ANGEWANDTE MATHEMATIK UND INFORMATIK
UNIVERSITÄT ZU KÖLN

Report No. 94.156

Practical Problem Solving with Cutting Plane Algorithms in Combinatorial Optimization

by

Michael Jünger
Gerhard Reinelt
Stefan Thienel

March 1994

Institut für Informatik
UNIVERSITÄT ZU KÖLN
Pohligstraße 1
D-50969 Köln

Addresses of the authors:

Michael Jünger
Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-50969 Köln
Germany
E-mail: mjuenger@informatik.uni-koeln.de
Telephone: 49 221 4705313

Gerhard Reinelt
Institut für Angewandte Mathematik
Universität Heidelberg
Im Neuenheimer Feld 294
D-69120 Heidelberg
Germany
E-mail: reinelt@titan.iwr.uni-heidelberg.de
Telephone: 49 6221 563171

Stefan Thienel
Institut für Informatik
Universität zu Köln
Pohligstraße 1
D-50969 Köln
Germany
E-mail: thienel@informatik.uni-koeln.de
Telephone: 49 221 4705307

Abstract

Cutting plane algorithms have turned out to be practically successful computational tools in combinatorial optimization, in particular, when they are embedded in a branch and bound framework. Implementations of such "branch and cut" algorithms are rather complicated in comparison to many purely combinatorial algorithms. The purpose of this article is to give an introduction to cutting plane algorithms from an implementor's point of view. Special emphasis is given to control and data structures used in practically successful implementations of branch and cut algorithms. We also address the issue of parallelization. Finally, we point out that in important applications branch and cut algorithms are not only able to produce optimal solutions but also approximations to the optimum with certified good quality in moderate computation times. We close with an overview of successful practical applications in the literature.

1 Introduction

Combinatorial optimization deals with a special type of mathematical optimization problems with the property that the set of feasible solutions corresponds to a finite set. In its most general form such a problem can be stated as follows.

Combinatorial optimization problem. *Given a finite set $\mathcal{I}$ (set of feasible solutions) and a function $f : \mathcal{I} \to \mathbb{R}$, find an element $I^\star \in \mathcal{I}$ with*

$$f(I^\star) = \max\{f(I) \mid I \in \mathcal{I}\}.$$

$\square$

Such a general optimization problem is of no use unless we have a reasonable characterization of the set of feasible solutions $\mathcal{I}$ and an algorithmic way for evaluating the objective function value for each $I \in \mathcal{I}$.

A characterization of feasible solutions is usually easy, in many cases they correspond to subgraphs of a given graph satisfying some structural property. Concerning evaluation of objective functions, there is a simple concept that is, despite its simplicity, applicable to many hard practical problems. Namely, the objective function is basically given by weight coefficients for the elements of the ground set leading to a linear objective function in the following sense.

Linear combinatorial optimization problem. *Given a finite set E, a set $\mathcal{I} \subseteq 2^E$ of subsets of E (feasible solutions) and a function $c : E \to \mathbb{R}$. For each set $F \subseteq E$ let $c(F) := \sum_{e \in F} c(e)$. Find a set $I^\star \in \mathcal{I}$ with*

$$c(I^\star) = \max\{c(I) \mid I \in \mathcal{I}\}.$$

We call a linear combinatorial optimization problem by $(E, \mathcal{I}, c)$. $\square$

A linear combinatorial minimization problem can be transformed to a maximization problem by multiplying the objective function with -1.

Throughout this paper we only treat problems with a linear objective function. Therefore we drop the term “linear”, and and use combinatorial optimization problem instead of linear combinatorial optimization problem.

Since the set of feasible solutions $\mathcal{I}$ is finite, a combinatorial optimization problem could in principle be solved by enumeration. However, the number of feasible solutions can be very large, even exponential in the size of the set E how the following example shows. A tour in a graph $G = (V, E)$ is a set of edges forming a circuit which visits every node exactly once. If the set of feasible solutions is given by all tours of a complete graph on n nodes, then we have $|E| = \frac{n(n-1)}{2}$ and $|\mathcal{I}| = \frac{(n-1)!}{2}$. Hence the enumerative method is in general impractical.

Combinatorial optimization problems are related to two optimization models: linear programming and (mixed) integer linear programming. Linear programming is one of the basic models in mathematical optimization. It is concerned with the following problem.

Linear optimization problem (LP). *Given a matrix $A \in \mathbb{R}^{(m,n)}$, and vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, find a vector $x^\star \in \mathbb{R}^n$, with*

$$c^T x^\star = \max\{c^T x \mid Ax \leq b\}.$$

The function $c^T x : \mathbb{R}^n \to \mathbb{R}$ is called the objective function. The inequalities in the system $Ax \leq b$ are called constraints. □

A linear optimization problem is also called a linear **programming problem** or a **linear program** (LP). A linear minimization problem can be transformed to a linear maximization problem by multiplying the objective function by -1. An equality constraint $a^T x = b_i$ can be expressed by the two inequalities $a^T x \leq b_i$ and $-a^T x \leq -b_i$. An inequality of the form $a^T x \geq b$ is equivalent to the inequality $-a^T x \leq -b_i$. Upper or lower bounds on variables can also be interpreted as inequalities. Therefore the definition given above can be used without loss of generality.

Linear programming problems are very well studied and can by now be solved routinely even on very large scale. Whereas several years ago only the simplex algorithm (see e.g. Chvatal (1983)) was able to solve large problems, we have now also interior point methods that can compete (for a state of the art survey see Lustig, Marsten and Shanno (1992b)).

Unfortunately, only a minority of practical problems can be modelled as pure linear programming problems. In many cases, some or all of the variables have to take integer values. By adding integrality conditions to a subset of the variables we get a linear mixed integer optimization problem.

Linear mixed integer optimization problem. *Given a matrix $A \in \mathbb{R}^{(m,n)}$, vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and a subset $I \subseteq \{1, \ldots, n\}$, find a vector $x^\star \in \mathbb{R}^n$ with*

$$c^T x^\star = \max\{c^T x \mid Ax \leq b,\ x_i \text{ integer for all } i \in I\}.$$

□

If all variables are required to be integer, i.e., $I = \{1, \ldots, n\}$, the linear mixed integer optimization problem becomes an **integer linear optimization problem**. If the variables of the set I have to be 0 or 1, the optimization problem is called **(mixed) zero-one optimization problem**.

There has been significant progress in the development of codes for solving (mixed) integer programming problems. But still, problems with several hundreds of variables and constraints can be difficult to solve depending on the specific problem structure.

Connections between combinatorial optimization and linear or zero-one linear optimization can be established as follows. For the finite ground set E let $\mathbb{R}^E$ be the $\mathbb{R}$-vectorspace indexed by the elements of E.

Incidence Vector. *Given a finite set E, and a set $F \subseteq E$, the incidence vector $\chi^F \in \mathbb{R}^E$ is defined by*

$$\chi_e^F = \begin{cases} 1, & \text{if } e \in F \\ 0, & \text{if } e \notin F \end{cases}$$

□

With a combinatorial optimization problem $(E, \mathcal{I}, c)$ we associate the polytope

$$P_{\mathcal{I}} = \operatorname{conv}\{\chi^I \mid I \in \mathcal{I}\},$$

i.e., the convex hull of the incidence vectors of feasible sets.

Because the incidence vectors are 0-1-vectors, they are exactly the vertices of the polytope $P_{\mathcal{I}}$. If we associate with the function $c : E \to \mathbb{R}$ of a combinatorial optimization problem a vector $c \in \mathbb{R}^E$, we can solve the combinatorial optimization problem by solving the optimization problem $\max\{c^T x \mid x \in P_{\mathcal{I}}\}$. Unfortunately, we do not know any efficient algorithm to solve an optimization problem, when the solution space is only defined as the convex hull of a set of points. However, according to classical results of Farkas, Weyl and Minkowski (see Schrijver (1986)) there exists a finite set of inequalities $Ax \leq b$, such that $P_{\mathcal{I}} = \{x \mid Ax \leq b\}$ (recall that an equation can be substituted by two inequalities). Hence we could in principle transform the combinatorial optimization problem $(E, \mathcal{I}, c)$ to the linear program $\max\{c^T x \mid Ax \leq b\}$.

In fact, there are finite algorithms to transform one representation of the polytope $P_{\mathcal{I}}$ into the other that can be used for very small problem instances (cf. Christof, Jünger and Reinelt (1991), Euler and Le Verge (1992) and Reinelt (1993) for examples). But, for combinatorial optimization problems the number of inequalities in $Ax \leq b$ is usually simply too large to be represented explicitly. It will turn out, however, that this difficulty can be dealt with at least to some extent.

The system of inequalities $Ax \leq b$ with $P_{\mathcal{I}} = \{x \mid Ax \leq b\}$ is also called the **linear description** of a combinatorial optimization problem. Unfortunately, for most combinatorial optimization problems only a very small part of the linear description is known. Moreover, for no $\mathcal{NP}$-hard combinatorial optimization problem a complete linear description could be given so far and Karp and Papadimitriou (1982) showed that it cannot be found unless $\mathcal{NP} = \text{co-}\mathcal{NP}$ (most computer scientists however assume that $\mathcal{NP} \neq \text{co-}\mathcal{NP}$). Nevertheless, we will show that even a partial linear description provides us with a rather powerful tool for the solution of hard combinatorial optimization problems.

Easier than the description of the polytope $P_{\mathcal{I}}$ of a combinatorial optimization problem $(E, \mathcal{I}, c)$ by linear inequalities is the formulation of an equivalent integer optimization problem, which is called **integer programming formulation** of the combinatorial optimization problem. Since here the variables are required to be integral, respectively binary, we can expect that a smaller number of constraints is sufficient for the description of the problem. Usually a comparatively small subset of the constraint system $Ax \leq b$ with

$P_{\mathcal{I}} = \{x \mid Ax \leq b\}$ gives an integer programming formulation. However, the number of required inequalities can still be exponential in the size of $|E|$.

We give some examples of combinatorial optimization problems and their respective integer programming formulations. The first three problems are defined on directed or undirected graphs and the set of feasible solutions consists of subgraphs with a special property. An integer optimization problem can be formulated by associating a variable x_e with every edge (arc) of the given graph (digraph) with the interpretation that $x_e = 1$ if the edge (arc) is in the subgraph and $x_e = 0$ otherwise. For a node set W of a graph we denote by $\delta(W)$ the set of edges with exactly one endnode in W (if $W = \{v\}$ we write $\delta(v)$). Edge sets induced by a node set W in this way are called **cuts**. For $W \subseteq V$ we denote by $E(W)$ the set of all edges in E with both endnodes in W. For an edge (arc) set F we denote by $x(F)$ the sum of the variables associated with the edges (arcs) in F.

The symmetric traveling salesman problem. Given the complete graph $K_n = (V_n, E_n)$ with edge weights c_e for every $e \in E_n$, the symmetric traveling salesman problem is to find a tour with minimum length, i.e., with minimum sum of its edge weights.

An integer programming formulation of the symmetric traveling salesman problem is given by

$$
\begin{array}{lll}
\min & c^T x & \\
\text{s.t.} & x(\delta(v)) = 2 & \text{for all } v \in V \qquad (1)\\
& x(\delta(W)) \geq 2 & \text{for all } \emptyset \neq W \subset V \qquad (2)\\
& 0 \leq x \leq 1 & \\
& x \text{ integer} &
\end{array}
$$

The equations (1) require that each node is incident to exactly two edges and are called **degree constraints**. The inequalities (2) forbid subtours, and are therefore called **subtour elimination constraints**. □

The max-cut problem. Given a graph $G = (V, E)$ with edge weights c_e for every $e \in E$, the max-cut problem is to find a subset $S \subseteq V$, such that the sum of the weights of the edges of the cut $\delta(S)$ induced by S is maximum.

The max-cut problem with nonpositive edgeweights can be solved in polynomial time by an an algorithms of Gomory and Hu (1961), the general problem is $\mathcal{NP}$-hard.

Since every cut and every cycle intersect in an even number of edges, we get the following integer programming formulation of the max-cut problem.

$$
\begin{array}{lll}
\max & c^T x & \\
\text{s.t.} & x(F) - x(C \setminus F) \leq |F| - 1 & \text{for all cycles } C \subseteq E \text{ and all } F \subseteq C,\ |F| \text{ odd} \qquad (3)\\
& 0 \leq x \leq 1 & \\
& x \text{ integer} &
\end{array}
$$

The inequalities (3) are the **odd cycle constraints**. □

The linear ordering problem. Given the complete directed graph $D_n = (V_n, A_n)$ with arc weights c_a for every $a \in A_n$, find a subset of A_n of maximum weight which contains no directed cycle and for every pair of nodes exactly one arc connecting these nodes. An arc set satisfying these conditions is called acyclic tournament and it is easy to see that it induces a linear ordering of the nodes of V_n.

The integer programming formulation is based on the fact that a subset of arcs with exactly one arc connecting each pair of nodes contains no cycle if it contains no cycle consisting of three edges. The integer optimization problem is the following.

$$
\begin{aligned}
\max \quad & c^T x & & \\
\text{s.t.} \quad & x(C) \le 2 & & \text{for all cycles } C,\ |C| = 3 \qquad (4)\\
& x_{ij} + x_{ji} = 1 & & \text{for all } 1 \le i < j \le n \qquad (5)\\
& 0 \le x \le 1 & & \\
& x \text{ integer} & &
\end{aligned}
$$

Whereas for the symmetric traveling salesman problem it has turned out that it is convenient to keep all the equations in the problem formulation, the simple structure of the integer programming formulation of the linear ordering problem is used to project out all x_{ij} with $i > j$ via the equation $x_{ij} = 1 - x_{ji}$. □

Note a subtle difference in these three problems. In the traveling salesman problem each feasible subgraph contains n edges, in the linear ordering problem each feasible set is given by $\frac{n(n-1)}{2}$ arcs, and cuts may consist of 0 up to $\lfloor \frac{n}{2} \rfloor \cdot \lceil \frac{n}{2} \rceil$ edges. This difference is of importance for the use of sparse graph techniques to be discussed in section 4.5.

To avoid the impression that there are only combinatorial optimization problems defined on graphs we give another important example.

The knapsack problem. Given n objects with weights $a_i \in \mathbb{R}$ and and values $c_i \in \mathbb{R}$ for each object $i = 1, \ldots, n$. The knapsack problem is to select a subset S of the objects that the sum of the weights of all objects in S is less than a given upper bound $b \in \mathbb{R}$ and the sum of the values of all objects in S is maximum. An integer programming formulation of this problem is the following.

$$
\begin{aligned}
\max \quad & c^T x \\
\text{s.t.} \quad & a^T x \le b \qquad (6)\\
& 0 \le x \le 1 \\
& x \text{ integer}
\end{aligned}
$$

□

All four examples are $\mathcal{NP}$-hard combinatorial optimization problems.

2 Cutting planes

Suppose we have to solve a linear optimization problem whose set of constraints is too large to be represented explicitly on a computer or too large to be handled by the LP-solver. In such a case we can still attempt to solve the problem with the following approach. We start with a small subset of the constraints and compute an optimum solution subject to these constraints. We now check if any of the constraints not in the current linear program is not satisfied. If such constraints are present, we add one or more of them to the current LP and resolve it. If no constraint is violated, then the current optimum solution also solves the original problem. This is the basic principle of the so-called **cutting plane approach**, whose name originates from the fact that the constraints added to the current LP "cut off" the current solution because it is infeasible for the original problem.

Note the important fact that the approach does not require that an explicit list of the constraints defining the original problem has to be present. Required is "only" a method for identifying inequalities that are valid for the original problem but violated by the current solution. In subsection 4.3 we will give an example for such an algorithm.

The essential tool for the solution of combinatorial optimization problems with cutting plane algorithms are valid inequalities.

Valid inequality of a combinatorial optimization problem. *Given an integer programming formulation* $\max\{c^T x \mid Ax \leq b, x \text{ integer}\}$ *of a combinatorial optimization problem, an inequality* $f^T x \leq f_0$ *is called valid, if* $f\overline{x} \leq f_0$ *for all feasible solutions* $\overline{x} \in \{x \mid Ax \leq b,\ x \text{ integer}\}$. □

If we know a class of valid inequalities, we have to be able to check if a constraint of this class is violated by the current solution, i.e., we must solve the following problem.

The general separation problem. *Given a class of valid inequalities for a combinatorial optimization problem, and a vector* $y \in \mathbb{R}^n$*, either prove that* y *satisfies all inequalities of this class, or find an inequality of this class which is violated by* y. □

An algorithm which solves the general separation problem is called **exact separation algorithm**. Unfortunately, exact algorithms are often not known for classes of valid inequalities or it can even be shown that the separation problem for a certain class of inequalities is $\mathcal{NP}$-hard. In this case we have to resort to a **heuristic separation algorithm**, which may find violated inequalities, but if it fails, it is not guaranteed that no constraint of the class is violated.

We state now a generic cutting plane algorithm for solving a combinatorial optimization problem. Let $\max\{c^T x \mid Ax \leq b,\ x \text{ integer}\}$ be an integer programming formulation.

Cutting Plane Algorithm

(1) Set $A' = \binom{I}{-I}$ and $b' = \binom{1}{0}$
where 1 and 0 are appropriate vectors of all 1's and 0's.

(2) Solve the LP $c^T\overline{x} = \max\{c^Tx \mid A'x \leq b', x \in \mathbb{R}^n\}$.

(3) If $\overline{x}$ corresponds to a feasible solution of the combinatorial optimization problem then output $\overline{x}$ and stop.

(4) Generate a cutting plane (f, f_0), $f \in \mathbb{R}^n$, $f_0 \in \mathbb{R}$ with

$f^T\overline{x} > f_0$

and

$f^Ty \leq f_0$ for all $y \in \{x \mid Ax \leq b, x \text{ integer}\}$.

(5) Add the inequality $f^Tx \leq f_0$ to the constraint system (A', b').

(6) Go to (2).

The above formulation of the algorithm takes into account that already the number of inequalities in the integer programming formulation can be too large for explicit representation. In such a case one can start with the trivial constraint system $0 \leq x \leq 1$. The inequalities $Ax \leq b$ are valid inequalities which can serve as cutting planes. However, an integral solution coming up in the course of the algorithm is only the incidence vector of a feasible solution of the combinatorial optimization problem if no inequality of the system $Ax \leq b$ is violated.

This algorithm is only correct under the assumption that all LPs in step (2) can be solved, that in step (4) always a cutting plane can be generated, and that the algorithm terminates after a finite number of iterations.

The requirement that a cutting plane can be generated as long as the LP-solution is not an incidence vector of a feasible solution of the combinatorial optimization problem is very strong. Also if a cutting plane algorithm does not find a cutting plane in any case, it may be of some use. Namely, if no more cuts can be generated, yet the solution is not a feasible incidence vector, the final objective function value gives an upper bound for the optimal solution. This value can be used to assess the quality of a known feasible solution, which e.g., results from a heuristic. In the case that no cutting plane can be generated in step (4), we have to resort to branch and bound (see section 3) to solve the problem to optimality.

The outlined cutting plane algorithm is rather rudimentary. We omit further refinements here, like elimination of constraints, sparse graph techniques, fixing of variables, etc., because they will be covered in the detailed outline of a branch and cut algorithm in section 4.

It is not clear at this point that such an approach is useful at all. Yet, this has been verified by many practical computations that are listed in section 7.

We now address the question how further cutting planes can be found if the optimum solution of the LP $\max\{c^Tx \mid A'x \leq b'\}$ is not integral, but no inequality of the system $Ax \leq b$ is violated.

First of all we can use classes of cutting planes that can be applied to any integer or mixed integer optimization problem. We call such cutting planes **general purpose**

cutting planes, since they are not problem specific and can be employed for the solution of every integer optimization problem.

The first cutting plane algorithms for integer and mixed integer programs were introduced by GOMORY (1958, 1960, 1963), who also proved that these algorithms terminate with an optimum solution after a finite number of iterations. Unfortunately, it turned out in practical experiments that the **Gomory cutting planes** provide very weak cuts leading to numerical problems, and only rather small integer optimization problems can be solved to optimality with this method.

CROWDER, JOHNSON AND PADBERG (1983) use **minimal cover inequalities** and $(1,k)$-**configurations** for the solution of zero-one linear optimization problems. These cutting planes are derived from facets (will be explained below) of the polytope $P_{\mathcal{I}}$ of the knapsack problem defined by each constraint in the formulation of the zero-one optimization problem. This approach is refined and generalized in VAN ROY AND WOLSEY (1987) and HOFFMAN AND PADBERG (1991).

For mixed zero-one linear optimization problems BALAS, CERIA AND CORNUEJOLS (1993a, 1993b) generate cutting planes by a **lift-and-project method**. The computational results are very promising and show that these cutting planes outperform the Gomory cuts.

Fenchel cuts for integer optimization problems are introduced in BOYD (1993a, 1993b, 1993c).

For solving combinatorial optimization problems, these general cuts seem to be of limited use. Successful computational work relies on cutting planes designed for the particular problem.

A very successful method for the derivation of problem specific cutting planes is the investigation of the polytope $P_{\mathcal{I}}$ associated with a combinatorial optimization problem $(E, \mathcal{I}, c)$. An inequality $d^T x \leq d_0$ with $d \neq 0$ is called a **valid inequality** with respect to a polytope $P \subseteq \mathbb{R}^n$ if $P \subseteq \{x \in \mathbb{R}^n \mid d^T x \leq d_0\}$. If $d^T x \leq d_0$ is a valid inequality and the intersection of the $(n-1)$-dimensional affine subspace $H = \{x \mid d^T x = d_0\}$ with P is neither empty nor equals P, then $F = P \cap H$ is called a **face** of P defined by the valid inequality $d^T x \leq d_0$. Let $s = \dim(P)$ be the dimension of P. The $(s-1)$-dimensional faces, i.e., the faces with maximum dimension, are called **facets** of P. If $F = P \cap \{x \mid d^T x \leq d_0\}$ is a facet the polytope P, the inequality $d^T x \leq d_0$ is called **facet defining inequality** for P.

Every facet of this polytope implies a valid inequality for the corresponding integer programming formulation. Facet defining inequalities of the polytope $P_{\mathcal{I}}$ are not dominated by any other valid inequality and therefore in some sense the best cutting planes. However, it is in general not simple to to find facet defining inequalities (see GRÖTSCHEL AND PADBERG (1985), NEMHAUSER AND WOLSEY (1988) and PULLEYBLANK (1989)). Furthermore, it should be kept in mind that, for algorithmic purposes, facet defining inequalities are only useful if their separation problem can be solved exactly or at least be attacked with heuristics.

Cutting plane algorithms using problem specific cutting planes, e.g. facet defining inequalities, often have to stop without finding an optimum solution. This can have two different reasons. First, for no $\mathcal{NP}$-hard combinatorial optimization problem a complete linear description is known. Second, even if a big class of facets is known, no efficient algorithm may be available for the solution of the exact separation problem of this class. Nevertheless, large instances of $\mathcal{NP}$-hard combinatorial optimization problems can be solved with the help of facet defining cutting planes in combination with a sophisticated enumeration procedure as we will outline in section 4.

Of course, one can also design hybrid cutting plane algorithms which combine general purpose cutting planes and problem specific cutting planes in the following way. When generating cutting planes we first try to separate a problem specific, preferably facet defining inequality. If this fails, we generate a general purpose cutting plane, e.g., a Gomory cut or a lift-and-project cut. Such hybrid algorithms are used by Miliotis (1978), Laporte and Norbert (1980) and Miliotis, Laporte and Norbert (1980).

3 Solving problems to optimality

The cutting plane approach outlined so far does not necessarily solve a problem instance to optimality for various reasons discussed above. We may get stuck at a solution which is not the incidence vector of a feasible solution of the combinatorial optimization problem. At this point we can employ another basic algorithmic technique for solving hard combinatorial optimization problems: **branch and bound** (or implicit enumeration). This technique was designed for the solution of mixed integer optimization problems by Land and Doig (1960) and Dakin (1965) and refined in the following years (see e.g. Gauthier and Ribiere (1977)).

Branch and bound is a divide-and-conquer approach trying to solve the original problem by splitting it into smaller problems for which upper and lower bounds are computed. The crucial part of a successful branch and bound algorithm is the computation of upper bounds for these subproblems. Here one uses the fundamental concept of relaxation.

Relaxation. *Given two combinatorial optimization problems $(E, \mathcal{I}, f)$, $(E', \mathcal{I}', f')$ and an injective function $\varphi : E \to E'$. $(\mathcal{I}', f')$ is a relaxation of $(\mathcal{I}, f)$, if $\varphi(I) \in \mathcal{I}'$ and $f(I) = f'(\varphi(I))$ for all $I \in \mathcal{I}$.* □

Hence a solution of the relaxed problem gives an upper bound on the optimum objective function value of the problem it was derived from. The tighter the relaxation, the better this bound will be. But a relaxation is only useful, if it can be treated efficiently by optimization algorithms.

By dropping the integrality conditions of the variables of an integer programming formulation of a combinatorial optimization problem we get a **linear programming relaxation**, which is basic in the context of cutting plane algorithms. This relaxation can be tightened by adding further valid inequalities.

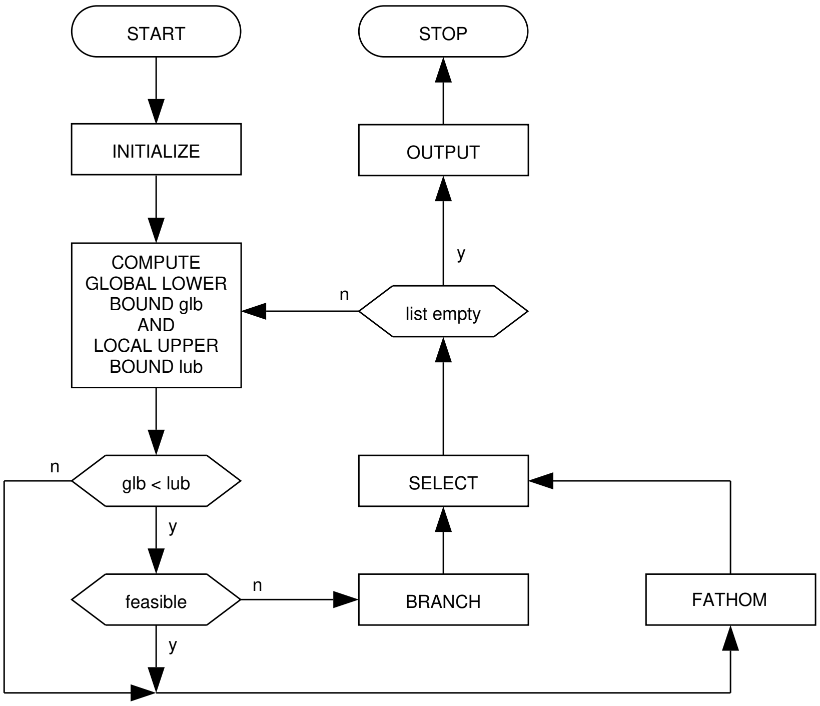

Figure 1. Flowchart of the branch and bound algorithm.

Before we start explaining the flowchart of the branch and bound algorithm of Figure 1, we introduce some terminology concerning upper bounds (derived from solving relaxations) and lower bounds (obtained by finding feasible solutions). We call an upper bound **local**, if it is only valid for a subproblem, and **global**, if it is a bound for the original problem. By solving a relaxation of the current problem, we obtain a **local upper bound** `lub` for the objective function value of the original problem. If the solution found for the relaxation happens to be feasible for the original problem (in which case it is also the optimum solution of the subproblem) and has higher objective function value than any feasible solution found so far, it is memorized and the **global lower bound** `glb` for the objective function value is increased accordingly.

A branch and bound algorithm maintains a list of subproblems of the original problem, which is initialized with the original problem itself. In each major iteration step the algorithm selects a subproblem from this list, computes a local upper bound for this subproblem, and tries to improve the global lower bound. If the local upper bound does not exceed the global lower bound, the current subproblem is fathomed, because its solution cannot be better than the best known feasible solution. Otherwise we check if the optimal solution of the relaxation of the subproblem is a feasible solution of the original problem. In this case we have solved the subproblem and thus, it is fathomed.

If the local upper bound exceeds the global lower bound and no feasible solution was found for the current problem, we perform a branching step by splitting the current subproblem into a collection of new subproblems whose union of feasible solutions contains all feasible solutions of the current subproblem. The simplest **branching rule** consists of defining two new subproblems in one of which a fixed variable is required to have the value 1 in the solution and in the other one to have the value 0 in the solution. Different branching rules are presented in section 4.

If the list of subproblems becomes empty, then the memorized feasible solution whose objective function value is equal to the global upper bound can be output as the optimum solution.

Important for the efficiency of a branch and bound algorithm is not only the quality of the used relaxation technique, but also the quality of the generated feasible solutions, since otherwise, the number of generated subproblems becomes rapidly very large.

Earlier approaches for the solution of hard combinatorial optimization problems combined problem specific cutting planes with a commercial branch and bound software (Crowder and Padberg (1980) and Grötschel and Holland (1991)). A relaxation of the integer programming formulation is attacked with a cutting plane algorithm, which generates preferably facet defining violated constraints. If the cutting plane procedure terminates with a solution which is not the incidence vector of a feasible solution, the final relaxation of the problem is solved to optimality by a commercial branch and bound algorithm, like e.g., IBM's MPSX-MIP (IBM (1979). The integer solution found by branch and bound might not necessarily be a feasible solution of the combinatorial optimization problem, if the final relaxation is not an integer programming formulation of the original problem. In this case violated constraints of the integer programming formulation are added to the relaxation, which is again subjected to the branch and bound algorithm. This process is iterated until the optimum solution of the combinatorial optimization problem is found. The disadvantage of this technique is that no problem specific cutting planes are separated in the subproblems generated by the branch and bound algorithm.

4 Branch and Cut

We will now discuss in detail the algorithmic approach for solving hard combinatorial optimization problems that has become standard in the last years: **branch and cut**. Its main difference from the classical branch and bound method is the use of LP relaxations and the employment of problem specific cutting planes at every node of the enumeration tree. This feature incurs several technicalities that make the design and implementation of branch and cut algorithms a nontrivial task. We will address such technical details and give some ideas for further enhancements that proved to be useful in practice.

The first combination of problem specific valid inequalities and branch and bound methods can be found in Miliotis (1976) for the traveling salesman problem. The use of facet defining cutting planes and enhanced automatic cutting plane generation in combination with branch and bound in the form we will outline in this section was first published in Grötschel, Jünger and Reinelt (1984b) for the linear ordering problem. The term "branch and cut" has been introduced in Padberg and Rinaldi (1987, 1991) for an algorithm for the solution of the traveling salesman problem. Here many important algorithmic details were investigated for the first time.

In our description, we proceed as follows. First we describe the enumerative part of the algorithm, i.e., we discuss in detail how branching and selection operations can be performed. Then we explain the work done in a subproblem of the enumeration. After discussing sparse graph techniques, we give a short survey on the required data structures.

There are two major ingredients of a branch and cut algorithm, the computation of global lower and local upper bounds. The upper bounds are produced by performing an ordinary cutting plane algorithm for each subproblem.

Two basic techniques for the computation of lower bounds (corresponding to feasible solutions of the original problem) are currently being used. The first method is to compute a good lower bound by some heuristics before the root node of the complete branch and cut tree is processed. Later this bound can only be improved, if the solution of a linear program is the incidence vector of a better feasible solution. The other method is the computation of lower bounds in the cutting plane phase by exploiting fractional LP-solutions. This technique may require more running time spent for heuristics, yet may decrease the total running time, since the size of the enumeration tree may be smaller.

A first outline of the branch and cut algorithm is given in the flowchart of Figure 2. Roughly speaking, the two leftmost columns describe the cutting plane phases within a single subproblem, the third column shows the preparation and execution of a branching operation, and in the rightmost column, the fathoming of a subproblem is performed.

4.1 Terminology

We give informal explanations of all steps of the flowchart. But before going into detail, we have to define some terminology.

Since in a branching step like in a branch and bound algorithm two (or more) new subproblems are generated, the set of all subproblems can be represented by a binary (k-nary) tree, which we call **branch and cut tree**. Hence we call a subproblem also **branch and cut node**. Figure 3 shows an example of a branch and cut tree. We distinguish between four types of branch and cut nodes. The node which is currently processed is called the **current branch and cut node**. The other unfathomed leaves of the branch and cut tree still have to be processed and are called the **active nodes**. Finally, there are the already processed **nonactive nodes**. A non-active node can either be **fathomed** or **not fathomed**.

Each variable has one of the following statuses during the computation: `atlower-bound`, `basic`, `atupperbound`, `settolowerbound`, `settoupperbound`, `fixedtolowerbound`, `fixedtoupperbound`. When we say that a variable is **fixed** to zero or one, it means that it is at this value for the rest of the computation. If it is **set** to zero (lower bound) or one (upper bound), this value remains valid only for the current branch and cut node and all branch and cut nodes in the subtree rooted at the current one in the branch and cut tree.

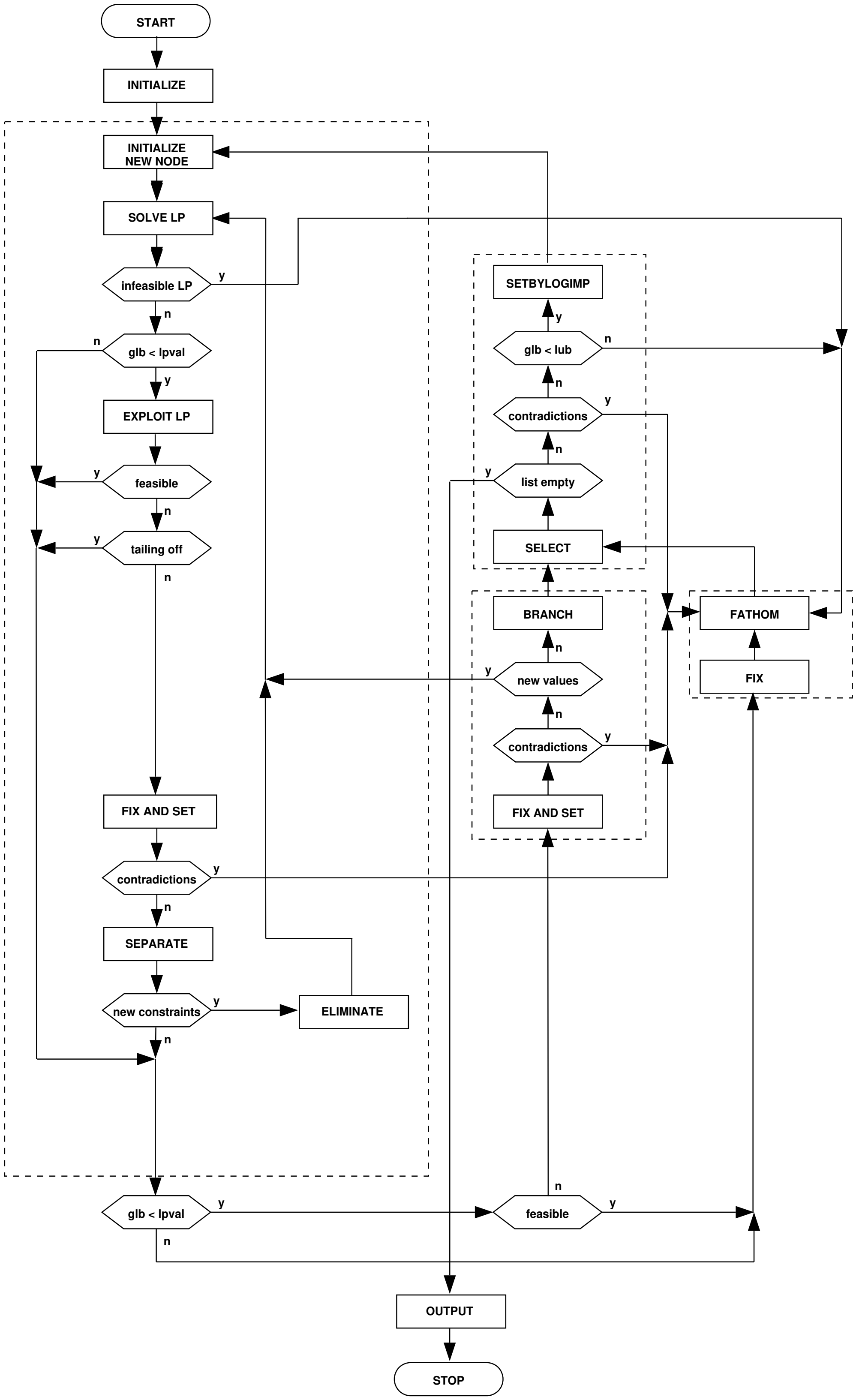

Figure 2. Flowchart of the branch and cut algorithm.

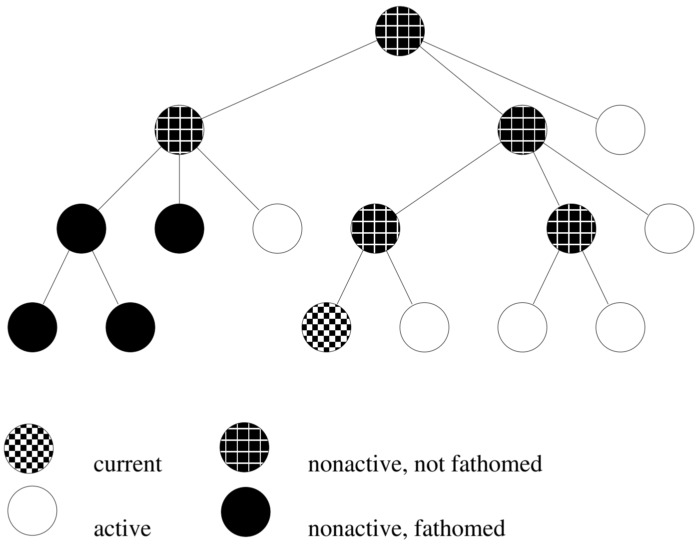

Figure 3. Branch and cut tree

The conditions for fixing and setting variables will be explained later. The meanings of the other statuses are obvious: As soon as an LP has been solved, each variable which has not been fixed or set receives one of the statuses `atlowerbound`, `basic` or `atupperbound` by the revised simplex method with lower and upper bounds.

Finally, the variable `lpval` always denotes the optimal value of the last LP that has been solved, which is also a local upper bound `lub` for the currently processed node, the global variable `glb` (global lower bound) gives the value of the currently best known feasible solution. The maximum upper bound of all active branch and cut nodes and the current branch and cut node is the global upper bound `gub` for the whole problem. The subtree rooted at the highest common ancestor of all active and the current branch and cut nodes is called the **remaining branch and cut tree**. Therefore, we call this highest common ancestor also the **root of the remaining branch and cut tree** and the local upper bound of this node is called `rootub`. The difference between `gub` and `rootub` will be discussed below.

Like in branch and bound terminology we call a subproblem **fathomed**, if the local upper bound `lpval` of this subproblem is less than or equal to the global lower bound `glb`, or if the subproblem becomes infeasible (e.g., if branching variables have been set in a way that the subproblem does not contain any feasible solution), or if the subproblem is solved, i.e., the solution of the LP-relaxation is a feasible solution of the original problem.

The branch and cut algorithm consists of three different parts: The enumerative frame, the computation of upper bounds and the computation of lower bounds. It is easy to identify the boxes of the flowchart of Figure 1 with the dashed boxes of the flowchart of Figure 2.

The central part is the upper bounding part which is performed after the selection of a new current subproblem. It consists of trying to solve the current problem by optimizing

over LP relaxations that are tightened by adding cutting planes. This bounding part is left,
– if the local upper bound is less than or equal to the global lower bound,
– if the LP solution is the incidence vector of a feasible solution,
– if no more cutting planes can be generated,
– if infeasibility of the current subproblem is detected,
– if the upper bound does not decrease significantly, although cutting planes are added (`tailing off`).

It is advantageous, although not necessary for the correctness of the algorithm, to reenter the bounding part if variables are fixed or set to new values by FIX AND SET, instead of creating new subproblems in BRANCH.

4.2 Enumerative frame

The enumerative frame consists of all parts of the branch and cut algorithm except the bounding part (the dashed box in the leftmost column of Figure 2).

INITIALIZE

After the input of problem, the set of active branch and cut nodes is initialized as the empty set. To initialize the global lower bound `glb`, feasible solutions are computed by some heuristic methods. We explain the details in subsection 4.4. Afterwards the root node of the complete branch and cut tree is processed by the bounding part.

BOUNDING

The computation of the lower and upper bounds is outlined in the subsections 4.3 and 4.4.

We continue the explanation of the enumerative frame at the ordinary exit of the bounding part (at the end of the first column of the dashed bounding box). If the current branch and cut node cannot contain a feasible solution which is better than the best known one ($\texttt{lpval} \leq \texttt{glb}$), or the final LP-solution is the incidence vector of a feasible solution (`feasible`), the node is fathomed. Otherwise a branching operation and the selection of another branch and cut node for further processing (third column of the flowchart) is prepared.

FIX AND SET

The routine FIX AND SET of Figure 2 consists of the four procedures FIXBYREDCOST, FIXBYLOGIMP, SETBYREDCOST and SETBYLOGIMP. If a branching operation is prepared, and the current branch and cut node is the root node of the branch and cut tree, the reduced cost of the nonbasic variables obtained from the LP-solver can be used to fix them forever at their current values by the routine FIXBYREDCOST. Namely, if the variable x_e is nonbasic and the reduced cost is r_e, x_e can be fixed to zero if $x_e = 0$ and $\texttt{rootub} + r_e < \texttt{glb}$ and x_e can be fixed to one if $x_e = 1$ and $\texttt{rootub} - r_e < \texttt{glb}$.

During the computational process, the value of `glb` increases, so that at some later point in the computation, one of these criteria can be satisfied, even though it is not satisfied at the current point of the computation. Therefore, after processing the root

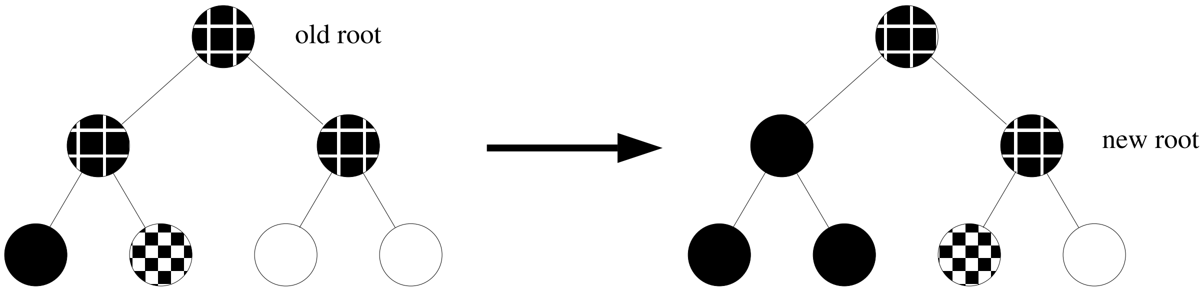

Figure 4. Root change of remaining branch and cut tree.

node of the branch and cut tree, a **list of candidates for fixing** of all nonbasic variables is made along with their values (0 or 1) and their reduced costs, and `rootub` is initialized.

This list of candidates for fixing and the value of `rootub` can be updated each time when we get a new root of the remaining branch and cut tree. We get a new root of the remaining branch and cut tree, if all nodes in all subtrees except one subtree of the old root are fathomed (see Figure 4). Since storing these lists in every node, which might eventually become the root node of the remaining active nodes in the branch and cut tree, would use too much memory space, the complete bounding part is processed a second time for the node, when it becomes the new root. If the constraint system for the recomputation could be initialized by those constraints, which were present in the last LP of the first processing of this node, only a single call of the simplex algorithm would be necessary. However, the storage of all these constraints would require again too much memory. The constraint system can be initialized for instance with the constraints of the last solved LP. If some constraints are separated heuristically, it is not guaranteed that the same local upper bound can be achieved as in the previous bounding phase. Therefore not only the reduced costs and statuses of the variables of this recomputation have to be used, but also the corresponding local upper bound as `rootub` in the subsequent calls of the routine FIXBYREDCOST. This explains why we distinguish between `gub` and `rootub`. If the basis is initialized by the variables contained in the best known feasible solution and the primal simplex algorithm is called, phase 1 of the simplex method can be avoided. Of course this recomputation is not necessary if the root of the remaining branch and cut tree is currently processed, e.g., the first processed node. The list of candidates for fixing should be checked by the routine FIXBYREDCOST whenever it has been freshly compiled or if the value of the global lower bound `glb` has improved since the last call of FIXBYREDCOST.

FIXBYREDCOST may find that a variable can be fixed to a value opposite to the one it has been set to (`contradiction`). This means that earlier in the computation, somewhere on the path of the current branch and cut node to the root of the branch and cut tree, an unfavorable decision has been made which led to this setting either directly in a branching operation or indirectly via SETBYREDCOST or SETBYLOGIMP (to be discussed below). Hence the current branch and cut node can be fathomed immediately.

After variables have been fixed by FIXBYREDCOST, FIXBYLOGIMP should be called. This routine tries to fix more variables by logical implications by exploiting the structure of a special combinatorial optimization problem. For instance, in the case of the symmetric traveling salesman problem, if two variables corresponding to edges incident to a node v are `fixedtoupperbound`, all other edges (variables) incident to the node v can be `fixedtolowerbound`. These fixings of variables by logical implications have no direct influence on the following optimization process, yet in this case they can decrease the size of the LP, as we will explain in INITIALIZE NEW NODE. Fixing by logical implications becomes more important for the max-cut problem. E.g., if two edges (u, w) and (v, w) are `fixedtoupperbound`, it follows that u and v must be on the same shore of the cut, i.e., the edge (variable) (u, v) can be `fixedtolowerbound`.

While fixings of variables are globally valid for the whole computation, variable settings are only valid for the current branch and cut node and all branch and cut nodes in the subtree rooted at the current branch and cut node. SETBYREDCOST sets variables by the same criteria as FIXBYREDCOST, but based on the local reduced cost and the local upper bound `lub` of the current subproblem rather than "globally valid reduced cost" and the upper bound of the root node `rootub`. Contradictions are possible if later the variable is fixed to the opposite value. In this case the current branch and cut node is fathomed.

SETBYLOGIMP is be called whenever SETBYREDCOST has successfully fixed variables, as well as after a SELECT operation, where a set branching variable might cause logical implications.. It tries to set more variables by logical implication similar to FIXBYLOGIMP.

Before starting a branching operation, some variables may have been fixed or set to new values (0 or 1), i.e., values different as in the current LP-solution. In this case it is advantageous to solve the new LP rather than performing the branching operation.

Variables can also be fixed and set during the bounding phase after the solution of an LP. Here it is sufficient to call the according procedures when a better global lower bound `glb` has been found, or when the objective function value of the LP has decreased significantly.

BRANCH

Some variable is chosen as the branching variable and two new branch and cut nodes, which are the two sons of the current branch and cut node, are created and added to the set of active branch and cut nodes. In the first son the branching variable is set to 1 in the second one to 0. If no constraints of the integer programming formulation is violated, then there is at least one fractional variable which is a reasonable candidate for a branching variable. However if a constraint of the integer programming formulation is violated, it is possible that all variables have an integral LP-value, yet the LP-solution is not a feasible solution of the original problem. In this case a variable with an integral LP-value has to be chosen as branching variable.

There is a variety of different strategies for the selection of the branching variable, so that we can present here only some of them. Let $\overline{x}$ be the solution of the last solved LP.

(1) Select a variable with value close to 0.5 which has a big objective function coefficient in the following way. Find L and H with $L = \min\{0.5, \max\{\overline{x}_e \mid \overline{x}_e \leq 0.5, e \in E\}\}$ and $H = \max\{0.5, \min\{\overline{x}_e \mid \overline{x}_e \geq 0.5, e \in E\}\}$. Let $C = \{e \in E \mid 0.75L \leq \overline{x}_e \leq H + 0.25(1-H)\}$ be the set of variables with value "close" to 0.5. From the set C the variable with maximum cost is selected, i.e., with maximum objective function coefficient. This method is similar to the one given in Padberg and Rinaldi (1991).

(2) Select the variable which has an LP-value closest to 0.5.

(3) Select the fractional variable (if available) which has maximum objective function coefficient.

(4) If there are fractional variables which are equal to 1 in the currently best known feasible solution, select the one with maximum cost of them, otherwise, apply strategy (1).

(5) Select a fractional variable (if available) which is closest to one, i.e., find a variable $e^\star$ with $\overline{x}_{e^\star} = \max\{\overline{x}_e \mid \overline{x}_e \leq 0.999\}$.

(6) Select a set L of promising branching variable candidates. Let $Ax \leq b$ be the constraint system of the last solved LP. Solve for each variable $i \in L$ the two linear optimization problems

$$v_0^i = \max c^T x, \quad \text{s.t. } Ax \leq b,\ x_i = 0$$

$$v_1^i = \max c^T x, \quad \text{s.t. } Ax \leq b,\ x_i = 1$$

and select the branching variable b with

$$\max\{v_0^b, v_1^b\} = \min_{i \in L} \max\{v_0^i, v_1^i\}.$$

Some running time can be saved if instead of the solution of the linear optimization problems to optimality only a restricted number of iterations of the simplex-method is performed. Then the objective function value might already indicate the "quality" of the branching variable, especially if a steepest edge pivot selection criterion is applied.

Computational experiments for the strategies (1), (3) – (5) can be found in Jünger, Reinelt and Thienel (1992). Strategy (6) has been suggested by Applegate, Bixby, Chvatal and Cook (1993).

A modification of strategy (1) based on ideas which have their origin in statistics is presented in Padberg and Rinaldi (1991). Other branching variable selection strategies can be found in Balas and Toth (1985).

Instead of partitioning the set of feasible solutions by branching on a variable, it is also possible to use a hyperplane intersecting the polytope defined by the current subproblem. In Clochard and Naddef (1993) a problem specific hyperplane for the symmetric traveling salesman problem is suggested.

Another modification of the branching process is branching on $k \geq 2$ variables or hyperplanes. In this case we get a 2^k-nary instead of a binary branch and cut tree.

SELECT

A branch and cut node is selected and removed from the set of active branch and cut nodes. If the list of active branch and cut nodes is empty, the best known feasible solution is the optimum solution. Otherwise the selected node is processed. After a successful selection the set variables (including the branching variables) must be adjusted. If it turns out that some variable must be set to 0 or 1, yet has been fixed to the opposite value in the meantime, we have a contradiction. In this case the branch and cut node is fathomed. If the local upper bound `lub` of the selected node does not exceed the global lower bound `glb`, the node is fathomed immediately and the selection process is continued.

Up to now we have not specified which node is selected from the set of active branch and cut nodes. There are three well-known enumeration strategies in branch and bound (branch and cut) algorithms: depth-first search, breadth-first search and best-first search. We define the **level** of a branch and cut node B as the number of edges on the path from the root of the branch and cut tree to B. In case of depth-first search a branch and cut node with maximum level in the branch and cut tree is selected from the set of active nodes in SELECT, whereas in breadth-first search a subproblem with minimum level is selected. In best-first search the "most promising" node becomes the current branch and cut node. For a maximization problem the node with maximum local upper bound among all active nodes is often considered as most promising.

Computational experiments for the symmetric traveling salesman problem (see Jünger, Reinelt and Thienel (1992)) show that depth-first search is an enumeration strategy with the "risc" of spending a lot of time in a branch of the tree, which is useless for computing better upper and lower bounds. Often the local upper bound of the current subproblem exceeds the objective function value of an optimum solution, however, this node cannot be fathomed, because no good lower bound is known. The same phenomenon occurs also sometimes when using breadth-first search, but it is very rare if the enumeration is performed in best-first search order.

The idea of **branch pausing**, suggested by Padberg and Rinaldi (1991), can also be helpful in this context. A value `target` is specified as an estimation for the value of the optimum solution. If the local upper bound `lub` falls below the value of `target` the current branch cut node is put back into the set of active nodes (together with its new local upper bound) without performing a branching operation.

It should be noted that it might be appropriate to perform the process of branching and selecting a new node in a different way. Instead of creating immediately two (or more) sons of node in a branching step, Padberg and Rinaldi (1991) suggested to add the current branch and cut node to the set of active nodes. When this node is selected from the set of active nodes it is processed again, since further cutting planes might be generated from the constraint pool (see SEPARATE and subsection 4.6), which have not been available during the first processing. After this second cutting plane phase for a subproblem, the sons of the corresponding branch and cut node are created, if the node could not be fathomed in the meantime.

FATHOM

If for a node the local upper bound `lub` does not exceed the global lower bound `glb`, or if a contradiction occured, or if an infeasible branch and cut node has been generated, or if the LP-solution is an incidence vector of a feasible solution, the current branch and cut node is deleted from further consideration. Even though a node is fathomed, the global lower bound `glb` may have changed during the last iteration, so that additional variables may be fixed by FIXBYREDCOST and FIXBYLOGIMP. The fathoming of nodes in FATHOM may lead to a new root of the branch and cut tree containing the remaining active nodes.

In JÜNGER, REINELT AND THIENEL (1992) the following modification of the procedure FATHOM is suggested. If a node is fathomed because of a contradiction, not only the node, where the contradiction has been found, can be deleted from further consideration, but all active nodes with the same "wrong" setting can be fathomed. Let the variable with the contradiction be e. If in another branch and cut node b the variable e is set to the "wrong" bound all active nodes (unfathomed leaves) in the subtree below b can be removed from the set of active nodes.

OUTPUT

The optimum solution is output and the algorithm stops.

4.3 Computation of upper bounds

The computation of upper bounds consists of all elements of the dashed bounding box except EXPLOIT LP. In EXPLOIT LP the lower bounds are updated, if the solution of the linear optimization problem is the incidence vector of a better feasible solution. Also improvement heuristics, using information of the LP-solutions, can be incorporated here as suggested in subsection 4.4.

For the computation of upper bounds LP-relaxations are solved iteratively, violated valid inequalities are added, and nonbinding constraints are deleted from the constraint matrix.

In this subsection we will also point out that an additional data structure for inequalities, which is called **pool**, is very useful, although not necessary for the correctness of the algorithm. However, if sparse graph techniques are used, which we outline in subsection 4.5, then at least a data structure similar to this pool is required.

The **active inequalities** are the ones in the current LP and are both stored in the pool and in the constraint matrix, whereas the **inactive constraints** are only present in the pool. The pool is initially empty. If an inequality is generated by a separation algorithm, it is stored both in the pool and added to the constraint matrix. Further details of the pool are outlined in subsection 4.6.

INITIALIZE NEW NODE

If the current branch and cut node is the root node of the branch and cut tree the LP is initialized by some small constraint system. Often the upper and lower bounds on the variables are a sufficient choice (e.g. for the max-cut problem). For the traveling salesman problem the degree constraints for all nodes are normally added. Augmenting the initial system by other promising cutting planes can sometimes reduce the overall running time (see Grötschel, Martin, Weismantel (1992a)). A primally feasible basis derived from a feasible solution can be used as a starting basis in order to avoid phase 1 of the simplex algorithm.

Any set of valid (preferably facet defining) inequalities can be used to initialize the first constraint system of subsequent subproblems. Yet, in order to guarantee monotonicity of the values of the local upper bounds in each branch of the enumeration tree, and to save running time, it is appropriate to initialize the constraint matrix by the constraints which were binding when the last LP of the father in the branch and cut tree was solved. These inequalities can be regenerated from the pool.

Since the basis of the father is dual feasible for the initial LP of its sons, phase 1 of the simplex method can be avoided by starting with this basis. The columns of nonbasic set and fixed variables can be removed from the constraint matrix to keep the LP small and if their status is `settoupperbound` or `fixedtoupperbound`, the right hand side of the constraint has to be adjusted, and the corresponding coefficients of the objective function must be added to the optimal value returned by the simplex algorithm in order to get the correct value of the variable `lpval`. Set or fixed basic variables should not be deleted, because this would lead to a neither primal nor dual feasible basis and require phase 1 of the simplex method. The adjustment of these variables can be performed by adapting their upper and lower bounds.

SOLVE LP

The LP is solved by the primal simplex method, if the basis is primal feasible (e.g., if variables have been added) or by the dual simplex method if the basis is dual feasible (e.g., if constraints have been added). The two-phase simplex method is required if the basis is neither primal nor dual feasible. This can happen if constraints necessary for the initialization of the first LP of a subproblem are not available since they had to be removed from the pool as we will describe in subsection 4.6.

The LP-solver is one of the bottlenecks of a branch and cut algorithm. Sometimes more than 90% of the computation time is spent in this procedure. Today, efficient implementations of the simplex method, like CPLEX (CPLEX (1993)) or OSL (IBM (1991)), and of interior point methods, like OSL (IBM (1991)), OB1 (Lustig, Marsten and Shanno (1992b)) CPLEXbarrier (CPLEX (1993)), or LOQO (Vanderbei (1992)), are competitive on solving linear optimization problems from scratch. However, a branch and cut algorithm requires a LP-solver with very efficient post-optimization routines. If sparse graph techniques (cf. subsection 4.5) are used, we also need the values of the dual variables for the computation of the reduced costs of nonactive variables and a technique for the regeneration of infeasible LPs (ADD VARIABLES) must be available.

The simplex method satisfies all the requirements of a branch and cut algorithm and it is used by nearly all implementations of cutting plane algorithms. Therefore we have outlined the algorithm in this section under the assumption that the simplex method is used. Since the LPs, which have to be solved in a cutting plane algorithm, are often highly degenerate, good pivot variable selection strategies, like the steepest-edge pivot variable selection criterion, are necessary. These degeneracies might even require some preprocessing of the LPs (GRÖTSCHEL, MARTIN AND WEISMANTEL (1992a)).

Interior point methods are not sensitive to degeneracies and seem to be more efficient for very large problems. However, the postoptimization routines are still not as sophisticated as those of the simplex method. Interior points methods have been used in cutting plane algorithms only by MITCHELL AND TODD (1992) and MITCHELL AND BORCHERS (1992, 1993). They report similar running times of their implementation of a branch and cut algorithm for the linear ordering problem as the implementation of GRÖTSCHEL, JÜNGER AND REINELT (1984b) using the simplex method (MPSX IBM (1978)). However, the better performance of new implementations of the simplex method could not be considered in this comparison. These results and an encouraging paper about warm starts of LUSTIG, MARSTEN, SHANNO (1992a) give us the hope that interior point methods, probably in combination with the simplex method (see BIXBY, GREGORY, LUSTIG, MARSTEN AND SHANNO (1992)), will improve the performance of the LP-solver in cutting plane algorithms.

EXPLOIT LP

First, we have to check if the current LP solution is the incidence vector of a feasible solution. If this is the case we leave the bounding part and fathom the current branch and cut node. Otherwise, most implementations of branch and cut algorithms proceed with the cutting plane generation phase. However, in JÜNGER, REINELT AND THIENEL (1992) it is suggested to exploit the fractional LP-solutions to improve the lower bound before additional cutting planes are generated. We will discuss these ideas in subsection 4.4. Before the separation phase is performed in SEPARATE, variables may be fixed or set as explained in FIX AND SET.

Often it is reasonable to abort the cutting plane part if no significant increase of `lpval` in the most recent LP-solutions has taken place. This phenomenon is called **tailing-off** (cf. PADBERG AND RINALDI (1991)). If during the last k (e.g. $k = 10$) iterations in the bounding part, `lpval` did not increase by more than p % (e.g. $p = 0.01$), new subproblems are created instead of generating further cutting planes. Good choices for the parameters p and k are both rather problem specific and dependent on the quality of the available cutting plane generation procedures.

SEPARATE

The separation phase is the central part of a branch and cut algorithm. It is tried to find violated globally valid (preferably facet-defining) constraints, which are added to the LP. We say an inequality is globally valid, if it is a valid inequality for every subproblem of the branch and cut algorithm. Facet defining inequalities for the polytope $P_{\mathcal{I}}$ are globally valid. We call a constraint locally valid, if it is only a valid inequality of a subproblem S and all subproblems of the subtree rooted at S. The use of locally valid inequalities in a branch and cut algorithm has not yet been investigated in detail and would require some minor changes in the outlined algorithm.

It may not always be advantageous to call any available separation algorithm in each iteration of the cutting plane generation. Experiments show that a hierarchy of separation routines is often preferable. Certain separation methods should only be performed, if others have failed. Before calling a time consuming exact separation algorithm, one should attempt to generate cutting planes with faster heuristic methods. However, this hierarchy is rather problem specific so that we cannot give a general recipe for its application. We refer to the publications on specific implementations.

The constraint pool provides us with another cutting plane generation technique. Inactive constraints which are violated by the current LP-solution can be regenerated from the pool. Of course this methods requires an efficient algorithm to perform this test and to transform the storage format of the constraint used in the pool into the storage format for the LP-solver. The pool-separation can be advantageous for classes of inequalities for which only heuristic separation routines are available. In this case it can happen that a constraint of this class is violated, yet cannot be identified by the heuristic. However, this cutting plane might have been generated earlier in the computational process (at a different LP-solution which has been more "convenient" to our heuristic). If this constraint is still contained in the pool, it can be reactivated now.

It can also happen that the pool-separation for a class of constraints is faster than a direct separation by a time consuming heuristic or an exact algorithm. So the pool separation should be performed before calling these algorithm. However, for other classes of constraints it can sometimes be observed that the pool separation is very slow in comparison to direct separation methods.

Since the pool can become very large during the computational process, it is necessary to limit the search in the pool for violated inequalities. For instance, the pool separation can be restricted to some classes of constraints. Therefore the pool separation should be carefully included into the hierarchy of separation algorithms and it requires many computational tests to find a strategy which is efficient for a specific combinatorial optimization problem.

For some combinatorial optimization problems like the max-cut problem and for larger instances of other problems, sometimes several hundred violated inequalities can be generated. However, it would be sufficient to add those constraints to the LP, which will be binding after the solution of the next LP. Unfortunately we do not know this subset of the generated inequalities. On the other hand, adding all the constraints to the matrix of the LP-solver can slow down the overall computation time. Therefore, depending on the performance of the LP-solver, only a limited number of constraints should be added to the LP. A straightforward approach is just stopping the cut generation when this limit is reached. A more sophisticated method might be generating as many constraints as possible, and afterwards selecting the "best" of them. A simple classification criterion is the degree of violence given by the value of the corresponding slack. For the symmetric traveling salesman problem PADBERG AND RINALDI (1991) propose as a measure the distance of the LP-solution from the projection of the cut into the affine space defined by the degree equations. The larger this distance the better the cut, yet, this method is computationally expensive.

The representation of the inequality for the LP-solver can have significant influence on its running time. For instance, equations of the integer programming formulation can be added to any valid inequality without changing the halfspace which it defines. However, the number of the non-zero coefficients in the inequality may differ. Normally, LP-solvers are more efficient if the number of non-zeros in the constraint matrix is small.

The solution of the separation problem is very problem specific. Therefore we only want to present an example for the symmetric traveling salesman problem.

A polynomial time algorithm for the solution of the exact separation problem of subtour elimination constraints can be directly derived from their definition in section 1: If the value of the minimum cut in the support graph (the graph with the LP-solution as edge weights) is greater or equal than 2, it is proved that the current LP-solution does not violate any subtour elimination constraint. Each cut with a value less than 2 induces a violated subtour elimination constraint.

A straightforward algorithm for the solution of the minimum cut problem in a graph $G = (V, E)$ with n nodes and m edges is the computation of the minimum s-t cut between each pair of nodes $s, t \in V$ and selecting from these $\binom{n}{2}$ cuts one with the minimum weight. However, GOMORY AND HU (1961) showed that — if some care is taken in the order of the computation of the s-t cuts — it is sufficient to solve $n-1$ minimum s-t cut problems. The minimum s-t cut problem can be solved in polynomial time. A survey on algorithms for the solution of this problem can be found in AHUJA, MAGNANTI AND ORLIN (1993).

The algorithm of GOMORY AND HU (1961) has been refined by GUSFIELD (1989) and PADBERG AND RINALDI (1990a). Further algorithm for the solution of the minimum cut problem have been introduced by NAGAMOCHI AND IBARAKI (1992a, 1992b), KARGER (1993) and KARGER AND STEIN (1993).

ELIMINATE

If only inequalities are added to the constraint matrix of the LP solver, yet no inequalities are eliminated, soon the size of the matrix might become to large to solve the linear programs in reasonable time and even the storage of the constraints in the matrix would require too much memory. Moreover, there are inequalities which become redundant for the rest of the computation. Therefore a strategy is required to maintain a reasonable sized matrix, yet not to eliminate important inequalities.

It is an obvious and simple strategy for the elimination of constraints to delete all active inequalities which are nonbinding in the last LP solution from the constraint structure before the LP is solved after a successful cutting plane generation phase. To avoid cycling, i.e., a constraint is eliminated, but already violated after the next LP-solution, either constraints should be only removed, if the value of the slack s is big enough (e.g. $s > 0.001$), or if they are nonbinding during several successive LP-solutions.

4.4 Computation of feasible solutions

For most combinatorial optimization problems a host of heuristics is available to compute feasible solutions which provide global lower bounds for the branch and cut algorithm. Traditionally the computation of a global lower bound is performed in the procedure INITIALIZE before the cutting plane generation and enumeration phase starts. Later better lower bounds are only found if the LP-solution is the incidence vector of a feasible solution. However, it can be observed that this happens rather seldomly. Therefore sophisticated heuristics must be applied in INITIALIZE to generate a good lower bound. Otherwise, the enumeration tree may grow too large.

In Jünger, Reinelt and Thienel (1992) a dynamic strategy, integrated in the cutting plane generation part, for the computation of lower bounds is presented, which we briefly outline. It turns out that the fractional LP solutions occuring in the upper bound computations in a branch and cut algorithm give hints on the structure of optimum or near optimum feasible solutions.

The basic requirement for the upper bound computations is efficiency in order not to inhibit the optimization process. While in the first stages high emphasis is laid on providing good feasible solutions, this emphasis is less in the later stages of the computational process. On the other hand, computing lower bounds can always be reasonable since new knowledge about the structure of optimum feasible solutions is acquired (e.g. because of fixed and set variables).

Exploiting the LP solution

Integer optimal solutions, i.e., incidence vectors of feasible solutions, will almost never result from the LPs occuring in the branch and cut algorithm. But, it can be observed that these solutions, although having many fractional components, give information on good feasible solutions. They have a certain number of variables equal to 1 or 0 and also a certain number of variables whose values are close to 1 or 0. This effect can be exploited to form a starting feasible solution for subsequent improvement heuristics.

We show how the information of the LP-values of the variables can be used for the construction of a feasible solution for the symmetric traveling salesman problem. We use the terms edge and variable of the integer programming formulation interchangeably, since they are in a one-to-one correspondence in our examples. First, we check if the current LP solution is the incidence vector of a tour. If this is the case we terminate the procedure EXPLOITLP. Otherwise, edges are sorted according to their values in the current LP solution. We give decreasing priorities to edges as follows:

- edges that are fixed or set to 1,
- edges equal to 1 or close to 1 in the current LP,
- edges occuring in several successive LPs.

This list is scanned and edges become part of the tour if they do not produce a subtour with the edges selected so far. This gives a system of paths and isolated nodes which now have to be connected. To this end a savings heuristic (Clarke and Wright (1964)), originally developed for vehicle routing problems, can be used, since the traveling salesman problem can be considered as a special vehicle routing problem involving only one vehicle.

This heuristic basically consists of successively merging partial tours to obtain a Hamiltonian tour. We select one node as base node and form partial tours by connecting this base node to the end nodes of each of the paths obtained in the selection step and also adding a pair of edges to nodes not contained in any path. Then, as long as more than one subtour is left, we compute for every pair of tours the savings that is achieved if the tours are merged by deleting in each tour an edge to the base node and connecting the two open ends. The two tours giving the largest savings are merged. Edges which are fixed or set to 0 should be avoided for connecting paths.

Another example is the construction of a feasible solution for the max-cut problem on a graph $G = (V, E, w)$ with edge weights w. After the solution of an LP we construct a graph $G' = (V, E, w')$ with $w'_e = |x_e - 0.5|$, where x_e is the LP-value of the variable e. We compute a maximum spanning tree in the graph G'. Afterwards, we select an arbitrary root node and color it black. All other nodes are colored black and white in the following way. We scan the tree starting at the root node by breadth-first search and color a node with the color of its father, if the LP-value of the tree edge is less than 0.5, and otherwise with the opposite color. The colored nodes define now the two shores of a cut. Edges with a high LP-value tend to have a black and a white endnode.

For the computation of feasible solutions for the linear ordering problem on a complete digraph $D_n = (V_n, A_n)$ with arc weights c_a we use the following approach as described in Grötschel, Jünger, Reinelt (1984b). We calculate for every node $v \in V_n$

$$s(v) = \sum_{u<v} c_{vu}(1 - x_{uv}) + \sum_{v<w} c_{vw}x_{vw}$$

and sort the nodes such that $s(v_1) \geq s(v_2) \geq \ldots \geq s(v_n)$. So we generate the linear ordering $v_1, v_2, \ldots, v_n$.

Improving the first solution

The previous step gives us a feasible solution which can be improved by local improvement heuristics. At least as long as a branch and cut algorithm is implemented on a sequential machine, we must take care of a proper distribution of CPU time between the lower bounding and the upper bounding part. In addition, the amount of work that is spent in the heuristics has to be controlled. Various strategies are possible.

– Fixed percentage of CPU time

  In advance a certain percentage of CPU time is specified that is spent for lower bound computations. Whenever, after having solved an LP, this percentage is not reached, a lower bound computation is initiated. This strategy has the disadvantage that it is not flexible and can miss LP solutions that would lead to an improvement of the current feasible solution.

– Fixed iteration number

In this case a lower bound computation is started whenever a certain number of LPs has been solved. This strategy has the same disadvantage as the previous one and, in addition, does not take care of the CPU time spent for the heuristics.

– Dynamic strategy

Here some possible guidelines are specified for increasing the chance that a lower bound computation is promising and should be initiated. Of course the strategy must be adapted to every special combinatorial optimization problem. After every LP, the construction heuristic is performed to exploit the LP solution. The improvement heuristic might be inhibited if the starting solution is much worse than the best solution found so far. Depending on the progress of the improvement heuristics it can be decided how much effort is spent in this part. For example, if the value of the improved solution comes close to the value of the best known solution very fast, then still more CPU time should be spent. But if progress is slow, then an early termination of the improvement might be favourable.

Improving the same feasible solution several times can be avoided by using a hashing scheme to detect identical feasible solutions. As the hash key, the objective function value of the feasible solution and the name of the heuristic which is applied to this feasible solution can be used. However, if there are many different feasible solutions with the same objective function value, another hash key should be chosen.

4.5 Problems with sparse solutions

Often combinatorial optimization problems involve a very large number of variables, yet a feasible solution is comparatively sparse. For instance, the symmetric traveling salesman problem on a complete graph of n nodes has $\binom{n}{2}$ variables. Yet, a tour consists only of n edges. Hence, the computational process can be accelerated, if a suitable subset of the edges is initially selected and appropriately augmented during the solution of the problem, if this is required for the correctness of the algorithm. However, sparse graph techniques can not be applied to problems with a dense solution structure like the max-cut problem or the linear ordering problem. Sparse graph techniques have been introduced by GRÖTSCHEL AND HOLLAND (1985).

We present techniques exploiting the sparsity of solutions only for combinatorial optimization problems defined on graphs. However this technique can be generalized for other problems, if the structure of the solutions is sparse, suitable subsets of the variables can be computed efficiently, and a method to generate the columns of nonactive variables is available.

In Figure 5 we present the modified flowchart for the use of sparse graph techniques. The gray boxes have to be added or changed. A subproblem, in which an infeasible LP is detected, cannot be fathomed at once, but it has to be checked, if the addition of nonactive variables can regenerate the feasibility. We explain this process in ADDVARIABLES.

Before leaving the bounding part, it has to be verified in PRICE OUT, if the LP-solution computed on the sparse graph is also optimal on the complete graph. Only in this case the variable `lpval` becomes a local upper bound `lub`. The application of the routine FIX AND SET has to be performed now more carefully. The procedure SETBYREDCOST can only be applied after an additional pricing step, in which no variable has to be added. This is also the case for FIXBYREDCOST if the root node of the remaining branch and cut tree is currently processed.

Suitable sparse graphs

The initial sparse graph is generated in the procedure INITIALIZE. For some problems a good choice for a sparse graph is the k-nearest neighbour graph. An other suitable subset of the edges may be the Delaunay graph (see also Reinelt (1992), Clochard and Naddef (1993)). Padberg and Rinaldi (1991) suggest to create heuristically a series of feasible solutions and initialize the sparse graph with all involved edges. If it cannot be guaranteed that the sparse graph contains a feasible solution, it should be augmented by the edges of a solution computed by a heuristic.

In addition to the sparse graph, the edges of the "reserve graph" can be computed. These edges are additional "promising" edges, which do not belong to the sparse graph. For instance, if the sparse graph is the 5-nearest neighbour graph, a suitable reserve graph is given by the edges which have to be added to get the 10-nearest neighbour graph. The reserve graph can be used in PRICE OUT and ADD VARIABLES.

The algorithm starts working on G, adding and deleting edges (variables) dynamically during the optimization process. We refer to the edges in G as **active** edges and to the other edges as **nonactive** edges.

ADD VARIABLES

Variables have to be added to the sparse graph if indicated by the reduced costs (handled by PRICE OUT) or if the current LP is infeasible. The latter may be caused by two reasons.

First, some active inequality has a void left hand side, since all involved variables are fixed or set and removed from the LP, but is violated. If all coefficients of nonactive variables in this inequality are nonnegative, it is clear from our strategy for variable fixings and settings that the branch and cut node is fathomed (all constraints are assumed to be of the form $a^T x \leq b_i$). However, if there is a nonactive variable with a negative coefficient, this variable may remove the violation. So it is added to the LP.

Second, the above condition does not apply, and the infeasibility is detected by the LP solver. In this case a pricing step is performed in order to find out if the dual feasible LP solution is dual feasible for the entire problem. Variables that are not in the current sparse graph (i.e., are assumed to be at their lower bound 0) and have negative reduced cost are added to the current sparse graph. An efficient way of computing the reduced costs is outlined in PRICE OUT.

If variables have been added, the new LP is solved. Otherwise, it is tried to make the

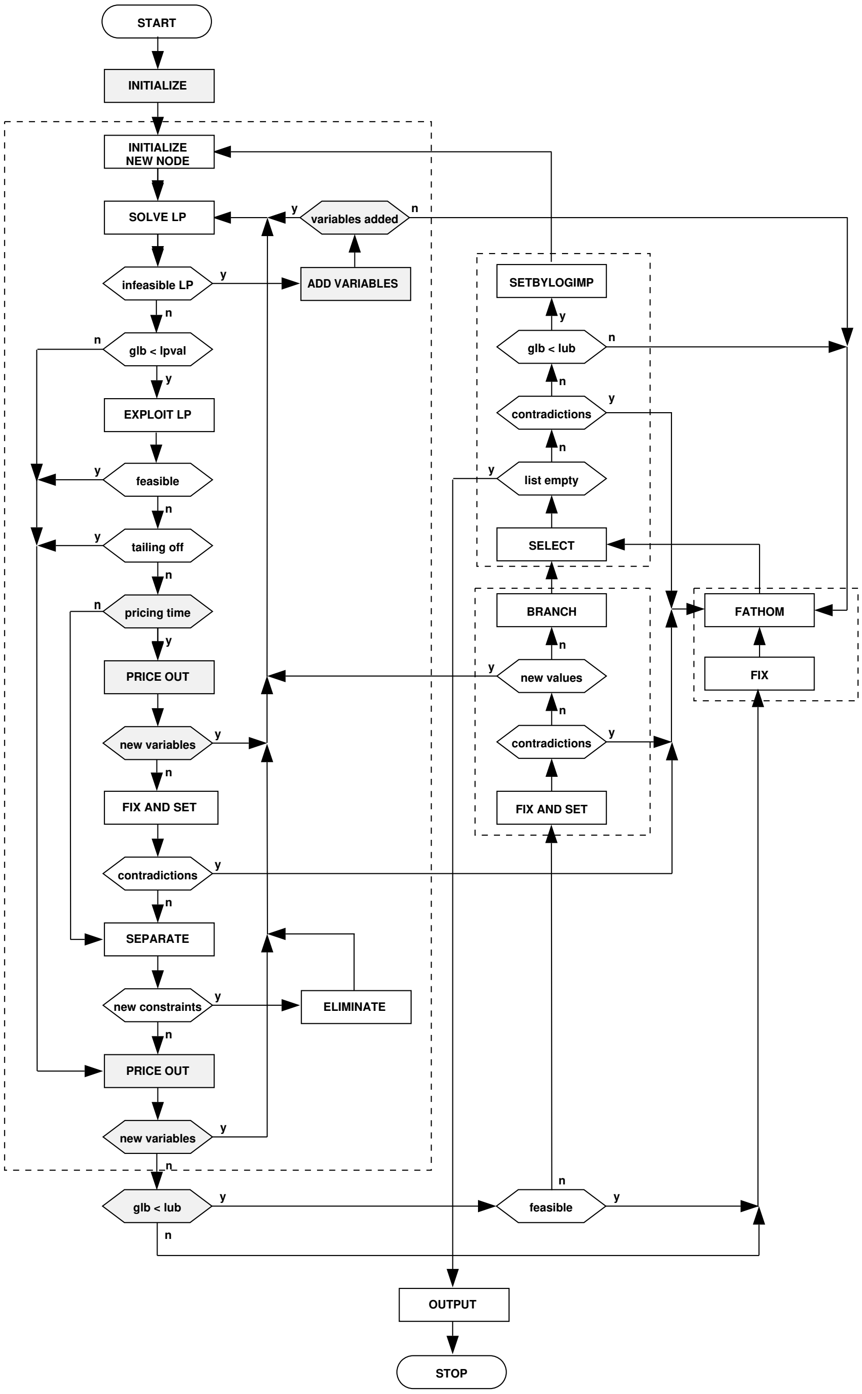

Figure 5. Flowchart of the branch and cut algorithm with sparse graph techniques.

LP feasible by a more sophisticated method. The LP value `lpval`, which is the objective function value corresponding to the dual feasible basis where primal infeasibility is detected, is an upper bound for the objective function value obtainable in the current branch and cut node. So if $\texttt{lpval} \leq \texttt{glb}$, the branch and cut node can be fathomed.

Otherwise, it is tried to add variables that may restore feasibility. First all infeasible variables are marked, including negative slack variables.

Let e be a nonactive variable and r_e be the reduced cost of e. An edge e is taken as a candidate only if $\texttt{lpval} + r_e \geq \texttt{glb}$. Let B be the basis matrix corresponding to the dual feasible LP solution, at which the primal infeasibility was detected. For each candidate e let a_e be the column of the constraint matrix corresponding to e and solve the system $B\overline{a}_e = a_e$. Let $\overline{a}_e(b)$ be the component of $\overline{a}_e$ corresponding to basic variable x_b. Increasing x_e "reduces some infeasibility" if one of the following holds.

– x_b is a structural variable (i.e., corresponding to an edge of G) and

$$x_b < 0 \text{ and } \overline{a}_e(b) < 0$$

or

$$x_b > 1 \text{ and } \overline{a}_e(b) > 0$$

– x_b is a slack variable and

$$x_b < 0 \text{ and } \overline{a}_e(b) < 0.$$

In such a case the variable e is added to the set of active variables and the marks are removed from all infeasible variables whose infeasibility can be reduced by increasing x_e. This can be done in the same hierarchical fashion as described below in PRICE OUT.

If variables can be added, the new LP is solved, otherwise the branch and cut node is fathomed. Note that all systems of linear equations that have to be solved have the same matrix B, and only the right hand side a_e changes. This can be utilized by computing a factorization of B only once, in fact, the factorization can be obtained from the LP solver for free. For further details on this algorithm, see Padberg and Rinaldi (1991).

PRICE OUT

Pricing is necessary before a branch and cut node can be fathomed. Its purpose is to check if the LP solution computed on the sparse graph is valid for the complete graph, i.e., all nonactive variables "price out" correctly. If this is not the case, nonactive variables with positive reduced cost are added to the sparse graph and the new LP is solved using the primal simplex method starting with the previous (now primal feasible) basis, otherwise the local upper bound `lub` and possibly the global upper bound `gub` can be updated.

Although the correctness of the algorithm does not require this, additional pricing steps can be performed every k (e.g. $k = 10$) solved LPs (see Padberg and Rinaldi (1991)). The effect is that nonactive variables which are required in a good or optimum feasible solution tend to be added to the sparse graph early in the computational process. If no variables are added, it can be also tried to fix or set variables by reduced cost criteria.

Let y be the vector of the dual variables, and A_e the column of an inactive variable e in the matrix A defined by the active constraints, and c_e the corresponding objective function coefficient, then the reduced costs of the variable e are given by $r_e = c_e - y^T A_e$.

The computation of the reduced cost for all inactive edges takes a lot of computational effort, but it can be performed significantly faster by an idea of PADBERG AND RINALDI (1991). If our current branch and cut node is the root of the remaining branch and cut tree, it can be checked if the reduced cost r_e of a nonactive variable e satisfies the relation $\texttt{lpval} + r_e < \texttt{glb}$. In this case this nonactive edge can be discarded forever. During the systematic enumeration of all edges of the complete graph, an explicit list of those edges which remain possible candidates can be made. In the early steps of the computation, too many such edges remain, so that this list cannot be completely stored with reasonable memory consumption. Instead, a partial list is stored in a fixed size buffer and the point where the systematic enumeration has to be resumed after considering the edges in the buffer is memorized. In later steps of the computation there is a good chance that the complete list fits into the buffer, so that later calls of the pricing routine become much faster than early ones.

In JÜNGER, REINELT AND THIENEL (1992) a further modification of the procedure PRICE OUT is presented. It can be observed that the reduced costs of edges not belonging to the reserve graph are seldomly positive, if the reserve graph is appropriately chosen. Hence, it turns out that a hierarchical approach is advantageous. Only if the "partial pricing" considering the edges of the reserve graph has not added variables, the reduced cost of all nonactive variables have to be checked.

To process PRICE OUT efficiently, the columns of the matrix for inactive variables have to be generated quickly. Suitable formats of the pool as suggested in PADBERG AND RINALDI (1991) and JÜNGER, REINELT AND THIENEL (1992) can provide an efficient method for these computations.

Computation of lower bounds

Sparse graph techniques can be also used for the computation of feasible solutions both if heuristics are applied only during the initialization phase or if they are integrated in the cutting plane generation phase.

A candidate subgraph is a subgraph of the complete graph on n nodes containing reasonable edges in the sense that they are "likely" to be contained in a good feasible solution. These edges are taken with priority in the various heuristics, thus avoiding the consideration of the majority of edges which are assumed to be of no importance. Various candidate subgraphs and the question of how to compute them efficiently are discussed in REINELT (1992) and JÜNGER, REINELT & RINALDI (1994).

The candidate subgraph can be related to the set of active variables in the linear programming problems, if the heuristics are integrated into the cutting plane generation part as described before. Basically, the candidate subgraph is initialized with some graph (e.g., the empty graph) and then edges are added whose corresponding values are close to one. In order to avoid too extensive growing of the candidate subgraph and to avoid being biased by LPs that were not recently solved, the candidate subgraph should be cleared in certain intervals (e.g. every 20th cutting plane phase) and reinitialized.

It should be noted that the feasible solution found by the heuristics should not be restricted to only using edges of the sparse graph. These edges are only considered with priority and lead to an acceptable CPU time. Usually, heuristics will introduce edges that are not active in the LP. These edges are added to the set of active variables. This is based on the assumption that these edges are also important for the upper bound computations and would be added to the LP in some pricing step anyway. This way the set of active variables is augmented without pricing.

4.6 Data structures

A cpu time and memory sensitive implementation of data structures is crucial for an efficient branch and cut algorithm.

Graph

Many combinatorial optimization problems are defined on graphs, i.e., the feasible solutions are subgraphs of a given graph. The representation of the (sparse) graph has a high influence on the running time. The operations on the graph used in the heuristics for the computation of feasible solutions and in the separation algorithms should be performed efficiently. If sparse graph techniques are applied, also edges have to be added to the graph data structure. Suitable representations of sparse graphs in a branch and cut code can be found in PADBERG AND RINALDI (1991) and JÜNGER, REINELT AND THIENEL (1992).

Branch & cut nodes

Although a subproblem is completely defined by the fixed variables and the variables that were set temporarily, it is necessary to store additional information at each node for an efficient implementation.

Of course it would be correct to initialize the constraint system of the first LP of a new selected node with the inequalities of the last processed node, as long as only globally valid inequalities are used as cutting planes. However, this would lead to tedious recomputations, and it is not guaranteed that we can regenerate all heuristically separated inequalities. So it is preferable to store in each branch and cut node pointers to those constraints in the pool, which are binding the last solved LP of the node. We initialize with these constraints the first LP of each son of that node.

If the simplex method is used to solve the linear programs, we store the basis of the last processed LP of each node, i.e., the statuses of the variables and the constraints. Therefore we can avoid phase 1 of the simplex algorithm, if we carefully restore the LP of the father and solve this first LP with the dual simplex method. Since the last LP of the father and the first LP of the son differ only by the set branching variable(s) or the added branching hyperplane(s), by variables set by SETBYLOGIMP and by variables, which have been fixed in the meantime, the basis of the father is dual feasible for the first LP of the son.

Active nodes

In SELECT a node is extracted from the set of active nodes for further processing. Every selection strategy defines an order on the active nodes. The minimum node is the next selected one. The representing data structure must allow efficient implementations of the operations `insert`, `extractmin` and `delete`. The operation `insert` is used after creation of new branch and cut nodes in BRANCH, `extractmin` is necessary to select the next node in SELECT and `delete` is called if we remove an arbitrary node from the set of active nodes. These operations are very well supported by a height balanced binary search tree, e.g., a red-black tree (Bayer (1972), Guibas and Sedgewick (1978), see also Cormen, Leiserson and Rivest (1990)) which provides $O(\log t)$ running time for these operations, if the tree consists of t nodes.

Temporarily set variables

A variable is either set if it is the branching variable or it is set by SETBYREDCOST or SETBYLOGIMP. If the modification of FATHOM of Jünger, Reinelt and Thienel (1992) is used, it is essential to determine efficiently all nodes where a certain variable is set. To avoid scanning the complete branch and cut tree, we apply a hash function to a variable right after setting and store in the slot of the hash table the set variable and a pointer to the corresponding branch and cut node. So it is quick and easy to find all nodes with the same setting by applying an appropriate hashing technique.

Constraint pool

In the description of the branch and cut algorithm we pointed out that an additional data structure for the constraints is useful for the initialization of LPs and for the separation phase. If sparse graph techniques are applied we use this additional data structure to generate new columns after addition of variables or when initializing a new node.

This constraint pool can grow very large. Therefore a memory sensitive format, which provides an efficient generation of the rows for the matrix of the LP-solver and a fast computation of the coefficients of inactive variables should be used. A very sparse node oriented format is often possible for many classes of facet defining inequalities of combinatorial optimization problems on graphs (see Padberg and Rinaldi (1991) and Jünger, Reinelt and Thienel (1992)).

If the pool is used as we suggested, this is the data structure using up the largest amount of memory. Therefore constraints also have to be carefully eliminated from the pool. For instance, it would be sufficient to keep only those inactive constraints in the pool, which have been binding, when the last LP of the father of at least one active node has been solved. These inequalities can be used to initialize the first LP of a newly selected node. After each selection of a new node we can try to eliminate those constraints from the pool which are neither active at the current branch and cut node nor necessary to initialize the first LP of an active node. If, nevertheless, the pool grows too large and the memory limit is reached, nonactive constraints must be removed from the pool. But now we cannot restore the complete LP of the father of an active node. In this case one can proceed as in FIX AND SET to initialize the constraint matrix and to get a feasible basis.

Sometimes it might also be appropriate to keep some set of important constraints always in the pool that they can be used for the pool separation.

5 Parallelization

Although a lot of progress has been made in the parallelization of exact and heuristic algorithms for combinatorial optimization problems during the last years (for a survey see Grama and Kumar (1992)), parallel approaches for cutting plane algorithms have not been used very often up to now. In Cannon (1988) and Cannon and Hoffman (1990) a parallel branch and cut algorithm for zero-one programming problems is presented.

In this survey we only want to point out some very basic ideas where the application of parallelization could be useful in a branch and cut algorithm.

There are four main bottlenecks in any branch and cut algorithm: the size of the enumeration tree, the computation of feasible solutions by heuristics, the generation of violated inequalities and the solution of the linear programs.

The processing of the subproblems of the branch and cut tree can be parallelized in a straightforward way. This approach has already been used for branch and bound algorithms on many different computer architectures. Also Applegate, Bixby, Chvatal and Cook (1993) use this method on a cluster of workstations for the solution of symmetric traveling salesman problems by branch and cut.

The improvement of feasible solutions by heuristics is completely independent from the rest of the branch and cut algorithm. At the start of the algorithm it is sufficient to have a lower bound derived from a feasible solution which is computed by a very simple heuristic. Therefore better feasible solutions can be computed in parallel to the branch and cut algorithm either in the traditional way or by exploiting fractional LP-solutions as outlined in subsection 4.4.

If the heuristics are integrated into the cutting plane generation part, some processors can perform improvement heuristics all the time. Only new starting solutions found by EXPLOIT LP must be broadcasted to the "improvement-processors", which only have to inform the other processors about better global lower bounds. Moreover, parallel improvement algorithms can be used, which are available for several combinatorial optimization problems.

Also different separation methods can be performed in parallel, but the hierarchy of separation routines should be redesigned. As already mentioned in section 4, the pool separation sometimes becomes a bottleneck of the separation process, especially if there are very dense constraints, i.e., inequalities with a very small number of non-zero elements. However this pool separation can be performed on a massively parallel machine. A lot of potential for parallelization is also contained in the separation routines themselves. For instance, often some basic combinatorial optimization problem has to be solved for the identification of violated constraints (e.g., a min-cut problem for the separation of subtour elimination constraints). Often parallel algorithm for these combinatorial optimization problems are already available.

Finally, parallel methods for the solution of linear programs are currently being developed.

Although the computation of reduced costs is not very expensive for problems considered so far in comparison to the mentioned bottlenecks of the algorithm, it can become quite time consuming for larger problems with very dense inequalities. In this case parallelization should be taken into consideration.

6 Provably Good Solutions

The cutting plane part together with the enumeration scheme provides an algorithm that is capable of solving an instance of a combinatorial optimization problem in "finite time". But, not even a rough estimation of the necessary running time for an instance can be given. The size of a problem instance is only an insufficient characterization of its hardness.

An algorithm, which provides successively improving lower bounds on the objective function value of an optimum solution and only guarantees that in the end an optimum solution is found, is not adequate for practical problem solving. In fact, it may even turn out to be useless. A reasonable practical requirement is that, on the first hand, a good feasible solution is given quickly and that, on the second hand, better solutions become available as more running time is spent.

Moreover, in practical problem solving optimum solutions are often not required. A practitioner might be satisfied with a feasible solution and a guarantee of the form that this solution is at most $p\%$ worse than the optimum solution. The values of the global lower bound `glb` and the global upper bound `gub` can be used to terminate the computation as soon as the guarantee requirement is satisfied.

A branch and cut algorithm as outlined in the previous sections produces a sequence of decreasing global upper bounds as well as a sequence of increasing feasible solutions. Figure 6 gives us a typical example of this effect in the case of the traveling salesman problem. However, the symmetric traveling salesman problem is a minimization problem, therefore we have to swap the terms "upper bound" and "lower bound". For an instance on 532 cities (problem `att532` from TSPLIB (Reinelt (1991a, 1991b))) we show the development of the upper and lower bounds during the first 15 minutes of the computation on a SUN SPARCstation 2. The jumps in the lower bounds are due to the fact that sparse graph techniques are applied. The validity of the LP-value as a global lower bound for the length of a shortest tour is only guaranteed after a pricing step in which all nonactive variables price out correctly. It should be mentioned that after 15 minutes the gap between the lower and upper bound is less than 0.5%, yet, another three hours are necessary to solve the problem to optimality with the implementation of Jünger, Reinelt and Thienel (1992).

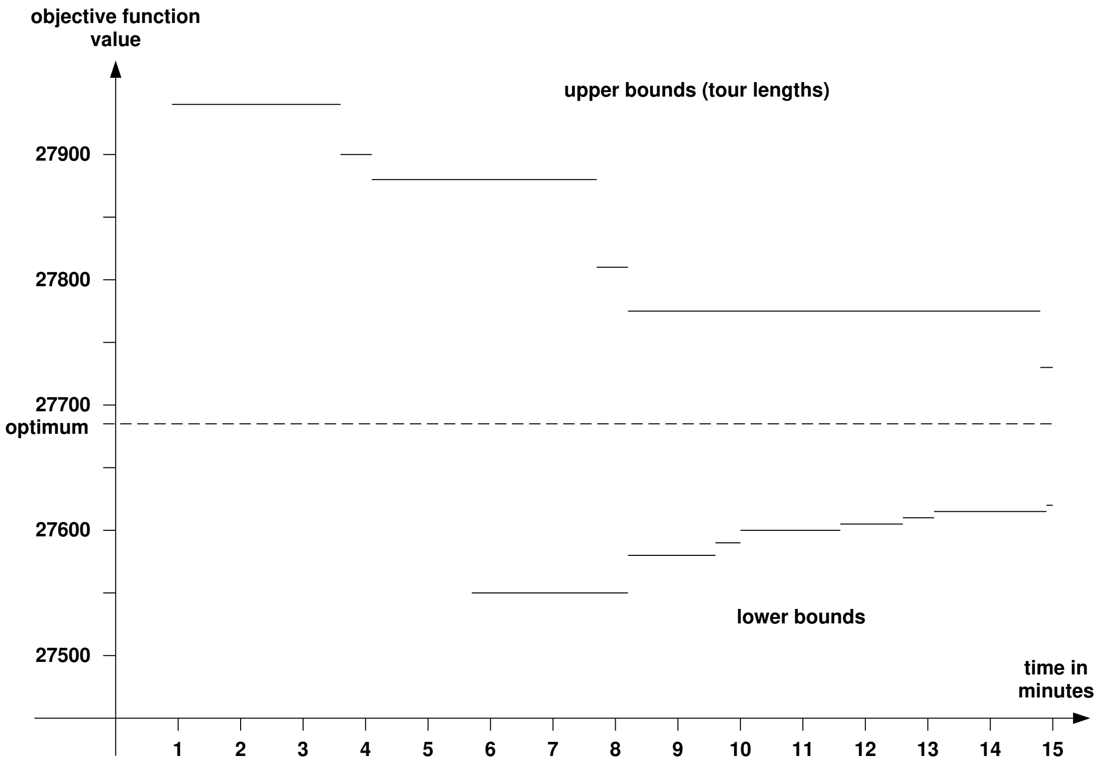

Figure 6. Gap versus time plot for att532.

7 Applications of cutting plane algorithms

Cutting plane approaches seem to have become the method of choice for attacking hard combinatorial and integer programming problems and there is an ever increasing list of publications reporting their successful application. We give reference to a selection of computational studies based on cutting plane algorithms sorted according to the specific optimization problem.

Equicut problem

The equicut problem consists of finding a max-cut $W \subseteq V$ in a graph $G = (V, E)$ with edge weights such that $\lfloor \frac{V}{2} \rfloor \leq |W| \leq \lceil \frac{V}{2} \rceil$. An application is the determination of ground states of Ising spin glasses at zero magnetization.
References: Brunetta, Conforti and Rinaldi (1994), Barahona and Casari (1988)

Generalized assignment problem

Given j jobs and $p \geq j$ persons and costs $c_{jq} \in \mathbb{R}$ if person q does job j. The assignment problem is to allocate to each person at most one job, such that all jobs are done and the total costs are minimized. In the generalized assignment problem, a person can perform more than one job, yet job j requires r_j units of time and each person q can spend at most t_q units of time to process its jobs.
Reference: Savelsbergh (1993)

Generalized traveling salesman problem

The generalized traveling salesman problem is the problem of finding a minimum length cycle $C \subseteq E_n$ in a graph $K_n = (V_n, E_n)$ with edge weights, where V is partitioned into $V = V_1 \cup V_2 \cup \ldots \cup V_k$, such that $|\bigcup_{e \in C} e \cap V_i| \geq 1$ for all $1 \leq i \leq k$. There is a variant of the problem in which $|\bigcup_{e \in C} e \cap V_i| = 1$ must hold.
Reference: Fischetti, Gonzalez and Toth (1994)

General (mixed) integer programming

We have defined the mixed integer programming problem in section 1.
References: Gomory (1958, 1960, 1963), Crowder, Johnson, Padberg (1983), Van Roy and Wolsey (1987), Cannon (1988), Cannon and Hoffman (1990), Hoffman and Padberg (1991), Balas, Ceria and Cornuejols (1993a, 1993b), Ceria (1993), Boyd (1993a, 1993b, 1993c), Savelsbergh, Sigismondi and Nemhauser (1994)

Graphical traveling salesman problem

The graphical traveling salesman problem is a variant of the traveling salesman problem in which each node must be visited at least once (it may be visited more than once) and each edge may be traversed more than once. When the triangle inequality holds for the intercity distances, an optimal solution is always a traveling salesman problem tour. Most problem instances have this property, therefore most computational studies have been carried out for the traveling salesman problem rather than the graphical traveling salesman problem (see below).
Reference: Miliotis, Laporte and Nobert (1981)

Graph partitioning and clustering problems

These problems have many different variants so that we can define only a generic problem variant. Given a graph $G = (V, E)$ and edge weights $c \in \mathbb{R}^E$, find a partition of the node set V in subsets $V_1, V_2, \ldots, V_k$ subject to some side constraints such that $\sum_{i=1}^{k} c(E(V_i))$ is maximum. Examples for side constraints are the restriction of the number of nodes in a subset, or that each subset V_i must induce a clique. Other variants of the graph partitioning problem are the max-cut and the equicut problem.
References: Wakabayashi (1986), Grötschel and Wakabayashi (1989), Weismantel (1992), Holm and Sorensen (1993)

Linear ordering problem

The linear ordering problem has been introduced in section 1. Applications of this problem are, e.g., triangulation of input-output matrices, aggregation of individual preferences, and minimization of completion time in special one-machine scheduling problems.
References: Grötschel, Jünger and Reinelt (1984a, 1984b), Reinelt (1985), Mitchell and Borchers (1992, 1993)

Matching problem

The b-matching problem for a graph $G = (V, E)$ with edge weights and a vector $b \in \mathbb{N}^V$ is to find a minimum weight edge set $F \subseteq E$ such that $|F \cap \delta(v)| \leq b_v$ for all nodes $v \in V$.

If the inequality is replaced by the equation $|F \cap \delta(v)| = b_v$ we get the perfect b-matching problem. Applications are plotting problems.
References: Grötschel and Holland (1985, 1987), Mitchell and Todd (1992)

Max-cut problem

We have presented the max-cut problem in section 1. Applications of this problem are the determination of the ground state of Ising spin glasses and the via minimization problem in VLSI design.
References: Grötschel, Jünger and Reinelt (1987), Barahona, Grötschel, Jünger and Reinelt (1988), Barahona, Jünger and Reinelt (1989), De Simone and Rinaldi (1992)

Maximum planar subgraph problem

Given a graph $G = (V, E)$ with edge weights, find a planar subgraph $G' = (V, E')$ of G of maximum total weight. Applications occur in automatic graph drawing and VLSI design.
Reference: Jünger and Mutzel (1993a, 1993b)

Multiple salesman problem

The multiple salesman problem is a variant of the traveling salesman problem. Instead of a single salesman, there are m salesmen all located at the same city (the depot). The problem is to find a collection of m edge disjoint cycles such that each city is visited exactly once (except the depot) and the total sum of the length of all cycles is minimum. Applications of this problems — often implying additional side constraints — can be found in vehicle routing (see below).
Reference: Laporte and Nobert (1980)

Network survivability problem

Given is a graph $G = (V, E)$ with edge weight $c \in \mathbb{R}^E$ and a vector $d \in \mathbb{N}_0^V$, find a set of edges $F \subseteq E$ at minimum cost $c(F)$ such that for any two nodes $v, w \in V$, there are $\min\{d_v, d_w\}$ many node disjoint paths in F between v and w. An application is the design of survivable networks, especially telephone networks in which the damage of some equipment does not disconnect important links.
References: Grötschel, Monma and Stoer (1992a, 1992b, 1994), Stoer (1992)

Node packing problem

In a graph $G = (V, E)$ with node weights $c \in \mathbb{R}^V$ we want to find a node set $W \subseteq V$ such that no two nodes $v, w \in W$ are adjacent in G and $\sum_{v \in W} c_v$ is maximum. An application is the allocation of radio frequencies to transmitters.
Reference: Nemhauser and Sigismondi (1992)

Rural postman problem

For a graph $G = (V, E)$ with edge weights $c \in \mathbb{R}^E$ and a set $R \subseteq E$ of required edges, find a connected Eulerian subgraph (a subgraph in which all node degrees are even) with edge set $F \subseteq E$ such that $R \subseteq F$ and $c(F)$ is minimum. If R is connected, the problem is a Chinese postman problem, which can be reduced to the matching problem described

above. In general, the rural postman problem is $\mathcal{NP}$-hard. Applications of this problem are plotting problems.
References: Corberan and Sanchis (1991), Jünger, Reinelt and Rinaldi (1994)

Scheduling

There are many variants of the scheduling problem so that we cannot give a brief formal definition. In general, the problem is to schedule n jobs on m machines subject to side constraints like some prespecified partial order on the jobs or a maximal number of jobs per machine at the same time. The objective function can be, for instance, the minimization of the production costs or the minimization of the completion time. There are many applications in production planning.
References: Nemhauser and Savelsbergh (1992), Applegate and Cook (1993)

Sequential ordering problem

Given a digraph $D = (V, A)$ with n nodes and arc weights $c \in \mathbb{R}^A$. Find a minimum length Hamiltonian path $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n) \in A$ such that given precedence constraints of the form "v_i before v_j" are satisfied. An application is robot motion planning, both in production and in storage systems.
Reference: Ascheuer, Escudero, Grötschel and Stoer (1993)

Set partitioning problem

The set partitioning problem is a zero-one optimization problems with side constraints of the form $Ax = \mathbb{1}$, where all coefficients of the matrix A are 0 or 1 and the right hand side $\mathbb{1}$ is a vector of ones. This problem has applications in airline crew scheduling.
Reference: Hoffman and Padberg (1993)

Steiner tree problem

Given a connected graph $G = (V, E)$, $T \subseteq V$, $c \in \mathbb{R}^E$, find $F \subseteq E$ with $c(F)$ minimum such that F induces a tree in G and $T \subseteq \bigcup_{e \in F} e$. This is the network survivability problem with $d_v \in \{0, 1\}$. An application is the design of telephone networks.
Reference: Chopra, Gorres and Rao (1992)

Steiner tree packing problem

For a connected graph $G = (V, E)$, node sets $T_1, T_2, \ldots, T_k \subseteq V$, capacities $w \in \mathbb{N}_0^E$, and costs $c \in \mathbb{R}^E$, find trees $F_1, F_2, \ldots, F_k \subseteq E$ with $T_i \subseteq \bigcup_{e \in F_i} e$ for all $i = 1, \ldots, k$ and $\sum_{i=1}^k |e \cap F_i| \leq w_e$ for all $e \in E$ such that $\sum_{i=1}^k c(F_i)$ is minimum. An application is the routing of nets in electronic circuits.
References: Martin (1992), Grötschel, Martin and Weismantel (1992a, 1992b)

Traveling salesman problem

The traveling salesman problem has been formulated in section 1. Applications are, e.g., the drilling of printed circuit boards, diffractometer control in x-ray crystallography, and vehicle routing.

References: Dantzig, Fulkerson and Johnson (1954), Miliotis (1976, 1978), Grötschel (1977, 1980), Padberg and Hong (1980), Crowder and Padberg (1980), Padberg and Grötschel (1985), Fleischmann (1985), Padberg and Rinaldi (1987, 1989, 1991) Grötschel and Holland (1991) Jünger, Reinelt and Thienel (1992), Clochard and Naddef (1993), Jünger, Reinelt and Rinaldi (1994),

Vehicle routing problem

The vehicle routing problem is a multiple salesman problem in which a demand d_v is associated with every city v and each salesman (vehicle) has to deliver d_v units of some commodity to each city v he visits. The total amount of the commodity that each salesman can carry is limited by the vehicle capacity c. Generalizations include, for example, different capacities for different vehicles, time windows on the delivery time, or upper limits for the length of the trips of the salesmen.
Reference: Cornuéjols and Harche (1993)

Windy postman problem

The windy postman problem is a directed variant of the Chinese postman problem.
References: Zaw Win (1987), Grötschel and Zaw Win (1992)

8 References

R.K. Ahuja, T.L. Magnanti and J.B. Orlin (1993), Network flows: theory, algorithms and applications, Prentice Hall, Eaglewood Cliffs.

D. Applegate and W. Cook (1993), A computational study of the job shop scheduling problem, *ORSA Journal on Computing* 3, 149-156.

D. Applegate, B. Bixby, V. Chvátal and W. Cook (1993), private communication.

N. Ascheuer, L.F. Escudero, M. Grötschel and M. Stoer (1993), A cutting plane approach to the sequential ordering problem (with applications to job scheduling in manufacturing), *SIAM Journal on Optimization* 3, 25–42.

E. Balas, S. Ceria, G. Cornuéjols (1993a), A lift-and-project cutting plane algorithm for mixed 0-1 programs, *Mathematical Programming* 58, 295–324.

E. Balas, S. Ceria, G. Cornuéjols (1993b), Solving mixed 0-1 programs by a lift-and-project method, to appear in *SODA 1993*.

E. Balas and P. Toth (1985), Branch and bound methods, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.), *The traveling salesman problem*, John Wiley & Sons, Chichester, 361–401.

F. Barahona and A. Casari (1988), On the magnetization of the ground states in two-dimensional Ising spin glasses, *Computer Physics Communications* 49, 417–421.

F. Barahona, M. Grötschel, M. Jünger and G. Reinelt (1988), An application of combinatorial optimization to statistical physics and circuit layout design, *Operations Research* 36, 493–513.

F. Barahona, M. Jünger and G. Reinelt (1989), Experiments in quadratic 0-1 programming, *Mathematical Programming* 44, 127–137.

R. Bayer (1972), Symmetric binary b-trees: Data structure and maintenance algorithms, *Acta Informatica* 1, 290–306.

R.E. Bixby, J.W. Gregory, I.J. Lustig, R.E. Marsten and D.F. Shanno (1992), Very large-scale linear programming: a case study in combining interior point and simplex method, *Operations Research* 40, 885-897.

E.A. Boyd (1993a), Generating Fenchel cutting planes for knapsack polyhedra, to appear in *SIAM Journal on Optimization*.

E.A. Boyd (1993b), Fenchel cutting planes for integer programs, to appear in *Operations Research*.

E.A. Boyd (1993c), Solving integer programs with Fenchel cutting planes and preprocessing, in: G. Rinaldi and L. Wolsey (eds.), *Proceedings of the third IPCO conference*, 209–220.

L. Brunetta, M. Conforti, G. Rinaldi (1994), A branch and cut algorithm for the equicut problem, Technical report, Instituto di Analisi dei Sistemi ed Informatica del CNR.

T.L. Cannon (1988), Large-scale zero-one linear programming on distributed workstations, Ph.D. thesis, George Mason University.

T.L. Cannon and K.L. Hoffman (1990), Large-scale zero-one linear programming on distributed workstations, *Annals of Operations Research*, 22, 181–217.

S. Ceria (1993), Lift-and-project methods for mixed 0-1 programs, Ph.D. thesis, Carnegie Mellon University.

S. Chopra, E.R. Gorres and M.R. Rao (1992), Solving the Steiner tree problem on a graph using branch and cut, *ORSA Journal on Computing* 3, 149–156.

V. Chvátal (1983), *Linear programming*, W.H. Freeman and Company, New York.

J.M. Clochard and D. Naddef (1993), Using path inequalities in a branch and cut code for the symmetric traveling salesman problem, in: G. Rinaldi and L. Wolsey (eds.), *Proceedings of the Third IPCO Conference*, 291–311.

T. Christof, M. Jünger and G. Reinelt (1991), A complete description of the traveling salesman problem polytope on 8 nodes, *Operations Research Letters* 10, 497–500.

G. Clarke and J.W. Wright (1964), Scheduling of vehicles from a central depot to a number of delivery points, *Operations Research* 12, 568–581.

A. Corberán and J.M. Sanchis (1991), A polyhedral approach to the rural postman problem, Working paper, Facultad de Matemáticas, Universidad de Valencia.

T.H. Cormen, C.E. Leiserson and R.L. Rivest (1990), *Introduction to algorithms*, MIT Press, Cambridge.

G. Cornuéjols and F. Harche (1993), Polyhedral study of the capacitated vehicle routung problem, *Mathematical Programming* 60, 21–52.

CPLEX (1993), Using the CPLEX callable library and CPLEX mixed integer library, CPLEX Optimization, Inc.

H. Crowder, E.L. Johnson and M.W. Padberg (1983), Solving large-scale zero-one linear programming problems, *Operations Research* 31, 803–834.

H. Crowder and M.W. Padberg (1980), Solving large-scale symmetric traveling salesman problems to optimality, *Management Science* 26, 495–509.

R.J. Dakin (1965), A tree search algorithm for mixed integer programming problems, *Computer Journal* 8, 250–255.

G.B. Dantzig, D.R. Fulkerson and S.M. Johnson (1954), Solution of a large-scale traveling salesman problem, *Operations Research* 2, 393–410.

C. De Simone and G. Rinaldi (1992), A cutting plane algorithm for the max-cut problem, Report 246, Instituto di Analisi dei Sistemi ed Informatica del CNR.

R. Euler and H. Le Verge (1992), A complete and irredundant linear description of the asymmetric traveling salesman polytope on 6 nodes, Technical report No. 684, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes, France.

M. Fischetti, J.J.S. González and P. Toth (1994), A branch and cut algorithm for the symmetric generalized traveling salesman problem, Technical report, University of Bologna.

B. Fleischman (1985), A cutting plane procedure for the traveling salesman problem on road networks, *European Journal of Operational Research* 21, 307–317.

J.-M. Gauthier and G. Ribière (1977), Experiments in mixed-integer linear programming using pseudo costs, *Mathematical Programming* 12, 26–47.

R.E. Gomory (1958), Outline of an algorithm for integer solutions to linear programs, *Bulletin of the American Mathematical Society* 64, 275–278.

R.E. Gomory (1960), Solving linear programming problems in integers, *Proceedings of the Symposium on Applied Mathematics* 10, 211–215.

R.E. Gomory (1963), An algorithm for integer solutions to linear programs, in: R.L. Graves and P. Wolfe (eds.), *Recent Advances in Mathematical Programming*, McGraw Hill, New York, 269–302.

R.E. Gomory and T.C. Hu (1961), Multi-terminal network flows, *SIAM Journal* 9, 551–570.

A.Y. Grama and V. Kumar (1992), Parallel processing of discrete optimization problems: A survey, Technical report, University of Minnesota.

M. Grötschel (1977), Polyedrische Charakterisierungen kombinatorischer Optimierungsprobleme, Hain, Meisenheim am Glan.

M. Grötschel (1980), On the symmetric traveling salesman problem: solution of a 120-city problem, *Mathematical Programming Studies* 12, 61–77.

M. Grötschel and O. Holland (1985), Solving matching problems with linear programming, *Mathematical Programming* 33, 243–259.

M. Grötschel and O. Holland (1987), A cutting plane algorithm for minimum perfect 2-matching, *Computing* 39, 327–344.

M. Grötschel and O. Holland (1991), Solution of large-scale symmetric traveling salesman problems, *Mathematical Programming* 51, 141–202.

M. Grötschel, M. Jünger and G. Reinelt (1984a), Optimal triangulation of large real world input output-matrices, *Statistische Hefte* 25, 261–295.

M. Grötschel, M. Jünger and G. Reinelt (1984b), A cutting plane algorithm for the linear ordering problem, *Operations Research* 32, 1195–1220.

M. Grötschel, M. Jünger and G. Reinelt (1987), Calculating exact ground states of spin glasses: a polyhedral approach, in: J.L. van Hemmen and I. Morgenstern (eds.), *Proceedings of the Heidelberg Colloquium on Glassy Dynamics*, Lecture Notes in Physics 275, Springer, Heidelberg, 325–353.

M. Grötschel, A. Martin and R. Weismantel (1992a), Packing Steiner trees: A cutting plane algorithm and computational results, Report No. SC 92-9, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

M. Grötschel, A. Martin and R. Weismantel (1992b), Routing in grid graphs by cutting planes, Report No. SC 92-26, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

M. Grötschel, C.L. Monma and M. Stoer (1992a), Polyhedral and computational investigations for designing communication networks with high survivability requirements, Report No. SC 92-24, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

M. Grötschel, C.L. Monma and M. Stoer (1992b), Computational results with a cutting plane algorithm for designing communication networks with low-connectivity constraints, *Operations Research*, 40, 309–330.

M. Grötschel, C.L. Monma and M. Stoer (1994), Design of Survivable Networks, to appear in M. Ball, T. Magnanti, C.L. Monma and G.L. Nemhauser (eds.), *Handbook on Operations Research and Management Sciences: Networks*, North Holland .

M. Grötschel and M.W. Padberg (1985), Polyhedral theory, in E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.): *The traveling salesman problem*, Wiley & Sons, Chichester.

M. Grötschel and Y. Wakabayashi (1989), A cutting plane algorithm for a clustering problem, *Mathematical Programming B* 45, 59–96.

M. Grötschel and Zaw Win (1992), A cutting plane algorithm for the windy postman problem, *Mathematical Programming* 55, 339–358.

L.J. Guibas and R. Sedgewick (1978), A diochromatic framework for balanced trees, in: *Proceedings of the 19th annual symposium on foundations of computer science*, IEEE Computer Society, 8–21.

D. Gusfield (1990), Very simple methods for all pairs network flow analysis, *SIAM Journal on Computing*, 19, 143–155.

J. Hao and J.B. Orlin (1992), A faster algorithm for finding the minimum cut in a graph, *Proceedings of the third annual ACM-SIAM symposium on discrete algorithms*, 165–174.

K. Hoffman and M.W. Padberg (1991), Improving LP-representations of zero-one linear programs for branch and cut, *ORSA Journal on Computing* 3, 121–134.

K. Hoffman and M.W. Padberg (1993), Solving airline crew scheduling problems by branch and cut, *Management Science* 39, 657–682.

S. Holm and M.M. Sørensen (1993), The optimal graph partitioning problem, *OR Spektrum* 15, 1–8.

IBM (1978), Mathematical programming system extended/370 (MPSX/370), Program reference manual, IBM Corporation.

IBM (1979), Mathematical programming system extended/370 (MPSX/370), Mixed integer programming/370 (MIP/370), Program reference manual, IBM Corporation.

IBM (1991), Optimization Subroutine Library - Guide and Reference (Release 2) Third Edition, IBM Corporation.

M. Jünger and P. Mutzel (1993a), Solving the maximum weight planar subgraph problem by branch and cut, in: G. Rinaldi and L. Wolsey (eds.),*Proceedings of the third IPCO conference*, 479–492.

M. Jünger and P. Mutzel (1993b), Maximum planar subgraphs and nice embeddings: Practical layout tools, Report No. 93.145, Angewandte Mathematik und Informatik, Universität zu Köln.

M. Jünger, G. Reinelt and G. Rinaldi (1994), The traveling salesman problem, Report No. 92.113, Angewandte Mathematik und Informatik, Universität zu Köln, to appear in M. Ball, T. Magnanti, C.L. Monma and G.L. Nemhauser (eds.), *Handbook on Operations Research and Management Sciences: Networks*, North Holland.

M. Jünger, G. Reinelt and S. Thienel (1992), Provably good solutions for the traveling salesman problem, Report No. 92.114, Angewandte Mathematik und Informatik, Universität zu Köln.

D.V. Karger (1993), Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm, Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms, 21–30.

D.V. Karger and C. Stein (1993), An $\tilde{O}(n^2)$ algorithm for minimum cuts, Proceedings of the 25th ACM Symposium on the Theory of Computing, San Diego, CA, 757–765.

R.M. Karp and C.H. Papadimitriou (1982), On linear characterizations of combinatorial optimization problems, *SIAM Journal on Computing* 11, 620–632.

A.H. Land and A.G. Doig (1960), An automatic method for solving discrete programming problems, *Econometrica* 28, 493–520.

G. Laporte and Y. Nobert (1980), A cutting plane algorithm for the m-salesmen problem, J. Ppl. Res. Soc. 31, 1017–1023.

I. Lustig, R.E. Marsten and D.F. Shanno (1992a), Computational experience with a globally convergent primal-dual predictor-corrector algorithm for linear programming, Report SOR 92-10, Program in Statistics and Operations Research, Department of Civil Engineering and Operations Research, Princeton University.

I. Lustig, R.E. Marsten and D.F. Shanno (1992b), Interior point methods for linear programming: computational state of the art, Report SOR 92-17, Program in Statistics and Operations Research, Department of Civil Engineering and Operations Research, Princeton University.

A. Martin (1992), Packen von Steinerbäumen: Polyedrische Studien und Anwendung, Report TR 92-4, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

P. Miliotis (1976), Integer programming approaches to the traveling salesman problem, Mathematical Programming 10, 367–378.

P. Miliotis (1978), Using cutting planes to solve the symmetric traveling salesman problem, Mathematical Programming 15, 177–188.

P. Miliotis, G. Laporte and Y. Nobert (1981), Computational comparison of two methods for finding the shortest complete cycle or circuit in a graph, R.A.I.R.O. Recherche opérationelle/Operations Research 15, 233–239.

J.E. Mitchell and B. Borchers (1992), A primal-dual interior point cutting plane method for the linear ordering problem, Report No. 204, Rensselaer Polytechnic Institute.

J.E. Mitchell and B. Borchers (1993), Solving real world linear ordering problems using a primal dual interior point cutting plane method, Report No. 207, Rensselaer Polytechnic Institute.

J.E. Mitchell and M.J. Todd (1992), Solving combinatorial optimization problems using Karmarkar's algorithm, *Mathematical Programming* 56, 245–284.

H. Nagamochi and T. Ibaraki (1992), A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph, *Algorithmica* 7, 583–596.

H. Nagamochi and T. Ibaraki (1992), Computing edge-connectivity in multigraphs and capacitated graphs, *SIAM Journal on Discrete Mathematics* 5, 54–66.

G.L. Nemhauser and G. Sigismondi (1992), A strong cutting plane/branch and bound algorithm for node packing, *J. Opl. Res. Soc.*, 25, 443–457.

G.L. Nemhauser, M.W.P. Savelsbergh (1992), A cutting plane algorithm for the single machine scheduling problem with release times, in M. Akgul, H. Hamacher, S. Tufecki (eds.), *Combinatorial Optimization: New Frontiers in the Theory and Practice*, NATO ASI Series F: Computer and Systems Sciences 82, Springer-Verlag, 63-84.

G.L. Nemhauser and L.A. Wolsey (1988), Integer and combinatorial optimization, John Wiley & Sons, New York.

M.W. Padberg and M. Grötschel (1985), Polyhedral computations, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, 307–360.

M.W. Padberg and S. Hong (1980), On the symmetric traveling salesman problem: a computational study, *Mathematical Programming Studies* 12, 78–107.

M.W. Padberg and M.R. Rao (1982), Odd minimum cut sets and b-matchings, *Mathematics of Operations Research* 7, 67–80.

M.W. Padberg and G. Rinaldi (1987), Optimization of a 532 city symmetric traveling salesman problem by branch and cut, *Operations Research Letters* 6, 1–7.

M.W. Padberg and G. Rinaldi (1989), A branch and cut approach to a traveling salesman problem with side constraints, *Management Science* 35, 1393–1412.

M.W. Padberg and G. Rinaldi (1990a), An efficient algorithm for the minimum capacity cut problem, *Mathematical Programming* 47, 19–36.

M.W. Padberg and G. Rinaldi (1990b), Facet identification for the symmetric traveling salesman polytope, *Mathematical Programming* 47, 219–257.

M.W. Padberg and G. Rinaldi (1991), A branch and cut algorithm for the resolution of large-scale symmetric traveling salesman problems, *SIAM Review* 33, 60–100.

W.R. Pulleyblank (1989), Polyhedral combinatorics, in G.L. Nemhauser, A.H.G. Rinnoy Kan and M.J. Todd (eds.), *Handbook on Operations Research and Management Sciences: Networks*, North Holland.

G. Reinelt (1985), *The linear ordering problem: algorithms and applications*, Heldermann, Berlin.

G. Reinelt (1991a), TSPLIB – A traveling salesman problem library, *ORSA Journal on Computing* 3, 376–384.

G. Reinelt (1991b), TSPLIB – Version 1.2, Report No. 330, Schwerpunktprogramm der Deutschen Forschungsgemeinschaft, Universität Augsburg.

G. Reinelt (1992), Fast heuristics for large geometric traveling salesman problems, *ORSA Journal on Computing*, 4, 206–217.

G. Reinelt (1993), A note on small linear-ordering polytopes, *Discrete & Computational Geometry* 10, 67–78.

M.W.P. Savelsbergh (1993), A branch-and-price algorithm for the generalized assignment problem, Report COC-93-02, Georgia Institute of Technology.

M.W.P. Savelsbergh, G.S. Sigismondi and G.L. Nemhauser (1994), MINTO, a Mixed INTeger Optimizer, *Operations Research Letters*, to appear.

A. Schrijver (1986), *Theory of linear and integer programming*, John Wiley & Sons, Chichester.

M. Stoer (1992), *Design of survivable networks*, Lecture Notes in Mathematics 1531, Springer, Heidelberg.

R.J. Vanderbei (1992), LOQO user's manual, Technical report, Program in Statistics & Operations Research, Princeton University.

T.J. Van Roy and L.A. Wolsey (1987), Solving mixed integer programming problems using automatic reformulation, *Operations Research*, 35, 45–57.

Y. Wakabayashi (1986), Aggregation of binary relations: Algorithmic and polyhedral investigations, Doctoral thesis, Universität Augsburg.

R. Weismantel (1992), Plazieren von Zellen: Theorie und Lösung eines quadratischen 0/1-Optimierungsproblems, Report TR 92-3, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

Zaw Win (1987), Contributions to routing problems, Doctoral thesis, Universität Augsburg.