PARALLEL BRANCH AND CUT FOR SET PARTITIONING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Márta Eső

January 1999

PARALLEL BRANCH AND CUT FOR SET PARTITIONING

Márta Eső, Ph.D.

Cornell University 1999

This thesis investigates three major steps in the solution process of the Set Partitioning Problem (SPP): problem size reduction techniques, LP-based feasible solution heuristics and Branch-and-Cut solution methodology. SPPs arise in many practical applications (airline crew scheduling, vehicle routing, circuit partitioning). Theoretical aspects of this problem have been studied for a long time, but only recently have computers become powerful enough to attack practical instances.

Problem size reduction methods reduce the set of variables and/or constraints through logical implications without eliminating optimal solutions to the original problem. We show that the reduction operations well-known in the literature, applied in any order to an SPP instance until no further reduction is possible, always produce the same reduced problem.

Finding good feasible solutions is essential for upper bounding in a Branch-and-Cut framework. Our LP-based feasible solution heuristic iterates a heuristic fixing phase with reduced cost fixing to improve the quality of the feasible solution. Our heuristic procedure is somewhat more conservative than earlier approaches in that it eliminates unnecessary variables instead of forcing variables into the solution.

Our parallel Branch-and-Cut procedure was implemented using the COMPSys framework. COMPSys provides the user with the necessary infrastructure to implement an efficient Branch-and-Cut application by handling tasks common for parallel Branch-and-Cut (search tree management, message passing, LP interface). To interface with COMPSys we implemented procedures particular to the SPP. We generate cuts both algorithmically and manually through a graphical user interface.

Our experiments were carried out on the IBM RS/6000 Scalable POWERparallel System of the Cornell Theory Center. Our test set included problems from airline crew scheduling and vehicle routing applications. Our computational results demonstrate our implementation to be an effective approach for solving SPPs of moderately large size.

Biographical Sketch

Márta Eső was born on April 5, 1968 in Szombathely, Hungary. She completed her undergraduate studies at the Eötvös Loránd University, Budapest, Hungary in June 1991, receiving both her Master of Science degree in Mathematics and her certificate for high school teaching. She entered the Ph.D. program in the School of Operations Research and Industrial Engineering at Cornell University in the Fall of 1991, where she received a Master of Science degree in Operations Research in January 1995. She participated in a work-study program at the IBM T.J. Watson Research Center, Yorktown Heights, from September 1997 to December 1998.

... to my Father's memory...

Acknowledgements

I would like to express my gratitude to my advisor Professor Leslie E. Trotter, Jr. for his guidance. I would also like to thank Professor Éva Tardos who served as my advisor during my first year at Cornell, and Professors Ronitt Rubinfeld, Paul Pedersen and Tapan Mitra for serving on my special committee. I am indebted to the faculty and staff of the School of Operations Research and Industrial Engineering at Cornell University for providing an excellent graduate program and a productive research environment.

We have worked very closely on the COMPSys project with Laci Ladányi, Ted Ralphs, Greta Pangborn, and Leo Kopman. Our research was made possible through the generous support of the Cornell Theory Center and the U.S. National Science Foundation. Edoardo Amaldi, Oktay Günlük, and Jean-François Pusztaszeri gave valuable suggestions during the development phase.

I am grateful to Dr. William Pulleyblank and all those at the Department of Mathematical Sciences at the IBM T.J. Watson Research Center who made it possible for me to participate in their work-study program. It was a very valuable experience to see optimization theory applied in practice. I would specifically like to thank Ranga Anbil, Francisco Barahona, and Jane Snowdon for their friendship and encouragement.

I thank all the graduate students at the OR&IE department and the many friends in Ithaca who made my stay at Cornell so enjoyable.

Most importantly, I would like to thank my family for their love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Set Partitioning Problem

The *Set Partitioning Problem* (SPP) in its general form can be presented as follows. Given a ground set S of m objects and a collection of subsets of S $(S_1, \ldots, S_n)$ with associated costs $c(S_j)$, $1 \leq j \leq n$, select some subsets of minimum (or maximum) total cost so that the selected subsets are disjoint and their union is the ground set. In other words, choose a minimum (or maximum) cost partitioning of the ground set.

A wide variety of practical applications have been modeled as SPPs during the past 50 years, including *crew scheduling* ([FR87], [Ger89], [AGPT91], [HP93]) *vehicle routing* ([BQ64], [Chr85], [BGKK97]), *political districting* ([GN70]), and *circuit partitioning* ([ECTA96]) just to name a few. References to further applications can be found in [GN72] (Chapter 8), [BP76], and [EDM90].

The most well studied and one of the earliest applications is *Airline Crew Scheduling.* The importance of crew scheduling in the airline industry is due to the fact that crew costs are exceeded only by the cost of fuel, thus small improvements in the solution translate to large dollar savings ([AGPT91]). Crew scheduling is a major step in schedule planning; it comes after timetables are created and aircraft are already assigned to the flights. The goal of crew scheduling is to assign crew members to the flights as cheaply as possible while a complex set of FAA regulations, union requirements and other internal operational rules are met. Modeled as a set partitioning problem, the ground set will be the collection of flights that need to be covered, while the subsets correspond to sequences of flights that a crew can operate (so called *pairings*). Constructing pairings is a complex process since the *legality* of the pairings (compliance with the rules and regulations) must be ensured. The cost of a subset reflects both crew compensation and penalties for undesired events like tight connections or *deadheading* (crew members are passengers on a flight). The SPP model itself does not capture requirements like crew availability at different stations; these requirements are usually added in the form of side constraints.

In the *Vehicle Routing Problem (VRP)* customer demands need to be served by a fleet of vehicles that depart from and return to the same location (the depot) so that the total cost incurred on the trips (e.g., the distance traveled by the vehicles) is as small as possible. Each customer must be serviced by exactly one vehicle and the vehicles have finite capacities. In the set partitioning model the customers will be the elements of the ground set, and feasible routes for individual vehicles form the subsets. The cost of a subset is the cost of the corresponding route. The use of the set partitioning formulation for solving the *general* VRP is not practical since too many subsets need to be enumerated and just to compute the cost of a subset is a hard problem in itself (requires solving a *Traveling Salesman Problem (TSP)* instance) [Chr85]. However, in practice very often there are additional requirements (e.g., rest rules for the drivers of the vehicles [BGKK97]) that restrict the number of subsets and the order in which the customers can be served within a route. While these requirements would need to be added as side constraints in the traditional formulation, they can be accommodated here by generating only those subsets that obey them.

Another early application of the SPP is *political districting.* A state is composed of small population units (e.g., counties, census tracts) that need to be grouped into political districts so that certain criteria on the population, contiguity and shape of the districts are met and the grouping is as acceptable as possible. Modeled as a set partitioning problem, the elements of the ground set correspond to the population units, and the subsets to proposed districts. The cost of a subset measures the undesirability of the corresponding district, and the solution to the SPP will provide the least undesirable way of partitioning the state into districts. A side constraint specifying the number of districts required is also added.

The *circuit partitioning problem* is the first step in the physical design stage of electronic circuit design. Physical design is preceded by logical design, where the components of the circuit and the interconnections between them are planned on paper without considering the actual placement of the components. Then, in the physical design stage, the plan is first divided into subcircuits (this is circuit partitioning) then the components are placed within these partitions and a routing between the subcircuits is determined. In a feasible partitioning the total size of components within the subcircuits and the pins required to connect the partitions must stay within specified bounds. The quality of a partitioning is hard to measure; balancing wire congestion and minimizing the number of connections between the subcircuits are commonly used. In the set partitioning model the ground set is comprised of the components and subsets correspond to subcircuits that satisfy the above requirements. The objective is to obtain the highest quality partitioning.

In all the applications discussed above significant effort must be spent on generating the subsets and computing their costs. The number of feasible subsets is exponential in the size of the ground set in general which makes listing all the subsets at once impractical. To overcome this problem a “good” collection of subsets is chosen first and then solving the SPP restricted to the current subsets and incorporating new “improving” subsets are iterated. Real-world applications do not always require optimal solutions, thus the iterative process can be aborted as soon as an acceptable quality solution is found. In this dissertation we will focus on how to solve SPPs; this task could be considered as solving “snapshots” of the above iterative process.

1.2 Set Packing, Covering and Partitioning

If we relax the requirement in the SPP that the chosen subsets be disjoint, the problem becomes the *Set Covering Problem* (SC). On the other hand, if the chosen subsets must be disjoint but their union may be a proper subset of the ground set, we have the *Set Packing Problem* (SP). Note that the objective is to minimize in the Set Covering and to maximize in the Set Packing Problem. Although both of these problems are relaxations of the Set Partitioning Problem, SP and SPP are equivalent, while SC is easier than SPP in some sense. As Balas and Padberg point it out in [BP76] this can be intuitively explained by observing that SPP and SP are "tightly constrained" (only one of the many subsets that contain an object may be chosen in a solution) while SC is "loosely constrained" (several subsets containing the same object can be chosen).

The above problems can be formulated as integer programming models by assigning decision variables $x_1, \ldots, x_n$ to the subsets indicating which subsets are chosen ($x_j = 1$ if S_j is chosen, 0 otherwise). The characteristic vectors of the subsets ($0-1$ vectors of length m that show which objects of the ground set are contained in a subset) are arranged into columns of a matrix A. Figure 1.1 gives the formulation of the three problems.

SPP and SP are equivalent in the sense that each can be written in the other's form so that the optimal solutions for the original and transformed problems will be the same. To see that any SP problem is an SPP, simply add slack variables (with zero objective function coefficients) to the constraints. Since the coefficient

$$(SPP)\qquad \begin{array}{rrcl} \min & c^Tx & & \\ & Ax & = & \mathbf{1}_m \\ & x & \in & \{0,1\}^n \end{array}$$

$$(SC)\qquad \begin{array}{rrcl} \min & c^Tx & & \\ & Ax & \geq & \mathbf{1}_m \\ & x & \in & \{0,1\}^n \end{array} \qquad\qquad (SP)\qquad \begin{array}{rrcl} \max & c^Tx & & \\ & Ax & \leq & \mathbf{1}_m \\ & x & \in & \{0,1\}^n \end{array}$$

Figure 1.1: Integer Programming formulation of the Set Partitioning, Covering and Packing Problems

matrix is 0-1 and for any feasible solution x to (SP) the left hand side is either 0 or 1, it is true that the slacks can take only values 0 or 1. So (SP) can be written as an SPP of the following form:

$$(SP)\qquad \begin{array}{rrcrcl} \max & c^Tx & + & \mathbf{0}_m^T s & & \\ & Ax & + & I_m s & = & \mathbf{1}_m \\ & x & & & \in & \{0,1\}^n \\ & & & s & \in & \{0,1\}^m \end{array}$$

On the other hand, an SPP can be written as an SP problem. Switch the min to a max and add artificial variables ($y_i \geq 0$) to the constraints and charge a penalty if they are at nonzero level (θy_i). Notice that if (SPP) is feasible then $y_i = 0$ in any optimal solution to the new formulation as long as θ is "large enough", that is, at least as large as the cost of any feasible solution to the original SPP. $\sum_{j=1}^n c_j$ is an obvious upper bound on this number. If (SPP) is not feasible then the same large

θ will force the cost of any feasible solution to the SP formulation to be at least θ.

$$(SPP) \qquad \begin{array}{rrcrcl} \max & -c^T x & - & \theta \mathbf{1}_m^T y & & \\ & Ax & + & I_m y & = & \mathbf{1}_m \\ & x & & & \in & \{0,1\}^n \\ & & & y & \geq & 0 \end{array}$$

Substituting $\mathbf{1}_m - Ax$ in the objective function for y we get $-c^T x - \theta \mathbf{1}_m^T y = -c^T x - \theta \mathbf{1}_m^T(\mathbf{1}_m - Ax) = (\theta \mathbf{1}_m^T A - c^T)x - \theta m$. We further relax the problem by dropping y from the constraints as well, thus increasing the size of the feasible region. But the set of optimal solutions will be unchanged since it is too expensive not to satisfy the constraints with equality. Thus we have obtained an SP form for the SPP.

$$(SPP') \qquad \begin{array}{rrcl} \max & -\theta m + (\theta \mathbf{1}_m^T A - c^T)x & & \\ & Ax & \leq & \mathbf{1}_m \\ & x & \in & \{0,1\}^n \end{array}$$

The SPP can be converted into an SC problem using the same logic. However, an SC problem cannot be written in a set partitioning form.

1.3 The Stable Set Problem and Set Partitioning

Consider the finite undirected graph $G = (V, E)$. A *stable set* (independent set, vertex or node packing) is an independent subset of the nodes, i.e., a set of nodes so that no two are connected by an edge. A *maximum stable set* is a stable set of maximum cardinality, its size is denoted by $\alpha(G)$. Assigning weights w to the nodes the weight of a subset of the nodes is simply the sum of the weights of the nodes. A *maximum weight stable set* is a stable set of largest weight ($\alpha_w(G)$).

The Maximum Weight Stable Set Problem (MWSSP) can be formulated as an Integer Program by assigning decision variables to the nodes of the graph:

$$\begin{aligned} \alpha_w(G) = \max \quad & w^T x \\ A_G^T x &\leq \mathbf{1}_{|E|} \\ x &\in \{0,1\}^{|V|} \end{aligned}$$

where A_G is the node-edge incidence matrix of G, that is, a $|V| \times |E|$ matrix of 0's and 1's where each column contains exactly two 1's in the rows that correspond to the endpoints of the column's edge.

Observe that this is a set packing problem with a special matrix (the transpose of the node-edge incidence matrix of a graph). On the other hand the Set Packing Problem can be viewed as an MWSSP on a special graph derived from the problem matrix of the SP as we will see below, thus SP and MWSSP are equivalent.

The notion of the *intersection graph* (or conflict graph) of an SP (SPP or SC) was first introduced by Edmonds ([Edm62]). The intersection graph corresponding to an $m \times n$ 0-1 matrix A is an undirected graph $G_A(V, E)$ where the nodes are assigned to the columns of A and edges join nodes whose corresponding columns are nonorthogonal.

It is clear that columns corresponding to variables at level 1 in a feasible solution to (SP) are pairwise orthogonal, thus the corresponding nodes in the intersection graph form a stable set. The converse of this statement is true as well, a stable set in the intersection graph corresponds to a feasible solution to (SP). Thus (SP) is equivalent to the MWSSP on the intersection graph (A_{G_A} denotes the node-edge incidence matrix of G_A; A_{G_A} has n rows and $O(n^2)$ columns):

$$\begin{array}{rrcl} \max & c^T x & & \\ & A_{G_A}^T x & \leq & \mathbf{1}_{|E|} \\ & x & \in & \{0,1\}^n \end{array}$$

Therefore, because of the equivalence of optimal solutions for (SPP) and (SP) and the way an SPP can be converted into an SP, we could solve an MWSSP instead of (SPP), only the objective function needs to be modified in the above formulation to incorporate a penalty for not meeting the inequalities with equality. Note that even though the coefficient matrix in this equivalent MWSS formulation has a nice structure, it is much larger than the original matrix A. Also, as Balas and Padberg point it out ([BP76]), the LP relaxation to the above MWSSP is much weaker than the one to (SP). Thus solving an MWSSP instead of an SPP is not a realistic alternative. However, the insight gained from the intersection graphs and graph theoretical results originally derived for the MWSSP can be utilized when solving SPPs (see Section 2.3).

1.4 Complexity and approximability

All the problems discussed in the previous sections (SPP, SP, SC and MWSSP) are NP-complete in general ([LK79], [GJ79]). Some special cases that can be solved in polynomial time will be mentioned in Section 2.3.

Before comparing the approximability of these problems let us define a special case of set covering, the *Minimum Weight Vertex Cover Problem (MWVCP)*. A *vertex cover* in a finite undirected graph is a subset of the nodes so that every edge is adjacent to at least one of these nodes. A *minimum weight vertex cover* is a vertex cover of smallest weight. MWVCP is a special case of SC where the problem matrix is the node-edge incidence matrix of the graph. Observe that stable sets and vertex covers are each other's complements; that is, a subset of the nodes is independent if and only if the nodes *not* in the subset form a vertex cover, and vice versa. From this it trivially follows that the MWVCP is also NP-complete. Note that while MWSSP on the intersection graph is equivalent to SP, the same is not true for MWVCP and SC.

While all the above mentioned problems belong to the same complexity class, they differ greatly in approximability. Because of the equivalence of SP and MWSSP we will compare here only MWVCP, SC and MWSSP.

An important approximability class is MAX-SNP (introduced in [PY91]; for a comprehensive survey see [Shm95]; for additional discussion of packing and covering problems see [Hoc95]). The problems in MAX-SNP turn out to be exactly those that can be approximated within a constant factor (there is a polynomial time algorithm that provides a solution with objective value within a constant factor of the optimum). MAX-SNP-hard problems do not have polynomial approximation schemes (families of polynomial time algorithms that approximate the optimal solution arbitrary closely) unless P = NP.

MWVCP is in MAX-SNP since it can be approximated within a factor of 2 (solve the LP relaxation and round). SC is harder to approximate, a greedy approach yields an $O(\log_2 n)$ approximation and it is proven that we cannot do significantly better ([Shm95]). The MWSSP cannot even be approximated within a logarithmic factor; it is shown that no approximation factor of the form $n^{\frac{1}{2}-\epsilon}$, $\epsilon > 0$ can be guaranteed unless P = NP ([Shm95]). While SC and MWSSP are not in MAX-SNP, they are MAX-SNP-hard.

1.5 Outline of the thesis

This thesis investigates three major steps in the solution process of the Set Partitioning Problem: problem size reduction techniques, LP-based feasible solution heuristics and Branch-and-Cut solution methodology. The Set Partitioning Problem and its close relatives, the Set Packing and Set Covering Problems arise in many practical applications. Theoretical aspects of these problems have been studied for a long time, but only recently have computers become powerful enough to attack practical instances. However, there are still many advances to be made on the implementation side.

Chapter 2 reviews some important aspects of polyhedral combinatorial optimization. We outline the two classic methods of solving integer programming models: the Cutting Plane and Branch-and-Bound algorithms; both of which rely on LP relaxations. The Branch-and-Cut algorithm combines the two into a more powerful method by incorporating cutting planes into the Branch-and-Bound framework. We introduce COMPSys, a generic parallel Branch-and-Cut framework that we used for implementing a Branch-and-Cut algorithm for the Set Partitioning Problem. In the remainder of the chapter we focus on the polyhedra associated with the three problems. In particular, we summarize what is known about generating facet defining valid inequalities for these problems (and how to lift them) in theory. Some of these inequality classes will reappear in Chapter 5 where we discuss our Branch-and-Cut implementation.

Chapter 3 discusses methods that, given a Set Partitioning Problem, reduce the set of variables and/or constraints through logical implications without eliminating optimal solutions to the original problem. Problem size reduction is useful not only for the original problems (practical problem instances very often contain a large amount of redundant information due to the way they are generated) but it can be used to propagate the effects of setting some variables to their lower or upper bounds (as it does in our Feasible Solution Heuristic and after reduced cost fixing in Branch-and-Cut). These reduction operations are interesting from the theoretical point of view as well; we show that the six reduction operations introduced in this chapter, applied in any order to a Set Partitioning instance until no further reduction is possible, will always produce the same reduced problem. Our implementation contains a module for each of the six reduction operations, these modules can be combined into strategies keeping different goals in mind.

Finding good feasible solutions for Set Partitioning Problems is notoriously difficult since the problem is usually very tightly constrained. In Chapter 4 we will discuss LP relaxation based feasible solution heuristics, first in general, then for our application in detail. Our implementation iterates these heuristics and reduced cost fixing to improve the quality of the feasible solution, which enables us to prove optimality of the feasible solutions found for many of the problems available in the literature. LP relaxation based feasible solution heuristics iterate setting variables to their upper and lower bounds, propagating the effects of these settings and re-solving the LP relaxations. Traditionally, setting variables to their upper bounds is favored since this reduces the problem size much faster (but can lead to a quick loss of feasibility). We take a somewhat more conservative approach and set insignificant variables to their lower bounds instead. In particular, we apply a heuristic procedure called "follow-on" fixing that is based on an idea originating in airline crew scheduling applications. The overall efficiency of an implementation like this depends on that of the problem size reduction and of the LP re-optimization. The latter is a nontrivial task; we discuss the difficulties encountered.

Chapter 5 describes our parallel Branch-and-Cut implementation in detail. The COMPSys framework handles all the tasks which are common for parallel Branch-and-Cut implementations (e.g., search tree and cut pool management, inter-process communication and LP solving), "all we had to do" was to implement some problem-specific user functions. We discuss the most interesting parts of this implementation in detail: preprocessing of the problem, logical fixing, cut generation (both algorithmically and manually) and lifting and choosing branching candidates (both variables and constraints) for strong branching.

Chapter 6 demonstrates a novel feature of the framework: cut generation "by hand" using a Graphical User Interface. This feature allows us to examine the current solution (in graphical form) and enter any inequalities through the GUI; the inequalities will be incorporated into the formulation if they are violated.

Our final computational experiments were carried out on the IBM RS/6000 Scalable POWERparallel System (SP) of the Cornell Theory Center. Appendix A contains details about the computing environment, as well as the test set. Our test problems consisted of four distinct sets of SPP models, two from airline crew scheduling and two sets of vehicle routing models.

In general our computational results demonstrate that Set Partitioning models of moderately large size can be solved to optimality within reasonable computation time limits. Our results for all three phases of the solution procedure compare favorably with other computational results in the literature.

1.6 Definitions and notation

All vectors are assumed to be column vectors. Constant vectors are denoted by bold letters, with their length in the subscript, e.g., $\mathbf{1}_m = (1, \ldots, 1)$. The matrix A is a 0-1 matrix of size $m \times n$; columns of A are denoted by a_j, rows by a^i, and entries by a_{ij}. The index sets of rows and columns are denoted by $M = \{1, \ldots, m\}$ and $N = \{1, \ldots, n\}$, respectively. The set of columns with a nonzero entry in row i is called the *support* of row i and it is denoted by $N^i = \{j \in N \mid a_{ij} = 1\}$, the set of rows intersecting column j is $M_j = \{i \in M \mid a_{ij} = 1\}$. (The index i usually runs through M, j through N.) c is a length n vector of integers unless otherwise specified. x and y are always vectors of variables. I_k is the $k \times k$ identity matrix.

Chapter 2

Background

2.1 Integer Programming and polytopes

Consider the Integer Program

$$
(IP) \qquad \begin{array}{rcl} \min & c^T x & \\ & Ax & R \quad b \\ & x & \in \quad \{0,1\}^n \end{array}
$$

where R is a length m array of relations ($\leq$, $=$ or $\geq$), $A \in \Re^{m \times n}$, $b \in \Re^m$, $c \in \Re^n$. Many Combinatorial Optimization problems can be formulated this way, including all the problems defined in the Introduction. In our examples the problem matrix A is 0-1 and the right hand side b is a vector of all 1's. However, the problem is NP-complete even for these special cases, so no polynomial time (in the size of the input data) algorithm is expected to be found for solving (IP). Note that here we are interested in methods for finding an optimal solution, while in applications near-optimal solutions obtained by *heuristics* might be acceptable.

In the rest of the section we review some basic facts about Integer Programming; standard references on this topic are [Sch86] and [NW88].

The optimal value of (IP) is denoted by z^*, an optimal solution by x^*. A binary vector $\bar{x}$ is called *feasible* if it satisfies the constraints $Ax \ R \ b$; in this case its objective value $\bar{z}$ provides an *upper bound* on z^*. The convex hull of all the feasible solutions to (IP) is $\mathcal{P}_{IP}$:

$$\mathcal{P}_{IP} = \text{conv}\{x \in \{0,1\}^n \mid Ax \ R \ b\},$$

which is a *polytope* by Weyl's theorem; that is, $\mathcal{P}_{IP}$ is the intersection of finitely many halfspaces:

$$\mathcal{P}_{IP} = \{x \in \Re^n \mid Hx \leq h\}.$$

If A and b are rational (integral) then so are H and h. If the above linear system were known, optimization over $\mathcal{P}_{IP}$ would simply mean solving a Linear Program (which can be done in polynomial time). Since (IP) is NP-complete in general, the complete description of $\mathcal{P}_{IP}$ with a linear system (H, h) is out of reach. We would be especially interested in *minimal* descriptions where no constraint can be expressed as a nonnegative linear combination of others. If the polytope is *full dimensional* the minimal description is essentially unique. However, even a minimal system may contain exponentially many inequalities.

Given any polytope $\mathcal{P}$, a hyperplane bounding the halfspace defined by a linear constraint is a *supporting hyperplane* if the halfspace contains the polytope and has a nonempty intersection with it. The intersection of a supporting hyperplane and

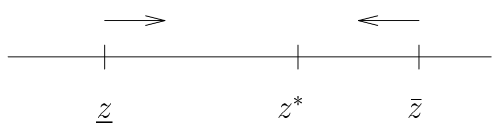


Figure 2.1: Lower and upper bound and the optimal solution

$\mathcal{P}$ is a *face* of the polytope. The $dim(\mathcal{P}) - 1$ dimensional faces are called *facets*. If a supporting hyperplane intersects the polytope in a facet then the corresponding linear constraint is said to be *facet defining*. The inequalities in a minimal linear system (H, h) are facet defining. The search for linear inequalities describing the IP polytope will be discussed in more detail in Sections 2.1.1 and 2.3.

A *relaxation* of (IP) is a problem whose feasible region contains all feasible solutions to (IP); that is, it contains $\mathcal{P}_{IP}$. Here we will consider only *LP relaxations*; that is, problems that are themselves Linear Programs (can be described with linear constraints and thus their feasible regions are polyhedra $\mathcal{P}_{LP}$). LP relaxations are useful since efficient algorithms exist to solve them and it is relatively easy to reoptimize after small changes in the formulation. Optimizing the same objective function $c^T x$ over the LP relaxation provides a lower bound $\underline{z}$ on z^*. A trivial LP relaxation to (IP) is to replace $x \in \{0, 1\}^n$ by $\mathbf{0}_n \leq x \leq \mathbf{1}_n$.

Methods for solving Integer Programs try to "close in" on the optimal solution from one or both sides by generating better and better feasible solutions that provide upper bounds or/and stronger and stronger LP relaxations that increase the lower bound; see Figure 2.1. The *integrality gap* $(\bar{z} - \underline{z})/\underline{z}$ measures how far the two bounds are from each other. When the optimum is known we can compute the *optimality gap* between an upper bound and the optimum as $(\bar{z} - z^*)/z^*$.

Two traditional ways to solve general IPs are Branch-and-Bound (B&B) and Cutting Plane methods. Both rely on relaxations, but as we will see, they are fundamentally different. The two are combined into a third, more powerful method called Branch-and-Cut.

2.1.1 Cutting plane methods

Cutting plane methods try to approximate the IP polytope from outside in a neighborhood of the optimal solution. They start from any LP relaxation of (IP). At each iteration the current LP relaxation is solved to optimality and then halfspaces that contain the entire IP polytope $\mathcal{P}_{IP}$ but not the optimal extreme point(s) of $\mathcal{P}_{LP}$ are identified and the corresponding linear constraints, or *cuts* are added to the LP relaxation. This is repeated until the optimal solution of the LP relaxation becomes integral (binary).

This procedure can be perceived as cutting off "corners" of the enclosing LP polytope until an optimal corner of the IP polytope surfaces. The identified linear constraints are not satisfied by optimal extreme points of $\mathcal{P}_{LP}$, thus they are called *violated* cuts or inequalities. From the IP's point of view these linear constraints are *valid* since they do not cut into $\mathcal{P}_{IP}$. The method of finding violated valid inequalities is called *cut generation* or *separation* (since the optimal corner of $\mathcal{P}_{LP}$ is separated from $\mathcal{P}_{IP}$). Generating *facet defining* inequalities is preferred since these are part of a minimal description of $\mathcal{P}_{IP}$. Intuitively, cutting off the optimal corner of the LP polytope with a facet defining inequality ensures that no more cuts will be necessary in the direction of the inequality's norm.

There are two important questions that must be asked here: *(i)* can we devise separation algorithms that are efficient both in theory and practice and *(ii)* can we ensure finite convergence of the cutting plane algorithm. In the textbook approach (due to Gomory) the violated inequalities are derived from certain rows of the current simplex tableau in polynomial time and the method is proved to converge in a finite number of iterations. However, it is widely believed that Gomory cuts are not effective in practice. For IP's with given structure (like those discussed in the Introduction) generating problem class specific families of cuts (preferably facet defining) may be more effective than generating general cuts. The drawbacks of this approach for specific problems are that not all families of valid (facet defining) inequalities might be known, and even if a family of cuts is known to be valid, it might be a hard problem in and of itself to separate for it. We will discuss problem specific cuts for the Set Packing and Covering polytopes in Section 2.3.

2.1.2 Branch-and-Bound

Branch-and-Bound (B&B) is a divide-and-conquer algorithm that also relies on relaxations. Note that although we will discuss B&B in terms of LP relaxations, other relaxations could be used as well. B&B starts out by solving an LP relaxation and if its solution is not integral then the IP feasible region is subdivided into subproblems which are optimized recursively. The optimal solution will be the best of the subproblem solutions.

A *search tree* keeps track of B&B's progress; the *root* of the search tree corresponds to the first LP relaxation and further *nodes* (*children* of the *parent* node) are created by subdividing feasible regions. The subdivision process is called *branching*; traditionally two subproblems are created by identifying a variable with a fractional value in the current LP solution and setting it to 0 in one branch and to 1 in the other branch (this is called *variable branching*). Note that we can create more than two subproblems (although not in case of branching on binary variables); or subdivide the feasible region along a hyperplane that is not necessarily perpendicular to any of the axes; that is, *branch on a constraint*. For instance, if the sum of some binary variables must not exceed 1 then we can create two subproblems by assuming that the sum is 0 in one branch and 1 in the other. We will see examples of both variable and constraint branching in Section 5.3. Note that no branching is necessary at a search tree node if the subproblem is infeasible or its solution is integral (that is, a feasible solution to (IP) is found); in this case the search tree node can be *fathomed*.

B&B would be a simple enumeration algorithm without *bounding*. Note that the solution to the LP relaxation at the children of a search tree node will never be lower than that at the parent (the LP relaxations at the children are more restrictive). Therefore no integral solution with objective value lower than the LP optimal value can be found in the subtree rooted in any search tree node. Thus, as soon as an *upper bound* is known, nodes with LP optimal value exceeding the upper bound can be fathomed.

B&B is usually implemented by keeping a list of *candidate nodes* that initially contains the root only. Then, in a general step of the algorithm, a node is chosen (and removed) from the candidate list of unfathomed nodes and the corresponding LP relaxation is solved. If the LP relaxation is infeasible or its value reaches the upper bound then the node is fathomed. If the solution is integral feasible the node is fathomed again; the upper bound is updated in case the solution value is lower. Otherwise a branching variable or constraint is chosen, the feasible region at the node is subdivided and a child node is created for each new subproblem and placed on the list of candidate nodes. The algorithm stops when the candidate list is exhausted.

Important issues that influence the behavior of the algorithm include the choice of the next node from the candidate list and the choice of the branching object (variable or constraint). In what follows we briefly outline the most popular approaches for general IPs. If the IP possesses a special structure other methods might be more effective.

A very popular rule for choosing the next node from the candidate list is to choose one with the lowest LP objective value. This rule leads to a small search tree since the node with the lowest LP objective value must be considered no matter how the nodes are enumerated. Another widely used rule is *last in first out (LIFO)* that leads to a depth-first enumeration of the search tree. This means that if a node is not fathomed and removed, one of its children is processed next. The advantage of this approach is fast and easy LP reoptimization at the child node since the formulation at the parent is already solved and going to the child node means only a bound change or an additional constraint.

The choice of the branching object is an equally complex issue. For example, choosing a variable whose value is near .5 in the LP optimal solution and has a large objective function coefficient is a good idea since the children are expected to have very different LP solutions and the LP optimal value in the branch where the variable is given the value 1 will be high (and the node is hopefully fathomable). *Strong branching* is a staple for any efficient B&B implementation. Here, instead of choosing just one branching object, a set of branching candidates is selected and the LP relaxations at the would-be children are *presolved* for a few iterations (this gives some insight as what would happen if the algorithm branched on each of the candidates). The "most promising" candidate is then chosen for branching (see Section 5.3.6 for further details in the context of our implementation).

2.1.3 Branch-and-Cut

Branch-and-Cut (B&C) incorporates cutting planes into the B&B framework. It generates valid inequalities at each search tree node to strengthen the LP relaxations and hence obtain better lower bounds. This can be viewed as implicitly using stronger LP relaxations at the search tree nodes by generating parts of the relaxations on the fly. This combined approach leads to faster fathoming of the nodes and usually to a far smaller search tree than pure B&B. Branch-and-Cut is particularly effective for those classes of problems for which problem specific cuts can be efficiently generated.

All the terminology introduced for B&B carries over to B&C. When processing a subproblem at a node, the operations of solving LP relaxations and adding valid inequalities (violated by the current LP optimal solution) are iterated first. When

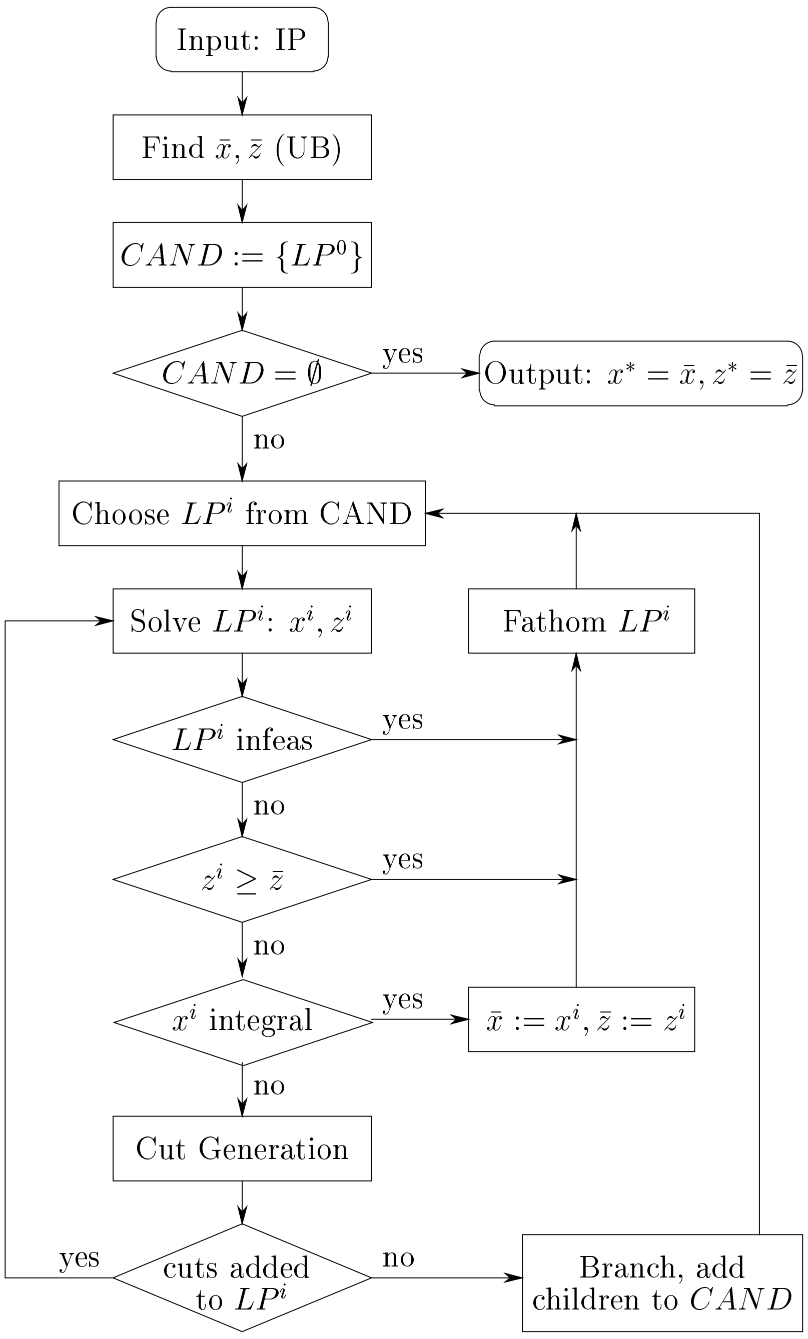


Figure 2.2: The flow of the Branch-and-Cut Algorithm

we are not able to generate more cuts or the LP objective value does not improve sufficiently (*tailing off*, see Section 5.3.5) then the algorithm resorts to branching. Note that valid inequalities might be *locally valid* only (valid for the search tree node and thus for the subtree rooted at that node) and not valid globally (throughout the search tree). Figure 2.2 gives an outline of a general B&C algorithm.

B&C implementations usually contain a *cut pool*, a repository of inequalities that were found violated for some subproblem. Since it is not unlikely that a cut generated for a particular subproblem might be both valid and violated for another subproblem, checking the cut pool before generating cuts might save a considerable amount of time. Of course, care has to be taken with cuts that are locally valid only.

2.2 The COMPSys framework

We used the COMPSys framework ([ELRT97]) to implement a Branch-and-Cut (B&C) procedure for the Set Partitioning Problem. COMPSys is a *generic parallel* B&C *framework* that was designed to aid the development of problem class specific B&C implementations. The user of this framework provides the problem specific information through user-written functions, while procedures which are common for parallel B&C implementations (like search tree and cut pool management, inter-process communication and LP solving) are handled by the framework and are completely transparent to the user. The user-written functions are well defined (the user needs only a minimal knowledge of the internal workings of the framework) and defaults are provided wherever possible, making adaptation to a specific problem easier.

COMPSys works in a distributed environment employing a master-slaves model. The search tree is managed by the master process while the slaves undertake all the other tasks of B&C. The parallelism in the framework is realized on a high level. The main source of parallelism is the observation that the nodes of the search tree can be processed simultaneously. In addition to this, LP solving and separation for a particular search tree node are also carried out in parallel, and separate processes maintain *cut pools* (collections of valid inequalities). We will give a detailed description of all the processes in Chapter 5.

COMPSys also includes such features as *strong branching*, *reduced cost fixing* (these two are usually implemented in IP solvers); as well as the possibility of using *decomposition* for separation, *branching on constraints* (not only variables), *multi-way branching* (more than two branches at a search tree node), *column generation* and a *graphical user interface* to aid debugging and separation (Chapter 6 provides details of this feature).

A preliminary version of the framework was originally implemented by T.K. Ralphs ([Ral95]) and L. Ladányi ([Lad96]) during their thesis research. This code has evolved into the current COMPSys framework with additional ideas and contributions by the above authors and the present author in addition to L. Kopman ([Kop99]) and G. Pangborn ([Pan]).

As different applications were implemented using the framework, ideas that could be used in a general setting were identified and “lifted” into the framework. Branching on cuts was first developed for the TSP ([Lad96]). A decomposition method for separation was conceived in the context of the VRP ([Ral95]); this result was further extended with the addition of Farkas cuts ([Kop99]). A Graphical User Interface (GUI) was added to ease debugging and help with identifying violated inequalities when testing was begun on SPP models (Chapter 6). The effectiveness of column generation in an SPP setting is currently being investigated ([Pan]). Through these research projects COMPSys has matured into a robust, efficient, easy-to-use platform for problem class specific Branch-and-Cut implementations.

2.3 The Set Packing and Covering polytopes

Let us denote the Set Partitioning, Packing and Covering Polytopes by $\mathcal{P}_{SPP}$, $\mathcal{P}_{SP}$ and $\mathcal{P}_{SC}$. It is easy to show that

$$\mathcal{P}_{SPP} = \mathcal{P}_{SP} \cap \mathcal{P}_{SC}.$$

Thus inequalities that are valid for $\mathcal{P}_{SP}$ or $\mathcal{P}_{SC}$ are also valid for $\mathcal{P}_{SPP}$. While there is very little known about the SPP polytope, the other two polytopes are well studied. Below we summarize some important facts about these polytopes, in particular, the characterizations of certain facet defining inequality classes. [BP76] and [Bor97] provide comprehensive surveys about these polytopes, including aspects that we will not discuss here.

2.3.1 $\mathcal{P}_{SP}$

Recall (Section 1.3) that SP is equivalent to MWSSP on the intersection graph $G_A = (V, E)$ corresponding to the problem matrix A. The argument there showed that the two problems have exactly the same feasible solutions; thus $\mathcal{P}_{SP}$ and the Stable Set polytope $\mathcal{P}_{SS} = \text{conv}\{x \in \{0,1\}^n \mid A^T_{G_A} x \leq \mathbf{1}_{|E|}\}$ are the same; we will simply denote this polytope by $\mathcal{P}$ in the remainder of this section. Also, we will use the words variable, column (of the problem matrix) and node (of the intersection graph) interchangeably.

Subproblems of the original problem play a very important role as we will see below. A subproblem arises from a *submatrix* A_{IJ} of A (where I is a subset of the rows and J is a subset of the columns). Then $G_{A_{IJ}}$, the intersection graph corresponding to A_{IJ}, is a subgraph of G_A. If I is the entire set of rows then $G_{A_{IJ}}$ is a *node induced subgraph* of G_A; we will deal only with this kind of subproblem in this section and we use the usual notation $G[J]$ for node induced subgraphs instead of $G_{A_{IJ}}$. The SP (SS) polytope of the subproblem is denoted by $\mathcal{P}(G[J])$.

$\mathcal{P}$ is full dimensional, $dim(\mathcal{P}) = n$, since the null vector and the n unit vectors are affinely independent and are all in $\mathcal{P}$. Thus $\mathcal{P}$ has a unique (up to positive scalar multiplication) minimal description with facet-defining linear inequalities.

$\mathcal{P}$ is *lower comprehensive*; that is, all nonnegative vectors not larger than a vector in the polytope are also in the polytope ($x \in \mathcal{P}$ and $0 \leq y \leq x$ implies $y \in \mathcal{P}$). Therefore, $\mathcal{P}(G[J])$ is the same as the projection of $\mathcal{P}$ onto the subspace defined by the variables in J. From this it follows that inequalities valid for $\mathcal{P}(G[J])$ can be "extended" with zero coefficients for the nodes in $V \setminus J$ to obtain valid inequalities for $\mathcal{P}$. Note that this is a trivial extension; these valid inequalities for $\mathcal{P}$ could most likely be made stronger by assigning nonzero coefficients to some of the variables in $V \setminus J$.

The process of computing coefficients for nodes in $V \setminus J$ to obtain a valid inequality for $\mathcal{P}$ from a valid inequality of $\mathcal{P}(G[J])$ is called *lifting* and the coefficients are called *lifting coefficients*. Note that the right hand side of the valid inequality will not change and the lifting coefficients cannot exceed this value. The lifting coefficients for nodes in $V \setminus J$ could be computed one by one (take a node not in J, compute its coefficient, add the node to J and continue), this is *sequential lifting*; or all at once (*simultaneous lifting*). Note that the valid inequality obtained at the end of sequential lifting depends on the order in which the nodes were considered (*lifting order*). Whether sequential or simultaneous lifting is used, the goal is to determine the largest possible lifting coefficients so that the lifted valid inequality is as strong as possible.

First consider sequential lifting. Given the valid inequality $\sum_{j \in J} \alpha_j x_j \leq \alpha^0$ for $\mathcal{P}(G[J])$, the largest possible lifting coefficient of a $v \in V \setminus J$ is $\alpha_v = \max\{0, \alpha^0 - z^*\}$ where z^* is the optimal solution value of the following SP:

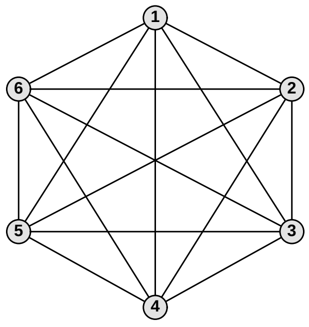
$$
\begin{array}{lrrcl}
 & z^* = \max & \sum_{j \in J} \alpha_j x_j & & \\
(LIFT) & & \sum_{j \in J} A_j x_j & \leq & \mathbf{1}_m - A_v \\
 & & x & \in & \{0,1\}^{|J|}
\end{array}
$$

Or equivalently, assume the node is chosen into a stable set, compute the value of a MWSS on its non-neighbors and subtract it from the right hand side to obtain the node's lifting coefficient. Note that α_v will be an integer. Also, the lifting coefficients of the variables not yet included into the inequality cannot increase as a result of lifting other variables first.

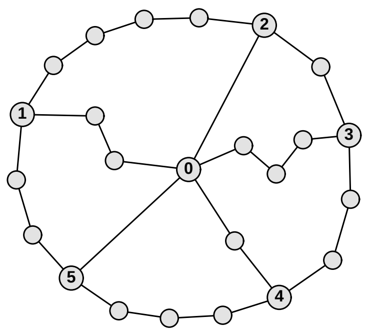
Solving (LIFT) is itself a difficult problem. In certain cases when $G[J]$ admits a special structure or α^0 is small, efficient algorithms can be devised for (LIFT), see Section 5.4 for some examples. Also, a weaker lifting coefficient can be obtained by using an upper bound on z^* instead of z^* (for instance, by optimizing the dual of its LP relaxation, [HP93]).

Sequential lifting with (LIFT) as a subroutine not only lifts a valid inequality of $\mathcal{P}(G[J])$ into a valid inequality of $\mathcal{P}$, but *lifting facet defining inequalities results in facet defining inequalities* as well (see [Pad73] and [NT74]). Thus each facet defining inequality of $\mathcal{P}$ is either (sequentially) lifted from a facet defining inequality of $\mathcal{P}(G[J])$ for some node induced subgraph $G[J]$, or it is facet defining only for $\mathcal{P}$ (i.e, its projection is not facet defining for any $\mathcal{P}(G[J])$). Therefore our goal is to characterize graph classes for which the corresponding polytopes have easily identifiable facets and then to look for such graphs as node induced subgraphs in the intersection graph G_A.

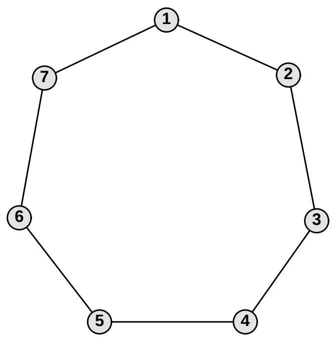
Now we outline some well-known graph classes and corresponding valid inequalities that are facet defining for the graph's Stable Set polytope. For each graph class we also indicate the complexity of the corresponding separation problem (that is, given a fractional solution vector x for a relaxation of the stable set problem of a graph, decide whether or not any inequality in the given class is violated by the solution). Figure 2.3 illustrates each graph class. For references to additional graph/inequality classes see [Bor97]. Note that the nonnegativity constraints



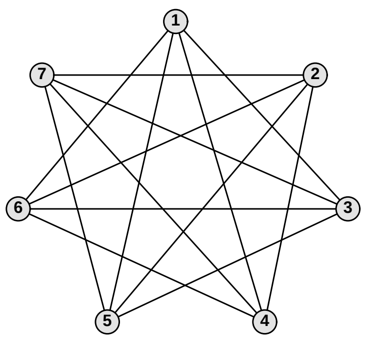
a. The clique K_6



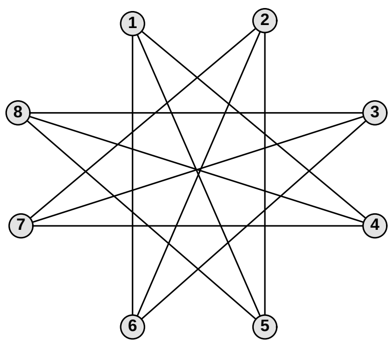
b. Wheel with 5 spoke-ends
$\mathcal{E} = \{3, 4\}$, $\mathcal{O} = \{1, 2, 5\}$



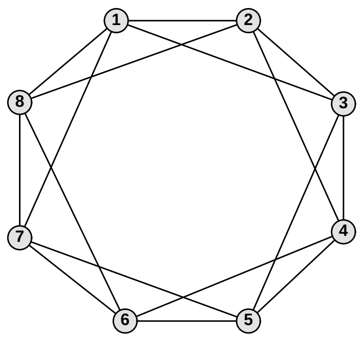
c. The odd hole $H(7)$



d. The odd antihole $\bar{H}(7)$



e. The web $W(8, 3)$



f. The antiweb $\bar{W}(8, 3)$

Figure 2.3: Graphs with facet defining valid inequalities

($x_j \geq 0$) are trivially facet defining for any Stable Set polytope.

Cliques

Graph: A *clique* K_n is a complete graph on n nodes.

Inequality: $\sum_{j \in K} x_j \leq 1$ where K is the node set of a clique.

Facet defining for $\mathcal{P}(G[K])$; also facet defining for $\mathcal{P}$ if the clique is maximal ([Pad73]).

Separation: NP-complete (equivalent to maximum weight clique). Clique inequalities are in the class of *orthonormal representation constraints* which are polynomial time separable [GLS88].

Odd Holes and Antiholes

Graph: An *odd hole* $H(2k+1)$ is a cycle on an odd number of nodes and no chord (edge between two non-consecutive nodes on the cycle).

An *odd antihole* $\bar{H}(2k+1)$ is the (edge) complement of an odd hole.

Inequality: $\sum_{j \in H} x_j \leq (|H| - 1)/2$, (odd hole inequality)

$\sum_{j \in \bar{H}} x_j \leq 2$ (odd antihole inequality), where H is the node set of an odd hole. Facet defining for $\mathcal{P}(G[H])$ / $\mathcal{P}(G[\bar{H}])$ ([Pad73]).

Separation: Polynomial time for odd holes ([GLS88], also in Section 5.4.2).

Complexity is not known for odd antiholes, but odd antihole inequalities are in the class of *matrix inequalities* which are polynomial time separable [LS90].

Note that the inequality $\sum_{j \in H} x_j \leq (|H| - 1)/2$ is still valid if H is only an odd cycle (not chordless), but the inequality is facet defining for $\mathcal{P}(G[H])$ only if the cycle is chordless.

Webs and Antiwebs

Graph: A *web* $W(n,k)$ ($n \geq 2$, $1 \leq k \leq n/2$ integers) has n nodes and edges $(i, i+k), \ldots, (i, i+n-k)$ (sums taken mod n) for all nodes i. An *antiweb* (or *circulant*) $\bar{W}(n,k)$ is the (edge) complement of the web $W(n,k)$. Note: cliques, odd holes and antiholes are special cases.

Inequality: $\sum_{j \in W} x_j \leq k$, (web inequality)

$\sum_{j \in \bar{W}} x_j \leq \lfloor n/k \rfloor$ (antiweb inequality), where W is the node set of a web. Facet defining for $\mathcal{P}(G[W])$ / $\mathcal{P}(G[\bar{W}])$ if n and k are relative prime [Tro75].

Separation: Complexity not known.

Wheels

Graph: A *wheel* is an odd cycle (*spoke-ends*) with an additional node adjacent to all nodes on the cycle (*hub*), each edge possibly replaced by a sequence of edges so that all the *face cycles* (cycles through the hub and two neighboring spoke-ends) are odd.

Inequality: $\sum_{j \in W} x_j + \sum_{j \in \mathcal{E}} x_j + (k-1)x_0 \leq (|W| + |\mathcal{E}|)/2 - 1 \quad (I_{\mathcal{E}})$ and

$\sum_{j \in W} x_j + \sum_{j \in \mathcal{O}} x_j + kx_0 \leq (|W| + |\mathcal{O}| + 1)/2 - 1 \quad (I_{\mathcal{O}})$

where W is the node set of the wheel, x_0 is the hub and $\mathcal{E}$ and $\mathcal{O}$ are spoke-ends of even/odd distance from the hub ($|\mathcal{E}| + |\mathcal{O}| = 2k+1$).

Both facet defining for $\mathcal{P}(G[W])$ if the distance between any two $\mathcal{E}$ (for $(I_{\mathcal{E}})$) or $\mathcal{O}$ (for $(I_{\mathcal{O}})$) nodes is at least 2 ([CC97]).

Separation: Polynomial time ([CC97], see [BM94b] for special case of wheel with 3 spoke-ends – subdivision of K_4).

Note that wheels with no nodes on the spokes (that is, the spoke ends are of distance one from the hub) can be obtained by lifting the hub into the odd hole inequality of the rim cycle.

There are well-known classes of graphs whose stable set polytopes can be characterized by a well determined set of inequality classes. If these inequalities can be separated in polynomial time then the corresponding MWSSP (and thus SP) can also be solved in polynomial time for these problems ([GLS88], the polynomial time equivalence of separation and optimization). Graphs for which the MWSSP can be solved in polynomial time using the above listed inequalities include (with the characterizing inequalities in parentheses): *bipartite graphs* with no isolated nodes (nonnegativity and edge $(x_i + x_j \leq 1)$); *perfect graphs* (nonnegativity and clique); *t-perfect graphs*, e.g., series-parallel graphs (nonnegativity, edge and odd hole). In genaral we say that a graph is *perfect for a set of inequality classes* if all the facet defining inequalities of the corresponding stable set polytope belong to one of the given inequality classes.

An interesting question is how to verify that the inequalities listed above are valid and facet defining. The clique, odd hole, odd antihole, web and antiweb inequalities are so called *rank inequalities*; the coefficients on the left hand sides are all ones and the right hand side is exactly the size of the maximum cardinality stable set in the corresponding graph:

$$\sum_{j \in V} x_j \leq \alpha(G).$$

Note that wheel inequalities are not rank inequalities in general. Rank inequalities are obviously valid for $\mathcal{P}(G)$ and for the stable set polytope of any graph that contains G as a node induced subgraph. Rank inequalities are also facet defining if the following condition due to Chvátal ([Chv75]) is met. An edge of G is called *α-critical* if the size of the maximum stable set increases when the edge is removed. Then if the α-critical edges on the original set of nodes form a connected graph, the rank inequality is facet defining for $\mathcal{P}(G)$. It is easy to check that the Chvátal-condition holds if G is a clique, odd hole, odd antihole, web or antiweb. Note that the Chvátal-condition is only sufficient; necessary conditions can be given in terms of *critical cutsets* ([BP76]).

Another very useful tool to derive valid inequalities is the Chvátal-Gomory procedure ([Chv73]). Intuitively, applying this method means taking nonnegative linear combinations of known valid inequalities for node induced subgraphs of G, rounding down to the nearest integer first each coefficient on the left hand side and then the right hand side value. As an illustration of this method, consider the odd cycle (with or without chords) $C(2k+1)$. Adding up the edge inequalities $x_j + x_{j+1} \leq 1$ for all $2k+1$ edges (sums taken mod $2k+1$) we obtain

$$2(x_1 + \ldots + x_{2k+1}) \leq 2k+1.$$

Dividing the sum by 2 and rounding down the right hand side to the nearest integer yields the odd cycle inequality

$$x_1 + \ldots + x_{2k+1} \leq k.$$

See Section 6.3 for a nontrivial application of this procedure.

The Chvátal-Gomory procedure can be used to generate *packing odd hole* inequalities. After an odd hole is located in the intersection graph, a row whose support contains the endpoints of the edge is chosen for each edge of the odd hole. Note that the nodes of the odd hole have coefficients two in the sum of these inequalities. Dividing the sum by two and rounding down both the coefficients and the value on the right hand side yields a valid inequality for the SP polytope. In fact, this inequality is a lifted odd hole inequality since variables in the odd hole have coefficients one and the right hand side is that of the odd hole inequality. This method is documented in [Bor97]; we have also used it in our implementation (see Section 5.4.2).

The wheel inequalities described above have been derived using the same method (the odd cycle inequalities for the face cycles and edge inequalities for some appropriately chosen edges on the rim were added up); this proves the validity of these inequalities. Proving that the wheel inequalities are facet defining for $\mathcal{P}(G[W])$ is based on *subdividing edges* (replacing an edge with a path), see [CC97] for details. Using graph operations (like extending graphs with nodes or cliques, substituting nodes and edges with paths, (de)composition of graphs ([BM94a]), clique identification ([Chv75]) are other standard ways of deriving valid (facet defining) inequalities, see [Bor97] for a comprehensive survey.

2.3.2 $\mathcal{P}_{SC}$

Unlike the Set Packing Problem, the Set Covering Problem does not have an equivalent graph theoretic formulation (it can be modeled with *hypergraphs* only). The SC polytope will be denoted by $\mathcal{Q}$ in this section. $\mathcal{Q}$ is full dimensional if each row of A contains at least two entries (the vector of all ones and vectors with one zero and $n-1$ ones are affinely independent and are in $\mathcal{Q}$).

Subproblems of the original problem are defined here again by submatrices; A_{IJ} is a submatrix of A with row set I and column set J. $\mathcal{Q}(A_{IJ})$ denotes the SC polytope of the subproblem.

$\mathcal{Q}$ is *upper comprehensive*; that is, all vectors not smaller than a vector (but not larger than 1) in the polytope are also in the polytope ($x \in \mathcal{Q}$ and $x \leq y \leq 1$ implies $y \in \mathcal{Q}$). This property implies that the polytope $\mathcal{Q} \cap \{x \in \Re^n \mid x_j = 1 \ \forall j \notin J\}$ is the same as the polytope resulting from projecting $\mathcal{Q}$ "upwards" onto the subspace defined by setting all variables not in J to one. Therefore valid inequalities for this polytope can be naturally extended to a valid inequalities for $\mathcal{Q}$ with zero coefficients for the variables not in J. Unfortunately, the parallel with set packing stops here; the polytope just defined is *not* the same as the SC polytope of the subproblem (which is $\mathcal{Q} \cap \{x \in \Re^n \mid x_j = 0 \ \forall j \notin J\}$).

Therefore, valid inequalities for a subproblem's SC polytope do not simply carry over to $\mathcal{Q}$; they have to be *lifted*. Lifting for SC is somewhat more complicated than for SP because the not yet lifted variables cannot simply be ignored (for more details see [NT74], [Sas89] and [NS89]). Thus valid inequalities for $\mathcal{Q}$ may be obtained by identifying classes of submatrices for which the corresponding SC polytopes have easily identifiable facets and then trying to lift these inequalities. However, lifting may result in an inequality which is not restrictive enough (or at all). A comprehensive list of references to such submatrix classes can be found in [Bor97]. Rank inequalities can be defined analogously (with the right hand side being the size of a minimum cover in the subproblem), along with sufficient or necessary conditions for the facet defining property.

The Chvátal-Gomory procedure can be readily adapted (rounding up instead of down). Indeed, similarly to the packing odd holes we can derive *cover odd hole* inequalities which are generated exactly the same way as the packing version (except for the direction of the rounding); see Section 5.4.2.

Chapter 3

Problem size reduction

Problem size reduction is the collective name for methods that, given a Set Partitioning Problem, reduce the set of variables and/or constraints through logical implications without eliminating optimal solutions to the original problem (but probably reducing the feasible region). Problem size reduction is an essential part of our feasible solution heuristic (Chapter 4) and is also invoked from the Branch-and-Cut framework to propagate the effects of variable fixing based on reduced costs and branching decisions (Section 5.3.2).

In the following sections first we define the reduction operations known in the literature, then we show that applying these reduction methods to a problem instance in any order until no more reductions are possible will always produce the same result. Then we discuss our implementation of these methods and conclude with computational results.

3.1 Description of reduction methods

First we give a few technical definitions that are used in the description of the reduction operations. Consider the Integer Programming formulation of the SPP introduced in Section 1.2. *Fixing a variable to zero* means that the variable, its objective function coefficient and the corresponding column are permanently removed from the problem formulation. *Removing a row* means removing that row from the problem matrix along with the corresponding right-hand side entry, and fixing any variable to zero whose resulting column has only zero entries. *Variables* can also be *fixed to one* during reduction. In this case all rows in this column's support can be removed since they will be satisfied by the variable fixed to one. Moreover, all other columns that belong to the support of any of these rows can be fixed to zero. Indices of variables fixed to one are recorded in a list we call ONES. Sometimes *columns are merged* during reduction which means that the (orthogonal) columns of two variables are combined into one column and the original columns are deleted from the formulation. The objective function coefficient of the merged variable will be the sum of the two objective function coefficients. Index pairs of merged variables are recorded in a list we call MERGES.

Variable fixing and merging can be interpreted in terms of the intersection graph (see Section 1.3). Fixing a variable to zero corresponds to removing a node with its adjacent edges from the graph. Fixing a variable to one corresponds to removing a node (and recording its index in ONES) and then removing all the nodes that are adjacent to it. Merging two columns corresponds to contracting two nonadjacent nodes of the graph into one node (and listing their indices in MERGES).

In the following we describe the reduction methods that are known in the literature ([BP76], [HP93]). For each method we justify that no optimal solution is lost by applying it. Figure 3.1 illustrates each case.

1. **Duplicate Columns (DUPC)**

   If two columns are identical then the one with the larger objective function coefficient can be removed from the problem.

   $$a_j = a_k \text{ for some } j, k \in N \implies$$

   $$\text{if } c_j > c_k \text{ then } x_j \text{ is fixed to } 0, \text{ else } x_k \text{ is fixed to } 0.$$

   Justification: A solution is not optimal if the more expensive of the identical columns is in the solution since it could be replaced by the cheaper one.

2. **Column is a sum of other columns (SUMC)**

   If a column can be expressed as a sum of other columns and the total cost of the columns in the sum is smaller than the cost of the single column then the column can be removed from the problem.

   $$a_j = \sum_{k \in K} a_k \text{ and } c_j \geq \sum_{k \in K} c_k \text{ for some } j \in N \text{ and } K \subseteq N \setminus \{j\} \implies$$

   $$x_j \text{ is fixed to } 0.$$

   Justification: A solution is not optimal if the expensive single column is in the solution since it could be replaced by the columns in the sum without increasing the cost of the solution.

Note: Although SUMC contains DUPC as a special case, it is reasonable to consider them separately since detecting duplicate columns is very fast.

3. **Row clique can be extended (CLEXT)**

   If a column is nonorthogonal to all columns in the support of a row, but is not in the support itself, then the variable corresponding to this column can be fixed to zero. In terms of the intersection graph, a node that extends a row clique can be removed.

   $$a_j^T a_k \geq 1 \ \forall k \in N^i \text{ for some } i \in M \text{ and } j \in N \setminus N^i \implies$$

   $$x_j \text{ is fixed to } 0.$$

   Justification: One of the columns from the support of the row must be chosen in every feasible solution. Since the column is nonorthogonal to every column in the support, it cannot be chosen if any of the columns in the support is chosen.

4. **Dominated rows (DOMR)**

   If the support of a row contains the support of another row then the row with the smaller support (the "shorter row") *dominates* the row with the larger support (the "longer row"). In this case the longer row can be removed along with the variables that are in the longer row's but not in the shorter row's support.

   $$N^i \subseteq N^l \text{ for some } i \neq l \implies$$

   $$x_j \text{ is fixed to } 0 \ \forall j \in N^l \setminus N^i, \text{ row } l \text{ is removed.}$$

Justification: One of the columns from the shorter row's support has to be chosen in any feasible solution. This column will make the longer row's equality satisfied as well, so variables corresponding to columns that are in the longer but not in the shorter row can be fixed to zero. After fixing these variables to zero the two rows become identical, so one of them (not necessarily the one which was originally the longer) can be removed.

Note that columns deleted with this method could be deleted by CLEXT, but DOMR is more efficient since it discovers many deletable columns at once, rather than one by one as CLEXT would do.

5. **Singleton row (SINGL)**

   If a row has only one nonzero entry in it (that is, only one column intersects the row) then the variable corresponding to this column can be fixed to 1.

   $$a_{ij} = 1 \text{ for } j \in N, \text{ but } a_{ik} = 0 \ \forall k \in N \setminus \{j\} \text{ for some } i \in M \implies$$

   $$x_j \text{ is fixed to 1.}$$

   Justification: The equality in the row that has only one column intersecting can be met only if the variable corresponding to this column is set to 1. (Note that according to the definition of fixing a variable to 1 variables with columns nonorthogonal to column j are fixed to zero.)

6. **Two rows differ by two entries (DTWO)**

   If the supports of two rows are identical except for two entries, one of which is in one of the rows and the other is in the other row, then, depending on whether the two columns are nonorthogonal or orthogonal, the two columns can either be removed or merged into one column (also one of the rows can be removed).

$$|N^i| = |N^l| \text{ and } N^i \oplus N^l = \{j,k\} \text{ for some } i,l \in M \implies$$

$$\text{if } a_j^T a_k \geq 1 \text{ then } x_j, x_k \text{ are both fixed to 0, else } x_j \text{ and } x_k \text{ are merged;}$$

$$\text{one of the rows is removed in both cases.}$$

Justification: Observe that the two variables will take identical values in any feasible solution. If they are nonorthogonal then they cannot both be one, thus they have to be fixed to zero. If they are orthogonal then they can be merged into a new column (with their costs added). In either case, there will be two identical rows, one of which can be deleted.

In the following sections the abbreviations introduced above will be used for both the occurrences of the above conditions and for the operations described.

3.2 Theorem of exhaustive reduction

Our main goal in this section is to prove that the above-described reduction operations, applied in any order to an SPP instance until no more reductions are possible, will always produce the same reduced problem.

We say that two reduction sequences (sequences of reduction operations) are *equivalent* if, when applied to the same SPP instance, the resulting reduced matrices

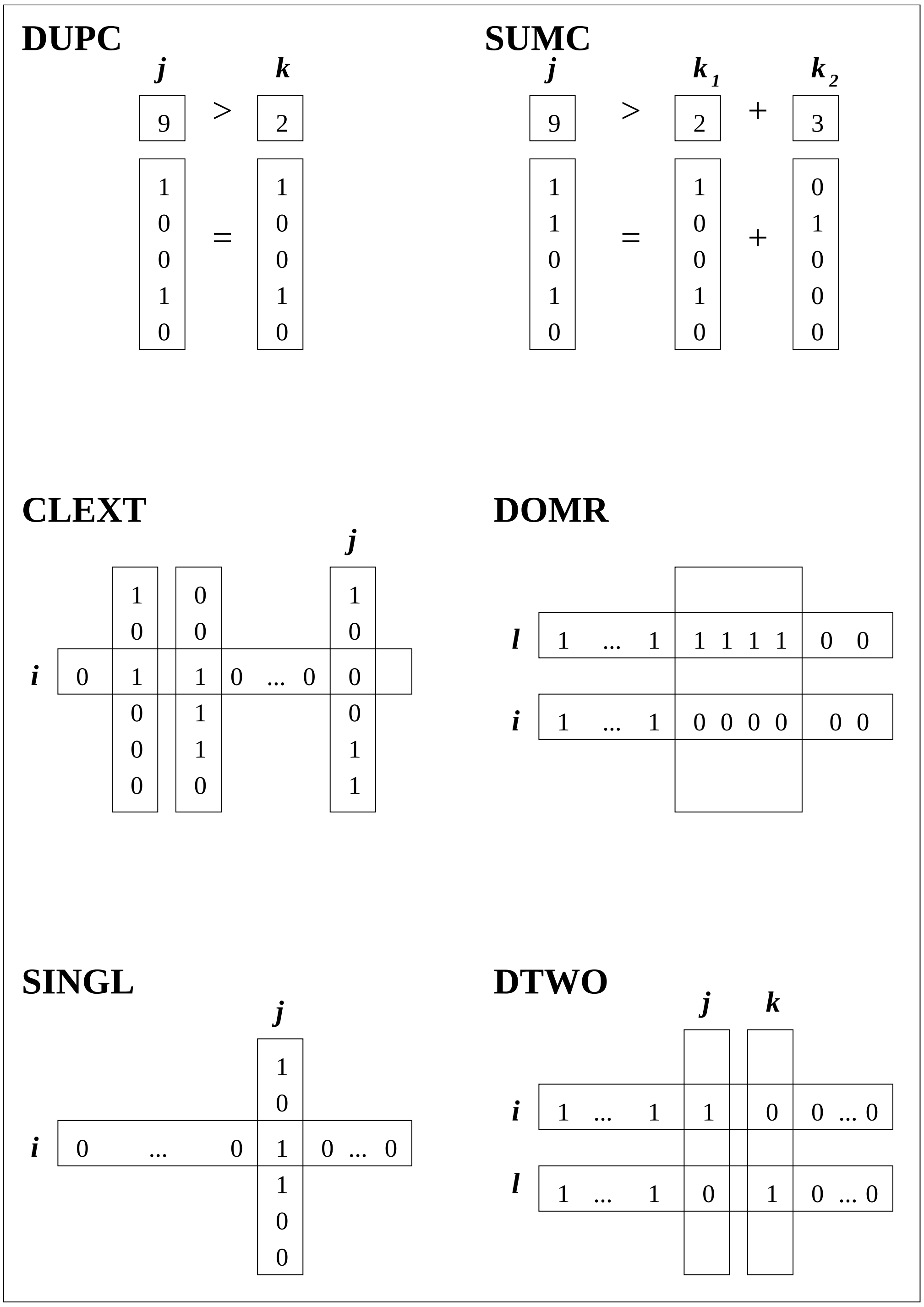


Figure 3.1: The six reduction methods

are identical up to a permutation of the rows and columns of one of the matrices. A reduction sequence is *exhaustive* if no reductions are possible after applying it.

Theorem 3.1 *Given an SPP instance, any two exhaustive sequences of DUPC, SUMC, CLEXT, DOMR, SINGL and DTWO are equivalent.*

First we show that any sequence of the above six reduction operations can be replaced by an equivalent sequence using only three types of these operations: SUMC, CLEXT and MERGE (which is a simplified version of DTWO defined below) followed by the possible deletion of duplicate rows and possible fixing of variables to one.

Observe that a SINGL operation can be thought of as a sequence of DOMR operations since the row with the singleton in it dominates all the other rows that the corresponding column intersects. After the DOMR operations the singleton row along with its column are still in the problem, but the column will intersect only this row (that is, we have an isolated node in the intersection graph). Fixing this column to one now means only recording its index in ONES and deleting its column and row from the matrix.

Also, DOMR can be replaced by a sequence of CLEXT operations since columns in the longer but not in the shorter row are all nonorthogonal to all columns in the shorter row. Then we are left with two identical rows and we delete the one that was deleted with DOMR.

Note that if the two columns are nonorthogonal in a DTWO instance (that is they can be deleted) then each extends the other row's clique, so these two columns could be deleted with two CLEXT operations. If the two columns are orthogonal, we replace DTWO with MERGE which simply merges the two columns but does not delete either row. In both cases we are left with two identical rows and we delete the one that was deleted with the original DTWO operation.

As we have seen earlier, DUPC is a special case of SUMC, so every DUPC operation can be replaced by a SUMC with only one summand.

It is obvious that the reduction sequence obtained by the above substitutions is equivalent to the original sequence. Also, since deletion of duplicate rows will not destroy old instances of reduction, will not create new instances and the other operations can only create but not destroy duplicate rows, these operations can be shuffled to the end of the reduction sequence while equivalence is preserved. Similarly, the fixing of isolated variables to one can be postponed until the very end of the reduction.

Now consider the original two exhaustive reduction sequences and apply the described substitutions (with removal of duplicate rows and fixing of isolated variables postponed to the end). **In the rest of the proof we will show that the two sequences are equivalent up to the point where duplicate rows are removed and isolated variables are fixed.** From this statement the theorem follows easily. Note that since SUMC, CLEXT and MERGE do not remove any of the rows, the two resulting matrices before the deletion of duplicate rows and fixing of isolated variables will be identical up to a permutation on the columns only. After this point the two sequences can differ only in which row of each duplicate pair to remove, and this difference can be accommodated by a permutation on the rows. Obviously the same isolated variables will be fixed to one. Thus the theorem is proved.

Consider the two exhaustive reduction sequences of the three operations SUMC, CLEXT and MERGE. Note that SUMC and CLEXT delete one column, and MERGE merges two columns; that is, the total number of columns in the current problem matrix is reduced by exactly one each time a reduction operation is applied, thus the reduction sequences are finite. For ease of explanation we associate time with the sequences and say that the reductions start at time 0 with each reduction step taking one unit of time.

We will prove the equivalence of the two sequences by induction on the number of columns in the matrix. If the number of columns is 1 then the statement is trivially true (the reduction sequences are empty). So we assume the statement is true for matrices with $n-1$ columns, and we prove the statement for matrices with n columns.

Consider the first operation in one of the exhaustive sequences. We will show that we can find an equivalent sequence to the other exhaustive sequence which starts with the same reduction. By applying the first operation to the original problem instance we are left with $n-1$ columns in the matrix; then the inductive statement shows that the two sequences are equivalent.

The whole proof is based on exchanging operations until the desired one is at the beginning of the other sequence. The first two lemmas show that an operation which is done at some time in a sequence can be swapped with operations preceding it one by one, back down to the time when the operation could have been first done. Then the last two lemmas show how to find the first operation of one sequence in the other (and what operations to look for if the first operation does not explicitly appear in the other sequence).

First we will see that that deletable/mergable columns do not become non-deletable/non-mergable as a result of other operations.

Lemma 3.2 *If a column is deletable at some time in a reduction sequence then it will remain deletable after any reduction operation that does not involve (does not delete or merge) this column. Similarly, if two columns are mergable then they will remain mergable after any reduction operation that does not involve either of the two columns.*

The following is a trivial corollary of Lemma 3.2 if the reduction sequence is exhaustive.

Corollary 3.3 *In an exhaustive reduction sequence, if a column could be deleted at some point then it is either deleted or merged with another column at some later time. If two columns could be merged at some point then they will either be merged or one of the columns is deleted or merged with another column at some later time.*

proof of Lemma 3.2 First assume that column v is deletable with SUMC at some time; that is, $v = \sum v^l$ and $c(v) > \sum c(v^l)$ for some columns $v^1, \ldots, v^k$. This instance could disappear if one of the summands is deleted or merged with another column. We will show that v remains deletable after such an operation.

1. If a summand v^l is deleted with SUMC; that is, $v^l = \sum w^j$ and $c(v^l) > \sum c(w^j)$ then the w^j's can be used for v^l in the sum for v since $c(v) > \sum_{i \neq l} c(v^i) + c(v^l) > \sum_{i \neq l} c(v^i) + \sum c(w^j)$. Thus v is still deletable with SUMC.

2. If a summand v^l is deleted with CLEXT; that is, there exists a row i so that v^l is nonorthogonal to all columns in the support of row i. Then v is nonorthogonal to all columns in row i's support since v intersects all rows that v^l does. Also, v does not intersect row i itself, since otherwise a summand would need to cover row i and thus be in row i's support and orthogonal to v^l, which contradicts our assumption. Therefore v can be still deleted, now with CLEXT instead of SUMC.

3. If a summand v^l is merged with a column then v must be among the common columns of the two rows that differ by two, thus the other column that v^l is merged with must be also a summand (since this is the only column that can cover the other differ-by-two row that v^l does not intersect). Therefore the merged column could be used instead of v^l and the other summand, thus v is still deletable using SUMC.

Second, assume that column v is deletable with CLEXT at some time; that is, there is a row i so that v is nonorthogonal to all columns in i's support. Then, since column deletion does not change the orthogonality relationship of remaining columns, v remains deletable with the same CLEXT operation after columns other than v are deleted from the problem. Also, when two columns are merged, all columns that were nonorthogonal to either of them will be nonorthogonal to the merged column. Therefore v remains deletable with the same CLEXT operation if a column in row i's support is merged.

Finally, assume that MERGE could be applied to two columns, v and w at some time; that is, there exist rows i and j such that their supports differ by two columns only: row i's support contains v but not w and row j's support contains w but not v. Since the two rows i and j are the same except for columns v and w, a column deletion or merge that does not involve v or w will not remove the MERGE opportunity for v and w. ■

As the previous lemma stipulates, we do not need to distinguish between deleting a column by SUMC or CLEXT. Thus, as shorthand we will write $del(v)$ for the deletion of column v, and $merge(v, w)$ for the merging of columns v and w.

Given two consecutive operations in a reduction sequence we say that the second operation is *independent* of the first if the column(s) deleted or merged in the second operation is (are) already deletable/mergable before the first operation. Now we show that two such operations can be interchanged. This will make sure that a reduction instance present at time T_0 but not done until time T can be "bubbled back" to time T_0.

Lemma 3.4 *Given a sequence of reductions containing two consecutive operations with the second independent of the first, there is an equivalent sequence with the two operations interchanged.*

proof Let us denote the two operations by O_1 and O_2 and assume their order is $O_1 O_2$ originally. Since the second operation is independent of the first, it could be done at the time when O_1 occurs in the original sequence. Moreover, columns involved in O_2 are not involved in O_1 thus, by Lemma 3.2, the column(s) deleted/merged by O_1 can be deleted/merged by an operation O_1' (perhaps not the same as O_1, see the proof of Lemma 3.2 for details) after O_2. So O_1O_2 can be replaced by O_2O_1' resulting in an equivalent reduction sequence (the resulting problem matrices will be identical if merged columns are inserted into the same positions in the new sequence as in the old sequence). ■

The following claims summarize small but important observations needed later in the proof.

Claim 3.5 *If a deletable column v is merged with another column w then the merged column vw is also deletable.*

proof Suppose the two differ-by-two rows are i and j, row i's support contains v but not w and row j's support contains w but not v. At the time when v and w are merged, v can be deleted only with CLEXT (based on some row $k \neq j$) and not with SUMC since a summand would need to cover row i, but all the columns in row i's support are in row j's support as well and v does not intersect row j. After merging v and w, the merged column vw is nonorthogonal to every column in row k's support and it does not intersect row k itself (otherwise w would need to intersect row k but w is orthogonal to v while columns in row k's support are not); thus it can be deleted with CLEXT based on the same row k. ■

Claim 3.6 *If v and w are mergable columns and v is deleted then w becomes deletable as well.*

proof Let i and j be the two differ-by-two rows as in the proof of Claim 3.5. After v is deleted w becomes nonorthogonal to all columns in i's support but it is not in the support itself, so it can be deleted with a CLEXT. ■

Claim 3.7 *Assume v and w are mergable but v is merged with a third column z instead. If z and w are orthogonal then vz and w are mergable, otherwise both vz and w are deletable.*

proof Let i and j be the two differ-by-two rows which show that v and w can be merged. Since v and z are orthogonal, the two rows i and j will differ by the two columns vz and w. Now if z and w are orthogonal then vz and w are orthogonal as well, so the two columns become mergable as soon as v and z are merged. Otherwise both vz and w can be deleted with CLEXT. ■

Claim 3.8 *The following replacements are equivalence-preserving.*

1. *Suppose $merge(v, w)$ $del(vw)$ is in the reduction sequence at some time. Then it can be replaced by $del(v)$ $del(w)$ if v is deletable at the same time.*

2. *Suppose $del(v)$ $del(w)$ is in the reduction sequence at some time. Then it can be replaced by $merge(v, w)$ $del(vw)$ if v and w are mergable.*

3. *Suppose $merge(v, w)$ $merge(vw, z)$ is in the reduction sequence at some time. Then it can be replaced by $merge(v, z)$ $merge(vz, w)$ if v and z are mergable. (Note that w and w are orthogonal.)*

4. *Suppose $merge(v, w)$ $del(vw)$ $del(z)$ is in the reduction sequence at some time.*

Then it can be replaced by $merge(v,z)$ $del(vz)$ $del(w)$ *if* v *and* z *are mergable and* w *and* z *are nonorthogonal.*

proof These four statements follow directly from Claims 3.6, 3.5, 3.7 and 3.7, respectively. Observe that the resulting problem matrices will remain the same if in the new sequence the merged columns are inserted into the same positions as in the old sequence. ∎

Now we go back to the proof of our main theorem. The first operation is either a column deletion or a merge of two columns. In Lemmas 3.9 and 3.10 we show that if a deletion/merge *could be done* at time T_0 in a reduction sequence then *there is* an equivalent sequence in which the deletion/merge is done at T_0. Applying the lemmas for time $T_0 = 0$ will prove the theorem since the first reduction instance is already present in the problem.

Lemma 3.9 *If* v *is a column deletable at time* T_0 *in an exhaustive sequence of reductions, then there is an equivalent sequence in which* v *is deleted at* T_0*.*

proof The column v can be deleted or merged at time T_0, or nothing happens to it. If it is deleted, we are done.

If it is merged then the merged column is deletable at time $T_0 + 1$ (Claim 3.5). The matrix has one less column at time $T_0 + 1$, so by induction there exists an equivalent sequence in which the merged column is deleted at time $T_0 + 1$. By replacing the merge of v and the other column and then the deletion of the merged column by the deletion of v followed by the deletion of the other column (part 1 of Claim 3.8) we obtain an equivalent sequence in which v is deleted at time T_0.

If nothing happens to column v at time T_0 then v is still deletable at time $T_0 + 1$ (Lemma 3.2). Applying the inductive statement there exists an equivalent sequence in which v is deleted at time $T_0 + 1$. We can swap the first two operations (Lemma 3.4) to obtain an equivalent sequence in which v is deleted at time T_0. ■

Note that while the above proof is existential, it is easy to give an algorithm that constructs the equivalent sequence. Indeed, if v is deleted at some time in the sequence of reductions then the deletion of v can be bubbled back to time T_0 (Lemma 3.4) and we are done. Otherwise, since the sequence is exhaustive, v will be merged with another column at some later time (Lemma 3.2). The merged column is deletable (Claim 3.5), so in turn it will be either deleted or merged further, and so on. The "supercolumn" V that contains v will be deleted sooner or later since the reduction sequence is exhaustive and finite.

Now consider the time when V is deleted. V became deletable right after it was merged from two columns, V' (containing v) and some column z. By Lemma 3.4, the deletion of V can be bubbled back to be right after the merge of V' and z. Then, since V' is deletable, $merge(V', z)$ $del(V)$ can be replaced by $del(V')$ $del(z)$ (part 1 of Claim 3.8). Continue this procedure with V' until the deletion of v appears in the equivalent sequence, and then bubble this operation back to time T_0.

Lemma 3.10 *If v and w are mergable at time T_0 in an exhaustive sequence of reductions, then there is an equivalent sequence in which v and w are merged at T_0.*

proof At time T_0 the two columns are either merged, one of them is deleted, one of them is merged with a third column or nothing happens to them. If they are merged with each other then we are done.

If one of the two columns is deleted then the other column becomes deletable at time $T_0 + 1$ (Claim 3.6), so there exists an equivalent sequence in which the other column is deleted at time $T_0 + 1$ (Lemma 3.9). Then applying part 2 of Claim 3.8 shows that we are done.

If one of the columns is merged with a third column (say v is merged with some column z) then vz and w are mergable or both are deletable at time T_0+1, depending on whether z was orthogonal to w or not (Claim 3.7). If they are mergable then, by the inductive statement, there exists an equivalent sequence in which vz and w are merged at time $T_0 + 1$. Applying part 3 of Claim 3.8 shows that we are done. Otherwise, there exists an equivalent sequence in which vz is deleted at time $T_0 + 1$ and w is deleted at time $T_0 + 2$ (Lemma 3.9). Now apply part 4 of Claim 3.8 to see that we are done.

If nothing happens to the two columns at time T_0 then they are still mergable at time $T_0 + 1$. By the inductive statement there exists an equivalent sequence in which these columns are merged at time $T_0 + 1$. Swapping the first two operations (Lemma 3.4) we get an equivalent sequence in which the two columns are merged at time T_0. ■

We can devise a constructive algorithm as in the previous case. If the two columns are merged at any time in the sequence then this operation can be bubbled back to time T_0. Otherwise one of the two columns is deleted or merged with another column (Lemma 3.2). Now columns containing v and w can be further merged until one of the supercolumns V and W is deleted or V and W are merged together.

Since the reduction sequence is exhaustive and finite, one of these two cases must happen eventually.

Assume that one of the supercolumns is deleted, say V. When this happens, W becomes deletable as well (Claim 3.6), and, as in Lemma 3.9, we can modify the sequence so that W is deleted immediately. If V and W are orthogonal then they are mergable at the time when V is deleted (Claim 3.7) thus we can replace $del(V)$ $del(W)$ with $merge(V, W)$ $del(VW)$ (part 2 of Claim 3.8) and default to the case in which the supercolumns are merged together.

Otherwise V and W are nonorthogonal, which means that there was a time when one of the supercolumns was merged with a column that was nonorthogonal to the other supercolumn (since then both of the columns could have been merged with other, different columns). After this merge both of the supercolumns became deletable (Claim 3.7), thus we can find an equivalent sequence in which V and W are deleted right after this merge. Assume that this merge produced V from V' and z. Since V' and W are orthogonal, z must be nonorthogonal to W; thus $merge(V', z)$ $del(V)$ $del(W)$ can be replaced by $merge(V', W)$ $del(V'W)$ $del(z)$ (part 4 of Claim 3.8) and we can default to the case in which the supercolumns are merged together.

Now assume that V and W are merged together. These columns became mergable right at the time when V and W were created (whichever happened later). Suppose V is the column that was created later by merging V' (a column containing v) and z. Since W already existed when this merge happened, the merge of V and W can be bubbled back to immediately follow the merge of V' and z. $merge(V', z)$

$merge(V,W)$ can be replaced by $merge(V',W)$ $merge(V'W,z)$ (part 3 of Claim 3.8) since V' and W are mergable (Claim 3.7) and z and W must be orthogonal if V and W are. We now continue this procedure with V' and W until the merge of v and w appears in the sequence.

3.3 How can new instances arise?

In the previous sections we have described six reduction methods and have shown that the order in which they are applied to a problem instance does not matter, provided the reductions are carried out exhaustively. For an efficient implementation we also need to know which reductions can lead to (and to what kind of) new reduction instances, so that we can avoid checking for reduction instances unnecessarily.

Consider first the three operations (SUMC, CLEXT and MERGE) that the six reduction methods can be replaced with. In Table 3.1 below entry (i,j) indicates whether reduction operation i can cause a new instance of type j.

It is clear that a new SUMC instance cannot be created by column deletion, so SUMC or CLEXT cannot create a new SUMC instance. On the other hand, SUMC can arise as the result of a MERGE when a merged column becomes the sum of some already existing columns.

It is possible to create a new CLEXT instance by column deletion, when all but one "bad" column in a row clique are nonorthogonal to a given "outside" column, and this bad column is deleted. This deletion cannot be a SUMC since all the

Table 3.1: Impact of reductions

	SUMC	CLEXT	MERGE
SUMC	NO	NO	YES
CLEXT	NO	YES	YES
MERGE	YES	YES	YES

summands that make up the deleted column must be orthogonal to the outside column and one of them must be in the row clique. On the other hand, it is easy to construct an example where the bad column in the row clique is deleted via a CLEXT operation. Merging columns can also create new CLEXT instances; either by merging the outside column with some other column and thus making it nonorthogonal to all columns in a row clique, or by merging the bad column in the row clique with another column and thus making it nonorthogonal to the outside column.

New MERGE instances can be created by all three operations; by deleting an "extra" column (via SUMC or CLEXT) so that two rows will differ by exactly two columns, or by merging two extra columns (based on two rows, one of which is different from the rows in the new instance).

Based on the observations that enabled us to substitute the original six reduction methods with three, we can extend the above table to include all the reduction methods.

DUPC is a special case of SUMC. New DUPC and SUMC instances can be created only by merging columns; deleting columns or duplicate rows has no influence here.

As a new CLEXT instance cannot be created by a SUMC operation, it cannot be created by a DUPC either. On the other hand, it can be created by any of the other four reduction operations.

A new DOMR instance is created by column deletion if the only column that intersects the shorter but not the longer row is removed. This column cannot be deleted by a DUPC or SUMC operation since a copy or summand of the deleted column that intersects the short but not the long row would remain in the problem. On the other hand, DOMR can be created by any of the other four operations.

A new SINGL instance can be created by any operation except by merging two columns (in this case the two rows based on which the columns are merged can have only one nonzero in them, but then the SINGL instance is already present). A SINGL can arise however by DTWO when the two columns are deleted.

A new DTWO instance can be created by any of the reduction operations.

3.4 Implementation

Our primary goal in the implementation was to achieve the most reduction in a reasonable amount of time. In our experience maximal reduction can usually be reached reasonably quickly if SUMC is not considered. Running SUMC to its full extent is prohibitively expensive for all but the smallest problems. Nevertheless, we have implemented an adaptive (limited) strategy for SUMC that proves to be both

Table 3.2: Impact of reductions – for all six instances

	DUPC	SUMC	CLEXT	DOMR	SINGL	DTWO
DUPC	NO	NO	NO	NO	YES	YES
SUMC	NO	NO	NO	NO	YES	YES
CLEXT	NO	NO	YES	YES	YES	YES
DOMR	NO	NO	YES	YES	YES	YES
SINGL	NO	NO	YES	YES	YES	YES
DTWO	YES*	YES*	YES	YES	YES**	YES

* only if the two columns are merged
** only if the two columns are deleted

effective and efficient. We are not aware of any other implementation that uses the SUMC reduction.

We approach the question of efficiency from three directions. First, for each of the six reduction types we have implemented a *module* that invokes a *reduction function* and a matrix compression subroutine repeatedly. The reduction function scans through all the columns or rows (or pairs of rows) of the current matrix to identify all instances of the particular reduction type. The deletable/mergable columns and rows are only marked during the scan, they are physically removed later during matrix compression. The use of reduction functions decreases the average time spent on examining a column or row for a reduction instance since some data structures commonly used by all columns/rows can be prepared in advance. Also, time is saved by not compressing the matrix every time a deletable column/row is identified.

Second, utilizing the results of Section 3.3, the modules are organized into *strategies*. Depending on our requirements, we can create strategies that achieve maximal reduction, or that cut down on the running time by limiting the use of the more expensive modules.

Third, the reduction functions are implemented assuming that the columns of the matrix are arranged in lexicographically increasing order (vector a is lexicographically smaller than vector b if the first nonzero entry of $b - a$ is positive). This allows us to use special techniques that speed up reduction instance identification considerably (see the description of DUPC, SUMC and CLEXT reduction functions below). The ordering is carried out before any reduction is started and then maintained throughout the computation. The initial ordering is reasonably inexpensive to obtain, and it takes very little effort to maintain. Removing columns from the matrix obviously does not destroy the ordering, but extra care must be taken when marking rows for deletion, or when inserting merged columns into the matrix.

In the remainder of this section we discuss the reduction modules and then the strategies in detail. At the end we summarize our computational results. Details about the main data structure and the parameters used in Reduce() are given in Appendix B.

3.4.1 Reduction modules

In this section we describe the reduction modules in general, then we give details about the implementation of the individual functions. But we first give a few more words about the matrix compression module.

As we have mentioned before, when a column or row is "deleted" during one of the reduction operations, it is not removed physically from the matrix right away since rewriting (possibly) the entire matrix would be too costly. Instead, the column or row is marked for removal, and the matrix is periodically updated; that is, the marked columns and rows are removed and the matrix (along with the objective vector) is compressed. This matrix compression is a module in itself that may be invoked from other modules or from the reduction strategies. Note that removing the marked columns and removing the marked rows are two independent tasks, so the two updates have been separated into two different modules. This is reasonable since, as we have seen earlier, the removal of duplicate rows could wait until the very end of the reduction without influencing the outcome. However, even though marked rows are skipped when the rows of the matrix are enumerated, it may be effective to remove them, as columns become shorter and thus operations involving entire columns (like determining orthogonality) become more efficient.

A general reduction module takes the problem matrix and objective vector as input along with the vectors ONES and MERGES, and a parameter `repeat_fraction` (see Figure 3.2). A reduced matrix (with some rows possibly marked for deletion), updated ONES and MERGES vectors, and the feasibility status of the problem (feasible, infeasible or feasibility not yet known) are returned. Each module invokes a particular reduction function that enumerates every column or row (or pairs of rows) of the matrix to see if the corresponding reduction operation can be applied. Columns marked for deletion are removed and the matrix is compressed if there was any reduction. The reduction function (and the compression afterwards) is repeated if at least `repeat_fraction` fraction of the columns in the current matrix are marked for deletion by the most recent application of this function.

We claim that in order to decide whether or not to repeat the reduction function it is enough to check whether columns were marked for deletion (even if both rows and columns could be marked). To see this assume that a reduction function marked some rows but no columns for deletion. Then it is clear that all the rows marked are duplicates of some other rows that remain in the problem. Thus, since removal of duplicate rows does not create new reduction instances, only duplicate rows already in the matrix could be deleted if the reduction function were repeated. However, as we will discuss for the individual reduction functions, in both of the cases when duplicate rows but no columns can be marked for deletion (namely, DOMR and DTWO) all pairs of unmarked rows are checked, so all the duplicate rows will be marked with one call of the reduction function. Thus the reduction function need not be repeated.

Note that the reduction functions in the DUPC and SUMC modules need not be repeated since these reduction operations do not lead to new instances of the same types, as we have seen in 3.3.

Finally, observe that if `repeat_fraction` is set to 0 then the reduction function will be repeated until no further reduction of this type is possible. On the other hand, the reduction function will be invoked only once if this parameter is set to 1. For each reduction function there are separate `repeat_fraction` parameters; they could be varied from the reduction strategies that invoke the modules.

```
Reduction module
  Input:   A, c, ONES, MERGES, repeat_fraction
  Output:  A', c', ONES', MERGES', feasibility status

  do {
     invoke reduction function
     if no reduction, return
     if infeasibility is detected in reduction function, return
     remove columns that are marked for deletion, compress matrix
     if matrix has no columns left then problem is feasible, return
  } while (at least repeat_fraction fraction of the columns
           have been deleted)
end
```

Figure 3.2: Description of a general reduction module

The DUPC reduction function

In the DUPC reduction function the columns of the matrix are enumerated one by one, from lexicographically smaller to larger. Due to the ordering, identical columns are located next to each other in the matrix. When duplicate columns are discovered then all but the cheapest column is marked for removal. To make this comparison even easier, identical columns are ordered from cheapest to most expensive during the initial lexicographical ordering (if two columns are identical in the matrix then the one with the smaller objective coefficient is considered to be lexicographically smaller). Therefore the first of the identical columns will be the one kept. Another way of detecting duplicate columns is to use hashing ([HP93]).

The SUMC reduction function

The SUMC reduction function enumerates columns of the matrix from left to right, for each column v trying to find columns that sum up to v with combined cost less than that of v. Any column which could be a summand for v is lexicographically smaller than v itself, so the lexicographical ordering of the columns insures that all potential summands lie left from v in the matrix (thus they have already been processed when v is being examined). Columns with the first nonzero at the same position as that in v are considered one by one, from right to left, as the first summand. When a column can be subtracted from v (with nonnegative remainder) then this method is continued for the remainder recursively (although the remainder is usually not a column in the matrix itself, its would-be position is determined and the process is continued from there). If there is no remainder left then costs are compared. If the sum is the more expensive then we backtrack, forcing the last summand to be the remainder. Otherwise, if the sum is the cheaper then v can be marked for deletion and the next column in the matrix is considered. However, since marked columns are skipped by the function, the sum replacing this marked column would need to be determined again if the column were to be a summand later. Therefore, to make use of the columns that could be marked for deletion by this reduction function, the columns are marked with a temporary marker and their objective function coefficient is replaced with the cost of the sum. Note that we are not looking for a cheapest sum to replace columns, we continue with the next column as soon as *any* sum cheaper than v is found. The recursion stops when a sum cheaper than v is discovered or there are no more columns as potential first summands for v. After each column is processed, columns marked temporarily are marked for deletion.

Although the use of temporary markers saves some time since already discovered sums are not searched for again, the above described algorithm still runs in time exponential in the size of the matrix. Therefore we introduced techniques that significantly reduce the running time by restricting the group of columns examined and by limiting the scope of search for suitable summands. We might not find all the SUMC reduction instances in the current matrix this way, so our implementation of the reduction function can be repeated. In order to compare columns we compute their cost per length ratios (the cost of the column over the number of ones in it). Then we examine only the (in this sense) most expensive columns, and only if a significant fraction of these are marked for deletion by SUMC will we continue with the next most expensive set of columns. This method prohibits too many columns from being examined when only a few could be deleted with SUMC. Also, examining the most expensive columns is only a heuristic guess; a group of columns could be chosen based on different criteria as well. The search for summands is limited by restricting the depth of recursion to a small number and by forbidding columns whose cost per length ratio is much larger than that of the remainder, to become summands. Note that limiting the depth of recursion limits the number of summands, though not necessarily to the same number since temporarily marked columns which can be expressed as sums of other columns can be summands themselves.

Although the above enhancements speed up SUMC considerably, it still remains slow in comparison to the other reductions. A good estimate on the running time of SUMC for a particular column can be obtained by observing that columns which are candidates to be first summands must have a common first row with our column (that is, they must lie in the same *block* of the lexicographically ordered matrix). Thus half of the columns of our column's block need to be considered on average as first summands. After the first summand is subtracted from the column, the same is true for the remainder. Therefore, if the depth of recursion is k, the amount of computation for one column is proportional to $(\textit{average block length})^k$. So in our implementation the depth of recursion and other parameters influencing computation time (e.g., whether expensive columns are considered as summands) are decided using the average block length. For problems with very large average block length SUMC is not even attempted.

The CLEXT reduction function

Our CLEXT reduction function enumerates the rows of the matrix one by one, for each row scanning through the columns and marking any column for deletion that extends the row's clique ([HP93] apparently do this similarly). A different approach would be to enumerate the columns of the matrix, marking a column for deletion when it extends at least one row's clique ([BC96]). Since with either of these methods every column has to be checked for nonorthogonality against all columns intersecting all rows not in the column's support, this algorithm is inefficient if it is implemented in a straightforward manner. However, the computation can be speeded up by not examining rows and columns unnecessarily, and by making the test of whether a particular column extends a row's clique more efficient. Some of our techniques rely heavily on the lexicographical ordering of columns.

First of all, observe that if a column intersects only one row then the corresponding row clique is maximal. Due to the lexicographical ordering of columns, columns of length one are very easy to spot because they are the lexicographically smallest columns intersecting their rows. Another observation is that if a row has a column with only two ones in it then all columns which could extend the row's clique must intersect the other row determined by the column with two ones. So only those columns that intersect the "other row" need to be considered, which is a significant reduction in the number of columns for sparse matrices. Also, if there are several "length two" columns intersecting the row then only columns that intersect *all* the "other rows" need to be considered. Since it would be costly to construct the intersection of several rows explicitly, the shortest of these other rows is chosen instead, and we make sure that columns which are candidates for extending the row clique are tested against the "length two" columns first. A third observation that further restricts the set of candidate columns is the following. Two columns are surely orthogonal if the last row which the first column intersects comes earlier in the matrix than the first row for the second column. Thus a column cannot extend a row's clique if this is true for the column and *any* of the columns in the row's support. Moreover, the row itself does not need to be included in the check since candidate columns do not intersect the row itself. Therefore, by determining the last (or second-to-last if the row to be extended is the last) row for each column in the row and then taking the earliest of these last rows, columns whose first row is later than the earliest last row need not be considered. Due to the lexicographical ordering of columns, all columns that precede the first column of the earliest last row can be skipped, that is, the enumeration of columns can begin with the first column of the earliest last row. Similarly, the last first row of columns intersecting a row can be determined, thus columns not intersecting the row whose last column comes earlier than the last first row need not be considered since they cannot be nonorthogonal to all columns in the row.

We have organized our CLEXT reduction function so that the rows of the matrix are enumerated in an outer loop. This enables us to prepare the row so that the one-by-one tests for the many columns not intersecting this row will be more efficient. First the row is sampled and the candidate columns are tested against the columns in the sample. Only if a candidate is nonorthogonal to every column in the sample will testing continue for the entire row. To make the test more effective, the "length two" columns intersecting the row are listed first in the sample; the rest is chosen randomly. The length of the sample is proportional to the size of the row's support; the factor of proportionality is regulated through parameters.

The DOMR reduction function

The DOMR reduction function considers each pair of rows and examines whether the shorter row dominates the longer; that is, whether the support of the shorter row is a subset of the support of the longer row. When a pair of dominating rows is found, columns whose indices are in the longer but not the shorter row's support are marked for deletion, along with one of the rows. Columns marked for deletion are removed from the matrix only after a full pass through the row pairs, so DOMR instances not yet in the matrix at the time when the function is invoked might not be discovered. This function also detects when two rows are duplicates of each other (the two supports are identical). Since this function enumerates all the row pairs in the matrix, the test for domination between two rows must be done efficiently. Our data structures provide us with ordered lists for the supports, enabling a fast comparison. Also, checking whether the first and last entries of the shorter support are between the first and last entries of the longer support before comparing the two supports entry-by-entry eliminates the need for explicit comparison of many row pairs.

As we have mentioned earlier, care must be exercised when deleting duplicate rows so as not to destroy the increasing lexicographical order of the columns. We claim that if the duplicate row which comes later in the matrix is deleted then the ordering will be maintained. Assume that i and j are two identical rows so that i comes first in the matrix, and that v and w are two columns so that v is lexicographically smaller than w. The only way for w to become lexicographically smaller than v by deleting one of the rows is if v and w are identical up to the removed row, v has a 0 while w has a 1 in this row, and v has a 1 while w has a 0 in the next row in which the two columns differ. This could occur if i is the row removed. On the other hand, if j is the row to be removed then, since the two rows are identical, the two columns would not be the same up to row j, contradicting our assumption. This shows that always removing the second of the two rows is justified.

The SINGL reduction function

The SINGL reduction function enumerates the rows of the matrix one-by-one. When a row with a single one in it is found, the index of the only intersecting column is added to ONES, and the consequences of this fixing are propagated; that is, rows intersecting this column are taken one-by-one, and for each row, the columns in its support and the row itself are marked for deletion. The implementation of this reduction function is straightforward, and the function itself is very fast.

The DTWO reduction function

The DTWO reduction function enumerates all pairs of rows, and for each pair checks whether the supports of the rows are of equal size and if so, whether they differ only in two entries. If this is the case, the two columns corresponding to these entries are compared, and if they are nonorthogonal then they are simply marked for deletion, otherwise their indices are listed in MERGES and the columns themselves are marked for deletion. Also, similar to DOMR, the later of the now identical rows is marked for deletion. The merged column will be constructed and inserted into the matrix when the matrix is compressed after the reduction function returns. As with the DOMR reduction function, this function will detect duplicate rows as well. Maintaining row supports as ordered lists makes a fast and straightforward implementation possible.

We have also implemented the module DUPR that marks duplicate rows for deletion. Although duplicate rows could be eliminated by DOMR and DTWO, this routine can be useful when our goal is the fast elimination of duplicate rows.

3.4.2 Reduction strategies

Reduction strategies are algorithms comprised of reduction modules. We have implemented two main reduction strategies, one that achieves maximal reduction, and another that achieves less reduction but generally executes much more quickly. Although all six reduction modules are included in both reduction strategies, individual reduction modules can be turned off through parameters.

Flags are introduced for the reduction modules indicating whether the previous application of the corresponding module was successful; that is, whether the reduction module has removed any columns. (As we have argued above, removing rows only will not create new reduction instances.) We note here that since SUMC is the least efficient among the reduction procedures, we invoke it only after the other modules are finished, so that the input matrix is as small as possible.

In the maximal reduction strategy the five reduction modules DUPC, SINGL, DOMR, DTWO and CLEXT are invoked (in this order) within a loop that repeats until no further reduction is possible. Each module is invoked at least once, but repeated only if there was success in other modules that might produce new instances for it (Table 3.2). Rows marked for deletion are removed from the matrix after we exit from the loop, and also after the DOMR and DTWO modules if at least 10% of the rows are marked for deletion. Within the modules the reduction functions are repeated until no more reduction of the type is possible (that is, `repeat_fraction` is zero). The SUMC reduction module is invoked after the loop (and the possible compression of the matrix); repetition of the reduction function is allowed. SINGL and DTWO are invoked if there was any reduction in the SUMC module, then the whole process (the five reduction modules followed by SUMC, a SINGL and a DTWO) repeats until no more reduction is possible.

Out of the five traditionally implemented modules CLEXT is by far the most expensive, and DOMR can take a significant amount of time when the number of rows is large. Therefore we try to invoke these modules sparingly in the fast reduction. Also, we make use of the `repeat_fraction` parameters to limit the number of times the reduction functions are invoked within the modules. In addition, another parameter, a global `repeat_fraction`, is introduced which does not allow the main loop in the strategy to repeat unless at least this fraction of the columns has been marked for deletion by the modules during the most recent pass through the loop. DUPC and SINGL are invoked only twice, first at the very beginning and then after the loop. The second DUPC is executed only if some columns have been merged. SINGL is repeated. Since DTWO is inexpensive, the module is invoked frequently, repetition of the reduction function is enabled. DOMR and CLEXT are repeated in a loop until one of these deletes significantly fewer columns than the other did the previous time it was invoked. On the other hand, if one of the modules deletes significantly more columns than the other did, then the other module is forced to repeat. The loop is repeated only if the overall number of columns marked for deletion is at least as much as stipulated by the global `repeat_fraction` parameter. Rows marked for deletion are removed after each call of the DOMR module. Then SUMC is invoked at the end of this strategy. We call this strategy the “fast” strategy.

In Section 3.4.4 the two strategies are compared in detail.

3.4.3 The Reduce() function

These strategies are accessible through a function called Reduce(). This function returns `TRUE` if the function completed successfully, and `FALSE` otherwise. Reduce() takes as an input and returns as an output a pointer to a structure containing, among other things, a column and row ordered version of the problem matrix, a set of parameters which determine the way the reduction is carried out, an array containing names of variables that are fixed to one (ONES), an array with names of variables whose columns have been merged (MERGES), and an entry that indicates the feasibility status of the problem.

When Reduce() is invoked, only part of this data structure (part of the column ordered matrix that describes the problem matrix, and the parameters) has to be filled, the rest will be added at the beginning of the function. After the local data structures are initialized, the columns of the matrix are ordered into lexicographically increasing order. Variables whose names are listed in ONES are fixed to one, and the effect of this fixing is propagated throughout the matrix. If any column or row is marked for deletion then the matrix is compressed (columns or rows might have been marked before the function was invoked). Then the reduction strategy indicated by the parameters is invoked and finally the local data structures are dismantled. Names of variables fixed to one during reduction are appended to the array ONES, and similarly names of merged variable pairs are appended to MERGES. Figure 3.3 gives an outline of Reduce().

If the matrix can be reduced to nothing, an optimal solution to the original problem can be deduced from the arrays ONES and MERGES. If there is a row

```
Reduce()
  Input:   pointer to the main data structure
  Output:  pointer to the main data structure
  Return value: TRUE if succeeded, FALSE otherwise

  initialize data structures (fill out optional fields)
  order columns of input matrix (lex increasing)
  fix variables listed in ONES to one
  compress matrix if necessary
  invoke reduction strategy
  clean up
end
```

Figure 3.3: Outline of the Reduce() function

with an empty support at any stage of the computation the original problem is infeasible. Otherwise the feasibility status of the problem could not be determined during reduction.

The main data structure and the parameters are described in Appendix B.

3.4.4 Computational results

In our test runs we have tried two main strategies: maximal and fast, each with the SUMC reduction function once enabled and once turned off (as we have indicated above, SUMC was invoked only after the other reductions had finished). Preliminary testing revealed that it is not worth to repeat the other reductions after SUMC since new reduction instances were created only a few times, and these were always isolated SINGL instances (the column intersecting the singleton row is also a singleton).

Four tests were conducted on the SP for all four sets of data (Section A.2): the maximal and fast strategies with SUMC disabled and our adaptive SUMC applied to the outputs of both. Tables 3.3, 3.4, 3.7 and 3.9 summarize our results for the maximal strategy and the SUMC following it while Tables 3.5, 3.6, 3.8 and 3.10 do the same for the fast strategy. For each problem instance the tables contain its name, original size (number of columns and rows); the lexicographical ordering time; the problem's size after the maximal/fast strategy (SUMC disabled) along with the running time; the time spent in CLEXT and DOMR routines (with their multiplicity); the average block length (Section 3.4.1), the percentage of columns deleted by and the running time of the adaptive SUMC routine; the final size of the reduced problem with the percentage of nonzeros deleted during the entire process. The experiments were carried out on a thin node of the SP (see Section A.1 for details about the architecture).

The input parameter settings are based on test runs. In the CLEXT reduction function 3% of each row's support was sampled (at least 10 but at most 200 columns were chosen). In the SUMC reduction function average block length was computed using the longest blocks that made up 90% of the columns (so that the very short blocks would not decrease the average too much). For problems with average block length less than 100 the depth of recursion was restricted to 3; for the rest of the problems, it was restricted to 2. Columns were considered in groups of 10% of the size of the matrix, and at least 20% (up to 50% for problems with large average block length) of these columns had to be deleted to continue with the next group of columns. Columns with cost per length ratio more than 1.5 times (down to 1.0 times for problems with large average block length) the remainder were not considered as prospective summands. SUMC was not even attempted for problems of average block size larger than 500. Note that SUMC was invoked with the same parameters on the output of both the maximal and fast strategies.

As described earlier (Section 3.4.2), the maximal reduction repeated each reduction function until there was no more reduction. In the fast strategy SINGL and DTWO were always repeated; DOMR was repeated if at least 5% of the columns were deleted with its previous application, and CLEXT was repeated when this percentage was at least 12.5%. The outer loop (invoking DOMR, CLEXT and DTWO repeatedly) was repeated if at least 12.5% of the columns were removed during the last pass through the loop. In addition to this, a heuristic version of CLEXT was run in the fast strategy; rows longer than the average row length were skipped (unless a "length two" column is present).

Our fast strategy (without SUMC) compares very well with the maximal reduction (without SUMC). As we expected, the execution time of CLEXT and DOMR dominate the total running time of both strategies. The fast strategy is considerably faster when it is able to cut down on the number of times these two reduction functions are invoked. In addition to the speed-up, the fast strategy has achieved the same reduction on many of the problems considered; the percentage of nonzeros deleted compared to the maximal strategy was worse by more than 2% on only 1 Set 1 and 4 Set 4 problems. Since these two strategies produced almost identical results, SUMC behaved the same way in the last two tests. As we have discussed earlier, our adaptive SUMC routine is very restricted. Nevertheless, when it is attempted at all, it can be very effective on certain sets of problems (`nw` and `v*` problems, deleting up to 85% of the columns). This fact might be attributed to the column generation techniques used for these problems.

For the problems in Set 1 we compare our results with those of Hoffman and Padberg ([HP93]) and Borndörfer ([Bor97]). Hoffman and Padberg implemented an equivalent (in terms of reduction) of DUPC, CLEXT, DOMR, SINGL and DTWO. However, they cut short the more time consuming routines (like the CLEXT and DOMR/DTWO equivalents) based on heuristics. Our maximal strategy (with SUMC disabled) achieves at least as much reduction as they did except for two problems (`nw26` and `nw24`, most likely typos in their paper). SUMC applied after the maximal or fast strategies further reduces the number of columns by at least 25% on 26 of the 43 `nw` problems.

We have learned about Borndörfer's work only after our implementation of Reduce() and this chapter were completed. His implementation contains two additional reduction methods that we were not aware of before: the first substitutes for singleton columns (columns with only one nonzero) in certain situations and the second removes all columns from the symmetric difference of two rows' supports if the entire symmetric difference is contained in a third row's support. He did not implement, however, the DTWO procedure, and applied only a limited version of CLEXT (considered only rows with supports not larger than 16). Our fast strategy (SUMC disabled) usually deletes a few more columns but a few less rows than his method on the Set 1 problems with the exception of `nw16` which is reduced to nothing by his method.

Borndörfer's method was also applied to the problems in Set 3 ([BGKK97]).

While his method deletes about 10% more rows on these problems than our fast strategy (SUMC disabled), our method removed more columns from the `v04*` and `t04*` problems but less from the `v16*` problems. Note that for the `t17*` problems the only reduction possible was DUPC. SUMC further reduced the number of columns by at least 30% in 7 of the 14 `v*` problems (although it was rather time consuming for 3 of these problems).

Our running times cannot be directly compared to those of the other two groups since they provide only a cumulative time for all reductions during a Branch-and-Cut algorithm.

In our final runs we used the fast strategy with SUMC for the first reduction. The reduced problems were saved into files, this is what the feasible solution heuristics and the Branch-and-Cut algorithm use as input. We use the fast strategy without SUMC everywhere else.

Table 3.3: Maximal reduction w/o SUMC followed by one SUMC, Set 1, part 1

	Original		lex	Maximal no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
aa01	8904	823	0.07	7532	607	7.29	4.86 (4)	2.28 (6)	27.01	0.04	0.04	7529	607	34.95
aa02	5198	531	0.03	3846	360	1.47	1.07 (4)	0.35 (5)	24.06	0.03	0.02	3845	360	41.42
aa03	8627	825	0.07	6694	537	8.06	5.53 (6)	2.31 (6)	27.80	0.07	0.04	6689	537	42.54
aa04	7195	426	0.05	6122	342	1.43	1.05 (2)	0.34 (4)	40.86	0.02	0.03	6121	342	27.87
aa05	8308	801	0.06	6235	521	5.79	3.24 (4)	2.37 (9)	26.00	0.18	0.02	6224	521	44.88
aa06	7292	646	0.06	5862	488	5.97	4.85 (6)	0.97 (7)	28.07	0.20	0.03	5850	488	33.15
kl01	7479	55	0.06	5915	47	0.83	0.78 (2)	0.02 (3)	300.00	0.00	0.04	5915	47	33.56
kl02	36699	71	0.33	16542	69	0.64	0.54 (1)	0.01 (1)	683.77	0.00	0.00	16542	69	55.21
nw01	51975	135	0.09	49903	135	3.10	1.29 (2)	1.68 (2)	619.27	0.00	0.00	49903	135	3.82
nw02	87879	145	0.18	85256	145	5.50	2.02 (2)	3.26 (2)	963.40	0.00	0.00	85256	145	2.74
nw03	43749	59	0.09	38956	53	0.73	0.01 (1)	0.49 (2)	1617.05	0.00	0.00	38956	53	12.35
nw04	87482	36	0.20	46189	35	1.46	1.06 (1)	0.17 (2)	2666.06	0.00	0.00	46189	35	47.98
nw05	288507	71	0.69	202593	62	4.02	0.03 (1)	2.86 (2)	7014.27	0.00	0.00	202593	62	30.93
nw06	6774	50	0.01	5956	38	0.10	0.00 (1)	0.05 (2)	321.47	6.28	1.52	5582	38	30.56
nw07	5172	36	0.02	3105	34	0.03	0.00 (1)	0.02 (2)	189.60	45.70	1.04	1686	34	69.52
nw08	434	24	0.00	352	21	0.00	0.00 (1)	0.00 (2)	29.27	72.73	0.01	96	21	83.36
nw09	3103	40	0.01	2301	38	0.02	0.00 (1)	0.02 (2)	138.07	58.02	0.98	966	38	71.12
nw10	853	24	0.00	655	21	0.01	0.00 (1)	0.00 (2)	54.36	85.50	0.02	95	21	92.92
nw11	8820	39	0.02	6482	34	0.07	0.00 (1)	0.04 (2)	395.13	74.58	12.62	1648	34	82.72
nw12	626	27	0.00	451	25	0.00	0.00 (1)	0.00 (2)	31.38	74.06	0.01	117	25	91.45
nw13	16043	51	0.03	10903	50	0.14	0.00 (1)	0.08 (2)	380.85	4.30	0.09	10434	50	35.93
nw14	123409	73	0.24	95172	70	1.92	0.01 (1)	1.38 (2)	2681.50	0.00	0.00	95172	70	23.12
nw15	467	31	0.01	405	29	0.03	0.03 (2)	0.00 (2)	28.69	0.00	0.00	405	29	14.66
nw16	148633	139	0.29	138947	135	9.09	0.03 (1)	8.23 (2)	1928.45	0.00	0.00	138947	135	7.72
nw17	118607	61	0.26	78173	54	1.67	0.01 (1)	1.14 (2)	3716.21	0.00	0.00	78173	54	35.88
nw18	10757	124	0.03	8439	110	0.33	0.00 (2)	0.26 (2)	161.98	4.83	0.83	8031	110	31.88
nw19	2879	40	0.00	2134	32	0.02	0.00 (1)	0.02 (2)	140.07	38.00	0.57	1323	32	63.41

Table 3.4: Maximal reduction w/o SUMC followed by one SUMC, Set 1, part 2

	Original		lex	Maximal no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
nw20	685	22	0.00	536	22	0.02	0.02 (2)	0.00 (2)	44.36	33.02	0.06	359	22	49.11
nw21	577	25	0.00	421	25	0.02	0.01 (2)	0.01 (2)	32.08	49.88	0.02	211	25	68.70
nw22	619	23	0.00	521	23	0.02	0.02 (2)	0.00 (2)	43.73	34.93	0.02	339	23	46.51
nw23	711	19	0.00	423	18	0.02	0.01 (2)	0.00 (2)	47.75	41.37	0.14	248	18	69.22
nw24	1366	19	0.00	926	19	0.02	0.01 (1)	0.00 (1)	106.38	65.77	0.30	317	19	79.17
nw25	1217	20	0.00	844	20	0.03	0.03 (1)	0.00 (1)	87.11	61.26	0.15	327	20	76.50
nw26	771	23	0.00	468	21	0.02	0.02 (2)	0.00 (2)	53.00	36.54	0.10	297	21	63.87
nw27	1355	22	0.00	817	22	0.05	0.04 (2)	0.00 (2)	73.80	48.84	0.13	418	22	73.76
nw28	1210	18	0.00	582	18	0.05	0.04 (2)	0.00 (2)	75.29	27.66	0.33	421	18	68.53
nw29	2540	18	0.01	2034	18	0.02	0.02 (1)	0.00 (1)	242.38	16.81	1.56	1692	18	32.88
nw30	2653	26	0.01	1877	26	0.10	0.08 (2)	0.01 (2)	172.80	50.19	1.08	935	26	66.11
nw31	2662	26	0.00	1728	26	0.15	0.14 (2)	0.00 (2)	173.00	36.34	0.60	1100	26	59.49
nw32	294	19	0.00	251	18	0.00	0.00 (1)	0.00 (1)	25.56	43.82	0.02	141	18	54.24
nw33	3068	23	0.01	2308	23	0.26	0.25 (2)	0.00 (2)	239.00	2.64	0.06	2247	23	27.00
nw34	899	20	0.00	718	20	0.02	0.02 (2)	0.00 (2)	72.11	42.76	0.38	411	20	58.00
nw35	1709	23	0.00	1191	23	0.15	0.14 (2)	0.00 (2)	99.73	47.69	2.05	623	23	63.52
nw36	1783	20	0.00	1244	20	0.16	0.16 (2)	0.00 (2)	146.25	1.37	0.04	1227	20	33.62
nw37	770	19	0.00	639	19	0.00	0.00 (1)	0.00 (1)	59.70	50.55	0.46	316	19	62.07
nw38	1220	23	0.00	723	21	0.22	0.22 (3)	0.00 (2)	82.25	12.59	0.60	632	21	50.18
nw39	677	25	0.00	565	25	0.02	0.02 (2)	0.00 (2)	42.50	49.20	0.06	287	25	60.75
nw40	404	19	0.00	336	19	0.01	0.01 (1)	0.00 (1)	31.30	28.87	0.03	239	19	42.87
nw41	197	17	0.00	177	17	0.00	0.00 (1)	0.00 (1)	15.09	51.41	0.00	86	17	61.08
nw42	1079	23	0.00	795	23	0.07	0.07 (2)	0.00 (2)	72.00	20.38	0.17	633	23	40.39
nw43	1072	18	0.01	982	17	0.01	0.00 (1)	0.00 (1)	99.11	44.30	0.51	547	17	51.39
us01	1053137	145	22.09	339441	86	246.56	235.94 (2)	6.67 (3)	14739.43	0.00	0.00	339441	86	77.64
us02	13635	100	0.13	5766	45	2.39	2.26 (3)	0.06 (3)	472.45	0.88	4.60	5715	45	79.37
us03	85552	77	1.00	20632	50	28.66	27.91 (2)	0.37 (5)	2083.89	0.00	0.00	20632	50	82.82
us04	28016	163	0.27	4207	99	2.33	1.93 (4)	0.27 (7)	223.35	1.59	0.08	4140	99	89.07

Table 3.5: Fast reduction w/o SUMC followed by one SUMC, Set 1, part 1

	Original		lex	Fast no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
aa01	8904	823	0.07	7580	610	1.71	0.64 (1)	0.89 (2)	27.19	0.04	0.04	7577	610	34.21
aa02	5198	531	0.03	3899	361	0.44	0.13 (1)	0.23 (3)	24.40	0.03	0.01	3898	361	40.25
aa03	8627	825	0.07	6839	548	1.58	0.45 (1)	0.94 (2)	28.12	0.09	0.04	6833	548	40.41
aa04	7195	426	0.05	6143	342	0.48	0.23 (1)	0.18 (2)	41.01	0.02	0.02	6142	342	27.59
aa05	8308	801	0.06	6416	538	1.35	0.49 (1)	0.68 (2)	26.14	0.19	0.02	6404	538	42.07
aa06	7292	646	0.06	5966	497	0.97	0.52 (1)	0.32 (2)	28.41	0.17	0.03	5956	497	30.83
kl01	7479	55	0.06	5957	47	0.09	0.04 (1)	0.02 (2)	302.33	0.00	0.04	5957	47	32.95
kl02	36699	71	0.33	16542	69	0.20	0.11 (1)	0.02 (1)	683.77	0.00	0.00	16542	69	55.21
nw01	51975	135	0.09	49903	135	1.58	0.58 (1)	0.85 (1)	619.27	0.00	0.00	49903	135	3.82
nw02	87879	145	0.18	85256	145	2.69	0.78 (1)	1.66 (1)	963.40	0.00	0.00	85256	145	2.74
nw03	43749	59	0.09	38956	53	0.44	0.01 (1)	0.21 (1)	1617.05	0.00	0.00	38956	53	12.35
nw04	87482	36	0.20	46189	35	0.66	0.33 (1)	0.09 (1)	2666.06	0.00	0.00	46189	35	47.98
nw05	288507	71	0.69	202593	62	2.33	0.03 (1)	1.21 (1)	7014.27	0.00	0.00	202593	62	30.93
nw06	6774	50	0.01	5956	38	0.06	0.00 (1)	0.02 (1)	321.47	6.28	1.54	5582	38	30.56
nw07	5172	36	0.02	3105	34	0.02	0.00 (1)	0.01 (1)	189.60	45.70	1.04	1686	34	69.52
nw08	434	24	0.00	352	21	0.00	0.00 (1)	0.00 (1)	29.27	72.73	0.00	96	21	83.36
nw09	3103	40	0.01	2301	38	0.02	0.00 (1)	0.00 (1)	138.07	58.02	0.99	966	38	71.12
nw10	853	24	0.00	655	21	0.00	0.00 (1)	0.00 (1)	54.36	85.50	0.02	95	21	92.92
nw11	8820	39	0.02	6482	34	0.05	0.00 (1)	0.02 (1)	395.13	74.58	12.64	1648	34	82.72
nw12	626	27	0.00	451	25	0.00	0.00 (1)	0.00 (1)	31.38	74.06	0.00	117	25	91.45
nw13	16043	51	0.03	10903	50	0.09	0.00 (1)	0.04 (1)	380.85	4.30	0.11	10434	50	35.93
nw14	123409	73	0.24	95172	70	1.17	0.01 (1)	0.64 (1)	2681.50	0.00	0.00	95172	70	23.12
nw15	467	31	0.01	451	29	0.01	0.01 (1)	0.00 (1)	29.79	0.00	0.01	451	29	2.76
nw16	148633	139	0.29	138947	135	4.89	0.02 (1)	4.03 (1)	1928.45	0.00	0.00	138947	135	7.72
nw17	118607	61	0.26	78173	54	1.00	0.01 (1)	0.49 (1)	3716.21	0.00	0.00	78173	54	35.88
nw18	10757	124	0.03	8439	110	0.17	0.00 (1)	0.12 (1)	161.98	4.83	0.83	8031	110	31.88
nw19	2879	40	0.00	2134	32	0.02	0.00 (1)	0.00 (1)	140.07	38.00	0.57	1323	32	63.41

Table 3.6: Fast reduction w/o SUMC followed by one SUMC, Set 1, part 2

	Original		lex	Fast no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
nw20	685	22	0.00	536	22	0.00	0.00 (1)	0.00 (1)	44.36	33.02	0.06	359	22	49.11
nw21	577	25	0.00	421	25	0.01	0.01 (1)	0.00 (1)	32.08	49.88	0.02	211	25	68.70
nw22	619	23	0.00	521	23	0.00	0.00 (1)	0.00 (1)	43.73	34.93	0.02	339	23	46.51
nw23	711	19	0.00	462	18	0.00	0.00 (1)	0.00 (1)	52.38	42.64	0.20	265	18	66.72
nw24	1366	19	0.00	926	19	0.01	0.01 (1)	0.00 (1)	106.38	65.77	0.31	317	19	79.17
nw25	1217	20	0.00	844	20	0.01	0.01 (1)	0.00 (1)	87.11	61.26	0.17	327	20	76.50
nw26	771	23	0.00	514	21	0.00	0.00 (1)	0.00 (1)	57.88	37.74	0.15	320	21	60.64
nw27	1355	22	0.00	817	22	0.03	0.03 (1)	0.00 (1)	73.80	48.84	0.13	418	22	73.76
nw28	1210	18	0.00	598	18	0.02	0.02 (2)	0.00 (2)	69.75	27.26	0.40	435	18	67.27
nw29	2540	18	0.01	2034	18	0.01	0.00 (1)	0.00 (1)	242.38	16.81	1.56	1692	18	32.88
nw30	2653	26	0.01	1878	26	0.03	0.02 (1)	0.00 (1)	172.80	50.21	1.09	935	26	66.11
nw31	2662	26	0.00	1728	26	0.06	0.06 (1)	0.00 (1)	173.00	36.34	0.59	1100	26	59.49
nw32	294	19	0.00	251	18	0.01	0.00 (1)	0.00 (1)	25.56	43.82	0.01	141	18	54.24
nw33	3068	23	0.01	2308	23	0.07	0.06 (1)	0.00 (1)	239.00	2.64	0.06	2247	23	27.00
nw34	899	20	0.00	718	20	0.02	0.02 (1)	0.00 (1)	72.11	42.76	0.38	411	20	58.00
nw35	1709	23	0.00	1191	23	0.09	0.09 (2)	0.00 (2)	99.73	47.69	2.06	623	23	63.52
nw36	1783	20	0.00	1246	20	0.06	0.06 (1)	0.00 (1)	146.50	1.36	0.04	1229	20	33.48
nw37	770	19	0.00	639	19	0.01	0.00 (1)	0.00 (1)	59.70	50.55	0.45	316	19	62.07
nw38	1220	23	0.00	762	21	0.06	0.06 (2)	0.00 (2)	86.62	14.30	1.12	653	21	48.23
nw39	677	25	0.00	565	25	0.01	0.01 (1)	0.00 (1)	42.50	49.20	0.05	287	25	60.75
nw40	404	19	0.00	336	19	0.00	0.00 (1)	0.00 (1)	31.30	28.87	0.03	239	19	42.87
nw41	197	17	0.00	177	17	0.00	0.00 (1)	0.00 (1)	15.09	51.41	0.01	86	17	61.08
nw42	1079	23	0.00	818	23	0.03	0.02 (1)	0.00 (1)	73.80	23.72	0.20	624	23	41.14
nw43	1072	18	0.01	982	17	0.01	0.00 (1)	0.00 (1)	99.11	44.30	0.51	547	17	51.39
us01	1053137	145	22.09	339464	86	125.71	117.59 (1)	3.89 (2)	14740.19	0.00	0.00	339464	86	77.64
us02	13635	100	0.13	5996	45	1.45	1.33 (2)	0.04 (3)	458.08	0.87	4.81	5944	45	78.38
us03	85552	77	1.00	20632	50	9.65	8.96 (1)	0.23 (3)	2083.89	0.00	0.00	20632	50	82.82
us04	28016	163	0.27	4207	99	1.02	0.65 (2)	0.25 (5)	223.35	1.59	0.08	4140	99	89.07

Table 3.7: Maximal reduction w/o SUMC followed by one SUMC, Set 3

	Original		lex	Maximal no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
v0415	7684	1518	0.05	4337	763	0.43	0.00 (2)	0.28 (2)	8.93	0.99	0.01	4294	763	45.05
v0416	19020	1771	0.13	11099	1001	0.91	0.00 (2)	0.66 (2)	27.01	67.29	31.19	3631	1001	87.92
v0417	143317	1765	1.60	55584	894	2.93	0.01 (2)	2.38 (2)	1089.24	0.00	0.00	55584	894	61.14
v0418	8306	1765	0.05	4827	953	0.62	0.00 (2)	0.41 (2)	7.56	1.33	0.02	4763	953	43.96
v0419	15709	1626	0.11	7744	845	1.27	0.01 (3)	0.97 (3)	19.21	62.62	19.72	2895	845	89.33
v0420	4099	958	0.02	2679	591	0.27	0.00 (2)	0.18 (2)	6.59	1.57	0.01	2637	591	36.49
v0421	1814	952	0.01	1176	464	0.14	0.00 (2)	0.08 (2)	3.05	0.43	0.00	1171	464	37.29
v1616	67441	1439	0.52	53073	1285	2.07	0.02 (2)	1.47 (2)	93.85	49.11	102.07	27011	1285	65.84
v1617	113655	1619	0.94	85759	1458	3.46	0.02 (2)	2.60 (2)	136.40	48.68	161.91	44009	1458	65.69
v1618	146715	1603	1.30	90998	1434	3.18	0.02 (2)	2.24 (2)	170.28	2.36	19.61	88852	1434	37.44
v1619	105822	1612	0.86	86032	1479	3.66	0.04 (2)	2.83 (2)	145.29	48.54	152.05	44274	1479	63.30
v1620	115729	1560	0.95	89624	1412	3.47	0.03 (2)	2.67 (2)	160.09	2.33	6.32	87536	1412	22.32
v1621	24772	938	0.17	16730	859	0.53	0.00 (2)	0.32 (2)	38.62	41.63	4.24	9765	859	66.02
v1622	13773	859	0.08	11123	787	0.41	0.00 (2)	0.25 (2)	25.88	31.49	1.85	7620	787	50.84
t0415	7254	1518	0.05	3198	894	4.35	3.80 (2)	0.37 (2)	10.00	0.06	0.02	3196	894	57.69
t0416	9345	1771	0.06	3171	992	4.37	3.81 (2)	0.41 (2)	8.18	0.09	0.01	3168	992	68.43
t0417	7894	1765	0.06	3572	926	5.31	4.74 (2)	0.38 (2)	11.13	0.14	0.02	3567	926	56.80
t0418	8676	1765	0.06	3931	1015	7.27	6.45 (2)	0.67 (3)	13.30	0.05	0.01	3929	1015	55.99
t0419	9362	1626	0.06	3176	912	2.32	1.86 (1)	0.34 (2)	9.60	0.22	0.01	3169	912	69.10
t0420	4583	958	0.03	1862	574	0.93	0.68 (1)	0.20 (3)	8.38	0.00	0.01	1862	574	62.76
t0421	4016	952	0.03	1669	570	1.41	1.22 (2)	0.13 (2)	7.09	0.06	0.00	1668	570	62.62
t1716	56865	467	0.69	11952	467	0.62	0.47 (1)	0.08 (1)	42.86	0.00	0.03	11952	467	75.47
t1717	73885	551	0.95	16428	551	0.72	0.49 (1)	0.14 (1)	54.38	0.00	0.05	16428	551	73.87
t1718	67796	523	0.89	16310	523	0.76	0.54 (1)	0.13 (1)	53.78	0.00	0.04	16310	523	72.47
t1719	72520	556	0.95	15846	556	0.79	0.53 (1)	0.16 (1)	51.15	0.00	0.05	15846	556	73.57
t1720	69134	538	0.86	16195	538	0.72	0.48 (1)	0.15 (1)	50.81	0.00	0.05	16195	538	72.89
t1721	36039	357	0.37	9043	357	0.22	0.12 (1)	0.05 (1)	41.74	0.00	0.02	9043	357	70.37

Table 3.8: Fast reduction w/o SUMC followed by one SUMC, Set 3

	Original		lex	Fast no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
v0415	7684	1518	0.05	4337	763	0.34	0.01 (1)	0.15 (1)	8.93	0.99	0.01	4294	763	45.05
v0416	19020	1771	0.13	11099	1001	0.66	0.00 (1)	0.35 (1)	27.01	67.29	31.17	3631	1001	87.92
v0417	143317	1765	1.60	55584	894	1.85	0.01 (1)	1.23 (1)	1089.24	0.00	0.00	55584	894	61.14
v0418	8306	1765	0.05	4827	953	0.50	0.00 (1)	0.22 (1)	7.56	1.33	0.01	4763	953	43.96
v0419	15709	1626	0.11	7747	847	0.60	0.01 (1)	0.35 (1)	19.16	62.60	19.83	2897	847	89.32
v0420	4099	958	0.02	2679	591	0.20	0.00 (1)	0.10 (1)	6.59	1.57	0.01	2637	591	36.49
v0421	1814	952	0.01	1176	464	0.12	0.00 (1)	0.05 (1)	3.05	0.43	0.00	1171	464	37.29
v1616	67441	1439	0.52	53073	1285	1.33	0.01 (1)	0.74 (1)	93.85	49.11	102.05	27011	1285	65.84
v1617	113655	1619	0.94	85759	1458	2.19	0.01 (1)	1.30 (1)	136.40	48.68	161.99	44009	1458	65.69
v1618	146715	1603	1.30	90998	1434	2.06	0.02 (1)	1.13 (1)	170.28	2.36	19.57	88852	1434	37.44
v1619	105822	1612	0.86	86032	1479	2.28	0.01 (1)	1.42 (1)	145.29	48.54	152.08	44274	1479	63.30
v1620	115729	1560	0.95	89624	1412	2.18	0.01 (1)	1.32 (1)	160.09	2.33	6.32	87536	1412	22.32
v1621	24772	938	0.17	16730	859	0.38	0.00 (1)	0.16 (1)	38.62	41.63	4.25	9765	859	66.02
v1622	13773	859	0.08	11123	787	0.30	0.00 (1)	0.12 (1)	25.88	31.49	1.86	7620	787	50.84
t0415	7254	1518	0.05	3199	894	1.26	0.87 (1)	0.19 (1)	10.00	0.06	0.01	3197	894	57.68
t0416	9345	1771	0.06	3186	993	1.29	0.90 (1)	0.22 (1)	8.24	0.09	0.01	3183	993	68.24
t0417	7894	1765	0.06	3653	926	1.52	1.06 (1)	0.21 (1)	11.54	0.14	0.02	3648	926	55.77
t0418	8676	1765	0.06	3937	1015	1.92	1.50 (1)	0.24 (1)	13.33	0.05	0.02	3935	1015	55.91
t0419	9362	1626	0.06	3176	912	1.30	0.90 (1)	0.18 (1)	9.60	0.22	0.01	3169	912	69.10
t0420	4583	958	0.03	1862	574	0.41	0.25 (1)	0.07 (1)	8.38	0.00	0.01	1862	574	62.76
t0421	4016	952	0.03	1669	570	0.34	0.19 (1)	0.07 (1)	7.09	0.06	0.00	1668	570	62.62
t1716	56865	467	0.69	11952	467	0.23	0.07 (1)	0.08 (1)	42.86	0.00	0.03	11952	467	75.47
t1717	73885	551	0.95	16428	551	0.32	0.07 (1)	0.14 (1)	54.38	0.00	0.05	16428	551	73.87
t1718	67796	523	0.89	16310	523	0.30	0.06 (1)	0.12 (1)	53.78	0.00	0.05	16310	523	72.47
t1719	72520	556	0.95	15846	556	0.32	0.04 (1)	0.16 (1)	51.15	0.00	0.05	15846	556	73.57
t1720	69134	538	0.86	16195	538	0.29	0.05 (1)	0.14 (1)	50.81	0.00	0.05	16195	538	72.89
t1721	36039	357	0.37	9043	357	0.13	0.03 (1)	0.05 (1)	41.74	0.00	0.02	9043	357	70.37

Table 3.9: Maximal reduction w/o SUMC followed by one SUMC, Sets 2, 4

	Original		lex	Maximal no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
0321.4	71201	1202	0.77	36181	1201	148.27	144.91 (4)	2.48 (2)	187.16	0.01	1.27	36179	1201	35.89
0331.3	45637	664	0.44	22125	664	12.96	12.39 (1)	0.43 (1)	189.87	0.03	0.72	22118	664	37.07
0331.4	46915	664	0.47	21626	664	9.02	8.58 (1)	0.31 (1)	162.32	0.10	0.70	21605	664	41.28
0341.3	45800	658	0.43	21163	656	14.37	13.25 (1)	0.82 (2)	156.22	0.01	0.57	21161	656	38.62
0341.4	46508	658	0.48	20315	655	8.45	7.58 (1)	0.62 (2)	139.61	0.01	0.49	20312	655	41.82
0351.3	64953	1156	0.72	34446	1156	42.55	40.86 (1)	1.40 (1)	196.38	0.01	1.68	34442	1156	33.50
0351.4	69922	1156	0.75	33779	1147	34.65	31.40 (1)	2.39 (2)	183.16	0.02	2.57	33771	1147	38.77
nf260	276752	2198	4.02	23751	1349	26.37	1.81 (2)	23.62 1(8)	34.32	0.00	0.24	23751	1349	94.00
sp1	6954	204	0.05	6807	198	6.66	6.54 (2)	0.07 (3)	219.39	0.00	0.04	6807	198	4.61
sp2	3686	173	0.02	3529	150	2.07	2.00 (2)	0.03 (3)	139.13	0.00	0.02	3529	150	16.38
sp3	1668	111	0.01	969	73	1.06	1.03 (4)	0.00 (4)	79.55	0.00	0.00	969	73	61.41
sp4	9144	368	0.08	8976	308	12.54	12.11 (2)	0.34 (4)	172.21	0.00	0.04	8976	308	16.74
sp5	13718	684	0.11	13045	628	30.53	28.81 (2)	1.46 (4)	104.91	0.49	0.24	12981	628	12.24
sp6	50722	2504	0.50	41061	2200	483.07	453.15 (4)	27.34 (7)	90.61	0.00	0.61	41061	2200	25.18
sp7	43459	2991	0.44	36507	2466	454.89	375.05 (3)	76.57 1(8)	71.90	0.00	0.27	36507	2466	26.93
sp8	91123	4810	0.91	72361	3810	783.98	670.68 (3)	104.38 (9)	85.24	0.00	0.68	72359	3810	30.85
sp9	50013	2917	0.45	28992	1832	211.91	192.57 (4)	17.68 (9)	77.44	0.00	0.18	28992	1832	61.97
sp10	13128	781	0.09	4452	369	9.86	8.88 (6)	0.81 (8)	53.56	0.00	0.06	4452	369	81.24
sp11	2775	104	0.01	528	64	1.47	1.45 (6)	0.00 (6)	60.12	0.00	0.00	528	64	88.14
sp12	84746	3218	0.94	74467	2811	627.61	547.73 (2)	76.07 (9)	112.10	0.00	2.31	74467	2811	20.43
sp14	47214	3217	0.64	42828	2743	370.06	321.93 (2)	45.21 (9)	64.59	0.00	0.29	42828	2743	20.20

Table 3.10: Fast reduction w/o SUMC followed by one SUMC, Sets 2, 4

	Original		lex	Fast no SUMC			Expensive redn fns		SUMC reduction			Final size		%nzs
name	cols	rows	time	cols	rows	time	CLEXT	DOMR	av bl	%	time	cols	rows	deld
0321.4	71201	1202	0.77	36184	1201	17.69	15.69 (1)	1.24 (1)	187.17	0.01	1.25	36182	1201	35.89
0331.3	45637	664	0.44	22125	664	4.92	4.34 (1)	0.42 (1)	189.87	0.03	0.71	22118	664	37.07
0331.4	46915	664	0.47	21626	664	3.27	2.83 (1)	0.30 (1)	162.32	0.10	0.69	21605	664	41.28
0341.3	45800	658	0.43	21163	656	5.80	5.10 (1)	0.41 (1)	156.22	0.01	0.56	21161	656	38.62
0341.4	46508	658	0.48	20315	655	3.15	2.58 (1)	0.31 (1)	139.61	0.01	0.48	20312	655	41.82
0351.3	64953	1156	0.72	34446	1156	18.22	16.45 (1)	1.40 (1)	196.38	0.01	1.67	34442	1156	33.50
0351.4	69922	1156	0.75	33779	1147	14.31	12.29 (1)	1.18 (1)	183.16	0.02	2.60	33771	1147	38.77
nf260	276752	2198	4.02	42652	1984	12.41	8.74 (1)	2.64 (1)	49.10	0.00	0.39	42652	1984	85.32
sp1	6954	204	0.05	6867	198	1.02	0.93 (1)	0.02 (1)	221.39	0.00	0.05	6867	198	3.89
sp2	3686	173	0.02	3576	151	0.36	0.31 (1)	0.01 (1)	141.13	0.00	0.02	3576	151	14.37
sp3	1668	111	0.01	1159	78	0.12	0.10 (2)	0.01 (3)	81.08	0.00	0.00	1159	78	51.80
sp4	9144	368	0.08	9020	309	1.73	1.50 (1)	0.08 (1)	173.06	0.00	0.04	9020	309	15.98
sp5	13718	684	0.11	13211	629	5.88	5.27 (1)	0.39 (1)	106.20	0.48	0.29	13148	629	10.46
sp6	50722	2504	0.50	41514	2212	57.92	46.48 (1)	8.82 (2)	91.81	0.00	1.02	41512	2212	24.10
sp7	43459	2991	0.44	37700	2535	64.48	52.25 (1)	9.89 (2)	73.61	0.00	0.28	37700	2535	22.40
sp8	91123	4810	0.91	72683	3846	134.26	98.90 (1)	25.80 (2)	85.31	0.00	0.67	72681	3846	29.93
sp9	50013	2917	0.45	29845	1864	33.00	23.06 (1)	7.59 (3)	79.01	0.00	0.20	29845	1864	60.01
sp10	13128	781	0.09	4644	382	1.21	0.53 (1)	0.48 (3)	53.08	0.00	0.06	4644	382	80.24
sp11	2775	104	0.01	760	72	0.38	0.35 (3)	0.00 (3)	78.00	0.00	0.01	760	72	80.58
sp12	84746	3218	0.94	74745	2831	155.25	132.98 (1)	18.82 (2)	112.51	0.00	2.09	74745	2831	19.46
sp14	47214	3217	0.64	43056	2764	73.23	59.14 (1)	10.92 (2)	64.92	0.00	0.29	43056	2764	19.19

Chapter 4

Feasible solution heuristics

A high quality (integral) feasible solution is essential to keep the size of the search tree manageable in a branch-and-bound framework. A method for finding such a feasible solution can be used both before branch-and-bound is started and later on at a search tree node. Thus it is desirable to find a very efficient procedure that can be applied many times without slowing down the exact solution method.

Finding good feasible solutions for Set Partitioning Problems is notoriously difficult since the problem is usually very tightly constrained, although much depends on the generators used to formulate the problems. In practice very expensive dummy columns are often included in the matrix to provide a starting feasible solution, but this solution is usually not sufficient for bounding.

In what follows we will discuss LP relaxation based feasible solution heuristics, first in general, then our application in detail. We will also show how to iterate these heuristics and *reduced cost fixing* to improve the quality of the feasible solution.

Finally, we indicate the difficulties associated with efficiently (re)solving LPs during the algorithm and conclude with computational results. A detailed description of the parameters used during our feasible solution heuristic can be found in Appendix C.

4.1 LP relaxation based heuristics

LP relaxation based feasible solution heuristics refers to methods where variables are heuristically set to their upper or lower bounds based on the most recent solution to the LP relaxation, the LP is re-solved and the process is repeated until either a feasible solution is found or the problem becomes infeasible. These methods can be thought of as a quick way to get to a leaf of the complete enumeration tree. Only one thread from the root to a leaf is investigated (no backtracking) and several levels are "skipped over" when multiple variables are set to their bounds.

Both the quality of the feasible solution delivered at the end (if any) and the running time depend on the heuristic setting of variables. The running time is usually dominated by the LP solver. Having fewer iterations means fewer LPs to solve, but it usually also means setting more variables to their bounds at a time, possibly compromising quality. On the other hand, setting fewer variables to their bounds usually implies more iterations but does not necessarily increase the running time since consecutive LP formulations might be "close enough" to use the optimal solution of the first one to *warmstart* the second. Section 4.3 describes warmstarting LPs in detail.

The popularity of these methods for the Set Partitioning Problem lies in the fact that because of the highly constrained nature of the problem many rows and columns can be eliminated from the formulation as a result of setting variables to their bounds, especially to one. However, fixing variables to one might be "dangerous" since essential columns could be eliminated causing the problem to become infeasible. Problem size reduction methods, like those described in Chapter 3 are utilized to propagate the effects of setting variables.

After a feasible solution has been obtained for the Set Partitioning Problem (or in general, for any $0-1$ integer programming problem), variables currently at value 0 in the LP solution whose reduced cost is larger than the *gap* (the difference between the objective value of the feasible solution and the lower bound provided by the LP optimum) can be removed from the problem. A procedure that scans through all the nonbasic variables and removes those with reduced costs larger than the gap is called *reduced cost fixing.* A subsequent application of problem size reduction methods can further decrease the size of the matrix. Note that the reduced problem has the same optimal value as the original one, thus it can replace the original problem in the subtree rooted at the search tree node where the feasible solution heuristic is invoked.

The quality of the feasible solution can be improved by combining the feasible solution heuristic with the reduced cost fixing procedure in a loop that repeats until there is nothing left in the matrix (the best feasible solution found so far is optimal) or either the heuristic is not able to find a better feasible solution or the reduced cost fixing cannot eliminate more variables. Since we are looking for feasible solutions that strictly improve the best found so far, the gap in the reduced cost fixing procedure could be decreased by the *granularity* (the minimal difference between non-identical feasible solution values) of the problem. Note that computing the granularity is a nontrivial task. However, a lower bound on the granularity can just as well be used to decrease the gap (for instance, 1 is such a lower bound for an SPP if all the objective function coefficients are integral).

The first detailed description of an LP based feasible solution heuristic (to our knowledge) is due to Hoffman and Padberg ([HP93]). Their approach is based on the assertion that small set partitioning problems are easy to solve. After applying their problem size reduction routines an outer loop is entered where solving the LP relaxation, fixing variables at level one in the LP optimal solution to one and setting some further variables in an inner loop are repeated until either a feasible solution is found or the problem becomes infeasible due to an "incorrect guess". In the inner loop the problem is first decomposed into smaller blocks by setting some more variables to their bounds, then in each block variables with values closest to one are set to one (and the effects of these settings are propagated by the problem size reduction routines) until "enough" variables are fixed in the block. After a feasible solution is found, variables are eliminated based on their reduced costs. Hoffman and Padberg's approach failed (did not deliver a feasible solution) on five of the 40 Set 1 problems that do not solve to integrality with the first LP relaxation. They report only cumulative running times in a Branch-and-Cut setting (feasible solution heuristic is run at several nodes of the search tree).

Borndörfer et al. ([BGKK97] and[Bor97]) implemented a similar "plunging method" (as they refer to it) that repeats solving the LP relaxation, rounding fractional values to the nearest integer and applying problem size reduction methods. Their approach includes a pivoting technique that allows for an efficient LP warm-start. Although applied both to the Set 1 and Set 3 problems, the quality of the feasible solutions produced by this approach is not reported and only cumulative running times are provided as for the previous approach.

An approach that explores more than just one thread of the complete enumeration tree was developed by Ladányi and Ralphs ([LR]). After the initial problem size reduction a search tree is built where at each node the LP relaxation is solved, variables at level one in the LP optimal solution are fixed to one, and then subproblems are created by selecting a few variables (usually those at nonzero levels in a row's support) asserting that exactly one of them is at level one in a feasible solution and setting each of these variables in turn to one. The search tree is investigated in a depth first search manner. This approach produced a feasible solution to all the Set 1 problems and found (but not proved) optimal solution for many. The running times were comparable to the overall branch-and-cut running times reported by Hoffman and Padberg.

Our approach is novel in the sense that it prefers fixing variables to zero instead of to one and that it repeats the feasible solution heuristic and reduced cost fixing modules in a loop. We not only found high quality feasible solutions to the Set 1 problems but also *proved* the optimality of the feasible solutions obtained for 30 of the 40 problems and found the optimal solution but did not prove their optimality for five more problems. Section 4.4 contains our computational results.

4.2 Our algorithm

In this section we will discuss our feasible solution heuristic implementation, including its integration into a loop with reduced cost fixing.

4.2.1 Heuristic variable fixing

The heart of our algorithm is the heuristic fixing phase; that is, a collection of heuristics that fix variables to zero or one based on the results of the most recent LP relaxation. The heuristic fixing phase has three major components. First, variables currently at level one are addressed. Then pairs of rows that are likely to be covered by the same column in an optimal solution are identified (this is called *follow-on fixing*). Third, some “unattractive” variables are eliminated from the problem. All the subroutines used here only mark variables to be fixed to one or zero, the actual eliminations (and propagation of their effect) are carried out by Reduce().

Variables at level one

Probably the most difficult question for an LP based feasible solution heuristic is what to do with variables at level one in the LP relaxation. Depending on how the problem was generated, the number of variables at level one compared to all those at nonzero levels might be high or small (see Section A.2). Certainly the most popular approach is to fix these variables to one (and even to round up values that are near to one). Although this would be justified for a special case of (SP), the *node packing problem* (Section 1.3), since in that case there exists an optimal solution in which these variables take value one ([NT75]), this is not valid for the set packing or set partitioning problems in general. However, fixing variables to one might cause infeasibility, as we can see in our computational results (Section 4.4). We have implemented this approach along with three others; the particular one used is controlled by a parameter.

The second approach is slightly less aggressive than fixing these variables to one. When a variable is fixed to one we remove all columns that intersect any row in the variable's support. On the other hand in the less aggressive approach we retain those columns that are common in all rows in the variable's support; that is, we delete the symmetric difference of row supports, for rows intersecting the variable's column. This is really an adaptation of the follow-on fixing idea that we will discuss below.

The third approach is to treat variables at level one the same way as other variables at nonzero level, that is, do nothing with them at this point. The fourth approach is an adaptive strategy, a mix of the three that decides which one is used based on the proportion of variables at level one compared to all the variables at nonzero levels.

Follow-on fixing

The original idea of follow-on fixing is folklore in crew scheduling, an early reference to it in a branch-and-bound setting can be found in [RF87]. We are given the schedule, say, of an airline for a given time interval. The schedule contains the departure/arrival times and stations for each flight segment, along with the specific type of aircraft to fly that segment. Our goal is to assign crews to the flight segments as cheaply as possible, complying with all rules and regulations. Modeled as a set partitioning problem, the rows of the matrix correspond to flight segments while the columns describe possible crew trips (Section 1.1). After the LP relaxation is solved, primal values of the columns in a row's support can be interpreted as likelihoods with which the corresponding crew trips cover the flight segment. Since it is preferable to keep a crew with the same aircraft during a workday, connecting flight segments (same aircraft) are considered *follow-ons* if they are likely to be covered by the same crew trips. Follow-on flight segments are *locked* (considered as one segment from now on) and all crew trips covering only one of them are removed from the problem.

Although in a general set partitioning problem usually no ordering can be imposed on the rows of the matrix, the follow-on idea can still be exploited. For any pair of rows we can determine the likelihood that they are covered by the same column by accumulating the primal solution values for all columns intersecting both rows. If this likelihood is large enough (larger than a certain threshold), all the columns that cover only one of the rows are removed from the matrix, causing the two rows to become identical. Notice that if a variable is at level one, the rows in its support are follow-ons.

In our implementation the follow-on fixing procedure is optional, it can be enabled/disabled by a parameter. Comparing all row pairs in a matrix might be too costly, so we select a subset of the rows (the selection is either random or it is based on dual values), and compare all row pairs in this subset. The fraction of rows chosen for this subset and upper and lower bounds for the threshold are also controlled through parameters. To keep the implementation fast and simple, follow-on fixing is carried out in several passes of comparing rows and applying Reduce(). Rows determined to be follow-ons in one pass are not compared with other rows before the matrix is reduced. The threshold is set to the upper bound when the follow-on fixing procedure is entered, then it is gradually decreased down to the lower bound if the number of columns eliminated is not sufficient.

Removing unattractive variables

A cautious approach of fixing those variables to zero that are insignificant based on the most recent LP relaxation is much less likely to eliminate optimal solutions than fixing variables to one. In our implementation the significance of the variables is determined by their reduced costs (the lower the reduced cost the more significant the variable), but other measures, like the ratio of the original objective function coefficient to the number of nonzeros in the column, could be used.

To make sure that the problem does not become infeasible due to an empty row, we control the deletion process as follows. The rows of the matrix are enumerated one by one, (the order is either random, or it is based on dual values, as determined by a parameter), and some of the least attractive variables are marked for removal. The procedure terminates if a certain fraction of the rows has been already considered, or enough variables are marked for removal.

There are many ways to decide which variables to delete from a certain row. We have experimented with several ideas; two of these proved to be acceptable. The first approach simply marks for removal a given fraction of the least significant variables that are at level zero in the current LP relaxation. Note that basic variables at zero level may be marked with this procedure, although they can be optionally unmarked later on.

In the second approach the variables in a row's support are considered from most significant to least significant, until their sum in the current LP solution surpasses some predetermined value (threshold). When this happens, a given fraction of the rest of the variables (starting with the least significant) are marked for removal. If the threshold is less than one then variables at nonzero levels may be marked for deletion, making the heuristic more aggressive. However, deletion of nonzero variables can be disabled through a parameter. On the other hand, if the threshold is one this approach will yield very similar results as the previous approach.

4.2.2 Unmarking variables

As we have seen in the previous section, variables to be deleted are marked only during the heuristic fixing phase; their actual elimination is carried out by Reduce() afterwards. We provide an opportunity to unmark some "important" columns before Reduce() is invoked. Note that this will not prohibit their removal by Reduce() if implied by some reductions.

Columns with only a few nonzeros are very often essential for integral feasibility. If the corresponding variable is of high cost then this variable will be unattractive in at least the first few LP relaxations and it is likely to be marked during the heuristic fixing phase. (Including expensive short columns to ensure integral feasibility is typical for a number of problem instance generators.) A parameter determines up to how many nonzeros a column can contain in order to be spared during the heuristic fixing phase. If the parameter is set to zero then no column will be unmarked. Protecting columns with one or two nonzeros is profitable, while values four and above are impractical since very often in this case no columns are left marked for deletion when the matrix is already small. To avoid a situation where all marked variables are unmarked by this procedure the parameter is temporarily decreased (but not below one) until some variables are left marked.

We may also unmark variables which are basic in the current LP optimal solution. A parameter enables this option. A different parameter can enable/disable the deletion of variables at nonzero level (which are always basic). This option can be used when we wish to remove all nonbasic variables from a row's support. Protecting basic variables also provides a smoother and faster LP warmstart.

4.2.3 Overview of our algorithm

In this section we summarize the general flow of our algorithm, see Figure 4.1 for an outline.

First our problem size reduction algorithm Reduce(), is invoked. If Reduce() returns with a feasible (hence optimal) solution or it has determined that the problem is infeasible, we are done. The initial problem size reduction is followed by solving the LP relaxation (variables are nonnegative instead of binary) of the reduced problem. If the LP relaxation has an integral optimal solution or it is infeasible then we are also done.

```
Feasible Solution Heuristic
  Input:   A, c, parameters
  Output:  feasibility status, A', c', ONES, MERGES

  invoke Reduce(); if integral opt soln found or infeas, return
  solve LP relaxation; if integral opt soln found or infeas, return

  while (iteration limit not reached) {
    copy current problem into tmp problem
    while (feasibility of tmp problem is not known) {
      invoke heuristics to fix variables in tmp problem
           if no vars fixed: heur failed (weak par setting), return
      use Reduce() to propagate effects of fixing
           if integral feas soln found or infeas, break
      solve LP relaxation of reduced tmp problem
           if integral feas soln found or infeas, break
      if original problem is feas invoke reduced cost fixing
          on tmp problem
    }

    if integral feas soln found above, compare w/ best found so far
           if better, update; if worse, return.
    if infeas detected above, heur failed (aggressive par setting),
           return
    otherwise we have a (better) feasible solution

    do {
      invoke reduced cost fixing on current problem
         if no vars fixed
            if first time since (better) feas soln found, return
            ow break out from this loop
      use Reduce() to propagate effects of fixing
           if integral feas soln found or infeas, optimal, return
      solve LP relaxation of reduced current problem
           if integral feas soln found or infeas, optimal, return
    } while (enough variables have been eliminated)
  }
end
```

Figure 4.1: Outline of Feasible Solution Heuristic

Next a major loop is entered where first we attempt to find a feasible solution to the *current problem* (the most recent reduced formulation that has the same optimal solution as the original problem) using a variety of heuristics, then, if we succeed, eliminate variables based on their reduced costs in the current LP relaxation (reduced cost fixing). This major loop is repeated until: (*i*) the heuristics in the first part of the major loop fails to provide a (better) feasible solution, or (*ii*) an integral solution to the current problem (thus optimal) is found as a result of the most recent reduced cost fixing, or (*iii*) a pre-set iteration limit is reached. Before the algorithm terminates, the best feasible solution found (if any) is displayed and the remainder of the problem is saved. Note that it is possible that the best feasible solution found is in fact optimal but the remainder problem is nonempty (and it might even be infeasible); this simply means that not all variables could be eliminated based on their reduced costs.

Note that at the top of the major loop we have a formulation that has the same optimal value as the original problem. The search for a feasible solution to the current problem starts with creating a temporary copy to which the heuristic will be applied. The temporary problem is used since the heuristic fixing phase might eliminate some columns and/or rows leading to the loss of all optimal solutions. Then a loop that repeats heuristic fixing of variables (see details below), propagating the effects of fixing using Reduce(), and re-solving the LP relaxation (if necessary), is entered. If a feasible solution to the original problem is known, variables are also eliminated based on their reduced costs after the LP relaxation is re-solved. This loop repeats until the heuristic is not able to fix any variables, a feasible solution to the temporary problem is found, or the temporary problem becomes infeasible.

The execution of the heuristic is determined by a set of parameters. If the heuristic was not able to fix any variables then the parameter settings were too weak. On the other hand, if the temporary problem became infeasible, the heuristic was invoked with the parameters set too aggressively. Our algorithm terminates in both cases, although the inner loop could be restarted after adjusting the parameters.

If the algorithm did not terminate until this point, a (better) feasible solution has been found. In the second part of the major loop variables in the current problem are eliminated repeatedly based on their reduced costs, until no more such reduction is possible (in practice, until no more than a certain percentage of variables is deleted). Reduce() is invoked to propagate the effects of this fixing and then the LP relaxation of the reduced (current) problem is re-solved. If either Reduce() or the LP solver finds the problem (integral) feasible then the best feasible solution found so far is optimal since all the reductions to the current problem preserve the optimal value. Also, if the current problem is found to be infeasible by Reduce() or by the LP solver, the best feasible solution found so far is again optimal. This is because we aim for a strictly better feasible solution during reduced cost fixing, as described earlier. If no reduction was possible the first time reduced cost fixing is invoked after a (better) feasible solution has been found then the algorithm terminates.

If only the first feasible solution is desired then the iteration limit can be set to 1. In our experience the major loop repeats only a few times for an average instance, and every subsequent pass is considerably faster than the previous one since reduced cost fixing usually eliminates many columns. Most of the additional time after the

the first feasible solution is found is spent in resolving the LP relaxations after reduced cost fixing.

4.3 Solving the LP relaxations (warmstart)

Our experiments were carried out using CPLEX v4.0.9 ([CPX95]). When interpreting observations related to solving LPs (especially to warmstarting) we have to keep in mind that using other LP solvers might result in significantly different conclusions.

In our implementation we have experimented with the primal simplex, dual simplex and barrier (with and without crossover) methods. Since set partitioning problems are highly degenerate in general, the primal simplex method performs very poorly on them. While the barrier method was slightly slower on the smaller problems than the dual simplex method, it is superior to dual simplex on medium and large problems. We have used the barrier method with dual crossover in our final experiments. (Reportedly, the dual simplex method has been improved significantly in CPLEX v6.0 and its performance is comparable to that of the barrier method on not only the smaller problems, [Bix98].)

As we have mentioned earlier, information from the previous LP can be used to warmstart the next LP (instead of solving it from scratch) if the formulation changed only a little. It might also be possible to construct an optimal solution without re-solving the LP if certain important columns and rows were not removed from the matrix. In CPLEX v4.0.9, the simplex methods can be warmstarted only using basis information, while barrier methods can use basis information and/or a primal-dual solution pair (primal-dual worked the best). In our implementation a parameter determines what warmstart information to use; there is no warmstart if this parameter is set to zero.

The effectiveness of the warmstart depends heavily on the implementation of the specific warmstart algorithm in the LP solver. Since we have no access to this information we made an effort to provide a high quality (near optimal) starting basis and primal-dual solution pair to the LP solver. We keep a copy of the basis information and primal-dual solution pair for both the current and temporary problems. Every time the corresponding LP relaxation is re-solved the optimal basis information and primal-dual solution pair are copied; and whenever Reduce() is invoked we tailor these copies to the new formulation. In what follows we discuss these updates in detail.

In CPLEX the basis information consists of the basis status of all the structural and artificial variables (these latter are added by CPLEX). To provide basis information for the warmstart we have to designate which variables are basic and which are nonbasic (and at which bound). Ideally, the columns corresponding to the variables designated to be basic should be linearly independent and the number of these variables should be exactly the number of rows. However, CPLEX is able to start from any basis information, it will eliminate dependencies from among the columns designated to be basic and extends the remaining set of independent columns into a valid basis. In our implementation we have included an option that lets us re-solve the LP from scratch instead of warmstarting if we think the basis information we can provide is far from a valid basis.

We update the basis information from the previous LP by keeping the basis status of all variables (structural and artificial) that were not deleted or merged and setting the basis status of merged columns to be basic if the column was merged from formerly basic columns, and nonbasic at lower bound otherwise. Note that if none of the following three events occur then this new basis information provides a valid basis (which also turns out to be optimal as we will see below). First, a basic structural variable is deleted, leaving too few independent columns (variables marked as basic) in the basis. Second, a basic structural variable is merged with some nonbasic variables; in this case the resulting column might not be independent from the rest of the columns in the former basis so we designate this merged variable nonbasic in the new basis information which again leaves too few independent columns. The last but probably most important case is when a row with a nonbasic artificial variable is deleted from the formulation since this might leave us with linearly dependent columns in the basis. Others (e.g., [BGKK97]) solve these problems by not removing basic structural variables, disallowing merging and also deleting only those rows whose artificial variables can be pivoted into the basis with a degenerate pivot.

The primal and dual solutions consist of values for both the structural and artificial variables. Primal solution values for structural variables are updated by averaging the values of variables that make up a column (columns could be merged columns); while dual solution values are approximated by adding the corresponding values. Primal solution values for artificial variables are computed explicitly, while dual values remain unchanged. If none of the three cases above occurs then the primal-dual pair created with these rules will be primal/dual feasible and the reduced costs of variables remain nonnegative if that was the case before the update; that is, optimality is retained. In this case it is not even necessary to re-solve the LP.

4.4 Computational results

After a considerable amount of experimenting we decided on the following parameter settings (we refer to them as default settings) for our final runs. We used the barrier method with dual crossover to solve the LP relaxations, with a primal-dual pair for warmstarting. A lower bound on the granularity was set to .99 for the Set 1, 3 and 4 problems and .0099 for the Set 2 problems. We did not limit the number of major iterations. In the heuristic fixing phase LP reoptimization is forced after 5% of the columns are eliminated. The adaptive strategy was used for variables at level one in the current LP relaxation (such variables were set to one if the ratio of the variables at level one to all the variables at nonzero level was at least .75, and they were skipped otherwise). Follow-on fixing was enabled, with the threshold starting at .99 but not going below .59. At least 50% of the row pairs (and all for the smaller problems) were examined during follow-on fixing, comparing those with the largest absolute dual values first. Unattractive variables were removed only if the follow-on fixing did not mark any variables for deletion. Up to 25% of the rows were selected randomly for this procedure, marking 10% of the least attractive variables currently at zero level for deletion. Columns with up to two nonzeros as well as basic variables were protected. The whole algorithm was given a time limit of 7200 seconds.

Tables 4.1, 4.2, 4.3 and 4.4 summarize the results of this run for all four problem sets. These tables contain the name, the optimal value of the first LP relaxation, the optimal value for problems in Set1 or the best feasible solution known for problems in Set 3 (with a "*" if the solution is known to be optimal); the upper bound obtained by our feasible solution heuristic (a "*" if optimality was proved and "F" if our heuristic failed), the optimality gap ($(\bar{z}-z^*)/z^*$) for problems in Set 1 and the integrality gap ($(\bar{z}-\underline{z})/\underline{z}$) for the rest of the problems (in [BGKK97] this latter is computed as $(\bar{z}-\underline{z})/\bar{z}$ for the Set 3 problems); the number of columns and rows left in the matrix; the number of times the major loop and the heuristic fixing phase (we refer to it as the minor loop) are repeated; the total time spent in the heuristic (less the time needed to read in the problem instance and carry out the initial Reduce()); the time spent in solving LPs and in Reduce() (with their multiplicity); and finally the first feasible solution found by the heuristic and the time spent until then.

The LP and Reduce() times make up almost all the time spent in the entire procedure, so the heuristic and reduced cost fixing algorithms themselves are very inexpensive. Usually the quality of the first feasible solution found is acceptable, however, the strength of our algorithm is that with the repeated use of reduced cost fixing the optimal solution is not only found but also proved for most of the "easy" problems. The time spent in further major iterations is not more than 10% of the total time, it is mostly used for (re)solving LP relaxations after reduced cost fixing.

A feasible solution has been found to all the Set 1 problems, optimality has been proved for 30 of the 40 problems, the optimal solution was found but not proved for 5 more problems (the remaining matrix is quite small for these problems), and the optimality gap is below 2% for the other 5 problems. The remaining matrices are very small for all but the 3 difficult problems (`a01`, `aa04` and `nw04`). The small problems are very often solved to optimality in one major iteration.

We have found a feasible solution within 2% of the first LP optimum for all the `v*` problems. The `t*` problems seem to be more difficult; a feasible solution (within $56 - 62\%$ of the LP optimum) was found for half of the `t17` problems, and no parameter setting resulted in a feasible solution for any of the `t04` problems. The ratio of variables at level one to all variables at nonzero levels is very high for the `v*` problems. The adaptive strategy takes advantage of this: feasible solutions are found very quickly for the `v04` problems (unfortunately all but one of the `v16` problems are missed because of their slightly smaller ratio). It is interesting to observe that reduced cost fixing was not able to eliminate any columns after a feasible solution was found. This is an indication that these problems are well-constructed.

The problems in Set 2 are very hard, our feasible solution heuristic always failed on these problems. The LPs are moderately difficult to solve, the global time limit was never reached. Our results for the Set 4 problems are mixed, for some of them the LPs are just very hard to solve (we ran out of time while solving LPs in `sp6`, `sp7`, `sp8`, `sp9`, `sp12` and `sp14`). For the rest of the problems the heuristic delivered an optimal solution in three cases, found a feasible solution for two more problems and failed for three. The problems for which the optimal solution was found seem to be easy since only one major iteration was needed.

We also include the results of some comparison runs for the Set 1 and 3 problems. Looking at the results of these runs confirms our choice of parameters. Although not reported here, similar experiments were also performed on the problems in the other two sets as well during preliminary testing (the results were the same as for the default, consistently failing for all problems that the default did).

Tables 4.5, 4.6, 4.7, and 4.8 contain results of these experiments for the Set 1 and Tables 4.9 and 4.10 for the Set 3 problems. The tables give the upper bound obtained by our feasible solution heuristic, the total time spent in the heuristic; the time spent in solving LPs (with multiplicity) for the first experiment and the number of major and minor iterations for the other experiments. The following describes these experiments and gives a few words of explanation for each.

- *Dual simplex (with basis warmstart) instead of barrier.* The barrier method is clearly much faster for all but the smallest problems. Note that neither of the LP solvers results in consistently better feasible solutions since the only difference between the two experiments is that the LP solvers may end up in different extreme points for the same LP relaxation, pushing the heuristic into different directions.

- *Follow-on fixing turned off.* This is clearly much worse than the default; it fails more often and takes more time and minor iterations. This shows that the follow-on approach is more robust than the removal of unattractive variables.

- *Follow-on threshold starts at* .79 *instead of* .99*.* This is a more aggressive setting, since more row pairs will be found to be follow-ons at first. This setting fails more often and the quality of feasible solutions found is slightly worse than the default setting. There is a slight speed-up for some of the problems that is due to having fewer major/minor iterations.

- *Rows with smallest (instead of* largest*) dual absolute values are compared first during follow-on fixing.* This approach fails for a few `v*` problems and delivers worse feasible solutions for some Set 1 problems (even if all row pairs are considered for the smaller problems the results can be different since the row pairs are enumerated in a different order).

- *Variables at level 1 in the LP relaxation are fixed to 1.* This is clearly a more aggressive approach than our default. As we expected, it fails more often or gives lower quality feasible solutions but it can be much faster. It is particularly good for the `v16` problems that the default approach missed.

- *Symmetric difference is applied to variables at level 1.* Slightly less aggressive than fixing the same variables to one, we can observe that the feasible solutions are slightly better but the running time is worse. Since we use follow-on fixing that achieves the same effect (but in several iterations), we decided to go with a combination of aggressive fixing and "doing nothing" in our default.

- *LP re-optimization is forced only if* 20% *of the columns are marked for deletion.* This approach is more aggressive than our default. While faster than the default, it fails or delivers worse feasible solutions very often. This shows that frequent re-optimization of the LP is important; our updates become weak if the matrix changes significantly.

- *Short columns are not protected.* This approach delivers slightly worse feasible solutions for the Set 1 problems and fails for the `v*` problems too often. This shows how important short columns can be for certain classes of problems. The running times are mixed.

We can conclude that our feasible solution heuristic with the default settings is a robust algorithm that delivers high quality feasible (often optimal) solutions for all the Set 1 and many of the Set 3 problems. The two main factors contributing to the running time are the LP solver and the problem size reduction. We see some possibility for improvement for both. More information on the LP solver's warmstart algorithm could lead to a better strategy for when and how to resolve LPs. The problem size reduction could be improved by implementing additional reduction methods. This might slow down the individual Reduce() calls but would most likely decrease the number of times Reduce() (and the LP solver) is invoked, so possibly reducing the overall running time. Saving time by omitting CLEXT, the most expensive part of Reduce() is a double-edged sword; the execution time usually decreased, but the solution quality significantly declined as well (it even failed on all but two of the `v*` problems, as well as on four of the Set 1 problems).

The quality of our feasible solution heuristic compares very favorably with Hoffman and Padberg's algorithm ([HP93]), see Appendix A.3 for earlier results. Even our first feasible solutions frequently dominate their result. We cannot directly compare the running times since they do not report separate execution times for their initial heuristic runs. Borndörfer ([Bor97]) does not report the value of the feasible solution found by his heuristic, only the integrality gap and the size of the matrix remaining after an additional application of his problem size reduction routine (if a feasible solution has been found). The quality of our results is better (the gap and the matrices remaining after our heuristic are smaller), but our execution times are longer. Borndörfer et al. ([BGKK97]) do not report the results of their initial feasible solution heuristic for the Set 3 problems.

Table 4.1: Feasible solution heuristic (default setting), Set 1, part 1

			Feasible solution heuristic							LP		Reduce		First feas sol	
name	lp opt	int opt	ub	o gap	cols	rows	M	m	time	time	freq	time	freq	ub	time
aa01	55535.44	56137	56172	0.06%	4764	549	2	57	39.88	18.52	71	20.59	121	56172	23.23
aa03	49616.36	49649	49649	0.00%	178	108	3	55	20.58	10.79	65	9.32	116	49713	19.66
aa04	25877.61	26374	26680	1.16%	5703	340	2	82	23.45	12.04	91	10.63	179	26680	13.89
aa05	53735.93	53839	53904	0.12%	956	298	4	151	22.64	10.11	190	11.47	323	53943	15.63
aa06	26977.19	27040	27040	0.00%	581	255	2	50	14.85	8.01	58	6.34	103	27040	13.82
kl01	1084.00	1086	*				1	6	1.72	1.43	9	0.23	16	1086	1.70
kl02	215.25	219	219	0.00%	1562	50	3	90	7.29	5.38	95	1.56	158	220	6.44
nw03	24447.00	24492	*				2	25	11.74	10.35	31	1.19	70	25125	11.54
nw04	16310.67	16862	17158	1.76%	5782	35	2	19	19.11	15.48	24	3.36	35	17158	16.59
nw06	7640.00	7810	*				3	16	1.52	1.24	22	0.23	47	9430	1.23
nw11	116254.50	116256	*				1	4	0.30	0.27	5	0.01	9	116259	0.29
nw13	50132.00	50146	*				1	10	2.02	1.94	11	0.01	27	50146	2.02
nw17	10875.75	11115	11115	0.00%	101	44	5	60	30.77	26.64	77	3.58	141	11865	29.74
nw18	338864.25	340160	*				1	33	5.13	3.62	35	1.32	77	363576	4.63
nw20	16626.00	16812	*				1	12	0.12	0.09	13	0.01	20	16965	0.12
nw21	7380.00	7408	*				1	16	0.09	0.05	17	0.00	22	7408	0.09
nw22	6942.00	6984	*				1	8	0.08	0.06	9	0.02	12	6984	0.08
nw23	12317.00	12534	*				1	8	0.08	0.04	9	0.01	14	12534	0.08
nw24	5843.00	6314	*				2	11	0.09	0.06	15	0.02	26	6514	0.09
nw25	5852.00	5960	*				1	16	0.12	0.07	17	0.02	29	5960	0.12

Table 4.2: Feasible solution heuristic (default setting), Set 1, part 2

			Feasible solution heuristic							LP		Reduce		First feas sol	
name	lp opt	int opt	ub	o gap	cols	rows	M	m	time	time	freq	time	freq	ub	time
nw26	6743.00	6796	*				1	10	0.08	0.06	11	0.01	21	6796	0.08
nw27	9877.50	9933	*				1	13	0.11	0.06	13	0.02	27	9933	0.11
nw28	8169.00	8298	*				2	11	0.11	0.08	15	0.03	24	9630	0.09
nw29	4185.33	4274	*				4	23	0.44	0.34	28	0.06	50	4378	0.38
nw30	3726.80	3942	*				1	13	0.23	0.15	14	0.06	21	3942	0.23
nw31	7980.00	8038	*				1	11	0.28	0.20	12	0.06	18	8046	0.28
nw32	14570.00	14877	*				2	14	0.07	0.05	16	0.02	26	15120	0.06
nw33	6484.00	6678	*				1	11	0.48	0.35	11	0.08	24	6682	0.48
nw34	10453.50	10488	*				1	10	0.10	0.06	11	0.03	21	10701	0.09
nw35	7206.00	7216	*				1	3	0.13	0.08	4	0.04	6	7216	0.13
nw36	7260.00	7314	7314	0.00%	29	9	3	17	0.35	0.25	19	0.07	28	7378	0.31
nw37	9961.50	10068	*				1	9	0.10	0.04	9	0.01	24	10068	0.10
nw38	5552.00	5558	*				2	15	0.20	0.13	16	0.05	23	5630	0.19
nw39	9868.50	10080	*				2	17	0.08	0.07	20	0.01	26	10758	0.06
nw40	10658.25	10809	*				2	24	0.13	0.06	25	0.01	45	11838	0.10
nw41	10972.50	11307	*				1	10	0.04	0.03	11	0.00	20	11838	0.04
nw42	7485.00	7656	*				1	14	0.18	0.13	15	0.01	23	7656	0.18
nw43	8897.00	8904	*				1	16	0.13	0.09	17	0.02	29	8904	0.13
us01	9963.07	10022	10101	0.79%	616	73	3	74	717.90	367.97	84	347.83	131	10199	715.82
us04	17731.67	17854	*				1	16	2.72	1.24	18	1.43	28	17862	2.71

Table 4.3: Feasible solution heuristic (default setting), Set 3

		best	Feasible solution heuristic							LP		Reduce		First feas sol	
name	lp opt	feas sol	ub	i gap	cols	rows	M	m	time	time	freq	time	freq	ub	time
v0415	2423977.00	* 2429415	2435833	0.49%	4294	763	1	16	0.97	0.79	16	0.07	26	2435833	0.94
v0416	2715490.67	* 2725602	2736885	0.79%	3609	982	1	27	1.15	0.74	27	0.21	40	2736885	1.12
v0417	2603308.50	* 2611518	2622525	0.74%	55584	894	1	22	11.68	9.62	22	0.49	28	2622525	11.21
v0418	2836836.67	* 2845425	2855469	0.66%	4761	951	1	35	1.25	0.81	35	0.17	45	2855469	1.21
v0419	2582994.00	* 2590326	2598124	0.59%	2870	822	1	14	0.71	0.54	14	0.05	14	2598124	0.69
v0420	1688793.33	* 1696889	1703734	0.88%	2636	590	1	22	0.58	0.38	22	0.07	27	1703734	0.57
v0421	1848949.00	* 1853951	1858977	0.54%	1171	464	1	11	0.24	0.17	11	0.01	13	1858977	0.24
v1616	1002954.62	* 1006460	1018536	1.55%	27011	1285	1	516	147.09	37.62	516	81.36	729	1018536	146.90
v1617	1098263.23	1102586	1115503	1.57%	44009	1458	1	642	239.34	68.43	642	126.95	893	1115503	238.96
v1618	1147777.67	1154458	1166107	1.60%	88852	1434	1	585	271.00	81.85	585	140.09	873	1166107	270.23
v1619	1150943.29	1156338	1168481	1.52%	44271	1476	1	507	215.62	50.16	508	128.08	783	1168481	215.20
v1620	1136666.52	* 1140604	1152624	1.40%	87536	1412	1	623	346.84	94.64	624	191.57	897	1152624	345.99
v1621	822339.42	* 825563	834602	1.49%	9758	854	1	328	54.20	11.64	328	29.95	495	834602	54.14
v1622	790076.50	* 793445	800572	1.33%	7620	787	1	47	2.60	1.96	47	0.26	76	800572	2.56
t0415	5125429.50	5590096	F				1	10	72.70	46.40	11	23.62	29		
t0416	5829948.77	6130217	F				1	9	120.29	90.81	10	26.73	28		
t0417	5610564.20	6043157	F				1	9	83.62	59.70	10	21.90	20		
t0418	6142664.90	6550898	F				1	10	229.27	179.83	11	45.91	31		
t0419	5644051.00	5916956	F				1	5	63.35	43.31	6	18.58	18		
t0420	3983951.22	4276444	F				1	10	21.74	12.84	11	8.00	30		
t0421	4057701.31	4354411	F				1	8	17.45	11.67	9	4.94	20		
t1716	121648.87	161636	F				1	69	89.30	66.44	69	20.23	235		
t1717	134531.02	184692	210489	56.46%	16428	551	1	87	113.80	83.97	88	26.36	263	210489	113.77
t1718	126334.47	162992	204086	61.54%	16310	523	1	95	132.71	103.45	96	25.87	298	204086	132.69
t1719	138708.87	187677	F				1	85	133.18	98.67	85	30.39	302		
t1720	126333.20	172752	200679	58.85%	16195	538	1	76	124.79	92.96	77	28.43	265	200679	124.77
t1721	103748.46	127424	F				1	98	37.38	26.32	98	9.37	265		

Table 4.4: Feasible solution heuristic (default setting), Sets 2 and 4

		Feasible solution heuristic							LP		Reduce		First feas sol	
name	lp opt	ub	i gap	cols	rows	M	m	time	time	freq	time	freq	ub	time
0321.4	35742.46	F				1	11	1634.92	1253.53	12	376.09	34		
0331.3	28402.76	F				1	18	290.63	157.68	19	129.84	58		
0331.4	29730.03	F				1	21	483.03	360.83	21	118.10	63		
0341.3	31004.06	F				1	30	893.40	742.66	31	147.37	85		
0341.4	34276.06	F				1	20	215.27	135.13	21	77.49	62		
0351.3	35032.59	F				1	16	1645.49	1189.50	17	449.47	51		
0351.4	34434.36	F				1	13	1835.28	1494.46	14	334.99	41		
nf260	47405.00	* 47420				1	1	40.38	37.86	2	1.47	2	47420	38.90
sp1	9987.80	11482	14.96%	5148	198	2	58	78.38	54.00	64	23.92	94	11482	55.22
sp2	13522.93	* 13914				1	16	6.56	3.22	18	3.19	25	13914	6.19
sp3	12766.12	* 12943				1	10	1.18	0.84	12	0.27	14	12943	1.10
sp4	11389.42	F				1	39	58.07	30.11	40	27.39	60		
sp5	27403.20	27637	0.85%	3764	587	2	23	1181.31	1088.97	31	90.55	50	27637	789.92
sp6	157414.80	F				1	5	7144.31	6943.81	6	199.89	5		
sp7	162349.98	F				1	1	7289.45	7234.99	2	54.28	1		
sp8	368714.87	F				1	1	7170.27	6957.01	2	212.75	2		
sp9	166705.53	F				1	18	7173.32	6595.98	19	571.17	46		
sp10	43045.72	F				1	24	24.22	13.49	25	10.23	47		
sp11	3093.13	F				1	14	0.88	0.66	15	0.18	14		
sp12	248004.45	F				1	1	7068.51	6662.20	2	405.60	3		
sp14	250210.43	F				1	1	7153.19	7027.61	2	125.18	2		

Table 4.5: Feasible solution heuristic (comparison runs), Set 1, part 1

		dual simpl w/ basis warmst				no follow-on				follow-on threshold .79				follow-on w/ small dual abs			
name	opt	ub	time	time	freq	ub	time	M	m	ub	time	M	m	ub	time	M	m
aa01	56137	56153	454.52	415.83	146	F	32.57	1	73	56172	37.43	2	58	57117	88.43	3	193
aa03	49649	49649	87.11	79.37	88	49649	23.12	3	210	49649	19.02	2	31	49649	21.44	2	40
aa04	26374	26555	151.63	131.34	185	F	13.44	1	61	27959	14.04	1	88	27009	37.10	3	194
aa05	53839	54250	96.75	82.17	138	F	19.81	1	106	53941	15.80	2	66	53904	21.61	3	104
aa06	27040	27040	47.37	40.92	47	27040	22.76	3	150	27040	19.44	3	79	27040	13.84	2	43
kl01	1086	*	1.59	1.22	37	1091	3.31	2	89	*	1.95	2	45	*	2.25	2	28
kl02	219	219	10.28	8.25	65	219	11.45	3	147	220	6.70	2	43	219	7.61	2	70
nw03	24492	*	12.11	10.60	51	*	20.97	2	118	*	11.86	2	25	*	11.76	2	21
nw04	16862	16942	24.14	19.67	39	22494	73.37	2	61	17158	19.12	2	19	17004	17.42	2	16
nw06	7810	9344	1.50	1.27	26	*	2.39	1	73	*	1.60	3	16	*	1.67	2	29
nw11	116256	*	0.24	0.22	5	*	0.27	1	7	*	0.28	1	4	*	0.24	1	4
nw13	50146	*	1.99	1.93	10	*	2.08	1	36	*	1.97	1	10	*	1.99	1	9
nw17	11115	11115	30.21	26.83	17	67719	48.02	1	124	11196	30.73	4	58	11115	33.54	5	99
nw18	340160	*	8.79	7.21	26	408414	7.23	1	97	*	5.01	1	22	408724	5.09	1	52
nw20	16812	*	0.10	0.04	10	*	0.20	1	29	*	0.07	1	12	*	0.11	1	9
nw21	7408	*	0.08	0.04	16	*	0.13	1	23	*	0.12	1	16	*	0.08	1	16
nw22	6984	*	0.10	0.05	10	*	0.29	2	50	*	0.09	1	8	*	0.07	1	7
nw23	12534	*	0.07	0.04	9	*	0.20	1	34	*	0.08	1	8	*	0.06	1	4
nw24	6314	*	0.11	0.04	15	*	0.15	1	12	*	0.09	2	11	*	0.09	2	14
nw25	5960	*	0.07	0.06	12	*	0.14	1	16	*	0.13	1	16	*	0.08	1	10

Table 4.6: Feasible solution heuristic (comparison runs), Set 1, part 2

		fix vars at one to 1				symm diff on vars at one				force LP resolve at 20%				short cols not protected			
name	opt	ub	time	M	m	ub	time	M	m	ub	time	M	m	ub	time	M	m
aa01	56137	F	19.77	1	41	F	40.63	1	47	F	24.50	1	19	56506	47.66	2	70
aa03	49649	49664	7.25	3	39	47971	7.04	2	18	49649	10.90	2	9	49649	21.42	2	49
aa04	26374	27376	20.64	2	109	26740	32.19	3	181	F	14.60	1	24	26541	26.84	3	120
aa05	53839	54144	13.32	3	91	52992	17.27	4	122	F	9.84	1	37	53935	19.79	3	117
aa06	27040	27045	8.56	3	47	27040	11.71	4	81	27040	12.41	4	24	27040	14.71	2	55
kl01	1086	*	1.37	2	25	*	1.56	2	38	*	1.57	2	35	1089	1.76	2	28
kl02	219	220	5.27	2	62	219	6.12	3	92	220	6.32	2	64	219	9.41	3	66
nw03	24492	25125	10.66	3	21	25086	10.66	3	22	*	11.55	2	30	*	13.33	2	26
nw04	16862	18016	16.64	2	23	17004	16.14	2	9	16964	19.33	3	26	16942	20.23	2	17
nw06	7810	*	1.33	1	15	*	1.60	2	12	*	1.26	1	19	*	1.37	1	13
nw11	116256	*	0.28	1	4	*	0.38	1	35	*	0.29	1	7	*	0.27	1	4
nw13	50146	*	1.99	1	10	*	2.07	1	33	*	2.00	1	16	*	1.99	1	6
nw17	11115	11673	30.19	3	46	11382	31.63	3	51	11115	32.42	5	33	11115	36.67	4	49
nw18	340160	*	5.20	3	68	*	6.57	2	47	419616	3.77	1	33	367156	7.47	2	58
nw20	16812	*	0.08	1	8	*	0.13	2	14	*	0.09	1	9	*	0.10	1	9
nw21	7408	*	0.10	2	18	*	0.08	1	13	*	0.09	1	18	*	0.08	1	9
nw22	6984	*	0.08	1	5	*	0.06	1	5	*	0.09	1	8	*	0.10	1	8
nw23	12534	*	0.06	1	4	*	0.07	1	7	*	0.06	1	5	*	0.05	1	5
nw24	6314	*	0.08	2	10	*	0.10	2	8	*	0.13	2	20	*	0.09	2	11
nw25	5960	*	0.11	1	11	*	0.11	1	16	*	0.11	1	15	*	0.09	1	8

Table 4.7: Feasible solution heuristic (comparison runs), Set 1, part 1, cont.

		dual simpl w/ basis warmst				no follow-on				follow-on threshold .79				follow-on w/ small dual abs			
name	opt	ub	time	time	freq	ub	time	M	m	ub	time	M	m	ub	time	M	m
nw26	6796	*	0.06	0.03	11	*	0.14	1	19	*	0.10	1	10	*	0.07	1	12
nw27	9933	*	0.10	0.06	11	*	0.24	1	46	*	0.12	1	13	*	0.10	1	13
nw28	8298	*	0.07	0.05	11	*	0.18	1	6	*	0.12	2	11	8688	0.11	2	11
nw29	4274	4338	0.39	0.27	26	4324	1.56	4	197	4274	0.35	3	15	4324	0.40	3	26
nw30	3942	*	0.20	0.13	12	*	0.44	1	28	*	0.21	1	14	*	0.21	1	11
nw31	8038	*	0.30	0.20	11	8038	0.66	2	23	*	0.28	1	11	*	0.27	1	10
nw32	14877	*	0.08	0.04	15	*	0.14	1	25	*	0.10	2	14	*	0.06	1	11
nw33	6678	*	0.45	0.33	13	*	1.17	1	28	*	0.50	1	11	*	0.46	1	13
nw34	10488	*	0.08	0.05	11	*	0.19	1	25	*	0.10	1	10	*	0.09	1	11
nw35	7216	*	0.10	0.08	5	*	0.29	1	20	*	0.13	1	3	*	0.13	1	13
nw36	7314	7314	0.37	0.26	26	7314	0.71	4	36	7314	0.38	4	21	7322	0.33	2	13
nw37	10068	*	0.09	0.05	10	*	0.21	1	30	*	0.07	1	9	*	0.07	1	9
nw38	5558	*	0.19	0.14	15	5558	0.30	2	32	*	0.19	2	15	5592	0.19	3	16
nw39	10080	*	0.06	0.04	13	*	0.16	1	28	*	0.12	2	17	*	0.07	2	7
nw40	10809	*	0.06	0.05	13	*	0.17	1	33	*	0.12	2	24	*	0.07	1	12
nw41	11307	*	0.04	0.01	11	*	0.09	1	17	*	0.03	1	10	*	0.04	1	11
nw42	7656	*	0.14	0.10	13	*	0.34	1	46	*	0.17	1	14	*	0.18	2	15
nw43	8904	*	0.13	0.07	18	*	0.27	1	48	*	0.14	1	16	*	0.13	1	12
us01	10022	10036	1255.46	841.04	31	10222	1326.31	2	98	10052	738.07	2	27	10036	360.96	4	94
us04	17854	*	3.18	1.71	31	*	5.05	1	59	*	2.75	1	16	*	2.64	1	33

Table 4.8: Feasible solution heuristic (comparison runs), Set 1, part 2, cont.

		fix vars at one to 1				symm diff on vars at one				force LP resolve at 20%				short cols not protected			
name	opt	ub	time	M	m	ub	time	M	m	ub	time	M	m	ub	time	M	m
nw26	6796	*	0.07	1	8	*	0.08	1	9	*	0.06	1	9	*	0.08	1	8
nw27	9933	*	0.09	1	9	*	0.08	1	11	*	0.11	1	9	*	0.10	1	10
nw28	8298	8688	0.11	2	11	*	0.09	2	10	*	0.13	2	11	*	0.09	2	8
nw29	4274	4338	0.40	3	21	*	0.42	4	24	*	0.41	4	17	4338	0.41	3	18
nw30	3942	*	0.26	1	11	*	0.24	1	12	*	0.21	1	9	*	0.24	1	14
nw31	8038	8046	0.24	3	9	8046	0.24	3	11	*	0.28	1	9	*	0.29	1	12
nw32	14877	*	0.05	2	7	*	0.06	2	11	*	0.08	2	13	*	0.08	2	10
nw33	6678	*	0.37	1	1	*	0.41	1	9	*	0.50	1	13	*	0.51	1	11
nw34	10488	*	0.07	1	5	*	0.06	1	7	*	0.10	1	9	*	0.07	1	9
nw35	7216	*	0.09	1	1	*	0.10	1	2	*	0.11	1	3	*	0.12	1	5
nw36	7314	7314	0.39	3	20	7314	0.41	3	19	7314	0.34	4	16	7314	0.32	3	15
nw37	10068	*	0.05	1	5	*	0.07	1	9	*	0.09	1	9	*	0.09	2	11
nw38	5558	*	0.20	2	10	*	0.20	2	12	*	0.17	2	12	*	0.16	2	12
nw39	10080	*	0.06	2	7	*	0.07	2	8	*	0.09	2	8	*	0.07	2	10
nw40	10809	*	0.08	1	15	*	0.11	2	24	*	0.07	1	16	*	0.06	1	11
nw41	11307	*	0.03	1	4	*	0.03	1	7	*	0.06	1	7	*	0.03	1	8
nw42	7656	*	0.17	1	13	*	0.18	1	13	*	0.16	1	12	*	0.14	1	11
nw43	8904	*	0.13	1	19	*	0.12	1	15	*	0.10	1	17	*	0.12	1	10
us01	10022	10051	717.59	3	61	10051	716.54	3	61	10051	554.50	4	45	10036	595.95	2	27
us04	17854	*	1.25	1	9	*	1.61	1	11	*	1.84	1	11	*	2.35	1	17

Table 4.9: Feasible solution heuristic (comparison runs), Set 3, part 1

		dual simpl w/ basis warmst				no follow-on				follow-on threshold .79				follow-on w/ small dual abs			
name	opt	ub	time	time	freq	ub	time	M	m	ub	time	M	m	ub	time	M	m
v0415	* 2429415	2437371	0.93	0.72	15	2436517	1.05	1	50	2435833	0.93	1	16	2436386	0.92	1	16
v0416	* 2725602	F	1.18	0.72	38	2738976	1.29	1	54	F	0.99	1	11	2739043	1.21	1	29
v0417	* 2611518	2622525	11.61	9.51	22	2619937	11.68	1	22	2622525	11.82	1	22	2622525	11.79	1	22
v0418	* 2845425	2854403	1.25	0.76	35	2855466	1.24	1	46	2858585	1.32	1	34	2856221	1.30	1	37
v0419	* 2590326	2598124	0.69	0.51	14	2601032	0.69	1	17	2598124	0.72	1	14	2598124	0.73	1	14
v0420	* 1696889	1703734	0.56	0.38	22	1706883	0.54	1	26	1703734	0.56	1	22	1704283	0.61	1	21
v0421	* 1853951	1858977	0.25	0.17	11	1859428	0.21	1	9	1858977	0.24	1	11	1858977	0.22	1	11
v1616	* 1006460	F	144.62	19.57	532	1024509	710.38	1	1378	F	143.67	1	433	1012015	65.26	1	94
v1617	1102586	1112822	215.39	34.05	641	1115627	1094.06	1	1523	1114867	243.71	1	660	1111830	241.50	1	450
v1618	1154458	1173014	259.83	51.10	684	1176239	1204.09	1	1693	1167792	276.45	1	603	1164444	286.54	1	492
v1619	1156338	F	195.58	28.41	381	1182351	1157.68	1	1711	1164669	222.13	1	530	1169130	268.49	1	526
v1620	* 1140604	1151571	341.81	106.93	581	1151869	1019.66	1	1585	1147928	348.13	1	655	1151575	290.80	1	411
v1621	* 825563	835181	47.06	5.60	299	831691	148.05	1	668	839018	49.74	1	296	F	53.53	1	272
v1622	* 793445	801323	2.54	1.89	46	F	2.95	1	94	F	2.38	1	25	F	2.46	1	27
t0415	5590096	F	129.35	103.88	12	F	45.91	1	24	F	101.72	1	11	F	72.68	1	11
t0416	6130217	F	165.69	137.41	9	F	52.03	1	18	F	69.50	1	8	F	83.60	1	11
t0417	6043157	F	218.08	186.10	12	F	75.21	1	19	F	40.04	1	4	F	96.45	1	11
t0418	6550898	F	366.26	318.32	10	F	100.89	1	19	F	114.78	1	8	F	180.81	1	7
t0419	5916956	F	159.24	132.95	7	F	47.56	1	16	F	77.98	1	9	F	111.77	1	8
t0420	4276444	F	43.29	35.35	12	F	13.76	1	17	F	19.49	1	12	F	18.51	1	8
t0421	4354411	F	32.82	26.39	10	F	14.37	1	18	F	27.72	1	11	F	19.43	1	11
t1716	161636	188291	422.27	401.48	77	F	128.45	1	279	F	80.55	1	83	F	120.72	1	62
t1717	184692	219528	993.33	963.88	123	F	218.83	1	376	F	134.10	1	69	F	126.55	1	78
t1718	162992	194455	852.58	823.70	105	F	236.18	1	441	F	131.34	1	102	224998	110.11	1	88
t1719	187677	240557	1042.84	1009.56	74	F	204.34	1	341	221419	136.63	1	103	246664	258.37	1	105
t1720	172752	208707	937.57	905.45	79	F	207.44	1	353	F	139.82	1	69	254175	130.12	1	101
t1721	127424	163035	207.34	197.34	83	F	79.45	1	372	176002	51.18	1	61	152250	33.22	1	91

Table 4.10: Feasible solution heuristic (comparison runs), Set 3, part 2

		fix vars at one to 1				symm diff on vars at one				force LP resolve at 20%				short cols not protected			
name	opt	ub	time	M	m	ub	time	M	m	ub	time	M	m	ub	time	M	m
v0415	* 2429415	2435833	0.92	1	14	1133371	3.54	1	49	2436974	0.96	1	10	2436168	0.89	1	14
v0416	* 2725602	2742706	1.04	1	26	1444320	6.30	1	48	F	1.01	1	11	2737659	1.04	1	20
v0417	* 2611518	2622525	11.89	1	21	1237709	18.36	1	70	F	10.50	1	11	2619237	11.52	1	17
v0418	* 2845425	2855469	1.24	1	34	F	4.39	1	53	2857629	1.20	1	26	2855279	0.98	1	17
v0419	* 2590326	2598124	0.76	1	13	1319864	4.61	1	26	F	0.62	1	6	2601974	0.71	1	12
v0420	* 1696889	1707732	0.52	1	14	1055119	2.44	1	51	1704067	0.52	1	15	F	0.42	1	5
v0421	* 1853951	1858977	0.23	1	10	901564	0.44	1	20	F	0.20	1	4	1858994	0.23	1	10
v1616	* 1006460	1021108	18.67	1	228	835990	46.09	1	285	F	110.45	1	368	F	47.14	1	81
v1617	1102586	1115769	29.15	1	219	907417	76.80	1	208	1114655	191.40	1	532	F	72.90	1	69
v1618	1154458	1176261	43.11	1	172	954214	76.01	1	203	1171281	219.26	1	513	F	110.95	1	77
v1619	1156338	1171019	31.79	1	205	1002471	76.15	1	305	F	144.71	1	305	F	76.55	1	71
v1620	* 1140604	F	52.76	1	220	962187	87.65	1	295	1149607	246.42	1	476	F	138.75	1	108
v1621	* 825563	836837	3.32	1	47	733645	6.96	1	82	835862	35.70	1	221	831757	15.02	1	58
v1622	* 793445	800250	2.48	1	35	709091	6.69	1	71	800247	2.46	1	30	800799	2.27	1	27
t0415	5590096	F	67.02	1	13	F	69.04	1	11	F	59.27	1	2	F	80.40	1	15
t0416	6130217	F	79.15	1	10	F	117.89	1	11	F	64.88	1	3	F	109.36	1	5
t0417	6043157	F	75.48	1	7	F	82.88	1	9	F	37.03	1	2	F	89.02	1	10
t0418	6550898	F	121.53	1	10	F	229.28	1	10	F	72.53	1	2	F	213.42	1	8
t0419	5916956	F	58.12	1	8	F	72.90	1	7	F	61.15	1	3	F	65.09	1	8
t0420	4276444	F	18.38	1	13	F	21.61	1	10	F	14.82	1	2	F	26.43	1	13
t0421	4354411	F	26.89	1	9	F	17.37	1	8	F	14.80	1	3	F	17.72	1	8
t1716	161636	189583	104.08	1	68	F	78.19	1	66	F	139.39	1	22	F	68.01	1	46
t1717	184692	231206	115.88	1	85	212770	124.10	1	73	207419	109.22	1	31	F	101.43	1	45
t1718	162992	193250	140.22	1	78	191408	135.31	1	93	F	67.33	1	30	F	94.14	1	48
t1719	187677	198186	122.88	1	91	F	127.15	1	102	F	71.56	1	37	F	149.29	1	28
t1720	172752	199804	123.38	1	87	200026	127.43	1	85	F	83.03	1	31	F	219.44	1	50
t1721	127424	F	36.12	1	81	158624	36.99	1	93	F	22.82	1	58	160467	29.70	1	54

Chapter 5

Interfacing with the Branch-and-Cut framework

Our Branch-and-Cut procedure, implemented using the COMPSys framework (Section 2.2), was applied to those problems that the Feasible Solution Heuristic did not solve to optimality (Section 4.4).

In the first section we describe the framework in further detail, emphasizing those points whose implementation was nontrivial for our application. The later sections will discuss these points in detail. Appendix D lists those COMPSys parameters that were important in our case, as well as those parameters that we added in our user-written functions. A detailed description of the user-written functions can be found in [EL97].

5.1 The COMPSys framework

As described earlier (Section 2.2), the COMPSys framework employs a master-slaves model. The functions of the master are split between two processes: the *Master* handles the input/output, maintains and distributes on request the (user-supplied) problem-specific information, and stores the best feasible solution found so far; while the *Tree Manager* keeps track of the search tree and distributes work to the slaves. The tasks were split this way to keep the Tree Manager generic, thus completely internal to the framework. All the problem-specific information is handled through the user functions of the Master process. Of course, the Master and the Tree Manager can run simultaneously on the same processor.

There are four classes of slave processes. Search tree nodes are sent to the *LP processes*. Each LP process has an associated *Cut Generator* where separation is attempted. *Cut Pool* processes corresponding to subtrees of the search tree maintain a collection of inequalities valid in the subtrees. If decomposition methods are used during separation, *Solution Pool* processes (also corresponding to subtrees) store extreme points of the enclosing polytope ([Ral95]). Since our implementation does not use decomposition, we will not discuss this feature here in more detail.

Besides the master and slaves processes there is a process reserved for the Graphical User Interface through which other processes can graphically display information. The GUI was implemented as a separate process mainly for technical reasons. The GUI is discussed in detail in Chapter 6.

Now we describe the general execution flow of COMPSys. Figure 5.1 (borrowed from [ELRT97]) illustrates the main tasks of and the information flow between the various processes. The user starts only the Master process which will spawn the Tree Manager. All the slave processes are spawned by the Tree Manager.

The Master (M) process starts by reading in the parameters from a file and requests the user to read in all the problem-specific input. Then the user is given control to run preprocessing and/or upper bounding procedures before the actual Branch-and-Cut starts. Now the Tree Manager is spawned, the user formulates the problem and constructs the root node of the search tree. Then the Master enters a loop of waiting for messages (like requests for problem-specific data, or arrival of new feasible solutions) and processing them.

The Tree Manager (TM) spawns the slave processes and sends the root node to an LP process, then it awaits new search tree nodes created by branching in the LP processes and in turn sends them out to idle LP processes. Parameters govern which search tree node is selected next for processing.

When an LP process (LP) is spawned, initial information is obtained from the Master; then the LP waits for a search tree node. Upon receiving a search tree node a loop is entered where first the user creates the corresponding LP relaxation and the relaxation is solved. If the LP relaxation is infeasible or the LP optimal value is higher than the current upper bound then the search tree node is fathomed. Otherwise, if the LP optimal solution is feasible for the original problem, then it is sent to the master process, the current best upper bound is updated and the node is fathomed. If *column generation* is desired then columns that would improve the objective value are added instead of fathoming (we so not use this feature in our

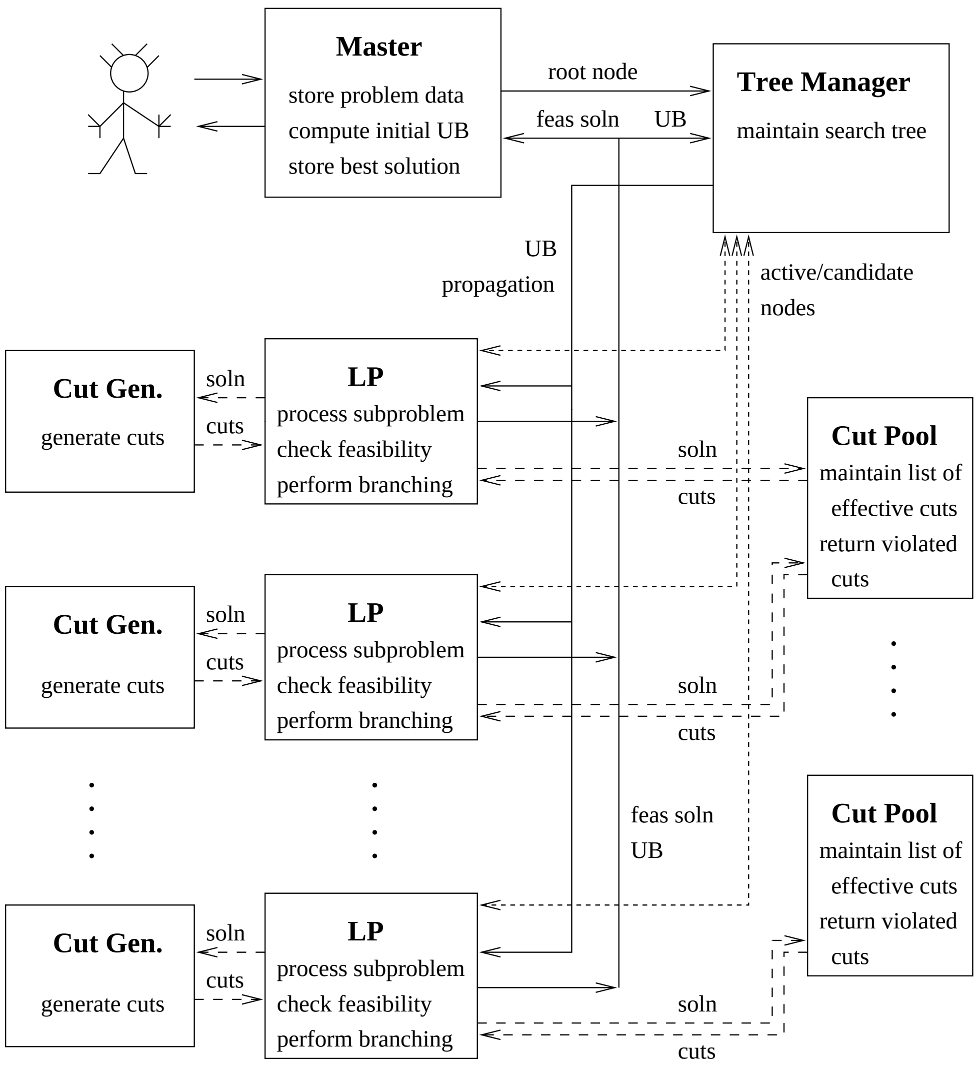


Figure 5.1: Processes of the COMPSys framework

implementation). Now the LP optimal solution is sent to the corresponding Cut Generator (and possibly to the Cut Pool) to obtain violated inequalities. While the Cut Generator and Cut Pool are working, the LP process identifies and possibly removes ineffective inequalities and variables that can be fixed to their bounds. Variable fixing consists of reduced cost fixing (done by the framework) and *logical fixing* done by the user. At this point the user can generate violated inequalities within the LP process as well (some cut types, like Gomory cuts, require the LP optimal tableau, which would be difficult to reproduce in the Cut Generator, and costly to send over). Next, inequalities obtained internally or from the Cut Generator and the Cut Pool are processed (may be *lifted* by the user), selectively added to the problem formulation and sent to the Cut Pool. If sufficiently strong violated inequalities have been generated, then the execution continues with re-solving the LP relaxation at the top of the loop. Otherwise, candidate branching objects (variables and constraints) are selected by the user and strong branching is performed. The new search tree nodes are sent back to the Tree Manager and the LP process waits for the next search tree node (possibly one of the newly generated nodes; this is called *diving*).

The Cut Generator (CG) process also obtains initial information from the Master. Afterwards it repeatedly receives LP optimal solutions from the corresponding LP process and returns valid inequalities violated by the solution. Note that the LP process may decide to re-solve the LP relaxation without waiting for all the cuts the Cut Generator could generate. In this case the Cut Generator will continue working with the old LP optimal solution until it receives a new one or it is not able to find more inequalities. This may cause the Cut Generator to lag behind the LP process.

The Cut Pool (CP) process, just like the other slaves, obtains initial information from the Master. Valid inequalities arriving from the LP process are stored away. When an LP optimal solution arrives, it is tested against the stored inequalities and those found to be violated are sent back to the LP process.

Before we describe in detail how we implemented the user-written functions of the Master, LP and Cut Generator processes for the SPP, we summarize the most important points in the light of the paragraphs above.

- **Master process**
  - Preprocessing and upper bounding before Branch-and-Cut starts
  - Formulating the problem and constructing the root node of the search tree
- **LP process**
  - Constructing the LP relaxation
  - Logical fixing of variables
  - Generating violated inequalities
  - Lifting violated inequalities obtained here or in CG or CP
  - Deciding whether to branch or continue with solving LPs
  - Choosing branching objects for strong branching and comparing the presolved results

- **Cut Generator process**
  - Generating and lifting violated inequalities
  - Generating violated inequalities "by hand" using the GUI

5.2 User-written functions of the Master process

The input/output functions that need to be supplied by the user (like reading in the parameter file and the problem instance and displaying a solution), and the function that interprets a feasible solution received from another process were straightforward to implement. The following two issues needed special consideration.

5.2.1 Preprocessing and upper bounding

This is where our initial Reduce() (Chapter 3) and Feasible Solution Heuristic (Chapter 4) could be invoked. Since we implemented these as separate programs, here we simply reproduce the matrix that was obtained at the end of our heuristic (Section 4.4).

The framework uses the dual simplex method for (re)optimizing the LP relaxations. For large Set Partitioning Problems this is very expensive (Section 4.3), so we solve the initial LP relaxation here using the barrier method with dual crossover and pass on the optimal basis to be used as warmstart information.

5.2.2 Formulating the problem and constructing the root

Our implementation reads in the (column ordered) problem matrix along with an upper bound and a feasible solution (if they exist). All this information, which may be modified during preprocessing, is passed on to both the LP and Cut Generator processes.

COMPSys distinguishes between two classes of variables and constraints (cuts, rows): base and extra. Base variables and constraints are always in the formulation for any search tree node, while extra variables and constraints may be removed. The user has to designate which initial variables and constraints belong to which class. New extra variables are obtained by column generation, and new extra constraints by separation. Information about base variables and constraints is sent to the LP process only once, while extra objects have to be communicated back and forth between the LP and the Tree Manager, increasing message size. On the other hand, having too many base objects is not only memory consuming, but it also slows down LP solving. This trade-off has to be balanced for each problem class.

Since SPPs usually have a large number of columns many of which can be eliminated by logical implications (i.e.; Reduce) as we go down in the search tree, we have decided to designate all variables as extra variables and to include all of them in the root description (so no column generation is necessary). On the other hand, to simplify our implementation we characterized all constraints as base constraints.

Also, the starting LP basis was taken as the optimal basis obtained from solving the initial LP relaxation during preprocessing.

5.3 User-written functions of the LP process

5.3.1 Constructing the LP relaxation

Here we simply put together the set partitioning problem matrix consisting of columns currently in the formulation (since all variables are extra variables, some are eliminated because of branching and logical implications as we go down in the search tree), the objective function, and the right hand side vectors. The framework will append the constraints corresponding to cuts that were added to the formulation during the solution process.

5.3.2 Logical fixing of variables

When a variable is permanently set to one of its bounds at a search tree node, logical implications, like those employed in Reduce() (Section 3.1), might further decrease the size of the problem. Also, if the reduction is significant, it might be worthwhile to look for an (improving) feasible solution.

Although the current problem formulation most likely will contain cuts that were not among the constraints of the original set partitioning problem, the reduction rules implemented in Reduce() apply to SPPs only. Originally we carried out the reductions based on the original set partitioning rows only, but it proved to be beneficial to include all equality cuts that have unit coefficients and right hand side values (the formulation remains a set partitioning formulation) and to incorporate similar constraints with a zero on the right hand side by marking the corresponding variables to zero. Such constraints are frequent when we allow branching on cuts (cliques) or threshold and follow-on branching (Section 5.3.6).

Our implementation of the logical fixing of variables always invokes Reduce() if at least one variable is newly fixed to one (setting variables to one usually implies other reductions) or the number of variables not yet fixed to their bounds decreased significantly since the last time logical fixing was applied at this search tree node.

The heuristic fixing phase of our Feasible Solution Heuristic is invoked only if some variables were fixed to one or the reduction was significant during logical fixing. How large a percentage of the variables must be eliminated before the heuristic is attempted depends on whether an upper bound is already known and on the size of the gap (the smaller the gap the less likely the heuristic is invoked). The Feasible Solution Heuristic is always warmstarted with the optimal basis and primal-dual information of the most recent LP relaxation solved by the framework.

If a (better) feasible solution is found by the heuristic, it is sent back to the Master (and the value to the Tree Manager) which will in turn propagate it to the other slave processes.

We will discuss the effectiveness of our logical fixing and feasible solution heuristic later in Section 5.5.

5.3.3 Generating violated inequalities

Currently we generate all our cuts in the Cut Generator process.

5.3.4 Lifting violated inequalities

Our goal here is to adjust and strengthen a given violated inequality (that may have been generated for an earlier solution) to the current problem formulation as much as possible. The framework can accommodate more than one lifted inequality for a cut, but we generate only one here. Lifting for cliques and (lifted) odd holes and antiholes is done sequentially; packing and cover odd holes are lifted simultaneously. See Section 2.3 for the definition of these cuts.

In sequential lifting first the variables not in the current formulation are removed (with the exception of variables on the odd hole or antihole) since these do not contribute to violation of the cut but could inhibit the addition of other variables which may contribute. Then some new variables are included into the inequality. Two lists of lifting candidates are formed; the first list will contain those variables at fractional levels in the current formulation, the second contains those at level zero. We will try to lift in the variables from the fractional list first, taking the variables in decreasing order of their value in the solution. Then the framework chooses the most violated inequalities to be included into the formulation. These cuts go through a second round of lifting; if the number of variables that were lifted in earlier for a cut does not exceed a given bound, those variables with the lowest reduced costs on the second list are considered (variables with zero reduced costs are always lifted in).

Clique inequalities are simply lifted by extending the cliques with new nodes that are adjacent to all nodes in the clique in the intersection graph. Thus a lifted clique is also a clique; we do not distinguish between cliques that are “fresh” from the cut generator and those that were already lifted before. Computing lifting coefficients for (lifted) odd holes and antiholes is done here exactly as in the Cut Generator (Section 5.4).

Packing and cover odd holes are lifted simultaneously by applying the Chvátal-Gomory procedure for the rows corresponding to the odd hole (Section 2.3); that is, these rows of the current formulation are added up, the sum is divided by two and the coefficients and the right hand side are rounded down for packing and up for cover odd holes.

5.3.5 Deciding whether to branch or continue solving LPs

After the current LP relaxation at the search tree node is solved and possibly some cuts are generated and/or received the user needs to decide whether to branch or continue with re-solving the LP relaxation. In general, we prefer cut generation to branching, since one overall goal is to keep the search tree small. On the other hand, a significant amount of time could be wasted in re-solving the LP relaxations if the added cuts fail to improve the relaxation sufficiently.

The framework provides two built-in options that we have experimented with: branching only if no violated inequalities could be generated, and branching if *tailing off* of the objective value is detected. The built-in function that checks tailing off was originally part of our set partitioning implementation but was later incorporated into the framework since it is a general procedure.

Two different tailing off interpretations are used in this function, the combined method determines tailing off when either interpretation determines so. The first

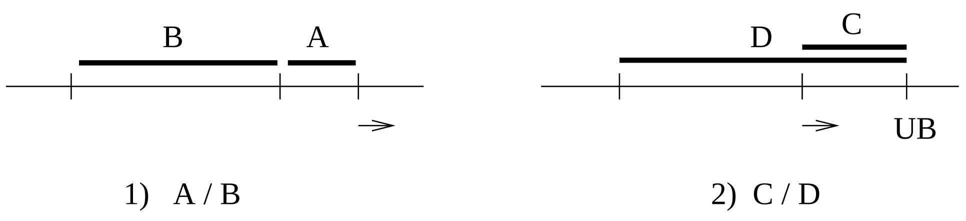


Figure 5.2: Tailing off

declares tailing off if the consecutive objective value differences decrease at a rate faster than geometric (that is, the average ratio of these differences is smaller than a given threshold). The second interpretation assumes the existence of an upper bound and declares tailing off if the distances of consecutive objective values from the upper bound decrease at a rate slower than geometric (that is, the average ratio of these distances is greater than a given threshold). Figure 5.2 illustrates which segments form the ratios in the two cases (the objective value increases in the direction of the arrow).

The values of the two thresholds mentioned above can be set through parameters. Another parameter pair determines the length of the "history" (how many ratios) included in the average. Tailing off is not checked if the history is not sufficiently long.

5.3.6 Choosing branching objects for strong branching and comparing the presolved results

Here we consider three decision points for branching (Section 5.1). First of all, a set of branching candidates needs to be selected for strong branching. Then, after the

LP relaxations at the would-be children of the current search tree node have been presolved by the framework for each branching candidate, one of the candidates must be chosen for branching. And third, based on the presolve information we need to decide what to do with each child node, in particular to decide which child should be selected to be kept at the current LP solver if diving is desired.

The COMPSys framework can handle both variable branching and branching on cuts. The framework provides cuts that were added previously because they were violated, but have since become slack (the constraint is no longer binding, the right-hand-side is not met with equality), and we can choose to branch on any of these. In addition to this, the framework accepts any branching object that can be described as a cut; that is, we provide coefficients for the variables currently in the formulation, a sense and a right-hand-side. Several standard default strategies are included for choosing branching variables. Our implementation is capable of picking a mix of four different branching objects: two kinds of cuts that we generate here (*follow-on* and *threshold* branching), formerly existing slack cuts, and finally, variables. The number of desired branching objects of each of these four types is determined by a parameter. The branching objects are generated in the above order; the standard default is to select at least a few branching variables if no branching cut of the previous three types is found.

Follow-on and threshold branching cuts are Type 3 Special Ordered Sets (as defined in [CPX95], Chapter 3), but while SOSs need to be specified before optimization for CPLEX, we create these sets based on the current LP optimal solution. In both cases we divide a row's support into two parts and in the two branches we require the variable covering the row to be chosen in one branch from the first part, and in the other from the second. Or, described as a cut (N^i denotes the support of some row i),

$$\sum_{j \in F} x_j = 1 \text{ or } 0, \quad F \subset N^i.$$

This is an extension of variable branching (where $|F|$ is one).

The advantage of these cuts is that they can be incorporated into the set partitioning formulation used by logical fixing and heuristics (Section 5.3.2).

Follow-on branching cuts are generated by determining row pairs for which the likelihood to be covered by the same variable (measured as the sum of primal solution values for the variables the two rows share) is between some given limits (parameters). We do not want this likelihood to be too large since this would be an overly conservative approach, and it is likely that the 0-branch is infeasible and the problem at the 1-branch is almost the same as at the parent. Therefore in our experiments we set the parameters so that the likelihood is higher than .5 but it is well below 1 (around .75). The follow-on row pairs are enumerated similarly as in the heuristic (Section 4.2.1) but we stop as soon as we have the required number of row pairs. We start by comparing the 5 rows with the largest absolute dual values to the remaining rows.

Threshold cuts are generated by selecting those rows that have the largest number of variables at fractional level in their supports, ordering the variables into decreasing order of their solution values for each row and considering the variables one by one until their combined solution values surpass a threshold (parameter). If all variables at fractional values in a row's support are selected then we drop the last variable from the chosen set since in this case the optimal solution to the LP relaxation in the 1-branch wouldn't be different from that at the parent.

The framework provides us with a list of all cuts that were added to the formulation previously (because they were violated) but since became slack. Since all cuts generated by the Cut Generator have integral coefficients and lifting maintains this property, the left hand side will be integral for any integral feasible solution. Therefore we measure the slack of a cut as the smaller of the distances of its left hand side from the nearest two integers. We examine all the slack cuts provided by the framework and choose the required number of cuts with the largest slacks (that is, those with slack values closest to .5, midway between the closest integer right hand side values). This is analogous to the "close to half" branching variable selection strategy (see below). Branching on a constraint like this, we expect to cut the feasible region deep in the middle, which is desirable during branching. Then two children are created as described in Section 2.1.2. At this time we allow branching only on clique inequalities. Branching on a slack clique inequality will result in two equality constraints, one with a zero and one with a 1 right hand side, so these cuts can be included into the logical fixing and heuristic as well.

We use three strategies for variable branching; the first one chooses variables with large fractional values and low cost to branch on ("close to one and cheap" rule). The logic here is that the zero branch will likely be short-lived since the variable must be replaced by many (most likely more expensive) other variables, resulting in a faster fathoming of the branch (due to high cost or infeasibility); in the one-branch the variable is fixed to one, which is likely to cause the elimination of additional variables during logical fixing.

The second option is to choose a variable "close to one-half and expensive". The effectiveness of this strategy is folklore; both of the branches are expected to "shake up" the solution, and the optimal value in the 1-branch is pushed up.

The third is a mixture of the other two; if there are variables at fractional level near one then the "close to one and cheap" rule is used, otherwise the "close to one-half and expensive" rule is applied.

Our second job here is to choose between the presolved branching candidates. There are several built-in options. The quality of a branching candidate can be evaluated based on either the presolved objective function values or the number of fractional values in the presolved LP solutions at the would-be children. Once the decision is made which evaluation method to use, either the lower or the higher of these values is selected from the values at the would-be children of each candidate, these values are compared across all the candidates and the candidate with either the lowest or the highest such value is chosen. Thus there are four cases for each evaluation method: choose the candidate with the lowest of the lower values at the children, highest of the lower values, lowest of the higher values or highest of the higher values. For example the "highest low objective value" rule means that the branching candidate whose child with the lower presolved objective value has the highest of these values is selected. Note that branching candidates with would-be children that can be fathomed based on the presolve information are preferred over the other candidates.

The built-in options can be intuitively explained as follows. The lowest low and lowest high objective value rules try to control the way the near-optimal solutions feasible at the current search tree node are inherited by the children (some feasible solutions might be shared). In the lowest low rule the child with the lower objective value is likely to keep most of these near-optimal feasible solutions; while in the lowest high case the children are more likely to retain roughly the same number of near-optimal solutions. The lowest low rule combined with diving into the lower child is traditionally used for finding near-optimal feasible solutions when the integrality gap is too large or no upper bound exists.

The highest low and highest high objective value rules aim to shape the search tree. The highest high rule tries to choose a candidate so that the child with the higher objective value can be quickly fathomed (thus producing a skewed search tree). On the other hand, the highest low rule tries to keep a uniformly high lower bound across the search tree (thus producing a balanced search tree). These rules are used when the current feasible solution is near-optimal and proving optimality is desired.

From the other four options the lowest low and lowest high fractionals rules also aim for finding integral solutions quickly. Unfortunately, while objective values are monotone, the number of fractional variables in a solution is not, so we cannot guarantee that this latter number will not increase after branching. For this reason the highest low/high fractionals rules are not effective for proving optimality; these two options were implemented for the sake of completeness.

From these options we have used the “highest low objective” rule in our final experiments.

Our third task is to choose one of the children if diving is desired. If no good upper bound is known then we forced diving into the "one-child", i.e., into the child with right hand side value of one. Otherwise we used a standard default in which the child with the lower objective value is chosen.

5.4 User-written functions of the Cut Generator process

Our only job in the Cut Generator process is to find valid inequalities that are not satisfied by the solution to the current LP relaxation. Currently we are able to generate clique, (lifted) odd hole and antihole, and packing and cover odd hole inequalities in the Cut Generator. These are standard types of cuts that can be heuristically separated in a reasonable amount of time. We have also experimented with generating cuts "by hand," that is, we graphically display the current solution and the cuts as they are generated, and try to identify inequalities of types different from those above that are violated by the current solution. If we choose this option, the cut generator process will wait for us until we signal that we can generate no more cuts. We call this option the "human cut generator." The human cut generator is implemented through the graphical user interface (described in Chapter 6) that runs on a separate processor.

A discussion of known valid inequalities for the Set Partitioning Problem can be found in Section 2.3; in this section we give details only of those inequalities that we have implemented.

All the cuts that we generate here can be described in terms of the intersection graph. When generating cuts in the cut generator process we restrict ourselves to the intersection graph corresponding to those variables that have fractional solution values, we call this the *fractional graph*. Although probably more violated inequalities could be found if we extended our search to the intersection graph of the entire problem, that graph would be too large for efficient cut generation.

We decided to parallelize cut generation so all our cut generation routines could be started at the same time (parameters determine which ones are started) and could produce cuts independently of each other. The reason for this is that different kinds of inequalities are effective for different kind of problems; a considerable amount of time could be wasted in the "wrong" cut generator if the cut generation routines are not ordered well (we might not even get to the interesting cut generator before the LP sends the next solution). We spawn these "slave" cut generator processes as soon as the cut generator "master" is started by the tree manager. We have four slaves: two generate cliques with the two different methods described below, one generates (lifted) odd holes, packing and cover odd holes, and the last generates (lifted) odd antiholes. We run the slaves on the same processor as the master cut generator so that we get the advantage of faster message passing and do not take up additional processors.

The cut generator slaves receive some initial information from their master as soon as they are started. Every time a new solution arrives from the LP to the master cut generator, a signal is sent to the slaves to abort work and wait for the new fractional graph. New violated cuts are sent back from the slaves to the master cut generator, which forwards them to the LP process. The slaves keep track of which cuts have been sent back to their master for a given fractional graph, so that the same cut is not sent back twice.

Now we discuss implementational details of the various cut generation routines.

5.4.1 Cliques

Recall from Section 2.3 that the *clique inequality* is

$$\sum_{v \in K} x_v \leq 1,$$

where K is a clique (a complete subgraph) in the intersection graph; that is, the columns corresponding to the variables in the sum are pairwise nonorthogonal. As we have mentioned before, we are looking for violated inequalities of the above form where K is a maximal clique. No polynomial time separation algorithm is known for separating these inequalities, therefore we employ heuristics.

We have implemented two heuristic methods that are frequently described in the literature ([HP93], [Bor97]). The first method (*star clique method*) is enumerative and guarantees to find a violated (maximal) clique inequality (if there exists one) if the enumeration is carried out fully. The second method (*row clique method*), originally due to Hoffman and Padberg ([HP93]), enumerates only those maximal cliques that can be obtained as extensions of row cliques (cliques corresponding to supports of rows in the fractional matrix). A greedy heuristic is substituted for the enumeration in practice if the enumeration would need to be carried out at too many nodes. Note that not all maximal cliques can be obtained as extensions of row cliques. (It is very easy to construct an example using the fact that if a row has an intersecting column with only one 1 in it, its row clique cannot be extended.)

The *star clique method* is based on the fact that any maximal (and also nonmaximal) clique containing a specific node v is a subgraph of the *star of* v (subgraph spanned by v and its neighbors). The algorithm runs until there are no nodes left in the graph. At every step of the algorithm a node is chosen, all maximal cliques are enumerated in its star and then the node is removed from the graph. A clique which is maximal for the remaining graph after some nodes are already eliminated might not be maximal for the original graph. But these maximal cliques (that can be extended on the already deleted nodes) can be disregarded since *all* maximal cliques containing the previously deleted nodes have been found earlier.

An important question is which node to choose next. A minimum degree node is a logical choice since its star is the smallest possible, thus enumeration on the nodes in the star is the fastest. Moreover, degree 0 or 1 nodes can be deleted at once. Other plausible strategies are choosing a maximum degree node (to discover largest maximal cliques early), or choosing a node with a large value in the current fractional solution (hoping to lead to a violated clique).

The row clique method is very similar but not the same as the reduction CLEXT (Section 3.1), where columns extending row cliques in the original formulation are removed from the problem. Here we consider only the fractional problem matrix, whose rows are shorter than those of the original matrix, so a column extending the row clique in the fractional matrix might not extend the row clique in the original matrix. To enumerate all the maximal cliques that are extensions of row cliques, the rows of the problem matrix are taken one by one. A candidate list of nodes that are adjacent to all nodes in the row clique is formed. Then all the maximal cliques are enumerated on the candidate list and are added to the nodes of the row. Any extended maximal clique found this way will be violated (the sum of values in a row is already 1).

The two methods were implemented as two different slave processes. The implementations were straightforward; in both methods we use parameters to determine the size of a subset of nodes above which a greedy maximal clique detection routine is substituted for complete enumeration. The row clique method is definitely faster for larger graphs; its advantage is that the enumerative or greedy clique detection needs to be carried out on a smaller subset of the graphs, and that a violated clique inequality is found as soon as *any* of the row cliques can be extended by at least one node.

We also screen the generated violated inequalities for minimum violation (regulated through a parameter) which allows us to send back inequalities to the CG master selectively.

5.4.2 Lifted odd holes, packing and cover odd holes

The *odd hole inequality* is of the form

$$\sum_{v \in H} x_v \leq \frac{|H|-1}{2},$$

where H is a chordless odd cycle (odd hole) in the intersection graph; that is, columns corresponding to consecutive variables in H (considering the last and first variable in H consecutive) are nonorthogonal, while all the rest of the variable pairs have orthogonal columns.

Instead of weighting the nodes of the graph, we can assign costs to the edges and give an alternative formulation of the odd hole inequality. Let the cost of edge vw be $c_{vw} = 1 - x_v - x_w$, then the sum of the costs on edges along the odd hole is

$$\sum_{vw\ edge\ in\ H} c_{vw} = \sum_{vw\ edge\ in\ H} (1 - x_v - x_w) = |H| - 2\sum_{v \in H} x_v,$$

thus the odd hole inequality can be written as

$$\sum_{vw\ edge\ in\ H} c_{vw} \geq 1.$$

Note that $0 \leq c_{vw} \leq 1$.

A polynomial time algorithm for separating violated odd hole inequalities (GLS-algorithm) was originally developed by Grötschel, Lovász and Schrijver [GLS88]. A bipartite graph is generated where the nodes of the intersection graph are repeated on both sides of the partition and two nodes on different sides of the partition are adjacent if and only if the corresponding nodes in the original graph were adjacent. Assign the costs defined above to the edges of the bipartite graph. A path between a node on one side of the partition and its duplicate on the other side corresponds to an odd cycle in the original graph (node repetition is possible). Since the weights are nonnegative, a path of weight less than 1 will correspond to an odd cycle that contains a violated odd hole. Also, a violated odd hole provides a path of weight less than 1 between any of its nodes and their duplicates.

Thus computing the *shortest* path between all nodes and their duplicates and taking the cheapest of these will either provide an odd cycle of weight less than 1 or prove that no violated odd hole exists. Computing the shortest path between two nodes of a graph can be done in polynomial time (Dijkstra's algorithm can be used since the edge weights are nonnegative), so separating for violated odd hole inequalities is polynomial.

Although the algorithm outlined above is very appealing, Hoffman and Padberg ([HP93]) consider it impractical since according to their experiments after a few iterations of cut generation and resolving the LP the algorithm usually comes up only with length 3 violated odd holes (triangles) that are also found by the violated clique detection heuristics. Longer but not necessarily violated odd holes are also very useful since they can be lifted to try to produce lifted violated odd hole inequalities (we show how to do this later in this section). On the other hand, Borndörfer ([Bor97]) and Nemhauser and Sigismondi ([NS92]) apply the GLS algorithm and do not report any shortcomings. When we implemented our odd hole separation procedure we were not aware of any earlier efficient implementation of the GLS algorithm in this context and Hoffman and Padberg's arguments were very convincing, so we decided to follow their advice and implemented a heuristic separation algorithm similar to theirs.

Our algorithm for finding (lifted) odd holes is best described in terms of the fractional intersection graph. A *level graph* is a breadth first arrangement of the nodes of the graph, rooted at a specific node. The root node is on level zero, its neighbors are on level one, its neighbors' neighbors are on level two, etc. Because of the breadth first enumeration of the nodes, adjacent nodes cannot be more than one level apart. Notice that if two adjacent nodes on the same level both have a path up to the root so that nodes on one path are not neighbors of nodes on the other path, the two paths and the edge between the two nodes form an odd hole. The costs defined earlier are assigned to the edges of the level graph.

The algorithm enumerates the levels one by one (starting with level two since triangles are not desired), and for every adjacent node pair on level l looks for the cheapest (with respect to the edge weights) possible length l path from the nodes up to the root. First a cheapest path that "steps up" one level at a time is searched for from one of the nodes to the root, and if such path is found, the neighbors of the nodes along the path are "blocked out" and a similar path is sought starting from the second node. If the second path exists as well then the costs of the paths are combined with the cost of the edge between the two nodes. If the cost of this odd hole is less than 1, its violated inequality is sent back to the CG master. Otherwise, we try to strengthen the inequality by lifting in some more variables from the fractional graph, as we will describe below. Then this procedure is repeated searching for a path from the second node first.

Usually we build more than one level graph, choosing the roots randomly with a probability inversely proportional to the degree of the nodes. About 5% but no more than 50 of the nodes are chosen. As in the violated clique identification algorithms, a parameter determines the minimum amount by which an odd hole must be violated in order that it be returned to the CG master.

As discussed earlier (Section 2.3), to compute the coefficient of a variable currently not in the inequality we subtract from the right hand side of the inequality the size of the largest weighted independent set in the subgraph that corresponds to the variables already in the inequality and the one to be added.

When lifting odd holes we call the variables that are added to the inequality *hubs*. Nodes whose coefficients are yet to be computed are the *hub candidates*. The order in which the coefficients of the hub candidates are computed (lifting order) affects the outcome. Our goal is to produce an inequality which is maximally violated, therefore at each step we try to choose the best hub candidate. To accomplish this, we iteratively compute the would-be coefficients for all the hub candidates and select the one with maximal increase in the left hand side (that is, the coefficient times the value of the variable in the current solution) until there are no more hub candidates.

Computing the lifting coefficient is the heart of this lifting algorithm. Note that there might already be some hubs that were lifted in earlier, so here we give a general algorithm for computing the coefficient of a hub candidate for a *lifted* odd hole. We consider the submatrix of the lifted odd hole and the hub candidate. We assume that the hub candidate has been chosen, so its neighbors in the subgraph can be deleted right away. Now assume that a maximal set of independent hubs already in the inequality is chosen; removing their neighbors from the odd hole leaves us with a collection of path segments. The size of an independent set for a path is simply half of the length of the path, rounded up (choose every other node). Then the value of a maximum weighted independent set given a maximal set of independent hubs is the sum of the maximum independent set sizes for the paths and the weighted value of the hubs. We recursively enumerate all weighted maximal independent sets on the hubs and compute the largest of the above values. The coefficient of the hub candidate is the difference of the right hand side of the inequality and the computed (largest) value. Note that the coefficient is zero if the computed value matches the right hand side, which is the case, for instance, if the hub candidate is not adjacent to any of the nodes in the odd hole.

Enumerating all subsets of the hubs for each hub candidate at each iteration may seem computationally expensive, but according to our experience the time is reasonable when we utilize the following observations.

- Nodes with at most two neighbors in the odd hole will have a coefficient zero, so these nodes need not be included into the list of hub candidates.
- As soon as the value of the weighted independent set reaches the value of the right hand side for a given set of hubs, the enumeration can be aborted since the coefficient of the hub candidate will be zero.
- Since the would-be coefficients of hub candidates cannot increase during sequential lifting, hub candidates with zero would-be coefficients can be removed from the candidate list.

Thus our violated odd hole detection algorithm can produce odd holes as well as lifted odd holes; these inequalities will be further lifted in the LP process using the same method to compute the lifting coefficients as described here. Our algorithm usually finds many more violated lifted odd holes than plain odd holes.

To find packing odd holes we start by locating odd holes with the help of the level graph exactly as above. Once an odd hole (violated or not) is found, a set of rows is chosen, one for each edge in the odd hole (Section 2.3). If more than one row's support contains both endpoints of an edge then the longest such row is selected (the goal is to include as many coefficients as possible into the final packing odd hole inequality). The generated packing odd hole inequalities are tested for violation and for those sufficiently violated the cut (the corresponding set of rows) is sent back to the Cut Generator master for forwarding to the LP. The LP process derives the packing odd hole inequality again on every variable in the current formulation (not only on those at fractional level). This is not necessary, but it is computationally inexpensive and usually results in a cut with much larger support.

The cover odd holes are generated very similarly. Now we look for odd holes with the sum of the fractional values on the nodes as small as possible (i.e., with large total edge costs). So we aim for the most expensive path to the root of the level graph instead of finding the cheapest one. Also, we choose the shortest rows for the edges in the odd hole trying to keep the left hand side of the inequality small. Observe that the cover odd hole inequality must be derived from the rows of the entire current matrix to be valid. However, its validity can be tested using the fractional matrix only, since all variables not in the fractional matrix are at level zero in the current solution. If a cover odd hole inequality proves to be violated here, the cut (corresponding rows) is sent back to the CG master and from there to the LP where the cut will be reconstructed for the entire formulation (this time it is necessary to do so).

5.4.3 Odd antiholes and lifted odd antiholes

We define the *odd antihole* inequality as

$$\sum_{v\in\bar{H}} x_v \leq 2,$$

where $\bar{H}$ is an odd antihole in the intersection graph (the edge-complement of an odd hole).

Separating heuristically for violated odd antiholes is straightforward as we can take the complement of the intersection graph and use any violated odd hole detection routine (with the right hand side fixed to 2). In our algorithm we used the same level-graph approach as for odd holes. Again, if the algorithm detects a violated odd antihole it is returned to the cut generator master at once, while non-violated odd antiholes are first lifted. The lifting procedure is also similar to the traditional sequential lifting of odd holes; first we create the set of hub candidates and then repeatedly compute the would-be coefficients for the candidates and choose the one that increases the left hand side the most.

Computing the lifting coefficients for the hub candidates is considerably easier for (lifted) odd antiholes than for (lifted) odd holes since here the coefficients cannot be larger than 2. The coefficient of a hub candidate is 2 if the hub candidate is adjacent to all nodes already in the (lifted) odd antihole, zero if the hub candidate has a non-neighbor with coefficient 2 or two nonadjacent non-neighbors, and 1 otherwise.

The lifting procedure for odd antiholes can be made more efficient by observing that

- Nodes with at most $(|\bar{H}|-1)/2$ neighbors on the odd antihole will have a coefficient zero, thus they need not be included in the hub candidate list.

- As for odd holes, hub candidates with zero would-be coefficients need not be considered further.

5.5 Computational Results

In our final runs we used the default settings of the framework except for the following. We set the time limit to two hours (7200 seconds) for the Branch-and-Cut procedure (note that this is a limit for the Tree Manager; reading in the problem instance and preprocessing and heuristics in the Master is not counted into this time). We used the upper bound obtained by our Feasible Solution Heuristic (Section 4.4) whenever available. The GUI was disabled for the runs on the SP. The number of LP – Cut Generator pairs was set to 1, 2, 4, 8 and 16. We used only one Cut Pool process, and experimented with both pure Branch-and-Bound and Branch-and-Cut with generating only clique inequalities or turning on all the cut generators we had (except for the sequential lifting of odd holes because this was much more time consuming than simultaneous lifting). The Tree Manager chose the node with the lowest presolved LP value from the candidate list to be processed next.

Threshold values were set to .33 and .99 for tailing off based on objective values and integrality gaps, respectively; the length of the history was limited to between 5 and 10 steps. 10 to 30 violated inequalities were added to the formulation in each iteration. The LP relaxations at the would-be children of branching candidates were presolved up to 500 dual simplex iterations (which meant that they were solved to optimality for most of the problems). The "highest-low objective" rule was used for branching object selection.

The number of cuts in the cut pool is limited to a few thousand. For a given LP solution only those cuts were checked which were originally generated at a higher level of the search tree than the solution and were recently found violated.

The parameters on the user side were set the following way. All the LPs (the initial LP relaxation or those during the feasible solution heuristic) were solved using CPLEX's barrier method with dual crossover. In the LP process logical fixing was always attempted after solution of the first LP in a node which is the "one-child" of its parent (that is, the node was obtained by setting a variable or the right hand side of a branching cut to 1), or if 2% of the variables were recently fixed to zero. The feasible solution heuristic was not invoked at all for problems with a very low integrality gap (less than 2%). Otherwise it was invoked with a probability of .5 at the top of the one-children and it was also invoked if 20% of the variables had been fixed since the most recent application of the heuristic.

The number of branching objects of each type (follow-on, threshold, slack cut and variable) varied depending on the difficulty of solving LP relaxations for the given problem. The lower and upper thresholds for selecting follow-on candidates were set to .60 and .80, respectively. The threshold for threshold branching was set to .65. We experimented with branching variables chosen both with the "close to half and expensive" and "close to one and cheap" rules.

As indicated above, all but sequentially lifted odd hole inequalities were generated. The minimum violation was set to 0.01 for all cut types.

Greedy clique detection was substituted for enumeration in the star clique and row clique methods when the number of nodes in the set on which the clique was to be enumerated exceeded 16. The node with the minimum degree was chosen to be the next in the star clique method. The number of hub candidates for sequential lifting of odd antiholes was limited to 100. Up to 10 levels (depending on the size and density of the fractional graph) of the level graph were investigated (which means odd holes of length up to 21 could be found); the algorithm continues with the next level or even next level graph when a violated inequality is obtained from the current level graph.

There is no "perfect" setting for the parameters that would work well with all problem types. We have spent considerable computational effort in fine-tuning some of the parameters for each set of test problems. In the next sections we will discuss our experiments for all four sets in detail.

Our parallel runs were carried out on the thin nodes of the SP (the computing environment is described in Section A.1). We reserved one processor for the Master, Tree Manager and Cut Pool processes. LP-CG pairs were placed on the same processor; we used 1, 2, 4, 8 or 16 of these. Thus the total number of processors used was 2, 3, 5, 9 or 17. The number of search tree nodes processed and the total execution time are the two main indicators of a B&C algorithm's performance. Pure B&B runs were carried out with one LP-CG pair only since communication time is not negligible compared to one LP solution time for the problems in our test bed. Parallel runs (more than one LP-CG pair) were carried out for problems requiring at least 100 nodes during B&C.

Intuitively, an algorithm is parallel efficient if doubling the number of processors used cuts the execution time in half. To quantify this concept, the *parallel speedup* (with p processors) is defined as

$$s(p) = \frac{\text{running time with 1 processor}}{p \cdot \text{parallel running time}}.$$

If this ratio is below 1 then the algorithm on p processors uses more resources than on one processor. *Linear speedup* is defined as a speedup with ratio 1; *superlinear speedup* as a speedup with ratio greater than 1. Note that it is not impossible to achieve superlinear speedup in an asynchronous parallel application when the order of certain events influences the flow of the algorithm. For instance, in a parallel B&C algorithm communication delay can cause search tree nodes to arrive at the Tree Manager in different order, resulting in completely different search trees.

We follow [Ral95] and [Lad96] and consider the number of LP-CG pairs only when measuring speedup (the time spent in the Tree Manager and in the Cut Pool is considered a constant overhead). Moreover, we define two speedup ratios, one based on the execution time and another based on the number of processed search tree nodes:

$$s_1(p) = \frac{\text{running time with 1 LP} - \text{CG pair}}{p \cdot \text{running time with } p \text{ LP} - \text{CG pairs}},$$

and

$$s_2(p) = \frac{\text{number of search tree nodes processed with 1 LP} - \text{CG pair}}{\text{number of search tree nodes processed with } p \text{ LP} - \text{CG pairs}}.$$

Note that the speedup based on the number of processed search tree nodes is linear (superlinear) if this number stays the same (decreases).

Runs with more than one LP-CG pairs were repeated three times and the average of the results is reported. This was done to compensate for the difference in the results due to communication delay.

5.5.1 Set 1 problems

Branch-and-Cut optimization was carried out for the 10 problems not solved to optimality by our Feasible Solution Heuristic (Section 4.4). As we observed earlier, the optimal solution was found but not proved for 5 of these problems, and the optimality gap was below 2% for the other 5. As it turns out 7 of the 10 problems are trivial for Branch-and-Cut, one problem (`nw04`) requires a little more computational effort, and only two problems (`aa01` and `aa04`) can be considered moderately difficult.

In our final experiments we used the default settings described above. The framework added at most 10 cuts in each iteration, the tailing off history was set to 5 iterations. The deepest level investigated in the level graph was limited only in the two hard problems (to level 2, thus length 5 odd holes) since the level-graph approach for odd hole detection is not efficient for dense fractional graphs such as those corresponding to these problems.

We present here 10 basic experiments with one LP-CG pair for each of the 10 problems. Each problem was run with pure B&B, B&C with separating clique inequalities only (both star and row cliques), and B&C with separating clique, packing and cover odd hole, and sequentially lifted odd antihole inequalities. In each of these three cases either only branching variables were selected or all types of branching objects (this second does not apply for B&B). Finally, candidate branching variables were selected with either the mixed or the "close to one-half and expensive" rule. The number of branching candidates totaled 6.

Tables 5.1 through 5.10 (one table for each problem) summarize the results of these experiments. The first three columns indicate the settings for the experiment (as explained above). This is followed by the number of processed / created search tree nodes (nodes that were created but not processed are those fathomed during branching) and the depth of the search tree; the worst lower bound on the nodes not yet processed and the best feasible solution value found (if optimality is not proved); the time when the best solution was first found (in case the upper bound is the optimal value, this space is left empty) and the overall execution time (as reported by the Tree Manger). For example, in Table 5.1 the third line summarizes the results of a Branch-and-Cut experiment when only clique inequalities were generated and only variables (chosen with the "close to one-half and expensive" rule) were considered as branching candidates. 97 search tree nodes were created, out of which 53 were processed (the rest were fathomed during branching), the depth of the search tree was 13. The problem was solved to optimality in 601.09 seconds; the optimal solution was found after 378.03 seconds. On the other hand, using the same settings but selecting the branching variables with the mixed rule, the algorithm timed out. 3069 search tree nodes were created, 1566 of them were processed (the rest were fathomed or waiting to be processed), the depth of the search tree was 41. The value of the best feasible solution is 56167 (found after 6300.19 seconds), the lower bound (the lowest presolved LP value at the unprocessed nodes) is 55809.21.

Table 5.1: Basic B&C experiments for `aa01`

aa01			lb: 55535.43		OPT: 56137			ub: 56172
			search tree		worst	best	time	
			nodes	depth	lb	soln	to best	total
B&B	var	1/2	181/345	17			1102.24	1102.56
		mix	3607/7199	43	55743.21	56162	6583.18	7208.01
B&C clique	var	1/2	53/97	13			378.03	601.09
		mix	1566/3069	41	55809.21	56167	6300.19	7205.80
	var cut	1/2	82/147	14			629.24	794.59
		mix	884/1699	27	55988.75	56137	6782.16	7205.96
B&C all	var	1/2	59/107	12			508.68	877.06
		mix	1416/2753	37	55827.43	56138	1719.94	7205.28
	var cut	1/2	79/135	11			628.04	1053.71
		mix	664/1181	22			3698.75	5824.87

Table 5.2: Basic B&C experiments for `aa04`

aa04			lb: 25877.60		OPT: 26374			ub: 26680
			search tree		worst	best	time	
			nodes	depth	lb	soln	to best	total
B&B	var	1/2	574/1143	28			2797.97	2976.77
		mix	3834/7649	43	26133.41	26456	7009.96	7201.60
B&C clique	var	1/2	204/365	19			1682.30	1686.31
		mix	2181/4311	35	26160.20	26492	200.89	7209.74
	var cut	1/2	575/1067	22			3575.54	3908.01
		mix	1460/2875	32	26229.68	26402	1183.34	7207.16
B&C all	var	1/2	191/343	17			1917.62	2222.81
		mix	1409/2757	33	26205.38	26451	5842.55	7205.63
	var cut	1/2	311/569	20			2892.11	2983.92
		mix	827/1567	23	26283.54	26375	2416.45	7208.77

We can see that six problems (aa03, aa06, nw17, nw36, kl02 and us01) solve in the root node when all cut generation is enabled, even though tailing off is checked (which can force early branching). In aa05 the optimal solution is also found during the first branching, but one of the children cannot be fathomed right away. When tailing off is disabled for this problem, no cuts are found after 16 iterations in the root thus branching is forced. Again, the optimal solution is found during strong branching.

We can observe that for these problems the mixed strategy (which always defaults to choosing a variable near one) does not work well. It is especially disastrous for aa01, aa04 and kl02. Another observation is that pure B&B is very often more effective than B&C. This can be viewed as further evidence that the problems in this set are relatively easy.

An interesting fact about nw04 is that when solving the problem with pure B&B one of the children is always fathomed during branching, so the search tree is a chain (the number of nodes created is twice the number of nodes processed).

In comparing our results (B&C with all cut generation and only variable branching) with those of Hoffman and Padberg ([HP93]) and Borndörfer ([Bor97]), we note that we solve all the Set 1 problems in no more search tree nodes than Hoffman and Padberg require. Also, for the three problems they consider difficult (aa01, aa04 and nw04) our running times represent at least an order of magnitude improvement. Compared to Borndörfer's results ([Bor97], default strategy), our algorithm processed slightly more search tree nodes than his for the two hard problems but not more for the remaining problems. Our running times are comparable to his

Table 5.3: Basic B&C experiments for `nw04`

nw04			lb: 16310.66	OPT: 16862		ub: 17158
			search tree		time	
			nodes	depth	to opt	total
B&B	var	1/2	38 / 77	38	0.44	15.04
		mix	64 / 127	48	0.44	13.67
B&C clique	var	1/2	36 / 71	25	4.51	40.69
		mix	69 / 127	28	3.17	77.92
	var cut	1/2	35 / 65	27	7.88	54.66
		mix	70 / 117	25	6.47	95.59
B&C all	var	1/2	35 / 65	26	2.35	52.74
		mix	42 / 77	28	3.53	55.61
	var cut	1/2	38 / 67	26	25.69	101.24
		mix	106 / 143	28	3.24	313.73

Table 5.4: Basic B&C experiments for `aa05`

aa05			lb: 62860.50	OPT: 53839		ub: 53904
			search tree		time	
			nodes	depth	to opt	total
B&B	var	1/2	4 / 7	3	2.26	2.26
		mix	46 / 87	14	2.47	13.17
B&C clique	var	1/2	5 / 9	3	3.08	3.34
		mix	6 / 11	5	3.14	3.22
	var cut	1/2	5 / 7	3	2.21	3.16
		mix	12 / 21	5	2.49	6.56
B&C all	var	1/2	2 / 3	1	9.96	11.91
		mix	5 / 9	4	18.30	18.51
	var cut	1/2	2 / 5	2	10.23	16.59
		mix	2 / 5	2	10.21	15.12

Table 5.5: Basic B&C experiments for `aa03`

aa03			lb: 75323.36	OPT: 49649	ub: 49649	
			search tree		time	
			nodes	depth	to opt	total
B&B	var	1/2	1 / 3	1		0.15
		mix	2 / 3	1		0.18
B&C clique	var	1/2	1 / 3	1		0.31
		mix	1 / 3	1		0.31
	var cut	1/2	1 / 3	1		0.30
		mix	1 / 3	1		0.32
B&C all	var	1/2	1 / 1	0		0.51
		mix	1 / 1	0		0.40
	var cut	1/2	1 / 1	0		0.49
		mix	1 / 1	0		0.53

Table 5.6: Basic B&C experiments for `aa06`

aa06			lb: 30314.18	OPT: 27040	ub: 27040	
			search tree		time	
			nodes	depth	to opt	total
B&B	var	1/2	6 / 9	3		2.25
		mix	15 / 27	8		4.90
B&C clique	var	1/2	2 / 5	2		1.59
		mix	1 / 3	1		1.31
	var cut	1/2	2 / 3	1		1.29
		mix	2 / 5	2		1.87
B&C all	var	1/2	1 / 1	0		3.28
		mix	1 / 1	0		3.39
	var cut	1/2	1 / 1	0		3.34
		mix	1 / 1	0		3.45

Table 5.7: Basic B&C experiments for `kl02`

kl02			lb: 224.25	OPT: 219		ub: 219
			search tree		time	
			nodes	depth	to opt	total
B&B	var	1/2	8 / 17	8		1.33
		mix	198 / 385	21		10.82
B&C clique	var	1/2	3 / 7	3		1.02
		mix	28 / 57	8		3.47
	var cut	1/2	315 / 621	26		40.36
		mix	1342 / 2635	23		149.34
B&C all	var	1/2	1 / 1	0		0.39
		mix	1 / 1	0		0.40
	var cut	1/2	1 / 1	0		0.40
		mix	1 / 1	0		0.39

Table 5.8: Basic B&C experiments for `nw17`

nw17			lb: 13764.00	OPT: 11115		ub: 11115
			search tree		time	
			nodes	depth	to opt	total
B&B	var	1/2	2 / 3	1		0.22
		mix	2 / 3	1		0.10
B&C clique	var	1/2	2 / 3	1		0.42
		mix	1 / 3	1		0.35
	var cut	1/2	2 / 3	1		0.41
		mix	1 / 3	1		0.36
B&C all	var	1/2	1 / 1	0		0.41
		mix	1 / 1	0		0.40
	var cut	1/2	1 / 1	0		0.45
		mix	1 / 1	0		0.41

Table 5.9: Basic B&C experiments for `nw36`

nw36			lb: 7762.00	OPT: 7314	ub: 7314	
			search tree		time	
			nodes	depth	to opt	total
B&B	var	1/2	4 / 9	4		0.12
		mix	4 / 9	4		0.15
B&C clique	var	1/2	2 / 3	1		0.36
		mix	2 / 3	1		0.35
	var cut	1/2	3 / 5	2		0.39
		mix	2 / 5	2		0.40
B&C all	var	1/2	1 / 1	0		0.44
		mix	1 / 1	0		0.46
	var cut	1/2	1 / 1	0		0.42
		mix	1 / 1	0		0.44

Table 5.10: Basic B&C experiments for `us01`

us01			lb: 9963.06	OPT: 10036	ub: 10101	
			search tree		time	
			nodes	depth	to opt	total
B&B	var	1/2	3 / 7	3	0.33	0.44
		mix	7 / 13	5	0.32	0.59
B&C clique	var	1/2	2 / 5	2	3.18	3.23
		mix	4 / 7	3	2.12	2.28
	var cut	1/2	2 / 5	2	2.15	2.53
		mix	6 / 11	5	1.58	2.83
B&C all	var	1/2	1 / 3	1	6.16	6.17
		mix	1 / 1	0	5.95	5.96
	var cut	1/2	1 / 1	0	4.21	4.22
		mix	1 / 1	0	3.75	3.76

Table 5.11: Parallel runs for `aa01`

aa01	lb: 55535.43		OPT: 56137	ub: 56172		
	search tree		time		$s_1(p)$	$s_2(p)$
	nodes	depth	to opt	total		
1	51.33 / 91.00	11.67	384.69	622.65	1.00	1.00
2	68.33 / 115.67	12.67	307.58	402.11	0.77	0.75
4	59.33 / 91.67	11.67	155.92	181.06	0.86	0.87
8	52.00 / 89.00	11.33	97.03	177.20	0.44	0.99

Table 5.12: Parallel runs for `aa04`

aa04	lb: 25877.60		OPT: 26374	ub: 26680		
	search tree		time		$s_1(p)$	$s_2(p)$
	nodes	depth	to opt	total		
1	282.67 / 537.00	24.67	2293.19	2404.63	1.00	1.00
2	267.67 / 506.33	21.67	1057.40	1111.41	1.08	1.06
4	187.67 / 334.33	15.67	318.39	350.18	1.72	1.51
8	233.67 / 411.00	17.33	229.90	239.90	1.25	1.21

on all but the two hardest problems. There are two reasons for this difference. First of all, [Bor97] used an internal CPLEX routine for strong branching which we strongly suspect is much more efficient computationally than the framework's strong branching. Considering that on these problems our algorithm spends more than two-thirds of its running time in strong branching, the internal CPLEX routine gives an enormous advantage. Second, they used the GLS algorithm for finding odd holes which seems to be more efficient for dense fractional graphs than the level graph approach.

We also ran the two hard problems with 2, 4 and 8 LP-CG pairs (using B&C with all the cuts and variable branching only). Tables 5.11 and 5.12 present the results (three runs averaged). `aa04` shows a superlinear speedup both in terms of running time and the number of processed search tree nodes. `aa01` shows close to linear speedup in terms of nodes processed, but it exhibits the law of diminishing returns when using 8 processors: there are so few search tree nodes that adding more processors didn't help, since for these processors there was nothing to work on.

5.5.2 Set 2 problems

We have experimented with these problems, but the results were not encouraging. As we have already seen in Table 4.4 the LP relaxations for these problems are extremely difficult. They were very hard both for reoptimization and strong branching. For this reason we selected only three variables as candidates for strong branching. In the first of two sets of runs we have selected these variables with the "close to one-half and expensive" rule and with the mixed rule in the second. Odd hole generation was disabled as well.

These problems proved to be so hard that with four LP-CG pairs in a two hour time frame we were able to process only approximately 35 nodes on the average and found a feasible solution only once (this solution was found by the feasible solution heuristic). Tables 5.13 and 5.14 show these results. We are not aware of any published results for these problems.

Table 5.13: Basic B&C experiments for Set 2 (branching on “close to one-half”)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
0321.4	8 / 9	4	35875.86	- / -		7224.34
0331.3	108 / 203	40	28994.61	34253.10 / -	3045.40	7217.91
0331.4	51 / 95	20	29968.41	- / -		7215.58
0341.3	86 / 157	22	31383.66	- / -		7222.75
0341.4	75 / 139	28	34443.64	- / -		7215.07
0351.3	11 / 15	4	35239.85	- / -		7221.17
0351.4	9 / 11	4	34551.91	- / -		7223.03

Table 5.14: Basic B&C experiments for Set 2 (mixed branching variable selection)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
0321.4	8 / 9	4	35879.24	- / -		7223.69
0331.3	26 / 45	8	28488.68	- / -		7223.73
0331.4	26 / 45	8	29778.25	- / -		7223.24
0341.3	34 / 61	10	31058.49	- / -		7224.87
0341.4	39 / 71	11	34320.70	- / -		7219.07
0351.3	9 / 11	4	35129.98	- / -		7223.07
0351.4	8 / 9	4	34539.79	- / -		7225.61

5.5.3 Set 3 problems

These problems can be divided into two groups, the `v*` and `t*` problems (they were generated at different stages of the modeling process).

The `v04` and `v16` problems are extremely degenerate; there are many feasible solutions around the optimal value. This was already apparent when we were able to find near-optimal feasible solutions with our heuristic. The LP relaxations are relatively easy to (re-)solve, both after adding constraints and during strong branching. Hence we were able to consider a wider selection of branching candidates, altogether 18. The fractional graphs are relatively small and sparse, thus we were able to use our packing and cover odd hole generators. Preliminary testing showed that branching on variables "close to one-half and expensive" is not effective, the mixed strategy was used instead. Two experiments were carried out for each problem, one with branching on variables only, the other with branching on both variables and cuts.

The `v04` problems are very easy except for `v0416`. Tables 5.15 and 5.16 present the results of the two experiments for these problems, using only one LP-CG pair. We were able to solve all problems to optimality in both cases. For these problems branching on cuts seems to give an advantage. We experimented with multiple LP-CG pairs for `v0416`. Tables 5.17 and 5.18 show that this problem scales very well for multiple LP-CG pairs. Also, the effectiveness of branching on cuts is even more pronounced.

The `v16` problems are much larger than the `v04` problems and although the LP relaxations are still not too difficult, the size of the search tree explodes because

Table 5.15: Basic B&C experiments for `v04` (branching on vars)

problem	search tree		OPT	initial	time	
	nodes	depth		ub	to opt	total
v0415	11 / 19	5	2429415	2435833	5.16	7.02
v0416	869 / 1387	42	2725602	2736885	10.13	506.56
v0417	10 / 15	3	2611518	2622525	41.06	61.41
v0418	8 / 15	6	2845425	2855469	4.66	9.04
v0419	1 / 1	0	2590326	2598124	1.08	1.09
v0420	1 / 1	0	1696889	1703734	0.61	0.61
v0421	1 / 1	0	1853951	1858977	0.89	0.90

Table 5.16: Basic B&C experiments for `v04` (branching on vars/cuts)

problem	search tree		OPT	initial	time	
	nodes	depth		ub	to opt	total
v0415	12 / 17	4	2429415	2435833	4.70	6.61
v0416	582 / 801	23	2725602	2736885	7.22	338.65
v0417	11 / 15	5	2611518	2622525	22.44	53.88
v0418	3 / 5	2	2845425	2855469	3.43	3.52
v0419	1 / 1	0	2590326	2598124	0.58	0.59
v0420	1 / 1	0	1696889	1703734	0.56	0.57
v0421	1 / 1	0	1853951	1858977	0.45	0.47

Table 5.17: Parallel runs for `v0416` (branching on vars)

v0416	lb: 2715490.66		OPT: 2725602		ub: 2736885	
	search tree		time		$s_1(p)$	$s_2(p)$
	nodes	depth	to opt	total		
1	1268.33 / 2097.00	39.67	11.75	692.89	1.00	1.00
2	1277.00 / 2141.00	39.00	9.99	347.76	1.00	0.99
4	1572.67 / 2580.33	38.67	9.02	180.91	0.96	0.81
8	1258.33 / 2081.67	38.00	8.40	72.27	1.20	1.01
16	1569.67 / 2582.33	36.67	6.57	49.55	0.87	0.81

Table 5.18: Parallel runs for `v0416` (branching on vars/cuts)

v0416	lb: 2715490.66		OPT: 2725602		ub: 2736885	
	search tree		time		$s_1(p)$	$s_2(p)$
	nodes	depth	to opt	total		
1	1129.33 / 1617.00	29.00	3.80	514.35	1.00	1.00
2	910.67 / 1279.67	27.33	2.10	150.81	1.71	1.24
4	1090.00 / 1541.67	31.00	2.85	106.81	1.20	1.04
8	1011.67 / 1427.00	31.33	3.86	58.70	1.10	1.12
16	800.00 / 1097.00	26.67	4.12	27.90	1.15	1.41

of the degeneracy. We have experienced that the the lower bound did not increase through several levels in the search tree. Therefore we used four LP-CG pairs for all but the last two these problems which are trivial, Tables 5.19 and 5.20 contain the results of our experiments. Having multiple LP-CG pairs also compensates for the computational inefficiency of the strong branching interface to CPLEX. (Even with four pairs we have processed significantly fewer search tree nodes than Borndörfer et al. [BGKK97].) We have also experimented with 16 LP-CG pairs for the five difficult problems in this group (branching on variables only). Having more computing power has generally improved both the upper and lower bounds but the search tree is still far from being enumerated in four out of the five cases. Surprisingly, `v1620` was solved to optimality in 375.04 seconds (see Table 5.21).

The `t*` problems are much more difficult than the previous sets. The LP relaxations are hard, and feasible solutions are scarce (we were able to find feasible solutions with the heuristic only for three out of the 13 problems). The fractional graphs are dense, thus generating odd holes was prohibitively expensive. We restricted the number of branching candidates to 9. In contrast to the `v*` problems, the "close to one-half and expensive" rule selected better branching variables. Again, we have experimented with branching on variables only and branching both on variables and cuts. Here we again used four LP-CG pairs. Tables 5.22 through 5.25 show the results of our experiments.

Although we have found feasible solutions for all the problems (always with the heuristic), the integrality gap is still over 10%.

Our results compare to those of Borndörfer et al. [BGKK97] reasonably well.

Table 5.19: Basic B&C experiments for `v16` (branching on vars)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
v1616	603 / 1199	101	1006156.06	1006460 / 1018536	837.37	7208.33
v1617	729 / 1445	105	1102258.78	1102637 / 1115503	966.15	7207.93
v1618	411 / 819	159	1152827.05	1154324 / 1166107	4725.44	7206.61
v1619	329 / 655	291	1155357.21	1157078 / 1168481	6265.43	7206.88
v1620	280 / 489	82	1140381.91	1140604 / 1152624	5505.25	7207.63
v1621	3 / 3	1	-	825563 / 834602	4.22	4.56
v1622	2 / 3	1	-	793445 / 800572	2.01	2.65

Table 5.20: Basic B&C experiments for `v16` (branching on vars/cuts)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
v1616	538 / 1051	123	1006232.90	1006460 / 1018536	315.01	7207.54
v1617	709 / 1409	68	1102220.26	1102586 / 1115503	539.60	7209.55
v1618	255 / 505	69	1152793.26	1154968 / 1166107	6819.14	7208.10
v1619	304 / 581	35	1155777.33	1156368 / 1168481	3412.55	7208.87
v1620	264 / 457	31	1140425.22	1140604 / 1152624	1485.61	7206.95
v1621	2 / 3	1	-	825563 / 834602	3.85	3.86
v1622	1 / 3	1	-	793445 / 800572	1.70	1.78

Table 5.21: Basic B&C experiments for `v16` with 16 LP-CG pairs

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
v1616	10470 / 20747	203	1006311.49	1006460 / 1018536	143.91	7255.96
v1617	13451 / 26711	107	1102357.00	1102586 / 1115503	1244.02	7250.27
v1618	6881 / 13653	175	1152999.94	1154018 / 1166107	3251.42	7240.58
v1619	6068 / 12051	251	1155866.81	1156557 / 1168481	7101.20	7266.00
v1620	684 / 871	43		1140604 / 1152624	151.40	375.04

Table 5.22: Basic B&C experiments for `t04` (branching on vars)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
t0415	307 / 589	68	5185684.06	5570767 / -	1209.68	7216.27
t0416	276 / 537	52	5892208.64	6093843 / -	336.50	7224.44
t0417	267 / 463	24	5688062.66	5951357 / -	5252.42	7224.32
t0418	199 / 359	29	6195440.35	6442906 / -	3876.42	7217.04
t0419	283 / 541	49	5714748.81	5910913 / -	716.16	7217.70
t0420	865 / 1681	45	4055025.59	4153696 / -	587.04	7223.03
t0421	875 / 1677	43	4126129.27	4290809 / -	1269.66	7219.05

Table 5.23: Basic B&C experiments for `t04` (branching on vars/cuts)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
t0415	382 / 637	19	5196761.22	- / -	-	7213.70
t0416	327 / 635	32	5897503.96	6088264 / -	503.70	7213.75
t0417	301 / 509	20	5690561.30	- / -	-	7223.29
t0418	220 / 383	17	6206432.44	- / -	-	7212.79
t0419	332 / 625	50	5719985.86	6022626 / -	2176.37	7221.35
t0420	1083 / 1985	28	4049232.06	- / -	-	7217.78
t0421	1093 / 2143	42	4122723.76	4290809 / -	321.88	7222.79

Table 5.24: Basic B&C experiments for `t17` (branching on vars)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
t1716	297 / 585	63	122492.51	168856 / -	4495.32	7225.10
t1717	122 / 237	40	135288.55	181375 / 210489	6738.36	7223.22
t1718	182 / 357	66	126847.64	172992 / 204086	6046.42	7217.09
t1719	128 / 249	37	139327.63	187717 / -	6991.11	7218.59
t1720	127 / 247	37	126982.53	179018 / 200679	5385.00	7218.83
t1721	801 / 1585	83	104821.47	128053 / -	4600.99	7224.77

Table 5.25: Basic B&C experiments for `t17` (branching on vars/cuts)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
t1716	326 / 643	70	122364.28	173692 / -	848.32	7222.57
t1717	130 / 253	45	135263.49	203840 / 210489	2428.68	7223.57
t1718	186 / 365	63	126808.38	163860 / 204086	6962.04	7217.19
t1719	172 / 337	53	139314.47	187222 / -	6620.44	7211.27
t1720	173 / 339	56	126934.68	171533 / 200679	6408.27	7211.04
t1721	899 / 1771	67	104679.26	126837 / -	4411.52	7212.83

The feasible solutions we find are of about the same quality (when optimality is not proved), so is the best lower bound we can achieve. We suspect that even though we have used four LP-CG pairs for the harder problems, this still just barely compensates for the tight coupling with CPLEX's internal strong branching routine.

5.5.4 Set 4 problems

From this problem set we have experimented with those that were neither solved to optimality by our Feasible Solution Heuristic nor were found infeasible. We have also omitted those problems where solving the LP relaxations took an inordinately long time. Tables 5.26 and 5.27 show the results for the remaining 4 problems. In these tests we have used a history of length 7 for detecting tailing off and at most 30 cuts were added per iteration. We selected 6 variables for strong branching and tested both the mixed and the "close to one-half and expensive" candidate selection rules. We did not attempt to generate violated odd hole constraints. These tests were carried out using four LP-CG pairs. These problems have not been publicly available, so there are no other published computational results for them.

5.5.5 Conclusion and future work

With our B&C implementation we have shown that the COMPSys framework is easy to adapt. It provided enough flexibility to implement methods fine-tuned for the Set Partitioning Problem and at the same time it made our programming task

Table 5.26: Basic B&C experiments for Set 4 (branching on "close to one-half")

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
sp1	128 / 233	70		11482 / 11482	0.00	6862.70
sp4	212 / 405	52	11821.79	12798 / -	1203.89	7222.80
sp5	2 / 3	1		27637 / 27673	484.53	888.24
sp10	519 / 903	42	44157.89	49835 / -	216.37	7218.44

Table 5.27: Basic B&C experiments for Set 4 (mixed branching variable selection)

	search tree		best	best / initial	time	
	nodes	depth	lower bound	upper bound	to best	total
sp1	127 / 223	75		11482 / 11482	0.00	6951.84
sp4	203 / 395	51	11843.05	12798 / -	271.49	7223.71
sp5	5 / 9	4		27637 / 27673	484.95	1423.37
sp10	582 / 1007	21	43839.22	49839 / -	899.53	7211.86

much easier.

There are areas where improvement could be made. Within COMPSys the strong branching interface to the LP solver needs to be improved.

On our part, we plan to continue research in several directions. First, we plan to experiment with the GLS-algorithm for finding odd holes to see whether it performs well for dense fractional graphs.

Second, new diving strategies should be explored. In the current implementation we found long "chains" diving to the bottom of the search tree. As a result, only few nodes close to the top of the tree were processed which resulted in weak lower bound for the hard problems (the integrality gaps were above 10% for the `t*` problems). More careful diving strategies might help to shrink the gap on these problems.

Chapter 6

The Graphical User Interface

The Graphical User Interface (GUI) is implemented as a separate process of the COMPSys framework. The GUI consists of two parts: an interactive graph drawing application (IGD) implemented purely in Tcl/Tk and an interface (DrawGraph) implemented in C that links the application to the other processes of the framework. DrawGraph is spawned by the Master process and communicates with the other processes via PVM. DrawGraph in turn forks a *wish* shell (a shell that accepts Tcl/Tk commands) and opens a pair of communication pipes, attaching them to the standard input and output of the wish shell (the technique was adapted from [Wel95]). Figure 6.1 illustrates the communication flow between COMPSys, the interface and the wish shell.

The GUI has been extensively used for debugging, both in the Cut Generator and in the LP processes. In this case, messages go from a process of the framework to IGD through the interface. Another, novel use of the GUI in the Set Partitioning

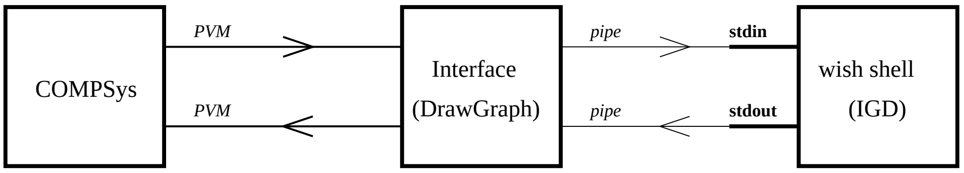


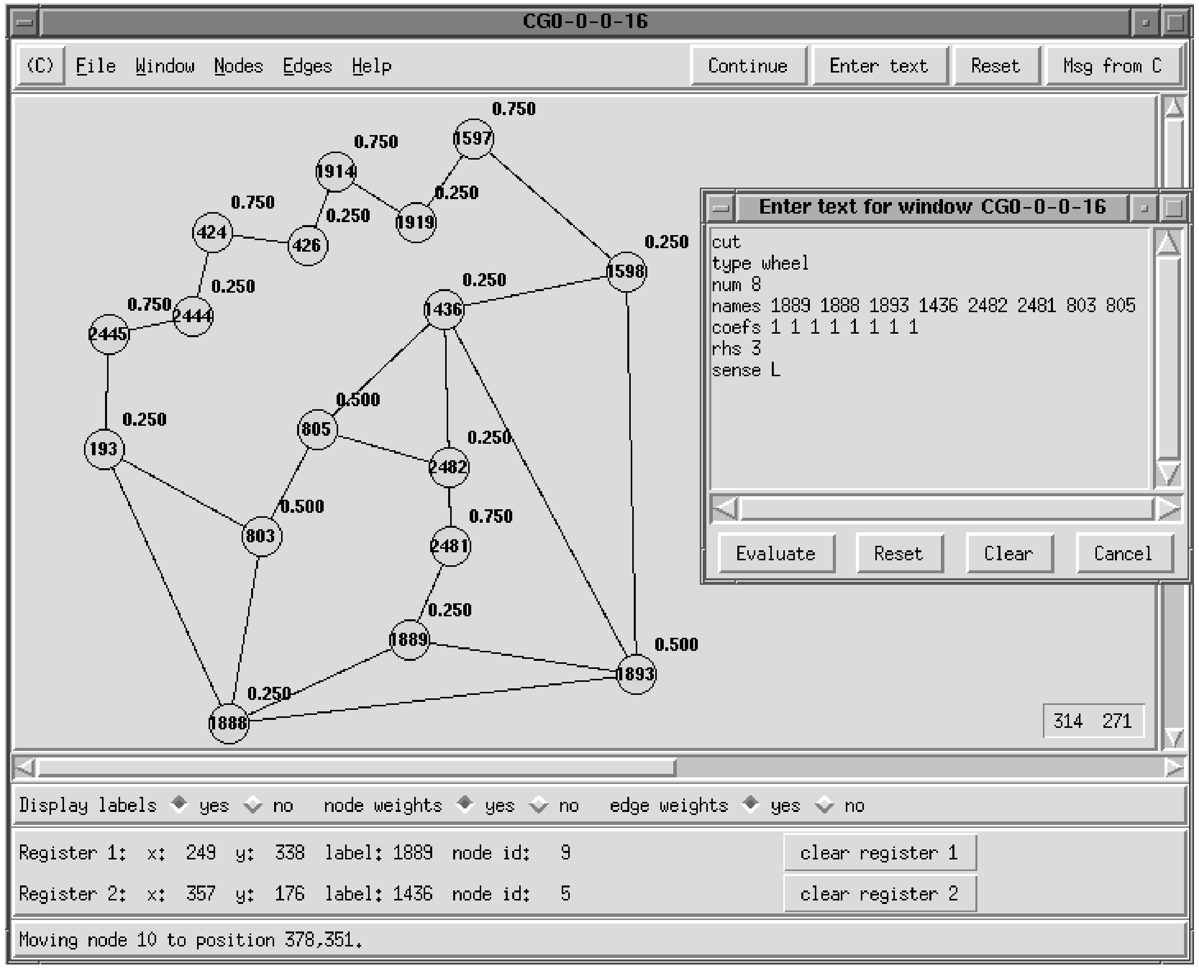
Figure 6.1: Communication flow between COMPSys and the GUI

setting is to send messages, namely cuts, from IGD to the Cut Generator process; that is, to generate violated cuts “by hand.” This enables us to test the effectiveness of cuts that are difficult (or not known how) to separate algorithmically. We will illustrate the “human cut generation” through an example in Section 6.3, after discussing IGD and DrawGraph.

6.1 Interactive Graph Drawing (IGD)

Tcl, an interpreted scripting language, extended by the Tk toolkit provides an environment that allows fast and relatively easy implementation of GUI’s that use windows and menus. IGD is implemented using Tcl version 7.5p1+ and Tk version 4.1p1+dash (it has also been tested under Tcl version 8.0p2+ and Tk version 8.0p2+dash and it will most likely work with higher versions as well). Download information about Tcl/Tk, manuals and related literature can be found at the Tcl WWW Info Site (`http://www.sco.com/Technology/tcl/Tcl.html`).

IGD is a library of Tcl/Tk functions for displaying, manipulating, scaling and printing undirected graphs (note that the package does not contain graph layout algorithms). Figure 6.2 gives an idea of the look and feel of the application; the



```
cut
type wheel
num 8
names 1889 1888 1893 1436 2482 2481 803 805
coefs 1 1 1 1 1 1 1 1
rhs 3
sense L
```

Display labels ◆ yes ◇ no   node weights ◆ yes ◇ no   edge weights ◆ yes ◇ no

```
Register 1:  x:  249  y:  338  label: 1889  node id:  9
Register 2:  x:  357  y:  176  label: 1436  node id:  5
```

Moving node 10 to position 378,351.

Figure 6.2: Screen shot of the GUI: problem v0416 at the root before branching

six graphs on Figure 2.3 were also created and printed using IGD. The library can be used as a stand-alone application without using the interface. In this case the library functions are sourced into a wish shell and can be invoked directly. As soon as one application window is displayed, the user can manipulate the windows and graphs via menus, buttons and mouse clicks (which are all bound to functions in the library), hence the adjective "interactive."

The basic units of the IGD application are windows (Figure 6.2). These windows contain menus, buttons, scrollbars and a drawing area called *canvas* where the graphs are displayed. The graphs displayed can be saved to and loaded from files in a special format designed for this application; the graph visible on the canvas can also be saved in a postscript file. New windows can be created, the display properties of windows (like fonts, node radii, dash patterns for the node and edge outlines) can be modified. A graph node is represented with a circle (different nodes can have different radii and different outlines), a label (a short text displayed within the circle) and an optional weight (a short text displayed North-East from the circle). The edges connecting the nodes are represented with lines (different edges can have different outlines) with optional weights (short texts placed East from the middle of the line). In the Set Partitioning setting the graph displayed is the fractional intersection graph (Sections 1.3 and 5.4), the node labels are the indices of the nodes in the application, and the weights are the LP solution values associated with the corresponding variables; the edges are all solid lines and no edge weights are used. The nodes of the graph can be moved around (the idea of how to implement this was borrowed from [Ous94]), edges attached to the moving node will move with the node. Moving nodes makes it easy to rearrange the graph so that special structures are easier to spot (like the wheel inequality in the graph on Figure 6.2, see Section 6.3).

The library with a short script to start the stand-alone application is available for download at `http://www.orie.cornell.edu/~eso/IGD/`. The package also contains a detailed description of the features outlined above and documentation of the library functions.

When IGD is used along with the DrawGraph interface, the library functions are sourced into the forked wish shell. The interface can invoke a function by simply placing a function call on the communication pipe attached to the shell's standard input, and whatever IGD places on the shell's output is caught by the interface. Additional functionalities bound to the buttons in the upper right corner of the window on Figure 6.2 were included so that the user can send back messages to the interface as well. The *Continue* button is used to hold up the interface (which in turn can hold up the framework) until the user is done with the the current graph; the *Enter text* button brings up a window into which the user can enter any text for interpretation by the interface (this is the smaller window on Figure 6.2); pressing the *Reset* button sends a request to the interface to redraw the same graph (this is useful since nodes of the graph can be moved around or deleted); and finally pressing the *Msg from C* button will bring up a window in which messages from the framework are displayed (we use this option to print out the violated inequalities found in the fractional graph during cut generation).

6.2 The interface (DrawGraph)

DrawGraph is a separate process of the framework spawned by the Master process. As soon as it is started, it forks the wish shell and sources the IGD library. Then it enters an infinite loop in which messages from IGD and from processes of the COMPSys framework are processed alternately. Messages from IGD are processed at once while messages received via PVM are placed into message buffers according to the addressee (the window to whom the message is addressed) and processed later. Each window opened through the interface has a unique "owner," the process of the framework that initiated the window. One process can own several windows and each window will accept messages only from its owner. This buffering of messages is necessary since the message flow to a window must be held up until the user has finished examining the graph displayed. Once one message from "each side" (IGD and the framework) of the interface has been read, one message per window (if any) is processed and the loop continues.

Similar to other processes of the framework, the user can customize the Draw-Graph process via user written functions. The only nontrivial user function is the interpretation of text entered from the application. We used this option only for entering cuts (violated valid inequalities). A cut is defined by its type (which can be any of the known cut types described in Section 2.3 or "other" if the inequality is none of these types), the number of variables on the left hand side with the names and coefficients, the value of the right hand side, the sense and range of the inequality (see the cut in the small window on Figure 6.2). Only the format of the cuts is checked here, their violation is computed in the Cut Generator.

6.3 Generating cuts by hand

The fractional intersection graph is displayed from the Cut Generator master either every time the LP relaxation is (re-)solved and the regular cut generation (described in Section 5.4) has already finished, or only when the built-in algorithms could not generate any violated inequalities (and branching would follow unless we can add some cuts). The first choice is beneficial for debugging as well, since we can see what kind of violated inequalities were generated internally. On the other hand, the second choice is very interesting, since here we can look for new types of cuts that we do not separate for in the Cut Generator. We enter cuts through the *Enter text* window as shown on Figure 6.2. The cuts are evaluated in the Cut Generator master, and those which are violated are sent to the LP process. All cuts entered this way are sent back to the message window of the GUI so that we can see the extent of violation for each cut.

Figure 6.2 presents the fractional intersection graph for the problem `v0416` before branching at the root of the search tree (after adding cuts internally and resolving the LP relaxation 16 times). Observe that there are no violated cliques, odd holes or antiholes in the graph (for instance, all maximal cliques are size 3 and the sum of the solution values in each case is exactly 1). However, it is easy to spot two violated wheel inequalities (one of those is entered in the small window).

Recall from Section 2.3 that one of the wheel inequalities $(I_{\mathcal{E}})$ is

$$\sum_{j \in W} x_j + \sum_{j \in \mathcal{E}} x_j + (k-1)x_0 \leq (|W| + |\mathcal{E}|)/2 - 1,$$

where W is all the nodes of the wheel, x_0 is the hub, and $\mathcal{E}/\mathcal{O}$ contain the spoke-ends that are of even/odd distance from the hub ($|\mathcal{E}| + |\mathcal{O}| = 2k+1$). This inequality is violated for the following two wheels (entering the first wheel is shown in the small window):

Wheel 1:

hub: `1889`; spoke-ends: $\mathcal{E} = \emptyset$, $\mathcal{O} = \{$`1888, 1893, 1436`$\}$;

spokes: `1889-1888`, `1889-1893`, `1889-2481-2482-1436`;

rim-paths between spoke-ends: `1888-1893`, `1893-1436`, `1436-805-803-1888`;

inequality: $\sum_{j \in W} x_j \leq 3$, value of left hand side: 3.25.

Wheel 2:

hub: `1598`; spoke-ends: $\mathcal{E} = \emptyset$, $\mathcal{O} = \{$`1888, 1893, 1436`$\}$;

spokes: `1598-1597-1919-1914-426-424-2444-2445-193-1888`, `1598-1893`, `1598-1436`;

rim-paths between spoke-ends: `1888-1893`, `1893-1436`, `1436-805-803-1888`;

inequality: $\sum_{j \in W} x_j \leq 6$, value of left hand side: 6.25.

Note that the two wheels have the same spoke-ends. Also, notice that the chain between `193` and `1598` could be replaced by `193-2445-1598` or `193-1597-1598` which would correspond to reversing the even subdivision (replacing an edge with a path that contains an even number of nodes) of the edge `2445-1598` or `193-1597`, respectively.

$$\begin{array}{rcr}
x_{193}+x_{2445}+x_{2444}+x_{424}+x_{426}+x_{1914}+x_{1919}+x_{1597}+x_{1598}+x_{1893}+x_{1888} & \leq & 5 \\
x_{1893}+x_{1436}+x_{1889}+x_{2481}+x_{2482} & \leq & 2 \\
x_{193}+x_{1888}+x_{803} & \leq & 1 \\
+x_{1436}+x_{2482}+x_{805} & \leq & 1 \\
x_{2445}+x_{2444} & \leq & 1 \\
x_{424}+x_{426} & \leq & 1 \\
x_{1914}+x_{1919} & \leq & 1 \\
x_{1597}+x_{1598} & \leq & 1 \\
x_{1889}+x_{2481} & \leq & 1 \\
x_{803}+x_{805} & \leq & 1 \\
\hline
2(x_{193}+x_{2445}+x_{2444}+x_{424}+x_{426}+x_{1914}+x_{1919}+x_{1597}+x_{1598}+x_{1893}+x_{1888}+x_{1436}+x_{1889}+x_{2481}+x_{2482}+x_{803}+x_{805}) & \leq & 15 \\
\hline
x_{193}+x_{2445}+x_{2444}+x_{424}+x_{426}+x_{1914}+x_{1919}+x_{1597}+x_{1598}+x_{1893}+x_{1888}+x_{1436}+x_{1889}+x_{2481}+x_{2482}+x_{803}+x_{805} & \leq & 7
\end{array}$$

Figure 6.3: Deriving a cut using the Chvátal-Gomory procedure

Figure 6.3 illustrates the Chvátal-Gomory procedure ([Chv73], also see Section 2.3.1) for this graph by deriving a cut that contains all the nodes of the graph. Add up the odd cycle inequalities for the following cycles:

`193-2445-2444-424-426-1914-1919-1597-1598-1893-1888` (length 11),

`1436-1893-1889-2481-2482` (length 5),

`193-803-1888` and `805-1436-2482` (both length 3);

and also add the edge inequalities for the edges `2445-2444`, `424-426`, `1914-1919`, `1597-1598`, `1889-2481`, `803-805`. Then all the nodes will be counted exactly twice on the left hand side while the right hand side adds up to 15. Dividing both sides by 2 we obtain the following valid inequality (which is violated by the current solution since the sum of the solution values on all the nodes is 7.5):

$$\sum_{j \in V} x_j \leq 7$$

where V denotes all the 17 nodes in the graph. A stable set of size 7 is for instance `193`, `2444`, `426`, `1597`, `1893`, `2481` and `805`. Notice that the above inequality is a rank inequality; thus, if the Chvátal condition (Section 2.3.1) holds, then it is also facet defining for the stable set polytope of the graph. Indeed, it is easy to see that all edges of the graph except `1436-1598` and `1888-1889` are α-critical, so the subgraph of all the α-critical edges on the 17 nodes is connected; that is, the Chvátal condition is satisfied.

Appendix A

Computation

A.1 Computing environment

- IBM RS/6000 Scalable POWERparallel System (SP)
- SP High Performance Switch 150 MByte/sec peak hardware bandwidth
- Processor type POWER2 Super Chip (P2SC) with 128 KByte data cache, 256 bit memory bus
- Thin nodes 120 MHz clock speed, 256MByte or 1 GByte memory;
  Wide nodes 135 MHz clock speed, 1 or 2 GByte memory
- Operating system AIX 4.2.1
- C compiler xlC 3.1.4.7 with flags
  "-O3 -qmaxmem=16384 -qarch=pwr2 -qtune=pwr2s"

- Message passing protocol PVM 3.3.11 [PVM]
- LP solver CPLEX 4.0.9 [CPX95]

All times reported are wall-clock times. Users of the SP get exclusive use of the assigned processors while running batch jobs. Therefore, especially for longer jobs, wall-clock time approximates CPU time closely.

A.2 The test bed

Our methods have been tested on four sets of problems (see details below). Two sets are Airline Crew Scheduling Problems originating at major airlines, and two are Vehicle Routing Problems from the ZIB Telebus Project. Problems in Sets 1 and 3 are referenced in the papers listed below but we are not aware of any publications about the remaining problems. All problems are publicly available.

Set 1 (55 problems)

origin:	major airlines
published:	originally in [HP93]; [BC96], [Bor97]
source:	`http://mscmga.ms.ic.ac.uk/jeb/orlib/sppinfo.html`
comments:	LP relaxation solves 15 of the problems to optimality

Set 2 (7 problems)

origin:	Telebus Project at ZIB
published:	–
source:	`http://www.zib.de/Optimization/index.en.html`
comments:	very hard problems

Set 3 (27 problems)

origin:	Telebus Project at ZIB
published:	[BGKK97]
source:	`http://www.zib.de/borndoerfer`
comments:	14 clustering and 13 chaining problems

Set 4 (14 problems)

origin:	major airline ([Anb])
published:	–
source:	`http://www.orie.cornell.edu/~eso`
comments:	removed side constraints before optimization

Tables A.1, A.2, A.3 and A.4 give some basic properties of these problems. In the first five columns the name and size (number of columns, rows, nonzeros and density) are listed. Then the best published feasible solution value follows. For Set 1 problems the optimal solution is known (and the number of ones in an optimal solution is also listed). For problems in Set 3 an "*" marks those problems where the best feasible solution is proved to be optimal.

After the initial problems were reduced using our fast strategy followed by one SUMC (see Section 3.4.4), the first LP relaxation of the problem was solved by CPLEX's barrier method with dual crossover (default parameter setting). The next five columns contain the optimal value of the LP relaxation (or "IP" if the solution is integral), the time spent solving the LP, the number of variables at level 1 and of those at other nonzero levels, and finally the ratio of variables at level 1 to all variables at nonzero levels.

The CPLEX MIP solver was also applied to the reduced problems. CPLEX MIP parameters were set to their default values except for the following (see Appendix C for description of these parameters):

- `tilim` was set to 7200 seconds;
- `varsel` was set to *strong branching* based on our preliminary tests;
- `epagap` and `objdif` were set to the *granularity* of the problems (.009999 for Set 2 problems, .9999 for the rest);
- `epgap` was set to 10^{-9} (its lowest possible value).

The last three columns contain the value of the best feasible solution found by the CPLEX MIP optimizer ("*" when it is optimal, "–" if no feasible solution was found), the number of search tree nodes (if a feasible solution was found) and the time spent in optimization. The number of search tree nodes is marked with a "+" if the tree was not completely enumerated. Note that in three cases (`t0415`, `t0417` and `t0419`) the CPLEX MIP optimizer has found a better feasible solution than the best published value.

Table A.1: Basic properties of problems in set 1, part 1

	Original problem				Optimal soln		Solve first LP relax after Reduce					CPLEX MIP		
name	cols	rows	nzs	dens	value	#1s	opt	time	#1s	#frac	ratio	soln	nodes	time
aa01	8904	823	72965	1.00	56137	102	55535.44	8.19	17	291	5.52	*	149	713.34
aa02	5198	531	36359	1.32	30494	81	IP	1.94	81					
aa03	8627	825	70806	0.99	49649	106	49616.36	5.66	69	91	41.67	*	2	69.98
aa04	7195	426	52121	1.70	26374	66	25877.61	3.74	5	224	2.18	*	189	648.44
aa05	8308	801	65953	0.99	53839	105	53735.93	5.42	53	142	26.42	*	11	91.01
aa06	7292	646	51728	1.10	27040	95	26977.19	4.94	51	112	31.29	*	10	81.48
kl01	7479	55	56242	13.67	1086	13	1084.00	1.07	5	16	23.81	*	8	9.52
kl02	36699	71	212536	8.16	219	17	215.25	4.03	4	27	12.90	*	125	634.37
nw01	51975	135	410894	5.86	114852	71	IP	13.50	71					
nw02	87879	145	721736	5.66	105444	72	IP	35.96	72					
nw03	43749	59	363939	14.10	24492	13	24447.00	10.72	8	7	53.33	*	2	25.45
nw04	87482	36	636666	20.22	16862	9	16310.67	15.53	7	6	53.85	–		7207.59
nw05	288507	71	2063641	10.07	132878	36	IP	105.25	36					
nw06	6774	50	61555	18.17	7810	8	7640.00	1.15	2	16	11.11	*	8	10.04
nw07	5172	36	41187	22.12	5476	6	IP	0.30	6					
nw08	434	24	2332	22.39	35894	12	IP	0.03	12					
nw09	3103	40	20111	16.20	67760	16	IP	0.15	16					
nw10	853	24	4336	21.18	68271	13	IP	0.01	13					
nw11	8820	39	57250	16.64	116256	19	116254.50	0.26	16	4	80.00	*	1	0.20
nw12	626	27	3380	20.00	14118	15	IP	0.02	15					
nw13	16043	51	104541	12.78	50146	22	50132.00	1.97	19	6	76.00	*	2	2.73
nw14	123409	73	904910	10.04	61844	26	IP	30.01	26					
nw15	467	31	2830	19.55	67743	7	IP	0.06	7					
nw16	148633	139	1501820	7.27	1181590	125	IP	116.42	125					
nw17	118607	61	1010039	13.96	11115	16	10875.75	27.12	7	21	25.00	*	8	171.36
nw18	10757	124	91028	6.82	340160	41	338864.25	3.14	27	36	42.86	*	1	5.05
nw19	2879	40	25193	21.88	10898	7	IP	0.19	7					

Table A.2: Basic properties of problems in set 1, part 2

	Original problem				Optimal soln		Solve first LP relax after Reduce					CPLEX MIP		
name	cols	rows	nzs	dens	value	#1s	opt	time	#1s	#frac	ratio	soln	nodes	time
nw20	685	22	3722	24.70	16812	5	16626.00	0.04	0	15	0.00	*	4	0.21
nw21	577	25	3591	24.89	7408	7	7380.00	0.03	3	7	30.00	*	1	0.03
nw22	619	23	3399	23.87	6984	7	6942.00	0.04	3	7	30.00	*	1	0.04
nw23	711	19	3350	24.80	12534	8	12317.00	0.03	4	6	40.00	*	19	0.49
nw24	1366	19	8617	33.20	6314	7	5843.00	0.04	4	6	40.00	*	2	0.06
nw25	1217	20	7341	30.16	5960	5	5852.00	0.04	1	8	11.11	*	2	0.10
nw26	771	23	4215	23.77	6796	6	6743.00	0.04	3	5	37.50	*	1	0.04
nw27	1355	22	9395	31.52	9933	5	9877.50	0.05	3	3	50.00	*	1	0.04
nw28	1210	18	8553	39.27	8298	3	8169.00	0.05	2	3	40.00	*	1	0.05
nw29	2540	18	14193	31.04	4274	4	4185.33	0.25	0	11	0.00	*	8	1.27
nw30	2653	26	20436	29.63	3942	4	3726.80	0.12	1	8	11.11	*	2	0.32
nw31	2662	26	19977	28.86	8038	4	7980.00	0.16	2	5	28.57	*	3	0.40
nw32	294	19	1357	24.29	14877	7	14570.00	0.03	4	4	50.00	*	9	0.10
nw33	3068	23	21704	30.76	6678	5	6484.00	0.33	2	6	25.00	*	1	0.36
nw34	899	20	5045	28.06	10488	4	10453.50	0.05	2	4	33.33	*	1	0.03
nw35	1709	23	10494	26.70	7216	6	7206.00	0.06	4	4	50.00	*	1	0.09
nw36	1783	20	13160	36.90	7314	4	7260.00	0.22	1	6	14.29	*	14	2.26
nw37	770	19	3778	25.82	10068	4	9961.50	0.03	2	4	33.33	*	1	0.04
nw38	1220	23	9071	32.33	5558	5	5552.00	0.11	1	6	14.29	*	1	0.10
nw39	677	25	4494	26.55	10080	5	9868.50	0.03	3	3	50.00	*	2	0.05
nw40	404	19	2069	26.95	10809	4	10658.25	0.03	0	9	0.00	*	1	0.04
nw41	197	17	740	22.10	11307	5	10972.50	0.02	3	3	50.00	*	2	0.02
nw42	1079	23	6533	26.32	7656	4	7485.00	0.11	1	7	12.50	*	4	0.31
nw43	1072	18	4859	25.18	8904	6	8897.00	0.06	1	7	12.50	*	1	0.08
us01	1053137	145	13636541	8.93	10022	14	9963.07	129.76	0	47	0.00	*	13	2088.99
us02	13635	100	192716	14.13	5965	12	IP	1.02	12					
us03	85552	77	1211929	18.40	5338	7	IP	5.04	7					
us04	28016	163	297538	6.52	17854	24	17731.67	0.97	12	24	33.33	*	1	1.89

Table A.3: Basic properties of problems in set 3

	Original problem				Best feas	Solve first LP relax after Reduce					CPLEX MIP		
name	cols	rows	nzs	dens	value	opt	time	#1s	#frac	ratio	soln	nodes	time
v0415	7684	1518	20668	0.18	* 2429415	2423977.00	0.75	414	44	90.39	*	940	184.28
v0416	19020	1771	58453	0.17	* 2725602	2715490.67	0.66	508	121	80.76	2725748	+ 45051	7200.32
v0417	143317	1765	531820	0.21	* 2611518	2603308.50	9.77	467	68	87.29	2612393	+ 2716	7206.09
v0418	8306	1765	20748	0.14	* 2845425	2836836.67	0.75	504	91	84.71	*	10423	2863.07
v0419	15709	1626	52867	0.21	* 2590326	2582994.00	0.54	454	73	86.15	*	499	93.15
v0420	4099	958	10240	0.26	* 1696889	1688793.33	0.36	298	66	81.87	*	4301	570.65
v0421	1814	952	3119	0.18	* 1853951	1848949.00	0.16	260	36	87.84	*	145	7.31
v1616	67441	1439	244727	0.25	* 1006460	1002954.62	8.65	490	180	73.13	1006503	+ 8333	7201.36
v1617	113655	1619	432278	0.23	1102586	1098263.23	15.22	523	261	66.71	1103266	+ 4953	7203.97
v1618	146715	1603	545337	0.23	1154458	1147777.67	30.51	521	244	68.10	–		7207.31
v1619	105822	1612	401097	0.24	1156338	1150943.29	15.05	490	351	58.26	1157479	+ 4594	7204.01
v1620	115729	1560	444445	0.25	* 1140604	1136666.52	39.04	464	368	55.77	1140771	+ 1119	7209.87
v1621	24772	938	76971	0.33	* 825563	822339.42	2.00	331	152	68.53	825563	+ 16776	7200.66
v1622	13773	859	41656	0.35	* 793445	790076.50	1.78	328	108	75.23	*	6241	2708.27
t0415	7254	1518	48867	0.44	5590096	5125429.50	16.30	100	784	11.31	5590095	+ 913	7204.27
t0416	9345	1771	62703	0.38	6130217	5829948.77	20.61	69	917	7.00	–		7205.41
t0417	7894	1765	54885	0.39	6043157	5610564.20	20.44	96	821	10.47	5951357	+ 784	7202.25
t0418	8676	1765	66604	0.43	6550898	6142664.90	28.14	68	942	6.73	–		7203.92
t0419	9362	1626	64745	0.43	5916956	5644051.00	17.15	42	858	4.67	5910913	+ 827	7207.36
t0420	4583	958	27781	0.63	4276444	3983951.22	4.62	30	537	5.29	–		7202.66
t0421	4016	952	24214	0.63	4354411	4057701.31	3.85	23	537	4.11	–		7200.11
t1716	56865	467	249149	0.94	161636	121648.87	8.01	0	436	0.00	224785	+ 914	7213.81
t1717	73885	551	325689	0.80	184692	134531.02	16.03	0	511	0.00	–		7216.08
t1718	67796	523	305064	0.86	162992	126334.47	10.79	0	497	0.00	–		7201.98
t1719	72520	556	317391	0.79	187677	138708.87	11.95	0	514	0.00	–		7202.15
t1720	69134	538	310512	0.83	172752	126333.20	13.26	0	513	0.00	–		7209.14
t1721	36039	357	148848	1.16	127424	103748.46	4.20	0	333	0.00	172841	+ 1376	7206.84

Table A.4: Basic properties of problems in sets 2 and 4

	Original problem				Solve first LP relax after Reduce					CPLEX MIP		
name	cols	rows	nzs	dens	opt	time	#1s	#frac	ratio	soln	nodes	time
0321.4	71201	1202	818344	0.96	35742.46	161.390	0	1038	0.00	–		7316.61
0331.3	45637	664	467206	1.54	28402.76	40.300	5	629	0.79	–		7204.67
0331.4	46915	664	431054	1.38	29730.03	39.920	0	572	0.00	–		7204.41
0341.3	45800	658	431675	1.43	31004.06	39.200	13	586	2.17	–		7202.25
0341.4	46508	658	384305	1.26	34276.06	35.870	2	538	0.37	–		7201.98
0351.3	64953	1156	846140	1.13	35032.59	149.980	11	980	1.11	–		7207.96
0351.4	69922	1156	804403	1.00	34434.36	145.890	1	977	0.10	–		7223.44
nf260	276752	2198	1382054	0.23	47405.00	45.210	462	16	96.65	* 47420	1	2663.99
sp1	6954	204	94688	6.67	9987.80	4.060	1	134	0.74	–		7205.04
sp2	3686	173	45066	7.07	13522.93	1.470	0	101	0.00	* 13914	3052	1139.42
sp3	1668	111	27178	14.68	12766.12	0.500	1	57	1.72	* 12943	7	7.86
sp4	9144	368	150881	4.48	11389.42	6.720	0	207	0.00	–		7204.32
sp5	13718	684	162572	1.73	27403.20	27.770	4	498	0.80	* 27637	3	448.47
sp6	50722	2504	550644	0.43	157414.80	487.780	5	1551	0.32	–		7216.99
sp7	43459	2991	499347	0.38	162349.98	675.170	5	1950	0.26	–		7214.43
sp8	91123	4810	1004473	0.23	368714.87	1165.500	23	2945	0.77	–		7261.00
sp9	50013	2917	742546	0.51	166705.53	122.360	3	1535	0.20	–		7211.43
sp10	13128	781	220703	2.15	43045.72	2.670	3	280	1.06	–		7202.94
sp11	2775	104	56686	19.64	3093.13	0.300	0	44	0.00	INFEAS		113.14
sp12	84746	3218	910022	0.33	248004.45	1375.820	2	2308	0.09	–		7227.58
sp14	47214	3217	523992	0.34	250210.43	970.020	1	2315	0.04	–		7231.95

A.3 Results by others

Here we summarize those results of Hoffman and Padberg ([HP93]), Borndörfer ([Bor97]), and Borndörfer et al. ([BGKK97]) that can be directly compared to our results.

Hoffman and Padberg's results for the problems in Set 1 are reported in Tables A.5 and A.6. These tables contain the name, original size (number of columns and rows), and the value of the optimal solution for each problem; the size of the problem after their initial problem size reduction; the upper bound obtained by their feasible solution heuristic ("F" if their heuristic failed and "IP" if the first LP relaxation provides an integral solution); the number of search tree nodes created by their Branch-and-Cut algorithm (not counting the root) and the total time they spent in the three phases of the solution process. They do not report execution times for their initial problem size reduction and heuristic procedures separately. The total time reported for `us01` is after the duplicate columns have already been eliminated from the problem. Their experiments were carried out on a RS/6000 model 550 machine for most of the problems and on a CONVEX model C-220 machine using one of its two processors for the four largest problems (marked with a "*"). They used the CPLEX Callable Library but they do not report the version number.

Borndörfer's results for the problems in Set 1 are collected in Tables A.7 and A.8. The tables contain the name, original size (number of columns and rows), and the value of the optimal solution for each problem; the size of the problem after his initial

Table A.5: Computational results by Hoffman and Padberg, Set 1, part 1

	Original		Optimal	Reduced		Heur	Tree	Total
name	cols	rows	value	cols	rows	value	size	time
aa01	8904	823	56137	7532	607	F	90	14441.00
aa02	5198	531	30494	3846	360	IP		10.15
aa03	8627	825	49649	6694	537	49713	0	48.42
aa04	7195	426	26374	6122	342	27080	494	139337.00
aa05	8308	801	53839	6235	521	54060	4	215.30
aa06	7292	646	27040	5862	488	27040	0	37.30
kl01	7479	55	1086	5957	50	1096	2	35.40
kl02	36699	71	219	16542	69	221	0	134.38
nw01	51975	135	114852	50069	135	IP		19.25
nw02	87879	145	105444	85258	145	IP		37.35
nw03	43749	59	24492	38964	59	25086	0	24.00
nw04	87482	36	16862	46190	36	19492	44	2642.00
nw05	288507	71	132878	202603	71	IP		192.50
nw06	6774	50	7810	5977	50	9616	0	10.41
nw07	5172	36	5476	3108	36	IP		0.74
nw08	434	24	67760	2305	40	IP		0.08
nw09	3103	40	35894	356	24	IP		0.53
nw10	853	24	68271	659	24	IP		0.13
nw11	8820	39	116256	6488	39	116259	0	2.05
nw12	626	27	14118	454	27	IP		0.09
nw13	16043	51	50146	10950	51	50240	0	4.29
nw14	123409	73	61844	95178	73	IP		87.60
nw15	467	31	67743	463	29	IP		0.10
nw16	148633	139	1181590	138951	139	IP		174.40
nw17	118607	61	11115	78186	61	11907	4	87.53
nw18	10757	124	340160	8460	124	392090	0	62.49
nw19	2879	40	10898	2145	40	IP		0.50

Table A.6: Computational results by Hoffman and Padberg, Set 1, part 2

	Original		Optimal	Reduced		Heur	Tree	Total
name	cols	rows	value	cols	rows	value	size	time
nw20	685	22	16812	566	22	16812	0	0.62
nw21	577	25	7408	426	25	7676	0	0.30
nw22	619	23	6984	531	23	6984	0	0.34
nw23	711	19	12534	473	18	13702	0	0.34
nw24	1366	19	6314	925	19	6568	0	0.56
nw25	1217	20	5960	844	20	6610	0	0.62
nw26	771	23	6796	473	18	7452	0	0.34
nw27	1355	22	9933	926	22	F	0	0.28
nw28	1210	18	8298	825	18	F	0	0.40
nw29	2540	18	4274	2034	18	4378	0	0.99
nw30	2653	26	3942	1884	26	3942	0	0.75
nw31	2662	26	8038	1823	26	9754	0	1.43
nw32	294	19	14877	251	18	15600	0	0.17
nw33	3068	23	6678	2415	23	7536	0	1.45
nw34	899	20	10488	750	20	11613	0	0.30
nw35	1709	23	7216	1403	23	7340	0	0.48
nw36	1783	20	7314	1408	20	7634	0	3.68
nw37	770	19	10068	639	19	10377	0	0.19
nw38	1220	23	5558	911	23	5712	0	1.35
nw39	677	25	10080	567	25	F	0	0.19
nw40	404	19	10809	336	19	11070	0	0.21
nw41	197	17	11307	177	17	F	0	0.06
nw42	1079	23	7656	895	23	7846	0	0.99
nw43	1072	18	8904	982	17	8904	0	0.38
us01	1053137	145	10022	370642	90	10075	0	1410.60
us02	13635	100	5965	9022	45	IP		4.78
us03	85552	77	5338	27084	53	IP		20.27
us04	28016	163	17854	6564	112	17854	0	11.19

problem size reduction; the result of his feasible solution heuristic (the integrality gap computed as $(\bar{z} - \underline{z})/\bar{z}$, and the size of the problem after applying the problem size reduction once more) and the total time of the initial reduction, heuristic, and one more application of the reduction if a feasible solution has been found. After this, the table lists the number of search tree nodes created (including the root) and the time spent in his Branch-and-Cut solution process using default strategy (as far as we understand these times do not contain the time spent in the initial problem size reduction and heuristic). The problem size reduction and heuristic were carried out on a Sun Ultra Sparc 1 Model 170E workstation, while the Branch-and-Cut experiments were run on a Sun Ultra Sparc 2 Model 200E workstation. CPLEX V5.0 was used as the LP engine.

Table A.9 summarizes the results of Borndörfer et al. for the Set 3 problems. First the name and original size (number of columns and rows) of the problem are given, followed by the size after the initial problem size reduction; the lower and upper bounds obtained by their Branch-and-Cut procedure (the space of the lower bound is left empty if the optimality of the upper bound has been proved); the number of search tree nodes created (including the root) and the time spent in their Branch-and-Cut solution process. Note that they also publish results for the `v16` problems with a 2 minute time limit which we do not present here (with this experiment they demonstrate the degenerate nature of these problems). These experiments were carried out on a Sun Ultra Sparc 1 Model 170E, with CPLEX V4.0.

We tried to compare the architectures using the information provided by The

Table A.7: Computational results by Borndörfer, Set 1, part 1

	Original		Optimal	Reduced		Heuristic				Tree	Total
name	cols	rows	value	cols	rows	gap	cols	rows	time	size	time
aa01	8904	823	56137	7625	616	3.46	7586	616	1.55	97	238.71
aa02	5198	531	30494	3928	361	IP			0.24		1.99
aa03	8627	825	49649	6970	558	0.43	6823	558	1.09	1	12.47
aa04	7195	426	26374	6200	343	5.21	6189	343	1.62	181	319.19
aa05	8308	801	53839	6371	533	0.37	6354	532	1.00	7	13.62
aa06	7292	646	27040	6064	507	0.25	892	419	1.18	3	8.76
kl01	7479	55	1086	5957	47	1.00	1151	44	0.36	3	1.79
kl02	36699	71	219	16542	69	2.16	3415	62	1.41	1	6.05
nw01	51975	135	114852	50069	135	IP			0.77		2.70
nw02	87879	145	105444	85258	145	IP			1.39		5.61
nw03	43749	59	24492	38956	53	2.70	421	50	1.62	1	7.34
nw04	87482	36	16862	46189	35	9.47	15121	35	2.31	85	319.19
nw05	288507	71	132878	202482	58	IP			9.00		30.72
nw06	6774	50	7810	5936	37	18.98	883	37	0.28	3	1.02
nw07	5172	36	5476	3104	33	IP			0.11		0.20
nw08	434	24	67760	349	19	IP			0.01		0.02
nw09	3103	40	35894	2296	33	IP			0.07		0.14
nw10	853	24	68271	643	20	IP			0.02		0.02
nw11	8820	39	116256	5946	28	0.00	32	25	0.04	1	0.56
nw12	626	27	14118	451	25	IP			0.02		0.02
nw13	16043	51	50146	10901	48	0.21	100	46	0.45	3	1.24
nw14	123409	73	61844	95169	68	IP			3.73		19.23
nw15	467	31	67743	465	29	IP			0.01		0.02
nw16	148633	139	1181590	1	0	PP			7.13	0	7.11
nw17	118607	61	11115	78173	54	14.48	11332	52	4.84	3	29.03
nw18	10757	124	340160	7934	81	8.44	7598	81	0.74	1	2.19
nw19	2879	40	10898	2134	32	IP			0.07		0.13

Table A.8: Computational results by Borndörfer, Set 1, part 2

	Original		Optimal	Reduced		Heuristic				Tree	Total
name	cols	rows	value	cols	rows	gap	cols	rows	time	size	time
nw20	685	22	16812	566	22	1.11	18	15	0.03	1	0.04
nw21	577	25	7408	426	25	9.91	51	19	0.03	1	0.05
nw22	619	23	6984	531	23	0.60	18	17	0.00	1	0.04
nw23	711	19	12534	430	12	2.65	27	11	0.03	1	0.06
nw24	1366	19	6314	926	19	11.04	43	16	0.02	1	0.07
nw25	1217	20	5960	844	20	11.41	101	20	0.02	1	0.09
nw26	771	23	6796	514	21	2.87	30	17	0.03	1	0.05
nw27	1355	22	9933	926	22	4.51	19	13	0.03	1	0.06
nw28	1210	18	8298	599	18	5.97	20	11	0.03	1	0.03
nw29	2540	18	4274	2034	18	13.06	488	17	0.08	3	0.37
nw30	2653	26	3942	1884	26	69.84	1884	26	0.07	3	0.56
nw31	2662	26	8038	1823	26	2.23	34	20	0.06	1	0.14
nw32	294	19	14877	241	17	3.64	42	13	0.01	3	0.03
nw33	3068	23	6678	2415	23	2.96	19	13	0.06	1	0.17
nw34	899	20	10488	750	20	3.18	20	15	0.01	1	0.04
nw35	1709	23	7216	1403	23	8.74	91	19	0.03	1	0.10
nw36	1783	20	7314	1408	20	∞	1408	20	0.05	5	0.49
nw37	770	19	10068	639	19	6.17	22	13	0.03	1	0.06
nw38	1220	23	5558	881	20	0.11	18	15	0.03	1	0.09
nw39	677	25	10080	567	25	8.27	29	11	0.01	3	0.05
nw40	404	19	10809	336	19	4.96	36	13	0.03	1	0.03
nw41	197	17	11307	177	17	4.23	15	12	0.01	1	0.02
nw42	1079	23	7656	820	19	2.23	19	15	0.04	1	0.08
nw43	1072	18	8904	983	17	8.30	983	17	0.04	1	0.08
us01	1053137	145	10022	351018	86	5.79	36201	86	57.95	3	228.68
us02	13635	100	5965	8946	44	IP			0.50		1.31
us03	85552	77	5338	23207	50	IP			3.32		5.50
us04	28016	163	17854	4285	98	0.73	86	69	1.05	1	1.63

Table A.9: Computational results by Borndörfer et al., Set 3

	Original		Reduced		Branch-and-Cut			Total
name	cols	rows	cols	rows	lower bound	upper bound	nodes	time
v0415	7684	1518	4536	598		2429415	9	5.68
v0416	19020	1771	11225	812		2725602	643	120.53
v0417	143317	1765	55769	715		2611518	41	174.07
v0418	8306	1765	4957	742		2845425	7	5.72
v0419	15709	1626	7852	650		2590326	1	3.99
v0420	4099	958	2593	417		1696889	1	1.31
v0421	1814	952	1134	286		1853951	3	0.72
v1616	67441	1439	52926	1230		1006460	1605	4219.41
v1617	113655	1619	85457	1409	1102357	1102586	3571	7200.61
v1618	146715	1603	90973	1396	1152989	1154458	296	7222.28
v1619	105822	1612	85696	1424	1156072	1156338	880	7205.74
v1620	115729	1560	89512	1365		1140604	8161	5526.43
v1621	24772	938	16683	807		825563	5	13.79
v1622	13773	859	11059	736		793445	3	9.69
t0415	7254	1518	3312	870	5163849	5590096	167	7218.94
t0416	9345	1771	3298	974	5882041	6130217	144	7207.46
t0417	7894	1765	3774	897	5656886	6043157	71	7310.58
t0418	8676	1765	4071	999	6185168	6550898	87	7239.54
t0419	9362	1626	3287	904	5689134	5916956	100	7251.57
t0420	4583	958	1872	562	4036526	4276444	362	7208.44
t0421	4016	952	1691	557	4113080	4354411	375	7213.44
t1716	56865	467	11952	467	122408	161636	69	7212.95
t1717	73885	551	16428	551	135539	184692	41	7331.93
t1718	67796	523	16310	523	127040	162992	44	7238.72
t1719	72520	556	15846	556	139332	187677	37	7281.77
t1720	69134	538	16195	538	127222	172752	38	7349.28
t1721	36039	357	9043	357	104698	127424	174	7243.42

Standard Performance Evaluation Corporation (`http://www.specbench.org`). The architectures used by Hoffman and Padberg are not listed there. The integer arithmetic of a thin node of the SP is slightly slower than that of the Ultra Sparcs. On the other hand, the floating point arithmetic of a thin node is about 60% and 23% faster than that of an Ultra Sparc 1 Model 170E and an Ultra Sparc 2 Model 200E, respectively. Note that our problem size reduction methods rely only on integer arithmetic.

Appendix B

Implementing Reduce()

In this appendix we describe the main data structure and the parameters used in Reduce(). See Section 3.4 for a general overview of the function.

B.1 Reduce main data structure

A field can be an input (`IN`), an output (`OUT`) or both (`IN/OUT`) for Reduce().

`reduce_params *rpar (IN)`

Parameters. Must be filled out before invoking Reduce().

`int feasibility (OUT)`

Indicates the feasibility status of the problem. Possible values are `FEASIBLE`, `INFEASIBLE`, `FEASIBILITY_NOT_KNOWN`.

`int ones_num (IN/OUT)`

As input, the number of variables to be fixed to one; as output, the number of variables fixed to one by Reduce() (including those in the input).

`int *ones (IN/OUT)`

As input, it contains the names of variables to be fixed to one; as output, it contains the names of variables that have been fixed to one by Reduce(). New variables fixed to one are appended after those in the input. Space must be allocated for `cmatrix->rownum` entries.

`int merged_num (IN/OUT)`

As input, the number of variable pairs that were merged before invoking Reduce(); as output, the total number of merged variable pairs.

`colname_pair *merged (IN/OUT)`

As input, it contains pairs of names of variables that were merged before invoking Reduce(). As output, it contains the names of all merged variable pairs. New merged pairs are appended after those in the input. Space must be allocated for `cmatrix->rownum` entries.

The `colname_pair` structure has two integer fields, `int name1` and `name2`. The merged variable will inherit the name `name1`.

`col_ordered *cmatrix (IN/OUT)`

Column ordered representation of the problem matrix. The `colnum`, `rownum`, `obj`, `matind` and `matbeg` fields must be filled before invoking Reduce().

`row_ordered *rmatrix (IN/OUT)`

Row ordered representation of the problem matrix.

B.2 Reduce parameters

`int verbosity`

Determines the amount of information written into the trace-file. Between `0` (nothing) and `5` (all information).

`int fix_lex_order`

TRUE/FALSE. Order the columns into lexicographically ascending order at the beginning of Reduce() or not. (Columns might be already ordered.)

`int strategy`

Which strategy to use. The following are implemented: maximal reduction without SUMC (0), maximal reduction with SUMC (1), fast reduction without SUMC (2), fast reduction with SUMC (3), SUMC only (4), remove duplicate columns and rows only (5).

`int dupc, sumc, clext, domr, singl, dtwo`

TRUE/FALSE. Reduction modules are enabled/disabled.

`double sumc_frac, clext_frac, domr_frac, singl_frac, dtwo_frac`

Between 0 and 1. The corresponding reduction function is repeated if at least this fraction of the columns in the current matrix are marked for deletion by the most recent application of the function (`repeat_fraction`). Note that DUPC need not be repeated.

`double all_frac`

Between 0 and 1. The loop in the fast strategy repeats if at least this fraction of the columns have been deleted during the last pass through the loop.

`double sumc_cost_avg_tolerance, sumc_tolerance_increment`

In the SUMC reduction function a column can be a prospective summand only if its cost per length ratio is below that of the remainder multiplied by `sumc_cost_avg_tolerance`. The smaller this parameter, the more restrictive the search. Therefore every time the reduction function is repeated `sumc_cost_avg_tolerance` is increased by `sumc_tolerance_increment`.

`int sumc_max_summands_num`

Limit on the depth of the recursion in the SUMC reduction function.

`double sumc_chunk, double sumc_chunk_frac`

Both between 0 and 1. In the SUMC reduction function only part of the columns are examined at a time, the size of the "chunk" is the number of columns multiplied by `sumc_chunk`. Only if at least `sumc_chunk_frac` fraction of the columns in the current chunk are marked for deletion will the reduction continue for the next chunk.

`int clext_samplelen_perc, clext_samplelen_min, clext_samplelen_max`

Together determine the sample length for a row in the CLEXT reduction function. The sample is the entire row if the size of the row's support is not more than `clext_samplelen_min`. Otherwise `clext_samplelen_perc` (between 0 and 100) percentage of the row's support is sampled, but not less than `clext_samplelen_min` and not more than `clext_samplelen_max` columns.

Appendix C

Implementing the feasible solution heuristic

First we describe a few internal CPLEX parameters for which non-default values were considered in our experiments. For some of these parameters non-default values were necessary for correctness, while for the rest the modified parameter settings were to improve efficiency. See Chapter 9 of the CPLEX manual ([CPX95]) for more details. Then we list the parameters that were referred to in the discussion of the feasible solution heuristic (Section 4.2.3).

C.1 CPLEX parameters

`double tilim`

Time limit on one optimization call (in seconds).

`int aggind, int coeredind, int depind, int preind`

TRUE/FALSE. Preprocessing options within CPLEX are enabled/disabled (CPLEX Aggregator, coefficient reduction, dependency check, CPLEX Presolve). Used default settings (all but the dependency check are enabled).

`int dpriind`

Dual simplex pricing algorithm. Used *steepest edge pricing* instead of default.

`int basinterval`

Simplex basis-file saving frequency. Set so that basis is never saved.

`int baralg`

Barrier algorithm. Used the default setting *primal-dual log barrier.*

`int brdir, double bttol, int ndsel, int varsel`

Control the way branching is done by the CPLEX MIP solver. Used default settings for `brdir` (branching direction), `bttol` (backtracking tolerance – how much the LP optimum can degrade before a new search tree node is chosen instead of one of the children) and `ndsel` (node selection strategy). *Strong branching* proved to be more effective than the default option for `varsel` (variable selection strategy).

`double epagap, epgap`

Absolute and relative MIP gap tolerances. Assuming that an integer feasible solution already exists, optimization is stopped if the absolute/relative difference between the feasible solution value and the LP objective value at the best remaining search tree node is less than the tolerance.

The absolute tolerance was set to `granularity` (see below) instead of 0. The relative difference (10^4 by default) was lowered to its smallest possible value of 10^{-9} since some of our problems have very large optimal values and several near-optimal feasible solutions.

`double objdif`

Absolute objective difference cutoff. A search tree node can be cut off if its LP optimum is within `objdif` of the best feasible value. Set to `granularity` (see below) instead of 0.

C.2 Heuristic parameters

`int dupc_at_loadtime`

TRUE/FALSE. Enable/disable deletion of duplicate columns next to each other in the input file.

`double granularity`

A lower bound on the true granularity of the problem (which is the minimum difference between non-identical integral feasible solution values).

Note that if in a problem all the objective function coefficients have at most k decimal digits then 10^{-k} is a lower bound on the true granularity.

`int what_to_do`

The main function can be used to run only Reduce() (0); solve the input problem as an IP (with CPLEX MIP) or as an LP (1); invoke Reduce() and then solve the IP or LP (2); or run the feasible solution heuristic (3).

`int major_itlim, int minor_itlim`

Iteration limit on the major loop and the heuristic variable fixing loop.

`int ip_or_lp`

Solve the input problem as an IP (0) or as an LP (1) (`what_to_do` is 1 or 2).

`int lp_method`

The LP method to be used is primal simplex (0), dual simplex (1), barrier with primal crossover (2), barrier with dual crossover (3) or barrier without crossover (4).

Note that in the feasible solution heuristic the first LP relaxation is always solved with a barrier method even if simplex is used later on.

`int lp_warmstart`

Setting the last three bits determines what information is used to warmstart the LP solver: basis status (last or 0 bit), primal feasible solution (1 bit) or dual feasible solution (2 bit). Basis status can be used for both simplex and barrier methods; primal and dual solutions are for barrier methods only. LPs are solved from scratch if none of the bits is set.

`int warmstart_advice`

TRUE/FALSE. If enabled, an LP is solved from scratch instead of using warm-start information if too many rows with nonbasic slacks were removed from the formulation since the LP was re-solved.

`int what_rel_cost`

A measure ("relative cost") based on which the significance of variables is compared. The options are original objective function coefficient divided by the size of the column's support (0) and current reduced cost (2).

`int do_crash, double crash_frac`

TRUE/FALSE. If crash is enabled, the heuristic starts with eliminating up to `crash_frac` fraction of the least significant columns.

`int vars_at_one_action, double vars_at_one_ratio`

How to deal with variables at level 1 in the current LP relaxation. For each variable at level 1 we can remove all variables from the symmetric difference of its rows' supports (1); fix the variable to 1 (2); treat it the same way as any other variable at nonzero level (3) or apply an adaptive strategy (4) (where (1) is used if the ratio of variables at level 1 to all variables at nonzero levels is at least `vars_at_one_ratio`, and (3) otherwise).

`double min_coldel_frac`

The LP is re-solved during the heuristic fixing phase if this fraction of the variables have been eliminated (by variable fixing or a subsequent Reduce()).

`int do_followon_fixing`

TRUE/FALSE. Follow-on fixing is enabled/disabled.

`double followon_threshold_ub, double followon_threshold_lb`

Between 0 and 1. Follow-on threshold upper and lower bounds.

`double followon_row_frac, int followon_roworder, int followon_choose`

The rows of the matrix are ordered based on `followon_roworder` (same options as for `roworder`) during follow-on fixing. `followon_row_frac` fraction of the rows are chosen from the top of the ordering; all pairs of the these rows are enumerated and compared (`followon_choose` determines if the row pairs are taken from the bottom (0), top (1) or opposite ends of the ordering (2)).

`int do_process_rows`

TRUE/FALSE. Enable/disable the procedure that removes unattractive variables. The procedure is invoked even if it is disabled when follow-on fixing is not able to remove any variables.

`int roworder`

Determines the order in which the rows of the matrix are enumerated. The options are either random (0), or based on the current dual values corresponding to the rows: from largest dual value to smallest (1), from smallest to largest (2), from largest absolute value to smallest (3) and from smallest absolute value to largest (4).

`int row_action, double threshold, double frac_above_cutoff,`
`double frac_all_zeros`

How to remove unattractive variables from a row. The options are deleting a given fraction (`frac_above_cutoff`) of variables above the cutoff determined by `threshold` (1) or removing a given fraction `frac_all_zeros` of variables at zero level from the row (4).

`double min_procrow_frac`

The procedure of removing unattractive variables terminates if at least this fraction of all the rows have been examined (even if not enough variables were marked for deletion).

`int del_zeros_only`

TRUE/FALSE. If set, no variables at nonzero levels in the current LP relaxation are marked for deletion during the heuristic fixing phase (but could be marked by a subsequent Reduce()).

`int protect_collen`

Any nonnegative integer. If positive, columns with supports up to this size are not marked for deletion during the heuristic fixing phase (but could be marked by a subsequent Reduce()).

`int protect_basic`

TRUE/FALSE. If set, basic variables in the current LP relaxation are not marked for deletion during the heuristic fixing phase (but could be marked by a subsequent Reduce()).

Appendix D

Implementing our Branch-and-Cut procedure

In this Appendix first we describe some COMPSys parameters that were essential for our implementation and experiments. Then the parameters that we introduced in the user functions are listed. The parameters within the two sections are grouped by the processes in which they occur. Chapter 5 describes our Branch-and-Cut implementation using the COMPSys framework. See [EL97] for a complete list of user-written functions and parameters of COMPSys.

D.1 COMPSys parameters

Global parameters

`int verbosity`

Determines the amount of output information (between 0 and 11).

`double granularity`

A lower bound on the true granularity of the problem (which is the minimum difference between non-identical integral feasible solution values).

`double upper_bound`

Upper bound on the optimal value (e.g., the value of a feasible solution).

`int time_limit`

Time limit on the B&C optimization (excluding reading the input and pre-processing/upper bounding in the Master process).

Parameters in the Master process

`int do_branch_and_cut`

TRUE/FALSE. Enable/disable the Tree Manager process.

`int do_draw_graph`

TRUE/FALSE. Enable/disable the DrawGraph process (the GUI).

Parameters in the Tree Manager process

There are parameters that determine the number and names of the processors used for the different processes (this is interesting for instance if we wish to use the same processor for more than one process – as we do with the Master and Tree Manager). Also, there are parameters not listed here that control *diving* (retaining one of the children after branching).

`int max_active_nodes`

Limit on the number of LP – Cut Generator pairs.

`int max_cp_num`

Limit on the number of Cut Pool processes (no CP is used if 0).

`int use_cg`

TRUE/FALSE. Enable/disable cut generation. B&C becomes B&B if cut generation is disabled.

`int node_selection_rule`

Determines which search tree node is selected for processing next. We always used the default option of selecting the node with the lowest lower bound (LP value). Other options include selecting a node with the highest lower bound, or enumerating the search tree in a breath-first or depth-first fashion.

Parameters in the LP process

A set of six parameters that we do not describe here in detail determine when to carry out reduced cost and logical fixing. Also, parameters control how long the LP is going to wait for cuts from CG and CP before re-solving the LP relaxation.

`double tailoff_obj_frac, tailoff_gap_frac`

`int tailoff_obj_backsteps, tailoff_gap_backsteps`

Threshold values and length of history for checking tailing off (5.3.5).

`int branch_on_cuts`

TRUE/FALSE. Branching on cuts is enabled/disabled.

`int max_cutnum_per_iter`

Limit on the number of violated inequalities (cuts) added to the formulation in one iteration.

`int max_presolve_iter`

Limit on the number of dual simplex iterations for presolving the LP relaxations at the would-be children of the branching candidates.

Parameters in the Cut Pool process

`int max_size, max_number_of_cuts`

The size of the memory that can be allocated for the cuts stored and a limit on their number. When the cut pool becomes full (one of these limits is exceeded) ineffective cuts are deleted from the pool.

`int check_which`

Determines which cuts should be checked for violation for a given LP solution. The default is to check those that were originally generated at a higher level in the search tree than the current solution *and* those that were found violated recently.

Parameters in the Cut Generator and DrawGraph processes

There are no parameters in CG that need to be mentioned here. Window characteristics can be set through parameters in DG.

D.2 Parameters in the user-written functions

Parameters in the Master Process

All the parameters discussed here are used in the `user_start_heurs` function.

`int first_reduce`

TRUE/FALSE. If the parameter is set, an initial Reduce() (Chapter 3) will be invoked right after the problem is read in. A complete set of Reduce() parameters (described in Appendix B) can also be specified.

`int first_lp_method`

The LP method to be used to solve the first LP relaxation. The options are primal simplex (0), dual simplex (1), barrier with primal crossover (2), barrier with dual crossover (3) or barrier without crossover (4).

`int rcfix, rcfix_lp_method, rcfix_lp_warmstart`

Setting `rcfix` enables reduced cost fixing. The other two parameters specify the LP method and what warmstart information to use during reduced cost fixing.

`int first_heur`

TRUE/FALSE. Our Feasible Solution Heuristic (Chapter 4) is invoked if the parameter is set. Note that a complete set of parameters (described in Appendix C) can also be specified.

Parameters in the LP Process

`double logfix_frac, do_heur_frac`

Attempt logical fixing (resp. feasible solution heuristic) if at least the above fraction of variables was set to zero since the most recent application of logical fixing (resp. feasible solution heuristic). Logical fixing (and feasible solution heuristic) is always invoked if a variable is fixed to one. A complete set of Reduce() (Appendix B) and Heuristic (Appendix C) parameters can also be specified to be used in these functions.

`int do_lift_in_lp`

TRUE/FALSE. Enable/disable lifting of violated inequalities.

`int followon_branch_num, threshold_branch_num, slackcut_branch_num`
`int variable_branch_num`

The number of branching candidates to be chosen from each each type. If `variable_branch_num` is not positive then the branching candidates are supplemented with branching variables until their number reaches 5.

`int branch_var_close_to_half`

TRUE/FALSE. If the parameter is set, branching variables are chosen with the "close to half and expensive" rule; otherwise they are chosen with the "close to one and cheap" rule.

`double threshold_branch_threshold,`

`double followon_branch_lowthreshold, followon_branch_highthreshold`

Thresholds for threshold and follow-on branching candidate selection, as described in Section 5.3.6. Between 0 and 1.

Parameters in the Cut Generator Process

`int do_human_cg, handmake_cuts_if_must`

TRUE/FALSE. The first parameter enables/disables cut generation through the GUI. The second parameter determines how frequently the fractional graph is displayed: only when the built-in cut generators fail to find a violated inequality or every time the LP is re-solved.

`int do_scl, do_rcl, do_oh, do_packing, do_cover, do_oah`

TRUE/FALSE. Enable/disable the corresponding cut generators (star clique, row clique, sequentially lifted odd holes, packing odd holes, cover odd holes and sequentially lifted odd antiholes.

`double scl_min_violation, rcl_min_violation, oh_min_violation`

`double packing_min_violation, cover_min_violation, oah_min_violation`

Inequalities of a given type are not considered violated if their violation is below the minimum.

`int scl_degree_threshold, rcl_degree_threshold`

Greedy clique detection is substituted for enumeration on a subset of the nodes if the number of nodes exceeds these thresholds (star and row clique routines).

`int scl_which_node`

Determines how to choose the next node in the star clique method. The options are choosing a node with minimum degree, with maximum degree, or with the highest fractional value.

`int oh_max_hubnum, oah_max_hubnum`

Limit on the number of hub candidates in the odd hole and odd antihole detection routines.

`int oh_max_checked_level,`

`int oh_next_level_when_found, next_level_graph_when_found`

The first parameter is the deepest level in the level graph that we investigate in the odd hole detection routine (the level graph is not built below this level). The last two parameters are TRUE/FALSE, they indicate whether the next level/next level graph should be taken when a violated inequality is found using nodes on the current level. Same for `oah`.

Bibliography

[AGPT91] R. Anbil, E. Gelman, B. Patty, and R. Tanga. Recent advances in crew-pairing optimization at American Airlines. *Interfaces*, 21(1):62–74, 1991.

[Anb] R. Anbil. Private communication.

[BC96] J.E. Beasley and P.C. Chu. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94:392–404, 1996.

[BGKK97] R. Borndörfer, M. Grötschel, F. Klostermeier, and Ch. Küttner. Telebus Berlin: Vehicle Scheduling in a Dial-a-Ride System. Technical Report SC 97-23, ZIB, 1997.

[Bix98] R. Bixby. Recent developments in CPLEX. Joint DIMACS-CMU-Georgia Tech. Workshop on Large Scale Discrete Optimization, May 27-29, 1998.

[BM94a] F. Barahona and A.R. Mahjoub. Compositions of graphs and polyhedra II: Stable sets. *SIAM Journal on Discrete Mathematics*, 7:359–371, 1994.

[BM94b] F. Barahona and A.R. Mahjoub. Compositions of graphs and polyhedra III: Graphs with no W_4 minor. *SIAM Journal on Discrete Mathematics*, 7:372–389, 1994.

[Bor97] R. Borndörfer. *Aspects of Set Packing, Partitioning, and Covering.* PhD thesis, Technischen Universität Berlin, December 1997.

[BP76] E. Balas and M.W. Padberg. Set partitioning: A survey. *SIAM Review*, 18(4):710–760, 1976.

[BQ64] M.L. Balinski and R.E. Quandt. On an integer program for a delivery problem. *Operations Research*, 12(2):300–304, 1964.

[CC97] E. Cheng and W.H. Cunningham. Wheel inequalities for stable set polytopes. *Mathematical Programming*, 77:389–421, 1997.

[Chr85] N. Christofides. Vehicle routing. In E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, chapter 12. Wiley, New York, 1985.

[Chv73] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305–337, 1973.

[Chv75] V. Chvátal. On certain polytopes associated with graphs. *Journal of Combinatorial Theory (B)*, 18:138–154, 1975.

[CPX95] CPLEX Optimization, Inc. *Using the CPLEX© Callable Library, Version 4.0*, 1995.

[ECTA96] M. Eben-Chaime, C.A. Tovey, and J.C. Ammons. Circuit partitioning via set partitioning and column generation. *Operations Research*, 44(1):65–76, 1996.

[Edm62] J. Edmonds. Covers and packings in a family of sets. *Bulletin of the American Mathematical Society*, 68:494–499, 1962.

[EDM90] E. El-Darzi and G. Mitra. Set covering and set partitioning: A collection of test problems. *Omega International Journal of Management Science*, 18(2):195–201, 1990.

[EL97] M. Eső and L. Ladányi. *CompSys User's Manual (unpublished)*, 1997.

[ELRT97] M. Eső, L. Ladányi, T.K. Ralphs, and L.E. Trotter. Fully parallel generic branch-and-cut. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, March 14-17 1997.

[FR87] J.C. Falkner and D.M. Ryan. A bus crew scheduling system using a set partitioning model. *Asia-Pacific Journal of Operational Research*, 4:39–56, 1987.

[Ger89] I. Gershkoff. Optimizing flight crew schedules. *Interfaces*, 19(4):29–43, 1989.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman and Company, New York, 1979.

[GLS88] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization.* Springer-Verlag, 1988.

[GN70] R.S. Garfinkel and G.L. Nemhauser. Optimal political districting by implicit enumeration techniques. *Management Science*, 16:B495–B508, 1970.

[GN72] R.S. Garfinkel and G.L. Nemhauser. *Integer Programming.* Wiley, New York, 1972.

[Hoc95] D. Hochbaum. Approximating covering and packing problems: set cover, vertex cover, independent set and related problems. In D. Hochbaum, editor, *Approximation Algorithms for NP-hard problems*, chapter 3. PWS Publishing Company, Boston, 1995.

[HP93] K.L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[Kop99] L. Kopman. *A New Generic Separation Routine and its Application in a Branch and Cut Algorithm for the Capacitated Vehicle Routing Problem, in preparation.* PhD thesis, Cornell University, Ithaca, NY, 1999.

[Lad96] L. Ladányi. *Parallel Branch and Cut and its Application to the Traveling Salesman Problem.* PhD thesis, Cornell University, Ithaca, NY, January 1996.

[LK79] J.K. Lenstra and A.H.G. Rinnooy Kan. Computational complexity of discrete optimization problems. *Annals of Discrete Mathematics*, 4:121–140, 1979.

[LR] L. Ladányi and T.K. Ralphs. Private communication.

[LS90] L. Lovász and A. Schrijver. Matrix cones, projection representations and stable set polyhedra. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1:1–17, 1990.

[NS89] P. Nobili and A. Sassano. Facets and lifting procedures for the set covering polytope. *Mathematical Programming*, 45:111–137, 1989.

[NS92] G.L. Nemhauser and G. Sigismondi. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operational Research Society*, 43(5):443–457, 1992.

[NT74] G.L. Nemhauser and L.E. Trotter. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6:48–61, 1974.

[NT75] G.L. Nemhauser and L.E. Trotter. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, New York, 1988.

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Pad73] M.W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[Pan] G. Pangborn. Private communication.

[PVM] Pvm: Parallel virtual machine.
Home page: `http://www.epm.orml.gov/pvm/pvm_home.html`.

[PY91] C.H. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[Ral95] T.K. Ralphs. *Parallel Branch and Cut for Vehicle Routing.* PhD thesis, Cornell University, Ithaca, NY, May 1995.

[RF87] D.M. Ryan and J.C. Falkner. A bus crew scheduling system using a set partitioning model. *Asia Pacific Journal of Operational Research*, 4:39–56, 1987.

[Sas89] A. Sassano. On the facial structure of the set covering polytope. *Mathematical Programming*, 44:181–202, 1989.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, New York, 1986.

[Shm95] D.B. Shmoys. Computing near-optimal solutions to combinatorial optimization problems. In W. Cook, L. Lovász, and P. Seymour, editors, *Advances in Combinatorial Optimization*, pages 355–397. AMS, Providence, RI, 1995.

[Tro75] L.E. Trotter. A class of facet producing graphs for vertex packing polyhedra. *Discrete Mathematics*, 12:373–388, 1975.

[Wel95] M. Welsh. Using Tcl and Tk from your C programs. *Linux Journal*, pages 26–33, February 1995.