Solving Airline Crew Scheduling Problems by Branch-and-Cut

Karla L. Hoffman • Manfred Padberg
Operations Research Department, George Mason University, 4400 University Drive, Fairfax, Virginia 22030
New York University, MEC 8-68, *New York, New York* 10012

The crew scheduling problem is one that has been studied almost continually for the past 40 years but all prior approaches have always *approximated* the problem of finding an optimal schedule for even the smallest of an airline's fleets. The problem is especially important today since costs for flying personnel of major U.S. carriers have grown and now often exceed $1.3 billion a year and are the second largest item (next to fuel cost) of the total operating cost of major U.S. carriers. Thus even small percentage savings amount to substantial dollar amounts. We present a *branch-and-cut* approach to solving *to proven optimality* large set partitioning problems arising within the airline industry. We first provide some background related to this important application and then describe the approach for solving representative problems in this problem class. The branch-and-cut solver generates cutting planes based on the underlying structure of the polytope defined by the convex hull of the feasible integer points and incorporates these cuts into a tree-search algorithm that uses automatic reformulation procedures, heuristics and linear programming technology to assist in the solution. Numerical experiments are reported for a sample of 68 large-scale real-world crew scheduling problems. These problems include both pure set partitioning problems and set partitioning problems with side constraints. These "base constraints" represent contractual labor requirements and have heretofore not been represented *explicitly* in the construction of crew schedules thus making it impossible to provide any measure of how far the obtained solution was from optimality. An interesting result of obtaining less costly schedules is that the crews themselves are happier with the schedules because they spend more of their duty time flying than waiting on the ground.
(*Zero-one Programming; Set Partitioning; Crew Scheduling; Polyhedral Cuts; Preprocessing; Heuristics; Automatic Reformulation; Branch-and-Cut; Scientific Computation*)

1. Introduction

For many years, almost all of the major U.S. airline companies (as well as many non-U.S. companies) have used a common mathematical modeling technique for assigning crews to flights. The goal is to minimize crew costs while satisfying the many constraints imposed by governmental and labor work rules. The starting point for formulating these problems is the airline's published flight schedule, which includes departure and arrival locations and times, and equipment type for each *flight segment* during a specific month. Flight segments are nonstop flights between pairs of cities. Crew schedules are determined separately for each of the company's "fleets" of particular aircraft types.

In a first step one identifies flight "rotations", i.e., sequences of flight segments for each fleet that begin and end at individual base locations and that conform to all applicable work rules. Rotations typically last two to five days depending on the work rules of the airlines and the fleet type; overseas flights require a longer rotation period. For a rotation to be considered feasible, it must conform to FAA regulations, union contract re-

0025-1909/93/3906/0657$01 25

quirements and certain company restrictions imposed by the carrier to assure the smooth transition of crews to flights. At the same time, a cost figure is assigned to each rotation. This figure represents all incremental costs associated with the rotation and can include costs associated with the length of time a crew is away from home, per diem and lodging expenses, the amount of flying time of the crew member, and possible "deadheading", i.e., transporting a crew on a flight where they are *not* serving passengers.

Given a set of feasible rotations, one can formulate the problem of finding the "best" collection of rotations such that each flight is covered by exactly one rotation as a *set partitioning problem* (SPP):

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to:} \quad & Ax = e_m, \qquad \text{(SPP)} \\ & x_j \in \{0, 1\} \quad \text{for} \quad j = 1, \ldots, n, \end{aligned}$$

where e_m is the vector having m entries equal to one and n is the number of rotations considered. Each row of the $m \times n$ matrix A represents a flight leg, and one introduces zero-one variables x_j, associated with each rotation j, such that $x_j = 1$ if rotation j is selected and zero otherwise. c_j is the cost associated with the jth rotation. The matrix A is constructed one column at a time, where

$$a_{ij} = \begin{cases} 1 & \text{if flight leg } i \text{ is covered by rotation } j, \\ 0 & \text{otherwise.} \end{cases}$$

Billions of feasible rotations exist for as few as 1,000 flight segments in a given period. Therefore, the airlines usually generate a far smaller, though still large, collection of rotations which is intended to be representative. The final crew schedule consists of a small selection of rotations from this large group (typically less than 150 to cover 1,000 flight segments). To get an idea of the complexity of the task of crew schedulers, we have asked a major U.S. airline to provide us with the dimensions of the problem they are facing. They told us that roughly 2,700 flights using over 10 different aircraft types, approximately 5,700 cockpit crew and 9,000 flight attendants need to be scheduled per day.

There are two major components to the crew scheduling problem. The first is the generation of feasible rotations (often called column or matrix generation) and the second, the optimization of set partitioning problems. The software currently in use by most U.S. carriers to generate the set of rotations and derive the final schedule was developed almost thirty years ago (Rubin 1973) and is available from a number of vendors. In order to develop schedules for groups of 1,000 flight segments, typical of present industry practice, the existing software solves a series of subproblems, generating optimal schedules for a large number of combinations of relatively small problems having a few hundred rotations, and 50 or fewer flight segments, and combining the solutions. While this method does yield usable results, there is no way of measuring how close the schedules obtained in this manner are to optimality. On the other hand, operating costs for flying personnel of major U.S. carriers often are in excess of $1.3 billion every year (Anbil et al. 1991) and are the second largest item (next to fuel cost) of the total operating cost for a major U.S. carrier. Thus even small percentage savings amount to substantial dollar amounts.

Historically, there are two general approaches for generating subsets of the feasible schedules. The first approach, one used by the ALPPS software package (Gerbracht 1978), begins by assigning a unique crew for each flight leg, thereby starting with a feasible, extraordinarily expensive schedule since each crew covers only one flight leg. The algorithm chooses columns from this set randomly until a certain number of flight legs are covered. For these flight legs, *all* feasible rotations are generated and a new set partitioning problem (consisting of only a small subset of the total flight legs but all possible columns for that subset) is sent to an optimizer to solve. If the solution to this optimization problem has a better objective function value than the value for the subset of columns from which the problem was generated, the optimal columns replace the previous columns. The random selection of columns begins the next iteration which creates a new subproblem for the optimizer to solve. This iterative process of choosing a subset of flight legs, the generation of feasible columns and the associated solution of an optimization problem continues until either one has exhausted the available machine-time or the subproblems solved have not im-

proved the solution for the last several iterations. See Bornemann (1982), Gerbracht (1978) and Anbil et al. (1991) for details on these procedures.

The alternative approach to subproblem selection has been that of considering all of the flight legs. Since the number of columns of the matrix A that would result from an exhaustive generation of all feasible rotations to this set of flight legs could easily number in the billions, these packages use heuristic procedures to generate columns that are representative of the entire set, i.e., they randomly generate feasible rotations using sampling rules that bias in favor of "low-cost" pairings. When a representative number of such rotations has been generated, an optimization procedure is called to solve this subproblem. One iterates through pairing generation and optimization until convergence is perceived or time limits are violated. This is apparently the way the software package TRIP, marketed by American Airlines Decision Technologies, treats crew scheduling problems for American Airlines.

A more recent approach to the generation of rotations is based on a graph theoretic approach whereby one presents allowable routes by a time-staged network and generates columns based on shortest path calculations (Lavoie et al. 1988, Desroschers and Soumis 1989, Barnhart et al. 1991). Since the airline industry uses the earlier methods based on the work of Rubin (1973), and there were no pertaining data on this newer method, we did not consider this recent work in our code development or testing. This approach constitutes an interesting alternative to solving the crew scheduling problem. But, like every *dynamic* column generation scheme, it requires special provisions in the code development and we left such development to possible future work when data become available.

Regardless of the approach to column generation, all of these approaches require the solution of a large number of set partitioning problems. It has been shown that as long as the subproblems to be solved are relatively small, linear programming (or linear programming coupled with branch-and-bound) is likely to provide integer solutions quickly; see Marsten and Shepardson (1981) and Gershkoff (1989). However, as the subproblem size increases (e.g., when a subproblem has more than 100 rows) the nonintegrality of the linear programming solution increases dramatically as does the length and size of the branching tree. This paper discusses an alternative approach to the solution of *large* set partitioning problems that we have tried out on problems having up to 825 rows and up to 1.05 million variables. Problems of this size are in most cases not tractable by the traditional methods.

All but two of the problems in this test set come from U.S. airline carriers; the remaining two were provided by a European carrier. They are representative of problems arising in industrial practice. Some of the largest problems in this test set were generated specifically to test the effect that using larger subproblems might have on the overall quality of the crew schedules obtained this way. Northwest Airlines (Barutt and Hull 1990), USAir and American Airlines (Anbil et al. 1991) have each reported that as one moves to larger subproblems, the quality of the solution improves substantially. Indeed, the largest problem in this test set was created by generating *all* feasible rotations for the smallest fleet of a major carrier in order to test the proposition. The problem involved only 145 flight legs (rows of the matrix A) and 1,053,137 rotations (columns of A). For this *smallest* problem of the airline, the optimal solution was 0.5% better than the solution found by the ALPPS software package which was obtained by solving a myriad of small problems. The optimal solution was obtained in less than 37 minutes of computation time! So not only is it possible to obtain a *correct* solution to the problem, but also the computation time was entirely within "reasonable" time limits. The fact that the solution was known to be optimal provided an additional benefit—a tool to measure the value of changing a flight leg time and its effect on crew costs.

A unique feature of our software package is the fact that it can solve problems having any number of "base constraints", i.e., constraints of the form

$$a^0 \leq \sum_{j \in B} a_j x_j \leq a^1$$

where $B \subseteq \{1, 2, \ldots, n\}$, $a_j > 0$ for $j \in B$ and $0 < a^0 < a^1$. Such constraints assure that an airline complies with work rules of the following kind: the aggregate number of hours that crews located at some base spend away from their crew base must be within specified limits during each duty period. These restrictions significantly constrain the allocation of available crews

among flights and are thus a major determinant of the total operating cost for the flying personnel. In a typical situation there may be as many as ten different "crew groupings" according to different aircraft types and each grouping may have as many as seven to ten different crew bases where a "base" refers to the home airports for the crew. Other available software is *incapable* of incorporating base constraints in the generation of schedules but must repetitively re-solve successive reformulations of the problem until *some* feasible schedule is obtained. American Airlines reports "rather than explicitly including the base constraints in the LP matrix, we have used a heuristic set of weighting factors to dynamically move the solution toward one that satisfies the base constraints" (see Gershkoff 1989). Such a procedure requires a lot of skill on the part of the analyst(s), exorbitant amounts of computer time, and—worst of all—it yields schedules of indeterminate quality. By incorporating base constraints directly in the solution process, our software package yields schedules satisfying all specified work rule restrictions at provably minimum cost. Moreover, it has been observed elsewhere that minimum cost crew schedules are more "efficient" in the sense that the crew working time is better utilized. This leads to a greater satisfaction on the part of all crew members involved.

There are four components to our *branch-and-cut* optimizer: a preprocessor that tightens the user-supplied formulation; a heuristic that yields "good" integer-feasible solutions quickly; a cut generation procedure—the *engine* of this overall approach—that tightens the linear programming relaxation, and a branching strategy that selects the "next" branching variable and determines the search-tree. Prior research has shown that incorporating these components into one solver has successfully solved classes of hard combinatorial optimization problems that were previously believed to be intractable. See Hoffman and Padberg (1990) for large unstructured zero-one problems; Padberg and Rinaldi (1991) for an algorithm for large-scale symmetric traveling salesman problems; Barahona et al. (1988) and Grötschel et al. (1989) for the optimization of problems related to VLSI design; Grötschel and Monma (1990) for solution procedures to network survivability problems, etc. This paper describes a successful application of this general framework to the class of problems known as set partitioning problems with or without base constraints.

Section 2 discusses most of the mathematical background used throughout the paper. Section 3 presents an overall description of the branch-and-cut optimizer. Section 4 details the major components of the preprocessor. Section 5 then describes a linear programming based heuristic designed to find quickly "good" feasible solutions to set partitioning problems with and without base constraints. Section 6 contains a description of the constraint generation procedures, the most important component of this solver since these are the procedures by which we effectively tighten the lower bound on the problem and often obtain integer solutions without *any* tree-search related enumeration. Finally, § 7 presents computational results illustrating numerical experiments on a sample of 68 real-world airline crew scheduling problems and in § 8 we draw some conclusions from our results. All problems in this test set are available to researchers and can be obtained by contacting Karla Hoffman at KHOFFMAN@GMUVAX.GMU EDU.

2. Mathematical Background

Mathematically, the airline crew scheduling problem with base constraints is formulated as the following zero-one programming problem:

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to:} \quad & Ax = e_m, \\ & d_1 \le Dx \le d_2, \\ & x \in \{0, 1\}^n, \end{aligned} \tag{SPB}$$

where A is an $m \times n$ matrix of zeros and ones, D is a $d \times n$ matrix of (typically) nonnegative rational numbers and the vectors d_1, d_2 have d rational numbers each. As previously, e_m is the vector of m ones. We denote by

$$P_{LP} = \{x \in \Re^n \mid Ax = e_m,\ d_1 \le Dx \le d_2,\ 0 \le x \le e_n\}$$

the *linear programming relaxation* of the constraint set of (SPB). The associated integer polytope is the *convex hull* of integer points of P_{LP}, i.e.,

$$P_I = \text{conv}\{x \in P_{LP} : x \text{ integer}\}$$

and we wish to minimize the linear objective function

over the polytope P_I. Clearly, we have $P_I \subseteq P_{LP}$ and only in very rare cases do we have equality in this relation. P_I can, however, be described by a finite system of linear inequalities. Among all such linear descriptions of P_I there exists one that is essentially unique. This system is necessarily a minimal one in the sense that each of its members defines a *facet* of P_I. Of course, P_I may be empty as well—but in practical problems this is hardly ever the case.

The study of facet defining inequalities for the polytope P_I is naturally an arduous one and only few general results about the facial structure of P_I are known (see, e.g., Balas and Padberg 1976) for the case where D is vacuous. We are thus lead to consider a *polyhedral relaxation* of P_I that, however, comes as "close" as possible to the polytope P_I. The way that we do this is to separate the constraint set of (SPB) into its two components and look at polyhedral relaxations within each one of the components. Evidently, from a theoretical point of view one could do better, but *ex post* our numerical experiments justify our approximation. The first components of the constraint set is the set partitioning structure and the polyhedral relaxation that we use is the associated *set packing polytope*

$$P_I^A = \text{conv}\{x \in \Re^n : Ax \leq e_m, x \in \{0, 1\}^n\}.$$

Another relaxation of the set partitioning part of (SPB) that we consider is the *set covering polyhedron* where the inequalities $\leq$ in the definition of P_I^A are replaced by $\geq$. We use some of the polyhedral theory developed for set covering in our software package as well. The second component of (SPB) is given by the base constraints, i.e., here we use as the polyhedral relaxation of P_I the polytope

$$P_I^D = \text{conv}\{x \in \Re^n : d_1 \leq Dx \leq d_2, x \in \{0, 1\}^n\}.$$

Clearly, we have $P_I \subseteq P_I^A \cap P_I^D \cap P_{LP}$ and thus all inequalities that are *valid* with respect to P_I^A or to P_I^D are valid inequalities for the polytope P_I. The term "valid inequality for P_I" simply denotes any linear inequality, e.g., $ax \leq a_0$, such that $ax \leq a_0$ holds for all $x \in P_I$. Of course, we are interested only in "cuts" or cutting planes for the LP relaxation P_{LP}, i.e., valid inequalities $ax \leq a_0$ for P_I such that $ax > a_0$ for at least some $x \in P_{LP}$. Among all valid inequalities for P_I we are interested in those that are as good as possible. In particular, since we are approximating the polytope already, we are interested in *facet defining* inequalities for P_I^A and P_I^D. Without going into unnecessary technical detail, facet defining inequalities for P_I^A and a relative of the polytope P_I^D have been the subject of much theoretical research in the 1970s beginning with the papers of Padberg (1971, 1973, 1975). To distinguish the resulting valid inequalities for P_I from the traditional "cuts" of integer programming we call valid inequalities that are derived from polyhedral considerations "polyhedral cuts". By their mathematical properties, polyhedral cuts are the *best possible* cuts for the polytope or polyhedron from which they are derived. To use them in numerical computation one needs to (a) *descriptively* identify and (b) *algorithmically* find violated ones or show that a violated one does not exist. See, e.g., Padberg (1979) for more detail.

To study the facial structure of the set packing polytope P_I^A one associates with the given matrix A its *intersection graph* $G_A = (N, E)$ as follows, see Figure 1 for an illustration. We define a node $j \in N = \{1, 2, \ldots, n\}$ for each column a^j of A and join two nodes $i \neq j \in N$ by an edge $(i, j) \in E$ if the columns a^i and a^j of A have at least one entry equal to $+1$ in common in some row of A, i.e., if the columns a^i and a^j are nonorthogonal. The data for the graph of Figure 1 are given in the appendix. The numbers in the circles of Figure 1 correspond to nodes of G_A, while the fractions next to each node give a feasible *fractional* solution to the associated problem (*SPP*). To every feasible zero-one vector $x \in P_I^A$ there corresponds a node set $S = \{j \in N : x_j = 1\}$ of G_A. Evidently, no two nodes of S are joined by an edge of G_A and such node sets are called *stable* or *independent* node sets of G_A. It is not difficult to prove the converse: to every stable node set S of G_A there corresponds a zero-one vector x of length n that belongs to P_I^A if we define $x \in \Re^n$ by $x_j = 1$ for all $j \in S$, $x_j = 0$ for all $j \notin S$. Now let, on the other hand, $K \subseteq N$ be a node set in G_A such that every pair of nodes $i, j \in K$ defines an edge $(i, j) \in E$, i.e., the node set $K \subseteq N$ induces a *complete subgraph* in G_A. Then clearly every stable set $S \subseteq N$ meets K in at most one element, i.e., $|S \cap K| \leq 1$, and thus all $x \in P_I^A$ satisfy the inequality

$$\sum_{k \in K} x_k \leq 1, \tag{1}$$

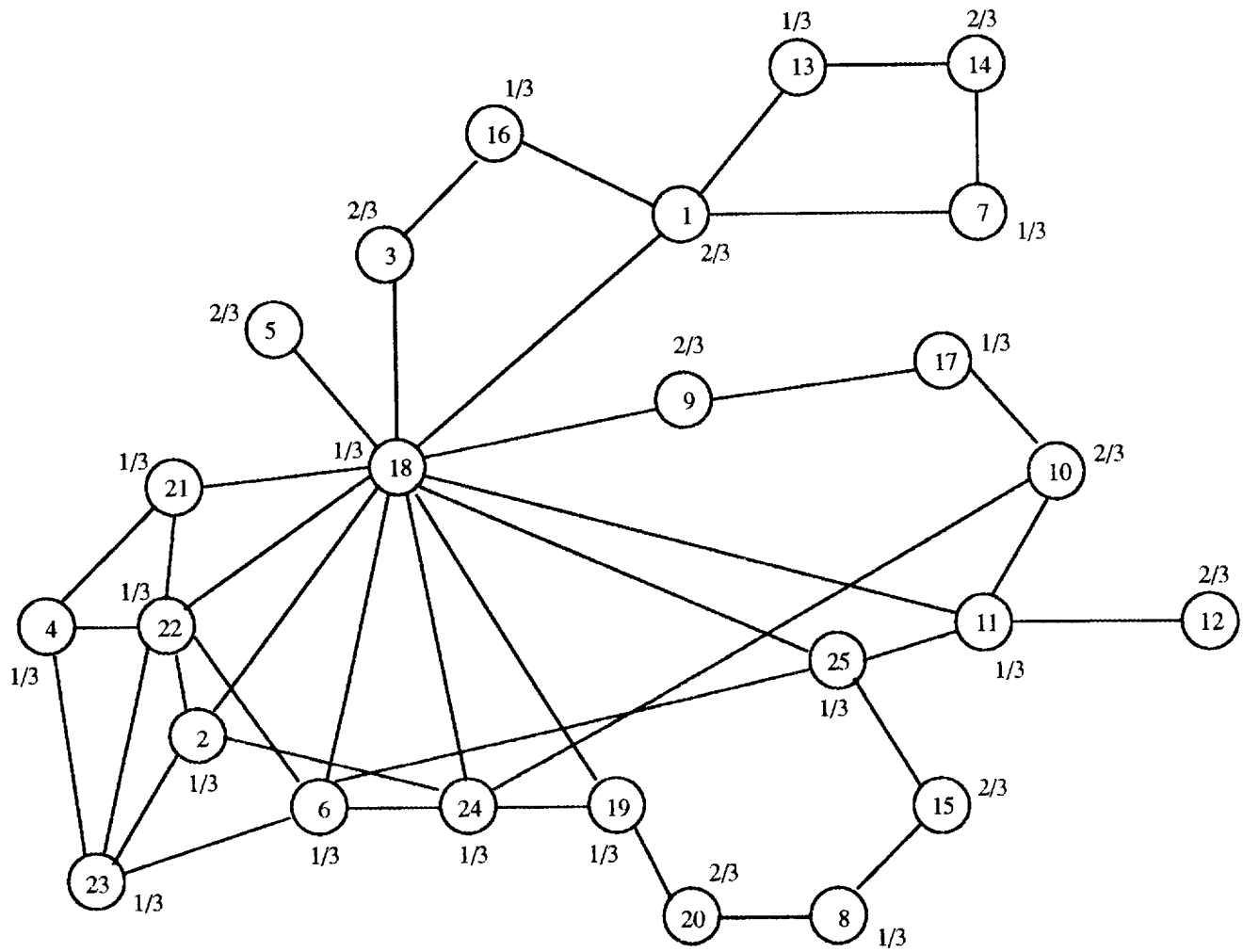
i.e., (1) is a valid inequality for P_I^A. If K is not *maximal*

with respect to its defining property, i.e., if there exists $j \in N - K$ such that node j is joined to *every* $k \in K$ by an edge of G_A, then we can replace K by $K \cup \{j\}$ in (1) and the resulting inequality is valid for P_I^A as well and "stronger" since it cuts off more *real* points of P_{LP} than the inequality (1). Thus one is led to consider maximal complete subgraphs or *cliques* in G_A and one of the oldest results about the facial structure of the polytope P_I^A due to Padberg (1971, 1973) says:

An inequality $\sum_{j \in K} x_j \leq 1$ *defines a facet of* P_I^A *if and only if* K *is the node set of a clique of* G_A.

Cliques are not the only *configurations* of the intersection graph G_A of the zero-one matrix A that give rise to facet defining inequalities of the polytope P_I^A. Rather we know from the pertaining theoretical work of the 1970s that odd cycles, their complements, so-called webs, etc., all give rise to facets of P_I^A and we will discuss this in more detail in the section on the constraint generation procedures. The important advantage—from a computational point of view as well as from a theoretical one—that one gains by studying the set packing polytope on the intersection graph is the fact that this proceeding permits an *algorithmic* approach to finding facet defining inequalities of P_I^A as well. And, of course, algorithms are the prerequisite for a computer implementation.

Figure 1 **Intersection Graph G_A for Sample Data**



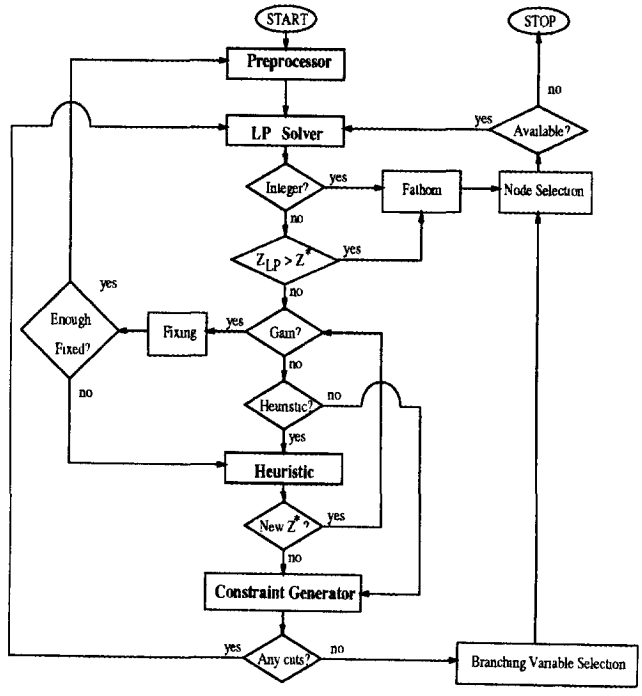
The second component of the problem (SPB) involves the polytope P_I^D. The elements of D are typically non-negative rationals and thus if $d_1 = 0$ in the definition of P_I^D then P_I^D is simply a multidimensional knapsack problem. The facial structure of the associated knapsack polytope has been studied in the 1970s as well, beginning with Padberg (1975), a paper that was written and distributed in early 1973. The way we utilize the pertaining theoretical results in our present software system is along the lines of the earlier work by Crowder et al. (1983) with several extensions to the basic idea that we have briefly described in Hoffman and Padberg (1990). Indeed all of the routines that we have described in the last mentioned paper have been incorporated into the present software system and we refer the reader to it for more detail on P_I^D.

3. A Branch-and-Cut Optimizer: Overview

The *Branch-and-Cut Optimizer* described here is designed to solve *to optimality* large set partitioning problems with base constraints having thousands of zero-one variables. It can alternatively be used as a heuristic procedure for obtaining "reasonable" feasible solutions and at the same time, a true lower bound on the optimal solution value. The latter is, of course, a yardstick to measure the quality of the solutions found. The program is written in Standard (1977 Ansi) FORTRAN which assures its portability. Not counting the LP solver, and after eliminating all comment statements, the software package has currently 19,022 lines of code. The two most code intensive pieces of this package are the cutting plane and preprocessing procedures for which there are 7,506 and 4,516 lines of code, respectively. Figure 2 provides a flow-chart of the major modules in the overall branch-and-cut approach; see also Hoffman and Padberg (1985). In this figure, Z^* is the best known integer and Z_{LP} the linear programming solution value. Gain is the increment in Z_{LP}.

In this software system, we begin by preprocessing the user-supplied formulation of the problem. Within the branch-and-cut context, "preprocessing" refers to reformulation techniques that can be performed automatically at any point within the branch-and-cut algorithm to improve or simplify a given formulation. These procedures transform a given formulation automatically into a "tighter" equivalent representation—i.e., one with variables fixed to either zero or one, redundant or inactive rows removed, and one where the difference in the objective function values between the solutions of the linear programming relaxation and the integer program, respectively, is as small as possible, while guaranteeing that an optimal solution to the original problem is not lost.

Figure 2 Flowchart of a Branch-and-Cut Optimizer



After preprocessing the user-supplied problem, a lower bound on the problem is determined by optimizing the linear programming relaxation. We have chosen to use CPLEX as our linear programming (LP) solver; see Bixby (1991). CPLEX has both a steepest-edge primal and steepest-edge dual simplex algorithm. The use of these procedures has significantly reduced the overall time required to solve the difficult, highly-degenerate linear programming subproblems arising from set partitioning problems when compared to our experience with earlier versions of this LP solver which did not have these features. Our software package is, however, sufficiently modular to permit the interfacing with other LP solvers, such as IBM's OSL.

An *upper bound* for the problem is obtained by calling

a linear programming based heuristic that iteratively reduces the size of the problem by setting variables to zero or one and examining the "logical" implications of such setting on other variables. It sends successively smaller problems (in terms of the number of "active" variables) to the linear programming solver which, in turn, directs the next round of setting. Typically, the heuristic finds "good" integer feasible solutions to the problem with or without base constraints quickly, but it is possible for it to fail in which case the upper bound is set equal to $+\infty$.

Given an upper and lower bound, we use *reduced cost fixing* to permanently fix variables; see Crowder et al. (1983) or Padberg and Rinaldi (1991) for more detail. If the reduced cost procedure fixes even a small proportion of the variables, such fixing can have logical implications for many other variables within the problem. We therefore return to the preprocessor whenever a specified percentage of the remaining variables are fixed by reduced cost fixing.

Once no more variables can be fixed, we have a "good" formulation of the problem and after solving the linear program, we begin the most important phase of the solver—the constraint generation phase. These constraints are based on polyhedral theory and are called *polyhedral cuts*. Section 6 briefly describes our constraint generation procedures for these polyhedral cuts. In particular, *clique* and *odd cycle inequalities* are generated and lifted in a manner that guarantees their validity for all feasible points to the integer set partitioning problem. For the base constraints we generate automatically constraints based on the polyhedral theory that we have described in Hoffman and Padberg (1990).

The violated constraints are appended to the original problem and the LP solver is called again. We iterate through this loop until one of the following cases prevails: (1) the solution is integer; (2) the LP is infeasible; (3) no additional cuts are generated (either because of our incomplete knowledge of the polyhedral structure or because of the incompleteness of our constraint generation procedures); (4) although cuts are generated, the objective function is not increasing sufficiently, i.e., we detect a "tailing off" of the procedure; or (5) the objective function has improved substantially relative to the bounds on other nodes of the search tree; we "pause this node", i.e., postpone its further development and look at more promising nodes.

If the first situation occurs and we are at the root node, i.e., at the top of the search tree, the solution obtained is optimal and the algorithm terminates. If this situation occurs within the search tree, we fathom the node, update the upper bound and continue the tree-search. If the second situation occurs at the root node of the tree, the overall problem is infeasible; if within the tree, we fathom the node and continue. If the third or fourth case occurs, we expand the tree. However, before expanding the tree we call the heuristic again in an attempt to update the upper bound information. If the heuristic finds a better feasible solution, the reduced cost fixing and logical fixing routines are called. If sufficiently many variables are fixed, the preprocessor is called again to tighten the problem formulation. The fifth case is similar to the third and fourth cases, except that the node is not expanded, but paused for further examination at some later time; see Padberg and Rinaldi (1991) for a more detailed discussion of these features of a branch-and-cut algorithm in the context of the symmetric traveling salesman problem.

In order to assure small search trees, our branching strategy changes depending on a variety of characteristics of the current LP solution. The strategy for branching is dependent upon the density of the matrix A, the number of fractional variables occurring in a specific row, whether any variable is sufficiently close to one, and whether there are any "outliers", i.e., variables whose objective function values are sufficiently different (statistically) from the average and that are thus likely to "move" the objective function at least on one branch of the search tree substantially away from its current value.

Thus, at every node of the branch-and-cut algorithm, the problem is reformulated, linear programs are solved, polyhedral cuts are generated, and a heuristic is called. Normally, the only reformulation done within the branching tree is the implied "setting" of variables based on the setting of the branching variables. At the root node or, if an improved upper bound has been obtained within the tree, permanent fixing is performed on the basis of the LP-reduced costs at the root node and the matrix is permanently altered, i.e., the preprocessor is called. We note that because the polyhedral cuts are

valid throughout the search tree inequalities that are generated in *any* part of the search tree are valid "globally", i.e., across the entire search tree, and the data structures for the constraint set remain unchanged when we move from one branch of the search tree to another unless the preprocessor is invoked. To share the set of generated constraints across different branches of the search tree is mathematically simply *not correct* when traditional cutting planes of integer programming, such as Gomory cuts or "intersection" cuts, are used.

4. Preprocessing

Preprocessing techniques fix variables permanently, remove rows and check for inconsistencies among constraints. Preprocessing is an effective and computationally inexpensive way of complementing the generation of "polyhedral cuts" which is, of course, the most effective way to tighten the linear programming relaxation of an integer program.

The preprocessing component of our branch-and-cut solver has two parts. In the first part it takes a given $m \times n$ zero-one matrix, the "user-supplied" set partitioning structure, and automatically and iteratively reformulates the problem so that both the linear programming relaxation of problem (SPB) and the zero-one problem itself become more tractable. Our computational results show that these techniques are highly effective in reducing the solution times of the linear programming subproblems and assist in the exact solution of *integer* set partitioning problems within a branch-and-cut framework. The second part takes the $d \times n$ matrix D of the base constraints and subjects it to all of the preprocessing that we have described in Hoffman and Padberg (1990) as well as additional preprocessing which we discuss below. For further discussion and a historical perspective on general preprocessing and formulation techniques we refer the reader to that paper.

Before the two major components of our preprocessing routines are invoked, the program "inspects" the user-supplied data and automatically permutes the rows and columns of the original problem. Columns that appear in the set partitioning structure are indexed $1, \ldots,$ NSPAR, while the remaining columns get the indices NSPAR+1, . . . , NSTRUC where NSTRUC = n is the total number of the variables of the problem. Likewise, the rows that appear in the set partitioning structure are put "on top" and the remaining ones thereafter. Each row of the set partitioning structure is stored as an *ordered list* of increasing column indices and likewise each column of A as an ordered list of increasing row indices. In doing this extra work, we create data structures that are standardized and thus easier to work with regardless of the user's indexing of the rows and columns of the problem. Because we have an ordered list representation of A we can perform most necessary operations, e.g., finding the intersection of several rows and/or several columns, rather rapidly using list processing techniques. Moreover, if NSPAR = 0 then the software system becomes exactly the software system ABC_OPT for the solution of general unstructured zero-one linear programs that we have described in Hoffman and Padberg (1990). ABC_OPT is thus a small, proper subset of the software system described here. In what follows we describe the preprocessor that is specific to the solution of (SPB).

4.1. Removing Duplicate Columns

Since the number of feasible rotations is astronomically large, the airline industry generates a very large number of "good" pairings according to appropriate heuristic criteria and then attempts to find good feasible solutions satisfying the set partitioning problem given by the collection of these pairings. Since the process of generating such subproblems is both random and heuristic, the same column may be generated more than once. In addition, rotations are generated for each crew base separately and the rotations are then combined to create a subproblem. Indeed, if there is no base constraint restriction on the problem, then two crews located at different bases or hubs could cover the same flight legs but have different associated costs. From an optimization perspective, only one of these columns needs to be considered (the one with the smaller cost). From a computational point of view duplicate columns increase the effort required to solve the overall problem and thus it is better to remove them from the data. Our procedure for identifying duplicate columns uses a hash-code based on the sum of the row indices, the first row index and the last row index. For all columns with the same hash-code a pair-wise check of the entries is performed. This procedure can be used also after rows have been

eliminated and columns have been merged which may cause two columns that were different originally to become identical.

4.2. Removing Dominated Rows

A second step in our preprocessing searches for rows that are properly contained in other rows. Let J and K be the sets of column indices associated with nonzero entries in rows j and k, respectively. If $J \cup K = J$ and $J \cap K = K$, then every variable $l \in J \backslash K$ must be zero in every feasible zero-one solution. (To show this, subtract row j from row k.) We therefore permanently fix all variables $l \in J \backslash K$ to zero and remove row j, since it is now redundant. A similar result is obtained when there exist rows differing in exactly two elements and where there is one unique index in each of the rows. Subtracting one such row from the other, yields $x_p = x_q$ where p and q are the respective unique column indices. We examine the columns p and q: if the two columns are orthogonal then columns p and q are merged into a single column having a cost of $c_p + c_q$. Otherwise, x_p and x_q must equal zero in every feasible zero-one solution and they are both permanently fixed to zero. After such fixing or merging, we remove one of the two rows since it is now redundant.

4.3. Creating a Clique Matrix

Let G_A be the intersection graph associated with the zero-one matrix A and let $M_i = \{ j \in N \mid a_{ij} = 1 \}$ be the set of columns of A that have an entry equal to $+1$ in row i of A. If there exists a clique in G_A with node set K such that M_i is a proper subset of K, then it follows that $x_j = 0$ for all $j \in K \backslash M_i$ since $\sum_{j \in K} x_j \leq 1$ is a valid inequality for P_I. Given the node set M_i, it is not difficult to determine the existence of a clique K that properly contains M_i. One simply scans all columns of A and determines those that are nonorthogonal to *all* columns in M_i. Either this set is empty and a "next" row is selected, or variables that can be fixed to zero are detected. In the latter case the zero-one matrix—and thus its associated intersection graph—are altered permanently. After having examined each row *individually*, a second pass through the resulting smaller matrix is initiated if any variables were fixed in order to benefit (possibly) from the changes that have occurred and to reduce the problem even further. It is clear that an iterative application of this procedure produces a zero-one matrix A such that every row of A corresponds to a clique in its associated intersection graph G_A. However, it is far from true that every clique in G_A is already a row of A. Rather additional cliques of G_A are generated "on the fly", see § 6. A discussion of the fixing of variables based on the clique structure for the set partitioning problem can be found already in Padberg (1973, Remarks 2.8 and 2.9) although we know of no prior implementation of this idea. When applied to the real-world problems of our computational study the "clique-detection" considerably reduces the number of variables of the original problem and, in addition, leads to a significant speed-up of the linear programming calculations, see § 7.

4.4. Reducing Base Constraints

The base constraints specify that the number of hours assigned to each crew base must be within specific minimum and maximum limits in accordance with the airline's manpower plan. These restrictions are constraints of the form

$$a^0 \leq \sum_{j \in B} a_j x_j \leq a^1$$

where $B \subseteq N$ is typically a proper subset of N, $a_j > 0$ for $j \in B$ and $0 < a^0 < a^1$. Since every base constraint is a (two-sided) knapsack constraint all of the preprocessing procedures discussed in Hoffman and Padberg (1991) can be applied to the constraints of this type and all of the preprocessing procedures described there form a subset of the routines of our present software package. In addition, we exploit the fact that the constraints of the set partitioning matrix A provide us with a set of SOS-constraints that can be used to strengthen the coefficient reduction. Specifically, let r be any row of the zero-one matrix A and let, as above, $M_r = \{ j \in N : a_{rj} = 1 \}$. If $M_r \subseteq B$ for some base constraint, then we can replace the corresponding constraint by the constraint

$$a^0 - a_{\min} \leq \sum_{j \in M_r} (a_j - a_{\min}) x_j + \sum_{j \in B \backslash M_r} a_j x_j \leq a^1 - a_{\min},$$

where $a_{\min} = \min \{ a_j : j \in M_r \}$. Evidently, such support reduction alters the knapsack constraint and if $B \backslash M_r \neq \emptyset$, then it is entirely possible that $a_j > a^1 - a_{\min}$ for some $j \in B \backslash M_r$ implying that $x_j = 0$ in every feasible solution to (SPB). Consequently the iterative application

of such preprocessing may substantially improve the user-supplied formulation of (SPB).

The basic ideas for this kind of automatic preprocessing of the set partitioning problems with and without base constraints may be quite old, but to the best of our knowledge they have not been implemented until now for the solution of large-scale problems. These procedures can also be used as a stand-alone algorithm to preprocess and improve a user-supplied formulation. Section 7 of this paper shows that the preprocessing substantially reduces the effort required to solve the linear programming (LP) relaxation of set partitioning problems and that *repeated* use of this procedure in a tree-search/LP-based zero-one problem solver reduces the time required to solve the *integer* problem substantially.

5. An LP-Based Heuristic

Our heuristic is based on the repeated solution of linear programs and exploits the well-known empirical fact that linear programming solutions to *small* set partitioning problems often have integer solutions while larger problems of the same general structure are highly fractional, i.e., the LP solutions have relatively many components that are strictly between zero and one. In our heuristic, we therefore attempt to sequentially break down the large problem into smaller problems. To accomplish this, we round a subset of the variables and use the preprocessing routines to detect further implications of such rounding. The linear programming solver is called iteratively to solve successively smaller problems and the resulting solution vector directs the next stage of the rounding, feasibility checking and logical implication setting.

The outer-loop of the heuristic algorithm consists of solving a linear programming problem and determining if the problem is infeasible, integer feasible or fractional. If either one of the first two cases is true, the heuristic terminates and updates the upper bound information if a better integer solution than any previously known was obtained. If the solution is fractional, the heuristic fixes first variables based on reduced cost information. Let GAP = ZSTAR-ZLP, where ZLP is the linear programming solution value and ZSTAR is some known upper-bound for the problem. Every variable at its lower bound whose reduced cost is greater than GAP is "set" to zero. "Setting to zero" within the heuristic means that during the current call to the procedure these variables will remain at that value. If no upper bound is known, i.e., if ZSTAR = $+\infty$, then the fixing procedure is bypassed. Next, all basic variables with value one and all variables at their upper bounds are set to one. Whenever a variable is set to one, all rows covered by that variable are removed and all other variables in any such rows are set to zero. Once these variables are removed, we check if there are any rows with a single column index. If so, those are fixed to one and the "ripple effects" on the rest of the problem of that setting are investigated and carried out.

The "inner-loop" examines the reduced problem obtained this way and continues to set variables until some specified proportion of the initial number of variables are fixed, until the problem decomposes into smaller disjoint blocks, or until the problem has been found to be infeasible because of a prior incorrect guess at the setting of variables. If infeasibility occurs, the heuristic terminates without providing an integer solution. Having created a typically much smaller subproblem, we call the linear programming solver again to direct our search and to determine the collection of variables to be set in the next round. There are three main components to the inner loop of this heuristic. First, the clique identification and row-inclusion routines described in the preprocessor are called. (We note that in our code the row inclusion routine does not merge columns but that it only identifies rows that can be removed and variables that can be fixed to zero. The merging is done in a separate routine, i.e., the package is modular by design.) A "block decomposition" routine then determines the number of disjoint components of the LP basis of the reduced problem. Finally, we attempt to split the matrix further into blocks by the controlled setting of fractional variables. The splitting routine works on each block separately. First, it finds the fractional variable in the block with a value closest to one; if there is a tie, then the one with the smallest cost is chosen. It sets that variable to one and calls routines that determine the ripple effects of that setting. The setting of variables within this block continues (always choosing the variable among the unset variables with value closest to one) until either the total number of

variables set (both to zero or one) is larger than some specified percentage of the total number of variables in the problem, or until the number of basic variables set to one within this block exceeds some percentage of the number of variables in that block. Of course, the setting of variables is somewhat "heuristic" and the exact decision rules were obtained after considerable experimentation. When we are done with a block, the routine examines the next block. However, as soon as the first of the two exiting criteria is met, only one basic variable in each of the remaining blocks is set to one. Before leaving the inner loop, the preprocessing routines are called again to examine the implications of the setting. Having determined all of the implications of the setting of the variables, the LP solver is called to solve the reduced problem. If the solution is integer, we update the upper bound information and exit. If not, the outer and inner loop of the heuristic are repeated.

Clearly, the linear program that must be solved in every major iteration of the heuristic becomes smaller and smaller and thus the heuristic converges rather rapidly. In § 7, we give numerical results for the heuristic on our sample data.

6. Constraint Generation

As we have outlined in § 2, we use three different relaxations of the underlying polytope P_I to generate polyhedral cuts in our branch-and-cut optimizer for the problem (SPB). The constraint generation procedures for the base constraint component of the overall problem have been described by us in sufficient detail in an earlier paper. So we will not review them here. The reader interested in replicating our experiments should refer to Hoffman and Padberg (1990) and the references in that paper. Here we concentrate solely on constraint generation as it relates to the set packing and set covering relaxations of (SPB) mentioned above. We begin with the set packing relaxation, though some of the following generalities apply to the set covering relaxation as well.

To derive theoretical results about the facial structure of the set packing polytope, one starts with configurations such as cliques, odd cycles, etc. in the intersection graph G_A defined by the $m \times n$ zero-one matrix A. If $m = 500$ and $n = 10{,}000$, say, this results in an unwieldy large graph—even if the density of A is very small. The density of A is the number of nonzeroes in the A matrix divided by the product of m and n and it is typically small (for problems having less than 5,000 variables, the densities range between 20% and 30%, while for larger problems, the density decreases to an average of less than 10%); see Tables 1 and 2 in § 7 where we have summarized the characteristics of our data set. The density of A clearly determines the number of edges of G_A and thus for $m = 500$ and $n = 10{,}000$ the number of edges of G_A could be in the tens of millions. Fortunately though there is no need to construct the *entire* graph G_A in order to utilize the theoretical results.

6.1. Size Reduction

At any point during the execution of a run that the constraint generator is called we have a solution vector $\bar{x} \in \Re^n$ to the current linear program that consists of the LP relaxation of (SPB) plus the previously added polyhedral cuts. Denote by

$$L = \{ j \in N : \bar{x}_j = 0 \}, \quad F = \{ j \in N : 0 < \bar{x}_j < 1 \},$$

$$U = \{ j \in N : \bar{x}_j = 1 \}$$

the index sets of variables that are at their lower bounds, that are strictly between zero and one and that are at their upper bounds, respectively, in the given solution vector $\bar{x} \in \Re^n$. As we are looking for a polyhedral cut, i.e., a facet defining inequality $ax \leq a_0$ of P_I^A such that $a\bar{x} > a_0$, we can, first of all, ignore all variables with index in L in the "identification" phase. Typically, L contains the bulk of all variables and we can simply ignore them. Second, keeping the graph theoretic interpretation of the set packing problem in mind, we can forget as well all variables within the index set U since every "neighbor" k of a node of U in the graph G_A necessarily satisfies $\bar{x}_k = 0$. This leaves us with the set F of fractional values of $\bar{x}$. To decide whether or not there exists a polyhedral cut that is violated by $\bar{x}$ it suffices thus to "inspect" the subgraph $G_F = (F, E_F)$ of G_A that is *induced* by the nodes of G_A that are in F. E_F are those edges of G_A that join nodes in F to other nodes in F. The definition of F involves strict inequalities and the *comparison* with 0 and 1. In a computer implementation this means, of course, that we have to work with *tolerances*. Depending upon what tolerances we use,

we can make G_F larger or smaller. We take care of this ambiguity by setting our tolerances for the zero/one check dynamically inside the constraint generator. In any case, the graph G_F is considerably smaller than G_A and thus a lot more manageable. Of course, we cannot totally neglect the nodes of G_A in $N\backslash F$ and we will come back to them.

Working with the subgraph G_F, that depends upon the "current" solution vector $\bar{x}$ which changes between successive calls to the constraint generation procedures, requires a procedure to set up the related data structures to represent the "current" graph on the computer "on the fly". The overhead in doing so is very low, however, compared to the benefits to be derived from working with a typically small graph having a couple of hundred—rather than several thousands—nodes and correspondingly fewer edges. The way we construct the subgraph G_F is clear from the construction of G_A (see § 2) since all one has to do is replace N in the definition with F. The data structures used to represent the graph on the computer are standard and are described in Padberg and Rinaldi (1991). The first of the constraint generation procedures is thus a procedure to construct the graph $G_F = (F, E_F)$.

To reduce the graph on which we will have to work even further we split G_F into its connected components. This is done by a simple *coloring routine*. Every connected component can be split up even further into its 2-connected blocks; see Figure 1 for an example. To this end we need to identify the *cutnodes* of every connected component of G_F, i.e., those nodes of G_F whose deletion increases the number of connected components of the remaining graph by at least one. This decomposition of G_F can be carried out rapidly and adds little in terms of overhead. For more detail on how this is done we refer the reader to Padberg and Rinaldi (1991), where exactly the same kind of decomposition has been used in the context of the symmetric traveling salesman problem. Clearly we do this work in order to reduce the *size* of the graph on which we have to carry out the identification of polyhedral cuts. Of course, if there exists a violated polyhedral cut in the original graph then there exists one in the reduced graph and vice-versa. So in the discussion that follows, we assume for notational convenience that G_F is a connected component or a 2-connected block of the original graph.

6.2. Lifting Valid Inequalities

In the graph G_F one now looks for a violated polyhedral cut $a_F x_F \le a_0$ such that $a_F \bar{x}_F > a_0$ where a_F and $\bar{x}_F$, respectively, are vectors with components indexed by the nodes in F, e.g., $\bar{x}_F = (\bar{x}_j : j \in F)$. How this is done we will describe below. First let us discuss what to do about the nodes of G_A that are in $N\backslash F$. While "extending" the violated inequality $a_F x_F \le a_0$ "trivially" by setting $a_j = 0$ for all $j \in N\backslash F$ produces a valid inequality for the polytope P_I^A (because of the nonnegativity of the constraint matrix A), it is not advisable to do so because we want an inequality that is as close as possible to a facet defining inequality for P_I^A. To this end we need to "lift" the inequality in a more meaningful way into the remaining variables with index in $N\backslash F$. "Lifting" of inequalities to obtain facets of polyhedra may be an old idea, but to the best of our knowledge it was first done in the dissertation of Padberg (1971) and later developed more generally in Padberg (1973, 1975)—with many other researchers joining the effort of describing polyhedra related to hard combinatorial optimization by way of facets in a similar vein. What is the problem that has to be solved? We are given an inequality $a_F x_F \le a_0$ that we will assume to be a facet defining inequality for the polytope that we get from P_I^A if we *restrict* the problem to the variables in the index set F, i.e., the polytope that we get if we drop all variables in $N\backslash F$. We want to determine a coefficient a_k for some $k \in N\backslash F$ such that $a_F x_F + a_k x_k \le a_0$ is valid and facet defining for the restricted problem that results if F is replaced by $F \cup \{k\}$. Clearly, we need to solve the "lifting problem"

$$\begin{aligned} \max \quad & a_F x_F \\ \text{subject to:} \quad & A_F x_F \le e_m - a^k, \qquad \text{(LIFT)} \\ & x_j \in \{0, 1\} \quad \text{for all} \quad j \in F, \end{aligned}$$

where A_F is the submatrix of the $m \times n$ zero-one matrix of A with columns in the set F and a^k is column k of A. In this notation the polytope that we get by restricting N to F is thus given by

$$P_I^F = \operatorname{conv}\{x_F \in \Re^{|F|} : A_F x_F \le e_m,\ x_F \in \{0, 1\}^{|F|}\}$$

and we have evidently $P_I^F \subseteq P_I^A$. How we solve (LIFT), we discuss later. So let z_k be the optimal objective function value of (LIFT). Since $a^k \ge 0$, we have $z_k \le a_0$ by

the validity of $a_F x_F \leq a_0$. Thus setting $a_k = a_0 - z_k$ we get a valid inequality for the enlarged polytope $P_I^{F\cup\{k\}}$ that is also facet defining. Replacing F by $F \cup \{k\}$ we can thus iterate until the original set $N\backslash F$ is exhausted and obtain a facet defining inequality for P_I^A this way. This approach—in terms of the underlying polyhedral theory—applies equally well to general bounded integer and mixed-integer programming problems when suitably modified for those problems.

At first sight, problem (LIFT) is a zero-one problem that grows in size as we "lift" more and more variables into the original inequality. Moreover, the outcome of the lifting process depends evidently upon the order in which we lift the variables x_k with $k \in N\backslash F$, see Padberg (1973) for a pertinent small example. For a computer implementation two issues arise; first, how does one solve (LIFT) as the number of variables grows in size and second, which "lifting order" does one use? If $|F| = 500$, say, and $|N\backslash F| = 9{,}500$, it is clearly out of the question to consider all *possible* lifting sequences and so we simply *randomize* the lifting order over the $|N\backslash F|$ variables. Simple as it sounds it rids us of a problem that we must resolve otherwise at an enormous computational cost. So we will have to "correct" or at least try to correct our shortcut later on. This leaves us with the problem of solving the problem (LIFT) for some *fixed lifting sequence* of the variables in $N\backslash F$. In our implementation, we lift *first* all of the zero-valued variables that correspond to the basic variables and then randomize the order of the remaining nonbasic variables to be lifted. By "randomization" we mean, of course, the use of *pseudo-random* numbers so that runs of the same data set can be reproduced exactly.

Let us consider, more generally, the solution of a problem of the type (LIFT) where $F \subseteq \{ j \in N : 0 < \bar{x}_j < 1\}$ and $k \in N\backslash F$ are arbitrary. Clearly, the size of the set F determines the computational effort required to solve (LIFT). Now if $a_j = 0$ for some variable $j \in F$ then that variable does not influence the objective function value z_k of (LIFT) and thus can be dropped from the problem. For the iterative application of "lifting" this means that *only* those variables for which the lifting coefficient $a_k = a_0 - z_k > 0$ augment the set F. This is one way of controlling the size of F. On the other hand, in (LIFT) we are effectively "forcing" variable x_k to equal $+1$ and thus all "neighbors" of node k in the graph G_F must assume a value of zero in every feasible solution to (LIFT). Denote F^* the set of columns that remains. So we have $a_j > 0$ for all $j \in F^*$ and every column a^j with $j \in F^*$ is orthogonal to column a^k. If the cardinality of F^* is small, i.e., $|F^*| \leq 16$ in our computer implementation, then it makes sense to find the optimum objective function value z_k by *complete enumeration* and one of the many procedures in our constraint generator is an *enumeration routine*. The exact determination of the lifting coefficient $a_k = z_k - a_0$ is of crucial importance in the first part of the constraint identification, i.e., when we are "finding" the set F as defined above, and as we shall see below, in this phase of the constraint generation, $|F^*|$ is indeed typically relatively small.

6.3. Approximate Lifting and Constraint Revision

If in the lifting problem $|F|$ and thus $|F^*|$ become relatively large, i.e., $|F^*| > 16$ in our computer implementation, then enumeration is out of the question and we need a different, rapid way of either calculating z_k exactly—or of approximating z_k from above. To see that the latter suffices suppose that z_k is replaced by an upper bound z_k^*, say. Then we have $a_0 - z_k^* \leq a_0 - z_k$ and thus setting $a_k = \max\{0, a_0 - z_k^*\}$ gives a valid inequality for the enlarged problem as well. We do commit an error though, in so far as the resulting valid inequality may no longer be facet defining. So we are sacrificing theory for the sake of computational speed and again our computational results justify the approximation. The question thus becomes: how does one find z_k^* quickly? Now consider the linear programming relaxation of the problem (LIFT). Then we get from duality theory:

$$\begin{aligned}
&\max\{ a_F x_F : A_F x_F \leq e - a^k,\ x_j \in \{0, 1\}\ \forall j \in F\} \\
&\quad \leq \max\{ a_F x_F : A_F x_F \leq e - a^k,\ x_F \geq 0\} \\
&\quad = \min\{ u^T(e - a^k) : u^T A_F \geq a_F,\ u \geq 0\} \\
&\quad \leq \min\{ u^T(e - a^k) : u^T A_F \geq a_F,\ u \geq 0,\ u \text{ integer}\}.
\end{aligned}$$

The last problem is a very simple problem that we can solve approximately by a *greedy algorithm* to get an upper bound z_k^* on the true optimum value z_k rather rapidly—which is important as we will have to solve this problem many times, especially when we lift the variables in $N\backslash F$ as defined in § 6.1. So the constraint generator contains a *greedy routine* as well. Remember

that the data structures for A are ordered lists. This is used in the greedy procedure as it is in most other procedures. Finally, as $|N \backslash F|$ may be rather large it does not pay to carry out the lifting—not even the approximate lifting—on all of them. As we are looking for an *optimal* solution to (SPB) and as we have the linear programming *reduced cost*, we can use the reduced cost as an indicator of the likelihood that a nonbasic variable enters the final zero-one solution vector. The higher its reduced cost, the less likely that the column will be in the final solution at value one. So we use a *cutoff* value based on the reduced cost and approximate the lifting coefficients by zero for columns that exceed the cutoff. The cutoff must, however, be set judiciously.

As should be clear from the discussion so far, we have made no assumption whatsoever about the inequality $a_F x_F \leq a_0$ to be lifted except validity. If $a_F x_F \leq a_0$ was obtained from a *clique* in G_F then evidently $a_0 = 1$ and we can specialize our lifting procedures even further. The same is the case (at least initially) if $a_F x_F \leq a_0$ was obtained from an odd cycle in G_F. The constraint generator consists indeed of many subroutines that are *tailored* to make both the identification of violated polyhedral cuts and the subsequent lifting of them to approximate the facets of P_I^A as efficient as possible. Before going into a more detailed analysis of how to "find" violated clique, odd cycle, etc. constraints, i.e., the "building blocks" for the inequality $a_F x_F \leq a_0$ to be lifted, let us discuss how to correct the "error"—to the extent that it is an error and can be corrected—that we have committed by *randomizing* the lifting sequence and/or *approximating* lifting coefficients.

Having added a number of violated polyhedral cuts, the software package calls the constraint generator typically with a set of "tight" polyhedral cuts, i.e., cuts that are satisfied at equality by the solution vector $\bar{x}$ of the current linear program. Indeed all cuts that were generated previously and that have a positive slack are simply "purged" from the constraint set and "forgotten". The alternative of placing "old" cuts that have become inactive at some point in the calculation into a "memory pool" for possible reactivation later on has not yet been implemented in our present software system. It is certainly one of the many things to do to improve the performance of our code even further, see the respective findings of Cannon and Hoffman (1990) for the software system ABC_OPT and those of Padberg and Rinaldi (1991) for the traveling salesman problem. So at any point of the calculation after the *first* call to the constraint generator there are active constraints in the current optimal LP basis that correspond to cuts that were previously found by the constraint generator and the lifting of which was done in random order. So evidently we may have missed variables that have become basic in the new LP solution. More precisely denote by $a_I x_I \leq a_0$ any such *active* polyhedral cut when restricted to the index F as defined in § 6.1 that is valid for P_I^A. Denote $H = \{ j \in F : a_j > 0 \}$ and $a_H x_H \leq a_0$ the corresponding inequality obtained by restricting $a_F x_F \leq a_0$ to the index set H. Clearly, by construction $a_H \bar{x}_H = a_0$ and thus lifting variables $k \in F \backslash H$ into the inequality $a_H x_H \leq a_0$—which is valid for P_I^A—produces a violated valid inequality for P_I^A if some (or more) lifting coefficient(s) for columns $k \in F \backslash H$ are positive. Of course, we can no longer claim that the inequality that we generate this way defines a facet of P_I^A, but given the way it was generated in the first place it cannot be very far from it either—just look up the computational record! So the constraint generator contains a *revise routine* which does a "revision" of constraints that were previously generated.

6.4. Clique Identification

Having discussed the major general procedures of the constraint generator, let us now turn to the specifics of how to find the "building blocks" for the various procedures that we have discussed so far. Evidently, clique constraints of the type discussed in § 2 are one family of constraints and some of the preceding—like the lifting procedures—can be specialized tremendously to achieve maximum efficiency. "Efficiency" here does not mean "polynomiality" in the computer science sense, but rather the normal English usage of the word. All of our procedures utilize the fact that the A matrix is stored both row-wise and column-wise as *ordered lists* which permits us to find the intersection of rows and/or columns rather rapidly. Indeed, all of our procedures are *polynomial* in the *input size* of the original problem (even the enumeration routine since its parameter enters as a constant in the complexity calculation), but polynomiality per se is too weak a criterion for the kind of "efficiency" that one needs in a computer implemen-

tation; see also Padberg and Rinaldi (1990) on this point. There are essentially three parts to clique identification.

In the *first procedure* we scan every row of A that has a nonempty intersection with the columns in F, i.e., as in § 6.1 F is the set of all fractionally valued variables of the current solution vector $\bar{x} \in \Re^n$. So let $M_r^F = \{j \in F : a_{rj} = 1\}$ for the current row r and find the set $K \subseteq F \backslash M_r^F$ of columns in F that are nonorthogonal to *all* columns in M_r^F. K can be found rapidly since the rows of A are ordered lists. If the set K is nonempty, then some or all of the columns in K form cliques with the columns of M_r^F in G_F that are necessarily violated. We identify a most-violated one and lift it into the remaining variables of the problem. Lifting of clique constraints is particularly easy since it can again be done using the ordered list structure of the constraint matrix.

The *second procedure* for clique detection uses again the fact that small problems can be solved quickly by enumeration. So let $d(v)$ denote the degree of node $v \in F$ and select a node v, say, of minimum degree of G_F. Every clique of G_F that contains v is itself a subset of the neighbors of v. Let $star(v)$ denote that set and note that $d(v) = |star(v)|$. If v is a *pending* node, i.e., $d(v) = 1$, then we delete the node v from the graph and repeat by selecting a next node. If v is a *simplicial* node, i.e., $\{v\} \cup star(v)$ forms a clique, we check if the corresponding clique constraint is violated. If so we lift and store it. In either case, we delete node v and repeat. If $d(v) \leq 16$, then we simply *enumerate* all possibilities. To do so we create, on the fly, data structures for the *complement graph* of the subgraph induced by the nodes in $star(v)$. For any graph $G = (V, E)$, the complement graph $\bar{G}$ is the graph having all nodes of G and all of the $|V|(|V| - 1)/2$ possible edges on the node set V that are *not* in G. Any clique in G defines a stable set in $\bar{G}$ and vice versa. So we create the complement graph and enumerate all stable sets in the complement graph of $star(v)$. Prior to enumerating, however, we call a greedy routine on the complement graph to find a violated clique quickly. If we find one, we do not enumerate. Otherwise, we do. Consequently, we either find a violated clique constraint that contains v or we have proven that no such clique constraint exists. In the first case we lift the corresponding constraint and store it. In either case, we delete the node from the data structure and repeat. The smaller graph that results from the node deletion may decompose into several 2-connected components and so we update the block decomposition of the resulting smaller graph in order to keep the graph as small as possible. If the minimum degree node of the graph has a degree greater than 16, then we determine a most violated clique constraint in $star(v)$ greedily as before, lift and store it if applicable, and delete the node and repeat. In the latter case, we may miss violated clique constraints. However, for large-scale problems and after the various graph reductions have been carried out, the minimum degree node frequently has a degree of 16 or less and deleting nodes successively from the graph creates graphs with smaller degrees. If all degrees of the input graph are less than or equal to 16, then our procedure is *exact*: it is guaranteed to find a violated clique constraint if one exists. Otherwise, it may or may not be exact. If the procedure is applied to the example given in Figure 1 with the node weights as specified, one shows that solution vector $\bar{x} \in \Re^{25}$ satisfies *all* clique constraints that can be derived from this graph. Note that we destroy the data structures for the graph by deleting nodes. This does not bother us since the setup of this data structure can be performed in *linear time* and is therefore no bottleneck for the overall computation.

The *third procedure* for clique detection is invoked only if the graph G_F used as an input for the second procedure is dense, i.e., if its total number of edges exceeds 50% of the number of possible edges of G_F. In this case we invoke the second procedure nevertheless and then we set up the complement graph. For every node of the complement graph we determine a maximum weight stable set containing that node *greedily* where, of course, the weights are given by the values $\bar{x}_j$ for $j \in F$. If the total weight exceeds one then we have found a violated clique in the original graph which is lifted and stored. Of course, it is entirely possible that the three procedures find the same violated clique constraint. However, due to our randomization of the lifting sequence we tend to identify distinct clique constraints even if the original "kernel" of the clique may be the same, where the kernel is the complete subgraph from which we start in order to produce a clique in G_A via randomized lifting. Nevertheless, because we do not want to have duplicate rows in the constraint matrix—for numerical reasons—the subroutine that manages the

storage of constraints checks for duplicate rows and rejects them.

6.5. Odd Cycle Identification

An *odd cycle without chords* in G_F is a set of nodes $C = \{u_1, u_2, \ldots, u_{2k+1}\}$ for $k \geq 2$ such that node u_{i-1} is joined by an edge to node u_i for $i \in \{1, \ldots, 2k+1\}$, where $u_0 = u_{2k+1}$, and no other pair of nodes of C is joined by an edge. The inequality

$$\sum_{u \in C} x_u \leq (|C| - 1)/2 \qquad (2)$$

is called an *odd cycle inequality*. It is evidently valid for P_I^A and defines a facet if we restrict P_I^A to the node set C. Thus it can be lifted to a facet of the polytope P_I^A by solving a sequence of lifting problems (LIFT) for all nodes, i.e., variables, in $N \backslash C$, see Padberg (1971, 1973). The lifting order matters and different orders frequently produce different facets of P_I^A. The problem that we face in a computer implementation of the theoretical result is simply: how do we find violated odd cycle inequalities? The textbook solution to this question goes as follows: from the graph $G_F = (F, E_F)$ and the weight vector $\bar{x}_F$ create a bipartite graph $K_F = (F_1, F_2, E^*)$, say, where $F_1 = F_2 = F$. For every edge $(u, v) \in E_F$, introduce a pair of edges (u, v'), $(v, u') \in E^*$ where $u', v' \in F_2$ are the duplicates of $u, v \in F_1$, respectively, and assign to them identical edge-weights $1 - \bar{x}_u - \bar{x}_v$. Now find for every node $u \in F_1$ the shortest path from u to its duplicate $u' \in F_2$, and pick the *minimum minimorum*. If there exists a violated odd cycle constraint, this algorithm will find it. As with most textbook solutions, it works on textbook examples. The trouble is that—while it is as easy to program as it is to state—this method will not produce the desired results. If there is no *violated* odd cycle in G_F it may return an *odd circuit* (i.e., something odd with repetition of nodes and edges!). Typically, after a few rounds of generating odd cycle constraints, none of the odd cycles as such give rise to violated inequalities of type (2). But we need an odd cycle C with $|C| \geq 5$ to start the lifting so that we can, hopefully, produce a violated *lifted* odd cycle inequality. So what do you do? In our computer implementation we have not incorporated the textbook approach. Indeed, we experimented with it and dropped it cold. (As one of the referees of this paper pointed out correctly, an odd circuit always contains some odd cycle without chords and the textbook approach can be modified to identify such an odd cycle having minimal slack in (2) from the odd circuit found by the algorithm. However, a *triangle* is also an odd cycle without chords. So if any three nodes u, v, w in a triangle of G_F satisfy $\bar{x}_u + \bar{x}_v + \bar{x}_w = 1$ and no odd cycle of length 5 or more is violated then the odd cycle returned by the textbook approach will typically be a triangle which gives rise to a *clique constraint*, but which is of no use for our purposes and which is why we have ruled out triangles by definition. Actually, this outcome is precisely what we observed in our experiments and given the facts that the constraints $Ax = e_m$, $x \geq 0$ are always satisfied and that the clique constraint identification is very effective it is not surprising that triangles are produced this way.)

We use instead the following procedure: pick a node $v \in F$, call it the root, and build a "layered" graph starting from it. Each level of the layered graph is defined by the *edge distance* that its nodes have from the root. Thus all neighbors of v are on level 1, the neighbors of the neighbors except v and those nodes that have already been assigned to level 1 form level 2, and so forth. More generally, the shortest path from level k to the root v contains exactly k edges. Any two nodes on level k that are joined by an edge and for which there exist two node disjoint paths to the root form an odd cycle that contains v. Now it is clear how to proceed: one iteratively constructs the layered graph level by level and assigns edgeweights of $1 - \bar{x}_u - \bar{x}_w$ to all edges $(u, w) \in E_F$ that are in the layered graph. Suppose we are at level $k \geq 2$ in the construction and let u and w be any two nodes on level k such that $(u, w) \in E_F$. Find a shortest path from u to the root, and "block" in the graph all neighbors of the nodes in the path except v, e.g., by assigning the corresponding edges a very large weight M. In the remaining graph find a shortest path from w to v that uses only nodes on the levels that are smaller than k. If a shortest path of length less than M exists, we have an odd cycle without chords containing u, v and w and thus the "kernel" of a facet of type (2) for our problem. If none exists, we take another edge on level k until they are exhausted. Then we construct the next level of the layered graph and repeat. The "blocking" of the neighbors of the nodes in the path from u to v assures that the resulting odd cycle has no

chords, i.e., no edges that connect any two nonconsecutive nodes of the cycle. Since the root node v is perfectly arbitrary, we can now pick a different one and repeat the process. To assure that we "touch" upon different parts of the graph G_F, we fix a *randomly* selected set of possible roots at the beginning. There is no need to run the procedure for all nodes of G_F. Also, while the procedure clearly has some polynomial bound on the number of operations, it would be too costly to execute it for all nodes. We limit ourselves to 20%–50% of all nodes in the iterative application depending upon the *density* of the graph G_F.

This construction of the layered graph translates into a nifty procedure for odd cycle detection. If the procedure finds an odd cycle then because of the definition of the edgeweights we find *violated* odd cycles—if they exist. Otherwise we get an odd cycle without chords such that the *slack* in constraint (2) is as small as possible. This constraint is then lifted into the remaining variables in $F\backslash C$ as *accurately as possible* to find violated constraints as follows: having obtained an odd cycle we determine up to five lifting coefficients *exactly* so as to produce a *most violated* lifted odd cycle inequality. More precisely, for every node in $F\backslash C$ we compute not only the exact lifting coefficient—since C is a cycle that can be done fast. However, we pick among all $k \in F\backslash C$ the one with maximum $a_k\bar{x}_k$ where a_k is the lifting coefficient. Ties are broken arbitrarily and we repeat this until we have augmented the original cycle inequality by up to five nodes from $F\backslash C$. Careful utilization of the data structures and the sparsity of the graphs involved permit substantial speed-ups of these *exact* lifting procedures. If we obtain a violated constraint then the remaining variables in $F\backslash C$ are lifted exactly into the inequality, whereas for the remaining basic (zero-valued) variables we may default to an approximate calculation of the lifting coefficients depending on the cardinality of the support of the inequality to be lifted. If the first part of the lifting did not produce a violated constraint, then we return to the layered graph routine and find another odd cycle from which the lifting is started again until the layered graph routine stops. If this cycle detection routine is applied to the example shown in Figure 1 one finds six lifted odd cycle constraints that are violated by the solution vector $\bar{x} \in \Re^{25}$ given there. While the routine has proven to be very effective in computational practice it is not an *exact* algorithm for finding a violated odd cycle constraint in arbitrary weighted graphs.

6.6. Identification of Other Constraints

Besides cliques and odd cycles our constraint generator also finds violated polyhedral cuts from other facet-producing configurations of the intersection graph. Among these are, in particular, the *complements* of odd cycles. So let $\bar{C}$ be the node set of the complement of an odd cycle in G_F. The node set of every odd cycle in the complement graph $\bar{G}_F$ of the G_F defines such a node set $\bar{C}$ and vice versa. How we find $\bar{C}$ should be clear: we produce the complement graph and run on it the cycle detection routine described above. The corresponding "kernel" of the inequality to be lifted has the form

$$\sum_{u \in \bar{C}} x_u \leq 2 \tag{3}$$

and when lifted into the variables $N\backslash\bar{C}$, the resulting inequality defines a facet of P_I^A. Because the right-hand side of the inequality (3) has a value of two, one can again *tailor* the lifting procedure so as to produce exact lifting coefficients efficiently. It is not difficult to figure out the mathematics of such a procedure and so we leave it as an exercise to the reader.

As far as the *set covering relaxation* of P_I is concerned, we have implemented constraint generation based on odd cycles as well. In this case, the inequality for an odd cycle C

$$\sum_{j \in C} x_j \geq (|C| + 1)/2 \tag{4}$$

defines under certain conditions a facet of the subpolytope that one obtains by projecting the remaining variables out of the problem at value one. We refer the reader to Sassano (1989) and Cornuejols and Sassano (1989) for the underlying mathematics of the constraint generation for the set covering relaxation of P_I. The details of the implementation have the same flavor as those described above.

As is the case almost always in scientific computation, code development requires a great deal of love and attention to detail. We believe to have given sufficient detail in this section so that our experiments can be *replicated* by anyone with some mathematical and computational training.

7. Computational Results

This is the "show and tell" section of our paper in which we present the empirical results that were produced by our experimental computer software system CREW_OPT which implements each of the components described in this paper. CREW_OPT is written entirely in FORTRAN and has been tested on a CONVEX C-220 mini-supercomputer, an IBM RISC6000 workstation, an HP Workstation and a variety of DEC hardware platforms, on which most of the development was done. Besides the linear programming solver, the major components of this software package consist of a preprocessor, a heuristic algorithm and a constraint generator. These components are invoked *repeatedly* across all branches of a search tree. We use the steepest-edge dual algorithm implemented in the linear programming solver CPLEX written by Robert E. Bixby of Rice University and marketed by CPLEX, Inc. As mentioned earlier, our software package is designed in such a way that other linear programming packages can be used in lieu of CPLEX. The test set consists of 55 pure set partitioning problems provided by four different airline corporations and 13 set partitioning problems with base constraints provided by two of those carriers. All but one of the base constraint problems have three bases, and thus, up to six knapsack constraints. The base constraint problem having 28,016 variables has eight bases thereby resulting in a problems having 15 knapsack constraints. (One lower-bound constraint can be eliminated since there is a lower bound of zero on the number of flying hours allowed for that base.)

Table 1 provides a description of the pure set partitioning problems and Table 4 provides comparable information for the set partitioning problems with side constraints. We note that only 52 of the 55 pure set partitioning problems are presented in the next few tables. The other three problems will be described separately below because they are intrinsically harder and require further discussion.

In Tables 1 and 4, the headings Cols, Rows and Dens provide the number of columns, the number of rows and the density of the original problem, where density refers to the percentage of nonzeroes in the A-matrix. The headings PCols, PRows and PDens provide comparable information after the *initial* preprocessing. Examination of the data in Tables 1 and 4, show that the

Table 1 Pure Set-Partitioning Problems: Problem Characteristics

Cols	Rows	PCols	PRows	Dens	PDens
197	17	177	17	22 10	22 27
294	19	251	18	24 30	25.81
404	19	336	19	26 95	26 83
434	24	356	24	22 39	22.36
467	31	463	29	19 55	21 00
577	25	426	25	24 89	24 33
619	23	531	23	23 87	24 10
626	27	454	27	20 00	19.06
677	25	567	25	26 55	26 25
685	22	566	22	24.70	25 00
711	19	473	18	24 80	24.87
770	19	639	19	25 83	25 89
771	23	473	18	23 77	23 12
853	24	659	24	21 18	21 25
899	20	750	20	28.06	28.16
1072	18	982	17	25 18	26.43
1079	23	895	23	26.33	26 05
1210	18	825	18	39 27	38 43
1217	20	844	20	30.16	30.15
1220	23	911	23	32 33	31 44
1355	22	926	22	31.52	30 76
1366	19	925	19	33 20	33 19
1709	23	1403	23	26.70	27 02
1783	20	1408	20	36.90	36.14
2540	18	2034	18	31 04	30 99
2653	26	1884	26	29 63	29 80
2662	26	1823	26	28 86	29 21
2879	40	2145	40	21 88	21 59
3068	23	2415	23	30.76	30.75
3103	40	2305	40	16 20	16 05
5172	36	3108	36	22.12	21 86
5198	531	3846	360	1 32	1 54
6774	50	5977	50	18 17	18 27
7292	646	5862	488	1 10	1.21
7479	55	5957	50	13 67	13 47
8308	801	6235	521	99	1 12
8627	825	6694	537	1.00	1 32
8820	39	6488	39	16 64	16 81
10757	124	8460	124	6 82	6 83
13635	100	9022	45	14 13	16 57
16043	51	10950	51	12 78	12 57
28016	163	6564	112	6 52	7 48
36699	71	16542	69	8 16	8 34
43749	59	38964	59	14 13	14 12
51975	135	50069	135	5 86	5 87
85552	77	27084	53	18.40	21.42
87879	145	85258	145	5 66	5 68
118607	61	78186	61	13 96	13 96
123409	73	95178	73	10 04	10 11
148633	139	138951	139	7 27	7 23
288507	71	202603	71	10 06	10.07
1053137	145	370642	90	9.1	9 8

Table 2 **Pure Set-Partitioning Problems: LP, IP and Heuristic Solution Values**

Cols	Rows	Z_{LP}	Z_{HEUR}	Z_{IP}
197	17	10972 5	failed	11307
294	19	14570.0	15600	14877
404	19	10658.3	11070	10809
434	24	35894.0		35894
467	31	67743.0		67743
577	25	7380 0	7676	7408
619	23	6942.0	6984	6984
626	27	14118 0		14118
677	25	9868 5	failed	10080
685	22	16626 0	16812	16812
711	19	12317.0	13702	12534
770	19	9961 5	10377	10068
771	23	6743.0	7452	6796
853	24	68271 0		68271
899	20	10453 5	11613	10488
1072	18	8897 0	8904	8904
1079	23	7485.0	7846	7656
1210	18	8169.0	failed	8298
1217	20	5852 0	6610	5960
1220	23	5552 0	5712	5558
1355	22	9877 0	failed	9933
1366	19	5843 0	6568	6314
1709	23	7206 0	7340	7216
1783	20	7260 0	7634	7314
2540	18	4185 3	4378	4274
2653	26	3726 8	3942	3942
2662	26	7980 0	9754	8038
2879	40	10898 0		10898
3068	23	6484.0	7536	6678
3103	40	67760 0		67760
5172	36	5476.0		5476
5198	531	30494.0		30494
6774	50	7640.0	9616	7810
7292	646	26977 2	27040	27040
7479	55	1084.0	1096	1086
8308	801	53735 9	54060	53839
8627	825	49616 4	49713	49649
8820	39	116254 5	116259	116256
10757	124	338864.3	392090	340160
13635	100	5965 0		5965
16043	51	50132.0	50240	50146
28016	163	17731 7	17854	17854
36699	71	215 3	221	219
43749	59	24447.0	25086	24492
51975	135	114852.0		114852
85552	77	5338 0		5338
87879	145	105444 0		105444
118607	61	10875 8	11907	11115
123409	73	61844 0		61844
148633	139	1181590 0		1181590
288507	71	132878 0		132878
1053137	145	9949.5	10075	10022

initial preprocessing has a proportionally greater effect on the larger problems. We wish to reemphasize, however, that preprocessing is employed as a tool that is used *repeatedly* within the overall branch-and-cut algorithm, and not merely as a one-time procedure for tightening the formulation. It is interesting to note that, for most problems, the initial preprocessing did not significantly alter the density of the matrix and that, in general, density decreases as the problem size increases.

Tables 2 and 5 provide information regarding the quality of the linear programming relaxation as a bound on the integer solution and the quality of the heuristic routine used as a "stand-alone" routine, i.e., Z_{HEUR} is the *first* integer feasible solution obtained by the heuristic, provided it did not "fail" to find one. Z_{LP} is the solution value of the LP problem after preprocessing and Z_{IP} is the optimal 0-1 solution value. If no number appears in the Z_{HEUR} column, it is because the linear programming solver provided the optimal integer solution and the heuristic procedure was not invoked.

Tables 3 and 6 provide a comprehensive picture of the effort expended overall by CREW_OPT and the proportion of effort expended in each of its major components. The headings Prep Calls, Lp Calls, and CG Calls refer to the number of times the preprocessing routines, the linear programming solver, and the constraint generation routines were called, respectively. The number of calls to the LP routine does not, however, include the calls to this routine within the heuristic procedure which we simply did not count. The number of cuts reported is the total number of polyhedral cuts generated during the entire solution process, and the number of nodes is the total number of nodes on the branching tree. The times reported for all but the four largest pure set partitioning problems are in CPU seconds on a RISC6000 model 550 machine. The times for the four largest pure set partitioning problems were obtained on a CONVEX model C-220 machine using one of its two processors because due to memory limitations we were unable to solve them on the RISC6000 machine; these problems are marked with an asterisk in Table 3. The largest problem in the table was solved in two stages. First duplicate columns were removed and the problem rewritten to a file. The total time for this duplicate removal was 842 seconds of which only 112 seconds were used to find the duplicates and the remainder for reading and writing the data on the CON-

Table 3 **Pure Set-Partitioning Problems: Branch and Cut Performance Measures**

# of Cols	# of Rows	Prep Calls	LP Calls	CG Calls	# of Cuts	# of Nodes	Prep Time	LP Time	Heur Time	CG Time	Total Time	Proc
197	17	3	5	1	2	0	0.02	0 03	0.0	0 0	0 06	CG
294	19	3	7	2	6	0	0 03	0 07	0 02	0 01	0.17	CG
404	19	2	4	1	4	0	0.03	0 05	0 08	0 01	0 21	CG
434	24	1	1	0	0	0	0 02	0 04	0 00	0 00	0 08	LP
467	31	1	1	0	0	0	0 03	0 04	0.00	0 00	0 10	LP
577	25	2	2	1	3	0	0 04	0 06	0 14	0 01	0.30	CG
619	23	7	3	2	6	0	0 03	0 09	0.13	0.02	0.34	CG
626	27	1	1	0	0	0	0 02	0 03	0 00	0 00	0 09	LP
677	25	3	5	1	2	0	0 05	0 08	0.03	0 00	0 19	CG
685	22	2	4	1	9	0	0 04	0 07	0.44	0 03	0.62	CG
711	19	2	4	1	3	0	0.03	0 09	0.13	0.02	0 34	CG
770	19	2	4	1	1	0	0 04	0 05	0 06	0 00	0 19	CG
771	23	3	5	1	3	0	0 05	0 08	0.17	0.01	0.34	CG
853	24	1	1	0	0	0	0 03	0 05	0 00	0.00	0 13	LP
899	20	3	5	1	1	0	0.05	0 10	0.09	0.00	0 30	CG
1072	18	2	2	1	1	0	0 05	0.08	0.17	0 01	0.38	CG
1079	23	6	6	4	17	0	0 06	0.15	0 59	0 07	0 99	CG
1210	18	2	4	1	2	0	0 04	0.09	0 16	0 10	0 40	CG
1217	20	3	5	1	7	0	0.06	0 10	0 36	0 03	0.62	CG
1220	23	2	4	1	3	0	0.08	0 13	1.06	0 01	1 35	CG
1355	22	2	4	1	2	0	0.06	0 06	0.08	0.00	0 28	CG
1366	19	7	14	3	15	0	0 09	0 13	0 23	0 03	0 56	CG
1709	23	2	4	1	1	0	0.09	0.10	0 18	0 00	0 48	CG
1783	20	4	13	10	180	0	0.11	0 91	0.26	2.15	3.68	CG
2540	18	6	12	3	14	0	0 15	0.32	0 32	0.03	0 99	CG
2653	26	8	8	0	2	0	0.16	0 26	0.20	0 01	0.75	CG
2662	26	3	7	3	28	0	0 16	0.38	0 41	0.28	1 43	CG
2879	40	1	1	0	0	0	0 16	0 28	0.00	0.00	0 50	LP
3068	23	5	9	2	7	0	0 20	0 31	0 70	0 05	1 45	CG
3103	40	1	1	0	0	0	0.12	0 27	0.00	0.00	0.53	LP
5172	36	1	1	0	0	0	0.21	0.28	0.00	0 00	0 74	LP
5198	531	1	1	0	0	0	3.90	6 00	0.00	0 00	10 15	LP
6774	50	6	10	3	33	0	0 72	2 28	6 39	0 42	10 41	CG
7292	646	8	12	3	74	0	11 99	17 56	5 89	1 44	37 30	CG
7479	55	14	32	20	229	2	0.98	4 15	7 35	22.16	35 40	BC
8308	801	9	53	42	345	4	18 32	73.75	23.73	92.93	215.30	BC
8627	825	12	12	2	37	0	11 07	29.81	6 03	0.93	48 42	CG
8820	39	2	4	1	3	0	0.46	0 74	0.45	0 01	2 05	CG
10757	124	1	3	1	15	0	0.68	8.16	13.61	38.84	62 49	CG
13635	100	1	1	0	0	0	1 24	2 58	0.00	0.00	4.78	LP
16043	51	2	5	2	4	0	0 86	1 90	0.83	0 01	4 29	CG
28016	163	2	2	0	0	0	2.87	2 38	4.39	0 04	11 19	CG
36699	71	5	10	8	127	0	2.95	16.48	50.31	62.10	134 38	CG
43749	59	5	8	5	20	0	3.07	11.48	6 90	0.18	24.00	CG
51975	135	1	1	0	0	0	2 76	13.92	0 0	0.0	19.25	LP
85552	77	1	1	0	0	0	6.83	7 86	0.0	0 0	20.27	LP
87879	145	1	1	0	0	0	3.07	27.88	0.0	0.0	37.35	LP
118607	61	13	43	15	48	4	8 07	34.68	34 67	2.32	87.53	BC
*123409	73	1	1	0	0	0	31 1	50 7	0.0	0 0	87.6	LP
*148633	139	1	1	0	0	0	44.3	122 2	0.0	0 0	174 4	LP
*288507	71	1	1	0	0	0	65 3	119 9	0.0	0 0	192 5	LP
*1053137	145	12	44	26	389	0	156.2	628 3	530 7	62.7	1410 6	CG

Table 4 Base Constraint Problems: Problem Characteristics

Cols	Rows	PCols	PRows	Dens	PDens
86	25	84	21	24 42	26.81
154	28	147	23	23.00	23 07
189	23	170	22	24 82	24.25
191	27	147	23	23.95	22 89
504	28	491	26	27 30	28 97
699	31	689	28	25 77	26 91
819	26	810	21	27 25	28.91
854	31	814	28	15 73	26.78
1586	32	1539	29	24 23	24 36
1609	25	1447	21	38 18	44 54
2296	30	49	17	32 66	43.94
28016	179	12672	130	7 05	7 75
85552	83	31852	57	19.47	22 67

VEX machine. The time reported in the table refers to the second stage where the resulting reduced problem with 370,642 variables was provided to and solved by CREW_OPT. The last column in this table, labeled Proc, indicates the procedures used to prove optimality. LP indicates that the first LP provided the integer optimal solution and thus only the preprocessor and the LP solver were used. HR indicates that in addition to the preprocessor and the LP solver, the heuristic was needed. Similarly, if the row has the notation CG, constraint generation was needed to prove optimality, and finally, BC denotes that *all* procedures were called including branching to construct a search-tree.

Each of the pure set partitioning problems within our test set having less than five thousand variables was solved in under four seconds with an average time of less than one second. Of all 52 problems in this set, only one required more than four minutes and that problem has over one million original variables. The total solution time for this largest problem was under 37 minutes which includes the time to read the entire problem, remove duplicate columns and rewrite the data for the problem without duplicates—*far less time than was required to generate the rotations associated with the problem*. Only three of the 52 problems required branching and the largest search tree produced by our software package had a total of four nodes. So almost all of the problems are very easily solved by a state-of-the-art combinatorial problem solver such as CREW_OPT.

An examination of the times reported in Table VI for problems with base constraints shows that the inclusion of base constraints makes the problem more difficult. Unlike the pure set partitioning problems, even relatively small problems have fractional solutions. In all cases optimal solutions are provided quickly. All but the largest problem were solved in under fifteen seconds. The largest problem, which had 85,552 variables, required slightly over one half hour to solve to proven optimality. In six of the thirteen cases tree-search was required.

The three other problems in the test set not yet described are pure set partitioning problems that require significantly more computational effort than the rest. For this reason the three problems became sort of our "problem children" which in turn prompted us to write more software—especially for the constraint generator. This lead to an overall improvement of CREW_OPT and the current package handles also these three "outliers" satisfactorily. The problems sizes and characteristics of these problems are presented in Table 7. We do not know what makes these problems more difficult. We have found nothing unique about these problems in terms of size, density, distribution of the nonzeroes, or distribution of the cost data. At present, CREW_OPT is the *only* code capable of proving optimality on any one of these problems. Problem NW04 requires 44 minutes to prove optimality, although the optimal solution of 16862 was found on the first node after only 5 minutes (298 seconds) on a IBM RISC6000 Model 550 machine. The linear programming relaxation has a solution value of 16310.7 (which is 3.3% lower than the optimal so-

Table 5 Base Constraint Problems: LP, IP and Heuristic Solution Values

Cols	Rows	Z_{LP}	Z_{HEUR}	Z_{IP}
86	25	7819 1	failed	8296
154	28	9794.6	failed	10420
189	23	8943 0	failed	10002
191	27	6329 4	failed	7036
504	28	5395 4	6032	5944
699	31	6051 0	6700	6056
819	26	5771 0	5788	5777
854	31	6431 1	6446	6446
1586	32	7165.8	failed	7526
1609	25	3676 4	failed	3768
2296	30	6932 5	failed	7556
28016	179	18236.4	18263	18263
85552	83	5349 5	failed	5697

Table 6 Base Constraint Problems: Branch and Cut Performance Measures

# of Cols	# of Rows	Prep Calls	LP Calls	CG Calls	# of Cuts	# of Nodes	Prep Time	LP Time	Heur Time	CG Time	Total Time	Proc
86	25	4	8	2	6	2	0 79	0 06	0.04	0.01	0.97	BC
154	28	7	48	26	106	8	0 55	0 38	0 10	0 20	1.51	BC
189	23	4	9	2	14	0	0.48	0.12	0.06	0 05	0.78	CG
191	27	3	12	5	10	2	0 90	0 14	0.16	0 08	1 45	BC
504	28	4	16	3	10	2	1 34	0 38	0 47	1 22	4 21	BC
699	31	2	4	1	2	0	0.71	0 10	0.08	0.01	0 94	CG
819	26	3	5	2	6	0	0 75	0.13	0 08	0.01	1 01	CG
854	31	3	3	5	39	0	0 71	0.13	0 15	0 00	1 04	HR
1586	32	10	29	6	60	2	2 47	1 18	2 40	8 14	15.30	BC
1609	25	4	8	2	5	0	1 43	0 53	0 45	1 63	4.25	CG
2296	30	2	4	1	5	0	1.22	0.04	0 03	0 01	1 41	CG
28016	179	3	3	0	0	0	3 05	2 47	4 21	0 0	10.46	HR
85552	83	11	129	79	1386	22	153.5	515 4	285.3	775.8	1734 2	BC

lution). To prove optimality required 44 nodes on a search tree, 253 calls to the LP solver and 14 calls to the preprocessor. 6,109 polyhedral cuts were generated in the process.

Problem AA1 required 4.01 hours to prove that the optimal solution value is 56137. A solution of 56138 was found at node one after only 6.3 minutes of computing time, with the optimal solution found at node 20. The linear programming relaxation has a solution value of 55535.4 (1.05% lower than the optimal solution). It required 90 nodes on the search tree to prove optimality. 3,787 polyhedral cuts were generated in the process.

The last of these problems, Problem AA4 has an optimal solution of 26374. It required 38.7 hours to prove optimality. The optimal solution was found at node 125. The linear programming relaxation has a solution value of 25877.6 (1.8% off of optimality). The integer feasible solutions found during this run had values 27080, 27030, 26993, 26707, 26453, 26448, 26402, and 26374. They were found within 53, 877, 1,423, 3,642, 7,224, 7,443, 7,602 seconds, respectively. It required 494 nodes on a search tree to prove optimality. 15,449 polyhedral cuts were generated in the process.

Table 7 Hard Set-Partitioning Problems: Problem Characteristics

PROBLEM	Cols	Rows	PCols	PRows	DENSITY
NW04	87482	36	46190	36	20 2
AA1	8904	823	7532	607	1.0
AA4	7195	426	6122	342	1 8

8. Conclusions

We conclude from examining the results of this extensive test effort that it is possible to solve very large set partitioning problems to proven optimality, even when additional side constraints that model the "true" problem that airline crew schedulers face are incorporated explicitly in the integer programming formulation. For the few instances when proving optimality requires substantial effort, our software package provided extraordinarily good solutions with worst case estimates on their correctness very early in the solution process.

Thus, by using the technology described above and solving *larger* set partitioning problems exactly in the process of determining less costly crew schedules than is done currently, the airline industry could see immediate and substantial dollar savings in their crew costs. More specifically, we think that it is possible, with to-

Table 8 Hard Set-Partitioning Problems: LP, IP and Heuristic Solution Values, and Time to Optimality

PROB NAME	LP VALUE	IP VALUE	HEUR SOL	Time to 2% of Opt	Time to 1% of Opt	Time to Optimality
NW4	16310 7	16862	19492	225	298	2642
AA1	55535 4	56137	failed	375	375	14441
AA4	25877 6	26374	27080	868	7443	139337

day's technology, to solve the *entire* problem as a *single* optimization problem. Column generation methodology similar to that employed for the traveling salesman problem (see Padberg and Rinaldi 1991) can be developed for the solution of large-scale crew scheduling problems as well. This allows very large integer programs to be handled in an iterative fashion that guarantees global convergence but does not require all columns to be generated and used in the solution process simultaneously. Future work will examine the efficacy of this approach.

Instead of directing resources towards efforts that provably reduce operating costs, during the past few years much of the industry's attention has focused on *revenue generation*—like yield management—that resulted in an enormous variety of fares for a given flight. More recently, due to outcries by travelers that passengers in neighboring seats were paying tremendously different fares for the same service on the same flight, fare structures have been simplified and the industry is now able to focus on other issues related to profitability and cost reduction. Increased competition induces greater willingness and the necessity to cut cost. We think that this is possible through improved planning; crew scheduling is a very important part of the airlines' business, and cutting cost here not only seems possible but it is also a problem that can be tackled *mathematically better* than done currently. In addition to cost savings to the airlines, greater job satisfaction of crew members results from improved planning since the crews spend more of their duty time flying and less time on the ground. There are other aspects of the industry, such as fleet assignment and maintenance scheduling, where combinatorial optimization software based on the mathematics of polyhedral theory is most likely to yield similar improvements as far the management of personnel, the planning of equipment utilization and maintenance and, most importantly, the reduction of operating cost are concerned.

We close by noting that much of this paper describes the general set partitioning problem with side constraints. Thus the solution techniques and therefore the package CREW_OPT can be applied to a wide variety of real-world problems different from crew scheduling. Crew scheduling was chosen because of the availability of data sets and the interest by the airline community. We would like to collect and distribute a large data set representative of problems having the general set partitioning structure. We therefore invite all readers having such problems to contact the authors.[1]

[1] This work would not have been possible without the availability of data from the airline industry and without many people within such corporations who provided us with the detailed background for this important problem Specifically, we wish to thank Lorraine Latosky and Fenton Hill of U S.Air, Jane Barrutt, Jeremy Schneider and Elroy Olson of Northwest Airlines, and Ranga Anbil of American Airlines for all of their assistance Convex Computer Corporation has graciously provided us with extensive machine time to develop and test the software package and we are most grateful for that assistance Finally, we wish to thank Greg Astfalk of Convex Computer Corporation not only for his untiring computer help and tutelage, but also for his enthusiasm and encouragement during this effort.

The authors were supported in part by grants from the Office of Naval Research (N00014-90-J-1324) and the Air Force Office of Scientific Research (F49620-90-C-0022).

8. Appendix

The data for the set partitioning problem from which Figure 1 is derived are given in *column-major* and *row-major* form in Table 9 which also gives the objective function coefficients which simply equal the number of ones in each column for this demonstration, e.g., column 1 of the problem has entries equal to one in rows 14, 26, 27 and 28 and zeros elsewhere and $c_1 = 4$ The *first* constraint of the problem is $x_9 + x_{18} = 1$, etc., and there are $m = 30$ rows and $n = 25$ columns The solution displayed in Figure 1 is a basic feasible solution to the linear programming relaxation of the corresponding set partitioning problem—and thus it is clear that no integer feasible solutions to this problem exist Indeed, the sample problem corresponds to a "block" of the linear programming solution to one of the problems in the test set and thus it was originally part of a much larger problem. How do you prove—using the constraint generation procedures described in this paper—that no integer solution exists? First, one runs through the clique detection routines of § 6 which find that no constraint of the type (1) is violated. This is an *exact* statement, i.e , the point $\bar{x} \in \mathfrak{R}^{25}$ displayed in Figure 1 satisfies *all* constraints (1) corresponding to cliques K of the intersection graph of Figure 1 Now running the odd cycle detection routine of § 6 the program CREW_OPT finds four odd cycles:

$$C_1 = \{4, 21, 18, 2, 23\}, \quad C_2 = \{4, 21, 18, 6, 22\},$$

$$C_3 = \{9, 17, 10, 24, 18\}, \quad C_4 = \{6, 24, 10, 11, 25\}$$

Only for the odd cycle C_3 is constraint (2) violated as such, while for the other odd cycles we get a nonnegative slack in (2). Lifting these "kernels" of facets of the corresponding set packing polytope P_I^A, the package CREW_OPT produces four *violated inequalities*.

$$x_2 + x_4 + x_{18} + x_{21} + 2x_{22} + x_{23} \leq 2,$$

$$x_4 + x_6 + x_{18} + x_{21} + 2x_{22} + x_{23} \leq 2,$$

$$x_9 + x_{10} + x_{17} + x_{18} + x_{24} \leq 2,$$

$$x_6 + x_{10} + x_{11} + x_{18} + x_{24} + x_{25} \leq 2,$$

which—incidentally—define facets of the corresponding polytope P_I^A. CREW_OPT then adds these four constraints, concludes that the resulting larger linear program has no feasible solution and stops with a corresponding message. The time to do all of this on a VAX 8800 is less than one second! We recommend that you do it by hand to see how the theory works and to really appreciate the help the computer provides with respect to these calculations. If you do it by hand, you will find two more violated lifted odd-cycle constraints, namely

$$x_6 + x_8 + x_{15} + x_{18} + x_{19} + x_{20} + x_{24} + x_{25} \leq 3,$$

$$x_9 + x_{10} + x_{11} + x_{17} + x_{18} \leq 2,$$

which CREW_OPT does not find because we programmed the odd cycle detection routines to select another node as "root node" as soon as some violated lifted odd cycle constraint at a given root node has been found by the procedure. The reason for doing so is clear if you think in terms of "large" graphs. *locally* we have found a violated polyhedral cut and now we wish to go into some other "distant" part of the graph and—hopefully—find a violated constraint there, again locally.

In order to construct the output for the example of this appendix, we had to "shut off" the preprocessor of the package CREW_OPT because if we use all "cannons" at our disposal to "finish off" this tiny problem, i.e., if we invoke all components of CREW_OPT, integer infeasibility of the sample problem is detected already in the preprocessor. You can verify that one also by hand calculation using the routines described in § 4, see especially the *merging* of two columns into one.

Table 9 Data for Sample Problem

$m = 30, n = 25$

Matrix A columnwise. Columns 1, . . . , 25 in sequential order:
{14, 26, 27, 28}, {7, 10, 21}, {13, 25}, {19, 20}, {12}, {6, 8, 22}, {27, 29}, {23, 24}, {1, 2}, {3, 4, 15}, {9, 15, 16}, {16}, {28, 30}, {29, 30}, {18, 24}, {25, 26}, {2, 3}, {1, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}, {5, 17}, {17, 23}, {11, 19}, {10, 11, 19, 20, 21, 22}, {20, 21, 22}, {4, 5, 6, 7}, {8, 9, 18}

Matrix A rowwise. Rows 1, . . . , 30 in sequential order:
{9, 18}, {9, 17}, {10, 17}, {10, 24}, {18, 19, 24}, {6, 18, 24}, {2, 18, 24}, {6, 18, 25}, {11, 18, 25}, {2, 18, 22}, {18, 21, 22}, {5, 18}, {3, 18}, {1, 18}, {10, 11}, {11, 12}, {19, 20}, {15, 25}, {4, 21, 22}, {4, 22, 23}, {2, 22, 23}, {6, 22, 23}, {8, 20}, {8, 15}, {3, 16}, {1, 16}, {1, 7}, {1, 13}, {7, 14}, 13, 14}

Cost of columns 1, . . . , 25:
4, 3, 2, 2, 1, 3, 2, 2, 2, 3, 3, 1, 2, 2, 2, 2, 2, 11, 2, 2, 2, 6, 3, 4, 3

Just to satisfy your curiosity (and our own) we also ran CREW_OPT to solve the associated maximum cardinality stable set problem on the graph displayed in Figure 1, see § 2 for the definition of a stable set. In this case, we wish to maximize the objective function $\sum_{j=1}^{25} x_j$ over the associated set packing polytope of P_I^A. The problem is brought into the form of (SPP) by adding 30 slack variables with zero cost, except that we set $c_{26} = 0.1$ in order to avoid an alternative integer optimum solution. The linear programming relaxation of the problem gives for the first 25 variables, $x_j = \frac{1}{2}$ for

$$j \in \{2, 4, 6, 8, 9, 10, 11, 12, 15, 17, 19, 20, 21, 23, 24, 25\},$$

$x_j = 1$ for $j \in \{1, 3, 5, 14\}$, and $x_j = 0$ otherwise as an optimal solution with an objective function of 12. If we run CREW_OPT as we have designed it, the heuristic finds the integer solution $x_j = 1$ for

$$j \in \{1, 2, 3, 4, 5, 6, 9, 10, 12, 14, 15, 20\},$$

$x_j = 0$ otherwise and—since its objective function is 12 and thus equal to the linear programming upper bound—the software stops since it has obtained and proved optimality of a solution to the problem. So we shut off the heuristic and ran the problem again. Drawing the intersection graph G_F corresponding to the LP optimal solution given above you will find that all clique-constraints (1) are satisfied and there are exactly two violated lifted odd cycles constraints. CREW_OPT produced the following constraints:

$$x_6 + x_{10} + x_{11} + x_{18} + x_{24} + x_{25} \leq 2,$$

$$x_6 + x_8 + x_{15} + x_{18} + x_{19} + x_{20} + x_{24} + x_{25} \leq 3.$$

When added to the linear program, the reoptimization yields the above 0-1 solution found by the heuristic and stops. As we did not want to spend time preprocessing the problem either, we had shut off that part of CREW_OPT as well. Total computation time on a VAX 8800 about half a second (0.65 secs) of CPU time!

References

Anbil, R., E. Gelman, B. Patty and R. Tanga, "Recent Advances in Crew Pairing Optimization at American Airlines," *Interfaces*, 21 (1991), 62–74.

Balas, E. and M. Padberg, "Set Partitioning: A Survey," *SIAM Review*, 18 (1976), 710–760.

Barahona, M., M. Grötschel, M. Junger and G. Reinelt, "An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design," *Operations Res.*, 36 (1988), 493–513.

Barnhart, C., E. L. Johnson, R. Anbil and L. Hatay, "A Column Generation Technique for the Long-haul Crew Assignment Problem," Report #COC-91-01, School of Industrial and Systems Engineering, Georgia Institute of Technology, 1991.

Barutt, J. and T. Hull, "Airline Crew Scheduling: Supercomputers and Algorithms," *SIAM News*, 23 (1990), November.

Bixby, R. E., "Implementing the Simplex Methods, Part I, Introduction; Part II, The Initial Basis," TR 90-32 Mathematical Sciences, Rice University, Houston, TX, 1990.

—— and E. Boyd, "Using the CPLEX Callable Library," Manual dis-

tributed by Cplex Optimization Inc , 7710-T Cherry Park, Houston, TX, 1990.

Bornemann, D. R , "The Evolution of Airline Crew Pairing Optimization," AGIFORS Crew Management Study Group Proceedings, Paris, May 1982.

Cannon, T L. and K. L. Hoffman, "Large-scale 0-1 Linear Programming on Distributed Workstations," *Annals of Operations Res.*, 22 (1990), 181–217.

Crowder, H , E. L Johnson and M. Padberg, "Solving Large Scale Zero-one Linear Programming Problems," *Operations Res.*, 31 (1983), 803–834.

Cornuejols, G. and A. Sassano, "On the 0,1 Facets of the Set Covering Polytope," *Mathematical Programming*, 43 (1989), 45–56.

Desrochers, M. and F. Soumis, "A Column Generation Approach to the Urban Transit Crew Scheduling Problem," *Transportation Sci.*, 23 (1989), 1–13.

Gerbracht, R., "A New Algorithm for Very Large Crew Pairing Problems," 18th AGIFORS Symposium, Vancouver, British Columbia, Canada, 1978

Gershkoff, I., "Optimizing Flight Crew Schedules," *Interfaces*, 19 (1989), 29–43.

Grötschel, M. and C. L. Monma, "Integer Polyhedra Arising from Certain Network Design Problems with Connectivity Constraints," *SIAM J. Discrete Mathematics*, 3 (1990), 502–523.

——, M. Jünger and G. Reinelt, "Optimal Control of Plotting and Drilling Machines. A Case Study," Report No. 184, Universität Augsburg, 1989.

Hoffman, K. L. and M. Padberg, "LP-based Combinatorial Problem Solving," *Annals of Operations Res.*, 4 (1985), 145–194.

—— and ——, "Techniques for Improving the LP-representation of Zero-one Linear Programming Problems," *ORSA J Computing*, 3 (1991), 121–134

Lavoie, S , M. Minoux and E. Odier, "A New Approach for Crew Pairing Problems by Column Generation with Application to Air Transportation," *EJOR*, 35 (1988), 45–58.

Marsten, R E and F Shepardson, "Exact Solution of Crew Problems using the Set Partitioning Mode: Recent Successful Applications," *Networks*, 11 (1981), 165–177.

Padberg, M., "Essays in Integer Programming," Ph.D. thesis, GSIA, Carnegie-Mellon University, Pittsburgh, PA, 1971

——, "On the Facial Structure of Set Packing Polyhedra," *Mathematical Programming*, 5 (1973), 199–215.

——, "A Note on Zero-one Programming," *Operations Res.*, 23 (1975), 833–837.

——, "Covering, Packing and Knapsack Problems," *Annals of Discrete Mathematics*, 4 (1979), 265–287.

—— and G Rinaldi, "An Efficient Algorithm for the Minimum Capacity Cut Problem," *Mathematical Programming*, 47 (1990), 19–36.

—— and ——, "A Branch-and-Cut Algorithm for the Solution of Large-scale Traveling Salesman Problems," *SIAM Rev.*, 33 (1991), 60–100

Rubin, J , "A Technique for the Solution of Massive Set Covering Problems, with Applications to Airline Crew Scheduling," *Transportation Sci* , 7 (1973), 34–48.

Sassano, A., "On the Facial Structure of the Set Covering Polytope," *Mathematical Programming*, 44 (1989), 181–202