The ABACUS System for Branch-and-Cut-and-Price Algorithms in Integer Programming and Combinatorial Optimization

MICHAEL JÜNGER
Institut für Informatik, Universität zu Köln, Pohligstraße 1, D-50969 Köln, Germany

STEFAN THIENEL
Software Design & Management GmbH & Co. KG, Thomas-Dehler-Straße 27, D-81737 München, Germany

SUMMARY

The development of new mathematical theory and its application in software systems for the solution of hard optimization problems have a long tradition in mathematical programming. In this tradition we implemented ABACUS, an object-oriented software framework for branch-and-cut-and-price algorithms for the solution of mixed integer and combinatorial optimization problems. This paper discusses some difficulties in the implementation of branch-and-cut-and-price algorithms for combinatorial optimization problems and shows how they are managed by ABACUS.

KEY WORDS Combinatorial Optimization Branch & Cut Branch & Price Software Frameworks Object Oriented Design

HISTORY AND MOTIVATION

Mathematical Programming, originating as a scientific discipline in the late forties, is concerned with the search for optimal decisions under restrictions. The most prominent mathematical models include linear programming, (mixed) integer linear programming and combinatorial optimization with linear objective function. A surprisingly large variety of problems in economics, mathematics, computer science, operations research and the natural sciences can be captured by these models and effectively solved by appropriate software. Years before computer science emerged as a scientific discipline, the protagonists of mathematical programming, who were mainly applied mathematicians or mathematically oriented economists, aimed at the development of algorithms that could solve the problem instances, in the beginning primarily economic and military planning applications, by hand calculations and, when electronic computers became available in the early fifties, via computer software.

Linear Programming

The father of linear programming is George B. Dantzig, who proposed the model

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \end{array} \tag{1}$$

CCC 0038–0644/00/000000–01

Received 26 February 1998
Revised 1 April 1999

where $c \in R^n, A \in I\!R^{m \times n}, b \in I\!R^m$, and invented the celebrated simplex algorithm for its solution. In the first sentence of his textbook "Linear Programming and Extensions" [1] he states: "The final test of a theory is its capacity to solve the problems which originated it" and before he proceeds to the acknowledgments, he writes: "The viewpoint of this work is constructive. It reflects the beginning of a theory sufficiently powerful to cope with some of the challenging decision problems upon which it was founded." Not only the model and the simplex algorithm have pertained to this day, but also the philosophy he summarizes in these two sentences remained a leitmotif for mathematical programmers until today, as Michel Balinski puts it in [2]: "First the real problems, then the theory ...", and we can safely add: "... and then implementing and testing it on the real problems." The simplex algorithm was implemented in the US National Bureau of Standards on the Standards Eastern Automatic Computer (SEAC) as early as 1952, and computational testing and comparison with competing methods was performed in a way that meets today's standards for such experiments. Slightly later, around 1954, William Orchard-Hays of the RAND Corporation designed the first commercial implementations of the (revised) simplex algorithm for the IBM CPC, IBM 701 and IBM 704 machines. Also direct commercial applications in the oil industry were started as early as 1952 by A. Charnes, W.W. Cooper and B. Mellon [3].

Integer Programming

It became soon clear, though, that in real problems integrality of some of the decision variables is required. If an optimum plan of activities requires using 1.3 airplanes, this makes no sense. Also, very often yes/no decisions are desired that can be modeled using binary variables. So the (mixed) integer (binary) linear programming model is the linear programming model in which (some) variables are required to take integral (binary) values. In 1958, Ralph Gomory [4] presented an elegant algorithmic solution to the integer linear programming problem that was later refined to the mixed integer case. Interestingly, he not only invented the "cutting plane" method and proved its finiteness and correctness, but he also implemented it in the same year on an E 101 and an IBM 704 computer in the then brand-new FORTRAN language, in order to see how it performs. Unfortunately, the early experiments where not very satisfactory in terms as practical efficiency, and Gomory's cutting plane method, even though appreciated for its theoretical beauty, was not recommended for practical computations until only a few years ago Balas, Ceria, Cornuéjols, and Natraj [5] gave computational evidence that it should be reconsidered. Instead, the branch-and-bound method proposed by A.H. Land and A.G. Doig in 1960 [6] became the method of choice in all commercial computer codes that were provided by many computer companies starting with linear programming in the sixties and mixed integer programming in the seventies.

Combinatorial Optimization

In combinatorial optimization with a linear objective function, the task consists of selecting from a family of subsets $\mathcal{F} \subseteq 2^E$ of a finite ground set E one $F \in \mathcal{F}$ that maximizes (minimizes) a linear objective function $\sum_{e \in F} c_e$ for coefficients $c_e \in I\!R, e \in E$. Mathematically, this is trivial, because an optimizing set F can be found by finite enumeration, however such a strategy is clearly unsatisfiable for practical computation. Some of the finest algorithms in computer science and mathematical programming that have been formulated for problems like the minimum spanning tree problem, the shortest path problem, or the matching problem by Kruskal [7], Dijkstra [8] and Edmonds [9], respectively, fit into this framework, as well as many others in combinatorics and (hyper-) graph theory.

In our examples, the finite ground set E consist of the edges of a graph G and the feasible solutions are the spanning trees, the paths connecting two specified nodes and the matchings in G, respectively.

Branch-and-Cut

Unlike these examples, most examples with practical applications have turned out to be NP-hard. Nevertheless, an obvious connection to binary integer programming leads to an algorithmic methodology that has been found out to be able to solve real world instances to optimality anyway. Namely, passing from the feasible subsets to their characteristic vectors, it is usually not hard to define the problem as a binary linear programming problem, whereas it is usually a long way, theoretically and in terms of implementation effort, to make the well developed linear programming techniques exploitable in effective computation.

We demonstrate this on a prominent example, the traveling salesman problem, in which the task is to find Hamiltonian cycle ("tour") with minimum total edge weight in a complete undirected graph. Incidentally, this is the problem for which now commonly used optimization techniques were first outlined by G.B. Dantzig, D.R. Fulkerson and S.M. Johnson in 1954 [10]. If x_{ij} is a variable corresponding to the edge connecting nodes i and j, and c_{ij} is the weight (length) of this edge, then

$$\begin{array}{lll} \text{minimize} & \sum_{i,j\in E} c_{ij}x_{ij} & \\ \text{subject to} & \sum_{j\in V} x_{ij} = 2 & \text{for all } i \in V \\ & \sum_{i\in S, j\notin S} x_{ij} \geq 2 & \text{for all } S \subset V,\ 3 \leq |S| \leq \left\lfloor \frac{|V|}{2} \right\rfloor \\ & 0 \leq x_{ij} \leq 1 & \text{for all } ij \in E \\ & x_{ij} \in \{0,1\} & \text{for all } ij \in E \end{array} \tag{2}$$

is an integer programming formulation in which the equations make sure that in the solution every node has degree two and the inequalities, called *subtour elimination constraints*, guarantee that we do not obtain a collection of subtours. The variables with value 1 in an optimum solution of this integer program are in one-to-one correspondence with the edges of an optimum tour. Discarding the integrality conditions, we obtain a linear programming relaxation (subtour relaxation) that can be used in a branch-and-bound environment: A simple scheme would solve the relaxation at every subproblem in which some variables have been set to 0 or 1. If the solution is not integral, two new subproblems, with one non-integral variable set to 1 in one and to 0 in the other, are created in a branching step. However, this linear programming relaxation contains an exponential number of constraints ($\Theta(2^n)$) for an instance on n nodes. This leads to the idea to solve the relaxation with a small subset of all constraints, check if the optimum solution violates any other of the constraints, and if so, append them to the relaxation. Thus we get a cutting plane method at each node of the enumeration tree that arises by branching. This method is called *branch-and-cut*. It was first formulated for and successfully applied to the linear ordering problem in [11].

There are many ways to strengthen the subtour relaxation by further classes of inequalities. Identifying some of them that violate a given fractional solution is called the *separation problem*. This problem can be solved in polynomial time for the subtour elimination constraints. Of course, it would be desirable to compute a complete description of the convex hull of the incidence vector of tours. It exists by classical theorems of Minkowski and Weyl. However, results of Karp and Papadimitriou [12] make it unlikely that we can compute it except for very small n. The number of inequalities needed is enormous, e.g., 42,104,442 for $n = 9$ and at least 51,043,900,866 for $n = 10$ [13].

The field of polyhedral combinatorics deals with identifying subsystems of such complete and irredundant linear descriptions. In order to make a subclass computationally exploitable, we must devise according separation algorithms, or at least useful heuristics, if the problem is proven to be or appears to be hard.

Computer implementations for branch-and-cut algorithms for the traveling salesman problem have been devised by a few research groups, the first, in which the term "branch-and-cut" was used for the first time, by Padberg and Rinaldi [14]. Such programs solve most instances with a few hundred cities routinely and also some instances with a few thousand cities, see [15] for a recent survey of the theory and practice of this method for the traveling salesman problem, [16] for an account how the same methodology can be applied to other combinatorial optimization problems, and [17] for a very recent annotated bibliography on branch-and-cut algorithms. The first essential ingredient that makes such an approach work consists of a theoretical part that requires creative analytical investigations on identifying appropriate classes of inequalities (polyhedral combinatorics) and the design of separation algorithms. Together with the implementation of the separation algorithms this is highly problem specific. The second ingredient is integrating the separation software into a cutting plane framework and this into an enumerative frame. This latter part is much less problem specific but requires a tremendous implementation effort that can be reduced by an appropriate software system.

Branch-and-Price

Before we describe such a system in the subsequent sections, we would like to present another example in order to explain a "dual" method to "branch-and-cut", namely *branch-and-price*: the binary cutting stock problem. Here, a set of n rolls of lengths $a_1, a_2, \ldots a_n$ has to be cut out of base rolls with length L. The problem is to determine a cutting strategy that minimizes the number of base rolls used. In 1961, P.C. Gilmore and R.E. Gomory [18] proposed the following model: The vector $b \in \{0,1\}^n$ represents a cutting pattern for a base roll if $\sum_{i=1}^{n} a_i b_i \leq L$. Let the columns of the matrix $B \in \{0,1\}^{n \times m}$ represent all possible cutting patterns. Then the problem can be modeled as the following binary linear programming problem:

$$\begin{array}{lll} \text{minimize} & \sum_{j=1}^{m} z_j & \\ \text{subject to} & \sum_{j=1}^{m} b_{ij} z_j = 1 & \text{for all } i \in \{1,2,\ldots,n\} \\ & 0 \leq z_j \leq 1 & \text{for all } j \in \{1,2,\ldots,m\} \\ & z_j \in \{0,1\} & \text{for all } j \in \{1,2,\ldots,m\} \end{array} \tag{3}$$

In any feasible solution $z_j = 1$ if and only if the j-th cutting pattern is used. In contrast to our previous example in which we had a polynomial numbers of variables and an exponential number of constraints, here we have the opposite situation. If we solve the problem on a small subset of the columns, we have to make sure that the missing columns can be safely assumed to be associated with 0 components of the optimum vector z. The simplex algorithm does not only give us a basic feasible solution (geometrically this corresponds to a vertex of the polyhedron defined by the restrictions) but also a short certificate for optimality consisting of a quantity y_i for each row i such that $\sum_{i=1}^{n} b_{ij} y_i \leq 1$ for all cutting patterns $B_{\cdot j} = (b_{1j}, b_{2j}, \ldots, b_{nj})^T$ that are present in the chosen subset. In order to determine if the same relation holds for all missing cutting patterns as well we can solve the knapsack problem

$$\begin{array}{lll} \text{maximize} & \sum_{i=1}^{n} & y_i b_i \\ \text{subject to} & \sum_{i=1}^{n} & a_i b_i \leq L \\ & & b_i \in \{0,1\} \end{array} \tag{4}$$

for which effective pseudo-polynomial algorithms exist. If the maximum is at most 1, our solution is optimal for the complete problem, otherwise the optimum pattern $b_1, b_2, \ldots, b_n$ found by the algorithm is appended as a new column to the formulation. This process of evaluating the missing columns is called "pricing" in linear programming terminology, and embedding the method into an enumerative frame leads to a *branch-and-price* algorithm. Cutting and pricing can be combined (and they are in all published state-of-the-art algorithms for the optimum solution of the traveling salesman problem) and this methodology is called *branch-and-cut-price*.

Software Systems

Software systems have aided the design of branch-and-cut-price algorithms to various degrees since they became popular in the early eighties. Linear programming and/or branch-and-bound algorithms implemented in systems such as MPSX/MIP [19], MINOS [20], LINDO [21], XMP [22], OSL [23], Cplex [24], XPRESS [25], MOPS [26] and various others (the complete list is so long that we cannot include all systems we know here) have been used by various authors in their specific implementations and experiments. One system that was developed slightly before ours, the MINTO system [27], most comfortably provides the basic tools used in branch-and-cut-price algorithms as well as interfaces to various software packages for linear programming.

The ever growing knowledge about algorithmic techniques calls for much more convenient systems if we want to maintain the good tradition in mathematical programming that new methods should be demonstrated to actually solve the problems that led to their development, in the spirit of George B. Dantzig, Ralph E. Gomory and other pioneers in mathematical programming.

We observed that the systems mentioned above are strong for the solution of general integer optimization problems but are neither convenient nor efficient enough for the solution of integer programs coming from hard combinatorial optimization problems. This was the "algorithmic motivation" for the development of ABACUS—A Branch-And-CUt System, a software framework for the implementation of branch-and-cut-and-price algorithms.

We also recognized that the realization of such a system in a traditional programming language, as used by the other systems, is difficult in its implementation, and—even worse—inconvenient and unsafe in its application. The "software motivation" for our work on ABACUS was that many of these problems evaporate if object oriented design principles and programming languages are used.

Instead of the classical *waterfall model* (see e.g., [28]) with its rather strict separation of the phases during the life cycle of a software system, in object oriented programming an *iterative and incremental life cycle* is more common [29, 30]. Typical for the development of object-oriented frameworks is that during the design new abstractions are discovered. Moreover, the object-oriented design usually simplifies the implementation of a new abstraction. We applied this development technique and observed that it is useful for the generation of an open-ended flexible system.

The design of the ABACUS software framework and its implementation in C++ is extensively discussed in [31] and [32], its application is explained in [33], [34], and [35]. In this article, we briefly sketch the design criteria and the basic design ideas, and then we point out some features of combinatorial optimization problems that cannot be handled by any of the previous software systems satisfactorily, and show how they are handled by ABACUS.

DESIGN CRITERIA

Already the design of a medium sized computer program is a nontrivial task. One of the difficulties is the presence of numerous concurrent alternatives in the design. Therefore, we require some rather general design principles, which we can follow during the concrete design of the classes (cf. [30, 29, 36]).

It is obvious that some of the following criteria are rather contradictory and cannot be satisfied at the same time. Therefore, we point out which one of two competing criteria had higher priority for the design of ABACUS.

Efficiency

ABACUS would be a redundant and boring tool if it were significantly slower than a problem specific implementation from scratch. Hence, efficiency was a primary design criterion.

However, this does not mean that we care about the optimal implementation of all functions and data structures. Usually, there is a small number of functions whose running time dominates all other parts of the code in linear programming based branch-and-bound algorithms. Therefore, it is important to provide practically efficient implementations of these parts. According to our experience this is the solution of the linear programs, the solution of the separation and/or pricing problem, and the heuristics. Luckily, from the point of view of the framework, the three latter parts are completely problem specific and the time for the solution of the linear programs is spent in another software package as long as the interfaces to the LP-solver are well designed.

Nevertheless, we provide practically efficient implementations of all those parts requiring the majority of the additional time, i.e., the CPU time minus the running time used for LP-solution, pricing, separation, and heuristics. We do not care about very fast implementations of functions requiring, e.g., 0.01% of the total CPU time, although we have to admit that it was hard to resist the fun of implementing, for instance, an efficient balanced tree instead of using a linear list.

Flexibility

ABACUS is a framework that is applicable to a large class of optimization problems including general mixed integer optimization problems and combinatorial optimization problems. Our first experience in using ABACUS for the solution of the traveling salesman problem, the general mixed integer optimization problem, the binary cutting stock problem [31], and user experience that we cite at the end of this article indicates that we reached this design criterion.

Easy to Use

Johnson and Foote [37] distinguish between the concepts of *white-box frameworks* and *black-box frameworks*. The implementation of a white-box framework has to be understood in order to use it, whereas in a black-box framework only the interfaces of the components have to be understood. In general, white-box frameworks are more flexible but more difficult to use. Most components of ABACUS are designed as a black-box framework. For technical reasons some member functions of classes derived in an application require some "white-box" features, but the user's guide [38] and the example [39] should provide the guidelines for a correct implementation.

The object-oriented design, in which a user has to derive her/his own application classes from a small number of base classes, provides rather simple interfaces, because the interfaces of the application functions that have to be implemented are well defined as (pure) virtual functions. Naturally, this concept is slightly more complex than a framework in which the user interfaces are already predefined functions, as e.g. in MINTO. However, this methodology provides a practically more efficient and flexible framework. We have given the same priority to the design goals flexibility for a large class of problems and simplicity of the interfaces for a user right after the efficiency of the system.

A good documentation, consistent use of the design criteria in the whole project, consistent implementation of different classes (e.g., naming conventions, order of presentation, etc.) are prerequisites

of an easily usable software system. However, the power of ABACUS results from its object-oriented design. Using this software system requires practical, i.e., implementational, experience with C++ and, in particular, with those features that are not already covered by C.

Extendibility

ABACUS is a system that should grow with the requirements of new applications. Future modifications and extensions are hard to predict. The majority of necessary modifications will be induced by the applications of ABACUS. We expect that the object-oriented design of the software will give us a very high flexibility. ABACUS is a collection of classes. A class has a public interface and a private implementation, which are strictly separated. The private part of a class can be changed without affecting the interfaces. Moreover, only changes of the protected and public parts of the few classes that are directly used by an application may affect the user.

Portability

ABACUS should be useful for many different applications by a big community of users. This implies that it cannot be restricted to a specific hard- or software environment. The implementation of ABACUS according to the latest working paper of the C++ standardization committee [40] and almost no use of operating system specific features provide a high portability. Portability means also that the design has to be independent of specific third-party software, in particular, the LP-solver.

One of the basic ideas of object-oriented programming is to simplify and encourage the reuse of existing code. This is also the spirit in which ABACUS is designed. Therefore, the justified question might come up why ABACUS does not make use of any existing library. Useful libraries may be libraries of data structures like LEDA [41], NIHCL [42], or STL [43]. Indeed, a subset of the basic data structures of ABACUS is also available in these libraries. However, we decided not to use any of these data structure libraries. Compared to the complete ABACUS system the parts that could be substituted by one of the mentioned libraries are rather small. We also wanted to specialize these data structures for our applications. Of course, this would also have been possible by inheritance, but concepts like coupling all objects of an instance of application with a global object would have been more awkward to implement. Finally, coupling ABACUS with such a library would certainly not have increased its portability.

Nevertheless, it should be no problem to use other libraries together with ABACUS. Using the graph data structures and algorithms of other software libraries for the implementation of a separation, pricing, or heuristic subroutine is certainly recommendable.

Basic Design Ideas

From a user's point of view, who wants to implement a linear programming based branch-and-bound algorithm, ABACUS provides a small system of base classes from which the application specific classes can be derived. All problem independent parts are "invisible" for the user such that she/he can concentrate on the problem specific algorithms and data structures.

The basic components are pure virtual functions, virtual functions, and virtual dummy functions. A pure virtual function has to be implemented in a class derived by the user of the framework, e.g., the initialization of the branch-and-bound tree with a subproblem associated with the application. In virtual functions we provide default implementations, which are often useful for a big number of applications, but can be redefined if required, e.g., the branching strategy. Finally, we use virtual dummy functions that are virtual functions that do nothing in their default implementations, but can be redefined in

derived classes, e.g., the separation of cutting planes. They are not a pure virtual functions as their definition is not required for the correctness of the algorithm.

Moreover, an application based on ABACUS can be refined step by step. Only the derivation of a few new classes and the definition of some pure virtual functions are required to get a branch-and-bound algorithm running. Then, this branch-and-bound algorithm can be enhanced by the dynamic generation of constraints and/or variables, heuristics, or the implementation of new branching or enumeration strategies.

Default strategies are available for numerous parts of the branch-and-bound algorithm, and they can be controlled via a parameter file. If none of the system strategies meets the requirements of the application, the default strategy can simply be replaced by the redefinition of a virtual function in a derived class.

COMBINATORIAL OPTIMIZATION PROBLEMS

Very prominent examples of combinatorial optimization problems, e.g., most vehicle routing problems, scheduling problems, or cutting problems, turn out to be $\mathcal{NP}$-hard. However, practical experiments show that with branch-and-cut-and-price methods optimum solutions for instances up to a certain problem specific size can be found in reasonable running time. The most successful implementations exploit the combinatorial structure of these optimization problems both in the representation of the problem and in the applied algorithmic techniques.

Problem Formulations

The set of feasible solutions of a combinatorial optimization problem is usually very well structured. Typical examples are problems that are defined as graph optimization problems. In most of these cases, the set of feasible solutions is given by a set of well defined subgraphs, e.g., Hamiltonian cycles for the traveling salesman problem, spanning tournaments for the linear ordering problem, various kinds of cuts for the family of cut problems (e.g., max-cut problem, equi-cut problem), planar subgraphs for the maximum planar subgraph problem, or certain cliques for the quadratic assignment problem (in the formulation of [44]). As an example for a problem that is not defined as a graph optimization problem, we observe that the feasible solutions of the cutting stock problem are certain collections of cutting patterns, which in turn are defined as feasible solutions of knapsack problems.

Such optimization problems can be described in two equivalent ways, either as an integer program (see, e.g., (2) and (3), or in a combinatorial way. Both representations have their advantages and disadvantages. The integer programming formulation is the one we need to apply branch-and-cut-price methods, but the real structure of the problem is hard to recognize if the problem is represented only by a matrix. This structural information is only present in a combinatorial formulation, that on the other hand is inadequate for applying integer programming techniques.

This loss of structural information in the integer programming formulation could be accepted if

- the transformation from the combinatorial formulation could be performed explicitly, and
- the transformed problem could be solved with standard methods or even standard software.

Unfortunately, both criteria are not satisfied by many combinatorial optimization problems.

Integer Programming Formulations of Exponential Size

The number of constraints or variables of the integer programming formulation of a combinatorial optimization problem can be exponential in the size of the combinatorial formulation of the same problem. The number of subtour elimination constraints in the integer programming formulation (2) of the traveling salesman problem on n cities is $\Theta(2^n)$. Even for small sized problem instances, the explicit generation of the constraint matrix is impossible. Therefore, a cutting plane approach is required to handle the large number of constraints as explained above.

On the other hand, the integer programming formulation of the cutting stock problem (3) has an exponential number of variables. If we want to determine its optimum solution by a branch-and-price algorithm, we require information about the combinatorial structure of the problem for the solution of the knapsack problems in the dynamic column generation.

It should be mentioned that there can be different integer programming formulations for the same combinatorial optimization problem. Moreover, there can be both formulations of polynomial size and of exponential size in the input data. While from a theoretical point of view the polynomially sized formulation is preferable, from a practical computational point of view it can turn out that the exponentially sized formulation is more suitable for the problem solution.

One such example is the cutting stock problem. The cutting stock problem can also be solved with the following integer program, where m is an upper bound on the numbers of required base rolls ($m \leq n$).

$$
\begin{array}{lll}
\text{minimize} & \sum_{j=1}^{m} y_j & \\
\text{subject to} & \sum_{i=1}^{n} a_i x_{ij} \quad \leq L y_j & \text{for all } j = 1, 2, \ldots, m \\
& \sum_{j=1}^{m} x_{ij} \quad = 1 & \text{for all } i = 1, 2, \ldots, n \\
& 0 \leq x_{ij} \leq 1 & \text{for all } i = 1, 2, \ldots, n,\ j = 1, 2, \ldots, m \\
& 0 \leq y_j \leq 1 & \text{for all } j = 1, 2, \ldots, m \\
& x_{ij} \in \{0, 1\} & \text{for all } i = 1, 2, \ldots, n,\ j = 1, 2, \ldots, m \\
& y_j \in \{0, 1\} & \text{for all } j = 1, 2, \ldots, m
\end{array}
\tag{5}
$$

In any feasible solution $y_j = 1$ if and only if base roll j is used, and $x_{ij} = 1$ if and only if roll i is cut from base roll j. This formulation has a rather weak linear programming relaxation and contains a lot of symmetries, which cannot be broken by setting a branching variable. In [45] it has been shown that formulation (3) with an exponential number of variables performs much better in practical computations than formulation (5) that has only a polynomial number of variables.

Problem Specific Cutting Planes

The best commercial solvers for integer programming problems are branch-and-cut algorithms using knapsack cuts (e.g., Cplex [24], MINTO [27], XPRESS [25]). Some experimental codes also work with Gomory and lift-and-project cuts (e.g., MIPO [46, 5]). The advantage of these cutting planes is that they can be used for almost any constraint matrix. Their disadvantage is that these cuts are rather weak, i.e., they are usually not facets of the polytope of the incidence vectors of feasible solutions. Combinatorial optimization problems contain a lot of structural information. This information helps to find a (partial) description of the convex hull the incidence vectors of the feasible solutions [47]. From the inequalities of this description we can derive cutting planes that usually turn out to be much better than general purpose cutting planes.

Problem specific cutting planes of combinatorial optimization problems are usually described in a combinatorial way. For optimization problems on graphs these cutting planes are given as special

weighted subgraphs. The variables associated with nodes and/or edges of the optimization problem obtain a coefficient according to the weight of the edge and/or node in the subgraph.

An example for problem specific cutting planes for the traveling salesman problem are the 2-matching inequalities [48]. A 2-matching inequality is defined by a subset of the nodes $H \subset V$, called the *handle*, and $2k+1$ subsets ($k \geq 1$) containing 2 nodes $T_1, T_2, \ldots, T_{2k+1}$ such that each set T_i intersects the handle H in exactly one node and the sets T_i are pairwise disjoint, i.e., $|T_i \cap H| = 1$ for all $i = 1, 2, \ldots, 2k+1$ and $T_i \cap T_j = \emptyset$ for all $i, j \in \{1, 2, \ldots, 2k+1\}, i \neq j$. The 2-matching inequality is of the form

$$\sum_{i \in H, j \notin H} x_{ij} + \sum_{l=1}^{2k+1} \sum_{i \in T_l, j \notin T_l} x_{ij} \geq 6k + 4. \tag{6}$$

Finding a violated 2-matching inequality is performed by the construction of a special graph and finding a minimum cut in this graph satisfying additional side constraints [49]. The combinatorial formulation of the traveling salesman problem, i.e., the original graph, must be available for the solution of the separation problem.

The application of such problem specific cutting planes improves branch-and-cut algorithms significantly in comparison to general purpose cutting plane algorithms using Gomory, lift-and-project, or knapsack cuts. For surveys on the successful usage of problem specific cutting planes see [16, 17].

Problem Specific Variable Generation

While there are problem independent cutting planes, the dynamic variable generation is always problem specific. Therefore, its application requires the knowledge of the combinatorial structure of the problem.

Conclusions for the Design of ABACUS

The previous discussion shows that in a branch-and-cut-and-price system both the integer programming formulation and the combinatorial formulation are required. Therefore, ABACUS provides an abstract class ABA_MASTER*. A problem specific combinatorial formulation can be embedded in a class derived from this class. Combinatorial representations of constraints and variables are the interface between the combinatorial problem representation and the constraint matrix. A user of ABACUS does not need to care about the interface to the linear programming solver, i.e., the representation of the constraint matrix, which is provided by ABACUS.

Representation of Constraints and Variables

Often the separation or pricing algorithms for combinatorial optimization problems return constraints or variables in a combinatorial representation, e.g., the subsets of nodes of a two-matching constraint (6) for the traveling salesman problem or a cutting pattern for the cutting stock problem. This combinatorial representation must be transformed to a row or column of the constraint matrix together with its right hand side or its objective function coefficient, respectively.

After the transformation, it might seem that a further storage of the combinatorial representation of the constraint or variable is not necessary anymore. But there are two reasons why the combinatorial representation should not be thrown away.

* In some earlier publications on ABACUS the class names did not have the prefix ABA_.

Compact Representation

A very useful technique in branch-and-cut-price algorithms is pool separation or pool pricing. For the moment, let us conceive of a pool as a collection of constraints or variables. How a pool can be actually implemented will be discussed later. The following constraint and variable management turns out to work very well. Any separated constraint or priced-in variable is both added to the linear program and to a pool of constraints and variables. In order to be able to solve the linear programs quickly, variables and/or constraints are removed from the linear program if they turn out to be redundant. A typical criterion for removing a constraint is if its slack variable is in the basis. A variable may be removed, e.g., if its value is zero. However, a removed constraint can be violated again later in the computation and it can turn out in subsequent iterations that a removed variable has a nonzero value in the optimum solution. If a removed item is still kept in the pool, we can always easily check if it should be added again.

There are two cases when this strategy turns out to be useful:

- The violation test in the pool is faster than a direct separation or pricing.
- If heuristics for separation or pricing are used, it can happen that a constraint or variable is violated, but is not detected by the heuristic. However, it might have been found earlier at a different LP-solution and can now be regenerated from the pool. In [50] it has been shown that pool separation can significantly speed up the optimization.

Since the number of dynamically generated constraints and variables can grow very large, a compact storage format is required. For combinatorial optimization problems such a compact format is usually not the storage of the nonzero coefficients, but a combinatorial representation of the item. Consider again the subtour elimination constraints. Instead of storing all nonzero coefficients with a space consumption of $\Theta(n^2)$ in the worst case, it is sufficient to store the nodes defining one shore of the cut. This format requires space that is only linear in the number of nodes. The data structure for the subtour elimination constraints can be directly extended to the 2-matching constraints.

A variable that is associated with the edge of a graph is represented by its two end nodes. This combinatorial format uses less memory than storing the complete column of the constraint matrix explicitly.

Liftable Representation

While a compact representation is important for saving space, a smart representation of constraints is necessary in order to perform a branch-and-cut-and-price algorithm correctly. If constraints and variables are generated dynamically, then coefficients of variables added in later iterations have to be computed.

Let us again consider the traveling salesman problem. While the number of edges in the complete graph, that are in a one-to-one correspondence with the variables in the integer programming formulation, is quadratic in the number of nodes, a feasible solution contains only as many edges as there nodes in the graph. Moreover, there is a high tendency that good, and in particular optimum, solutions are mainly composed of short edges. These observations are exploited by branch-and-cut-and-price algorithms in the following way. Initially, we start with a small number of variables, e.g., the k-nearest neighbor graph for a small number k or the Delaunay graph for geometric problems. This active set of variables is dynamically augmented by variables having negative reduced cost in order to guarantee optimality on the complete graph (for details see [14, 50]).

Suppose now that there is a subtour elimination constraint in the current LP-relaxation and a variable is added. If we had stored the subtour elimination constraint only in the sparse vector format of the

linear program the computation of the coefficient of the variable would be very cumbersome. In the combinatorial representation of the constraint the computation of the coefficient is straightforward. It is 1 if exactly one of the end nodes of the corresponding edge is contained in the set S defining the subtour elimination constraint, otherwise it is 0.

Conclusions for the Design of ABACUS

Our discussion shows that a combinatorial representation of constraints and variables should be available not only at generation time, but during the entire execution of the algorithm.

Motivated by the necessary distinction of the combinatorial representation of constraints and variables and their linear programming format we will use the following terminology in the sequel. While a *constraint* is the combinatorial representation of an equation or inequality, a *row* is the sparse vector format of a constraint required by the linear programming solver. Each equation or inequality is stored in the combinatorial constraint format when it is generated. Only when it is added to the linear program, the row format is generated. If the coefficient of an active variable is nonzero, the index and the coefficient of the variable are stored in the row format. Therefore, the row format of a constraint depends on the set of active variables and can change during the execution of the algorithm if variables are generated dynamically.

Like with constraints and rows we also distinguish between a *variable* – the combinatorial representation of a variable – and a *column* – the sparse vector representation of a variable.

This notational distinction of constraints and rows, and variables and columns, respectively, is also reflected in the class hierarchy of the ABACUS system. The classes ABA_ROW and ABA_COLUMN are classes for rows and columns that are used in the interface to the linear programs. Constraints and variables are represented by the abstract classes ABA_CONSTRAINT and ABA_VARIABLE. From these two base classes, problem specific constraint and variable classes must be derived, the constraint/variable specific data structures must be declared and the pure virtual functions of the base classes that return the coefficient of a variable/constraint must be defined.

POOLS

Constraints and variables are essential components of branch-and-cut-and-price algorithms. The number of constraints and variables that has to be handled may be huge. A special data structure, which we call *pool*, stores constraints and variables in their combinatorial representation in a memory efficient way and provides us additional separation and pricing methods.

We mentioned already that every constraint and variable either induced by the problem formulation or generated in a separation or pricing step is stored in a pool. We will see later that it is advantageous to keep separate pools for variables and constraints. Then, we will also discuss when it is useful to have also different pools for different types of constraints or variables. But for simplicity we assume now that there is only one variable pool and one constraint pool.

A constraint or variable usually belongs to the set of active constraints or variables of several subproblems that still have to be processed. Hence, it is advantageous to store in the sets of active constraints or variables only pointers to constraints or variables, which are stored at some central place, i.e., in a pool that is a member of the corresponding master of the optimization, i.e, in the class ABA_MASTER or a derived class. Our practical experiments show that this memory sensitive storage format is of very high importance.

Pool Separation/Pricing

From the point of view of a single subproblem a pool may not only contain active but also inactive constraints or variables. The inactive items can be checked in the separation or pricing phase, respectively. We call these techniques pool separation and pool pricing. Motivated by duality theory we use the notion "separation" also for the generation of variables, i.e., for pricing (see [32]).

During the regeneration of constraints and variables from the pools we also have to take into account that a constraint or variable might only be locally valid. E.g., a constraint may be valid for the subproblem in which it was generated and all descendant subproblems in the enumeration tree, but invalid for all other subproblems.

The pool separation is another reason for using different pools for variables and constraints. Otherwise, each item would require an additional flag and a lot of unnecessary work would have to be performed during the pool separation.

Pool separation is also one of the reasons why it can be advantageous to provide several constraint or variable pools. E.g., some constraints might be more important during the pool separation than other constraints. In this case, we might check the "important" pools first and only if we fail in generating any item we might proceed with other pools or continue immediately with direct separation techniques.

Other classes of constraints or variables might be less important in the sense that they cannot or can only very seldomly be regenerated from the pool (e.g., locally valid constraints or variables). Such items can be kept in a pool that immediately removes all items that do not belong to the active constraint or variable set of any subproblem that still has to be processed. A similar strategy might be useful for constraints or variables requiring a big amount of memory.

Finally, there are constraints for which it is advantageous to stay active at all times (e.g., the constraints of the problem formulation in a general mixed integer optimization problem, or the degree constraints for the traveling salesman problem). Also for such constraints separate pools are advantageous.

Garbage Collection

When a lot of constraints or variables have been generated in the computation process, the pools may have become very large. In the worst case this may cause an abnormal termination of the program if it runs out of memory. But already earlier the optimization process may be slowed down since pool separation takes too long. Of course, the second point can be avoided by limited strategies in pool separation, which we will discuss later. But to avoid the first problem we require suitable cleaning up and garbage collection strategies.

The simplest strategy is to remove all items that do not belong to any active variable or constraint set of any active or open subproblem in a garbage collection process. The disadvantage of this strategy is that good items, that are accidentally momentarily inactive, can be removed. A more sophisticated strategy is to count the number of linear programs or subproblems where this item has been active and removing initially only items with a small count.

Unfortunately, if the enumeration tree grows very large or if the number of constraints and variables that are active at a single subproblem is high, then even the above brute force technique for the reduction of a pool turns out to be insufficient.

Hence, we divide constraints and variables into two groups. One consists of the items that must not be removed from the pool and the other of those items that can either be regenerated in a pricing or separation phase or are not important for the correctness of the algorithm. If we use the data structures we will describe now, then we can remove safely an item of the second group.

Pool Slots

So far, we have assumed that the sets of active variables or constraints store pointers to variables or constraints, respectively, which are stored in pools. If we remove a variable or constraint, i.e., delete the memory we have allocated for this object, then errors can occur if we later access the removed item from a subproblem. These fatal errors could be avoided if a message is sent to every subproblem where the deleted item is currently active. This technique would require additional memory and running time. Instead, we use a data structure that can handle this problem very simply and efficiently.

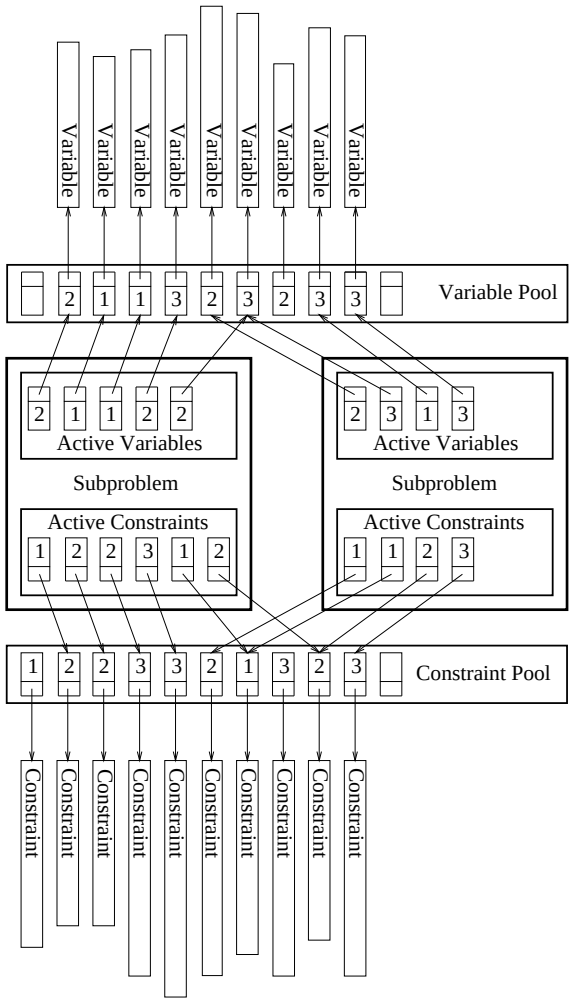

Figure 1. The pool concept

A pool is not a collection of constraints or variables, but a collection of *pool slots* (class ABA_POOLSLOT). Each slot stores a pointer to a constraint or variable or a null-pointer, if it is void. The sets of active constraints or variables in the subproblems store pointers to the corresponding slots instead of storing pointers to the constraints or variables directly. So, if a constraint or variable has been removed,

a null-pointer will be found in the slot and the subproblem recognizes that the constraint or variable must be eliminated since it cannot be regenerated. The disadvantage of this method is that our program may run out of memory due to too many unused slots.

In order to avoid this problem, we add a version number as data member to each pool slot. Initially the version number is 0 and becomes 1 if a constraint or variable is inserted in the slot. After an item in a slot is deleted, a new item can be inserted into the slot. Each time a new item is stored in the slot, the version number is incremented. The sets of active constraints and variables do not only store pointers to the corresponding slots but also the version number of the slot when the pointer is initialized. If a member of the active constraints or variables is accessed, we compare its original and current version number. If these numbers are not equal, we know that this is not the constraint or variable we were originally pointing to and remove it from the active set. We call the data structure storing the pointer to the pool slot and the original version number a *reference to a pool slot* (class ABA_POOLSLOTREF). Hence, the sets of active constraints and variables are arrays of references to pool slots. We give an illustration for this pool concept in Figure 1.

Standard Pool

The class ABA_POOL is an abstract class that does not specify the storage format of the collection of pool slots. The simplest implementation is an array of pool slots. The set of free pool slots can be represented as a linked list. This concept is realized in the class ABA_STANDARDPOOL. Moreover, a ABA_STANDARDPOOL can be static or dynamic. A dynamic ABA_STANDARDPOOL is automatically enlarged, when it is full, an item is inserted, and the cleaning up procedure fails. A static ABA_STANDARDPOOL has a fixed size and no automatic reallocation is performed.

More sophisticated implementations might keep an order of the pool slots such that "important" items are detected earlier in a pool separation and a limited pool separation might be sufficient. A criterion for this order could be the number of subproblems where this constraint or variable is active or has been active. We will consider such a pool in a future release.

Default Pools

The number of pools is very problem specific and depends mainly on the separation and pricing methods. Since in many applications a pool for the variables, a pool for the constraints of the problem formulation as a (mixed) linear integer program, and a pool for cutting planes are sufficient, we implemented this default concept. If not specified otherwise, in the initialization of the pools, in the addition of variables and constraints, and in the pool pricing and pool separation, these default pools are used. We use a static ABA_STANDARDPOOL for the default constraint and cutting planes pools. The default variable pool is a dynamic ABA_STANDARDPOOL, because the correctness of the algorithm requires that a variable which does not price out correctly can be added in any case, whereas the loss of a cutting plane that cannot be added due to a full pool has no effect on the correctness of the algorithm as long as it does not belong to the integer programming formulation.

If, instead of the default pool concept, an application specific pool concept is implemented, then the user of the framework must make sure that there is at least one variable pool and one constraint pool and these pools are embedded in a class derived from the class ABA_MASTER.

With this concept we provide a high flexibility: an easy to use default implementation, which can be changed by the redefinition of virtual functions and the application of non-default function arguments.

All classes involved in this pool concept are designed as generic classes such that they can be used both for variables and constraints.

Duplicate Constraints or Variables

In the course of a branch-and-cut-and-price algorithm several thousands of constraints and variables may be generated. For some instances of the traveling salesman problem* of the TSPLIB [51], Table I presents some statistics on the constraints generated by the branch-and-cut-and-price algorithm of D. Naddef and S. Thienel [52, 53] using ABACUS. We observe that several thousand cuttings planes

Table I. Statistics on generated TSP-constraints

Problem	cutting planes	subtours	duplicate subtours
pr76	777	78	28
ts225	85247	8799	7508
pr299	1539	485	145
att532	2608	827	286
nrw1379	5713	1609	636
pr2392	7241	3202	825

need to be generated in order to find the optimum solution. We would like to keep many of them in the constraint pool in order to regenerate them easily in a pool separation and in order to initialize the first linear program of a subproblem with the constraint system of the last linear program of its ancestor in the enumeration tree. However, our main memory is limited. Therefore, it is necessary to limit the size of a constraint pool.

Another observation is that a constraint may turn out to be currently redundant, hence it is removed, but later it may be violated again. If the constraint is kept in the pool, it can be regenerated easily. However, for large pools and constraint classes, for which fast separation algorithms are available, a separation from scratch can be faster than checking the complete pool. The disadvantage of this technique is that a constraint already contained in the pool is generated a second time and therefore stored twice in the pool. This can lead to multiple copies of a single constraint in the pool.

The third column of Table I shows the total number of subtour elimination constraints that have been separated in the course of the optimization. The separation of a subtour elimination constraint is equivalent to the solution of a minimum cut problem. It turns out that checking the pool for violated subtour elimination constraints before performing an exact separation slows down the solution time. Omitting the pool separation of subtour elimination constraints leads to the duplicate generation and storage of 25% to 85% of all generated subtour elimination constraints.

In order to avoid the undesired effect of duplicate storage of the same constraint, we have designed a special pool. If a constraint is inserted in this pool and a copy of this constraint is already present, this copy is used instead of storing a second item in the pool. Since the constraints are stored in the pool in the combinatorial format, the comparison between two constraints depends on the type of the constraint and can be defined in a class derived from the base class `ABA_CONSTRAINT`. In order to obtain a good performance of the search for a constraint we apply hashing techniques. Therefore, in addition, a constraint specific function returning a hash key must be defined.

We explained this special pool only for constraints, but of course it can be also used for variables.

* The number that is part of the name of the problem is the number of cities.

LINEAR PROGRAMS

While pools store the combinatorial representation of constraints and variables, the sparse vector format of constraints and variables is necessary for the solution of the linear programming relaxation. Moreover, linear programs might not only be used for the solution of relaxations in the subproblems, but they can also be used for other purposes, e.g., for the separation of lift-and-project cutting planes of zero-one optimization problems [46, 54] or within heuristics for the determination of good feasible solutions in mixed integer programming [55].

Therefore, we provide two basic interfaces for a linear program. The first one is in a very general form for linear programs defined by a constraint matrix stored in some sparse format. The second one is designed for the solution of the LP-relaxations in the subproblems. The main differences to the first interface are that the constraint matrix is stored in the abstract `ABA_VARIABLE`/`ABA_CONSTRAINT` format instead of the `ABA_COLUMN`/`ABA_ROW` format and that fixed and set variables are eliminated.

Another important design criterion is that the solution of the linear programs is independent of the used LP-solver, and plugging in a new LP-solver is simple.

The Basic Interface

The result of these requirements is the class hierarchy of Figure 2. The class `ABA_LP` is an abstract base class providing the public functions that are usually expected: initialization, optimization, addition of rows and columns, deletion of rows and columns, access to the problem data, the solution, the slack variables, the reduced costs, and the dual variables. These functions do some minor bookkeeping and call a pure virtual function having the same name but starting with an underscore (e.g, `optimize()` calls `_optimize()`). The functions starting with an underscore are exactly the functions that have to be implemented by an LP-solver.

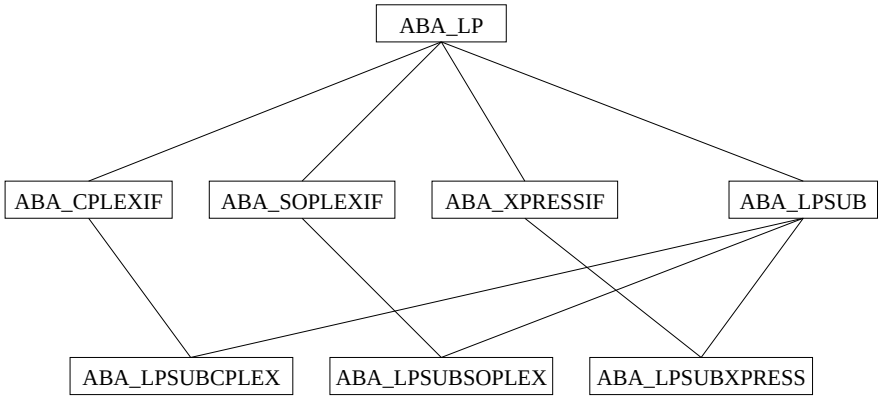

Figure 2. The linear programming classes

The LP-Solvers

The class `ABA_CPLEXIF` implements these solver specific functions for the LP-solver Cplex [24]. If a linear program is solved with Cplex, an object of the class `ABA_CPLEXIF` is instantiated. As long as only public members that are inherited from the class `ABA_LP` are used, except the constructors,

(which is usually sufficient) using another LP-solver means only replacing the name ABA_CPLEXIF by its name in the instantiation after a corresponding class for this solver has been implemented.

The advantage of encapsulating the implementation in an extra class instead of using only the private functions is that several LP-solvers can be made available within the framework and changing the solver means only changing the name in the instantiation.

In version 2.0 of ABACUS an interface to the LP-solver SoPlex [56] is implemented in the class ABA_SOPLEXIF, and in version 2.2 an interface to the LP-solver XPRESS-MP [25] is implemented in the class ABA_XPRESSIF.

Linear Programming Relaxations

The most important linear programs that are solved within this system are the LP-relaxations solved in the optimization of the subproblems. However, the active constraints and variables of a subproblem are not stored in the format required by the class ABA_LP. Therefore, we have to implement a transformation from the ABA_VARIABLE/ABA_CONSTRAINT format to the ABA_COLUMN/ABA_ROW format.

Two options are available for the realization of this transformation: either it can be implemented in the class ABA_SUB or in a new class derived from the class ABA_LP. We decided to implement such an interface class, which we call ABA_LPSUB, for the following reasons. First, the interface is better structured. Second, the subproblem optimization becomes more robust for later modifications of the class ABA_LP. Third, we regard the class ABA_LPSUB as a preprocessor for the linear programs solved in the subproblem, because fixed or set variables can be eliminated from the linear program before submitting it to the solver. It depends on the used solution method if all fixed or set variables should be eliminated. If the simplex method is used and a basis is known, then only nonbasic fixed and set variables should be eliminated. If the barrier method is used, we can eliminate all fixed and set variables. The encapsulation of the interface between the subproblem and the class ABA_LP supports a more flexible adaption of the elimination to other LP-solvers in the future and also enables us to use other LP-preprocessing techniques, e.g., constraint elimination, or changing the bounds of variables under certain conditions (see [57]), without modifying the variables and constraints in the subproblem. Preprocessing techniques other than elimination of fixed and set variables are currently not implemented.

Solving Linear Programming Relaxations

The subproblem optimization in the class ABA_SUB uses only the public functions of the class ABA_LPSUB, which is again an abstract class independent of the used LP-solver. A linear program solving the relaxations within a subproblem with the LP-solver Cplex is defined by the class ABA_LPSUBCPLEX, which is derived from the classes ABA_LPSUB and ABA_CPLEXIF. In the same way, the class ABA_LPSUBSOPLEX is derived from the classes ABA_LPSUB and ABA_SOPLEXIF for the solution of linear programming relaxations with SoPlex, and the class ABA_LPSUBXPRESS is derived from the classes ABA_LPSUB and ABA_XPRESSIF for the solution of linear programming relaxations with XPRESS-MP.

The classes ABA_LPSUBCPLEX, ABA_LPSUBSOPLEX, and ABA_LPSUBXPRESS only implement a constructor passing the arguments to the base classes. Using a different LP-solver in this context requires the definition of a class equivalent to these classes and a redefinition of the virtual function ABA_LPSUB *generateLp(), which is a one-line function that allocates an object of the desired class, and returns a pointer to the corresponding object.

Therefore, it is easy to use different LP-solvers for different ABACUS applications and it is also possible to use different LP-solvers in a single ABACUS application. For instance, if there is a very

fast method for the solution of the linear programs in the root node of the enumeration tree, but all other linear programs should be solved by Cplex, then only a simple modification of `ABA_SUB::generateLp()` is required.

In order to avoid multiple instances of the class `ABA_LP` in objects of the classes `ABA_LPSUBCPLEX`, `ABA_LPSUBSOPLEX` and `ABA_LPSUBXPRESS`, the classes `ABA_CPLEXIF`, `ABA_SOPLEXIF`, `ABA_XPRESSIF`, and `ABA_LPSUB` are virtually derived from the class `ABA_LP`. In order to save memory, we do not make copies of the LP-data in any of the classes of this hierarchy except for the data that is passed to the LP-solvers Cplex, SoPlex, or XPRESS-MP within the classes `ABA_CPLEXIF`, `ABA_SOPLEXIF`, or `ABA_XPRESSIF`.

BRANCHING

In a branch-and-cut algorithm, cutting planes are the most important ingredient and better cutting planes and separation algorithms are the key to the solution of larger problems. However, it usually cannot be guaranteed that the problem is solved by a cutting plane algorithm. Therefore we have to resort to branching. In this case good branching strategies have to be available.

In most software systems, only the standard branching strategy of changing the bounds of a variable can be used in a simple way. However, for combinatorial optimization problems, problem specific strategies can be better. A problem specific branching strategy for the traveling salesman problem can be derived from the observation that every tour must cross each cut an even number of times [58].

In a branch-and-price algorithm, branching is even more important. First, finding an optimum feasible solution by a column generation algorithm is rather unlikely, since column generation is only a smart way of solving the linear programming relaxation. Second, the classical branching technique of setting a variable to one of its bounds cannot be applied in most cases, since the branching rule must be compatible with the pricing scheme.

Different Branching Strategies

In a framework for linear programming based branch-and-bound algorithms it must be possible that many different branching strategies can be embedded. Standard branching strategies are branching on a binary variable by setting it to 0 or 1, changing the bounds of an integer variable, or splitting the solution space by a hyperplane such that in one subproblem $a^T x \geq \beta$ and in the other subproblem $a^T x \leq \beta$ must hold. A straightforward generalization is that instead of one variable or one hyperplane we use k variables or k hyperplanes, which results in a 2^k-nary enumeration tree instead of a binary enumeration tree.

Another branching strategy is branching on a set of equations ${a_1}^T x = \beta_1, \ldots, {a_l}^T x = \beta_l$. Here, l new subproblems are generated by adding one equation to the constraint system of the father in each case. Of course, as for any branching strategy, the complete set of feasible solutions of the father must be covered by the sets of feasible solutions of the generated subproblems.

For branch-and-price algorithms, often different branching rules are applied. Variables not satisfying the branching rule must be eliminated and it may be necessary to modify the pricing problem. The branching rule of Ryan and Foster [59] for set partitioning problems also requires the elimination of a constraint in one of the new subproblems.

The Branching Rules of ABACUS

We require on the one hand a rather general concept for branching, which does not only cover all mentioned strategies, but is also extendable to “unknown” methods. On the other hand it should be

simple for a user of the framework to adapt an existing branching strategy like branching on a single variable by adding a new branching variable selection strategy.

Again, an abstract class is the basis for a general branching scheme, and overloading a virtual function provides a simple method to change the branching strategy. We have developed the concept of *branching rules*. A branching rule defines the modifications of a subproblem for the generation of a descendant in the branch-and-cut tree. In a branching step, as many rules as new subproblems are instantiated. The constructor of a new subproblem receives a branching rule. When the optimization of a subproblem starts, the subproblem makes a copy of the member data defining its ancestor, i.e., the active constraints and variables, and makes the modifications according to its branching rule.

The abstract base class for different branching rules is the class ABA_BRANCHRULE, which declares a pure virtual function that modifies the subproblem according to the branching rule. We must declare this function in the class ABA_BRANCHRULE instead of the class ABA_SUB because otherwise adding a new branching rule would require a modification of the class ABA_SUB.

From the class ABA_BRANCHRULE we derive classes for branching by setting a binary variable (class ABA_SETBRANCHRULE), for branching by changing the upper and lower bound of an integer variable (class ABA_BOUNDBRANCHRULE), for branching by setting an integer variable to a value (class ABA_VALBRANCHRULE), and branching by adding a new constraint (class ABA_CONBRANCHRULE).

This concept of branching rules should allow almost every branching scheme. Especially, it is independent of the number of generated descendants of a subproblem. Further branching rules can be implemented by deriving new classes from the class ABA_BRANCHRULE and defining the pure virtual functions for the corresponding modifications of the subproblem.

In order to simplify changing the branching strategy we implemented the generation of branching rules in a hierarchy of virtual functions of the class ABA_SUB. By default, the branching rules are generated by branching on a single variable. If a different branching strategy is implemented, a virtual function must be redefined in a class derived from the class ABA_SUB.

In a specific branch-and-cut algorithm we often only want to modify the branching variable selection strategy. A new branching variable selection strategy can be implemented by redefining a virtual function.

Generalized Strong Branching

In the Cplex system the strong branching method is implemented. Its basic idea is that a candidate set of promising branching variables is selected. For each candidate the first linear program for each one of the two potential sons is (partially) solved. The variable, for which the minimal absolute objective function change in the two corresponding subproblems is maximal, is selected.

We have generalized this technique in ABACUS in various ways:

- The effect of a branching rule cannot only be tested for branching variables, but for any branching rule.
- A branching rule cannot only be evaluated by the solution of the first linear programs of the potential sons, but by an arbitrary method.
- The comparison between two branching rules cannot only be performed by comparing the change of the objective function, but in an arbitrary way.

Our default generalized strong branching is very similar to the concept of Cplex. Modifications can be performed very easily by the redefinition of virtual functions. Generalized strong branching can be

performed for problem specific branching rules by providing two virtual functions that modify and unmodify the current linear programming relaxation according to the branching rule.

OTHER FRAMEWORKS IN MIXED INTEGER PROGRAMMING

In this section we compare ABACUS with other frameworks for the solution of mixed integer programming problems.

The majority of the implemented linear programming based branch-and-bound algorithms (e.g., see [16] for a bibliography of implementations of cutting plane and branch-and-cut algorithms) are mainly computer programs for the solution of a special problem, e.g., the traveling salesman problem or the mixed zero-one optimization problem, but are not software systems building a framework for the solution of a bigger class of optimization problems.

Of course, every solver for general mixed integer programs can be considered a system for all problems that can be formulated as mixed integer programs. However, features for the exploitation of the special structure of problems and sophisticated data structures are missing in programs as lp_solve [60], or XPRESS [25].

According to the survey of Saltzman [61] only a few solvers for mixed integer optimization problems allow the addition of problems specific functions. In this section, we briefly compare three prominent systems, namely Cplex, OSL, and MINTO, with ABACUS.

Cplex

Cplex [24] provides a callable library for the solution of mixed integer optimization problems. User interaction is possible by changing many different parameters of the system. However, the usage of problem specific structures is only possible in a very limited way. A preferred order of the branching variables and a set of special ordered sets constraints can be added. Direct interaction in the optimization process is only possible with the help of callback-functions, i.e., before the first subproblem is solved and after the solution of the linear program a user function can be called, e.g., for adding/deleting columns or constraints. No other interaction is possible. Cplex does not provide any data structures for the user.

OSL

OSL [23] provides features for the solution of mixed integer optimization problems very similar to Cplex. Problem specific functions can be added for the choice of the branching variable, the enumeration strategy, adding problem specific cutting planes, and fixing of variables.

MINTO

MINTO [62, 27] provides a solver for general mixed integer programs with automatic constraint classification, preprocessing, heuristics, and constraint generation and can be also used as a framework for the implementation of branch-and-cut or branch-and-price algorithms. Problem specific functions for addition and deletion of cuts and variables, heuristics, enumeration strategy, and the selection of the branching variable can be added. Further user interaction is possible at certain fixed points in the program.

On a first glance it might seem that ABACUS and MINTO are similar systems, but the design of both systems is rather different. The reason might be that ABACUS has its historical roots in

combinatorial optimization while the design of MINTO seems to be essentially influenced by general mixed integer programming.

While the strengths of MINTO are its preprocessor and the built in cutting plane generation for general mixed integer optimization problems, the advantages of ABACUS are its abstract constraint and variable representation, the flexible pool concept, and the integration of cutting plane and column generation algorithms motivated by duality theory. Moreover the object oriented design of ABACUS provides more flexibility and extendibility.

SAMPLE APPLICATIONS

ABACUS can currently (version 2.2) be downloaded from

```
http://www.informatik.uni-koeln.de/
          ls_juenger/projects/abacus/distribution.html
```

for the following platforms:

- IBM RS6000 / Aix
- SGI / Irix 6.2
- SUN SPARC / Solaris 2.6
- SUN SPARC / Sun OS 4.1.3
- HP9000 / HPUX 10.20
- DEC Alpha / OSF1
- Intel / Linux 2.0
- Intel / Windows NT 4.0

ABACUS is currently used for a wide range of different integer and combinatorial optimization problems. We present some of them in the following list. In many cases, we can give a reference in which the problem specific use of ABACUS is explained.

- binary cutting stock problem [31] (Univ. Köln)
- crew scheduling [63] (Carmen Systems Sweden, Siemens München)
- crossing minimization in graph drawing [64] (MPI Saarbrücken)
- discrete tomography [65] (TU München)
- feedback arc set problem (Univ. Köln)
- feedback vertex set problem [66] (Univ. Heidelberg)
- frequency assignment in cellular phone networks (ZIB Berlin)
- integer cutting stock problem [67] (Univ. Köln)
- linear ordering problem [68, 13] (Univ. Heidelberg)
- location problems (Univ. Bruxelles, Univ. Grenoble)
- lot sizing problems (Univ. Louvain la Neuve)
- maximum cut problem [69] (Univ. Köln, IASI-CNR Roma)
- mixed integer optimization problem [31] (Univ. Köln)
- multi layer crossing minimization [70] (Columbia Univ., Univ. Köln, MPI Saarbrücken)
- multiple sequence alignment [71] (MPI Saarbrücken)
- network design problem (ZIB Berlin)
- one machine scheduling problem with precedence constraints (Univ. Michigan)
- physical mapping of chromosomes [72] (Univ. Heidelberg, Univ. Köln, MPI Saarbrücken)
- pickup and delivery problem [73] (Univ. Köln)
- planar augmentation [74] (MPI Saarbrücken)

- planning of telecommunication networks [75] (Univ. Bruxelles)
- processor scheduling (Univ. Buenos Aires)
- quadratic assignment problem [76] (Univ. Köln)
- secondary structure alignment [77] (MPI Saarbrücken)
- time tabling (Univ. Heidelberg)
- transportation of airline parts [78] (Univ. Utrecht)
- traveling salesman problem [52, 53] (Univ. Grenoble, Univ. Köln)
- two layer planarization in graph drawing [79, 80] (MPI Saarbrücken)
- two-dimensional compaction in graph drawing [81] (MPI Saarbrücken)
- verification techniques for distributed systems [82] (TU München)
- weighted betweenness problem [83] (Univ. Heidelberg)

CONCLUSION

The distinction in ABACUS between the combinatorial representation and the sparse vector representation of a constraint or a variable is advantageous for solving combinatorial optimization problems practically efficiently by branch-and-cut-and-price algorithms. The pools and the interfaces to the linear programs are mainly a consequence of this central idea. The branching rules of ABACUS allow a simple and flexible implementation of branching strategies that are required for solving hard combinatorial optimization problems by branch-and-cut-and-price methods.

The ABACUS system is available for several computer platforms and is being successfully used by many people for a large variety of hard combinatorial optimization problems.

REFERENCES

1. George B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, 1963.
2. Michel L. Balinski, 'Mathematical programming: Journal, society, recollections', in Jan Karel Lenstra, Alexander H.G. Rinooy Kan, and Alexander Schrijver (eds.), *History of mathematical programming*, CWI North-Holland, 1991, pp. 5–18.
3. A. Charnes, William W. Cooper, and B. Mellon, 'Blending aviation gasoline—a study of programming interdependent activities in an integrated oil company', *Econometrica*, **20** (1952).
4. Ralph E. Gomory, 'Outline of an algorithm for integer solutions to linear programs', *Bulletin of the American Mathematical Society*, **64**, 275–278 (1958).
5. Egon Balas, Sebastian Ceria, Gerard Cornuéjols, and N.R. Natraj, 'Gomory cuts revisited', *OR Letters*, **19**, 1–10 (1996).
6. A.H. Land and A.G. Doig, 'An automatic method for solving discrete programming problems', *Econometrica*, **28**, 493–520 (1960).
7. Joseph B. Kruskal, 'On the shortest spanning subtree of a graph and the traveling salesman problem', *Proceedings of the American Mathematical Society*, **7**, 48–50 (1956).
8. Edsger W. Dijkstra, 'A note on two problems in connection with graphs', *Numerische Mathematik*, **1**, 269–271 (1959).
9. Jack Edmonds, 'Paths, trees and flowers', *Canadian Journal of Mathematics*, **17**, 449–467 (1965).
10. George B. Dantzig, D. Ray Fulkerson, and Selmer M. Johnson, 'Solution of a large scale traveling salesman problem', *Operations Research*, **2**, 393–410 (1954).
11. Martin Grötschel, Michael Jünger, and Gerhard Reinelt, 'A cutting plane algorithm for the linear ordering problem', *Operations Research*, **32**, 1195–1220 (1984).
12. Robert M. Karp and Christos H. Papadimitriou, 'On linear characterizations of combinatorial optimization problems', *SIAM Journal on Computing*, **11**, 620–632 (1982).
13. Thomas Christof, 'Low-dimensional 0/1-polytopes and branch-and-cut in combinatorial optimization', *Ph.D. Thesis*, Universität Heidelberg, 1997.

14. Manfred W. Padberg and Giovanni Rinaldi, 'A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems', *SIAM Review*, **33**, 60–100 (1991).
15. Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi, 'The traveling salesman problem', in M. Ball, T. Magnanti, C.L. Monma, and G.L. Nemhauser (eds.), *Network Models*, volume 7 of *Handbook on operations research and management sciences*, North Holland, Amsterdam, 1995, pp. 225–330.
16. Michael Jünger, Gerhard Reinelt, and Stefan Thienel, 'Practical problem solving with cutting plane algorithms in combinatorial optimization', in William Cook, Lázló Lovász, and Paul Seymour (eds.), *Combinatorial Optimization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1995, pp. 111–152.
17. Alberto Caprara and Matteo Fischetti, 'Branch-and-cut algorithms', in Mauro Dell'Amico, Franceso Maffioli, and Silvano Martello (eds.), *Annotated bibliographies in combinatorial optimization*, Wiley, 1997, pp. 45–64.
18. Paul C. Gilmore and Ralph E. Gomory, 'A linear programming approach to the cutting stock problem', *Operations Research*, **9**, 849–859 (1961).
19. IBM corporation, *IBM Mathematical Programming System Extended/370 (MPSX/370), Mixed Integer Programming/370 (MIP/370)*, 1979.
20. B.A. Murtagh and Michael Saunders, 'Minos 5.4 user's guide', *Technical report*, Stanford University, Dept. of Operations Research, 1995.
21. LINDO SYSTEMS Inc., *LINDO user's manual*, 1997.
22. Roy Marsten, 'The design of the XMP linear programming library', *ACM Transations of Mathematical Software*, **7**, 481–497 (1981).
23. IBM Corporation, *Optimization Subroutine Library - Guide and Reference, Release 2.1*, 1995.
24. Cplex, *Using the Cplex Callable Library*, Cplex Optimization, Inc, 1997.
25. Dash Associates, *XPRESS-MP, Optimisation Subroutine Library*, 1995.
26. Uwe H. Suhl, 'MOPS—Mathematical OPtimization System, software tools for mathematical programming', *European Journal of Operational Research*, **72**, 312–322 (1994).
27. George L. Nemhauser and Martin W.P. Savelsbergh, 'Functional description of MINTO, a Mixed INTeger Optimizer, version 2.3', *Technical report*, Georgia Institute of Technology, School of Industrial and Systems Engineering, 1996.
28. I. Sommerville, *Software Engineering*, Addison-Wesley, Reading, Massachusetts, 1992.
29. G. Booch, *Object-oriented analysis and design with applications*, The Benjamin Cummings Publishing Company, Redwood City, California, 1994.
30. B. Stroustrup, *The C++ programming language—2nd edition*, Addison-Wesley, Reading, Massachusetts, 1993.
31. Stefan Thienel, 'ABACUS—A Branch-And-CUt system', *Ph.D. Thesis*, Institut für Informatik, Universität zu Köln, 1995.
32. Michael Jünger and Stefan Thienel, 'The design of the branch-and-cut system ABACUS', *Technical report*, Institut für Informatik, Universität zu Köln, 1997.
33. Michael Jünger and Stefan Thienel, 'Introduction to ABACUS—A Branch-And-CUt System', *Operations Research Letters*, **22**, 83–95 (1998).
34. Stefan Thienel, 'A simple TSP-solver', *Technical report*, Institut für Informatik, Universität zu Köln, http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus/abacus_tutorial.ps.gz, 1996.
35. Stefan Thienel, *ABACUS 2.0: User's Guide and Reference Manual*, Institut für Informatik, Universität zu Köln, 1997. http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus/manual.ps.gz.
36. B. Meyer, *Reusable software, the base object-oriented component libraries*, Prentice Hall, Hertfordshire, 1994.
37. R.E. Johnson and B. Foote, 'Designing reusable classes', *Journal of Object-Oriented Programming*, **1**, 22–35 (1988).
38. Stefan Thienel, 'ABACUS 1.2: User's guide and reference manual', *Technical report*, Institut für Informatik, Universität zu Köln, 1996.
39. Stefan Thienel, 'A simple TSP-solver', *Technical report*, Institut für Informatik, Universität zu Köln, http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus/abacus_tutorial.ps.gz, 1996.
40. INFORMATION PROCESSING SYSTEM Accredited Standards Committee, X3, *The ISO/ANSI C++ Draft*, 1995. http://www.cygnus.com/misc/wp/.
41. Kurt Mehlhorn, Stefan Näher, and Christian Uhrig, *The LEDA User Manual Version R 3.4.1*, Max-Planck-Institut für Informatik, Saarbrücken, 1996.
42. K.E. Gorlen, S.M. Orlow, and P.S. Plexico, *Data abstraction and object-oriented programming in C++*, John Wiley & Sons, 1990.

43. M. Lee, D. Musser, and A. Stepanow, 'The standard template library', *Technical report*, Rensselaer Polytechnic Institute, 1995. http://www.cs.rpi.edu/~musser/stl.html.
44. Michael Jünger and Volker Kaibel, 'On the SQAP polytope', *Technical report*, Institut für Informatik, Universität zu Köln, 1996.
45. Pamela H. Vance, Cynthia Barnhart, Ellis J. Johnson, and George L. Nemhauser, 'Solving binary cutting stock problems by column generation and branch-and-bound', *Computational Optimization and Applications*, **3**, 111–130 (1994).
46. Egon Balas, Sebastian Ceria, and Gerard Cornuéjols, 'A lift-and-project cutting plane algorithm for mixed 0-1 programs', *Mathematical Programming*, **58**, 295–324 (1993).
47. Alexander Schrijver, 'Polyhedral combinatorics', in R. Graham, Martin Grötschel, and L. Lovász (eds.), *Handbook of Combinatorics*, volume 2, Elsevier, 1995, pp. 1649–1704.
48. Vasek Chvátal, 'Edmonds polytopes and weakly hamiltonian graphs', *Mathematical Programming*, **5**, 29–40 (1973).
49. Manfred W. Padberg and M. Ram Rao, 'Odd minimum cut sets and b-matchings', *Mathematics of Operations Research*, **7**, 67–80 (1982).
50. Michael Jünger, Gerhard Reinelt, and Stefan Thienel, 'Provably good solutions for the traveling salesman problem', *Zeitschrift für Operations Research*, **40**, 183–217 (1994).
51. Gerhard Reinelt, 'TSPLIB—a traveling salesman problem library', *ORSA Journal on Computing*, **3**, 376–384 (1991).
52. Denis Naddef and Stefan Thienel, 'Efficient separation routines for the symmetric traveling salesman problem I: general tools and comb separation', *Technical report*, Institut für Informatik, Universität zu Köln, 1998. to appear.
53. Denis Naddef and Stefan Thienel, 'Efficient separation routines for the symmetric traveling salesman problem II: separating multi handle inequalities', *Technical report*, Institut für Informatik, Universität zu Köln, 1998. to appear.
54. Egon Balas, Sebastian Ceria, and Gerard Cornuéjols, 'Solving mixed 0-1 programs by a lift-and-project method', *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993, pp. 232–242.
55. Karla Hoffman and Manfred W. Padberg, 'Solving airline crew scheduling problems by branch-and-cut', *Management Science*, **39**, 657–682 (1993).
56. Roland Wunderling, *SoPlex, The Sequential object-oriented simplex class library*, 1997. http://www.zib.de/-Optimization/Software/Soplex/.
57. Martin W.P. Savelsbergh, 'Preprocessing and probing for mixed integer programming problems', *ORSA Journal on Computing*, **6**, 445–454 (1994).
58. Jean Maurice Clochard and Denis Naddef, 'Using path inequalities in a branch-and-cut code for the symmetric traveling salesman problem', Lawrence Wolsey and Giovanni Rinaldi (eds.), *Proceedings on the Third IPCO Conference*, 1993, pp. 291–311.
59. David M. Ryan and B.A. Foster, 'An integer programming approach to scheduling', in A. Wren (ed.), *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, North Holland, Amsterdam, 1981, pp. 269–280.
60. M.R.C.M. Berkelaar, *lp_solve 2.0*, 1995. ftp://ftp.es.ele.tue.nl/pub/lp_solve.
61. M.J. Saltzman, 'Broad selection of software packages available', *OR/MS Today*, 42–51 (1994).
62. George L. Nemhauser, Martin W.P. Savelsbergh, and Gabriele C. Sigismondi, 'MINTO, a Mixed INTeger Optimizer', *Operations Research Letters*, **15**, 47–58 (1994).
63. Fredrik Engel, 'Branching algorithms for crew scheduling problems', *Master's Thesis*, Chalmers University, 1997.
64. Petra Mutzel and Thomas Ziegler, 'The constrained crossing minimization problem - a first approach', *Operations Research Proceedings 1998, Springer*, 1999, pp. 125–134.
65. Peter Gritzmann, Dieter Prangenberg, Sven de Vries, and Markus Wiegelmann, 'Success and failure of certain reconstruction and uniqueness algorithms in discrete tomography', *Technical report*, Technische Universität München, 1997.
66. Meinrad Funke and Gerhard Reinelt, 'A polyhedral approach to the feedback vertex set', W.H. Cunningham, S.T. McCormick, and M. Queyranne (eds.), *Integer Programming and Combinatorial Optimization (5th International IPCO Conference, Vancouver, Canada, June 1996, Proceedings)*, volume 1084 of *Lecture Notes in Computer Science*. Springer Verlag, 1996, pp. 445–459.
67. Joachim Kupke, 'Ein Branch-and-Price Algorithmus für das ganzzahlige Verschnittproblem', *Master's Thesis*, Institut für Informatik, Universität zu Köln, 1998.

68. Thomas Christof and Gerhard Reinelt, 'Combinatorial optimization and small polytopes', TOP *(Spanish Statistical and Operations Research Society)*, **4**, 1–64 (1996).
69. Michael Jünger and Giovanni Rinaldi, 'Relaxations of the max cut problem and computation of spin glass ground states', *Technical Report 97.300*, Institut für Informatik, Universität zu Köln, 1997. To appear in: P. Kischka (ed.), Proc. SOR '97, Jena.
70. Michael Jünger, Eva K. Lee, Petra Mutzel, and Thomas Odenthal, 'A polyhedral approach to the multi-layer crossing minimization problem', *G. di Battista (ed.) Proc. Graph Drawing '97, Lecture Notes in Computer Science 1353, Springer, Heidelberg*, 1997, pp. 13–24.
71. Knut Reinert, Hans-Peter Lenhof, Petra Mutzel, Kurt Mehlhorn, and John D. Kececioglou, 'A branch-and-cut algorithm for multiple sequence alignment', *First Annual International Conference on Computational Molecular Biology*. ACM, 1997.
72. Thomas Christof, Michael Jünger, John Kececioglu, Petra Mutzel, and Gerhard Reinelt, 'A branch-and-cut approach to physical mapping of chromosomes by using end-probes', *Journal Computational Biology*, **4**, 433–447 (1997).
73. Norbert Pesch, 'Das pickup-and-delivery problem – betrachtet als modifiziertes traveling-salesman-problem', *Master's Thesis*, Institut für Informatik, Universität zu Köln, 1998.
74. Sergej Fialko, 'Das Planare Augmentierungsproblem', *Master's Thesis*, Universität des Saarlandes, Saarbrücken, 1997.
75. Bernhard Fortz, 'Design of survivable networks with bounded rings', *Ph.D. Thesis*, Service de Mathématiques de la Gestion, Université Libre de Bruxelles, 1998.
76. Volker Kaibel, 'Polyhedral combinatorics of QAPs with less objects than locations', *Proc. Sixth Conference on Integer Programming and Combinatorial Optimization, Springer*, 1998, pp. 409–422.
77. Hans-Peter Lenhof, Knut Reinert, and Martin Vingron, 'A polyhedral approach to RNA sequence structure alignment', *Proceedings on the Second Annual International Conference on Computational Molecular Biology RECOMB*, 1998.
78. Bram Verweij, Karen Aardal, and Goos Kant, 'On an integer multicommodity flow problem from the airplane industry', *Technical Report UU-CS-1997-38*, Department of Computer Science, Utrecht University, Utrecht, 1997.
79. Petra Mutzel, 'An alternative method for crossing minimization on hierarchical graphs', *S. North (ed.) Proc. Graph Drawing '96, LNCS 1190, Springer Verlag*, 1997, pp. 318–333.
80. Petra Mutzel and René Weiskircher, 'Two layer planarization in graph drawing', *K.Y. Chwa and O.H. Ibarra (eds.) Proc. ISAAC '98, LNCS 1533, Springer Verlag*, 1998, pp. 69–78.
81. Gunnar Klau and Petra Mutzel, 'Optimal compaction of orthogonal grid drawings', *Proceedings of the seventh conference on integer programming and combinatorial optimization IPCO'99*, 1999.
82. Javier Esparza and Stephan Melzer, 'Verification of safety properties using integer programming: Beyond the state equation', *Technical report*, Technische Universität München, 1997.
83. Thomas Christof, Markus Oswald, and Gerhard Reinelt, 'Consecutive ones and a betweenness problem in computational biology', *Proceedings of the 6th conference on integer programming and combinatorial optimization (IPCO98)*, 1998.