations performed is bounded by a polynomial in $|a_1| + \cdots + |a_n|$. (This can also be defined on the Turing machine model: it corresponds to polynomial running time where the encoding of an integer a by a string of a 1s is used. Thus a pseudopolynomial algorithm is also called *polynomial in the unary encoding*.) Clearly, every polynomial algorithm is pseudopolynomial, but not the other way around: testing primality in the trivial way by searching through all smaller integers is pseudopolynomial but not polynomial.

An algorithm (with input integers $a_1, \ldots, a_n$) is *strongly polynomial* if it can be implemented on the RAM machine in $O(n^c)$ steps with numbers of $O((\langle a_1 \rangle + \cdots + \langle a_n \rangle)^c)$ digits for some $c > 0$. Clearly every strongly polynomial algorithm is polynomial, but not the other way around: e.g., the Euclidean algorithm is polynomial but not strongly polynomial. On the other hand, Kruskal's algorithm for shortest spanning trees is strongly polynomial.

Further reading. Bachem et al. (1983), Ford and Fulkerson (1962), Gondran and Minoux (1979), Grötschel et al. (1988), Lawler et al. (1985), Nemhauser et al. (1989), Nemhauser and Wolsey (1988), Schrijver (1986).

2. The greedy algorithm

Kruskal's algorithm and matroids. The most natural principle we can try to build an optimization algorithm on is *greediness*: building up the solution by making the best choice locally. As the most important example of an optimization problem where this simple idea works, let us recall Kruskal's algorithm for a shortest spanning tree from chapters 2, 9 and 40.

Given a connected graph G with n nodes, and a length function $c: E(G) \rightarrow \mathbb{Z}$, we want to find a shortest spanning tree (where the length of a subgraph of G is defined as the sum of the lengths of its edges). The algorithm constructs the tree by finding its edges $e_1, e_2, \ldots, e_{n-1}$ one by one:

- e_1 is an edge of minimum length;
- e_k is an edge such that $e_k \not\in \{e_1, \ldots, e_{k-1}\}$, $\{e_1, \ldots, e_k\}$ is a forest and

$$c(e_k) = \min\{c(e) \mid e \not\in \{e_1, \ldots, e_{k-1}\} \text{ and } \{e_1, \ldots, e_{k-1}, e\} \text{ is a forest}\} .$$

Each step of the algorithm makes locally the best choice: this is why it is called a *greedy* algorithm. Kruskal's theorem (going back actually to Borůvka in 1926; see Graham and Hell (1985) for an account of its history) asserts that $\{e_i: 1 \leq i \leq n-1\}$ is a shortest spanning tree of G, i.e., *the spanning tree constructed by the greedy algorithm is optimal.*

The graph structure plays little role in the algorithm; the only information about the graph used is that "$\{e_1, \ldots, e_{k-1}, e\}$ is a forest". In fact, this observation is one of the possible routes to the notion of *matroids*.

Let S be a finite set, $c: S \rightarrow \mathbb{Z}_+$, a cost function on S, and $\mathcal{F} \subseteq 2^S$ a *hereditary family* (independence system) of subsets of S, i.e., a set of subsets such that $X \in \mathcal{F}$ and $Y \subseteq X$ implies $Y \in \mathcal{F}$. The goal is to find $\max\{c(X) = \sum_{e \in X} c(e) \mid X \in \mathcal{F}\}$. In this more general setting, the greedy algorithm can still be easily formulated. It constructs a maximal set X by finding its elements $e_1, e_2, \ldots$ one by one, as follows:

- e_1 is an element of maximum cost in $\bigcup_{X \in \mathcal{F}} X$,
- e_k is defined such that $e_k \not\in \{e_1, \ldots, e_{k-1}\}$, $\{e_1, \ldots, e_k\} \in \mathcal{F}$ and

$$c(e_k) = \max\{c(e) \mid e \not\in \{e_1, \ldots, e_{k-1}\}, \text{ and } \{e_1, \ldots, e_{k-1}, e\} \in \mathcal{F}\}\,.$$

The algorithm terminates when no such element exists. We call a set $X_{\mathrm{gr}} := \{e_1, \ldots, e_r\}$ obtained by this algorithm a *greedy solution*, and let X_{opt} denote an optimum solution to our problem. In chapter 9 it is shown that the greedy solution is optimal for every cost function if and only if $(E, \mathcal{F})$ is a matroid.

There are other problems where the greedy algorithm (with an appropriate interpretation) gives an optimum solution. The notion of *greedoids* (see chapter 9, or Korte et al. 1991) is an attempt to describe a general class of such problems. Further examples are polymatroids (see chapter 11), coloring of various classes of perfect graphs (see chapter 4) etc. In fact, an optimization model where the greedy solution is optimal was described by Monge in 1781!

Greedy heuristic. But in most optimization problems, greed does not pay: the greedy solution is not optimal in general. We may still use, however, a greedy algorithm as a heuristic, to obtain a "reasonable" solution (and in practice it is very often used indeed).

We measure the quality of the heuristic by comparing the value of the objective function at the heuristic solution with its value at the optimum solution. To be precise, consider, say, a minimization problem. For convenience, let us assume that every instance has a positive optimum objective function value v_{opt}, say. For a given instance, let v_{heur} denote the objective value achieved by a given heuristic (if the heuristic includes free choices at certain points, we define v_{heur} as the value achieved by the worst possible choices). We define the *performance ratio* of a heuristic as the supremum of $v_{\mathrm{heur}}/v_{\mathrm{opt}}$ over all problem instances. The *asymptotic performance ratio* is the lim sup of this ratio, assuming that $v_{\mathrm{opt}} \rightarrow \infty$. For a maximization problem, we replace this quotient by $v_{\mathrm{opt}}/v_{\mathrm{heur}}$.

Rank quotients. Consider a hereditary family $\mathcal{F} \subseteq 2^E$ and a weight function $c: E \to \mathbb{Z}_+$ on E again. We want to find a maximum weight member of $\mathcal{F}$. Let the greedy algorithm, just as above, give a set X_{gr}, and let X_{opt} denote an optimum solution.

Since we are maximizing, trivially

$$c(X_{\mathrm{gr}}) \leqslant c(X_{\mathrm{opt}}) .$$

Define, for $X \subseteq E$, the *upper rank of X* by

$$r(X) = \max\{|Y| : Y \subseteq X, Y \in \mathcal{F}\} .$$

We also define the *lower rank* of X by

$$\rho(X) = \min\{|Y| : Y \subseteq X, Y \in \mathcal{F}, \nexists U \in \mathcal{F} \text{ with } Y \subset U \subseteq X\} .$$

Note that matroids are just those hereditary families with $r = \rho$. In general, define the *rank quotient* of $(E, \mathcal{F})$ by

$$\gamma_{\mathcal{F}} = \min\left\{\frac{\rho(X)}{r(X)} : X \subseteq E, r(X) > 0\right\} .$$

Note that $\gamma_{\mathcal{F}}$ is just the worst ratio between $c(X_{\mathrm{gr}})$ and $c(X_{\mathrm{opt}})$ for 0–1 valued weight functions c. The following theorem of Jenkyns (1976) and Korte and Hausmann (1978) gives a performance guarantee for the greedy algorithm by showing that 0–1 weightings are the worst case.

Theorem 2.1. *For every hereditary family $(E, \mathcal{F})$ and every weight function $c: E \to \mathbb{Z}_+$, we have*

$$c(X_{\mathrm{gr}}) \geqslant \gamma_{\mathcal{F}} c(X_{\mathrm{opt}}) .$$

As an application, consider the greedy heuristic for the matching problem. Here G is a graph, $E = E(G)$, and $\mathcal{F}$ consists of all matchings, i.e., sets of edges with no common endnode. We claim that $\gamma_{\mathcal{F}} \geqslant \frac{1}{2}$. In fact, if $X \subseteq E$ and M is a smallest non-extendible matching in X then the $2|M|$ endpoints of the edges in M cover all edges in X, and hence a maximum matching in X cannot have more than $2|M|$ edges. Thus we obtain that *for every weighting, the greedy algorithm for the matching problem has performance ratio at most* 2.

Greedy blocking sets. We discuss a greedy heuristic with a somewhat more involved analysis. Let $(V, \mathcal{H})$ be a hypergraph. A *blocking set* or *cover* of the hypergraph is a subset $S \subseteq V$ that meets (blocks) every member of $\mathcal{H}$. Let S_{opt} denote a blocking set with a minimum number of elements, and define the *covering* (*or blocking*) *number* by $\tau(\mathcal{H}) := |S_{\mathrm{opt}}|$. (To compute $\tau(\mathcal{H})$ is NP-hard in general; cf. also chapters 7 and 24.)

A *greedy blocking set* S_{gr} is constructed as follows. Choose a vertex v_1 of maximum degree. If $v_1, \ldots, v_k$ have been selected, choose v_{k+1} to block as many members of $\mathcal{H}$ not blocked by $v_1, \ldots, v_k$ as possible. We stop when all members of $\mathcal{H}$ are blocked.

Concerning the performance of this algorithm, we have the following bound (Johnson 1974, Stein 1974, Lovász 1975).

Theorem 2.2. *Let $\mathcal{H}$ be a hypergraph with maximum degree Δ. Then for the size of any greedy blocking set S_{gr},*

$$|S_{\mathrm{gr}}| \leq \left(1+\frac{1}{2}+\cdots+\frac{1}{\Delta}\right)\tau(\mathcal{H}) .$$

(The "error factor" on the right-hand side is less than $1+\ln \Delta$. It is easy to see that the ratio $1+1/2+\cdots+1/\Delta$ cannot be improved.)

Proof. Let k_i denote the number of vertices in S_{gr} selected in the phase when we were able to block exactly i new edges at a time. Let $\mathcal{H}_i$ denote the set of edges first blocked in this phase. So $|\mathcal{H}_i| = ik_i$, and

$$|S_{\mathrm{gr}}| = k_\Delta + k_{\Delta-1} + \cdots + k_1 .$$

Now consider the optimum blocking set S_{opt}. Every vertex in S_{opt} (in fact, every vertex of $\mathcal{H}$), blocks at most i edges from $\mathcal{H}_i \cup \mathcal{H}_{i-1} \cup \cdots \cup \mathcal{H}_1$, since a vertex blocking more from this set should have been included in the greedy blocking set before phase i. Hence

$$|S_{\mathrm{opt}}| \geq \frac{1}{i}\,(ik_i + (i-1)k_{i-1} + \cdots + k_1) . \tag{1}$$

Multiplying this inequality by $1/(i+1)$ for $i=1,\ldots,\Delta-1$, and by 1 for $i=\Delta$, and then adding up the resulting inequalities, we obtain that

$$\left(1+\frac{1}{2}+\cdots+\frac{1}{\Delta}\right)|S_{\mathrm{opt}}| \geq k_1 + \cdots + k_\Delta = |S_{\mathrm{gr}}| . \qquad \square$$

Note that in this proof, we do not directly compare $|S_{\mathrm{gr}}|$ with $|S_{\mathrm{opt}}|$; the latter is not available. Rather, we use a family (1) of lower bounds on $|S_{\mathrm{opt}}|$. For each i, $(1/i)(\mathcal{H}_1 \cup \cdots \cup \mathcal{H}_i)$ can be viewed as a *fractional matching* (see chapters 7, 24), and so the theorem can be sharpened by using the fractional matching (or cover) number τ^* on the right-hand side. In fact, the fractional cover number τ^* is a relaxation of the cover number τ, obtained by formulating τ as the optimum value of an integer linear program and dropping the integrality constraints.

Greedy travelling salesman tours. Recall the travelling salesman problem (TSP): given a graph $G=(V,E)$ on $n \geq 3$ nodes, and "distances" $c: E \rightarrow \mathbb{Z}_+$, find a Hamilton circuit with minimum length. Usually it is not an essential restriction of generality to assume that G is a complete graph. Moreover, in many important applications, the distance function c satisfies the triangle inequality: $c_{ij} + c_{jk} \geq c_{ik}$ for any three distinct nodes i, j, and k. We shall always restrict ourselves to the special case of the complete graph with lengths satisfying the triangle inequality (called the *metric case*).

The travelling salesman problem is NP-hard. Linear programming provides a practically quite efficient method to solve it (cf. section 8). Here we discuss two greedy heuristics.

The first one, called NEAREST NEIGHBOR heuristic, is an obvious idea: Choose some arbitrary node and visit it; from the last node visited go to the closest not yet visited node; if all nodes have been visited, return to the first node. This heuristic does make locally good choices, but it may run into traps. Figure 2.1 shows the result of a NEAREST NEIGHBOR run for a TSP consisting of 52 points of interest in Berlin, starting at point 1, the Konrad-Zuse-Zentrum. It is clear that this is far from being optimal. In fact, series of metric n-city TSP instances can be constructed where the tour built by the NEAREST NEIGHBOR heuristic is $\Omega(\log n)$ as long as the optimum tour length (Rosenkrantz et al. 1977).

The second greedy heuristic, called NEAREST INSERTION, will never show such a poor performance. It works as follows. We build up a circuit going through more and more nodes.

- Start with any node v_1 (viewed as a circuit with one node).
- Let T_k be a circuit of length k already constructed. Choose a node v_k not on T_k and a node u_k on T_k such that the distance $c_{u_k v_k}$ is minimal. Delete one of the edges of T_k incident with u_k, and connect its endpoints to v_k, to get a circuit T_{k+1}.
- After n steps we get a Hamiltonian circuit T_{nins}.

Another way to describe this heuristic is the following. The tree F formed by

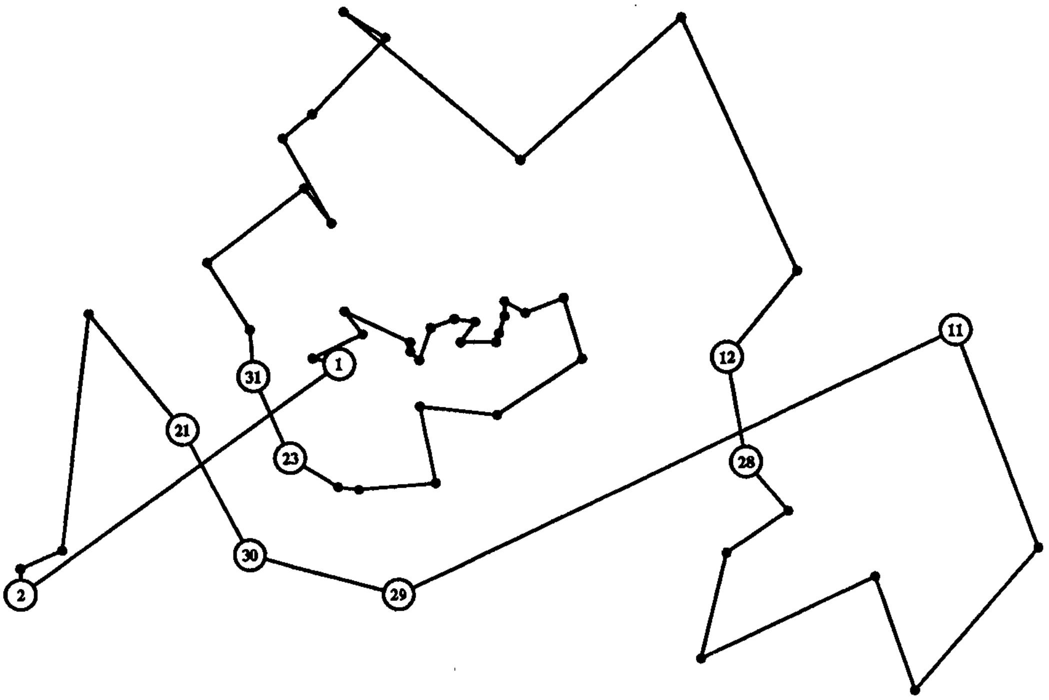

Figure 2.1. A nearest neighbor tour of a 52-city-problem.

the edges $v_k u_k$ is constructed by Prim's algorithm, (see, e.g., Cormen et al. 1990) and so it is a shortest spanning tree of G. We double each edge of F to obtain an Eulerian graph. An Euler tour of this visits every vertex at least once; making shortcuts, we obtain a tour that visits all nodes exactly once.

Theorem 2.3. *T_{nins} is at most twice as long as the optimum tour.*

Proof. Let T_{opt} denote an optimum tour. We make two observations. First, deleting any edge from T_{opt} we get a spanning tree. Hence

$$c(T_{\text{opt}}) \geq c(F) . \tag{2}$$

Second, inserting v_k into T_k we increase T_k's length by at most $2c_{u_k v_k}$; adding up these increments, we get

$$c(T_{\text{nins}}) \leq 2c(F) . \quad \square$$

Note that this argument (implicitly) uses a *dual heuristic* also: inequality (2) makes use of the fact that the length of the shortest spanning tree is a lower bound on the length of the shortest tour. The value of this dual heuristic is easily found by the greedy algorithm. In fact, a somewhat better lower bound could be obtained by considering *unicyclic* subgraphs, i.e., those subgraphs containing at most one circuit. The edge-sets of such subgraphs also form a matroid. The bases of this matroid are connected spanning subgraphs containing exactly one circuit; for this chapter, we call such subgraphs 1-trees. A 1-tree with minimum length can be found easily by the greedy algorithm, and since every tour is a 1-tree, this gives a lower bound on the minimum tour. We will see in section 9 how to further improve this lower bound.

There are several other greedy-like heuristics for the travelling salesman problem, such as farthest insertion, cheapest insertion, sweep, savings etc. Many of them do not have, however, a proven constant performance ratio, although some of them work better in practice than the nearest insertion heuristic (see Reinelt 1994).

The heuristic for the traveling salesman problem with the best known performance ratio is due to Christofides (1976). It uses the shortest spanning tree F, but instead of doubling its edges to make it Eulerian, it adds a matching M with minimum length on the set of nodes with odd degree in F. (Such a matching can be found in polynomial time, see chapter 3.) It is easy to see that $2c(M) \leq c(T_{\text{opt}})$, and hence the Christofides heuristic has performance ratio of at most $\frac{3}{2}$.

Bin packing. Let $a_1, \ldots, a_n \leq 1$ be positive real numbers ("weights"). We would like to partition them into classes ("bins") $B_1, \ldots, B_k$ so that the total weight of every bin is at most 1. Our aim is to minimize the number k of bins. Let k_{opt} be this minimum. To compute k_{opt} is NP-hard.

A trivial lower bound on how well we can do is the roundup of the total weight $w := \sum_i a_i$. The following simple (greedy) heuristic already gets asymptotically within a factor of 2 to this lower bound.

NEXT-FIT HEURISTIC. We process $a_1, a_2, \ldots$ one by one. We put them into one bin until putting the next a_i into the bin would increase its weight over 1. Then we close the bin and start a new one. We denote by k_{nf} the number of bins used by this heuristic.

Theorem 2.4. $k_{\mathrm{nf}} < 2w + 1$; $k_{\mathrm{nf}} < 2k_{\mathrm{opt}}$.

Proof. Let $k := k_{\mathrm{nf}}$, and denote by w_i the weight put in bin B_i by the NEXT-FIT heuristic $(1 \leq i \leq k)$. Then clearly $w_i + w_{i+1} > 1$ (since otherwise the first weight in B_{i+1} should have been put in B_i). If k is even, this implies

$$w = (w_1 + w_2) + (w_3 + w_4) + \cdots + (w_{k-1} + w_k) > k/2\,,$$

ɪnd hence

$$k < 2w \leq 2k_{\mathrm{opt}}\,.$$

If k is odd, we obtain

$$\begin{aligned} w &= \tfrac{1}{2}(w_1 + (w_1 + w_2) + (w_2 + w_3) + \cdots + (w_{k-1} + w_k) + w_k) \\ &\geq \tfrac{1}{2}(k - 1 + w_1 + w_2) > \frac{k-1}{2}\,, \end{aligned}$$

whence

$$k < 2w + 1 \leq 2k_{\mathrm{opt}} + 1\,,$$

and hence, $k \leq 2k_{\mathrm{opt}} - 1 < 2k_{\mathrm{opt}}$. □

The following better heuristic is still very "greedy".

FIRST-FIT HEURISTIC. We process $a_1, a_2, \ldots$ one by one, putting each a_i into the first bin into which it fits. We denote by k_{ff} the number of bins used by this ʾeuristic.

The following bound on the performance of FIRST FIT, due to Garey et al. (1976), is substantially more difficult to prove than the previous one.

Theorem 2.5. $k_{\mathrm{ff}} \leq \lceil \frac{17}{10} k_{\mathrm{opt}} \rceil$. *There exist lists with arbitrarily large total weight for which* $k_{\mathrm{ff}} \geq \lceil \frac{17}{10} k_{\mathrm{opt}} \rceil - 1$.

A natural (still "greedy") improvement on the FIRST-FIT heuristic is to preprocess the weights by ordering them decreasingly, and then apply FIRST-FIT. We call this the FIRST-FIT DECREASING heuristic, and denote the number of bins it uses by k_{ffd}. This preprocessing does improve the performance, as shown by the following theorem of Johnson (1973).

Theorem 2.6. $k_{\text{ffd}} \leqslant \lceil \frac{11}{9} k_{\text{opt}} \rceil + 4$. *There exist lists with arbitrarily large total weight for which* $k_{\text{ffd}} = \lceil \frac{11}{9} k_{\text{opt}} \rceil$.

There are other greedy-like heuristics for bin packing, e.g., best fit, whose performance ratio is similar; the heuristic called harmonic fit is slightly better. (See Coffman et al. 1984 for a survey.) More involved heuristic algorithms for the bin packing problem use linear programming and achieve an asymptotic performance ratio arbitrarily close to 1; see section 8.

The first two heuristics above are special in the sense that they are *on-line*: each weight is placed in a bin without knowing those that follow, and once a weight is placed in a bin, it is never touched again. On-line heuristics cannot achieve as good a performance ratio as general heuristics; it is easy to show that for any on-line heuristic for the bin packing problem there exists a sequence of weights (with arbitrarily large total weight) for which it uses at least $\frac{3}{2} k_{\text{opt}}$ bins. This lower bound can be improved to 1.54, van Vliet (1992).

The knapsack problem. Given a knapsack with total capacity b, and n objects with weights $a_1, \ldots, a_n$ of value $c_1, \ldots, c_n$, we want to pack as much value into the knapsack as possible. In other words, given positive integers $b, a_1, \ldots, a_n, c_1, \ldots, c_n$, we want to maximize $\sum_i c_i x_i$ subject to the constraints $\sum i\, a_i x_i \leqslant b$ and $x_i \in \{0, 1\}$, $i = 1, \ldots, n$. To exclude trivial cases, we may assume that $a_1, \ldots, a_n \leqslant b$. We denote by C_{opt} the optimum value of the objective function.

The knapsack problem is NP-hard; we will see in section 9, however, that one can get arbitrarily close to the optimum in polynomial time. Here we describe a greedy algorithm that has performance ratio at most 2.

It is clear that we want to select objects with small weight and large value; so it is natural to put in the knapsack an object with largest "weight density" c_i/a_i. Assume that the objects are labelled so that $c_1/a_1 \geqslant c_2/a_2 \geqslant \cdots \geqslant c_n/a_n$. Then the greedy algorithm consists of selecting $x_1, x_2, \ldots, x_n \in \{0, 1\}$ recursively as follows:

$$x_i = \begin{cases} 1, & \text{if } a_i \leqslant b - \sum_{j=1}^{i-1} a_j x_j, \\ 0, & \text{otherwise}. \end{cases}$$

Theorem 2.7. *For the solution* $x_1, \ldots, x_n$ *obtained by the* (*weight density*) *greedy algorithm, we have*

$$\sum_{i=1}^{n} c_i x_i \geqslant C_{\text{opt}} - \max_i c_i .$$

It follows that comparing the greedy solution with the best of the trivial solutions (select one object), we get a heuristic with performance ratio at most 2.

Proof. Let x be the solution found by the greedy algorithm, and y be an optimum solution. If x is not optimal, there must be an index j such that $x_j = 0$ and $y_j = 1$;

consider the least such j. Then we have for the greedy solution,

$$\begin{aligned} C_{\mathrm{gr}} := \sum_{i=1}^{n} c_i x_i \geqslant \sum_{i=1}^{j} c_i x_i = \sum_{i=1}^{j} c_i y_i + \sum_{i=1}^{j} c_i(x_i - y_i) \\ \geqslant \sum_{i=1}^{j} c_i y_i + \sum_{i=1}^{j} \frac{c_j}{a_j} a_i(x_i - y_i) = \sum_{i=1}^{j} c_i y_i + \frac{c_j}{a_j}\Big(\sum_{i=1}^{j} a_i x_i - \sum_{i=1}^{j} a_i y_i\Big) . \end{aligned} \tag{3}$$

Since j was not chosen by the greedy algorithm, we have here

$$\sum_{i=1}^{j} a_i x_i \geqslant b - a_j .$$

Furthermore, the feasibility of y implies that

$$\sum_{i=1}^{j} a_i y_i \leqslant b - \sum_{i=j+1}^{n} a_i y_i .$$

Substituting in (3), we obtain

$$C_{\mathrm{gr}} \geqslant \sum_{i=1}^{j} c_i y_i + \frac{c_j}{a_j}\Big(\sum_{i=j+1}^{n} a_i y_i - a_j\Big) \geqslant \sum_{i=1}^{j} c_i y_i - c_j = C_{\mathrm{opt}} - c_j . \qquad \square$$

3. Local improvement

The greedy algorithm and its problem-specific versions belong to a class of algorithms sometimes called (one-pass) *construction heuristics*. A myopic rule is applied and former decisions are not reconsidered. The purpose of these heuristics is to find a "reasonable" feasible solution very fast. But, of course, a locally good choice may lead to a globally poor solution.

Exchange heuristics for the travelling salesman problem. We have seen such unpleasant behavior in fig. 2.1: initially, short connections are chosen but, in the end, very long steps have to be made to connect the "forgotten" nodes. This picture obviously calls for a "repair" of the solution. For instance, replacing the edges from 12 to 28 and from 11 to 29 by the edges from 11 to 12 and 28 to 29 results in considerable saving. Obviously, further improvements of this kind are possible.

The 2-OPT heuristic formalizes this idea. It starts with some tour T, for instance a random tour or a tour obtained by a construction heuristic. Then it checks, for all pairs of non-adjacent edges uv, xy of T, whether the unique tour S formed by deleting these two edges and adding two edges, say $S := T \setminus \{uv, xy\} \cup \{ux, vy\}$ is shorter than T (this is the case, e.g., when the segments uv and xy cross in the plane). If so, T is replaced by S and the exchange tests are repeated. Otherwise the heuristic stops. Figure 3.1 shows the result of 2-OPT started with the tour in fig. 2.1.

There is an obvious generalization of this method: instead of removing two

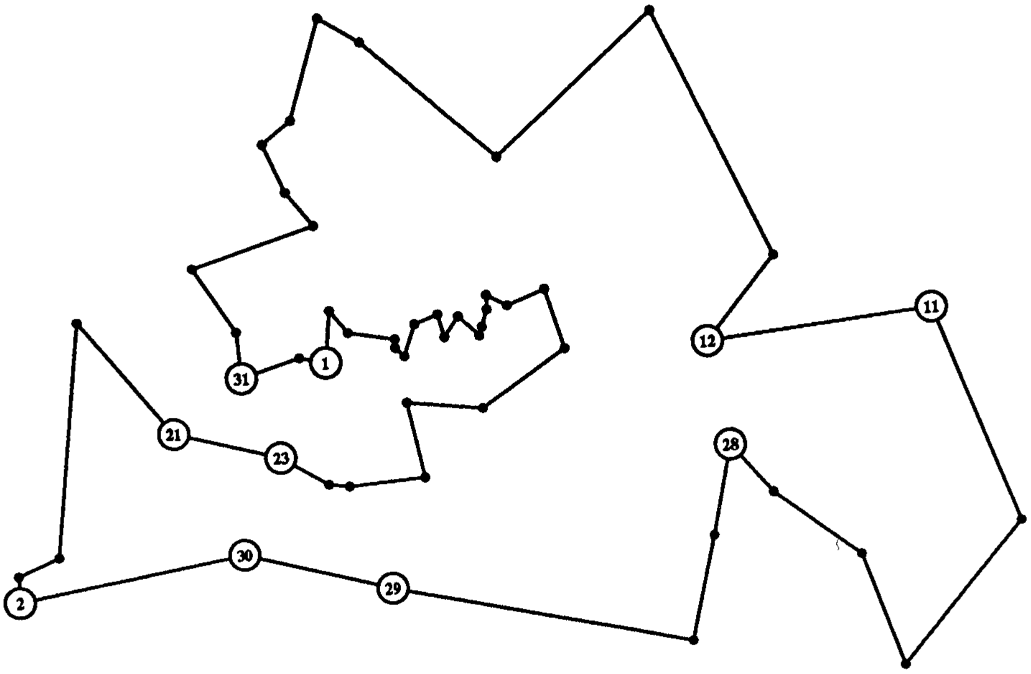

Figure 3.1. A 2-OPT tour of a 52-city-problem.

edges, we delete r non-adjacent edges from T. Then we enumerate all possible ways of adding r other edges such that these r edges together with the remaining r paths form a tour. If the shortest of these tours is shorter than the present tour T we replace T by this shorter tour and repeat. This method is called the r-OPT heuristic.

These heuristics are prototypical *local improvement techniques* that, in general, work as follows. We have some feasible solution of the given combinatorial optimization problem. Then we do some little operations on this solution, such as removing some elements and adding other elements, to obtain one or several new solutions. If one of the new solutions is better than the present one, we replace the present one by the new best solution and repeat.

The basic ingredient of such improvement or exchange heuristics is a rule that describes the possible manipulations that are allowed. This rule implicitly defines, for every feasible solution, a set of other feasible solutions that can be obtained by such a manipulation. Using this interpretation, we can define a digraph D whose vertex set is the set Ω of feasible solutions of our combinatorial optimization problem (this is typically exponentially large) and where an arc (S, T) is present if T can be obtained from S by the manipulation rule.

Now, local improvement heuristics can be viewed as algorithms that start at some node of this digraph and search, for a current node, the successors of the node. If they find a better successor they go to this successor and repeat. The term *local search algorithms* that is also used for these techniques derives from this interpretation.

There are many important algorithms that fit in this scheme: the simplex method (see chapter 30), basis reduction algorithms (chapter 20), etc. The simplex method is somewhat special in the sense that it only gets stuck when we have reached the optimum. Usually, local improvement algorithms may run into *local optima*, i.e., solutions from which the given local manipulations do not lead to any better solution.

Maximum cuts. Let us look at another example. Suppose we are given a graph $G = (V, E)$ and we want to find a cut $\delta(W) = \{ij \in E : i \in W,\ j \not\in W\}$, $W \subseteq V$, of maximum size. (This is *the cardinality max-cut problem* for G.) The problem has a natural weighted version, where each edge has a (say, rational) weight, and we want to find a cut with maximum weight.

We start with an arbitrary subset $W \subseteq V$. We check whether W (or $V \backslash W$) contains a node w such that fewer than half of its neighbors are in $V \backslash W$) (or W). If such a node exists we move it from W to $V \backslash W$ (or from $V \backslash W$ to W). Otherwise we stop.

Termination of this *single exchange heuristic* in $\mathrm{O}(n^2)$ time is guaranteed, since the size of the cut increases in every step. The cut produced by this procedure obviously has a size that is at least one half of the maximum cardinality of a cut in G, and in fact, it can be as bad as this, as the complete bipartite graph shows.

The single exchange heuristic has an obvious weighted version: we push a node w to the other side if the sum of the weights of edges linking w to nodes on the other side is smaller than the sum of the weights of the edges linking w to nodes on the side of w. This version, of course, also terminates in finite time, but the number of steps may be exponential (Haken and Luby 1988) even for 4-regular graphs. On the other hand Poljak (1993) proved that the single exchange heuristic for the weighted max-cut problem terminates in polynomial time for cubic graphs (see section 9).

For a typical combinatorial optimization problem, it is easy to find many local manipulation techniques. For instance, for the max-cut problem we could try to move several nodes from one side to the other or exchange nodes between sides; for the TSP we could perform node exchanges instead of edge exchanges, we could exchange whole sections of a tour, and we could combine these techniques, or we could vary the number of edges and/or nodes that we exchange based on some criteria. In fact, most people working on the TSP view the well-known heuristic of Lin and Kernighan (1973) and its variants as the best local improvement heuristics for the TSP known to date (based on the practical performance of heuristics on large numbers of TSP test instances). This heuristic is a dynamic version of the r-OPT heuristic with varying r.

The last statement may suggest that improvement heuristics are the way to solve (large scale) TSP instances approximately. However, there are some practical and theoretical difficulties with respect to running times, traps and worst-case behavior.

Running time of exchange heuristics. A straightforward implementation of the r-OPT heuristic, for example, is hopelessly slow. It takes $\binom{n}{r}$ tests to check whether a tour can be improved by an r-exchange. Even for $r = 3$, the heuristic runs almost forever on medium size instances of a few thousand nodes. To make this approach practical, a number of modifications limiting the exchanges considered are necessary. They are usually based on insights about the probability of success of certain exchanges, or on knowledge about special structures (an instance might be geometrical, e.g., given by points in the plane and a distance function). Well-designed fast data structures play an important role. The issue of speeding up TSP heuristics is treated in depth, e.g., in Johnson (1990), Bentley (1992), and Reinelt (1994). With these techniques TSP instances of up to a million cities have been solved approximately. Such observations apply to many other combinatorial optimization problems analogously.

It may sound strange, but for many exchange heuristics there is no proof that these heuristics terminate in polynomial time. This is the case even with such a basic and classical algorithm as the simplex method! For certain natural pivoting rules we know that they may lead to exponentially many iterations, while for others, it is not known whether or not they terminate in polynomial time; no pivoting rule is known to terminate in polynomial time. As another, simpler example, we mention that although we could prove an $O(n^2)$ running time for the cardinality version of the max-cut heuristic, its weighted version is not polynomial, as mentioned above.

A single pass through the loop of the r-OPT heuristic for the TSP takes $O(n^r)$ time but it is not clear how to bound the number of tours that have to be processed before the algorithm terminates with an r-OPT *tour*, i.e., a tour that cannot be improved by an r-exchange. Computational experience, however, shows that exchange heuristics usually do not have to inspect too many tours until a "local optimum" is found.

Quality of the approximation. Although the solution quality of local improvement heuristics is often quite good, these heuristics may run into traps, for instance r-OPT tours, whose value is not even close to the optimum. It is, in general, rather difficult to prove worst-case bounds on the quality of exchange heuristics. For an example where a performance ratio is established, see the basis reduction algorithm in chapter 19.

It is probably fair to say that, for the solution of combinatorial optimization problems appearing in practice, fast construction heuristics combined with local improvement techniques particularly designed for the special structures of the application are the real workhorses of combinatorial optimization. That is why this machinery receives so much attention in the literature and why new little tricks or clever combinations of old tricks are discussed intensively. Better solution qualities or faster solution times may result in significant cost savings, in particular for complicated problems of large scale.

Aiming at local optima. There are several examples when we are only interested in finding a local optimum: it is the structure of the local optimum, and not the value of the objective function, that concerns us. A theoretical framework for such "polynomial local search problems" was developed by Johnson et al. (1988). Analogously to NP, this class also has complete (hardest) problems; the weighted local max-cut problem is one of these (Schäffer and Yannakakis 1991).

Consider an optimum solution W of the max-cut problem for a graph $G = (V, E)$. Clearly, every node is connected to at least as many points on the opposite side of the cut than on its own side. If we only want a cut with this property, any locally optimal cut (with respect to the single exchange heuristic) would do.

Assume now that we want to solve the following more general problem: given two functions $f, g: V \to \mathbb{Z}_+$ such that $f(v) + g(v) = d_G(v) - 1$ for all $v \in V$, find a subset $W \subseteq V$ such that for each node v, the number of nodes adjacent to v on its own side of the cut is at most

$$\begin{aligned} &f(v)\,, \quad \text{if } v \in W\,,\\ &g(v)\,, \quad \text{if } v \in V \setminus W\,. \end{aligned}$$

It is not difficult to guess an objective function (over cuts) for which such cuts are exactly local optima:

$$\phi(W) := |\delta(W)| + f(W) + g(V \setminus W)\,.$$

Once this is found, it follows that a cut with the desired property exists, and also that it can be found in polynomial time by local improvement. A much more difficult, but in principle similar, application of this idea is the proof of Szemerédi's regularity lemma (see chapter 23). Here again a tricky (quadratic) objective function is set up, which is locally improved until a locally almost optimal solution is found; the structure of such a solution is what is needed in the numerous applications of the Regularity lemma (see also Alon et al. 1994 for the algorithmic aspects of this procedure).

A beautiful example of turning a structural question into an optimization problem is Tutte's (1963) proof of the fact that every 3-connected planar graph has a planar embedding with straight edges and convex faces. He considers the edges as rubber bands, fixes the vertices of one face at the vertices of a convex polygon, and lets the remaining vertices find their equilibrium. This means minimizing a certain quadratic objective function (the energy), and the optimality criteria can be used to prove that this equilibrium state defines a planar embedding with the right properties. The algorithm can be used to actually compute nice embeddings of planar graphs. A similar method for connectivity testing was given by Linial et al. (1988).

Randomized exchange. A very helpful idea to overcome the problem of falling into a trap is to randomize. In a randomized version of local search, a random neighbor of the current feasible solution is selected. If this improves the objective function, the current feasible solution is replaced by this neighbor. If not, it may still be replaced, but only with some probability less than 1, depending on how much worse the objective value at the neighbor is. This relaxation of the strict descent rule may help to jump out of traps and eventually reach a significantly better solution.

A more general way of looking at this method is to consider it as generating a random element in the set Ω of feasible solutions, from some given probability distribution Q. Let $f: \Omega \rightarrow \mathbb{R}_+$ be the objective function; then maximizing f over Ω is just the extreme case when we want to generate a random element from a distribution concentrated on the set of optimum solutions. If, instead, we generate a random point w from the distribution Q in which $Q(v)$ is proportional to $\exp(f(v)/T)$, where T is a very small positive number, then with large probability w will maximize f. In fact, a randomized algorithm that finds a solution that is (nearly) optimal with large probability is equivalent to a procedure of generating a random element from a distribution that is heavily concentrated on the (nearly) optimal solutions.

To generate a random element from a distribution over a (large and complicated) set is of course a much more general question, and is a major ingredient in various algorithms for enumeration, integration, volume computation, simulation, statistical sampling, etc. (see Jerrum et al. 1986, Dyer and Frieze 1992, Sinclair and Jerrum 1988, Dyer et al. 1991, Lovász and Simonovits 1992 for some of the applications with combinatorial flavor). An efficient general technique here is *random walks* or *Markov chains*. Let $G = (\Omega, E)$ be a connected graph on Ω, and assume, for simplicity of presentation, that G is non-bipartite and d-regular. If we start a random walk on G and follow it long enough, then the current point will be almost uniformly distributed over Ω. How many steps does "long enough" mean depends on the spectrum, or in combinatorial terms, on global connectivity properties called expansion rate or conductance, of the graph (see also chapter 31).

The Metropolis filter. In optimization, we are interested in very non-uniform, rather than uniform, distributions. Fortunately, there is an elegant way, called the *Metropolis filter* (Metropolis et al. 1953), to modify the random walk, so that it gives any arbitrary prescribed probability distribution. Let $F: \Omega \rightarrow \mathbb{R}_+$. Assume that we are at node v. We choose a random neighbor u. If $F(u) \geqslant F(v)$ then we move to u; else, we flip a biased coin and move to u only with probability $F(u)/F(v)$, and stay at v with probability $1 - F(u)/F(v)$.

Let Q_F denote the probability distribution on Ω defined by the property that $Q_F(v)$ is proportional to $F(v)$. The miraculous property of the Metropolis filter is the following.

Theorem 3.1. *The stationary distribution of the Metropolis-filtered random walk is* Q_F.

So choosing $F(v) = \exp(f(v)/T)$, we get a randomized optimization algorithm.

Unfortunately, the issue of how long one has to walk gets rather messy. The techniques to estimate the conductance of a Metropolis-filtered walk are not general enough, although Applegate and Kannan (1990) have been able to apply this technique to volume computation.

Simulated annealing. Coming back to optimization, let us follow Kirckpatrick et al. (1983), and call the elements of Ω *states* (of some physical system), $1/F(v)$ the *energy* of the state, and T the *temperature*. In this language, we want to find a state with minimum (or almost minimum) energy. A Metropolis-filtered random walk means letting the system get into a stationary state at the given temperature.

The main, and not quite understood, issue is the choice of the temperature. If we choose T large, the quality of the solution is poor, i.e., the probability that it is close to being optimal is small. If we choose T small, then there will be barriers of very small probability (or, equivalently, with very large energy) between local optima, and it will take extremely long to get away from a local, but not global optimum.

The technique of *simulated annealing* suggests to start with the temperature T sufficiently large, so that the random walk with this parameter mixes fast. Then we decrease T gradually. In each phase, the random walk starts from a distribution which is already close to the limiting distribution, so there is hope that the walk will mix fast. (A similar trick works quite well in integration and volume computation; see Lovász and Simonovits 1992.) Theoretical and practical experiments have revealed that it matters a lot how long we walk in a given phase (*cooling schedule*).

There are many empirical studies with this method; see Johnson et al. (1989, 1991) or Johnson (1990). There are also some general estimates on its performance (Holley and Stroock 1988, Holley et al. 1989). Examples of problems, in particular of the matching problem, are known where simulated annealing performs badly (Sasaki and Hajek 1988, Sasaki 1991, Jerrum 1992), and some positive results in the case of the matching problem are also known (Jerrum and Sinclair 1989). The conclusion that can be drawn at the moment is that simulated annealing is a potentially valuable tool (if one can find good cooling schedules and other parameters), but it is in no way a panacea as was claimed in some papers pioneering this topic.

Approaches named taboo search, threshold accept, evolution or genetic algorithms and others are further variants and enhancements of randomized local search. The above judgement of simulated annealing applies to them as well, see Johnson (1990).

4. Enumeration and branch-and-bound

There is a number of interesting combinatorial optimization problems for which beautiful polynomial time algorithms exist. We will explain some of them in subsequent sections. We now address the issue of finding an optimum solution for an NP-hard problem. In the previous two sections we have outlined heuristics that produce some feasible and hopefully good solutions. Such a solution may even be optimal right away. But how does one verify that?

The basic trouble with integer programming and combinatorial optimization is the non-existence of a sensible duality theory. The duality theorem of linear programming (see chapter 30), for instance, can be used to prove that some given feasible solution is optimal. In the (rare) cases where duality theorems in integral solutions like the max-flow min-cut theorem (see chapter 2 or 30) or the Lucchesi–Younger theorem (see chapter 2) exist, one can usually derive a polynomial time solution algorithm. For NP-hard problems one should not expect to find such theorems. Unfortunately, nothing better is known than replacing such a theory by brute force.

Trivial running time estimates reveal that the obvious idea of simply enumerating the finitely many solutions of a combinatorial optimization problem is completely impractical in general. For instance, computing the length of all $\frac{1}{2} \cdot 15!$ ($\sim 0.6 * 10^{12}$) tours of a 16 city TSP instance takes about 90 hours on a 30 MIPS workstation. Even a teraflop computer will be unable to enumerate all solutions of a ridiculously small 30 city TSP instance within its lifetime.

Unless $\mathrm{P} = \mathrm{NP}$, there is no hope that we will be able to design algorithms for NP-hard problems that are asymptotically much better than enumeration. However, we can try to bring problem instances of reasonable sizes (appearing in practice, say) into the realm of practical computability by enhancing enumeration with a few helpful ideas.

The idea of the *branch-and-bound* approach is to compute tight upper and lower bounds on the optimum value in order to significantly reduce the number of enumerative steps. To be more specific, let us assume that we have an instance of a minimization problem. Let Ω be its set of feasible solutions.

To implement branch-and-bound, we need a dual heuristic (relaxation), i.e., an efficiently computable lower bound on the optimum value. This dual heuristic will also be called for certain subproblems.

We first run some construction and improvement heuristics to obtain a good feasible solution, say T, with value $c(T)$, which is an upper bound for the optimum value c_{opt}.

Now we resort to enumeration. We split the problem into two (or more) subproblems. Recursively solving these subproblems would mean straightforward enumeration. We can gain by maintaining the best solution found so far and computing, whenever we have a subproblem, a lower bound for the optimum value of this subproblem, using the dual heuristic. If this value is larger than the value of the best solution found so far, we do not have to solve this subproblem.

To be more specific, let us discuss a bit the two main ingredients, *branching* and *bounding*.

We assume that the splitting into subproblems (the *branching*) is such that the set Ω of feasible solutions is partitioned into the sets Ω' and Ω'' of feasible solutions of the subproblems. We also assume that the subproblems are of the same type (e.g., the dual heuristic applies to them). It is also important that the branching step requires little bookkeeping and is computationally cheap. If Ω

consists of subsets of a set S, then a typical split is to choose an element $e \in S$ and to set

$$\Omega' := \{I \in \Omega \mid e \in I\}\,, \qquad \Omega'' := \{I \in \Omega \mid e \notin I\}\,.$$

The *bounding* is usually provided by a *relaxation*: by problem-specific investigations we introduce a new problem, whose set of feasible solutions is Γ, say, such that $\Omega \subseteq \Gamma$ and the objective function for Ω extends to Γ. Suppose X minimizes the objective function over Γ, then the value $c(X)$ provides a lower bound for c_{opt} since all elements of Ω participated in the minimization process. If X is, in fact, an element of Ω we clearly have found an optimum solution of Ω. (If $X \notin \Omega$, we may still be able to make use of it by applying a construction and/or improvement heuristic that starts with X and ends with a solution Y, say, in Ω. If $c(Y) < c(T)$ we set $T := Y$ to keep track of our current best solution.)

To give some examples, useful relaxations for the symmetric TSP are perfect 2-matchings (unions of disjoint circuits that cover all nodes) or 1-trees (cf. sections 2 and 9). For the asymmetric TSP a standard relaxation is obtained by considering unions of directed circuits that cover all nodes (which can be easily reduced to a bipartite perfect matching problem), or the r-arborescence problem.

A particularly powerful method is based on LP-relaxations. This is covered in depth in section 8. There are some general methods to improve relaxations; one technique is called Lagrangian relaxation and will be discussed in section 9.

Returning to the algorithm, we maintain a list of unsolved subproblems, and a solution T that is the current best. In the general step, we choose an unsolved subproblem, say Ω^i, from the list and remove it. We optimize the objective function over the relaxation Γ^i of Ω^i. Let X be an optimum solution. There are several possibilities.

- First, X may be feasible for Ω^i. In this case we have found an optimum over Ω^i and can completely eliminate all elements of Ω^i from the enumeration process. One often says that this branch is *fathomed*. If $c(X) < c(T)$, we reset $T := X$ also.
- Second, if $c(X) \geqslant c(T)$ then no solution in Γ^i and hence no solution in Ω^i has a value that is smaller than the current champion. Hence this branch is also fathomed and we can eliminate all solutions in Ω^i.
- Third, if $c(X) < c(T)$ (and X is not feasible for Ω^i), we have done the computation in vain. (We may still try to make use of X to obtain a solution better than T as above.) We split Ω^i into two or more pieces, and put these on our list of unsolved subproblems.

The branch-and-bound method terminates when the list of subproblems is empty. The iteratively updated solution T is the optimum solution. Termination is, of course, guaranteed if the set Ω of feasible solutions is finite.

Although the global procedure is mathematically trivial, it is a considerable piece of work to make it computationally effective. The efficiency mainly depends on the quality of the lower bound used. Most of the mathematics that is developed for the solution of hard problems is concerned with the invention of better and better relaxations, with their structural properties and with fast algorithms for their solution.

5. Dynamic programming

Dynamic programming is a general technique for optimum decision making. It was originally developed (Bellman 1957) for the solution of discrete-time sequential decision processes. The process starts at a given initial state. At any time of the process we are in some state and there is a set of states that are reachable from the present state. We have to choose one of these. Every state has a value and our objective is to maximize the value of the terminating state. Such an optimization problem is called a *dynamic program*.

Virtually any optimization problem can be modeled by a dynamic program. There is a recursive solution for dynamic programs which, however, is not efficient in general. But the dynamic programming model and this recursion can be used to design fast algorithms in cases where the number of states can be controlled. We illustrate this by means of a few examples.

The subset-sum problem. Given positive integers $a_1, a_2, \ldots, a_n, b$, decide whether there exist indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ for some k such that $a_{i_1} + \cdots + a_{i_k} = b$. This problem, called *subset-sum problem*, is NP-complete; however, there is a pseudopolynomial algorithm to solve it.

First, consider an obvious algorithm using enumeration. Clearly the subset-sum problem has a solution for a given input $(a_1, \ldots, a_n, b)$, if and only if it has a solution either for $(a_1, \ldots, a_{n-1}, b - a_n)$ or for $(a_1, \ldots, a_{n-1}, b)$. So an instance of the subset-sum problem for n numbers can be reduced to two subproblems with $n - 1$ numbers each. Building up a search tree based on this observation yields an $O(2^n)$ algorithm (which basically enumerates all subsets of the a_i).

But looking at this tree more carefully, we see that, at least if b is small compared with 2^n, it has a crucial property: *the same subproblem occurs on many branches*! In fact, there are only nb distinct subproblems altogether: for each $c \leq b$ and each $m \leq n$, the subset-sum problem with input $a_1, \ldots, a_m, c$. Imagine that the branches of the search tree "grow together" if the same subproblem occurs: we get a "search diagraph" D. The nodes of this acyclic digraph are labelled with pairs (c, m), and there is an edge from (c, m) to $(c', m - 1)$ if either $c' = c$ or $c' = c - a_m$. The subset-sum problem is solvable if and only if there is a dipath from (b, n) to $(0, 0)$. Such a dipath can be found (if it exists) in $O(bn)$ time by searching D either from $(0, 0)$ or from (b, n).

Along the same lines, one can devise an algorithm for the knapsack problem with running time polynomial in $b + \sum_i \langle c_i \rangle$.

Minimal triangulation of a convex polygon. Given a convex polygon P with n

vertices in the plane, we want to find a triangulation with minimal total edge length. (The length c_{ij} of each edge ij is known.)

If the vertices of P are numbered consecutively 1 through n, take edge $1n$ and consider the vertex i with which it forms a triangle in the triangulation. For a given i, it suffices to find optimal triangulations of the two polygons with vertices $1, \ldots, i$ and $i, \ldots, n$, respectively, which can be done independently, see fig. 5.1. So we have produced $2(n-2)$ subproblems.

If we are not careful, repeating this process could lead to exponentially many distinct subproblems. But note that if we choose the triangle containing the edge $1i$ to cut the polygon $1, \ldots, i$, then we get two subproblems corresponding to convex polygons having only one edge that is not an edge of P.

In general, given two vertices i and j with $i \leqslant j-2$, let $f(i,j)$ denote the minimum total length of diagonals triangulating the polygon with vertices $(i, i+1, \ldots, j)$. Then clearly $f(i, i+2) = 0$ and

$$\begin{aligned} f(i,j) = \min\{ & \min_{i+2 \leqslant k \leqslant j-2} \{f(i,k) + f(k,j) + c_{ik} + c_{kj}\}, \\ & f(i+1, j) + c_{i+1,j}, f(i, j-1) + c_{i,j-1}\}. \end{aligned} \tag{4}$$

The answer to the original question is $f(1,n)$.

We can represent the computation by a "search digraph" whose nodes correspond to all the polygons with vertex set $(i, i+1, \ldots, j)$, where $1 \leqslant i, j \leqslant n$, $i \leqslant j-2$. We set $f(i,j) = 0$ if $j = i+2$, and can use (4) recursively if $j > i+2$. There are $O(n^2)$ subproblems to solve, and each recursive step takes $O(n)$ time. So we get an $O(n^3)$ algorithm.

Steiner trees in planar graphs. Let $G = (V, E)$ be a graph with edge lengths $c_e > 0$, and let $T \subseteq V$ be a set of "terminals". A *Steiner tree* in G is a subtree of G that contains all nodes of T. The *Steiner tree problem* is the task of finding a shortest

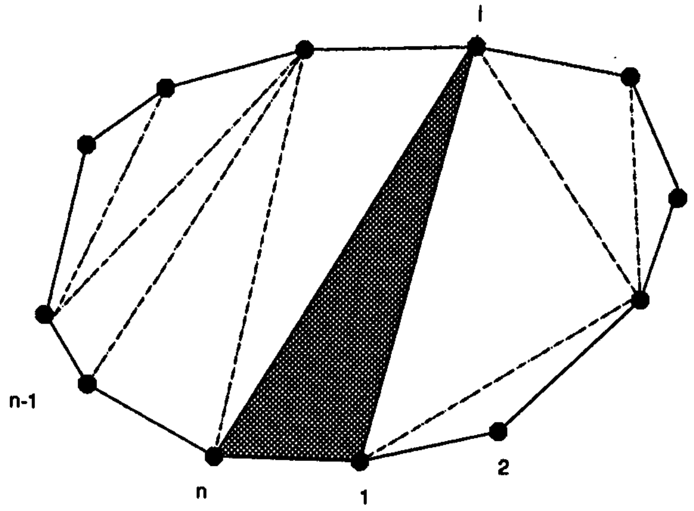

Figure 5.1. Optimum triangulation of a convex polygon.

Steiner tree. This problem is NP-hard in general, even for planar graphs. But in the case of a planar graph when all the terminal nodes are on one, say, on the outer face C, a shortest Steiner tree can be found by dynamic programming as follows.

Let us first look at a minimum Steiner tree B. Pick any node v of B and let B' be the union of some branches of B that are rooted at v and that are, in addition, consecutive in the natural cyclic order of the edges leaving v. Let $T' := T \cap V(B')$. We observe the following (see fig. 5.2, where v is represented by a black circle):

- There is a path $P \subseteq C$ whose endnodes are terminals such that $T' = V(P) \cap T$.
- B' is a minimum length Steiner tree with respect to the terminal set $T' \cup \{v\}$.
- v is on the outer face of the subgraph $B' \cup P$.

These observations motivate the following dynamic program for the solution of our Steiner tree problem. For every path $P \subseteq C$ whose end nodes are terminals and every node $v \in V$, we determine a shortest Steiner tree B' with respect to the set of terminal nodes $(V(P) \cap T) \cup \{v\}$ with the additional requirement that v is on the outer face of $B' \cup P$.

If P consists of just one terminal then such a Steiner tree can be found by a shortest path calculation.

Suppose that we have solved this subproblem for all nodes $v \in V$ and all paths $P \subseteq C$ containing at most k terminal nodes. To solve the subproblem for some node $v \in V$ and a path $P \subseteq C$ containing $k+1$ terminal nodes, we do the

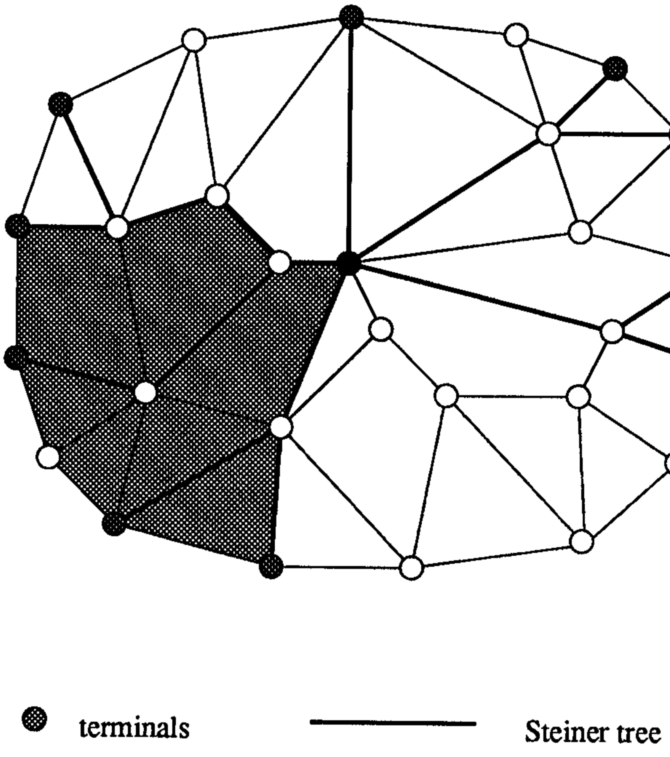

Figure 5.2. Steiner tree in planar graphs.

following. Let $t_1, \ldots, t_{k+1}$ be the terminals contained in P in the natural order. For every node $w \in V$ and every two subpaths P_1, P_2 of P, where P_1 connects t_1 to t_j and P_2 connects t_{j+1} to t_{k+1}, $1 \leq j \leq k$, we solve the subproblems for w and P_1 and for w and P_2 to get two trees B_1 and B_2. We also compute a shortest path Q from w to v. Among all the sets $B_1 \cup B_2 \cup Q$ computed this way we choose the one with minimum length. This is an optimum solution of our subproblem for v and P.

To get a minimum length Steiner tree, consider a path $P \subseteq C$ that contains all terminal nodes and choose the shortest among all solutions of subproblems for v and P with $v \in V$.

This algorithm is due to Erickson et al. (1987) and is based on ideas of Dreyfus and Wagner. It can be extended to the case when all terminals are on a fixed number of faces.

There are many other non-trivial applications of the idea of dynamic programming; for example, Chvátal and Klincsek (1980) use it to design a polynomial time algorithm that finds a maximum cardinality subset of a set of n points in the plane that forms the vertices of a convex polygon (cf. chapter 17, section 7.2).

Optimization on tree-like graphs. There are many NP-hard problems that are easy if the underlying graph is a tree. Consider the stable set problem in a tree T. We fix a root r and, for every node x, we consider the subtree T_x consisting of x and its descendants. Starting with the leaves, we compute, for each node x, two numbers: the maximum number of independent nodes in T_x and in $T_x - x$. If these numbers are available for every son of x, then it takes only $O(d(x))$ time to find them for x. Once we know them for T_r, we are done. So a maximum stable set can be found in a linear time.

Similar algorithms can be designed for more general "tree-like" graphs, e.g., series–parallel graphs. A general framework for "tree-like" decompositions, developed by Robertson and Seymour in their "Graph minors" theory, leads to very general dynamic programming algorithms on graphs with bounded tree-width. See chapter 5 for the definition of tree-width and for examples of such algorithms.

6. Augmenting paths

In local search algorithms, we try to find very simple "local" improvements on the current solution. There are more sophisticated improvement techniques that change the current solution globally, usually along paths or systems of paths. These methods are often called *augmenting path* techniques. Since they occur throughout this Handbook, we refrain from describing any of them here; let it suffice to quote the most important applications of the method of alternating paths: maximum flows and packing of paths, chapter 2; maximum matchings (weighted and unweighted), maximum stable sets in claw-free graphs, chapter 3;

edge-coloring, chapter 4; matroid intrersection matroid matching, and submodular flows, chapter 11.

7. Uncrossing

Uncrossing is a technique to simplify complicated set-systems, while maintaining certain key properties. One can find many applications of the uncrossing procedure in this Handbook as a proof technique; it is applied in the theory of graph connectivity and flows (chapter 2), matchings (chapter 3, or Lovász and Plummer 1986), and matroids (chapters 11, 30). It is worth pointing out, however, that uncrossing can be viewed as an algorithmic tool, that constructs, from a complicated dual solution, a dual solution with a tree-like structure. This way it is sometimes possible to derive an optimum integral dual solution from an optimum fractional dual solution.

As an illustration, consider the problem of finding a maximum family of rooted cuts in a digraph $G = (V, A)$ with root r such that every arc a occurs in at most l_a of these cuts, where the $l_a \geqslant 0$ are given integer values ("lengths"). (A rooted cut or r-cut, is the set of arcs entering S, i.e., with tail in $V \backslash S$ and head in S for some non-empty $S \subset V$, $r \not\in S$; cf. chapter 30.) Assume that we have a *fractional packing*, i.e., a family $\mathcal{F}$ of r-cuts and a weight $w_D \geqslant 0$ for every $D \in \mathcal{F}$ such that $\sum_{D \ni e} w_D \leqslant l_a$ for every arc a (ellipsoidal or interior point methods, as well as averaging procedures, may yield such "fractional solutions"). As a consequence of Fulkerson's optimum arborescence theorem (chapter 30, Theorem 5.7), we know that there exists an integer solution with the same value, i.e., a family of at least $\sum_D w_D$ r-cuts with the prescribed property. But how to find this?

For each r-cut D in the digraph G, we denote by $S(D)$ a set $S \subseteq V \backslash \{r\}$ such that D is the set of edges entering S. Let $\mathcal{H} = \{S(D)\colon D \in \mathcal{F}\}$. Call two r-cuts D_1 and D_2 *intersecting* if all three sets $S(D_1) \cap S(D_2)$, $S(D_1) \backslash S(D_2)$ and $S(D_2) \backslash S(D_1)$ are non-empty. Assume that $\mathcal{F}$ contains two intersecting cuts D_1 and D_2, and let D' and D'' denote the r-cuts defined by $S(D_1) \cap S(D_2)$ and $S(D_1) \cup S(D_2)$, respectively.

Decrease w_{D_1} and w_{D_2} by ε and increase $w_{D'}$ and $w_{D''}$ by ε, where $\varepsilon := \min\{w_{D_1}, w_{D_2}\}$ (if, say, D' does not belong to $\mathcal{F}$, then we add it to $\mathcal{F}$ with $w_{D'} = 0$). It is easy to check that this yields a new fractional packing with the same total weight. The family $\mathcal{F}$ lost one member (one of D_1 and D_2), and gained at most two new members (D' and D''). If the new family contains two intersecting cuts, then we "uncross" them as above. It can be shown that the procedure terminates in a polynomial number of steps (see Hurkens et al. 1988 for a discussion of this).

When the uncrossing procedure terminates, the family $\mathcal{H}$ is *nested*, i.e., there are no intersecting pairs of cuts in $\mathcal{F}$. Such a family has a tree structure; $\mathcal{H}$ can be obtained by selecting disjoint subsets of $V \backslash \{r\}$, then disjoint subsets in these subsets, etc. It is not difficult to see that the number of members of $\mathcal{H}$ is at most $2|V| - 3$.

Choose $D \in \mathcal{F}$ such that w_D is not an integer and $S(D)$ is minimal. There is a unique cut $D' \in \mathcal{F}$ such that $S(D') \supset S(D)$ and $S(D')$ is minimal. Add ε to w_D and subtract ε from $w_{D'}$, where $\varepsilon := \min\{\lceil w_D \rceil - w_D, w_{D'}\}$. It is easy to check using the integrality of ℓ) that this results in a fractional packing with the same value, and now either w_D is an integer or $w_{D'} = 0$. After at most $2n-3$ repetitions of this shift, we get a fractional packing with all weights integral, which trivially gives the family as required.

8. Linear programming methods

A very successful way to solve combinatorial optimization problems is to translate them into optimization problems for polyhedra and utilize linear programming techniques. The theoretical background of this approach is surveyed in chapter 30 where also many examples of the application of this method are provided. We will concentrate here on the implementation of the linear programming approach to practical problem solving, and on the use of linear programming in heuristics.

We will assume that we have a combinatorial optimization problem with linear objective function like the travelling salesman, the max-cut, the stable set, or the matching problem. Let us also assume that we want to find a feasible solution of maximum weight. Typically, an instance of such a problem is given by a ground set E, an objective function $c: E \rightarrow \mathbb{R}$ and a set $\mathcal{I} \subseteq 2^E$ of feasible solutions such as the set of tours, of cuts, of stable sets, or matchings of a graph. We transform $\mathcal{I}$ into a set of geometric objects by defining, for each $I \in \mathcal{I}$, a vector $\chi^I \in \mathbb{R}^E$ with $\chi^I_e = 1$ if $e \in I$ and $\chi^I_e = 0$ if $e \not\in I$. The vector χ^I is called the *incidence* (*or characteristic*) *vector* of I. Now we set

$$\mathcal{P}(\mathcal{I}) := \operatorname{conv}\{\chi^I \in \mathbb{R}^E \mid I \in \mathcal{I}\} ,$$

i.e., we consider a polytope whose vertices are precisely the incidence vectors of the feasible solutions of our problem instance. Solving our combinatiorial optimization problem is thus equivalent to finding an optimum vertex solution for the following linear program

$$\max c^{\mathrm{T}} x , \quad x \in P(\mathcal{I}) . \tag{5}$$

However, (5) is only a linear program "in principle" since the usual LP-codes require the polyhedra to be given by a system of linear equations and inequalities. Classical results of Weyl and Minkowski ensure that a set given as the convex hull of finitely many points has a representation by means of linear equations and inequalities (and vice versa). But it is by no means simple to find, for a polyhedron given in one of these representations, a complete description in the other way. By problem specific investigations one can often find classes of valid and even facet-defining inequalities that partially describe the polyhedra of interest. (Many of the known examples are described in chapter 30.)

What does "finding" a class mean? The typical situation in polyhedral combinatorics is the following. A class of inequalities contains a number of inequalities that is exponential in $|E|$. It is well-described in the sense that we can (at least) decide in polynomial time whether a given inequality, for a given instance, belongs to the class. This is a minimal requirement; we need more for a class to be really useful (see *separation* below). Also, the class should contain strong inequalities; best is when most of the inequalities define facets of $P(\mathcal{I})$. Except for special cases, one such class (or even a bounded number of such classes) will not provide a complete description, i.e., $P(\mathcal{I})$ is strictly contained in the set of solutions satisfied by all these inequalities.

The cut polytope. To give an example, let us discuss the max-cut problem. In this case a graph $G = (V, E)$ is given (for convenience we will assume that G is simple), and we are interested in the convex hull of all incidence vectors of cuts in G, i.e.,

$$\mathrm{CUT}(G) := \mathrm{conv}\{\chi^{\delta(W)} \in \mathbb{R}^E \mid W \subseteq V\} .$$

This polytope has dimension $|E|$. For any edge $e \in E$, the two *trivial inequalities* $0 \leq x_e \leq 1$ define a facet of CUT(G) if and only if e is not contained in a triangle. About some other classes of facets, we quote the following results of Barahona and Mahjoub (1986).

Theorem 8.1. *Let $G = (V, E)$ be a graph.*

(a) *For every cycle $C \subseteq E$ and every set $F \subseteq C$, $|F|$ odd, the* odd cycle inequality

$$x(F) - x(C)\backslash F) := \sum_{e \in F} x_e - \sum_{e \in C \backslash F} x_e \leq |F| - 1$$

is valid for CUT(G); *it defines a facet of* CUT(G) *if and only if C has no chord.*

(b) *For every complete subgraph $K_p = (W, F)$ of G, the* K_p-inequality

$$x(F) \leq \left[\frac{p}{2}\right]\left[\frac{p}{2}\right]$$

is valid for CUT(G); *it defines a facet of* CUT(G) *if and only if p is odd.*

Applications of the max-cut problem arise in statistical mechanics (finding ground states of spin glasses, see chapter 37) and VLSI design (via minimization). Both applications are covered in Barahona et al. (1988). But the max-cut problem comes up also in many other fields. Structural insights from different angles resulted in the discovery of many further (and very large) classes of valid and facet-defining inequalities. Studies on the embeddability of finite metric spaces, for instance, lead to the class of hypermetric inequalities; there are the classes of clique-web, suspended tree, circulant, path-block-cycle, and other inequalities. A comprehensive survey of this line of research can be found in Deza and Laurent (1991); cf. also chapter 41, section 2.

There are a few special cases where it is known that some of the classes of inequalities suffice for a characterization of CUT(G). For example, setting

$$P_C(G) := \{x \in \mathbb{R}^E \mid 0 \leq x_e \leq 1 \text{ for all } e \in E,\ x(F) - x(C \setminus F) \leq |F| - 1 \text{ for all cycles } C \subseteq E \text{ and all } F \subseteq C,\ |F| \text{ odd}\}\ , \qquad (6)$$

Barahona and Mahjoub showed that CUT(G) = $P_C(G)$ holds if and only if G is not contractible to the complete graph K_5. But for a general graph G, the union of all the known classes of inequalities does not provide a complete description of CUT(G) at all.

Separation. Let us review at this point what has been achieved by this polyhedral approach. We started out with a polytope $P(\mathcal{I})$ and found classes of inequalities $A_1x \leq b_1$, $A_2x \leq b_2, \ldots, A_kx \leq b_k$, say, such that all inequalities are valid and many facet-defining for $P(\mathcal{I})$. The classes are huge and thus we are unable to use linear programming in the conventional way by inputting all constraints. Moreover, even if we could solve the LPs, it is not clear whether the results provide helpful information for the solution of our combinatorial problem. Although the situation looks rather bad at this point we have done a significant step towards solving hard combinatorial optimization problems in practice. We will now outline why.

A major issue is to figure out how one can solve linear programs of the form

$$\begin{array}{ll} \text{maximize} & c^{\mathrm{T}}x \\ \text{subject to} & A_1x \leq b_1 \\ & \quad\vdots \\ & A_kx \leq b_k \end{array} \qquad (7)$$

where some of the matrices A_i have a number of rows that is exponential in $|E|$, and are only implicitly given to us. To formulate the answer to this question we introduce the following problem.

Separation problem. Let $Ax \leq b$ be an inequality system and y a vector, determine whether y satisfies all inequalities, and if not, find an inequality violated by y.

Suppose now that we have a class $\mathcal{A}$ of inequality systems $Ax \leq b$. (Example: Consider the class consisting of all odd cycle inequalities for CUT(G), for each graph G.) For each system $Ax \leq b$, let $\varphi := \min(\langle a_i \rangle + \langle \beta_i \rangle)$, where the minimum is taken over all rows $a_ix \leq \beta_i$ of the system. We say that the *optimization problem for $\mathcal{A}$ can be solved in polynomial time* if, for any system $Ax \leq b$ of $\mathcal{A}$ and any vector c, the linear program $\max\{c^{\mathrm{T}}x \mid Ax \leq b\}$ can be solved in time polynomial in $\varphi + \langle c \rangle$, and we say that the *separation problem for $\mathcal{A}$ can be solved in polynomial time* if, for any system $Ax \leq b$ of $\mathcal{A}$ and any vector y, the separation problem for $Ax \leq b$ and y can be solved in time polynomial in φ and $\langle y \rangle$.

Theorem 8.2. *Let $\mathscr{A}$ be a class of inequality systems, then the optimization problem for $\mathscr{A}$ is solvable in polynomial time if and only if the separation problem for $\mathscr{A}$ is solvable in polynomial time.*

For a proof and extensions of this result, see Grötschel et al. (1988).

The idea now is to develop polynomial time separation algorithms for the inequality systems $A_1x \leq b_1, \ldots, A_kx \leq b_k$ in (7). It turns out that this task often gives rise to new and interesting combinatorial problems and that, for many hard combinatorial optimization problems, there are large systems of valid inequalities that can be separated in polynomial time.

We use the max-cut problem again to show how *separation algorithms*, i.e., algorithms that solve the separation problem can be designed. We thus assume that a graph $G = (V, E)$ is given and that we have a vector $y \in \mathbb{R}^E$, $0 \leq y_e \leq 1$ for all $e \in E$. We want to check whether y satisfies the inequalities described in Theorem 8.1.

To solve the *separation problem for the odd cycle inequalities of Theorem* 8.1(a) in polynomial time, we define a new graph $H = (V' \cup V'', E' \cup E'' \cup E''')$ that consists of two copies of G, say $G' = (V', E')$ and $G'' = (V'', E'')$ and the following additional edges E'''. For each edge $uv \in E$ we create the two edges $u'v''$ and $u''v'$. The edges $u'v' \in E'$ and $u''v'' \in E''$ are assigned the weight y_{uv}, while the edges $u'v''$, $u''v' \in E'''$ are assigned the weight $1 - y_{uv}$. For each pair of nodes $u', u'' \in W$, we calculate a shortest (with respect to the weights just defined) path in H. Such a path contains an odd number of edges of E''' and corresponds to a closed walk in G containing u. Clearly, if the shortest of these (u', u'')-paths in H has length less than 1, there exists a cycle $C \subseteq E$ and an edge set $F \subseteq C$, $|F|$ odd, such that y violates the corresponding odd cycle inequality. (C and F are easily constructed from a shortest path.) If the shortest of these (u', u'')-paths has length at least 1, then y satisfies all these inequalities (see Barahona and Mahjoub 1986).

Trivially, for p fixed, one can check all K_p-inequalities in polynomial time by enumeration, but it is now known whether there is a polynomial time algorithm to solve the separation problem for all complete subgraph inequalities of Theorem 8.1(b). In this case one has to resort to *separation heuristics*, i.e., algorithms that try to produce violated inequalities but that are not guaranteed to find one if one exists.

It is a simple matter to show that the integral vectors in the polytope $P_C(G)$, see (7), are exactly the incidence vectors of the cuts of G. This shows that every integral solution of the linear program

$$\begin{aligned} &\text{maximize} \quad c^{\mathrm{T}}x \\ &\quad \text{(i)} \quad 0 \leq x \leq 1 \\ &\quad \text{(ii)} \quad x \text{ satisfies all odd cycle inequalities of Theorem 8.1(a)} \\ &\quad \text{(iii)} \quad x \text{ satisfies all } K_p\text{-inequalities of Theorem 8.1(b)} \end{aligned} \tag{8}$$

is the incidence vector of a cut of G. In particular, the optimum value of (8) (or any subsystem thereof) provides an upper bound for the maximum weight of a cut.

Theorem 8.2 and the exact separation routine for odd cycle inequalities outlined above show that the linear program (8) (without system (iii)) can be solved in polynomial time. So an LP-relaxation of the max-cut problem can be solved in polynomial time that contains (in general) exponentially many inequalities facet-defining for the cut polytope CUT(G). The question now is whether this technique is practical and whether it will help solve max-cut and other hard combinatorial optimization problems.

Outline of a standard cutting plane algorithm. Theorem 8.2 is based on the ellipsoid method. Although the algorithm that proves 8.2 is polynomial, it is not fast enough for practical problem solving. To make this approach usable in practice one replaces the ellipsoid method by the simplex method and enhances it with a number of additional devices. We will sketch the issues coming up here. We assume that, by theoretical analysis, we have found an LP-relaxation such as (7) of our combinatorial optimization problem.

The initial linear program. We group the inequalities of (7) such that the system $A_1x \leqslant b_1$ is not too large and contains those inequalities that we feel should be part of our LP initially.

This selection is a matter of choice. In the max-cut problem, for instance, one would clearly select the trivial inequalities $0 \leqslant x \leqslant 1$. For the large classes of the other inequalities, the choice is not apparent. One may select some inequalities based on heuristic procedures. In the case of the travelling salesman problem, see (9.10) of chapter 30 and section 2 of this chapter, in addition to the trivial inequalities, the degree constraints $x(\delta(v)) = 2$ for all $v \in V$ are self-suggesting. For the packing problem of Steiner trees (a problem coming up in VLSI routing), for example, a structural analysis of the nets to be routed on the chip was used in Grötschel et al. (1992a) to generate "reasonable" initial inequalities. This selection helped to increase the lower bound significantly in the early iterations and to speed up the overall running time.

Initial variables. For large combinatorial optimization problems the number of variables of the LP-relaxation may be tremendous. A helpful trick is to restrict the LPs to "promising" variables that are chosen heuristically. Of course in the end, this planned error has to be repaired. We will show later how this is done. For the travelling salesman problem, for instance, a typical choice are the 2 to 10 nearest neighbors of any node and the edges of several heuristically generated tours. For a 3000 city TSP instance, the number of variables of the initial LP can be restricted from about 4.5 million to less than 10 thousand this way; see Applegate et al. (1994), Grötschel and Holland (1991), and Padberg and Rinaldi (1991) for descriptions of variable reduction strategies for the TSP.

There are further preprocessing techniques that, depending on the type of problem and the special structure of the instances, can be applied. These techniques are vital in many cases to achieve satisfactory running times in practice. Particularly important are techniques for structurally reducing instance sizes by decomposition, for detecting logical dependencies, implicit relations, and bounds that can be used to eliminate variables or forget certain constraints forever. For space reasons we are unable to outline all this here.

Cutting plane generation. The core of a cutting plane procedure is of course the identification of violated inequalities. Assume that we have made our choice of initial constraints and have solved the initial linear program $\max\{c^{\mathrm{T}}x \mid A_1x \leq b_1\}$. In further iterations we may have added additional constraints so that the current linear program has the form

$$\begin{array}{ll} \text{maximize} & c^{\mathrm{T}}x \\ \text{subject to} & Ax \leq b\,. \end{array}$$

We solve this LP and suppose that y is an optimum solution. If y is the incidence vector of a feasible solution of our combinatorial problem we are done. Otherwise we want to check whether y satisfies all the constraints in $A_2x \leq b_2, \ldots, A_kx \leq b_k$. We may check certain small classes by substituting y into all inequalities. But, in general, we will run all the separation routines (exact and heuristic) that we have, to find as many inequalities violated by y as possible. It is a very good idea to use several different heuristics even for classes of inequalities for which exact separation algorithms are available. The reason is that exact routines typically find only a few violated constraints (the most violated ones), while separation heuristics often come up with many more and differently structured constraints.

To keep the linear programs small one does also remove constraints, for instance those that are not tight at the present solution. It is sometimes helpful to keep these in a "pool" since an optimum solution of a later iteration might violate it again, and scanning the pool might be computationally cheaper than running elaborate separation routines (see Padberg and Rinaldi 1991).

In the initial phase of a cutting plane procedure the separation routines may actually produce thousands of violated constraints. It is then necessary to select "good ones" heuristically, again, to keep the LPs at a manageable size, see Grötschel et al. (1984) for this issue.

Another interesting issue is the order in which exact separation routines and heuristics are called. Although that may not seem to be important, running time factors of 10 or more may be saved by choosing a suitable order for these and strategies to give up calling certain separation heuristics. An account of this matter is given in Barahona et al. (1988).

There are more aspects that have to be considered, but we are unable to cover all these topics here. It is important to note that, at the present state of the art, there are still no clear rules as to which of these issues are important or almost irrelevant for a combinatorial optimization problem and its LP relaxation considered. Many computational experiments with data from practical instances of realistic sizes are necessary to obtain the right combination of methods and "tricks".

Pricing variables. In the cutting plane procedure we have now iteratively called the cutting plane generation methods, added violated inequalities, dropped a few constraints and repeated this process until we either found an optimum integral solution or stopped with an optimum fractional solution y for which no violated constraint could be found by our separation routines. Now we have to consider the "forgotten variables". This is easy. For every initially discarded variable we generate the column corresponding to the present linear constraint system and compute its reduced costs by standard LP techniques. If all reduced costs come out with the correct sign we have shown that the present solution is also optimum for the system consisting of all variables. If this is not the case we add all (or some, if there are too many) of the variables with the wrong sign to our current LP and repeat the cutting plane procedure. In fact, using reduced cost criteria one can also show that some variables can be dropped because they can provably never appear in any optimum solution or that some variables can be fixed to a certain value.

Branch-and-cut. This process of iteratively adding and dropping constraints and variables may have to be repeated several times before an optimum solution y of the full LP is found. However, for large instances this technique is by far superior to the straightforward method of considering everything at once. If the final solution y is integral, our combinatorial optimization problem is solved. If it is not integral, we have to resort to branch-and-bound, see section 4. There are various ways to mix cutting plane generation with branching, to use fractional LP-solutions for generating integral solutions heuristically, etc. It has thus become popular to call the whole approach described here *branch-and-cut.*

Clearly, this tremendous theoretical and implementational effort only pays if the bounds for the optimum solution value obtained this way are very good. Computational experience has shown that, in many cases, this is indeed so. We refer the interested readers to more in-depth surveys on this topic such as Grötschel and Padberg (1985), Padberg and Grötschel (1985), and Jünger et al. (1995) for the TSP, or to papers describing the design and implementation of a cutting plane algorithm for a certain practically relevant, hard combinatorial optimization problem. These papers treat many of the issues and little details we were unable to cover here. Among these papers are: Applegate et al. (1994), Grötschel and Holland (1991), Padberg and Rinaldi (1991) for the TSP (the most spectacular success of the cutting plane technique has certainly been achieved here); Barahona et al. (1988) for the max-cut problem with applications to ground states in spin glasses and via minimization in VLSI design; Grötschel et al. (1984) for the linear ordering problem with applications to triangulation of input–output matrices and ranking in sports; Hoffman and Padberg (1993) for the set partitioning problem with applications to airline crew scheduling; Grötschel and Wakabayashi (1989) for the clique partitioning problem with applications to clustering in biology and the social sciences; Grötschel et al. (1992b) for certain connectivity problems with applications to the design of survivable telecommunication networks; Grötschel et al. (1992a) for the Steiner tree packing problem with applications to routing in VLSI design.

The LP-solvers used in most of these cases are advanced implementations of the simplex algorithm such as Bixby's CPLEX or IBM's OSL. Investigations of the use of interior point methods in such a framework are on the way. A number of important issues like addition of rows and columns and postoptimality analysis, warm starts, etc. are not satisfactorily solved yet. But combinations of the two approaches may yield the LP-solver of the future for this approach.

Linear programming in heuristics. So far, we have used linear programming as a dual heuristic, to obtain upper bounds on (say) the maximum value of an integer program. But solving the linear relaxation of an optimization problem also provides primal information, in the sense that it can be used to obtain a (primal) heuristic solution.

Of course, solving the linear relaxation of an integer linear program, we may be lucky and get an integer solution right away. Even if this does not happen, we may find that some of the variables are integral in the optimum solution of the linear relaxation, and we may try to fix these variables at these integral values. This is in general not justified; a notable special case when this can be done was found by Nemhauser and Trotter (1974), who proved the following. Consider a graph $G = (V, E)$ and the usual integer linear programming formulation of the stable set problem:

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in V} x_i \\ \text{subject to} \quad & x_i \geq 0 \ (i \in V) \\ & x_i + x_j \leq 1 \ (ij \in E) \\ & x \text{ integral}. \end{aligned} \tag{9}$$

Let x^* be an optimum solution of the linear relaxation of this problem. Then there exists an optimum solution x^{**} of the integer program such that $x_i^* = x_i^{**}$ for all i for which x_i^* is an integer.

In general, we can obtain a heuristic primal solution by fixing those variables that are integral in the optimum solution of the linear relaxation, and rounding the remaining variables "appropriately". Properties of this heuristic were studied in detail by Raghavan and Thompson (1987). However, it seems that this natural and widely used scheme for a heuristic is not sufficiently analyzed. Here we discuss some results where linear programming combined with appropriate rounding procedures gives a provably good primal heuristic.

A polynomial approximation scheme for bin packing. The following polynomial time bin packing heuristic, due to Fernandez de la Vega and Lueker (1981), has asymptotic performance ratio $1 + \varepsilon$, where $\varepsilon > 0$ is any fixed number. A more refined application of this idea gives a heuristic that packs the weights into $k_{\text{opt}} + O(\log^2(k_{\text{opt}}))$ bins (Karmarkar and Karp 1982).

First, we solve the following restricted version. We are given integers $k, m > 0$, weights $1/k < a_1 < \cdots < a_m \leq 1$ and a multiplicity n_j for each weight a_j. Let k_{opt}

be the minimum number of bins into which n_j copies of a_j $(j = 1, \ldots, m)$ can be packed. Then we can pack the weights into $k_{\text{opt}} + m$ bins, in time polynomial in n and $\binom{m+k}{k}$.

To obtain such a packing, let us first generate all possible combinations (with repetition) of the given weights that fit into a single bin. Since each weight is at least $1/k$, such a combination has at most k elements, and hence the number of different combinations is at most $\binom{m+k}{k}$, and they can be found by brute force. Let $T_1, \ldots, T_N$ be these combinations. Each T_i can be described by an integer vector $(t_1^i, t_2^i, \ldots, t_m^i)$, where t_j^i is the number of times weight a_j appears in combination i.

Consider the following linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} y_i \\ \text{subject to} \quad & y_i \geq 0 \\ & \sum_{i=1}^{N} t_j^i y_i \geq n_j \; (j = 1, \ldots, m) . \end{aligned} \tag{10}$$

Let Y denote the optimum value. Every packing of the given weights into bins gives rise to an (integral) solution of this linear program (y_i is the number of times combination T_i is used), hence

$$k_{\text{opt}} \geq \mathrm{Y} .$$

On the other hand, let y^* be an optimum basic solution of (10), and consider $\lceil y_i^* \rceil$ bins packed with combination T_i. Since at most m of the y_i^* are non-zero, we get a total of $\sum_i \lceil y_i^* \rceil < Y + m \leq k_{\text{opt}} + m$ bins, which clearly accommodate the whole list.

Now let $0 < x_1 \leq x_2 \leq \cdots \leq x_n \leq 1$ be an arbitrary list L of weights, and let $0 < \varepsilon < 1$ be also given. Set $w := \sum_i x_i$, and define l by $x_l < \varepsilon/2 \leq x_{l+1}$ (set $l := 0$ if $x_1 \geq \varepsilon/2$). Set $a_i := x_{l+ih}$ $(i = 1, \ldots, m)$, where $h := \lceil \varepsilon w \rceil$ and $m := \lfloor (n - l)/h \rfloor$. Consider a list L' consisting of h copies of each a_i and $n - l - hm$ copies of 1. Let k'_{opt} be the minimum number of bins into which L' can be packed; by the solution of the restricted problem described above, we can pack L' into $k'_{\text{opt}} + m$ bins in polynomial time. Trivially, we get from this a packing of the weights $x_{l+1}, \ldots, x_n$ into $k'_{\text{opt}} + m$ bins. The remaining (small) weights $x_1, \ldots, x_l$ are packed by FIRST-FIT into the slacks of these bins and, if necessary, into new bins.

To compare the number k_{heur} of bins used this way with k_{opt}, we distinguish two cases. If, in the last step of FIRST-FIT, we had to open a new bin, then every bin (except possibly the last one) is filled up to $1 - \varepsilon/2$, and hence

$$k_{\text{heur}} \leq 1 + \frac{1}{1 - \varepsilon/2} w .$$

Since w is a trivial lower bound on k_{opt}, this shows that

$$k_{\text{heur}} \leq (1 + \varepsilon) k_{\text{opt}} + 1 .$$

So assume that we do not open a new bin in the last phase, and hence we use at most $k'_{\text{opt}} + m$ bins. To compare this with k_{opt}, consider an optimum packing of L. Then from the list L', the h copies of a_1 can be put in the place of $x_{l+h+1}, \ldots, x_{l+2h}$, the h copies of a_2 can be put in the place of $x_{l+2h+1}, \ldots, x_{l+3h}$ etc. At the end we are left with h weights, which we can accommodate using h new bins. Hence

$$k'_{\text{opt}} \leq k_{\text{opt}} + h \,,$$

and so

$$k_{\text{heur}} \leq k'_{\text{opt}} + m \leq k_{\text{opt}} + h + m \,.$$

Since

$$h \leq \varepsilon w + 1 \leq \varepsilon k_{\text{opt}} + 1 \,,$$

and

$$m \leq \frac{n-l}{h} < \frac{w}{\varepsilon/2} \bigg/ (\varepsilon w) = \frac{2}{\varepsilon^2} = \mathrm{O}(1) \,,$$

this proves that the asymptotic performance ratio is at most $1 + \varepsilon$ as claimed.

Blocking sets in hypergraphs with small Vapnik–Červonenkis dimension. The following discussion is based on ideas of Vapnik and Červonenkis, which have become very essential in a number of areas in mathematics (statistics, learning theory, computational geometry). Here we use these ideas to design a randomized heuristic for finding a blocking set in a hypergraph.

Let $(V, \mathcal{H})$ be a hypergraph and consider an optimum fractional blocking set of $\mathcal{H}$, i.e., an optimum solution x^* of the linear program

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in V} x_i \\ \text{subject to} \quad & x_i \geq 0 \; (i \in V) \\ & \sum_{i \in E} x_i \geq 1 \; (E \in \mathcal{H}) \,. \end{aligned} \tag{11}$$

The optimum value of program (11) is the *fractional blocking number* $\tau^* := \tau^*(\mathcal{H})$, which can serve as a lower bound on the *covering* (or blocking) *number* $\tau(\mathcal{H})$. Now we use this linear program to obtain a heuristic solution.

Consider an optimum solution x, and define $p_i = x_i/\tau^*$. Then $(p_i \colon i \in V)$ can be viewed as a probability distribution on V, in which every edge $E \in \mathcal{H}$ has probability at least $1/\tau^*$. Let us generate nodes $v_1, v_2, \ldots$ independently from this distribution, and stop when all edges are covered. It is easy to see that with very large probability we stop in a polynomial number of steps, so this procedure is indeed a (randomized) blocking set heuristic, which we call the random node heuristic. Let t_{heur} be the size of the blocking set produced (this is a random variable). What is the expected performance ratio of the random node heuristic, i.e., the ratio $\mathrm{E}(t_{\mathrm{heur}})/\tau(\mathscr{H})$?

It is clear that if we consider a particular edge E, then it will be hit by one of the first $k\tau^*$ nodes with probability about $1-1/\mathrm{e}^k$. Hence if $k > \ln|\mathscr{H}|$, then the probability that every edge is hit is more than $\frac{1}{2}$. Hence we get the inequality

$$t_{\mathrm{heur}} \leqslant (\ln|\mathscr{H}|)\tau^* \leqslant (\ln|\mathscr{H}|)\tau ,$$

i.e., we obtain the bound $\ln|\mathscr{H}|$ for the performance ratio of the random node heuristic. This is not interesting, however, since the greedy heuristic does better (see Theorem 2.2).

Adopting a result of Haussler and Welzl (1987) from computational geometry (which in turns is an adaptation of the work of Vapnik and Červonenkis in statistics, see Vapnik 1982), we get a better analysis of the procedure. The *Vapnik–Červonenkis dimension* of a hypergraph $(V, \mathscr{H})$ is the size of the largest set $S \subseteq V$ such that for every $T \subseteq S$ there is an $E \in \mathscr{H}$ such that $T = S \cap E$.

Theorem 8.3. *Let $\mathscr{H}$ be a hypergraph with Vapnik–Červonenkis dimension d and fractional blocking number τ^*. The expected size of the blocking set returned by the random node heuristic is at most* $16d\tau^* \log(d\tau^*)$.

Proof. We prove that if we select $N := \lceil 8d\tau^* \log(d\tau^*)\rceil$ nodes from the distribution p, then with probability more than $\frac{1}{2}$, every edge of $\mathscr{H}$ is met. Hence it follows easily that $\mathrm{E}(t_{\mathrm{heur}}) \leqslant 2N$.

The proof is not long but tricky. Choose N further nodes $v_{N+1}, \ldots, v_{2N}$ (independently, from the same distribution). Set $s = N/(2\tau^*)$. Assume that there exists a set $E \in \mathscr{H}$ such that $E \cap \{v_1, \ldots, v_N\} = \emptyset$. Chebychev's inequality gives that for any such edge $E \in \mathscr{H}$,

$$\mathrm{Prob}(|E \cap \{v_{N+1}, \ldots, v_{2N}\}| \geqslant s) > \tfrac{1}{2} .$$

Hence we obtain

$$\begin{aligned}&\mathrm{Prob}(\exists E\colon E \cap \{v_1, \ldots, v_N\} = \emptyset, |E \cap \{v_{N+1}, \ldots, v_{2N}\}| > s)\\ &\quad > \tfrac{1}{2}\,\mathrm{Prob}(\exists E\colon E \cap \{v_1, \ldots, v_N\} = \emptyset) .\end{aligned}$$

We estimate the probability on the left-hand side from above as follows. We can generate a $(2N)$-tuple from the same distribution if we first generate a set S of $2N$ nodes as before, and then randomly permute them. For a given $E \in \mathscr{H}$ that meets S in at least s elements, the probability that after this permutation E avoids the first half of S is at most

$$\frac{\binom{2N-s}{N}}{\binom{2N}{N}} \leqslant \left(1 - \frac{s}{2N}\right)^N < \mathrm{e}^{-s/2} .$$

We do not have to add up this bound for all E, only for all different intersections

$E \cap S$. The number of sets of the form $E \cap S$ is at most

$$\binom{2N}{d} + \binom{2N}{d-1} + \cdots + 1 < (2N)^d ,$$

by the Sauer-Shelah theorem (see chapter 24, section 4). Hence the probability that there is an edge $E \in \mathcal{H}$ that meets S in at least s elements but avoids the first half is at most $(2N)^d \mathrm{e}^{-s/2} < 1/4$.

Hence

$$\mathrm{Prob}(\exists E : E \cap \{v_1, \ldots, v_N\} = \emptyset) < \tfrac{1}{2} . \qquad \square$$

With some care, the upper bound can be improved to (essentially) $\mathrm{O}(d\tau^* \log \tau^*)$, which is best possible in terms of these parameters, see Komlós et al. (1992).

Approximating a cost-minimal schedule of parallel machines. Machine scheduling problems arise in hundreds of versions and are a particular "playground" for approximation techniques. We outline here an LP-based heuristic for the following problem of scheduling parallel machines with costs (this problem is also called the *generalized assignment problem*). Suppose that we have a set J of n independent jobs, a set M of m unrelated machines, and we want to assign each job to one of the machines. Assigning job j to machine i has a certain cost c_{ij} and takes a certain time p_{ij}. Our task is to find an assignment of jobs to machines such that no machine gets more load than a total of T time, and the total cost does not exceed a given bound C, i.e., we look for a job assignment with maximum time load (*makespan*) at most T and cost at most C.

If all the p_{ij} are the same, then this is a weighted bipartite matching problem, and so can be solved in polynomial time. However, for general p_{ij}, the problem is NP-hard. Since there are two parameters (T and C), there are several ways to formulate what an approximate solution means, and there are various algorithms known to find them. Each of these is based on solving a linear relaxation of the problem and then "rounding" the solution appropriately; this technique was introduced by Lenstra et al. (1990). A combinatorially very interesting "rounding" of the solution of the linear relaxation was used by Shmoys and Tardos (1993), which we now sketch.

Consider the following linear program:

$$\begin{aligned} &\min c^{\mathrm{T}}x := \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij}x_{ij} , && \\ &\sum_{j=1}^{m} p_{ij}x_{ij} \leqslant T , && \text{for } i = 1, \ldots, m , \\ &\sum_{i=1}^{m} x_{ij} = 1 , && \text{for } j = 1, \ldots, n , \\ &x_{ij} \geqslant 0 , && \text{for } i = 1, \ldots, m,\ j = 1, \ldots, n , \end{aligned}$$

$$x_{ij} = 0\,, \qquad \text{if } p_{ij} > T, i = 1, \ldots, m, j = 1, \ldots, n\,. \tag{12}$$

Clearly, every integral solution y of (12) with cost $c^{\mathrm{T}}y \leq C$ provides a feasible solution of the generalized assignment problem, and thus, (12) is a natural LP-relaxation of the generalized assignment problem. (The explicit inclusion of the last condition plays an important role in the approximation algorithm.) Let us replace the right hand side T of the first m inequalities of (12) by $2T$ and let us denote this new LP by (12′). Now Shmoys and Tardos prove the following. *If* (12) *has a (possibly fractional) solution* x^* *with cost* $c^* := c^{\mathrm{T}}x^*$ *then* (12′) *has a* $0/1$*-solution with cost* c^*. In other words, if the LP (12) has a solution with cost at most C, then there is an assignment of jobs to machines with the same cost (at most C) and makespan at most $2T$.

The trick is, of course, in "rounding" the real solution x^*. This is done by using x^* to construct an auxiliary bipartite graph and then finding a minimum cost matching in this graph (cf. section 9), which then translates back to an assignment of cost at most C and makespan at most $2T$. These details must be omitted here, but note that the "rounding" involves a nontrivial graph-theoretic algorithm.

9. Changing the objective function

Consider an optimization problem in which the objective function involves some "weights". One expects that if we change the weights "a little", the optimum solutions do not change, or at least do not change "too much". It is surprising how far this simple idea takes us: it leads to efficient algorithms, motivates linear programming, and is the basis of fundamental general techniques (scaling, Lagrangean relaxation, strong polynomiality).

Kruskal's algorithm revisited. Let $G = (V, E)$ be a connected graph and $c: V \rightarrow \mathbb{Z}$, the length function of its edges. We want to find a shortest spanning tree. Clearly, adding a constant to all edges does not change the problem in the sense that the set of optimum solutions remains the same. Thus, we may assume that the lengths are non-negative.

Now let us push this idea just a bit further: we may assume that the shortest edge has length 0. Then it is easy to see that shrinking this edge to a single node does not alter the length of the shortest spanning tree. We can shift the edge-weights again so that the minimum length of the remaining edges is 0; hence, we may contract another edge, etc.

It is easy to see that this algorithm to construct a shortest spanning tree is actually Kruskal's algorithm in disguise: the first edge contracted is the shortest edge; the second, the shortest edge not parallel to the first, etc. (Or is greediness a disguise of this argument?)

Minimum weight perfect matching in a bipartite graph. Given a complete bipartite graph $G=(V,E)$ with bipartition (U,W), where $|U|=|W|$, and a cost function $w: E \to \mathbb{Z}$, we want to find a perfect matching M of minimum weight.

The idea is similar to the one mentioned in the previous section. By adding the same constant to each weight, we assume that all the weights are non-negative. But now we have more freedom: if we add the same constant to the weights of all edges incident with a given node, then the weight of every perfect matching is also shifted by the same constant, and so the set of optimum solutions does not change. Our aim is to use this transformation until a perfect matching with total weight 0 is obtained; this is then triviually optimal.

Let G_0 denote the graph formed by edges with weight 0. Using the unweighted bipartite matching algorithm, we can test whether G_0 has a perfect matching. If this is the case, we are done; else, the algorithm returns a set $X \subseteq U$ such that the set of neighbors $N_{G_0}(X)$ of X is smaller than X. If ε is the minimum weight of any edge from X to $W \backslash N_{G_0}$, we add ε to all the edges out of $N_{G_0}(X)$ and $-\varepsilon$ to all the edges out of X. This transformation preserves the values of the edges between X and $N_{G_0}(X)$, and creates at least one new node connected to X by a 0-edge. Any perfect matching changes its weight by the same value $-\varepsilon|X| + \varepsilon|N_{G_0}(X)| < 0$.

It remains to show that this procedure terminates, and to estimate the number of iterations it takes. One way to show this is to remark that at each iteration, either the maximum size of an all-0 matching increases, or it remains the same, but then the set X returned by the unweighted bipartite matching algorithm (as described in chapter 3) increases. This gives an $O(n^2)$ bound on the number of iterations.

One can read off from this algorithm Egerváry's min–max theorem on weighted bipartite matchings (see chapter 3 for extensions of the algorithm and the theorem to non-bipartite graphs).

Theorem 9.1. *If $G=(V,E)$ is a bipartite graph with bipartition (U,W), where $|U|=|W|$, and $w: E \to \mathbb{Z}$ is a cost function, then the minimum weight of a perfect matching is the maximum of $\sum_{i \in V} \pi_i$,* where *$\pi: V \to \mathbb{Z}$ is a weighting of the nodes such that $\pi_i + \pi_j \leq c_{ij}$ for all $ij \in E$.*

Optimum arborescences. Given a digraph $G=(V,A)$ and a root $r \in V$, an *arborescence* is a spanning tree whose edges are oriented away from r. Let us assume that G contains an arborescence with root r. (This is easily checked.) Let $l: A \to \mathbb{Z}$ be an assignment of "lengths" to the arcs. The *optimum arborescence problem* is to find an arborescence of minimum length (see also chapter 30). An optimum arborescence can be found efficiently by the following algorithm due to Edmonds (1967a).

Again, we may assume that the lengths are non-negative, since this can be achieved by adding a constant to every arc length.

Consider all the arcs of G going into some node $v \neq r$. Any arborescence will contain exactly one of these arcs. Hence we may add a constant to the length of all the arcs going into v without changing the set of optimum arborescences.

For every $v \neq r$, we add a constant to the arcs going into v so that the minimum

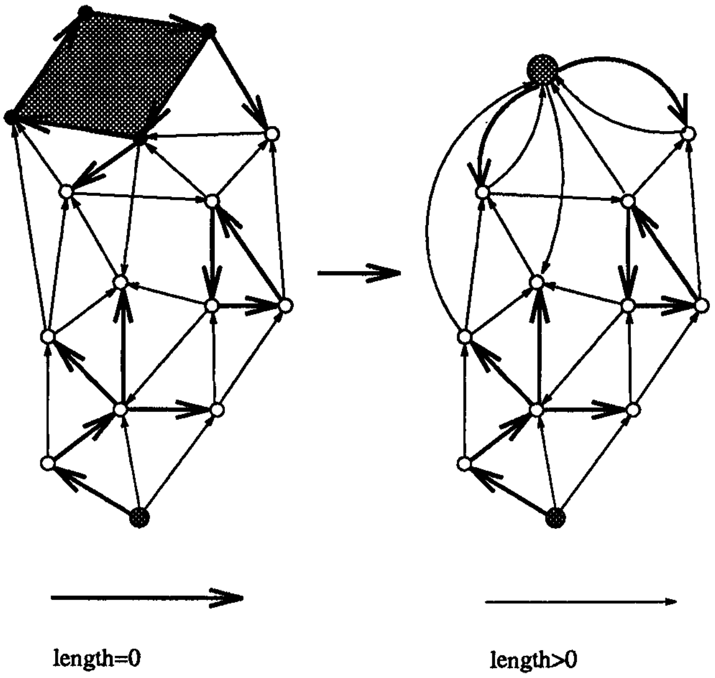

Figure 9.1. The optimum arborescence algorithm.

length of arcs entering any given vertex is 0. Consider the subgraph of all the arcs of length 0. If there exists an arborescence contained in this subgraph, we are done. Otherwise, there must be a cycle of 0-arcs (fig. 9.1).

We contract this 0-cycle, to get the graph G'. It is easy to check that the minimum length of an arborescence in G' is equal to the minimum length of an arborescence in G. Thus we can reduce the problem to a smaller problem, and then proceed recursively. One reduction requires $O(m)$ time, so this leads to an $O(mn)$ algorithm.

Similarly as in the case of the weighted bipartite matching algorithm, we can use this algorithm to derive Fulkerson's optimum arborescence theorem (chapter 2, Theorem 6.4).

More involved applications of the idea of shifting the objective function without changing the optimum solutions include the out-of-kilter method (see chapter 2, section 5).

Scaling I: From pseudopolynomial to polynomial. Consider a finite set E and a collection $\mathcal{F}$ of subsets of E. (in the cases of interest here, $\mathcal{F}$ is implicitly given and may have size exponentially large in $|E|$, e.g., the set of all Eulerian subgraphs of a digraph). Let a weight function $w : E \to \mathbb{Q}$ be also given. Our task is to find a member $X \in \mathcal{F}$ with maximum weight $w(X) := \sum_{e \in X} w(e)$.

Let us round each weight $w(e)$ $(e \in E)$ to the nearest integer. Does this change the set of optimum solutions? On course, it may; but in several situations, connections between the original and the rounded problem can be established so that solving the rounded (and, sometimes, simpler) problem helps in the solution of the original. Before rounding, we may of course multiply each $w(e)$ by the same positive scalar; combined with rounding, this becomes a powerful technique. It was introduced by Edmonds and Karp (1972) to show the polynomial time solvability of the minimum cost flow problem. Since then, scaling has become one of the most fundamental tools in combinatorial optimization, in particular in flow theory (see, e.g., Goldberg et al. 1990).

We illustrate the method on a simple, yet quite general example.

Theorem 9.2. *Let Ψ be a family of hypergraphs and consider the optimization problem* $\max\{w(X)\colon X \in \mathcal{F}\}$ *for members* $(E, \mathcal{F}) \in \Psi$ *and objective functions* $w\colon E \to \mathbb{Z}_+$. *Assume that there exists an "augmentation", i.e., an algorithm that checks whether $X \in \mathcal{F}$ is optimal and if not, returns an $X' \in \mathcal{F}$ with $w(X') > w(X)$; also assume that the augmentation algorithm runs in time polynomial in $\langle w \rangle$. Then the optimization problem can be solved in time polynomial in $\langle w \rangle$.*

Proof. Note that a pseudopolynomial algorithm for this problem is obvious: start with any $X \in \mathcal{F}$ and augment until optimality is achieved. The number of augmentations is trivially bounded by $w(E)$. (Another obvious bound is $2^{|E|}$.) It is easy to construct examples where this trivial algorithm is not polynomial.

To achieve this in polynomial time, let $k := \max_e \lfloor \log w(e) \rfloor$, and define new objective functions $w_j := \lfloor w/2^{k-j} \rfloor$. We solve the optimization problem for the objective function w_0, then for $w_1, \ldots$, finally for $w_k = w$. Since w_0 is 0/1-valued, we can apply the pseudopolynomial algorithm to find the optimizing set.

Assume that we have an optimizing set X_j for w_j. This is of course also optimal for $2w_j$, which is very close to w_{j+1}: we have, for each $e \in E$,

$$2w_j(e) \leq w_{j+1}(e) \leq 2w_j(e) + 1 .$$

Hence, we have for any set $X \in \mathcal{F}$,

$$w_{j+1}(X) \leq 2w_j(X) + n \leq 2w_j(X_j) + n \leq w_{j+1}(X_j) + n .$$

Thus, the set X_j is almost optimal for the objective function w_{j+1}, and the trivial algorithm starting with X_j will maximize w_{j+1} in at most n iterations. Therefore, w will be maximized in a total of $O(nk)$ iterations. □

As an example, consider the problem of finding an Eulerian subdigraph of maximum weight in a directed graph $D = (V, A)$ with arc weights w_a. Then $(A, \mathcal{F})$, where $\mathcal{F} := \{C \subseteq A\colon C \text{ Eulerian}\}$, is a hypergraph. To apply Theorem 9.2 we have to design an augmentation subroutine. Given some Eulerian subdigraph C we construct an auxiliary digraph D_C by reversing the arcs in $A \setminus C$ and changing the signs of the weights on these edges. C is not a maximum weight Eulerian digraph if and only if D_C contains a directed circuit of negative total weight. Such a circuit can be found in polynomial time by shortest path techniques.

For more involved applications of these scaling techniques see chapter 2, section 5.

Scaling II: From polynomial to strongly polynomial. Strong polynomial solvability of a problem is often much more difficult to prove than polynomial solvability; for example, it is not known whether linear programs can be solved in strongly polynomial time. It is therefore remarkable that Frank and Tardos (1987) showed that, for a large class of combinatorial optimization problems, polynomial solvability implies strong polynomial solvability (see also chapter 30).

Theorem 9.3. *Let Ψ be a family of hypergraphs and assume that there exists an algorithm to find* $\max\{\sum_{e\in X} w(e): X \in \mathcal{F}\}$ *for every* $(E, \mathcal{F}) \in \Psi$ *and* $w: E \to \mathbb{Z}$ *in time polynomial in* $\langle w\rangle$. *Then there exists a strongly polynomial algorithm for this maximization problem.*

Proof. The goal is to find an algorithm in which the number of arithmetic operations is bounded by a polynomial in $n = |E|$, and does not depend on $\langle w\rangle$ (we also need that the numbers involved do not grow too wild, but this is easy to check). So the bad case is when the entries of w are very large. Frank and Tardos give an algorithm that replaces w by an integer vector w' such that every entry of w' has at most $O(n^3)$ digits, and w and w' are maximized by the same members of $\mathcal{F}$.

The key step is the construction of the following *diophantine expansion* of the vector w:

$$w = \lambda_1 u_1 + \lambda_2 u_2 + \cdots + \lambda_n u_n ,$$

where $u_1, \ldots, u_n$ are integral vectors with $1 \leq \|u_i\|_\infty \leq 4^{n^2}$, and the coefficients λ_i are rational numbers that decrease very fast:

$$|\lambda_{i+1}| < \frac{1}{2n\|u_{i+1}\|_\infty} |\lambda_i| \quad (i = 1, \ldots, n-1).$$

Such an expansion can be constructed using the simultaneous diophantine approximation algorithm (see chapter 19). This expansion has the property that for any two sets $X, Y \subseteq E$,

$$w(X) \leq w(Y) \iff u_i(X) \leq u_i(Y) \text{ for } i = 1, \ldots, n .$$

Now letting

$$w' := 8^{n^3} u_1 + 8^{n^3-n^2} u_2 + 8^{n^3-2n^2} u_3 + \cdots + 8^{n^2} u_n ,$$

we have for any two sets $X, Y \subseteq E$,

$$w(X) \leq w(Y) \iff w'(X) \leq w'(Y) .$$

Thus w and w' are optimized by the same sets, and we can apply our polynomial time algorithm to maximize w'. Since $\langle w' \rangle = O(n^3)$, this algorithm will be strongly polynomial. □

Applying the result to our previous example, we obtain a strongly polynomial algorithm for the maximum weight Eulerian subdigraph problem. More generally, this technique yields strongly polynomial algorithms, among others, for linear programs with $\{-1, 0, 1\}$-matrices (e.g., for the minimum cost flow problem).

Poljak (1993) applied an even more general version of scaling to show that the single exchange heuristic of the max-cut problem is strongly polynomial for cubic graphs (while exponential for 4-regular graphs). Let $G = (V, E)$ be a graph and $c: E \to \mathbb{Z}$, a weighting of its edges. The idea is that the run of the heuristic is determined if we know, for each node v and each partition of the edges incident with v into two classes, which class has larger weight. So we consider a family of inequalities, each of which is of the type

$$x_i + x_j \geq x_k \quad \text{or} \quad x_i + x_j + 1 \leq x_k \tag{13}$$

(where i, j and k are three edges adjacent to a node). We know that this system has a solution (the original weights). Poljak proves that then the system has an integral solution with $1 \leq |x_i| \leq 2|V| - 1$ for all i. Replacing the original weights with these new weights, the single exchange heuristic runs as before, but now it clearly terminates in $O(|V|^2)$ time.

Scaling III: *Heuristics*. Recall from section 5 that the 0/1-knapsack problem is NP-hard, but it is polynomially solvable if the weight coefficients a_i are given in unary notation. This fact was combined with a scaling technique by Ibarra and Kim (1975) to design a fully polynomial approximation scheme for the knapsack problem.

Fix any $\varepsilon > 0$. We may assume that $c_1 \geq c_2 \geq \cdots \geq c_n$. Let C_{opt} denote the optimum value of the knapsack, and let C be an upper bound on C_{opt}; a good value for C can, e.g., be found by running the greedy heuristic for the knapsack problem, for which Theorem 2.7 gives

$$\frac{C}{2} \leq C_{\text{opt}} \leq C .$$

Let $0 \leq m \leq n$ be the largest index such that $c_m > \varepsilon C/4$, and define

$$\bar{c}_i = \left\lceil \frac{8c_i}{\varepsilon^2 C} \right\rceil .$$

Clearly $\bar{c}_i \leq 8/\varepsilon^2$, and so these numbers are polynomially bounded in $1/\varepsilon$.

The idea is to solve a knapsack problem omitting the "small" weights and replacing the "big" weights $c_1, \ldots, c_m$ by their unary approximations $\bar{c}_1, \ldots, \bar{c}_m$. Then we use the "small" weights to fill up greedily as much of the remaining space as possible.

For every integral value d with $0 \leq d \leq 8/\varepsilon^2$, we determine a solution x^d of the knapsack problem, by solving the following auxiliary optimization problem:

$$\begin{aligned}&\text{minimize} \quad \sum_{i=1}^{m} a_i x_i \\ &\text{subject to} \quad \sum_{i=1}^{m} \bar{c}_i x_i = d\,, \\ &\qquad x_i \in \{0,1\}, i = 1, \ldots, m\,. \end{aligned} \tag{14}$$

This is basically a subset-sum problem with a linear objective function and can be solved, by the same dynamic programming argument, in polynomial time. (In fact, we get the optimum solution for all values of d in a single run, which takes $O(n/\varepsilon^2)$ time for the execution of all problems (14).) Let $x_1^d, \ldots, x_m^d$ be the solution found (if any exists).

Now we choose the remaining variables. These variables $x_{m+1}, \ldots, x_n$ must satisfy the following constraints:

$$\begin{aligned}&\sum_{i=m+1}^{n} a_i x_i \leq b - \sum_{i=1}^{m} a_i x_i^d\,, \\ &x_i \in \{0,1\}\,, \end{aligned} \tag{15}$$

and we want to maximize

$$\sum_{i=m+1}^{n} c_i x_i\,.$$

If the right-hand side in (15) is non-negative, then this is just another (auxiliary) knapsack problem, which we solve by the greedy algorithm in $O(n)$ time for each d (thus using $O(n/\varepsilon^2)$ time in total). Let $x_{m+1}^d, \ldots, x_n^d$ be the solution of the knapsack obtained (if it exists, i.e., if (15) has non-negative right-hand side).

Theorem 9.4. *For at least one d with $0 \leq d \leq 8/\varepsilon^2$, the solution $(x_1^d, \ldots, x_n^d)$ exists, and*

$$\sum_{i=1}^{n} c_i x_i^d \geq (1-\varepsilon) C_{\text{opt}}\,.$$

Proof. Let $y_1, \ldots, y_n$ be a (true) optimum solution of the original knapsack problem, and consider the value

$$d := \sum_{i=1}^{m} \bar{c}_i y_i\,.$$

Clearly

$$d \leq \frac{8}{\varepsilon^2 C} \sum_{i=1}^{m} c_i y_i \leq \frac{8}{\varepsilon^2}\,.$$

Fix this choice of d. Then trivially $x_1^d, \ldots, x_m^d$ exists, and by their optimality for the auxiliary subset-sum problem (14), we have

$$\sum_{i=1}^{m} a_i x_i^d \leq \sum_{i=1}^{m} a_i y_i \leq \sum_{i=1}^{n} a_i y_i \leq b\,.$$

Thus for this d, (15) has non-negative right-hand side and the solution $(x_1^d, \ldots, x_n^d)$ exists. Moreover,

$$\begin{aligned}\sum_{i=1}^{m} c_i x_i^d &\geq \frac{\varepsilon^2 C}{8} \sum_{i=1}^{m} \bar{c}_i x_i^d = \frac{\varepsilon^2 C}{8} \sum_{i=1}^{m} \bar{c}_i y_i \\ &\geq \sum_{i=1}^{m} c_i y_i - \frac{\varepsilon^2 C}{8} \sum_{i=1}^{m} y_i\,.\end{aligned} \tag{16}$$

Here

$$\sum_{i=1}^{m} y_i \leq \frac{1}{c_m} \sum_{i=1}^{m} c_i y_i \leq \frac{4}{\varepsilon C} C_{\text{opt}}\,,$$

and so

$$\sum_{i=1}^{m} c_i x_i^d \geq \sum_{i=1}^{m} c_i y_i - \frac{\varepsilon}{2} C_{\text{opt}}\,.$$

Furthermore, observe that $(y_{m+1}, \ldots, y_n)$ is a solution of (15), and hence by Theorem 2.7,

$$\sum_{i=m+1}^{n} c_i x_i^d \geq \sum_{i=m+1}^{n} c_i y_i - c_{m+1} \geq \sum_{i=m+1}^{n} c_i y_i - \frac{\varepsilon}{2} C_{\text{opt}}\,.$$

Thus

$$\sum_{i=1}^{n} c_i x_i^d \geq \sum_{i=1}^{m} c_i y_i - \varepsilon C_{\text{opt}} = (1-\varepsilon) C_{\text{opt}}\,. \qquad \square$$

Lagrangian relaxation. Consider the (symmetric) travelling salesman problem again. For any vertex v, every tour uses two edges adjacent to v. Hence if we add the same constant to the length of every edge incident with v, we shift the value of every tour by the same number, and hence the problem remains essentially unchanged. By doing so for every vertex, we may bring the problem to a nicer form.

So far, this is the same idea as in the weighted bipartite matching algorithm above. Unfortunately, this does not lead to a complete solution; we cannot in general obtain an all-0 tour by shifting edge-weights like this. But we may use this method to improve dual heuristics. We have seen that a minimum length of a 1-tree is an easily computable lower bound; let us shift lengths so that this lower bound is maximized. This way we obtain a very good dual heuristic due to Held and Karp (1970).

It is not immediate how this new optimization problem can be solved. To describe a method, recall from matroid theory (chapter 11) that the convex hull of 1-trees with vertex set V is described by the constraints

$$\begin{aligned} &x_e \geqslant 0\,, \\ &x(A) \leqslant r(A)\,, \quad A \subseteq E\,, \\ &x(E) = n\,, \end{aligned} \tag{17}$$

Here $r(A)$ is the rank in the matroid whose bases are the 1-trees: if $c(A)$ is the number of connected components in (V, A), then

$$r(A) = \begin{cases} n - c(A) + 1\,, & \text{if } A \text{ contains a circuit}\,, \\ n - c(A) = |A|\,, & \text{if } A \text{ contains no circuits}\,. \end{cases}$$

If we want to restrict the feasible solutions to allow only tours, a natural step is to write up the degree constraints:

$$\sum_{j \in V \setminus \{i\}} x_{ij} = 2 \quad (i \in V)\,. \tag{18}$$

The objective function is

$$\text{minimize} \sum_e c(e)x_e\,.$$

The integral solutions of (17) and (18) are exactly the tours; if we drop the integrality constraints, we obtain a relaxation.

While it is trivial to minimize any objective function subject to constraints (17) using the greedy algorithm, constraints (18) spoil this nice structure. So let us get rid of the constraints (18) by multiplying them by appropriate multipliers λ_i, adding their left-hand sides to the objective function, and omitting them from the constraint set. Note that this leads to shifting the lengths of edges at nodes, as described above.

For any fixed choice of the multipliers, adding the left-hand sides of an equality constraint to the objective function does not change the problem (the objective function is shifted by the right-hand side); but then, dropping the constraint may decrease the optimum value. Can we choose the multipliers so that the optimum does not change? The answer is yes, and it is worth formulating the generalization of the Duality theorem of linear programming that guarantees this.

Theorem 9.5. *Consider a linear program with constraints split into two classes*:

$$\begin{aligned} \text{minimize} \quad & c^{\mathrm{T}}x \\ \text{subject to} \quad & Ax \geqslant a \\ & Bx \geqslant b\,. \end{aligned} \tag{19}$$

Then the optimum value of this program is the same as the optimum of the *following min–max problem*:

$$\max_y \{\min_x\{(c^{\mathrm{T}} - y^{\mathrm{T}}B)x \mid Ax \geq a\} + y^{\mathrm{T}}b, y \geq 0\} . \tag{20}$$

Similarly as in the Duality theorem, one can allow equations among the constraints, and then the corresponding multipliers y are unconstrained.

For any particular choice of the vector y, the minimum

$$\phi(y) := \min_x\{(c^{\mathrm{T}} - y^{\mathrm{T}}B)x \mid Ax \geq a\} + y^{\mathrm{T}}b$$

is a lower bound on the optimum value of (19). It is not difficult to see that the function ϕ, called the *Lagrange function* of (19), is a concave function, and hence various methods (subgradient, ellipsoid) are available to compute its maximum. Note that as long as we are only using this as a dual heuristic, we do not have to solve this problem to optimality: any reasonable y provides a lower bound.

Applying this technique to the travelling salesman problem, often rather good lower bounds are obtained. For example, the optimum value of the Lagrange function of the 52-city TSP of fig. 2.1 is equal to the length of the shortest tour.

It is worth mentioning that the Lagrangian relaxation method gives a result about exact solutions too.

Theorem 9.6. *Let $P \subseteq \mathbb{R}^n$ be a polytope and assume that every linear objective function $c^{\mathrm{T}}x$ ($c \in \mathbb{Z}^n$) can be minimized over P in time polynomial in $\langle c \rangle$. Then for every matrix $A \in \mathbb{Z}^{m \times n}$, and vectors $a \in \mathbb{Z}^m$ and $c \in \mathbb{Z}^n$, the minimum*

$$\min_x \{c^{\mathrm{T}}x \mid x \in P, Ax \leq a\}$$

can be computed in time polynomial in $\langle A \rangle + \langle a \rangle + \langle c \rangle$.

In other words, adding a few constraints to a nice problem does not spoil it completely. While this result could also be derived by other means (e.g., by the ellipsoid method), the Lagrangian approach is computationally much better if the number m of additional constraints is small.

10. Matrix methods

Determinants and matchings. Let G be a graph with n nodes. In chapter 3, a randomized algorithm (cf. Edmonds 1967b, and Lovász 1979) is described that decides if a graph G has a perfect matching. The method is based on the fact, proved by Tutte (1947) that $\det A(G,x)$ is identically 0 if and only if G has no perfect matching, where $A(G,x)$ is the skew symmetric $n \times n$ matrix defined by

$$A(G,x)_{ij} = \begin{cases} x_{ij}\,, & \text{if } ij \in E(G) \text{ and } i < j\,, \\ -x_{ij}\,, & \text{if } ij \in E(G) \text{ and } i > j\,, \\ 0\,, & \text{otherwise}\,. \end{cases}$$

Generating random values for x_{ij}, and computing the determinant, we obtain a randomized matching algorithm. The following simple lemma, due to Schwartz (1980), can be used to estimate the probability of error.

Lemma 10.1. *Let $f(x_1, \ldots, x_p)$ be a polynomial, not identically* 0, *in which each variable has degree at most k. Choose the x_i independently from the uniform distribution on $\{0, 1, \ldots, N-1\}$. Then*

$$\operatorname{Prob}(f(x_1, \ldots, x_p) = 0) \leq \frac{k}{N}.$$

Since an $n \times n$ determinant can be evaluated in $O(n^{2.39\cdots})$ time (Coppersmith and Winograd 1982), this randomized algorithm has a better running time than the best deterministic one, whose time complexity is $O(n^{5/2})$ (Even and Kariv 1975).

We recall two variants of the determinant-based matching algorithm from chapter 3. The algorithm above determines whether a given graph has a perfect matching; but it does not give a perfect matching. To actually find a perfect matching, we can delete edges until we get a graph G_0 with a perfect matching such that deleting any further edge results in a graph with no perfect matching. Clearly, G_0 is a perfect matching itself.

Instead of this pedestrian procedure, Mulmuley et al. (1987) found an elegant randomized algorithm that finds a perfect matching at the cost of a single matrix inversion. The method is based on the following nice probabilistic lemma.

Lemma 10.2. *Let $(E, \mathscr{H})$ be a hypergraph and assign to each $e \in E$ a random weight w_e from the uniform distribution over $\{1, \ldots, 2|E|\}$. Then with probability at least $\frac{1}{2}$, the edge with minimum weight is unique.*

This lemma implies that if we substitute $x_{ij} = 2^{y_{ij}}$ in $A(G, x)$ (where each y_{ij} is uniformly chosen from $\{0, \ldots, 2n^2\}$) and then invert the resulting matrix A then, with probability at least $\frac{1}{2}$, those entries in the Schur product $A^{-1} \circ A$ having an odd numerator form a perfect matching. (The Schur product $C = A \circ B$ of two $n \times n$ matrices is defined by $C_{ij} = A_{ij}B_{ij}$.)

A special value of this method is that it is parallelizable, using polynomially many processors and polylog time. This depends on Csánky's theorem (see chapter 29, section 5) that gives an NC-algorithm for determinant computation and matrix inversion. Every known NC-algorithm for the perfect matching problem is randomized and uses determinant computation.

Given a graph $G = (V, E)$, an integer k, and $F \subseteq E$, the *exact matching problem* is to determine if there exists a perfect matching M in G such that $|M \cap F| = k$. This problem is not known to be in P, but it is easily solved in randomized polynomial time by the determinant method. Consider the matrix $A(G, x)$, and substitute yz_{ij} for x_{ij} if $ij \in F$, where y is a new variable. This way we obtain a matrix $A(G, z, y)$. Then Tutte's theorem on determinants and matchings can be extended as follows.

Theorem 10.1. *The coefficient of y^k in the Pfaffian* $\mathrm{Pf}(A(G, z, y))$ *is not identically* 0 *in the variables z iff there exists a perfect matching M such that $|M \cap F| = k$.*

This theorem suggests the following algorithm: Substitute random integers $\bar{z}_{ij} \in \{0, \ldots, N-1\}$ in $A(G, y, z)$. The value of $\det A(G, y, \bar{z})$ is a polynomial in y, and all its coefficients can be computed in polynomial time. Compute $\mathrm{Pf}(A(G, y, \bar{z})) = \sqrt{\det A(G, y, \bar{z})}$, which is also a polynomial in y by definition. The coefficient of y^k gives the answer. (See chapter 3, section 7 for other applications of this idea.)

Determinants and connectivity. The method of reducing a combinatorial optimization problem to checking a polynomial (usually determinantal) identity and then solving this in randomized polynomial time via Schwartz's lemma 10.1 is not restricted to matching theory. Chapter 36 contains examples where this method is used in electrical engineering and statics. The papers Linial et al. (1988) and Lovász et al. (1989) contain various algorithms to determine the connectivity of a graph along these lines. Let us formulate one of these. Let G be a graph and consider, for each (unordered) pair ij with $i = j$ or $ij \in E(G)$, a variable x_{ij}. Let $B(G, x)$ be the matrix

$$B(G, x)_{ij} = \begin{cases} x_{ij}, & \text{if } i = j \text{ or } ij \in E(G), \\ 0, & \text{otherwise}. \end{cases}$$

Theorem 10.2. *The graph G is k-connected iff*

$$\textit{no } (n-k) \times (n-k) \textit{ subdeterminant of } B(G, x) \textit{ is identically } 0. \qquad (*)$$

This theorem suggests the following randomized k-connectivity test: substitute in $B(G, x)$ independent random integers from, say, $\{0, \ldots, 2^n\}$, and check condition (*). Unfortunately, there is no polynomial time algorithm known to check (*) for a general matrix; however, due to the very special structure of $B(G, x)$, it suffices to check only a "few" subdeterminants. Let us select, for each vertex i, a set A_i of $k-1$ neighbors (if a node has degree less than $k-1$ then the graph is clearly not k-connected). Then the following can be shown: if G is not k-connected, then one of the subdeterminants of $B(x)$, obtained by deleting the rows belonging to some $A_i \cup \{i\}$ and the columns belong to some $A_j \cup \{j\}$, is identically 0.

This leads to the evaluation of $O(n^2)$ determinants of size $(n-k) \times (n-k)$. With a little care, one can reduce this number to $O(nk)$. For $k < n/2$, it is worth inverting the matrix $B(G, x)$ and then check $O(nk)$ subdeterminants of size $k \times k$ using Jacobi's theorem (see chapter 31).

Semidefinite optimization. Polyhedral combinatorics can be viewed as a theory of linear inequalities valid for the incidence vectors of various set-systems. It is quite natural to ask for quadratic inequalities (and, of course higher degree inequalities) valid for these incidence vectors. This idea leads to real algebraic geometry and its study has just begun.

At first sight it seems that we are getting too much too easily. Let $G = (V, E)$ be a graph, $V = \{1, \dots, n\}$, and consider the following system of equations:

$$x_i^2 = x_i \quad \text{for every node } i \in V\,, \tag{21}$$

$$x_i x_j = 0 \quad \text{for every edge } ij \in E\,. \tag{22}$$

Trivially, the solutions of (21) are precisely the 0–1 vectors, and so the solutions of (21)–(22) are precisely the incidence vectors of stable sets. Unfortunately, there is little known about the solutions of systems of quadratic equations. In fact, this construction shows that even the solvability of such a simple system of quadratic equations (together with a linear equation $\sum_i x_i = \alpha$) is NP-hard.

However, we can use this system to derive some other constraints. (21) implies that for every node i,

$$x_i = x_i^2 \geq 0\,, \qquad 1 - x_i = (1 - x_i)^2 \geq 0\,. \tag{23}$$

Using this, (22) implies that for every edge ij,

$$1 - x_i - x_j = 1 - x_i - x_j + x_i x_j = (1 - x_i)(1 - x_j) \geq 0\,. \tag{24}$$

So we can derive the edge constraints from (21)–(22) formally. We can also derive the clique constraints. Assume that nodes $1, \dots, k$ induce a complete subgraph. We start with the trivial inequality

$$(1 - x_1 - \cdots - x_k)^2 \leq 0\,.$$

Expanding, we get

$$1 + \sum_{i=1}^{k} x_i^2 - 2\sum_{i=1}^{k} x_i + 2\sum_{i \neq j} x_i x_j \geq 0\,.$$

Here the first sum is just $\sum_i x_i$ by (21) and the third sum is 0 by (22), so we get

$$1 - x_1 - \cdots - x_k \geq 0\,. \tag{25}$$

In the special case when the graph is perfect, we obtain all constraints for the stable set polytope STAB(G) in a single step. (STAB(G) is the convex hull of all incidence vectors of stable sets of G.)

The algorithmic significance of this observation is that it leads to a polynomial time algorithm to compute the stability number of a perfect graph. By general consequences of the ellipsoid method (see chapter 30), it suffices to design a polynomial time algorithm that checks whether a given inequality

$$\sum_i c_i x_i \leq \gamma \tag{26}$$

is valid for STAB(G). By the above arguments, it suffices to check whether (26) can be derived from (21) and (22) as above. Formalizing, this leads to the question: do there exist real multipliers μ_i $(i \in V)$ and λ_{ij} $(ij \in E)$, and linear polynomials $l_1, \ldots, l_m$ such that

$$\sum_{k=1}^{m} l_k^2 + \sum_{i=1}^{n} \mu_i(x_i^2 - x_i) + \sum_{ij \in E} \lambda_{ij} x_i x_j = \gamma - \sum_{i=1}^{n} c_i x_i .$$

This is equivalent to saying that there exist λ's and μ's such that the $(n+1) \times (n+1)$ matrix $P = (p_{ij})$ defined by

$$p_{ij} := \begin{cases} \gamma\,, & \text{if } i = j = 0\,, \\ (\mu_i - c_i)/2\,, & \text{if } j = 0, i > 0\,, \\ (\mu_j - c_j)/2\,, & \text{if } i = 0, j > 0\,, \\ -\mu_i\,, & \text{if } i = j > 0\,, \\ \lambda_{ij}/2\,, & \text{if } ij \in E\,, \\ 0\,, & \text{otherwise}\,, \end{cases}$$

is positive semidefinite.

More generally, we can consider optimization problems of the type

$$\begin{array}{ll} \text{maximize} & \sum_i c_i y_i \\ \text{subject to} & P(y) \text{ is positive definite}\,, \end{array} \tag{27}$$

where $P(y)$ is a matrix in which every entry is a linear function of the y_i. Grötschel et al. (1981) describe a way, using the ellipsoid method, to solve such problems to arbitrary precision in polynomial time. (The key facts are that the feasible domain is convex and to check whether a given x satisfies the constraints can be done using Gaussian elimination; we ignore numerical difficulties here that are non-trivial.) The Duality theorem can also be extended to such semidefinite programs (see Wolkowitz 1981). Recently Alizadeh (1992) showed that Karmarkar's interior point method can also be extended to such semidefinite programs in a very natural way, which is much more promising from a practical point of view.

The method sketched here can be used to generate other classes of inequalities for STAB(G) and to show their polynomial time solvability. It is not restricted to the stable set problem either; in fact, it can be applied to any 0–1 optimization problem. Interesting applications of a related method to the max-cut problem were given by Delorme and Poljak (1990) (see also Mohar and Poljak 1990).

Both of these semidefinite relaxations can be combined with randomized rounding methods (section 8). This results in interesting approximation algorithms for the max-cut problem (Goemans and Williamson 1994) and for chromatic number (Karger et al. 1994).

These methods can be extended from quadratic to higher-order inequalities.

For these extensions, see Lovász and Schrijver (1990) and Sherali and Adams (1990).

References

Alizadeh, F.

[1992] Combinatorial optimization with semi-definite matrices, in: *Integer Programming and Combinatorial Optimization, Proc. IPCO '92,* eds. E. Balas, G. Cornuéjols and R. Kannan (Carnegie Mellon University Printing, Pittsburgh, PA) pp. 385–405.

Alon, N., R.A. Duke, H. Lefmann, V. Rödl and R. Yuster

[1994] The algorithmic aspects of the Regularity Lemma, *J. Algorithms* **16**, 80–109.

Applegate, D., and R. Kannan

[1990] Sampling and integration of near logconcave functions, in: *Proc. 23th ACM STOC,* pp. 156–163.

Applegate, D., R. Bixby, V. Chvátal and W. Cook

[1994] *Finding Cuts in the TSP,* Preliminary report.

Bachem, A., M. Grötschel and B. Korte

[1983] *Mathematical Programming: The State of the Art* (Springer, Heidelberg).

Barahona, F., and A.R. Mahjoub

[1986] On the cut polytope, *Math. Programming* **36**, 157–173.

Barahona, F., M. Grötschel, M. Jünger and G. Reinelt

[1988] An application of combinatorial optimization to statistical physics and circuit layout design, *Oper. Res.* **36**, 493–513.

Bellman, R.

[1957] *Dynamic Programming* (Princeton University Press, Princeton, NJ).

Bentley, J.L.

[1992] Fast algorithms for geometric traveling salesman problems, *ORSA J. Comput.* **4**, 387–411.

Christofides, N.

[1976] *Worst-case Analysis of a new Heuristic for the Travelling Salesman Problem,* Technical Report (Graduate School of Industrial Adminstration, Carnegie-Mellon University, Pittsburgh, PA).

Chvátal, V., and G. Klincsek

[1980] Finding largest convex subsets, *Congress. Numerantium* **29**, 453–460.

Coffman Jr, E.G., M.R. Garey and D.S. Johnson

[1984] Approximation algorithms for bin-packing – an updated survey, in: *Algorithm Design for Computer System Design,* eds. G. Ausiello, M. Lucertini and P. Serafini (Springer, New York) pp. 49–106.

Coppersmith, D., and S. Winograd

[1982] On the asymptotic complexity of matrix multiplication, *SIAM J. Comput.* **11**, 472–492.

Cormen, T.H., C.E. Leiserson and R.L. Rivest

[1990] *Introduction to Algorithms* (MIT Press, Cambridge, MA).

Delorme, C., and S. Poljak

[1993] Laplacian eigenvalues and the maximum cut problem, *Math. Programming A* **62**, 557–574.

Deza, M., and M. Laurent

[1991] *A Survey of the known Facets of the Cut Cone,* Report No. 91722-OR (Forschungsinstitut für Diskrete Mathematik, Universität Bonn).

Dyer, M., and A. Frieze

[1992] Computing the volume of convex bodies: A case where randomness provably helps, in: *Probabilistic Combinatorics and Its Applications,* ed. B. Bollobás, *Proc. Symp. Appl. Math.* **44**, 123–170.

Dyer, M., A. Frieze and R. Kannan

[1991] A random polynomial time algorithm for approximating the volume of convex bodies, *J. Assoc. Comput. Mach.* **38**, 1–17.

Edmonds, J.

[1967a] Optimum branchings, *J. Res. Nat. Bur. Standards B* **71**, 233–240.

Edmonds, J., and R.M. Karp

[1972] Theoretical improvements in algorithmic efficiency for network flow problems, *J. Assoc. Comput. Mach.* **19**, 248–264.

Edmonds, J.R.

[1967b] Systems of distinct representatives and linear algebra, *J. Res. Nat. Bur. Standards B* **71**, 241–247.

Erickson, R.E., C.L. Monma and A.F. Veinott

[1987] Send-and-split method for minimum concave-cost network flows, *Math. Oper. Res.* **12**, 634–664.

Even, S., and O. Kariv

[1975] An $O(n^{5/2})$ algorithm for maximum matching in general graphs, in: *16th Annu. Symp. on Foundations of Computer Science* (IEEE Computer Society Press, New York) pp. 100–112.

Fernandez de la Vega, W., and G.S. Lueker

[1981] Bin packing solved within $1+\varepsilon$ in linear time, *Combinatorica* **1**, 349–355.

Ford Jr, L.R., and D.R. Fulkerson

[1962] *Flows in Networks* (Princeton University Press, Princeton, NJ).

Frank, A., and É. Tardos

[1987] An application of simultaneous Diophantine approximation in combinatorial optimization, *Combinatorica* **7**, 49–65.

Garey, M.R., R.L. Graham, D.S. Johnson and A. Yao

[1976] Resource constrained scheduling as generalized bin packing, *J. Combin. Theory A* **21**, 257–298.

Goemans, M.X., and D.P. Williamson

[1994] .878-Approximation algorithms for MAX CUT and MAX 2SAT, in: *Proc. 26th ACM Symp. on Theory of Computing* (ACM, New York) pp. 422–431.

Goldberg, M.V., É. Tardos and R.E. Tarjan

[1990] Network flow algorithms, in: *Paths, Flows, and VLSI-Layout*, eds. B. Korte, L. Lovász, H.J. Prömel and A. Schrijver (Springer, Heidelberg) pp. 101–164.

Gondran, M., and M. Minoux

[1979] *Graphes et Algorithmes* (Editions Eyrolles, Paris).

Graham, R.L., and P. Hell

[1985] On the history of the minimum spanning tree problem, *Ann. Hist. Comput.* **7**, 43–57.

Grötschel, M., and O. Holland

[1991] Solution of large-scale symmetric travelling salesman problems, *Math. Programming* **51**, 141–202.

Grötschel, M., and M.W. Padberg

[1985] Polyhedral theory, in: *The Travelling Salesman Problem: A Guided Tour through Combinatorial Optimization*, eds. E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (Wiley, Chichester) pp. 251–305.

Grötschel, M., and Y. Wakabayashi

[1989] A cutting plane algorithm for a clustering problem, *Math. Programming B* **45**, 59–96.

Grötschel, M., L. Lovász and A. Schrijver

[1981] The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica* **1**, 169–197.

Grötschel, M., M. Jünger and G. Reinelt

[1984] A cutting plane algorithm for the linear ordering problem, *Oper. Res.* **32**, 1195–1220.

Grötschel, M., L. Lovász and A. Schrijver

[1988] *Geometric Algorithms and Combinatorial Optimization* (Springer, Heidelberg).

Grötschel, M., A. Martin and R. Weismantel

[1992a] Packing Steiner trees: a cutting plane algorithm and computational results, Preprint 92–9 (Konrad-Zuse-Zentrum für Informationstechnik Berlin), *Math. Programming*, to appear.

Grötschel, M., C. Monma and M. Stoer

[1992b] Computational results with a cutting plane algorithm for designing communication networks with low connectivity constraints, *Oper. Res.* **40**, 309–330.

Haken, A., and M. Luby

[1988] Steepest descent can take exponential time for symmetric connection networks, *Complex Systems* **2**, 191–196.

Haussler, D., and E. Welzl
[1987] ε-nets and simplex range queries, *Discrete Comput. Geom.* **2**, 127–151.

Held, M., and R.M. Karp
[1970] The travelling salesman problem and minimum spanning trees, *Oper. Res.* **18**, 1138–1162.

Hoffman, K.L., and M. Padberg
[1993] Solving airline crew-scheduling problems by branch-and-cut, *Manage. Sci.* **39**, 657–682.

Holley, R., and D. Stroock
[1988] Simulated annealing via Sobolev inequalities, *Comm. Math. Phys.* **115**, 553–569.

Holley, R., S. Kusuoka and D. Stroock
[1989] Asymptotics of the spectral gap with applications to the theory of simulated annealing, *J. Funct. Anal.* **83**, 333–347.

Hurkens, C.A.J., L. Lovász, A. Schrijver and É. Tardos
[1988] How to tidy up your set-system? in: *Combinatorics, Proc. Coll. Eger 1987,* eds. L. Lovász and V.T. Sós, *Colloq. Math. Soc. János Bolyai* **52**, 309–314.

Ibarra, O.H., and C.E. Kim
[1975] Fast approximation algorithms for the knapsack and sum of subset problems, *J. Assoc. Comput. Mach.* **22**, 463–468.

Jenkyns, T.
[1976] The efficacy of the "greedy" algorithm, in: *Proc. 7th South-Eastern Conf. on Combinatorics, Graph Theory and Computing* (Utilitas Math., Winnipeg) pp. 341–350.

Jerrum, M.R.
[1992] Large cliques elude the Metropolis process, *Rand. Struetures Algorithms* **3**, 347–359.

Jerrum, M.R., and A.J. Sinclair
[1989] Approximating the permanent, *SIAM J. Comput.* **18**, 1149–1178.

Jerrum, M.R., L.G. Valiant and V.V. Vazirani
[1986] Random generation of combinatorial structures from a uniform distribution, *Theor. Comput. Sci.* **43**, 169–188.

Johnson, D.S.
[1973] *Near-optimal allocation algorithms,* Ph.D. Dissertation (MIT, Cambridge, MA).
[1974] Approximation algorithms for combinatorial problems, *J. Comput. System. Sci.* **9**, 256–298.
[1990] Local optimization and the travelling salesman problem, in: *Proc. 17th Coll. on Automata, Languages and Programming* (Springer, Heidelberg) 446–461.

Johnson, D.S., C.H. Papadimitriou and M. Yannakakis
[1988] How easy is local search? *J. Comput. System Sci.* **37**, 79–100.

Johnson, D.S., C.R. Aragon, L.A. McGeoch and C. Schevon
[1989] Optimization by simulated annealing: an experimental evaluation; Part I, graph partitioning, *Oper. Res.* **37**, 865–892.
[1991] Optimization by simulated annealing: an experimental evaluation; Part II, graph coloring and number partitioning, *Oper. Res.* **39**, 378–406.

Jünger, M., G. Reinelt and G. Rinaldi
[1995] The travelling salesman problem, in: *Network Models, Handbooks in Operations Research and Management Science,* Vol. 7, eds. M.O. Ball, T.L. Magnanti, C.L. Monma and G.L. Nemhauser (North-Holland, Amsterdam).

Karger, D., R. Motwani and M. Sudan
[1994] Approximate graph coloring by semidefinite programming, in: *35th Annu. Symp. on Foundations of Computer Science* (IEEE Computer Society Press, New York) to appear.

Karmarkar, N., and R.M. Karp
[1982] An efficient approximation scheme for the one-dimensional bin-packing problem, in: *23rd Annu. Symp. on Foundations of Computer Science* (IEEE Computer Society Press, New York) pp. 312–320.

Kirckpatrick, S., C.D. Gelatt and M.P. Vecchi
[1983] Optimization by simulated annealing, *Science* **230**, 671–680.

Komlós, J., J. Pach and G. Woeginger
[1992] Almost tight bounds on epsilon-nets, *Discrete Comput. Geom.* **7**, 163–173.

Korte, B., and D. Hausmann

[1978] An analysis for the greedy algorithm for independence systems, *Ann. Discrete Math.* **2**, 65–74.

Korte, B., L. Lovász and R. Schrader

[1991] *Greedoids* (Springer, Heidelberg).

Lawler, E.L., J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys

[1985] eds., *The Travelling Salesman Problem: A Guided Tour through Combinatorial Optimization* (Wiley, Chichester).

Lenstra, J.K., D.B. Shmoys and É. Tardos

[1990] Approximation algorithms for scheduling unrelated parallel machines, *Math. Programming A* **46**, 259–271.

Lin, S., and B.W. Kernighan

[1973] An effective heuristic algorithm for the traveling salesman problem, *Oper. Res.* **21**, 498–516.

Linial, N., L. Lovász and A. Wigderson

[1988] Rubber bands, convex embeddings, and graph connectivity, *Combinatorica* **8**, 91–102.

Lovász, L.

[1975] On the ratio of optimal fractional and integral covers, *Discrete Math.* **13**, 383–390.

[1979] Determinants, matchings, and random algorithms, in: *Fundamentals of Computation Theory, FCT'79*, ed. L. Budach (AkademieVerlag, Berlin) pp. 565–574.

Lovász, L., and M.D. Plummer

[1986] *Matching Theory, Ann. Discrete Math.* **29**.

Lovász, L., and A. Schrijver

[1990] Cones of matrices and setfunctions, and 0–1 optimization, *SIAM J. Optim.* **1**, 166–190.

Lovász, L., and M. Simonovits

[1992] On the randomized complexity of volume and diameter, *Proc. 33rd IEEE Annu. Symp. on Foundations of Computer Science* (IEEE Computer Society Press, New York) pp. 482–491.

Lovász, L., M. Saks and A. Schrijver

[1989] Orthogonal representations and connectivity of graphs, *Linear Algebra Appl.* **114/115**, 439–454.

Metropolis, N., A. Rosenblut, M. Rosenbluth, A. Teller and E. Teller

[1953] Equation of state calculation by fast computing machines, *J. Chem. Phys.* **21**, 1087–1092.

Mohar, B., and S. Poljak

[1990] Eigenvalues and the max-cut problem, *Czech. Math. J.* **40**, 343–352.

Monge, G.

[1781] *Déblai et remblai*, Memoires de l'Academie des Sciences.

Mulmuley, K., U.V. Vazirani and V.V. Vazirani

[1987] Matching is as easy as matrix inversion, *Combinatorica* **7**, 105–113.

Nemhauser, G.L., and L.E. Trotter Jr

[1974] Properties of vertex packing and independence system polyhedra, *Math. Programming* **6**, 48–61.

Nemhauser, G.L., and L.A. Wolsey

[1988] *Integer and Combinatorial Optimization* (Wiley, Chichester).

Nemhauser, G.L., A.H.G. Rinnooy Kan and M.J. Todd

[1989] *Optimization, Handbooks in Operations Research and Management Science*, Vol. 1 (North-Holland, Amsterdam).

Padberg, M.W., and M. Grötschel

[1985] Polyhedral computations, in: *The Travelling Salesman Problem: A Guided Tour through Combinatorial Optimization*, eds. E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (Wiley, Chichester) pp. 307–360.

Padberg, M.W., and G. Rinaldi

[1991] A branch-and-cut algorithm for the solution of large-scale traveling salesman problems, *SIAM Rev.* **33**, 1–41.

Poljak, S.

[1994] Integer linear programs and local search for max-cut, *SIAM J. Comput.*, to appear.

Raghavan, P., and C.D. Thompson
[1987] Randomized rounding: a technique for provably good algorithms and algorithmic proofs, *Combinatorica* **7**, 365–374.

Reinelt, G.
[1994] *The Traveling Salesman: Computational Solutions for TSP Applications, Lecture Notes in Computer Science,* Vol. 840 (Springer, Heidelberg).

Rosenkrantz, D.J., R.E. Stearns and P.M. Lewis II
[1977] An analysis of several heuristics for the traveling salesman problem, *SIAM J. Comput.* **6**, 563–581.

Sasaki, G.
[1991] The effect of the density of states on the Metropolis algorithm, *Inform. Process. Lett.* **37**, 159–163.

Sasaki, G.H., and B. Hajek
[1988] The time complexity of maximum matching by simulated annealing, *J. Assoc. Comput. Mach.* **35**, 387–403.

Schäffer, A., and M. Yannakakis
[1991] Simple local search problems that are hard to solve, *SIAM J. Comput.* **20**, 56–87.

Schrijver, A.
[1986] *Theory of Integer and Linear Programming* (Wiley, Chichester).

Schwartz, J.T.
[1980] Fast probabilistic algorithms for verification of polynomial identities, *J. Assoc. Comput. Mach.* **27**, 701–717.

Sherali, H.D., and W.P. Adams
[1990] A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems, *SIAM J. Discrete Math.* **3**, 411–430.

Shmoys, D.B., and É. Tardos
[1993] An approximation algorithm for the generalized assignment problem, *Math. Programming A* **62**, 461–474.

Sinclair, A.J., and M.R. Jerrum
[1988] Conductance and the rapid mixing property for Markov chains: The approximation of the permanent resolved, in: *Proc. 20th ACM STOC* (ACM, New York) pp. 235–244.

Stein, S.K.
[1974] Two combinatorial covering theorems, *J. Combin. Theory A* **16**, 391–397.

Tutte, W.T.
[1947] The factorisation of linear graphs, *J. London Math. Soc.* **22**, 107–111.
[1963] How to draw a graph, *Proc. London Math. Soc.* **13**, 743–768.

van Vliet, A.
[1992] An improved lower bound for on-line bin-packing algorithms, *Inform. Process. Lett.* **43**, 277–284.

Vapnik, V.N.
[1982] *Estimation of Dependences Based on Empirical Data* (Springer, New York).

Wolkowitz, H.
[1981] Some applications of optimization in matrix theory, *Linear Algebra Appl.* **40**, 101–118.