GOMORY CUTTING PLANE ALGORITHM USING EXACT ARITHMETIC

By

Kristin Farwell

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Mathematics

Approved by the
Examining Committee:

John Mitchell, Thesis Adviser

Michael Kupferschmid, Member

Joseph Ecker, Member

Kristin Bennett, Member

Aparna Gupta, Member

Rensselaer Polytechnic Institute
Troy, New York

January 17, 2006
(For Graduation May 2006)

GOMORY CUTTING PLANE ALGORITHM USING EXACT ARITHMETIC

By

Kristin Farwell

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Mathematics

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

John Mitchell, Thesis Adviser
Michael Kupferschmid, Member
Joseph Ecker, Member
Kristin Bennett, Member
Aparna Gupta, Member

Rensselaer Polytechnic Institute
Troy, New York

January 17, 2006
(For Graduation May 2006)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In the field of Operations Research (OR) we solve problems dealing with optimization. One of the most studied problems in OR is linear programming problems. These are problems that are maximization or minimization of a linear objective function subject to linear constraints. If we have the added burden of having restrictions on some of the variables that must also be integral then we have a Mixed Integer Programming Problem. If all of the variables must be integral then this is a Pure Integer Programming Problem. These are the types of problems that we are going to be studying more in depth. One method used to solve Integer Programming Problems are known as cutting planes. There are many different types of cutting planes. One type of cutting plane is known as Gomory cutting planes. Gomory cutting planes have been studied in depth and utilized in various commercial codes. We will show that by using exact arithmetic rather than floating point arithmetic, we can produce better cuts. The main reason for this is the addition of slack variables to the Gomory inequalities. For exact arithmetic, this slack variable has to itself be integral whereas for floating point arithmetic, the slack could be a floating point number.

What we have done is write an exact arithmetic simplex program which incorporates some of the recent advances like LU factorization of the basis matrix and steepest edge pivoting rule. We have explored the size of the certain key vectors in the simplex algorithm. We also wrote code that finds the all of the various forms of Gomory cutting planes in exact arithmetic and adds them to the simplex tableau and reoptimizes using dual simplex. We also wrote code that drops certain Gomory cuts when they are not useful anymore. We explored the size of the Gomory cutting planes and how it relates to the size of vectors in the simplex algorithm. Since the majority of the time is spent reoptimizing, we wanted to find a way to utilize the power of CPLEX solver and our exact Gomory cutting planes.

Another use for exact arithmetic is when we have dual degeneracy or in other words, alternate optima for two dimensional problems. We pivoted to these alter-

nate optima and cut off this additional solution which reduced the number of total Gomory iterations by performing just one extra simplex pivot.

CHAPTER 1
Introduction

Solution methods for integer programming problems have been looked at for years. Many approaches have been developed. Enumeration, cutting planes, and branching techniques are a few of the methods used to solve general integer programming (IP) problems. However before we delve into the world of integer programming, we must start at the beginning and discuss linear programming (LP) problems which are defined in Definition 1.

Definition 1 *All linear programming problems can be modelled in the following form*

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & Ax & = & b \quad (LP) \\ & x & \geq & 0 \end{array}$$

where $c, x \in \mathbb{R}^n, b \in \mathbb{R}^m$, *and* $A \in \mathbb{R}^{m \times n}$.

Linear programming problems have been studied extensively. For a more in depth study of the theory of LP's, the reader is referred to [6]. A wide variety of solution techniques exists for LP's such as the simplex algorithm, the ellipsoid algorithm, and interior point methods. The Simplex algorithm and interior point methods work extremely well and are incorporated in many commercial software packages such as CPLEX, AMPL, COIN-OR, and others.

Before we get too far, let us look at an example to illustrate why there is a need to solve integer programming problems. Suppose that we have a knapsack that can hold up to 13 pounds and we have the books from table 1.1 to choose from. We want to take the most important books from Table 1.1 and still stay under the weight limit of the knapsack.

The problem is formulated as the following binary programming problem:

type	size	value
Chemistry	4	9
Physics	7	10
Calculus	4	5
French	2	2

Table 1.1: Size and Value of Books to be put in Knapsack

Example 1

$$\begin{array}{lrcrcrcrcl} \max & 9x_1 & + & 10x_2 & + & 5x_3 & + & 2x_4 & & \\ \text{s.t.} & 4x_1 & + & 7x_2 & + & 4x_3 & + & 2x_4 & \leq & 13 \\ & x_i & \in & \{0,1\} & & & & & & \end{array}$$

One possible way to solve this problem is to look at every possible combination of books. For this problem we need to check 16 possible combinations of books. If n were the number of books then we need to check 2^n possible combinations. This is a few more than I would want to check for a library. In general the knapsack problem is difficult to solve so we relax the part that is causing the problem, i.e. the fact that we are only allowed to have the variables to be either 0 or 1. Since we are allowing the variables to take on values between and including 0 and 1, our model becomes

$$\begin{array}{lrcrcrcrcl} \max & 9x_1 & + & 10x_2 & + & 5x_3 & + & 2x_4 & & \\ \text{s.t.} & 4x_1 & + & 7x_2 & + & 4x_3 & + & 2x_4 & \leq & 13 \\ & 0 & \leq & x_i & \leq & 1 & & & & \end{array}$$

This model is called the linear programming relaxation. The solution to this LP relaxation is:

$$x_1 = 1 \quad x_2 = 1 \quad x_3 = \tfrac{1}{2} \quad x_4 = 0$$

This solution suggests that we only take half of our Calculus book. Clearly this is not acceptable. We need a way to allow two out of these books instead of

two and a half. Notice that if we add the constraint

$$x_1 + x_2 + x_3 \leq 2 \tag{1.1}$$

then the solution above is no longer valid and it is cutoff. This constraint is valid since the sum of the weights of those books is 15 pounds which is larger than the weight limit of 13 pounds. We can only include two of those books listed. This constraint is known as a cutting plane, in particular, a set covering cutting plane. With constraint (1.1) added to the LP relaxation we have the following formulation:

$$\begin{array}{lrcrcrcrcl} \max & 9x_1 & + & 10x_2 & + & 5x_3 & + & 2x_4 & & \\ \text{s.t.} & 4x_1 & + & 7x_2 & + & 4x_3 & + & 2x_4 & \leq & 13 \\ & x_1 & + & x_2 & + & x_3 & & & \leq & 2 \\ & 0 & \leq & x_i & \leq & 1 & & & & \end{array}$$

The solution obtained to this new linear programming problem is:

$$x_1 = 1 \quad x_2 = 1 \quad x_3 = 0 \quad x_4 = 1$$

This solution tells us that the best books to put into the knapsack are the Chemistry, Physics, and French books. This combination of books has a value of 21 and the combined weight of only 13 pounds. So the constraint of binary programming problem has also been satisfied without resorting to cutting any books in half. These set covering inequalities can be used to help solve the knapsack problem. However, even with these special cutting planes, general integer programming problems (IP) and even the knapsack problem are NP-hard problems. In Cook's paper [7], he shows that satisfiability is NP-Complete and as such was the first problem shown to have this property. NP-Complete problems are feasibility problems that cannot be solved in polynomial time unless P=NP. Determining whether P=NP is one of the Millennium Challenge problems [8]. NP-hard is defined in Definition 2.

Definition 2 *A problem is considered NP-hard if there is an NP-complete problem that can be polynomially reduced to it. [20]*

Even though IP's are NP-hard, we still have a need to model certain problems like the knapsack problem that involves solutions that only have integer variables. Airplane crew scheduling, tickets for sporting events, and car rental problems are examples that contain certain variables that need to be integer. It would not be necessary to have a cockpit that has 4.45823 seats for the pilots. We would need either 4 or 5 seats. Consequently, solution techniques still need to be improved upon. Therefore this paper is focused on solutions to general integer programming problems which are formulated in Definition 3.

Definition 3 *General integer programming problems can be modelled as*

$$\begin{array}{lrcll} \min & c^T x & & & \\ \text{s.t.} & Ax & = & b & (IP) \\ & x & \in & \mathbb{Z}^+ & \end{array}$$

where $c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$, *and* $x \in \mathbb{Z}^+ = \{x : x \geq 0 \text{ and } x \in \mathbb{Z}\}$

Currently, one of the best ways found to solve integer programming problems is known as branching. Branching splits up the problem into two different sub-problems by choosing a variable that is not integer in the LP relaxation and allowing this variable to take on the value of the integer greater than it and the value of the integer less than it. For example, if $x_2 = 5.5$, then split this variable into two different sub-problems by allowing $x_2 \geq 6$ in one and $x_2 \leq 5$ in the other. Branching can be combined with various techniques in an attempt to get to the solution of a general integer programming problem, i.e. Branch-and-Price, Branch-and-Bound, and Branch-and-Cut. If we concentrate on one of these techniques and improve upon it then we can improve the combined branching technique.

This thesis focuses on a cutting plane technique, in particular Gomory cutting planes. In general, the goal of a cutting plane is to cut off the optimal solution to the linear programming relaxation and in so doing a part of the linear programming problem's feasible region without cutting any of the feasible integer solutions. In Example 1, we saw how a cutting plane can be used to solve a simple knapsack problem. Recent work on Gomory cutting planes is detailed in Chapter 3 and

studied in articles [21] and [18].

A cutting plane is generated from the LP relaxation solution of the integer programming problem. If the LP relaxation solution is not integral then a cutting plane is generated by cutting off this solution. For simple two-dimensional examples, cutting planes can be generated graphically. Example 2 shows how cutting planes can be found graphically.

Example 2

$$\begin{array}{lrrrrrl} \min & - & 6x_1 & - & 5x_2 & & \\ \text{s.t.} & & 3x_1 & + & x_2 & \leq & 11 \\ & - & x_1 & + & 2x_2 & \leq & 5 \\ & & x_1 & , & x_2 & \in & \mathbb{Z}^+ \end{array}$$

This IP can be solved using cutting planes. We will relax it by ignoring the integrality of the variables. With this in mind we solve the following problem:

$$\begin{array}{lrrrrrl} \min & - & 6x_1 & - & 5x_2 & & \\ \text{s.t.} & & 3x_1 & + & x_2 & \leq & 11 \\ & - & x_1 & + & 2x_2 & \leq & 5 \\ & & x_1 & , & x_2 & \geq & 0 \end{array}$$

The optimal solution to the linear programming relaxation is $(2\frac{3}{7}, 3\frac{5}{7})$ which is straightforward to see either graphically (see Figure 1.1) or by way of the simplex algorithm. Since this solution is not integral, cutting planes can be generated, specifically the cutting plane,

$$x_1 + x_2 \leq 6 \tag{1.2}$$

as seen in red in Figure 1.1.

This cutting plane is obtained by taking $\frac{3}{7}$ of the first constraint and $\frac{2}{7}$ of the second and rounding appropriately.

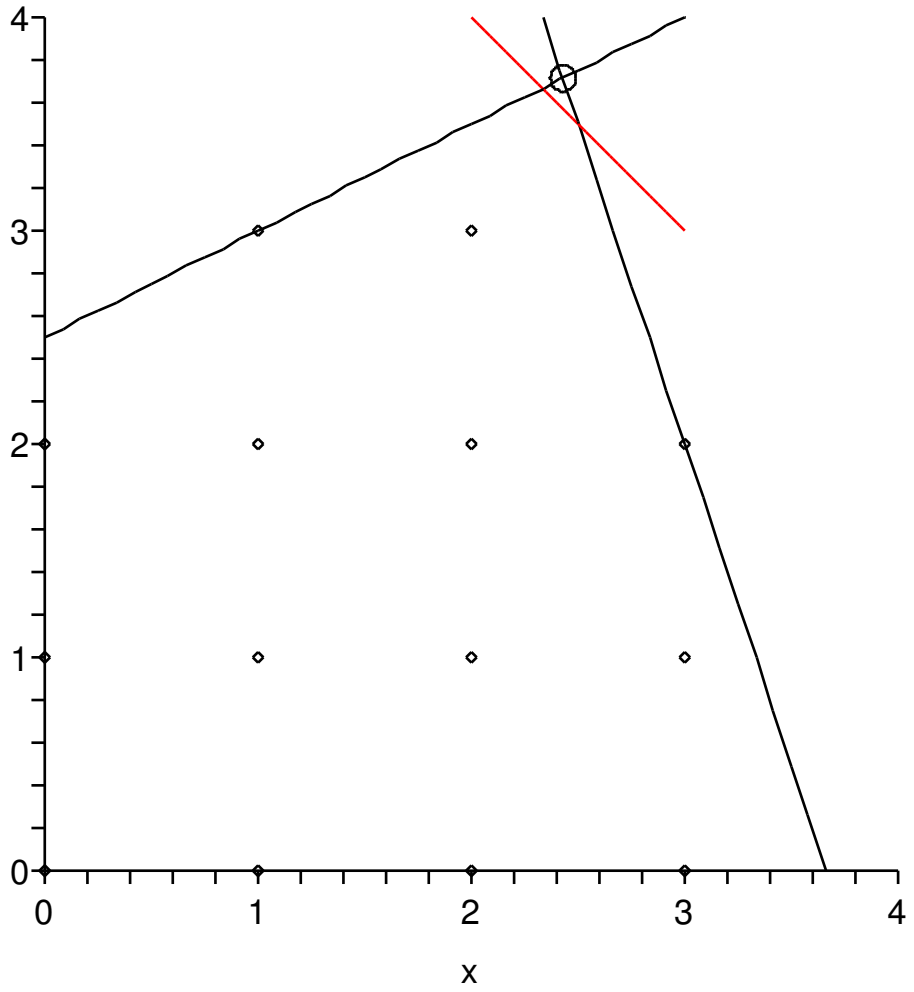

Figure 1.1: A weak cutting plane on a two dimensional integer programming problem found in [19]

$$\frac{3}{7}(3x_1 + x_2 \leq 11) + \frac{2}{7}(-x_1 + 2x_2 \leq 5)$$
$$x_1 + x_2 \leq \frac{43}{7}$$
$$x_1 + x_2 \leq 6$$

Graphically this cut is not a particularly strong cut because it fails to remove a large portion of the feasible region. Better cuts would be ones that remove the largest portion of the feasible region without removing any integer points. The best cutting planes would be the cutting planes generated from the convex hull of the

integer points. Convex hulls are defined in Definition 4.

Definition 4 *Given a set $X \subseteq \mathbb{R}^n$, the convex hull of X is defined as*

$$conv(X) = \left\{ x : x = \sum_{i=1}^{t} \lambda_i x^i \ni \sum_{i=1}^{t} \lambda_i = 1, \ with \ \lambda_i \geq 0 \ for \ i = 1, ..., t \right\} \quad (1.3)$$

where $\{x^1, ..., x^t\} \in X$. [21] According to [6], Carathéodory showed $t \leq n + 1$.

Some examples of better cutting planes which are suitable for the first example are:

$$x_1 + x_2 \leq 5 \quad (1.4)$$

$$x_2 \leq 3 \quad (1.5)$$

$$-x_1 + x_2 \leq 2 \quad (1.6)$$

$$x_1 \leq 3 \quad (1.7)$$

These particular cutting planes are significant since they form the convex hull of the feasible integer points as shown in Figure 1.2. Actually each of these cutting planes is a facet of the convex hull of the feasible integer points. Facets are defined below in Definition 16, however the following definitions need to be developed first.

Definition 5 *A set $S \in \mathbb{R}^n$ is a subspace of $\mathbb{R}^n$ if every linear combination of points in S is also in S.*

Definition 6 *A point $z \in \mathbb{R}^n$ is an affine combination of x and y if $z = \lambda x + (1-\lambda)y$ for some $\lambda \in \mathbb{R}$.*

Definition 7 *A set M is affine if every affine combination of points in M is also in M.*

Definition 8 *The points $a^1, \ldots, a^k$ are affinely independent if the vectors $a^2 - a^1, a^3 - a^1, \ldots, a^k - a^1$ are linearly independent.*

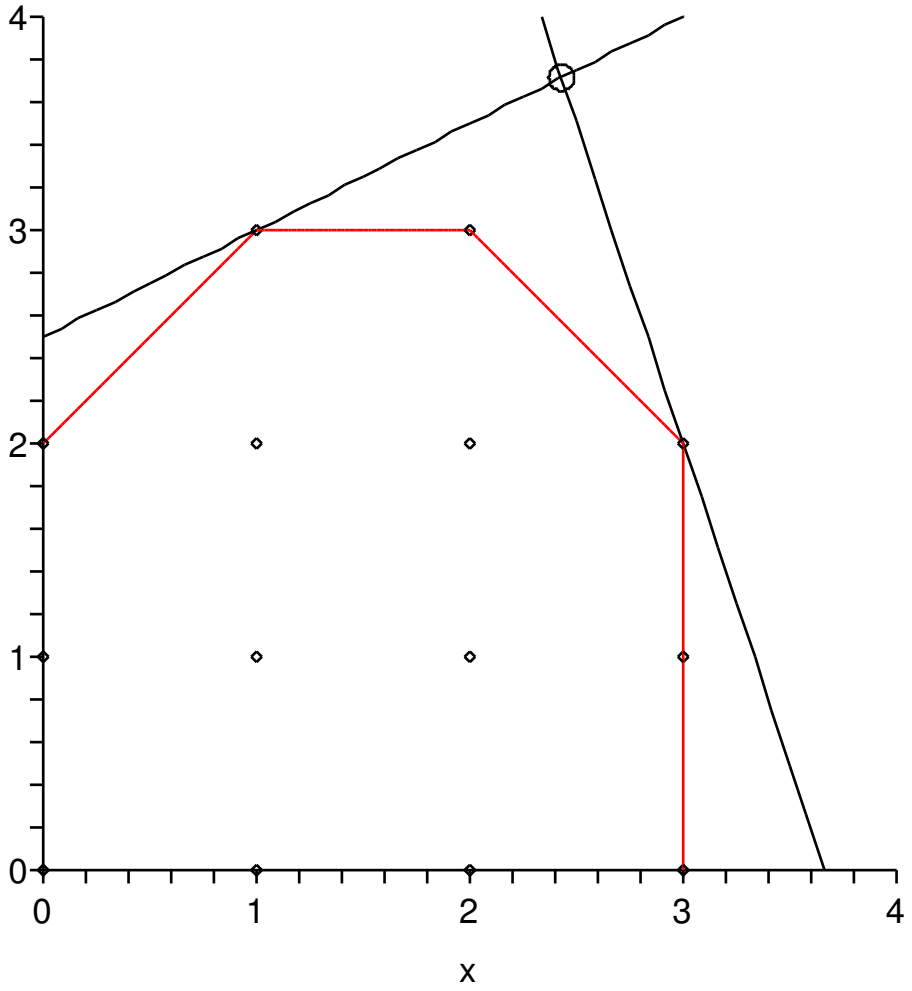

Figure 1.2: Convex Hull of the two dimensional integer programming problem found in [19]

Definition 9 *Given a scalar α and a vector $a \in \mathbb{R}^n$, the set $\{x : a^T x \geq \alpha\}$ is a halfspace.*

Definition 10 *A polyhedron is a finite intersection of halfspaces.*

The feasible region of a linear program is a polyhedron since all of the constraints are halfspaces.

Definition 11 *The dimension of a subspace is the maximum number of linearly independent vectors in it.*

Definition 12 *The dimension of an affine space is the dimension of the corresponding subspace.*

Definition 13 *The affine hull of a set is the set of all affine combinations of points in the set.*

Definition 14 *The dimension of a polyhedron is the dimension of its affine hull.*

Definition 15 *Let P be a polyhedron. Let H be the hyperplane $H := \{x : a^T x = \alpha\}$. Let $Q = P \cap H$. If $a^T x \geq \alpha$ for all $x \in P$ then Q is a face of P.*

Definition 16 *Let P be a polyhedron of dimension d. A face of dimension $d-1$ is a facet. A face of dimension 1 is an edge. A face of dimension 0 is a vertex.*

Facets are extremely important in solving IP's. If the linear programming relaxation is solved with the facets of the integer points included then the optimal solution would also be optimal for the IP. For example if the constraints (1.4)-(1.7) were added to the LP relaxation of Example 2 and solved using the simplex algorithm, the optimal solution to this new LP is also the solution to the IP. This is because the simplex algorithm finds extreme point solutions to LP's and all of the extreme points are integral points. These particular facet defining cuts were easy to find graphically for this example. However, in the more general cases they are extremely hard to find.

We have noticed that we can generate these cutting planes logically as in Example 1 or graphically as in Example 2. In both of these examples, we needed the solution to the LP relaxation of the integer programming problem. Thus, we need to have a way to solve the linear programming problems.

CHAPTER 2
Simplex

The cutting planes in the previous chapter were developed both graphically and logically. This is simple enough for the examples found in the first chapter, however this is not acceptable for larger problems. A more general method needs to be developed. Gomory suggested using specific cutting planes from the optimal form of the LP relaxation using the simplex tableau. We feel that using exact Gomory cutting planes could produce better cutting planes than the standard floating point arithmetic. We must first discuss the simplex algorithm since we need to find solutions to linear programming problems.

2.1 Primal Simplex Algorithm

In this work, we decided to use an implementation of the revised simplex algorithm found in [11]. Refer to [6] regarding the theory of the revised simplex algorithm and [3] for more information on the progress of LP solvers. The LP solver developed in this research does not implement every possible advancement. However, we have tried to incorporate some of the recent advancements. The advancements that we have included are sparse matrix storage of the simplex tableau, PLU factorization of the basis matrix, the use of eta matrices, and steepest edge pivoting rules. We have implemented both simplex and dual simplex. There are some exact simplex solvers given out freely. One is exlp found at web address http://members.jcom.home.ne.jp/masashi777/exlp.html which uses LU factorization and steepest edge pivoting rule for the dual simplex algorithm. When we started this project, we attempted to use their code but it had some bugs. They claim that their bugs have been worked out as of October 2005.

A discussion of the basic simplex algorithm is needed. Recall the general linear programming problem formulation from Definition 1 is known as primal formulation:

$$\begin{array}{lrcll} \min & c^T x & & & \\ \text{s.t.} & Ax & = & b & (P) \\ & x & \geq & 0 & \end{array}$$

and the dual to this LP formulation is:

$$\begin{array}{lrcll} \max & b^T y & & & \\ \text{s.t.} & A^T y & \leq & c & (D) \\ & y & free & & \end{array}$$

From this primal and dual formulation, we have the following theorem:

Theorem 1 *Strong Duality Theorem: Suppose (P) is feasible with finite optimal value. Then (D) is feasible and there exists solution (x^*, y^*) such that x^* is primal feasible and y^* is dual feasible with $c^T x^* = b^T y^*$.*

Another vital concept to LP's is the idea of complementary slackness which is found in the next theorem:

Theorem 2 *Complementary Slackness: A pair of primal and dual feasible solutions are optimal if and only if whenever $x_i > 0$ then $s_i = 0$ and if $s_i > 0$ then $x_i = 0$, where $s = c - A^T y$.*

More details on LP's and proofs to these two theorems can be found in [6].

Given the primal formulation in (P) we can separate the variables into two groups, the basic and the nonbasic variables.

Definition 17 *Let the set of columns (variables) that are basic form the matrix $\boldsymbol{B}$ and the set of columns (variables) that are nonbasic form the matrix $\boldsymbol{N}$.*

Therefore, we can write the original LP into the following form.

$$\begin{array}{lrcrl} \min & c_B^T x_B & + & c_N^T x_N & \\ \text{s.t.} & Bx_B & + & Nx_N & = b \\ & x_B & , & x_N & \geq 0 \end{array}$$

Since B is a square and nonsingular matrix as shown in [6], B is comprised of the columns of A that solve the equation $Bx_B = b$ with $x_N = 0$. Thus, we can write the above LP formulation into the following form by multiplying the constraints by B^{-1}.

$$\begin{array}{lrcrcl} \min & c_B^T x_B & + & c_N^T x_N & & \\ \text{s.t.} & x_B & + & B^{-1}Nx_N & = & B^{-1}b \\ & x_B & , & x_N & \geq & 0 \end{array}$$

When the constraints are solved for x_B and then substituting them into the objective function and simplifying yields:

$$\begin{array}{lrcrcll} \min & c_B^T B^{-1}b & + & (c_N^T - c_B^T B^{-1}N)x_N & & & \\ \text{s.t.} & x_B & + & B^{-1}Nx_N & = & B^{-1}b & (LP) \\ & x_B & , & x_N & \geq & 0 & \end{array}$$

Since $x_N = 0$, then if a coefficient in front of one of the variables contained in x_N is strictly negative then any increase in this variable will cause the objective value to decrease. This vector of coefficients is called the reduced cost which is defined as:

Definition 18 *The reduced cost is*

$$\overline{c}_N^T = c_N^T - c_B^T B^{-1}N \tag{2.1}$$

When all of the reduced costs are non-negative then the tableau is in optimal form because none of the non-basic variables can be changed from zero to cause a decrease in the objective value. Once the non-basic variable is chosen then we need to choose a basic variable to leave. This leaving variable should be chosen in such a way as to maintain primal feasibility. This is accomplished by performing the minimum ratio test.

Definition 19 *Minimum ratio test is*

$$\min\left\{\frac{B^{-1}b_i}{B^{-1}N_{ij}} : B^{-1}N_{ij} > 0\right\} \tag{2.2}$$

where i,j is the row, column respectively.

If all of the $B^{-1}N_{ij} \leq 0$ then we can make this non-basic variable as large as we want to. Thus the problem is unbounded. Now we perform a Gaussian pivot on this element of the tableau. Once the tableau is pivoted, we continue in this manner of choosing negative reduced costs and performing the minimum ratio test until we are either optimal or unbounded. Every simplex algorithm must contain these two components however there are many choices as to how to implement these.

The basic simplex algorithm for solving LP's is summarized in Figure 2.1.

Step 0.	Inputs x_B, c_N, B, N
Step 1.	Choose a negative reduced cost. This is the pivot column.
Step 2.	Use the minimum ratio test to find the pivot row.
Step 3.	Pivot in this row and column.
Step 4.	If optimal, stop.
Step 5.	Else go to Step 2.

Figure 2.1: Basic Simplex Algorithm

There are a couple of issues that still needs to be addressed. First the basic simplex algorithm can contain cycling so we must make appropriate choices regarding the reduced cost and minimum ratio in order to avoid this. Bland's Rule is an example of an anti-cycling choice.

Definition 20 *Bland's Rule or smallest subscript rule is to choose the smallest subscript for the entering and leaving variables. The simplex algorithm terminates if Bland's Rule is used [5].*

This choice will guarantee that simplex will terminate but this is not usually the best option. Cycling is a rare occurrence so this option is usually used when one detects cycling in the algorithm and then one switches back to another rule. For examples of cycling see [9].

As stated before we need to decide on how we are going to choose a negative reduced cost. One possible way to choose a negative reduced cost is to choose the most negative. This choice is very easy to implement and causes a pretty good chance of decreasing the objective value. There is also the option of the greatest

decrease. In this option you choose the leaving and entering variables that will cause the greatest amount of decrease to the objective value. This involves substantial amount of work since you have to check every possible combination. However, the option that we have chosen to implement is the steepest edge pivoting rule. This does not involve as much work as the greatest decrease but it still yields good results. It yields good enough results that it is the most common choice for commercial solvers like CPLEX.

The steepest edge pivoting rule involves finding the reduced cost that corresponds to the hyperplane that forms the steepest edge with the objective. This is accomplished by finding the 2-norm of the nonbasic columns.

Let

$$\eta_i = -B^{-1}n_i. \tag{2.3}$$

where n_i is the i^{th} column of the nonbasic matrix.

Then the steepest edge norm is computed by

$$\gamma_i = \|\eta_i\|_2^2 + 1 \tag{2.4}$$

for $i \in \boldsymbol{N}$.

Once we have these steepest edge norms, we find the best steepest edge reduced cost by dividing the reduced cost by the steepest edge norms and this is the column that will become basic.

$$jp = \arg\max \left\{ \frac{c_{N_k}^2}{\gamma_k} : c_{N_k} < 0, k \in \boldsymbol{N} \right\} \tag{2.5}$$

So, x_{jp} is the entering variable. For more in depth information on steepest-edge pivoting rules refer to [4] for dual steepest edge, and [11] for primal steepest edge.

Another problem with the basic simplex algorithm is the input of the basic variables. Sometimes the initial set of basic variables is a difficult thing to require to be given. Most of the time we are not given the initial basis so we must find it. One way to remedy this is to use the artificial problem. When the artificial problem

is solved then either we have found an initial feasible basis or we have found the original LP to be infeasible.

The artificial problem is a linear program that looks similar to the original linear program but with a new objective function and the constraints have been changed slightly.

$$\begin{array}{lrcccl} \min & & & e^T s & & \\ \text{s.t.} & Ax & + & s & = & b \quad (AP) \\ & x & , & s & \geq & 0 \end{array}$$

where e is a vector of ones and $s \in \mathbb{R}^m$. If the solution to this problem is $s = 0$ then there exists an initial feasible point. Note that $s = 0$ is the same as having the objective value equal to zero. Once this solution is found then the variables in the LP can be separated into two camps, the basic and non-basic variables.

Now that we have a set of variables that are basic we have a B that is nonsingular. However, finding the explicit form of the inverse (EFI) directly is extremely costly. So we need some way to represent B^{-1}. One way found in [9] is to keep track of the eta matrices that when multiplied by B would give the pivoted tableau. We have chosen instead to use a combination of eta matrices and an LU factorization with partial pivoting of the basis matrix.

First we need to find a P, L, and a U matrix that satisfies equation 2.6 where P is a permutation matrix, L is a lower triangular matrix and U is an upper triangular matrix.

$$PB = LU \tag{2.6}$$

We decided to use the representation found in [2]. This formulation is extremely useful because P only needs a vector to store its elements. L and U can both be stored in the same matrix since there is only ones on the diagonal of L. So the LU matrix looks like this

$$LU = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1m} \\ l_{21} & u_{22} & u_{23} & \cdots & u_{2m} \\ l_{31} & l_{32} & u_{33} & \cdots & u_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{m1} & l_{m2} & l_{m3} & \cdots & u_{mm} \end{bmatrix}$$

Once we have the factorization of B in the form in equation (2.6), we can solve the system of equations below:

$$Bx = b \tag{2.7}$$

If we pre-multiply equation (2.7) by P then we have

$$PBx = Pb \tag{2.8}$$

Since $PB = LU$, we can use the LU factorization of PB and perform the following steps to solve for x.

$$\begin{aligned} PBx &= Pb && (2.9) \\ LUx &= Pb && (2.10) \\ \text{Solve } Ly &= Pb && (2.11) \\ \text{Solve } Ux &= y && (2.12) \end{aligned}$$

A similar process can be used to solve the system of equations:

$$B^T x = b \tag{2.13}$$

This requires the fact that P is an orthogonal matrix which means that $P^T P = I$ and multiplying B^T by I.

$$B^T P^T P x = b \tag{2.14}$$

$$(PB)^T P x = b \tag{2.15}$$

$$(LU)^T P x = b \tag{2.16}$$

$$U^T L^T P x = b \tag{2.17}$$

$$\text{Solve } U^T y = b \tag{2.18}$$

$$\text{Solve } L^T z = y \tag{2.19}$$

$$\text{Multiply } x = P^T z \tag{2.20}$$

When we are solving the systems of equations $Bx = b$ and $B^T x = b$, we need to solve systems of the form $Lx = b$, $Ux = b$, $L^T x = b$, and $U^T x = b$. Notice that when we have to solve $U^T x = b$ and $L^T x = b$, that U^T and L^T is similar to L and U respectively except the diagonal. So just a minor change needs to be made to take this into account. Therefore we will only discuss $Lx = b$ and $Ux = b$. Let us first suppose we are solving $Lx = b$. The solution to this is called forward solving because to solve this system you solve for x_1 and then from there you can get the solution for x_2 and so on. So your final looks like

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 - l_{21}x_1 \\ b_3 - l_{31}x_1 - l_{32}x_2 \\ \vdots \\ b_m - l_{m1}x_1 - l_{m2}x_2 \cdots - l_{m,m-1}x_{m-1} \end{bmatrix}$$

Now if we are given $Ux = b$ then we solve it using backward solving. It is known as backward solving because you have to find x_m first which is just $\frac{b_m}{u_{mm}}$ then you use x_m to get x_{m-1} and so on until you reach x_1. So your solution looks like

$$\begin{bmatrix} x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \\ x_m \end{bmatrix} = \begin{bmatrix} \frac{b_{m-2}\cdots - u_{1,m-2}x_{m-2} - u_{1,m-1}x_{m-1} - u_{1,m}x_m}{u_{11}} \\ \vdots \\ \frac{b_{m-2} - u_{m-2,m-1}x_{m-1} - u_{m-2,m}x_m}{u_{m-2,m-2}} \\ \frac{b_{m-1} - u_{m-1,m}x_m}{u_{m-1,m-1}} \\ \frac{b_m}{u_{mm}} \end{bmatrix}$$

We do not calculate the LU factorization of the basis matrix at every iteration, this would be extremely time consuming. Instead we use an eta matrix to store the new basis matrix information. Since the new basis matrix and the old basis matrix only differ by one column we can use eta matrix. The new basis matrix looks like

$$\overline{B} = BE_1 \tag{2.21}$$

where

$$E_1 = \begin{bmatrix} 1 & 0 & \cdots & e_{1q} & \cdots & 0 \\ 0 & 1 & \cdots & e_{2q} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & e_{qq} & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & e_{mq} & \cdots & 1 \end{bmatrix}$$

So we have the following formulation of the basis.

$$PB = LUE_1E_2...E_q \tag{2.22}$$

where q is the number of eta matrices. The eta matrices are just the identity matrix with a column changed, so we only store the column and the place where the column is different. The eta matrix storage looks like the following:

$$E = \begin{bmatrix} \cdots & e_1 & \cdots \\ \cdots & e_2 & \cdots \\ & \vdots & \\ \cdots & e_q & \cdots \end{bmatrix}$$

where e_1 is the vector of the first eta matrix E_1. We also need a vector to store where the columns are different

$$p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_q \end{bmatrix}$$

where p_1 is the column of E_1 which is different than the identity matrix.

Now that we have the full factorization of the basis matrix given in (2.1), we still need to solve the equations $Bx = b$ and $B^T x = b$.

$$PB = LUE_1E_2...E_q \tag{2.23}$$

and we want to solve $Bx = b$ then pre-multiply by P and substituting PB and we get

$$\begin{aligned} LUE_1E_2...E_qx &= Pb && (2.24)\\ \text{Solve}Ly &= Pb && (2.25)\\ \text{Solve}Uz &= y && (2.26)\\ \text{Solve}E_1x_1 &= z && (2.27)\\ \text{Solve}E_2x_2 &= x_1 && (2.28)\\ \cdots & && (2.29)\\ \text{Solve}E_qx &= x_{q-1} && (2.30) \end{aligned}$$

To solve $B^T x = b$ we again multiply by $P^T P = I$

$$
\begin{aligned}
B^T P^T P x &= b && (2.31)\\
(PB)^T P x &= b && (2.32)\\
(LUE_1E_2...E_q)^T P x &= b && (2.33)\\
E_q^T...E_2^T E_1^T U^T L^T P x &= b && (2.34)\\
\text{Solve} E_q^T x_q &= b && (2.35)\\
&\cdots && (2.36)\\
\text{Solve} E_2^T x_2 &= x_3 && (2.37)\\
\text{Solve} E_1^T x_1 &= x_2 && (2.38)\\
\text{Solve } U^T y &= x_1 && (2.39)\\
\text{Solve } L^T z &= y && (2.40)\\
\text{Multiply } x &= P^T z && (2.41)
\end{aligned}
$$

Now we have to solve system of equations involving eta matrices namely $Ex = b$ and $E^T x = b$. Let us look at solving $Ex = b$ first. It is easy to verify that the solution to this system is

$$
\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 - e_{1q}x_q \\ b_2 - e_{2q}x_q \\ \vdots \\ \frac{b_q}{e_{qq}} \\ \vdots \\ b_m - e_{mq}x_q \end{bmatrix}
$$

We also have to solve a system of equations involving the transpose of an eta matrix or $E^T x = b$. The solution to this system is

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \frac{b_q - \sum_{i \neq q} e_{iq} x_i}{e_{qq}} \\ \vdots \\ b_m \end{bmatrix}$$

After accumulating eta matrices, CPLEX uses the default setting of refactorizing the basis after 100 iterations of the simplex algorithm. So we have chosen to refactorize after 100 also. Most of the problems that we solve are small enough as not to refactorize but it is still included in our code. The reason to refactorize is that instead of solving 100 eta matrices and the original LU factorization, it is usually better just to find the new LU factorization of the basis matrix.

Now that we have all of the elements of the basic simplex algorithm we are going to transition to talking about our implementation of the revised simplex method.

2.2 Revised Simplex Algorithm

The revised Simplex Algorithm has been developed in order to take advantage of the fact that we do not need to find all of the data in the tableau. This allows for less storage requirements and the ability to solve larger problems. The theory of the revised Simplex Algorithm can be found in [6]. We implemented the revised simplex algorithm found in [11] shown in Figure 2.2.

We have decided to use a steepest-edge primal simplex algorithm. The changes to the basic simplex algorithm found in Figure 2.1 are basically the initializing and updating of the steepest-edge norms. Besides that everything else is needed to perform the simplex algorithm. Note that Step 2 in Figure 2.2 is the choosing of the reduced cost and Step 5 is the minimum ratio test. For the primal simplex, we initialize the steepest-edge norms like in Step 0 but CPLEX initializes them with just 1. We talk more about this in the dual simplex section.

Inputs:	B (basis matrix), N (nonbasic matrix), c (objective function), x_B (current solution), $basis$, and $nonbasis$
Step 0.	Initialize: $c_{B_i} = c[basis[i]], i = 0..m-1$ $z_i = c_i - (B^{-1}N)_i^T \cdot c_B, i = 0..n-m-1$ $\gamma_i = (B^{-1}N)_i \cdot (B^{-1}N)_i + 1, i = 0..n-m-1$ $\overline{B} = B$
Step 1.	If $z_k \geq 0$, where $k = 0..n-m-1$ then $\overline{B}$ is optimal. Stop.
Step 2.	Else $jp = \arg\max \left\{ \frac{z_k^2}{\gamma_k} : z_k < 0, k = 0..n-m-1 \right\}$. x_{jp} is the entering variable.
Step 3.	Solve $Bw = A_{jp}$. w is the jp^{th} column. This is used to find the new eta matrix.
Step 4.	If $w_k \leq 0$, $k = 0..m-1$ then we are infeasible. Stop.
Step 5.	Else $ip = \arg\min \left\{ r_k = \frac{x_{B_k}}{w_k} : w_k > 0, k = 0..m-1 \right\}$. Set $\Delta = r_{ip}, w_p = w_{ip}$ This is the minimum ratio test. x_q is the leaving variable.
Step 6.	Update $x_{B_{ip}} \leftarrow \Delta$ $x_{B_i} \leftarrow x_{B_i} - w \cdot \Delta$ for $i = 0..m-1$ and $i \neq ip$
Step 7.	Change $z_q \leftarrow c_q - w \cdot c_B$ Change $\gamma_q \leftarrow 1 + w \cdot w$
Step 8.	Solve $\overline{B}^T \overline{w} = w$. Use the same storage vector.
Step 9.	Update basis, nonbasis, and eta matrix. Set the ip^{th} column of the identity matrix equal to $\overline{w}$ to get E_p $\overline{B} \leftarrow \overline{B} \cdot E_p$ where p is the number of eta matrices
Step 10.	Solve $\overline{B}^T y = e_{ip}$
Step 11.	Find $\alpha_N = \overline{N}^T y$.
Step 12.	Update $z_k \leftarrow z_k - z_q \cdot \alpha_{N_k}$ for $k \neq q$.
Step 13.	Update $\gamma_k \leftarrow \gamma_k - 2\alpha_{N_k} \overline{w}^T \overline{N}_k + (\alpha_k)^2 \gamma_q$ for $k \neq q$.
Step 14.	Update $z_q \leftarrow -\frac{z_q}{w_p}$ and $\gamma_q = \frac{\gamma_q}{w_p^2}$
Step 15.	Go to Step 1.

Figure 2.2: Revised Simplex Algorithm

2.3 Dual Simplex

If we have an LP that is dual feasible but not primal feasible then we use the dual simplex algorithm. Most of the previous concepts can also be applied to the dual simplex. If we have a dual feasible-primal infeasible system then we can still perform the simplex method with a few minor changes. Instead of choosing a

negative reduced cost, we choose a negative $B^{-1}b$ value to find our pivot row. Then we perform a minimum ratio test similar to the previous one except that we need to maintain dual feasibility by performing the following operation.

Definition 21 *Minimum ratio test for the dual*

$$\min\left\{\frac{c_j}{-B^{-1}N_{ij}} : B^{-1}N_{ij} < 0\right\} \tag{2.42}$$

where i,j is the row, column respectively.

Using this pivot column will insure that we stay dual feasible.

The basic dual simplex algorithm for solving LP's is summarized in Figure 2.3.

Step 0.	Inputs x_B, c, B, N
Step 1.	Choose a negative $B^{-1}b$. This is the pivot row.
Step 2.	Use the dual minimum ratio test to find the pivot column.
Step 3.	Pivot in this row and column.
Step 4.	If optimal, stop.
Step 5.	Else go to Step 2.

Figure 2.3: Basic Dual Simplex Algorithm

Since the dual simplex is similar to the primal we do not need to list the full dual simplex algorithm that we implemented. There are a couple of changes that need to be mentioned though. We used the dual simplex algorithm found in [4]. In [4], they suggest a couple of different ways to initialize and update the the steepest edge norms.

The first change is how to choose the entering variable. We choose the entering variable by the rule in equation 2.43.

$$ip = \arg\max\left\{\frac{(B^{-1}b)_k^2}{\gamma_k} : k = 0..m-1\right\} \tag{2.43}$$

The steepest norms can be calculated in each iteration by equation 2.44.

$$\gamma_k = (e_k^T B^{-1})(e_k^T B^{-1})^T \tag{2.44}$$

However there is an effective update found in equation 2.45 for this choice.

$$\gamma_k = \gamma_k - 2\left(\frac{w_k}{w_{ip}}\right) e_k^T B^{-1} y + \left(\frac{w_k}{w_{ip}}\right)^2 y^T y \tag{2.45}$$

where y is the variable in Step 10, w is the variable in Step 3 in Figure 2.2. Since the steepest edge norms do not need to be exactly computed then update them using exact arithmetic but then we truncated the values down to eight digits of accuracy. This way we are not saving huge fractions that we do not need. Since most of the basic variables are slack variables (where $\gamma_k = 1$) then CPLEX also uses an approximation to the steepest edge norms. They initialize all of the steepest edge norms to be 1. We decided to do the same thing for the dual simplex.

CHAPTER 3
Exact Arithmetic and Matrix Implementation

We have discussed in great detail the primal and dual simplex algorithms. We have decided to use an exact arithmetic version of both, the primal and dual simplex. Note that the simplex algorithm only performs Gaussian pivots which will not produce any irrational numbers as long as the given data is not irrational. Thus we can use fractions to represent all of the data. Since we have the theory of simplex, we need to discuss the issues regarding exact arithmetic. Exact arithmetic causes more complexity.

3.1 Exact Arithmetic

The first thing that needs to be developed is a way to handle exact arithmetic. Exact arithmetic is a way to represent numbers and perform arithmetic operations as fractions rather than using approximate representations. Performing standard arithmetic operations becomes more difficult than performing these in floating point arithmetic. When working in exact arithmetic, all data are rational, with the numerator and denominator of each rational stored. Let us look at two standard fractions:

$$f_1 = \frac{a}{b} \tag{3.1}$$

$$f_2 = \frac{c}{d} \tag{3.2}$$

If we want to add or subtract these fractions then

$$f_1 \pm f_2 = \frac{ad \pm cb}{bd} \tag{3.3}$$

Notice that this operation involves three multiplications and an addition or

subtraction respectively. If we want to multiply f_1 and f_2 then

$$f_1 \cdot f_2 = \frac{ac}{bd} \tag{3.4}$$

two multiplications are needed. Note that division will use two multiplications also. Another operation that needs to be considered is comparison. Normally, comparing two numbers can be done by just using less than or greater than. However, with fractions, comparison needs a little more work. To know if $f_1 > f_2$ then we need to perform the following operations:

$$f_1 > f_2 \Leftrightarrow a \cdot d > b \cdot c \tag{3.5}$$

So, exact comparison uses two multiplication and a standard comparison. Lastly, we want to keep the numerator and denominator relatively prime so that the fractions will take up the least amount of memory. This process of reducing the fractions to lowest terms can be the most time consuming and operation intensive of all of the operations talked about thus far. There are a couple of algorithms to consider. Euclid's algorithm is described in great detail in both [16] and [17]. Another algorithm described in [17] is the one that is normally used in exact code. However, we would like to look at Euclid's algorithm even though it could use more operations than the other algorithm. For example, if we want to find the greatest common divisor (gcd) between 48 and 38. Notice that

$$\frac{48}{38} = 1 + \frac{10}{38} \tag{3.6}$$

If $\frac{48}{38}$ is to be reduced, then $\frac{10}{38}$ must also be able to be reduced by the same number. This means that $\{10, 38, 48\}$ must all have the same gcd as $\{38, 48\}$. Now we can look at

$$\frac{38}{10} = 3 + \frac{8}{10} \tag{3.7}$$

The same reasoning can be applied so we end up with

$$\frac{10}{8} = 1 + \frac{2}{8} \tag{3.8}$$

Lastly,

$$\frac{8}{2} = 4 + \frac{0}{2} \tag{3.9}$$

We can now say that 2 is the gcd since 2 is the largest number that divided $\{2, 8, 10, 38, 48\}$. Notice that we are just using modulo operations to find the remainder of the two numbers. Refer to Table 3.1 for a condensed version of what is described above.

Large	Small	Remainder
48	38	10
38	10	8
10	8	2
8	2	0

Table 3.1: Euclid's Algorithm for computing the gcd

Euclid's algorithm stops when we reach a zero remainder and the small number is the gcd that we were looking for. Once the gcd is found then we take the fraction by dividing the numerator and denominator by the gcd.

$$\frac{48}{38} = \frac{\frac{48}{2}}{\frac{38}{2}} = \frac{24}{19} \tag{3.10}$$

If we reach a remainder of 1 then that means that the two numbers are relatively prime. Basically, Euclid's algorithm uses the following theorem

Theorem 3

$$\gcd(a, b) = \gcd(a, b \bmod a) \tag{3.11}$$

given $a < b$.

Another algorithm found in [17] uses the theorem found in Theorem 4.

Theorem 4

$$\gcd(a, b) = \gcd(\min(a, b), |a - b|) \tag{3.12}$$

There is the need to reduce fractions due to the limited amount of memory a computer has available. If we did not reduce the fractions then the numbers could grow exponentially with every operation since every exact arithmetic operation uses multiplication. One question that needs to be addressed is whether to reduce after every arithmetic operation or simply to reduce when the numbers get large. There are advantages and disadvantages for both.

Granlund wrote a series of programs that he put into a callable C library called gmp.h (GNU multiple precision library) [14] that performs all of the operations discussed above. This library has programs which perform the storage and arithmetic operations of exact fractions using dynamic memory allocation. Besides needing enough memory for the numerator and denominator, we need three extra memory units to store the negative sign, the division bar and the end of character. This allows for fractions as large as needed (as long as there is enough memory). Knowing that we have a way to store the fractions and perform arithmetic operations, we can concentrate on writing the simplex algorithm in exact arithmetic. Fortran did not have the tools to use this library and hence our code is written in C.

3.2 Exact vs. Real Arithmetic Simplex Algorithm

Since we are using exact arithmetic, a couple of things become easier, namely, checking the reduced cost and the minimum ratio test. When the reduced cost gets closer to zero, the standard implementation needs to compare these numbers to a tolerance to see if it is zero. CPLEX uses a default tolerance of 10^{-5} for integer programming as to whether numbers are integral or not. This makes the assumption that the numbers close to zero are exactly zero. Also, one must be careful about negative reduced costs that are close to zero. Do you pivot on these columns or just assume that they are zero? If you pivot on these columns and it is supposed to be zero then it becomes an extra pivot on a dual degenerate basis. If you do not pivot because you assume it to be zero and it is not zero then you are not actually optimal. Examples can be contrived that exploit both of these flaws. However, using exact

arithmetic alleviates both of these issues. Let's look at an example with a tolerance of 10^{-4}. Given the problem:

Example 3

$$\begin{array}{lrcrcrcr} \min & -1.0001x_1 & - & 2x_2 & & & & \\ \text{s.t.} & x_1 & + & 2x_2 & + & x_3 & = & 10000 \\ & x_i & \in & \mathbb{Z}^+ & & & & \end{array}$$

Bring x_2 into the basis, the tableau looks like

10000	$-.0001$	0	1
5000	$\frac{1}{2}$	1	$\frac{1}{2}$

If the tolerance is 10^{-4} then we would assume $-.0001$ is zero and hence not pivoted on. This would result in an answer of

$$x_1 = 0 \quad x_2 = 5000 \quad x_3 = 0$$

with an objective value of -10000. This answer is not the optimal answer. The optimal answer is

$$x_1 = 10000 \quad x_2 = 0 \quad x_3 = 0$$

with an objective value of -10001. This would make a difference if, for example x_1 represented the number of bushels of corn and x_2 represented the number of gallons of honey a farmer should sell.

Besides having problems with integer programming problems, CPLEX can actually give wrong answers because of the tolerances used. Here is another example of what could go wrong when using tolerances (Example 4).

Example 4

$$\begin{array}{lrcrcr} \min & -1x_1 & - & 1x_2 & & \\ \text{s.t.} & .499999x_1 & + & .5x_2 & = & 1 \\ & x_1 & + & 2x_2 & = & 3 \\ & x_i & \geq & 0 & & \end{array}$$

CPLEX gives a solution of $x_1 = x_2 = 1$ with an objective value of -2. This point is not feasible.

Look at Example 5 for another example of problems with using

Example 5

$$\begin{array}{lrcrcl} \min & -1x_1 & - & 1x_2 & & \\ \text{s.t.} & .3333333x_1 & + & .6666667x_2 & = & 1 \\ & x_1 & + & 2x_2 & = & 3 \\ & x_i & \geq & 0 & & \end{array}$$

Here CPLEX gives a wrong answer saying that the solution is $x_1 = 3$ and $x_2 = 0$. This point is again not feasible. These are just plain two dimensional linear programming problems. This is what can happen when tolerances are used. No matter what tolerance is used one can generate examples that cannot be solved or worse give the wrong answer.

3.3 Matrix Conditioning

We need to solve a system of equations when using the simplex algorithm. Let us say we are solving the equations:

$$Ax = b \tag{3.13}$$

where $A = (a_{ij})$ and A is nonsingular $A \in \Re^{mxm}$. The solution to this system is

$$x = A^{-1}b \tag{3.14}$$

or according to [15]

$$x = \frac{1}{detA}(adjA)b \tag{3.15}$$

The $adjA = (a_{ij}^{'})^T$ where $a_{ij}^{'}$ is the cofactor of a_{ij}. Now the solution can be written as

$$x_j = \frac{1}{detA} \sum_{i=1}^{m} a_{ij}^{'} b_i \tag{3.16}$$

Since the cofactors of a matrix are multiplications then notice that the complexity of A^{-1} is a sum of multiplications. This is extremely costly when dealing with fractions.

Earlier in this chapter we talked about how large fractions could get. Plus we discussed the extra memory for the storage besides the numerator and denominator. However since the numerator and denominator are the only quantities that change as far as the storage requirements are concerned then we will define the maximum size of a vector as in definition 22.

Definition 22 *Maximum size of a vector is the total number of digits in the element of the vector that has the size of the numerator plus the size of the denominator to be the greatest in the whole vector.*

The maximum size of a vector is an important number because once the memory has been allocated for a variable then it does not get reallocated if a smaller number gets placed in that variable. This cuts down on a lot of reallocation issues like segmentation of memory.

While we are solving systems of equations in steps 3 and 10 for the primal simplex and we solve the same systems for the dual simplex, we are interested in what the maximum size of the solution vector for each simplex iteration. For example, one unique simplex implementation idea for exact arithmetic is to choose the entering variable according to the rule of choosing the w-vector with the smallest size requirement. This might incur more simplex iterations but it might cut down on the total storage requirements needed in the simplex algorithms since the w-vector is used to compute the new eta matrix. Therefore solving subsequent systems of equations based on this eta matrix would take less time than an eta matrix with huge fractions to deal with. However, we have stuck with the convention of choosing

the reduced cost based on the steepest edge since this has at the very least positive results for floating-point arithmetic.

3.4 Matrix Implementation

One implementation issue is how to store the tableau of the simplex. We are using a sparse matrix storage called YSMP (Yale Sparse Matrix Protocol). Instead of storing a lot of the zeros of the tableau, we use three vectors to represent the data. The first vector is the data vector. This stores the numbers in row-major order. The second vector is the "row vector". Each element in this vector tells the spot where that row begins. The third vector is the "column vector". Each element represents the column that that data element is in. For example, if we have a following matrix:

$$\begin{bmatrix} 1 & -2 & 0 & 0 & 0 & 3 \\ 0 & 0 & -3 & 4 & 0 & 0 \\ 8 & 0 & 0 & 0 & 7 & 0 \\ 10 & 1 & 0 & 0 & 0 & 2 \end{bmatrix}$$

The vectors would look like this

$$\begin{aligned} M &= [1 \quad -2 \quad 3 \quad -3 \quad 4 \quad 8 \quad 7 \quad 10 \quad 1 \quad 2] \\ IM &= [0 \quad 3 \quad 5 \quad 7 \quad 10] \\ JM &= [1 \quad 2 \quad 6 \quad 3 \quad 4 \quad 1 \quad 5 \quad 1 \quad 2 \quad 6] \end{aligned}$$

Now we have all of the necessary material to proceed to talking about the theory and implementation of Gomory cutting planes.

CHAPTER 4
Gomory Cutting Planes

Gomory's cutting plane algorithm attempts to solve integer programming problems by cutting off the linear programming relaxation solution until an integer solution is found. These cuts are generated from the rows of the optimal LP relaxation simplex tableau. If not all of the basic variables are integer then there exists a row i that has a fractional $B^{-1}b$ value. From this point on, we are going to refer to $B^{-1}b$ as just b. This row has the form:

$$x_{B_i} + \sum_{j \in NB} a_{ij} x_j = b_i \tag{4.1}$$

where NB is the set of non-basic variables in the simplex tableau. Rewriting (4.1) in terms of the fractional and integer parts yields:

$$x_{B_i} + \sum_{j \in NB} \lfloor a_{ij} \rfloor x_j + \sum_{j \in NB} f(a_{ij}) x_j = \lfloor b_i \rfloor + f(b_i) \tag{4.2}$$

where $f(a) = a - \lfloor a \rfloor$. Rearranging the terms, (4.2) becomes:

$$x_{B_i} + \sum_{j \in NB} \lfloor a_{ij} \rfloor x_j = \lfloor b_i \rfloor + f(b_i) - \sum_{j \in NB} f(a_{ij}) x_j \tag{4.3}$$

We can now write (4.3) as an inequality:

$$x_{B_i} + \sum_{j \in NB} \lfloor a_{ij} \rfloor x_j \leq \lfloor b_i \rfloor + f(b_i) \tag{4.4}$$

Since everything is integer except $f(b_i)$ then the following inequality is also satisfied:

$$x_{B_i} + \sum_{j \in NB} \lfloor a_{ij} \rfloor x_j \leq \lfloor b_i \rfloor \tag{4.5}$$

The optimal solution to the LP relaxation will violate constraint (4.5), unless

b_i is integral.

Note that from solving equation (4.1) for x_{B_i} we have

$$x_{B_i} = b_i - \sum_{j \in NB} a_{ij} x_j \tag{4.6}$$

If x_{B_i} in (4.6) is substituted into (4.5) then the following inequality is valid

$$\sum_{j \in NB} f\left(a_{ij}\right) x_j \geq f\left(b_i\right) \tag{4.7}$$

The original Gomory fractional cuts can be strengthened in a couple of different ways. The first that will be discussed is found in [18] but is attributed to Gomory in [12].

> "For any integer t, the cut
>
> $$\sum_{j \in NB} f(ta_{ij}) x_j \geq f(tb_i) \tag{4.8}$$
>
> is satisfied by all non-negative integer solutions to (4.6) and therefore by all solutions to (IP). Also, if $f(b_i) < \frac{1}{2}$, t is positive and $\frac{1}{2} \leq t \cdot f(b_i) < 1$, then (4.8) dominates (4.7)." [18]

Dominant inequalities are valid inequalities that cut off more of the feasible region than the inequalities that they dominate. For a strict definition found in [21] and for your convenience look at Definition 23.

Definition 23 *If $\pi x \leq \pi_0$ and $\mu x \leq \mu_0$ are two valid inequalities, $\pi x \leq \pi_0$ dominates $\mu x \leq \mu_0$ if there exists $u > 0$ such that $\pi > u\mu$ and $\pi_0 \leq u\mu_0$, and $(\pi, \pi_0) \neq (u\mu, u\mu_0)$.*

The second technique talked about in [18] and attributed to Gomory in [13] is a strengthening cut that can be generated by looking at the mixed integer Gomory cuts. We first need to develop a theorem found in [21]

Theorem 5 *If $X = \{(x, y) \in \mathbb{R}^1_+ \times \mathbb{Z}^1 : y \leq b + x\}$, and $f = b - \lfloor b \rfloor > 0$, the inequality*

$$y \leq \lfloor b \rfloor + \frac{x}{1-f} \tag{4.9}$$

is valid for X.

Let us look at just the integer parts of the mixed integer Gomory cuts found in [21]. To do this, like before there must be a row i which has a fractional right hand side b or we must be optimal.

$$x_{B_i} + \sum_{j \in NB} a_{ij} x_j = b_i \tag{4.10}$$

We can break up the nonbasic variables into two sets

$$x_{B_i} + \sum_{f_j \leq f_0} a_{ij} x_j + \sum_{f_j > f_0} a_{ij} x_j = b_i \tag{4.11}$$

where $f_j = f(a_{ij})$ and $f_0 = f(b_i)$. We can rewrite equation (4.11) as

$$x_{B_i} + \sum_{f_j \leq f_0} \lfloor a_{ij} \rfloor x_j + \sum_{f_j \leq f_0} f_j x_j + \sum_{f_j > f_0} \lceil a_{ij} \rceil x_j - \sum_{f_j > f_0} (1 - f_j) x_j = b_i \tag{4.12}$$

Rearranging the terms by putting the integer parts on the left and the other parts on the right.

$$x_{B_i} + \sum_{f_j \leq f_0} \lfloor a_{ij} \rfloor x_j + \sum_{f_j > f_0} \lceil a_{ij} \rceil x_j = b_i - \sum_{f_j \leq f_0} f_j x_j + \sum_{f_j > f_0} (1 - f_j) x_j \tag{4.13}$$

Notice that the above equation can be written as an inequality

$$x_{B_i} + \sum_{f_j \leq f_0} \lfloor a_{ij} \rfloor x_j + \sum_{f_j > f_0} \lceil a_{ij} \rceil x_j \leq b_i + \sum_{f_j > f_0} (1 - f_j) x_j \tag{4.14}$$

Using Theorem 5, we obtain the valid inequality,

$$x_{B_i} + \sum_{f_j \leq f_0} \lfloor a_{ij} \rfloor x_j + \sum_{f_j > f_0} \lceil a_{ij} \rceil x_j \leq \lfloor b_i \rfloor + \sum_{f_j > f_0} \frac{1 - f_j}{1 - f_0} x_j \tag{4.15}$$

Substituting x_{B_i} from equation (4.11) into inequality (4.15) and simplifying yields

$$f_0 \leq \sum_{f_j \leq f_0} f_j x_j + \sum_{f_j > f_0} \frac{f_0(1-f_j)}{1-f_0} x_j \tag{4.16}$$

Notice that when $f_j \leq f_0$ then:

$$f_j \leq f_0 = f_0 \frac{1-f_0}{1-f_0} \leq f_0 \frac{1-f_j}{1-f_0} \tag{4.17}$$

and when $f_j > f_0$ then:

$$f_0 \frac{1-f_j}{1-f_0} \leq f_0 \frac{1-f_0}{1-f_0} = f_0 < f_j \tag{4.18}$$

Thus, we can rewrite inequality (4.16) as

$$f_0 \leq \sum_{j \in NB} \min \left\{ f_j, f_0 \frac{1-f_j}{1-f_0} \right\} x_j \tag{4.19}$$

is valid and (4.19) is stronger than (4.7).[18]

One might presume that we can use the first strengthening technique on this inequality. However, this is not so because all of the coefficients are less than f_0. To verify this note equations (4.17) and (4.18).

On the other hand, we can use the second strengthening (4.19) on the first strengthening technique (4.8). This we will refer to as combination strengthening.

$$\sum_{j \in NB} \min \left\{ f(ta_j), f(tb_i) \frac{1-f(ta_j)}{1-f(tb_i)} \right\} x_j \geq f(tb_i) \tag{4.20}$$

This theory also works on the objective function row. Notice that $c^T x = -z$ so if $-z$ is fractional then the Gomory cutting plane for the objective function row can be added to the tableau. Therefore the same formulas all work with the objective function.

Out of the four possible choices of Gomory cutting planes (normal, t-Gomory, strong, and combination), we have chosen to just use strong Gomory cutting planes. Strong, t-Gomory, and combination strengthening have all been proven to be stronger

than the normal Gomory cutting planes so we will not use normal Gomory cutting planes. Using the same proof that strong Gomory cutting planes are stronger than normal Gomory cutting planes, we can show that combination is stronger than t-Gomory cutting planes. So now we have two choices. We do not want to apply both of these techniques to every row of the simplex tableau so we should choose one. There is no way to compare these two choices. However, we have found that strong works very well when f_0 is very small since then all of the coefficients in (4.19) must be less than f_0. The combination first increases f_0 as large as it can without going above an integer and then applies the strong technique which is counterproductive since strong works the best when f_0 is small. So, the combination of applying the strong to the t-Gomory cutting planes is not usually better than applying the strong cuts when f_0 is small. When f_0 is large (greater than $\frac{1}{2}$) then the t-Gomory cutting planes are not used so the combination is exactly the same as the strong. Therefore, we have decided to just use strong Gomory cutting planes.

The general Strong Gomory cutting plane algorithm is listed in Figure 4.1.

Step 1.	Find the Simplex Tableau.
Step 2.	Find the Strong Gomory Cutting Planes associated with each row that has a fractional right hand side.
Step 3.	Add these cutting planes to the Simplex tableau maintaining primal feasibility.
Step 4.	Use the Dual Simplex Algorithm to find a solution to the new LP.
Step 5.	If optimal, stop.
Step 6.	Else go to Step 2.

Figure 4.1: Exact Strong Gomory Cutting Plane Algorithm

We now look at an example which illustrates all of the cuts described above.

Example 6

$$\begin{array}{lrcrcl} \min & - & 2x_1 & - & 3x_2 & \\ \text{s.t.} & & x_1 & - & x_2 & \leq 1 \\ & & 4x_1 & + & x_2 & \leq 28 \\ & & x_1 & + & 4x_2 & \leq 27 \\ & & x_1 & , & x_2 & \in \mathbb{Z}^+ \end{array}$$

The simplex tableau of the linear programming relaxation is

0	−2	−3	0	0	0
1	1	−1	1	0	0
28	4	1	0	1	0
27	1	4	0	0	1

The optimal form of this tableau is

$\frac{82}{3}$	0	0	0	$\frac{1}{3}$	$\frac{2}{3}$
$\frac{2}{3}$	0	0	1	$-\frac{1}{3}$	$\frac{1}{3}$
$\frac{17}{3}$	1	0	0	$\frac{4}{15}$	$-\frac{1}{15}$
$\frac{16}{3}$	0	1	0	$-\frac{1}{15}$	$\frac{4}{15}$

Let us look at the third constraint

$$x_2 - \frac{1}{15}x_4 + \frac{4}{15}x_5 = \frac{16}{3} \tag{4.21}$$

Using the rounding technique described in (4.5), we find the floor of all of the coefficients to get the following valid inequality

$$x_2 - x_4 \leq 5 \tag{4.22}$$

Equation (4.21) can also be rounded using the fractional parts as described in (4.7) by finding out how much each coefficient has rounded down.

$$\frac{14}{15}x_4 + \frac{4}{15}x_5 \geq \frac{1}{3} \tag{4.23}$$

Note that (4.22) and (4.23) are actually equivalent. If we put both of the inequalities in terms of the original variables x_1 and x_2 then (4.22) becomes

$$\begin{aligned} x_2 - x_4 &\leq 5 \\ x_2 - (28 - 4x_1 - x_2) &\leq 5 \\ 4x_1 + 2x_2 &\leq 33 \end{aligned}$$

and (4.23) becomes

$$\begin{aligned} \frac{14}{15}x_4 + \frac{4}{15}x_5 &\geq \frac{1}{3} \\ \frac{14}{15}(28 - 4x_1 - x_2) + \frac{4}{15}(27 - x_1 - 4x_2) &\geq \frac{1}{3} \\ -4x_1 - 2x_2 + \frac{100}{3} &\geq \frac{1}{3} \\ 4x_1 + 2x_2 &\leq 33 \end{aligned}$$

Notice that in (4.23) the right hand side of the inequality $b_3 = \frac{1}{3}$ fits the criteria in (4.8). So (4.23) can be multiplied by $t = 2$ to get a stronger valid inequality.

$$2 \cdot \left(\frac{14}{15}x_4 + \frac{4}{15}x_5\right) \geq 2 \cdot \left(\frac{1}{3}\right) \tag{4.24}$$

$$\frac{28}{15}x_4 + \frac{8}{15}x_5 \geq \frac{2}{3} \tag{4.25}$$

Rounding (4.25)

$$\frac{13}{15}x_4 + \frac{8}{15}x_5 \geq \frac{2}{3} \tag{4.26}$$

Note that (4.26) is stronger than (4.23) because (4.25) is equivalent to (4.23) but (4.26) is harder to satisfy than (4.25) since it has been rounded. Inequality (4.26) in terms of the original variables is:

$$\frac{13}{15}x_4 + \frac{8}{15}x_5 \geq \frac{2}{3} \tag{4.27}$$

$$\frac{13}{15}(28 - 4x_1 - x_2) + \frac{8}{15}(27 - x_1 - 4x_2) \geq \frac{2}{3} \tag{4.28}$$

$$4x_1 + 3x_2 \leq 38 \tag{4.29}$$

Using the other strengthening technique described in (4.19) the cut (4.23)

$$\frac{14}{15}x_4 + \frac{4}{15}x_5 \geq \frac{1}{3}$$

becomes

$$\min\left\{\frac{14}{15}, \frac{1}{3}\cdot\frac{\frac{1}{15}}{\frac{2}{3}}\right\}x_4 + \min\left\{\frac{4}{15}, \frac{1}{3}\cdot\frac{\frac{11}{15}}{\frac{2}{3}}\right\}x_5 \geq \frac{1}{3} \tag{4.30}$$

$$\min\left\{\frac{14}{15}, \frac{1}{30}\right\}x_4 + \min\left\{\frac{4}{15}, \frac{11}{30}\right\}x_5 \geq \frac{1}{3} \tag{4.31}$$

$$\frac{1}{30}x_4 + \frac{4}{15}x_5 \geq \frac{1}{3} \tag{4.32}$$

Notice that (4.32) is stronger than (4.23) since the coefficient of x_4 is less, making (4.32) harder to satisfy.

When (4.32) is put in terms of the original variables, it becomes

$$\frac{1}{30}x_4 + \frac{4}{15}x_5 \geq \frac{1}{3} \tag{4.33}$$

$$\frac{1}{30}(28 - 4x_1 - x_2) + \frac{4}{15}(27 - x_1 - 4x_2) \geq \frac{1}{3} \tag{4.34}$$

$$4x_1 + 11x_2 \leq 78 \tag{4.35}$$

In Figure 4.2, the green line is the original Gomory cutting plane, the red line is the t-Gomory cutting plane and the blue line is the strong-Gomory cutting plane. Notice how the red and the blue lines both dominate the green line.

We will look at Example 7 to illustrate the use of these strong Gomory cutting planes and to show how these cutting planes can solve integer programming problems.

Example 7

$$\begin{array}{llrcrcl} \min & - & x_1 & - & x_2 & & \\ \text{s.t.} & & 29x_1 & + & x_2 & \leq & 87 \\ & & x_1 & + & 29x_2 & \leq & 87 \\ & & x_1 & , & x_2 & \in & \mathbb{Z}^+ \end{array}$$

The graph of the constraint set for this problem (generated by Maple) is seen in Figure 4.3. Notice that graphically the solution will be (2,2).

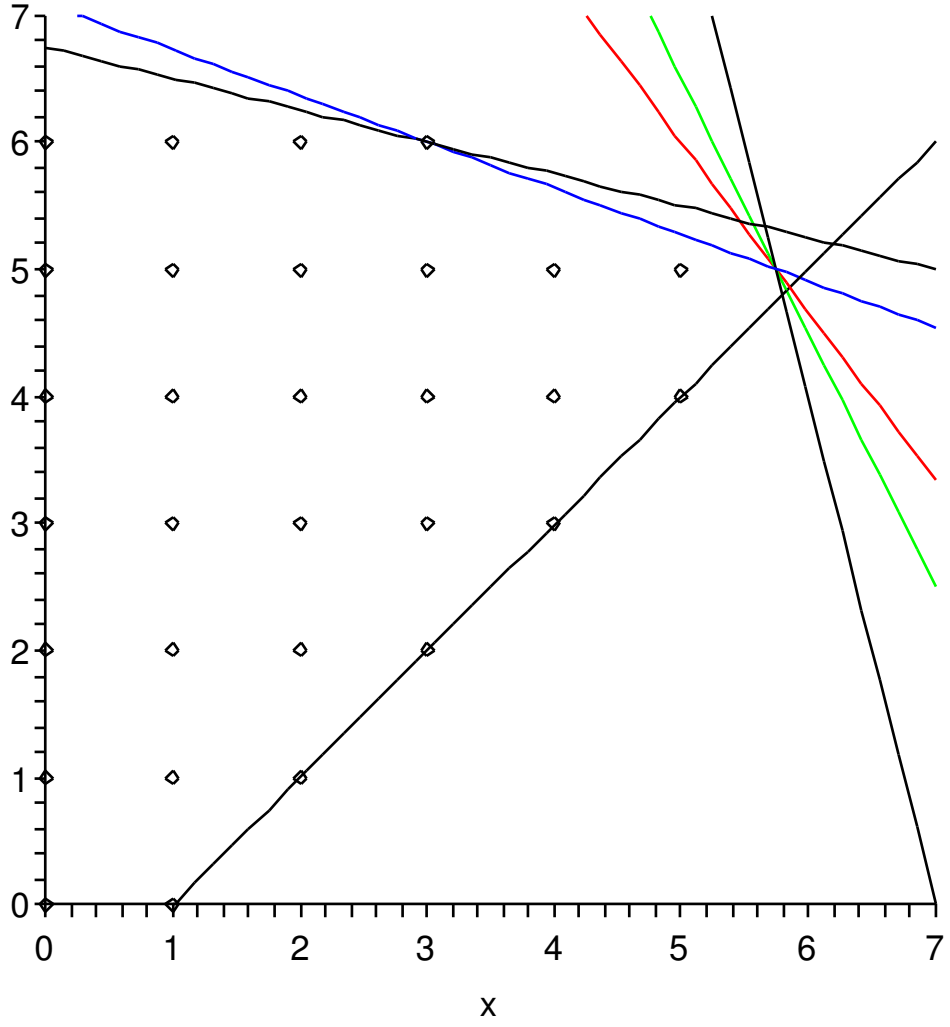

Figure 4.2: Graph of the Original and all of the types of Gomory cutting planes

The simplex tableau of the linear programming relaxation for this problem is

0	−1	−1	0	0
87	29	1	1	0
87	1	29	0	1

The optimal form of this tableau is

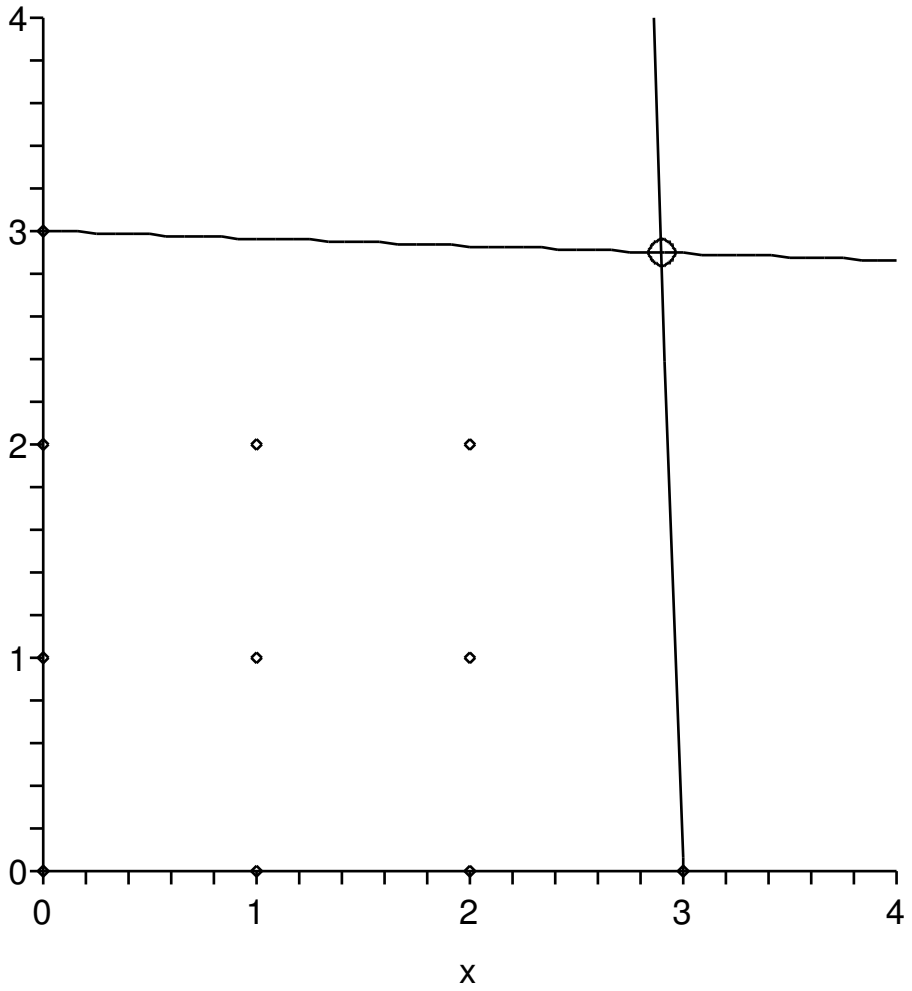

Figure 4.3: Original constraints of a simple example

$\frac{29}{5}$	0	0	$\frac{1}{30}$	$\frac{1}{30}$
$\frac{29}{10}$	1	0	$\frac{29}{840}$	$-\frac{1}{840}$
$\frac{29}{10}$	0	1	$-\frac{1}{840}$	$\frac{29}{840}$

The strong Gomory cutting planes are generated from each row that has a fractional right hand side using the rules in equation (4.19). Following the algorithm found in Figure 4.1, we add these to the tableau. Proceeding in this same fashion for three iterations we found the optimal value of -4 at the point $(2, 2)$. Whereas, CPLEX only achieves a lower bound on the objective value of -4.0205.

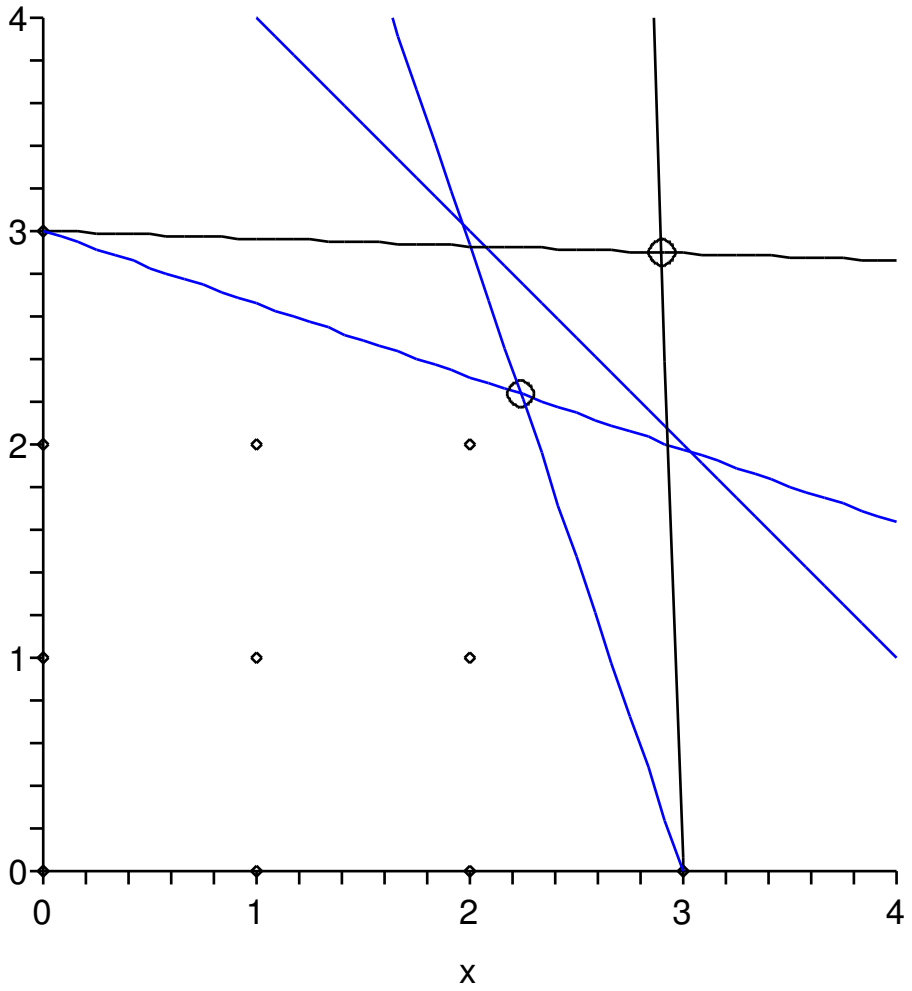

Figure 4.4: First set of Gomory cutting planes for a simple example

The progression of the cutting planes for each iteration is found in Figure 4.4 to Figure 4.6. Notice that the solution to each LP relaxation is cutoff by these cutting planes. This is the reason for the term cutting planes.

There are other types of cutting planes that CPLEX has available and can be found at http://eaton.math.rpi.edu:16080/cplex90html/usrcplex/solveMIP10.html.

The different kind of cuts listed on this website are clique cuts, cover cuts, disjunctive cuts, flow cover cuts, flow path cuts, gomory fractional cuts, generalized upper bound (GUB) cover cuts, implied bound cuts, and mixed integer rounding cuts. Clique cuts are used when there are relationships between binary variables. We

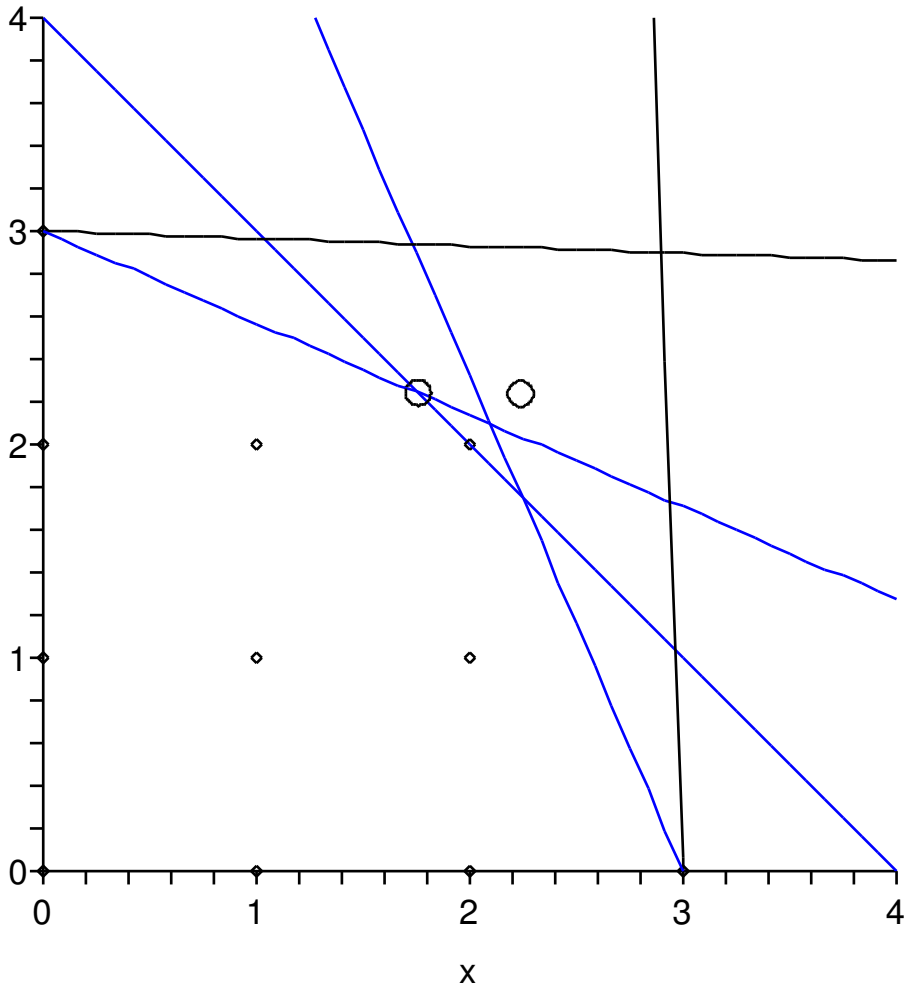

Figure 4.5: Second set of Gomory cutting planes for a simple example

deal with general integer programming problems which are not necessarily binary. Cover cuts are the cuts discussed in chapter 1. These cuts are for knapsack type problems where the variables are binary also. Disjunctive cuts are used when the problem has been branched on and we have two subproblems. These particular cuts are valid for each of their respective subproblems but not for the full LP relaxation. Flow cover and flow path cuts are used when the problem has continuous variables. These cuts treat the continuous variables as nodes and applies binary variables to whether the flow is on or off. GUB cuts are used when the sum of binary variables can be made to be less than or equal to 1. Implied bound cuts are used when

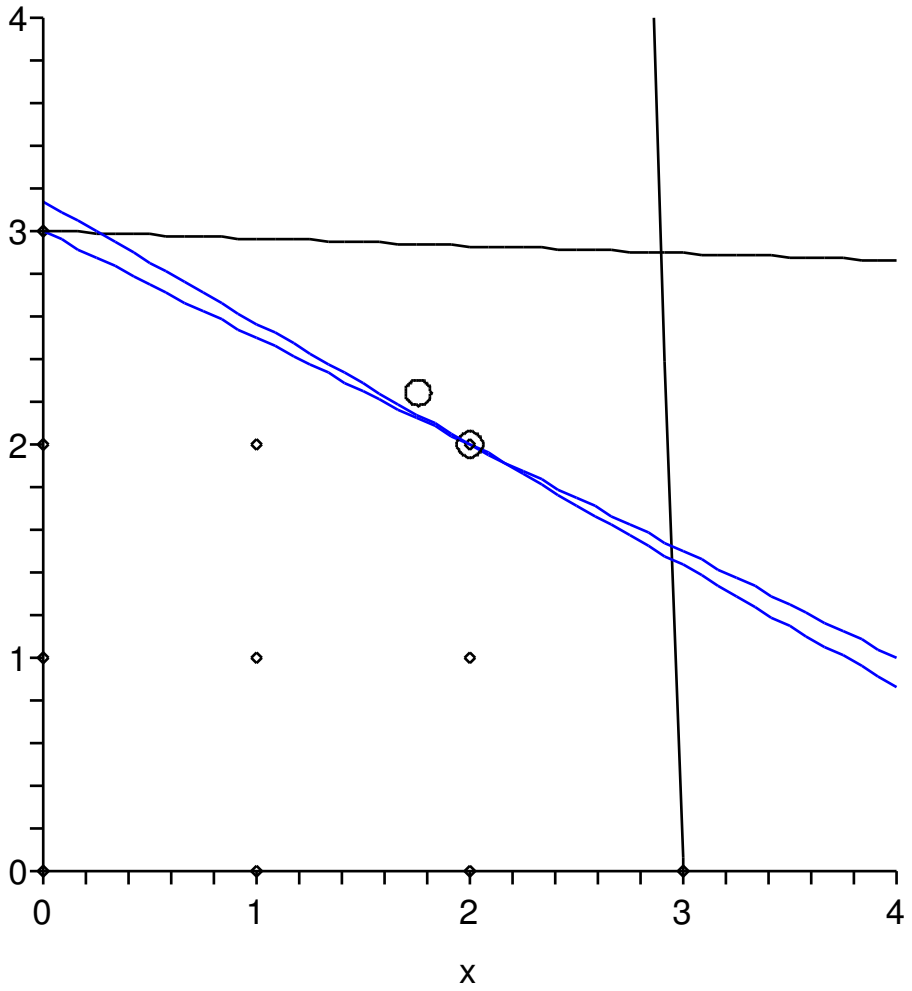

Figure 4.6: Final set of Gomory cutting planes for a simple example

the binary variables can imply something about the continuous variables in the problem. Mixed integer rounding cuts are generated when we have both continuous and integer variables and we can perform integer rounding on the integer variables.

We have talked about the theory of Gomory cutting planes. The best way to use Gomory cutting planes is to add in as many Gomory cuts as possible. We are now going to present why exact arithmetic will yield better Gomory cuts than floating point arithmetic.

CHAPTER 5
Gomory Cutting Planes in Exact Arithmetic

Gomory Cutting Planes Algorithms can be found in certain commercial codes like CPLEX. We are going to illustrate the use of exact arithmetic Gomory cutting planes. Also, we will show an example that can be solved by only needing to use cutting planes. We will use this example to describe the different results we collected and how they are use. We have started an example in an earlier chapter and so let us continue with that example. Recall that Example 6 has the following formulation:

$$\begin{array}{lrcrcl} \min & -2x_1 & - & 3x_2 & & \\ \text{s.t.} & x_1 & - & x_2 & \leq & 1 \\ & 4x_1 & + & x_2 & \leq & 28 \\ & x_1 & + & 4x_2 & \leq & 27 \\ & x_1 & , & x_2 & \in & \mathbb{Z}^+ \end{array}$$

Refer to Figure 5.1 to find the graph of the constraint set, all of the feasible integer points and the solution to the LP relaxation. Notice that the solution to the IP would be

$$\begin{aligned} x_1 &= 5 \\ x_2 &= 5 \end{aligned}$$

This problem does not have an ill-conditioned constraint matrix. Using exact arithmetic Gomory cutting planes, this problem can be solved. We are only going to talk about strong Gomory cutting planes presented earlier in (4.19) because these are the ones that we have determined to be the best ones. The optimal tableau of the LP relaxation is:

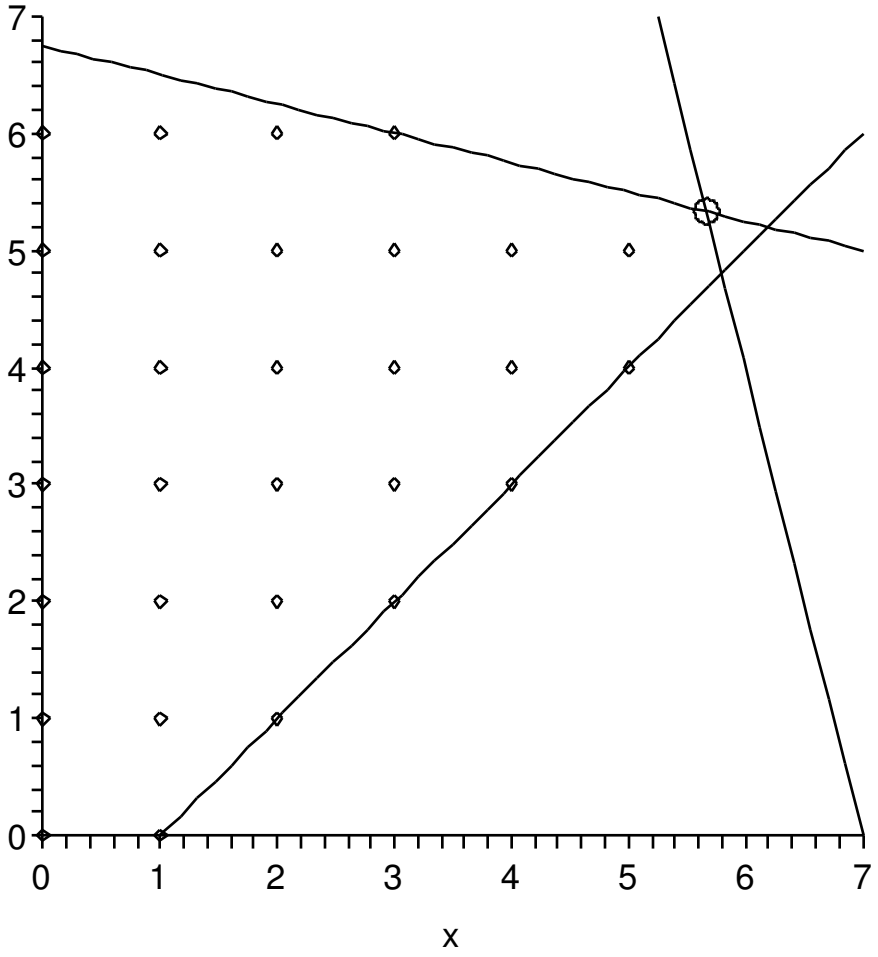

Figure 5.1: Feasible Integer Points for Example

$\frac{82}{3}$	0	0	0	$\frac{1}{3}$	$\frac{2}{3}$
$\frac{2}{3}$	0	0	1	$-\frac{1}{3}$	$\frac{1}{3}$
$\frac{17}{3}$	1	0	0	$\frac{4}{15}$	$-\frac{1}{15}$
$\frac{16}{3}$	0	1	0	$-\frac{1}{15}$	$\frac{4}{15}$

Notice that each of the constraint rows have fractional b-values and the objective value is fractional. So cuts can be generated from all of the rows of the tableau. The standard Gomory cuts rounding on the fractional parts using (4.7) are

$$\begin{aligned}
\frac{1}{3} &\leq \frac{1}{3}x_4 + \frac{2}{3}x_5 && (5.1)\\
\frac{2}{3} &\leq \frac{2}{3}x_4 + \frac{1}{3}x_5 && (5.2)\\
\frac{2}{3} &\leq \frac{4}{15}x_4 + \frac{14}{15}x_5 && (5.3)\\
\frac{1}{3} &\leq \frac{14}{15}x_4 + \frac{4}{15}x_5 && (5.4)
\end{aligned}$$

Through experimentation and the arguments from the previous chapter, we have found that it is usually best just to stick with the strong Gomory cuts developed in (4.19) rather than using the combination. With this in mind, the strong Gomory cutting planes become

$$\begin{aligned}
\frac{1}{3} &\leq \frac{1}{3}x_4 + \frac{1}{6}x_5 && (5.5)\\
\frac{2}{3} &\leq \frac{2}{3}x_4 + \frac{1}{3}x_5 && (5.6)\\
\frac{2}{3} &\leq \frac{4}{15}x_4 + \frac{2}{15}x_5 && (5.7)\\
\frac{1}{3} &\leq \frac{1}{30}x_4 + \frac{4}{15}x_5 && (5.8)
\end{aligned}$$

Adding the inequalities (5.5) - (5.8) to the optimal tableau would not suffice since the new slack variables could be non-integer in their current form. To overcome this, each inequality is multiplied by the greatest common denominator and then divided by the greatest common divisor of the coefficients. So the inequalities become

$$\begin{aligned}
2 &\leq 2x_4 + x_5 && (5.9)\\
2 &\leq 2x_4 + x_5 && (5.10)\\
5 &\leq 2x_4 + x_5 && (5.11)\\
10 &\leq 1x_4 + 8x_5 && (5.12)
\end{aligned}$$

respectively. Inserting the positive slacks gives the following equations

$$-2 = -2x_4 - x_5 + s_1 \quad (5.13)$$

$$-2 = -2x_4 - x_5 + s_2 \quad (5.14)$$

$$-5 = -2x_4 - x_5 + s_3 \quad (5.15)$$

$$-10 = -x_4 - 8x_5 + s_4 \quad (5.16)$$

Before continuing let us look at the storage requirements of Gomory cutting planes. By adding all of the constraints to the tableau we have doubled the number of rows. Since we are using YSMP format for matrices we only need to add in the non-zero elements of these constraints. However, we are using fractions that have a numerator and denominator. Recall that besides the numerator and denominator, fractions also need 3 extra memory units as discussed in an earlier chapter. However every fraction needs these 3 and they are static and so we are only going to talk about the storage of the numerator and denominator. Even if we have an integer value there is still a one in the denominator. We use an average to discuss the size of the Gomory cutting planes. The first Gomory cutting plane shown in 5.13 has an average storage of 2. To calculate this storage, we find the size of the numerator plus the size of the denominator. The size is the number of digits in base 10 of the number. We use a program in the Granlund library that computes the length of an integer. However, it always returns a value that is either exactly right or 1 more than needed. The reason for this is that this program is usually used to compute how much memory it needs so if the number is close then it will give a result that is one more. If we computed this in base 2 then it would be exact but harder for us to visualize so I used base 10 anyways. So for each Gomory cutting plane above we have the size requirements found in Table 5.1.

So if we were to plot this on a graph then we would only have two data points namely (1,2) and (1,2.5). The one stands for the first Gomory iteration.

Let us go back to were we left off. By adding the constraints 5.13 through 5.16 to the optimal tableau we get

Gomory Cutting Plane	Avg. Size
5.13	2
5.14	2
5.15	2
5.16	2.5

Table 5.1: Average Size of Gomory cutting planes

$\frac{82}{3}$	0	0	0	$\frac{1}{3}$	$\frac{2}{3}$	0	0	0	0
$\frac{2}{3}$	0	0	1	$-\frac{1}{3}$	$\frac{1}{3}$	0	0	0	0
$\frac{17}{3}$	1	0	0	$\frac{4}{15}$	$-\frac{1}{15}$	0	0	0	0
$\frac{16}{3}$	0	1	0	$-\frac{1}{15}$	$\frac{4}{15}$	0	0	0	0
-2	0	0	0	-2	-1	1	0	0	0
-2	0	0	0	-2	-1	0	1	0	0
-5	0	0	0	-2	-1	0	0	1	0
-10	0	0	0	-1	-8	0	0	0	1

Reoptimizing using dual simplex gives

26	0	0	0	0	0	0	0	$\frac{2}{15}$	$\frac{1}{15}$
1	0	0	1	0	0	0	0	$-\frac{1}{5}$	$\frac{1}{15}$
$\frac{26}{5}$	1	0	0	0	0	0	0	$\frac{11}{75}$	$-\frac{2}{75}$
$\frac{26}{5}$	0	1	0	0	0	0	0	$-\frac{4}{75}$	$\frac{1}{25}$
3	0	0	0	0	0	1	0	-1	0
3	0	0	0	0	0	0	1	-1	0
2	0	0	0	1	0	0	0	$-\frac{8}{15}$	$\frac{1}{15}$
1	0	0	0	0	1	0	0	$\frac{1}{15}$	$-\frac{2}{15}$

Since the slacks s_1 and s_2 in the tableau above are equal to 3 (positive), these constraints are not tight. If the constraint is not tight then most likely it is not a very useful constraint. In an attempt to control the growth of the tableau, we wish to eliminate constraints that are dominated (refer to definition 23). However dominated constraints are too complicated to find so it is easier to simply drop constraints that are not currently tight. Even though none of the original constraints are tight we

do not drop these. The original constraints are usually never totally cutoff by the Gomory cutting planes. So we never drop the original constraints.

Notice that there are only two constraints that have fractional b-values. So we again calculate the strong Gomory cutting planes until we reach the optimal solution. In Table 5.2, we list the Gomory iteration, the non-zero elements of the Gomory cutting planes, and their size.

Iteration	Non-Zeros				Avg. Size
1	-2	-2	-1	1	2
1	-2	-2	-1	1	2
1	-5	-2	-1	1	2
1	-10	-1	-8	1	2.5
2	-30	-22	-1	1	2.5
2	-15	-1	-3	1	2.25
3	-105	-1	-4	1	2.5
3	-180	-24	-5	1	2.75
3	-150	-7	-2	1	2.5
3	-300	-1	-56	1	2.75
3	-270	-10	-27	1	3
3	-210	-2	-21	1	2.75
4	-585	-27	-1	1	3
4	-6435	-93	-11	1	3.5
4	-585	-3	-1	1	2.75
4	-195	-11	-1	1	2.75
4	-585	-15	-1	1	3

Table 5.2: Size of Gomory cutting planes for each Gomory Iteration

Notice that in Table 5.2 that the size is getting progressively bigger as we get more Gomory cutting planes. We have put these in a graph to get an idea of how large these numbers grow and it can be found in Figure 5.2.

Figures 5.3-5.6 show the strong Gomory cuts as they would be in the variables x_1 and x_2 per Gomory iteration.

One reason why we can solve this using exact arithmetic is because the slack variables that are added into the Gomory cutting planes have to be integral. If floating point arithmetic is used then we would have to allow the slacks to be real numbers. When the slacks are not integral, one cannot make cuts as strong since the slack can take on many more values than the pure integer Gomory cuts. So,

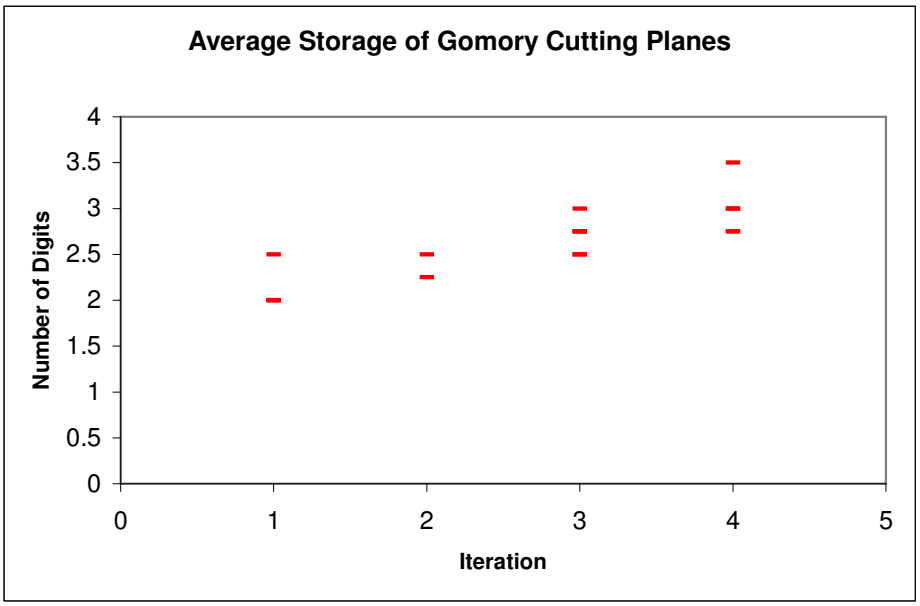

Figure 5.2: Average Size of Gomory Cutting Planes for Example 6

weaker mixed-Gomory cuts need to be implemented since the slacks could be real numbers.

In order to compare CPLEX with our code, we have used a relative optimality percentage to display the data. To compute this relative optimality percentage (ROP) we first calculate the relaxation gap where the relaxation gap is defined as the difference between the objective values of the LP relaxation and the actual optimal solution. After each iteration, we take the difference between the objective value at the current iteration and the LP relaxation and divide it by the relaxation gap. When this is multiplied by 100, it gives us the relative optimality percentage. Mathematically this percentage is defined as

$$ROP = \frac{it_i - LPR}{OPT - LPR} \cdot 100 \tag{5.17}$$

where LPR is the LP relaxation objective value, and OPT is the optimal objective value. Notice that when $it_i = LPR$ then the $ROP = 0\%$ and when $it_i = OPT$ then the $ROP = 100\%$. So implies that all problems start at the point (0,0) and hopefully ends at (it_i, 100).

The results are in Figure 5.7 for the exact code vs. CPLEX aggressive Gomory

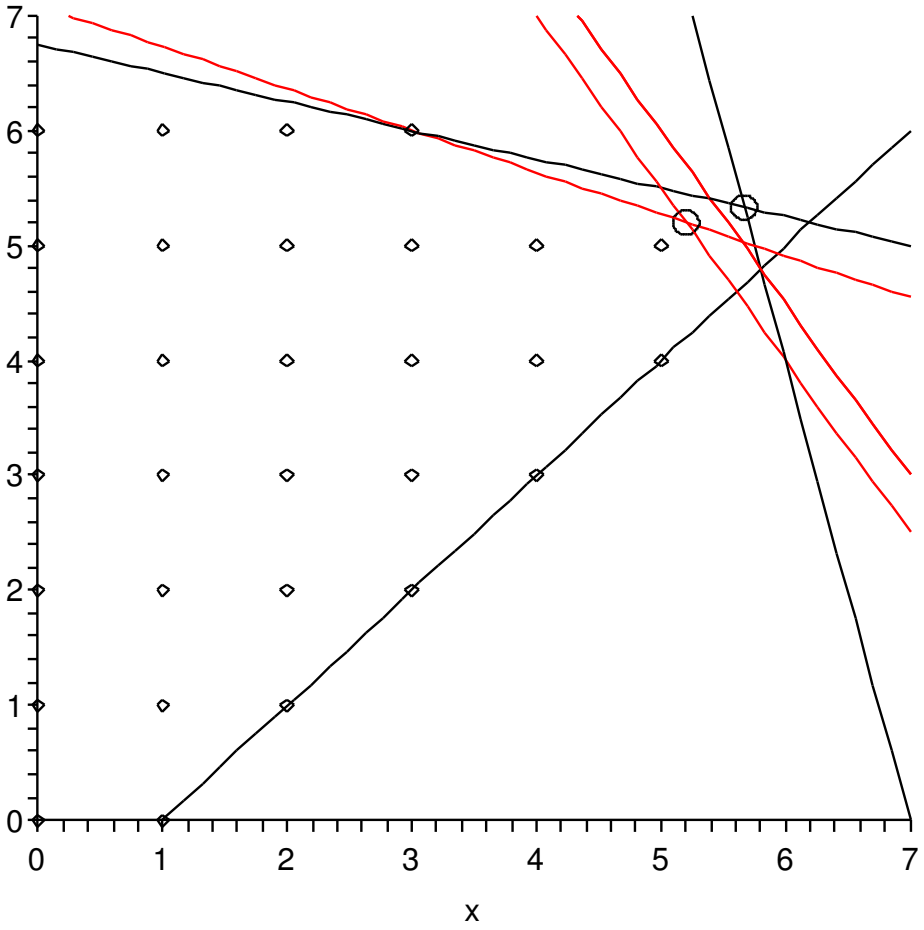

Figure 5.3: First Set of Gomory Cutting Planes

cuts. Notice that the exact code produces the reaches 100% ROP after 3 iterations and after just 4 iterations it yields the correct solution of $x_1 = x_2 = 5$ with an objective value of -25. Each point in Figure 5.7 refers to the ROP value of the current Gomory iteration. Table 5.3 lists the objective value, the figure with that objective value, and the ROP value.

CPLEX gets close but does not yield the optimal solution after going for 5 iterations. LU Mine is the most recent code that includes advancements like LU factorization of the basis matrix, steepest edge pivoting rules and strong Gomory cutting planes. Strong Gomory is the old code that used pivot matrices and most negative reduced cost rules. Combination is the old code using the combination

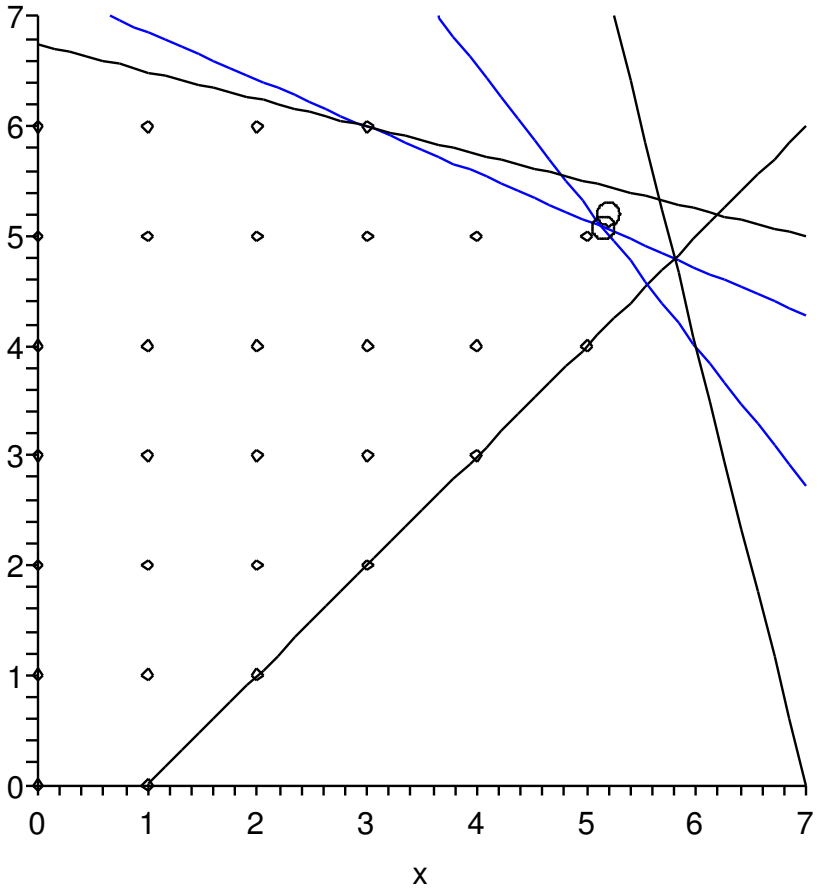

Figure 5.4: Second Set of Gomory Cutting Planes

Figure	Objective Value	ROP (%)
5.1	-27.3333	0
5.3	-26	57.1422
5.4	-25.5385	76.9211
5.5	-25	100
5.6	-25	100

Table 5.3: Objective Value and ROP for Example 6

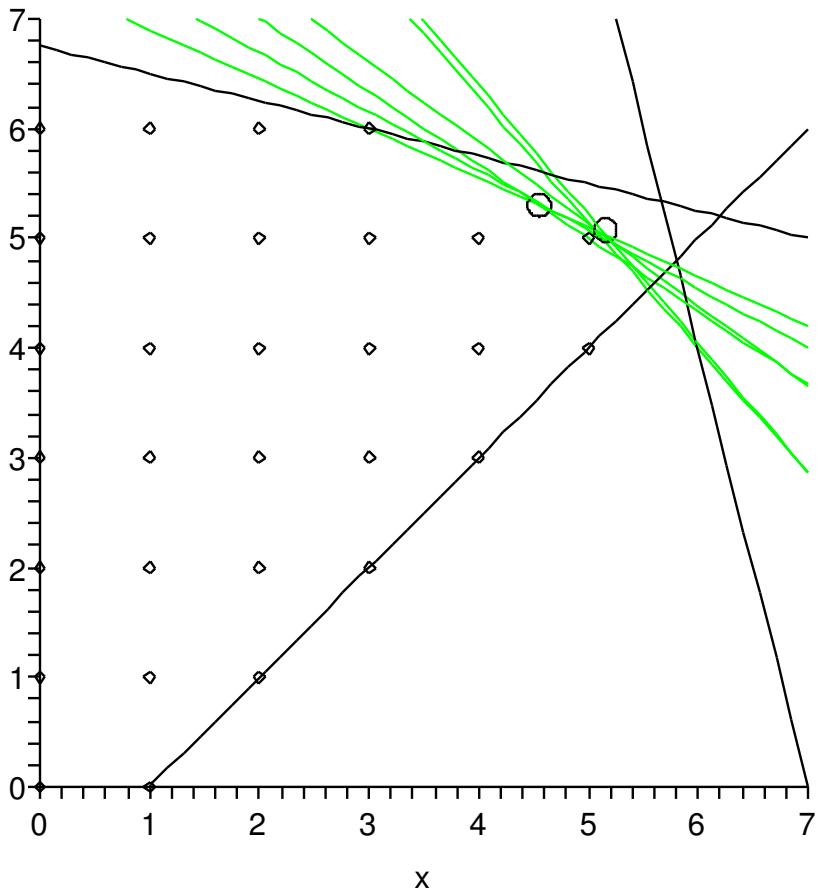

Figure 5.5: Third Set of Gomory Cutting Planes

strengthening. The CPLEX settings that we used can be found in Figures 8.1 and 8.2.

Now that we have defined the ROP and the CPLEX information, we can include the results from Example 7 found in Figure 5.8. Notice that again CPLEX cannot reach the optimal solution just using Gomory cutting planes. Also notice that both of the results that CPLEX gives has a trailing off. This trail off is the reason that cutting planes are not utilized to their full potential. Normally, cutting planes are used in a Branch and Cut algorithm to progress the solution. However, as soon as the solution starts to have this trail off then the commercial codes start branching sooner then actually needed. The cutting planes could be utilized more and we could progress much further during each subproblem and possibly reduce

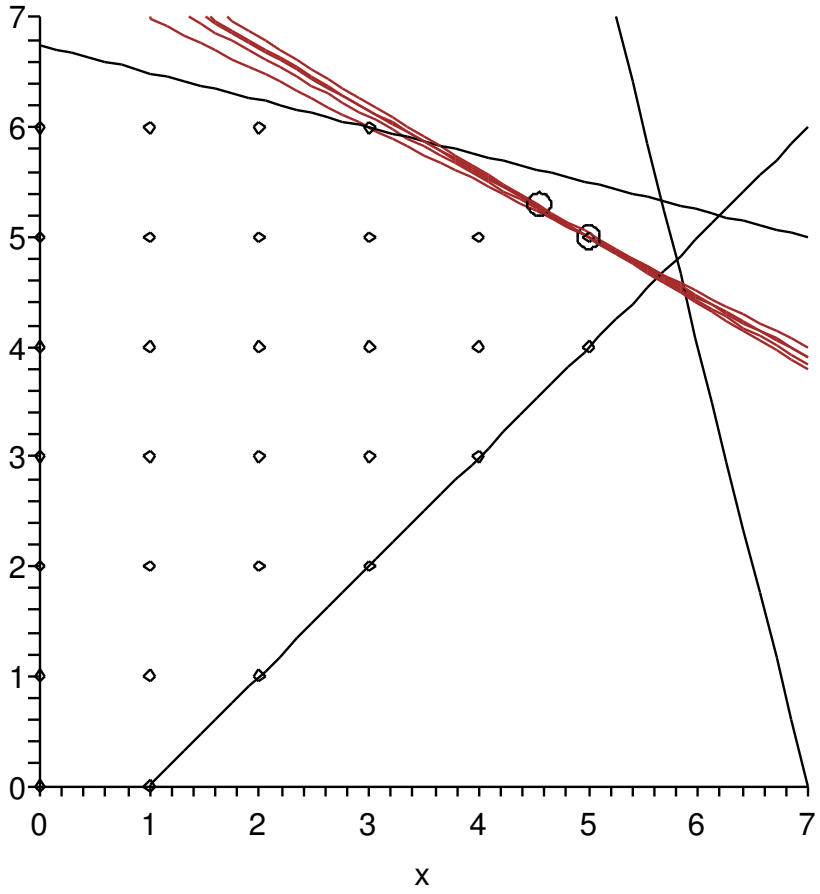

Figure 5.6: Fourth Set of Gomory Cutting Planes

the number of subproblems that needs to be solved.

5.1 Weaker Gomory Cuts

Earlier we talked about multiplying the Gomory cutting planes by the greatest common divisor and then dividing by the greatest common factor. This allows us to add in a slack variable that is also restricted to be integral. This causes very large numbers to be added in to the simplex tableau. So we could just slightly weaken the constraints by dividing by power of ten and perform Gomory rounding (since all of the variables still have to be integral). Let us suppose we have the following Gomory cutting plane.

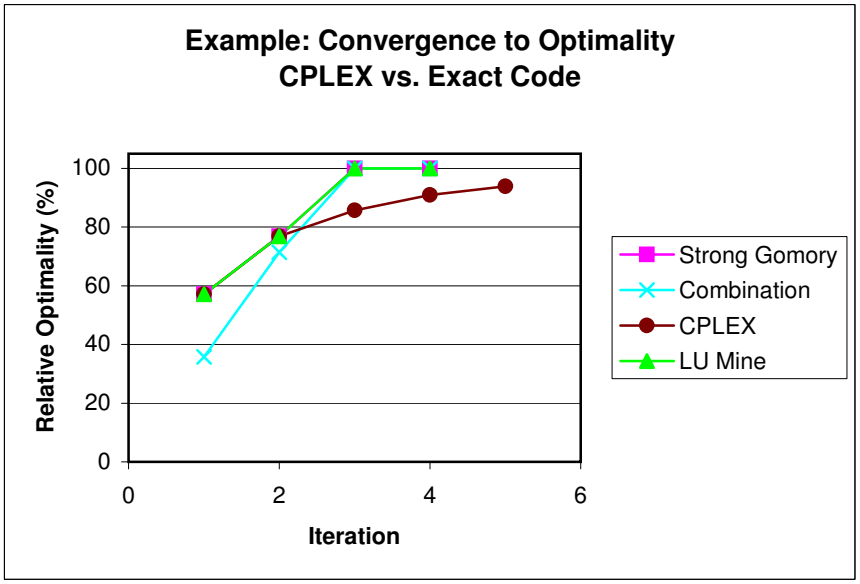

Figure 5.7: Results of CPLEX vs. Exact Code for Example

$$-649532 = -450301x_1 - 500477x_2 + s_1 \tag{5.18}$$

Let us divide this equation by 100 and perform the Gomory rounding. Then we would get this inequality.

$$-6496 \geq -4504x_1 - 5005x_2 \tag{5.19}$$

It seems like this would not be a terrible approximation to the full version. We would not necessarily have to use 100. We could use certain cutoff points like rounding everything to 50, 100 or even 300 digits. In later chapters we discuss the results of using these weaker cuts on real problems.

Besides the slacks having to be integral, another major advantage is the knowledge of the degenerate solutions. Degenerate solutions have always caused problems in linear programming and integer programming.

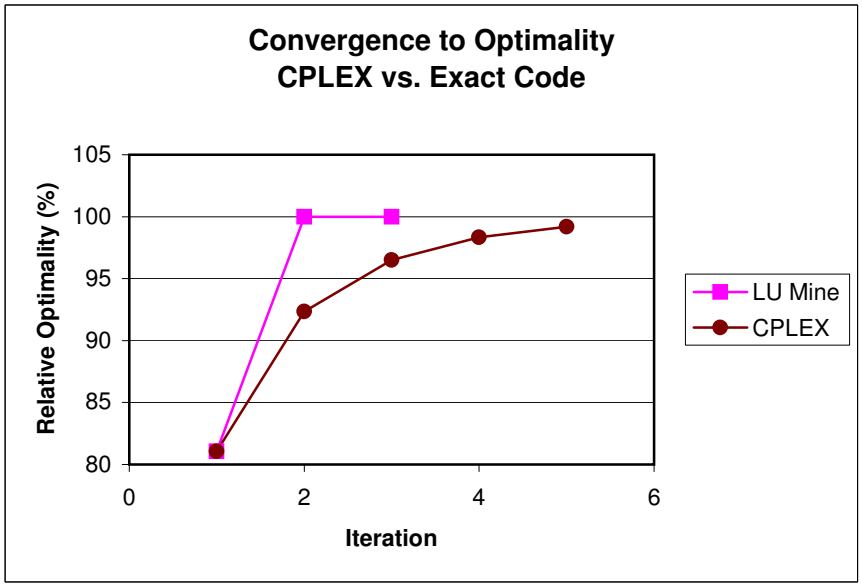

Figure 5.8: Results of CPLEX vs. Exact Code for Example

CHAPTER 6
Degenerate Pivots

We have seen that Gomory cutting planes can be used to try finding solutions to an integer programming problem. Since we are using exact arithmetic we have the advantage of knowing when we have degeneracy. Dual degeneracy can cause trouble by leaving the objective at its current value after the Gomory cutting planes have been added and the dual pivots performed. Through the use of the alternate optima we might be able to cut off more solutions and consequently reduce the number of iterations.

Definition 24 *A linear program is called dual degenerate when there is a nonbasic reduced cost that is zero after the program has reached canonical form. [9]*

However, this might be avoided by pivoting to all dual degenerate solutions (by performing just one primal simplex pivot per degenerate point) and adding in all of the appropriate Gomory cutting planes hence cutting off all degenerate points. Adding all of these Gomory cutting planes will cause an abundance of cuts that will not necessarily make a difference. To illustrate the use of degenerate pivots, the following example is provided:

Example 8

$$\begin{array}{lrcrcrl} \min & - & 7x_1 & - & 4x_2 & & \\ \text{s.t.} & & 14x_1 & + & 8x_2 & \leq & 63 \\ & & & & 2x_2 & \leq & 7 \\ & & x_1 & , & x_2 & \in & \mathbb{Z}^+ \end{array}$$

The constraint set graphically is found in Figure 6.1.

The relaxation of the integer program in tableau form is

0	−7	−4	0	0
63	14	8	1	0
7	0	2	0	1

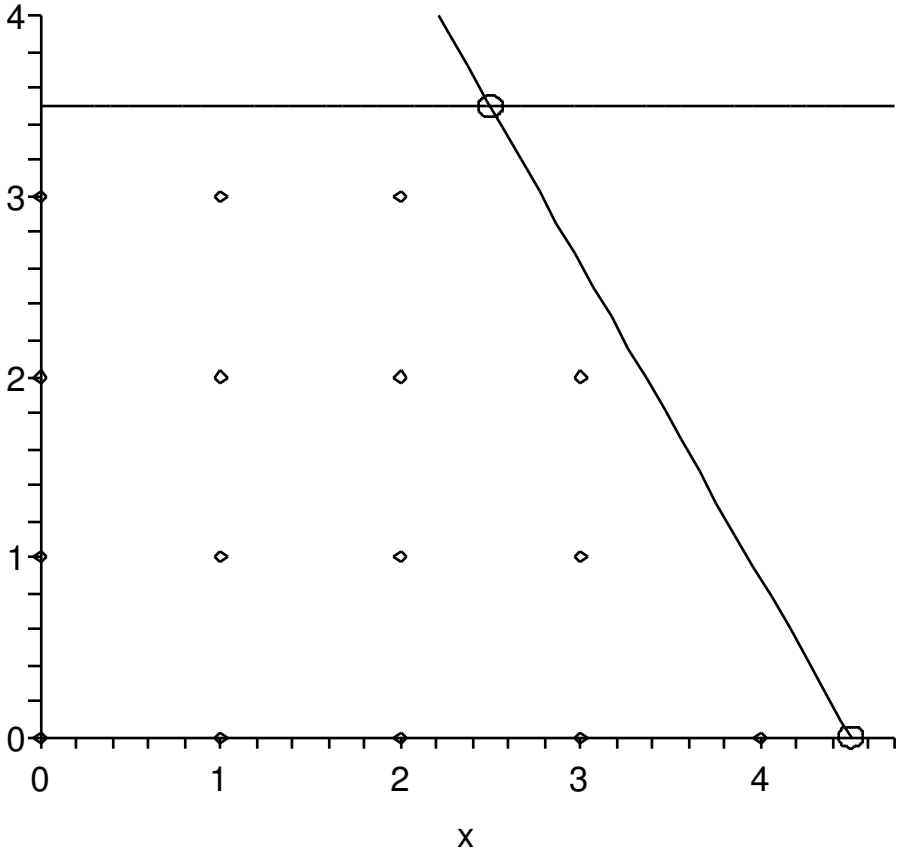

Figure 6.1: An example with multiple alternate optima

A canonical form of the relaxation is

$\frac{63}{2}$	0	0	$\frac{1}{2}$	0
$\frac{9}{2}$	1	$\frac{4}{7}$	$\frac{1}{14}$	0
7	0	2	0	1

Using the rules developed previously for finding the strong Gomory cutting planes, we get inequality (6.1) from the objective row which is the red line in Figure 6.2 and we get inequality (6.2) from the first constraint which is in green in Figure 6.2:

$$7x_1 + 4x_2 \leq 31 \tag{6.1}$$

$$7x_1 + x_2 \leq 28 \tag{6.2}$$

Pivoting on the dual degenerate variable yields the following optimal form tableau.

$\frac{63}{2}$	0	0	$\frac{1}{2}$	0
$\frac{5}{2}$	1	0	$\frac{1}{14}$	$-\frac{2}{7}$
$\frac{7}{2}$	0	1	0	$\frac{1}{2}$

The objective row is the same so it is not necessary to add the same constraint in. However, two inequalities can be formed from the other two constraints. These inequalities are in blue in Figure 6.2 and are:

$$7x_1 + 8x_2 \leq 42$$

$$x_2 \leq 3$$

Adding in these inequalities to the second tableau yields:

$\frac{63}{2}$	0	0	$\frac{1}{2}$	0	0	0	0	0
$\frac{5}{2}$	1	0	$\frac{1}{14}$	$-\frac{2}{7}$	0	0	0	0
$\frac{7}{2}$	0	1	0	$\frac{1}{2}$	0	0	0	0
-1	0	0	-1	0	1	0	0	0
-7	0	-6	-1	0	0	1	0	0
-7	0	0	-1	-4	0	0	1	0
-1	0	0	0	-1	0	0	0	1

Notice that we are no longer in canonical form. The tableau is cleaned up by pivoting on the basic variables and then performing the dual simplex to get back to optimality. Reoptimizing, notice that we again have two alternate optima and

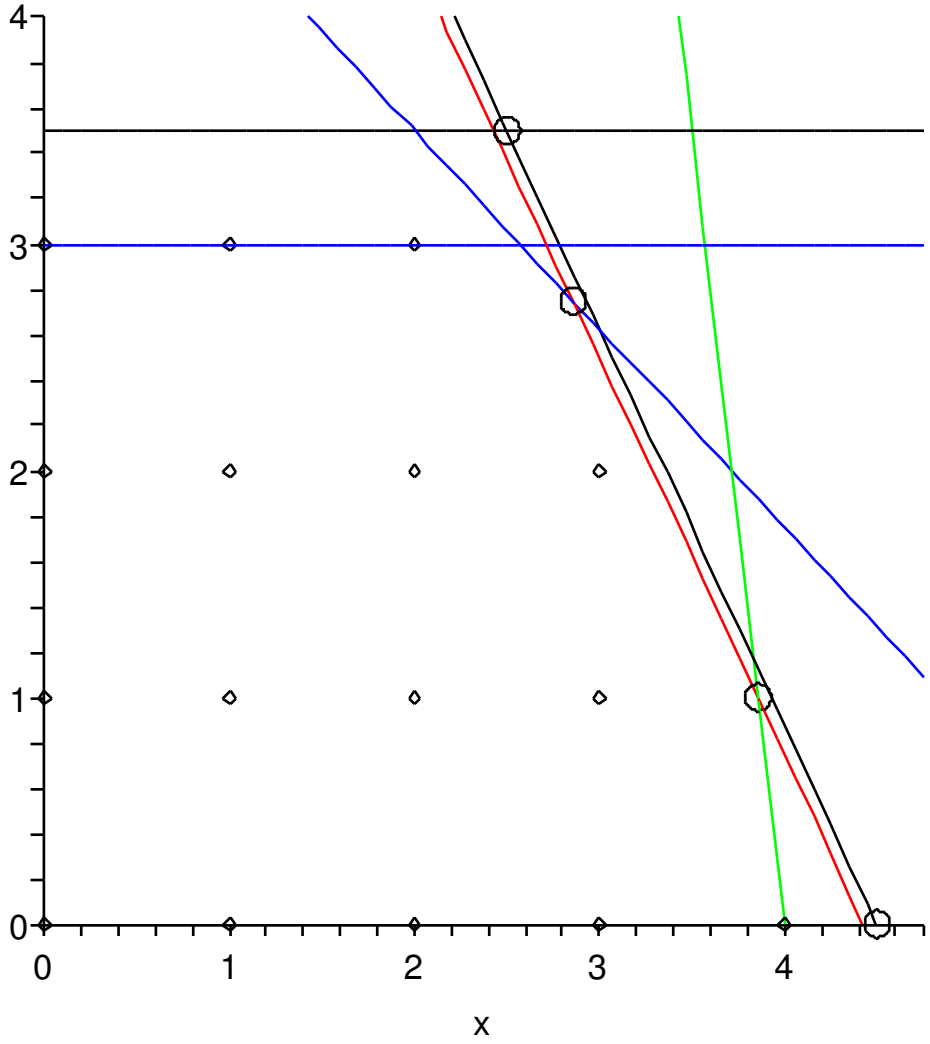

Figure 6.2: Graph after the first Gomory cuts added with multiple alternate optima

proceed as above, refer to Figure 6.2.

We continue until we find the optimal point. It only took six iterations using strong Gomory cuts and dual degenerate pivots, instead of taking 11 to 13 iterations. Also refer to Figure 6.3 to see the comparison between all of the implementations and CPLEX. CPLEX stays on the LP relaxation for most of the iterations. This use of alternate optima works extremely well for two dimensional cases. However, pivoting to all alternate optima and cutting them all off is a difficult task and it is

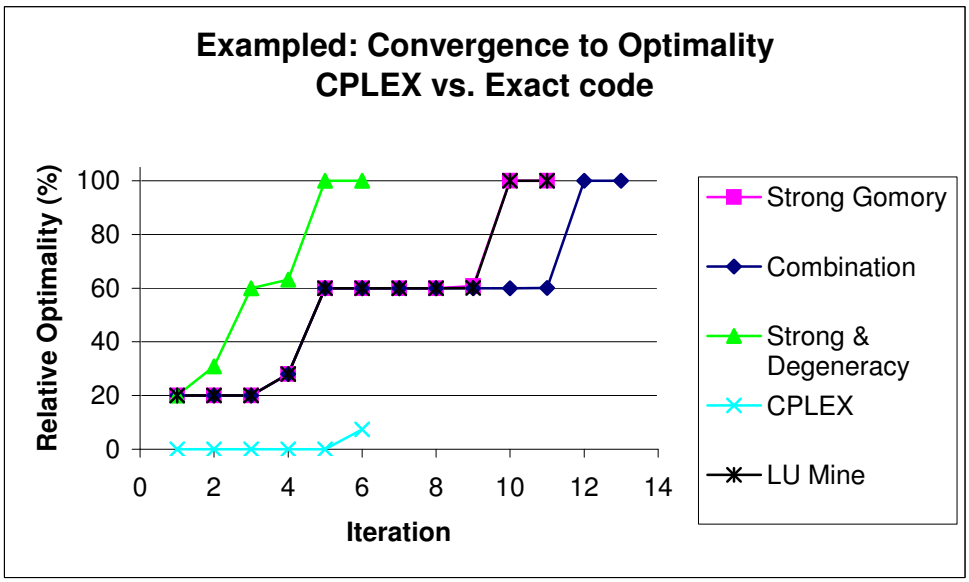

Figure 6.3: Results with multiple alternate optima

probably not necessary. We refer the reader to [10] for more information on pivoting to alternate optima. We can though simulate something that is similar to this but that does not produce the same results. It is discussed further in the next chapter.

Since we are adding in a tremendous amount of cuts, we need a way to add in the best cuts and to drop cuts that are not useful any longer.

CHAPTER 7
Adding and Dropping Constraints

Gomory cuts are linear inequalities that cut off the solution of the linear programming relaxation. It would most tighten the relaxation to add in all possible Gomory cutting planes and keep them around. However, adding and keeping all Gomory cutting planes could possibly double the size of the tableau which uses a lot of memory. Certain rules need to be used to add and drop certain constraints so that we reduce the number of dominated constraints.

One attempt for adding cuts is to add only constraints that are the dominant constraints. If we are given two inequalities that are the same except for the right hand side then the inequality that rounds further down will be tighter. For example, if we have these two rows in the tableau:

$$\frac{23}{4}x_1 - \frac{1}{8}x_2 + x_3 = \frac{11}{10} \tag{7.1}$$
$$\frac{19}{4}x_1 + \frac{7}{8}x_2 + x_3 = \frac{19}{10} \tag{7.2}$$

The standard Gomory cutting planes without strengthening would be

$$\frac{3}{4}x_1 + \frac{7}{8}x_2 \geq \frac{1}{10} \tag{7.3}$$
$$\frac{3}{4}x_1 + \frac{7}{8}x_2 \geq \frac{9}{10} \tag{7.4}$$

Notice that (7.3) is dominated by (7.4). So, we could actually just add in equation (7.4). However in general, it is hard to tell when one constraint is dominant over another unless we do an element by element comparison. Also, with strengthening, the constraints cannot be compared. Inequality (7.3) has a lower right hand side so both of the Gomory strengthening techniques would greatly improve this constraint. So, we have decided that we add in all constraints and try to drop the

most appropriate ones later.

Dropping constraints is vital to the code because if we do not, the constraint matrix grows too big to be stored. For this reason, we have included code for dropping constraints. One rule used to drop a constraint is if the slack variable is basic and is larger than a specified value. This value is generally zero since we want to drop as many constraints as possible to avoid memory issues. For large slack values it usually means that that constraint is not reducing the feasible region or that the current solution is far away from the constraint. So most likely these constraints are not currently relevant.

Another rule used to drop constraints is if one constraint is a multiple of another, see [1]. It is difficult to tell if two constraints are exact multiples of each other unless one does an element by element comparison of the two constraints. However, if one of these constraints happens to be pivoted on then it is easy to tell. If a constraint has a right hand side of zero and only two elements in that row then we know that they were multiples of each other. For example, suppose we have these two rows somewhere in our constraint matrix:

$$x_1 - 5x_2 + x_3 + s_1 = 6 \tag{7.5}$$

$$3x_1 - 15x_2 + 3x_3 + s_2 = 18 \tag{7.6}$$

If we happen to pivot on one of these rows, e.g. x_3 of the first constraint, then the two constraints become

$$x_1 - 5x_2 + x_3 + s_1 = 6 \tag{7.7}$$

$$0x_1 + 0x_2 + 0x_3 - 3s_1 + s_2 = 0 \tag{7.8}$$

The new second constraint only has two elements and it is equal to zero so we can drop this constraint since it is redundant. Recall that (7.6) in the inequality formulation is just 3·(7.5) in the inequality formulation and notice that (7.8) states that $s_2 = 3s_1$ which is the same thing. Since we are using YSMP format then to

check whether or not a row has two elements and the right hand side is equal to zero is very easy. If the difference between successive elements of the row vector is 2 then that row only has two elements.

In general, we could drop any constraint that does not include any original variables, and if the coefficients of all of the nonbasic variables are nonpositive. Then the basic slack variable is just a nonnegative linear combination of the other added slack variables plus a nonnegative value. However, we would need to perform an element by element search of this row to determine whether we could drop this or not. So, we do not drop constraints that have this form.

There is a time when we do not drop any constraints. When there is no decrease in the objective function then we should not drop cuts. For example, see Figures 7.1 - 7.4. Here is an example where if we drop constraints then we will never have a decrease in the objective function. Notice that the objective value is the same for all of the extreme points. Suppose that we find the LP Relaxation solution to be in the upper left hand corner of Figure 7.1. Now let us say that we find a cutting plane given in Figure 7.2. We cutoff the first solution and now resolving gives us another solution. It could have been either solution so without loss of generality, let us say that it is the solution plotted in Figure 7.2. Continuing in this manner, we cutoff this second solution with another cutting plane shown in Figure 7.3. Notice that the first cutting plane is no longer active. We would normally drop this first constraint. If we perform another such iteration then we would cutoff the next solution with a cutting plane shown in Figure 7.4 and get another solution. If we continue this for a number of iterations, we could end up right back at the beginning with having to cutoff points that were already cutoff from an earlier cutting plane. This is a sort of cycling for Gomory cutting planes.

By not dropping any cuts when the objective value is the same will essentially produce similar results to the degenerate case without having to pivot to every possible alternative optimal solution. So we cannot drop cuts that all have the same objective value. Thus we must either pivot to all of the points and cut them all off or we must not drop any cuts with the same objective value. For smaller problems either of these ideas would work extremely well. However, for larger problems both

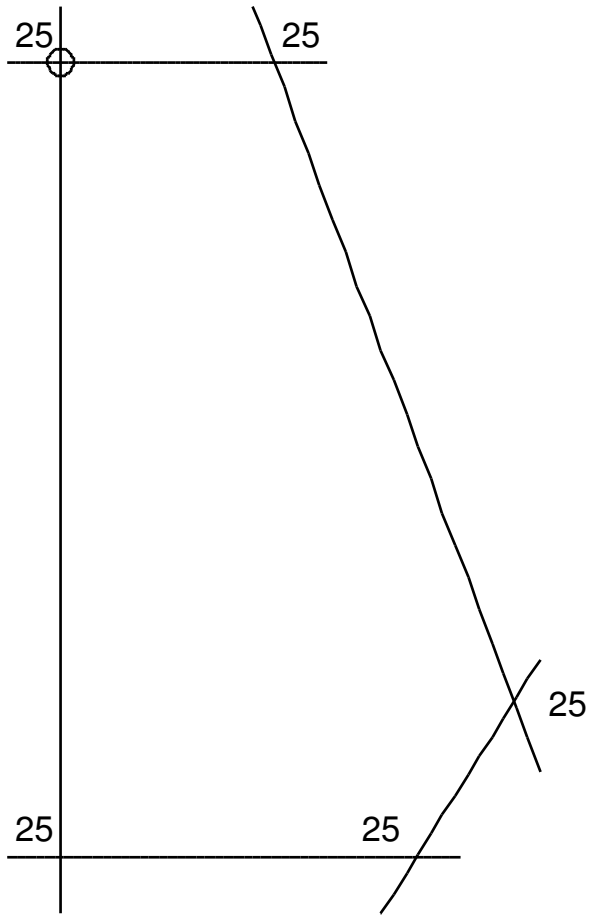

Figure 7.1: Example with dropping

of these situations are not good. If we pivot to all of the alternate optima then we could be doing a lot of extra work. If we do not drop any cuts then we could end up exponential growth of the size of the simplex tableau. What we have decided to do is to not drop any cuts for three iterations if the objective value is the same. A couple of the MIPLIB examples solve in a later chapter has this exact property and this worked well.

We have discussed many examples so far, but most of them were small examples. What happens when we extend these ideas on real world problems? We will

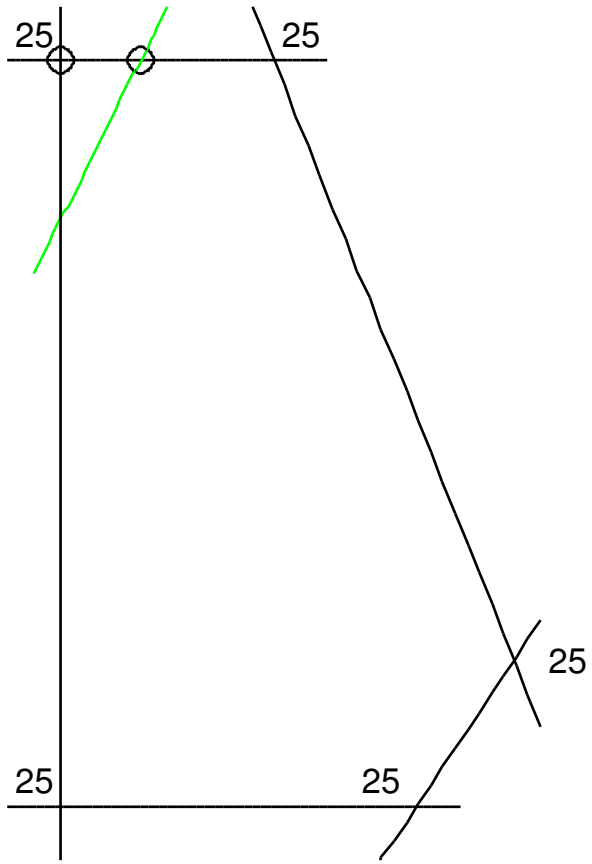

Figure 7.2: Example with dropping

show results of our implementation on a library of test problems known as MIPLIB.

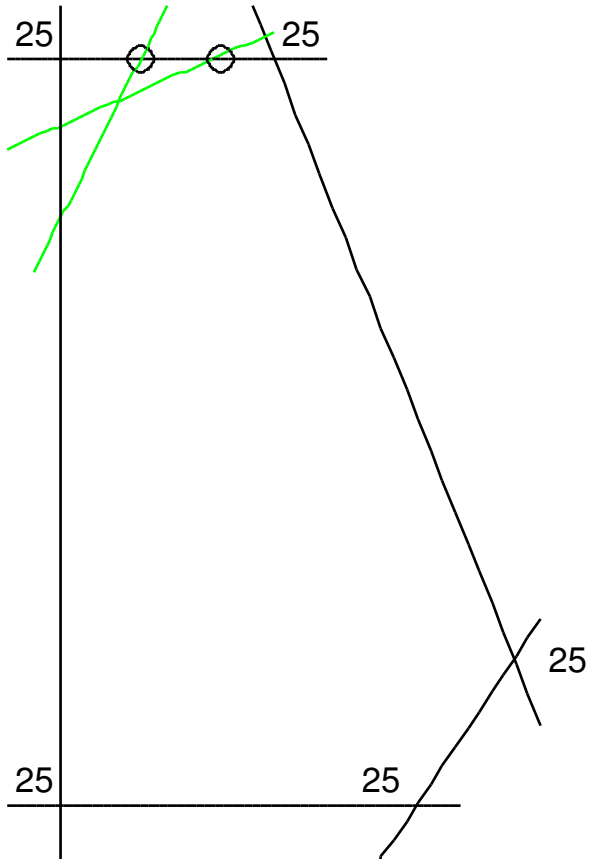

Figure 7.3: Example with dropping

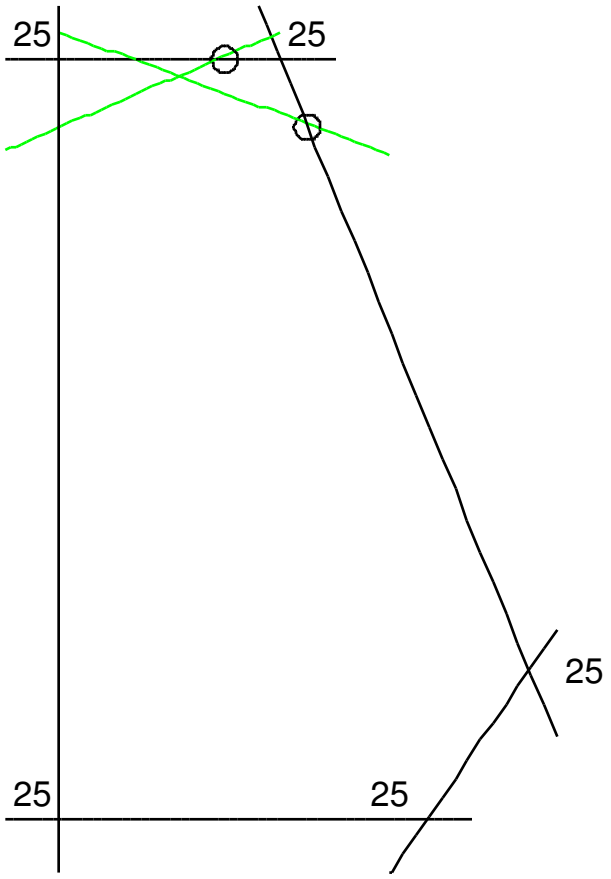

Figure 7.4: Example with dropping

CHAPTER 8
Results

Previously we have seen very small examples that had special properties that exploit the abilities of exact arithmetic. We want to compare our code with some commercial software that has Gomory cutting planes included. We decided to use CPLEX 9.0 for this comparison. CPLEX is a widely used program to solve all types of operations research type problems. CPLEX also allows users to change certain settings. We wanted to compare apples to apples so the settings that we have chose for CPLEX are in Figure 8.1.

1. $set \rightarrow mip \rightarrow cuts \rightarrow all \rightarrow -1$

   This will turn off all cuts for integer programming problems.

2. $set \rightarrow mip \rightarrow cuts \rightarrow gomory \rightarrow 1(2)$

   This will turn on conservative (aggressive) Gomory cuts. Since they are using floating-point arithmetic we think that aggressive Gomory cuts are ones that will perform rounding that is closer to integers than conservative Gomory cuts.

3. $set \rightarrow mip \rightarrow limits \rightarrow nodes \rightarrow 1$

   This setting will use only one Branch and Bound node. We just want to know how well Gomory cuts perform so we turn off the branching.

4. $set \rightarrow preprocessing \rightarrow presolve \rightarrow n$

   This setting will turn off the presolver. We want to be solving the same problem as CPLEX so we do not allow CPLEX to get any extra advantage by getting rid of extra variables or rows.

5. $set \rightarrow mip \rightarrow strategy \rightarrow heuristicfreq \rightarrow -1$

   This setting will turn off the heuristics that find solutions to integer programming results without any theoretical backing.

Figure 8.1: CPLEX Settings

In order to gather the necessary results per iteration, we made two additional settings to CPLEX found in Figure 8.2.

1. $set \rightarrow mip \rightarrow display \rightarrow 2, 3, 4, or 5$

This will display LP information. The higher the number will result in more information about the LP. We generally use 3 because this number gives us the necessary information without anything extra.

2. $set \rightarrow simplex \rightarrow display \rightarrow 2$

Since we are interested in recording where the objective value is this will display the objective value after each iteration.

Figure 8.2: CPLEX Settings Data Gathering

We needed some larger problems to test whether our code was worth pursuing. We chose to use the library MIPLIB (Mixed Integer Programming LIBrary). MIPLIB is a library of difficult mixed integer programming problems which are based on real life examples. There are three versions of MIPLIB available, MIPLIB, MIPLIB 3.0, and MIPLIB 2003. All of the problems used in this research can found at http://miplib.zib.de/. We have gathered some of the smaller examples that are pure integer programming problems. We choose smaller problems because we know that exact code will have its limits and we are not interested in mixed integer programming problems. Table 8.1 is a list of the problems that were used.

The definitions of the column headings in Table 8.1 are contained in Table 8.2. Note all of the problems are pure binary problems except GT2.

As in an earlier chapter in order to compare the codes, we are going to use the relative optimality percentage as the measure of how good all of the codes are. Table 8.3 compares all of the codes with CPLEX.

The definitions of what the column headings for Table 8.3 are listed in Table 8.4.

First thing to notice in Table 8.3 is that for the problems that are not found to be 100% that CPLEX is never among the best codes. It is always either Old code with Strong Gomory cutting planes or the new code with LU factorization/Steepest edge pivoting rules and Strong Gomory cutting planes. Notice that LU code also

Name	Rows	Cols	LP Relax	Optimal	Difference	% Difference
BM23	20	27	20.5709	34	13.4291	65.28
GT2 *	29	188	13460.2331	21166	7705.7669	57.25
LSEU	28	89	834.6824	1120	285.3176	34.18
MOD008	6	319	290.9311	307	16.0689	5.52
P0033	16	33	2520.5717	3089	568.4283	22.55
P0040	23	40	61796.55	62027	230.45	.37
P0201	133	201	6875	7615	740	10.76
P0282	241	282	176867.5033	258411	81543.4967	46.1
P0291	252	291	1705.1288	5223.749	3518.6202	206.36
PIPEX	25	48	773.751	788.263	14.512	1.88
SENTOY	30	60	-7839.278	-7772	67.278	.87 **

Table 8.1: MIPLIB problems

Name:	Name of the problem
	* All of the problems are binary except GT2
Rows:	Number of rows of the original problem
Cols:	Number of columns of the original problem
LP Relax:	Objective value of the linear programming relaxation
Optimal:	Objective value of the optimal value
Difference:	Difference between the LP Relaxation and the Optimal Value
% Difference:	Percent difference to the LP Relaxation
	** GT2 has the percent difference to the Optimal

Table 8.2: Definition Table for Table 8.1

Name	CPLEX	Strong/Old	Combination/Old	LU/Steepest/Strong
BM23	26.8402	**32.9813**	**32.9813**	**32.9813**
GT2	99.3676	**100**	**100**	**100**
LSEU	67.0647	71.3376	71.6106	**73.442**
MOD008	28.3871	**43.9912**	31.5449	31.5449
P0033	70.1518	**75.8985**	60.3786	72.7318
P0040	100	100	100	100
P0201	14.6172	39.6159	33.6377	**44.0009**
P0282	14.7679	16.2835	15.3754	**18.4281**
P0291	85.9277	87.1634	89.1826	**92.8621**
PIPEX	54.6189	49.9516	43.0607	**56.8424**
SENTOY	25.7474	**27.1679**	26.2819	25.8417

Table 8.3: Relative Optimality Percentages

Name:	Name of the problem
CPLEX:	Highest relative optimality percentage reached by CPLEX
Strong/Old:	Highest relative optimality percentage reached by the Old Simplex code written. This code did not use LU factorization and it used most negative reduced cost. The strong refers to using only strong Gomory cuts.
Combination/Old:	Same as Strong/Old but it only used T-strong Gomory cutting planes.
LU/Steepest/Strong:	Highest relative optimality percentage reached by the new Simplex code. This code uses LU factorization and steepest edge pivoting rule. It only uses Strong Gomory cuts.

Table 8.4: Definition Table for Table 8.3

only gets beat by CPLEX one time and that is SENTOY and the LU code is less than .2% away from CPLEX.

Secondly, notice that the combination of strong and t-Gomory cutting planes is not as good as just the strong ones. We have included these as well for completeness.

We noticed that CPLEX decides to stop after only 5 iterations each time. We stop because of a couple of reasons. One reason is that the code is running too long. This usually happens when either the fractions get too large and/or the tableau grows too big so we stop the code. The code could also stop if we are getting too close to the upper bound of the number of non-zero elements. The memory for this code is immense as demonstrated by how large the fractions grow so we set a limit of 35000 non-zeros. This is part of the reason we cannot solve any larger problems than the ones chosen.

We broke up the problems into groups that we thought had the same properties.

8.1 P0040

P0040 is a very boring problem. Each of the codes reach a value of 100% of optimality. Also, they all reach optimality after just one iteration. P0040 is too trivial to even include any graphs or anything.

8.2 GT2

GT2 has an interesting feature. Before CPLEX 9.0, none of the earlier versions of CPLEX could solve this problem to optimality even using the full blown version of CPLEX. Now it solves this problem to optimality only using Gomory cutting planes. The results in graphical format are in Figure 8.3. Our code does not find the optimal point but it reaches the optimal objective value.

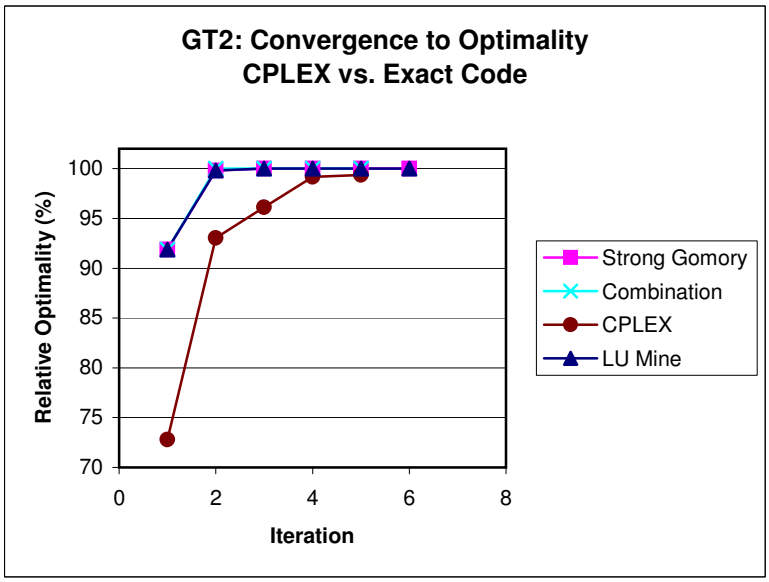

Figure 8.3: Results of CPLEX vs. Exact Code for GT2

8.3 MOD008

For MOD008 both of the LU code and the Combination get stuck at the objective value of 296. CPLEX does not even reach 296. Where the Old code seems to do really well. This problem has some degeneracy so certain pivots will definitely do better than other pivots. See Figure 8.4 for the results.

8.4 BM23

This next problem BM23 has a similar feature to P0033 and PIPEX. They all get stuck on a certain integer objective value and stay there for awhile. BM23 gets stuck on the objective value of 25 and ROP value of 32.98% and does not proceed further as shown in Figure 8.5. Note that all of our code is better than CPLEX.

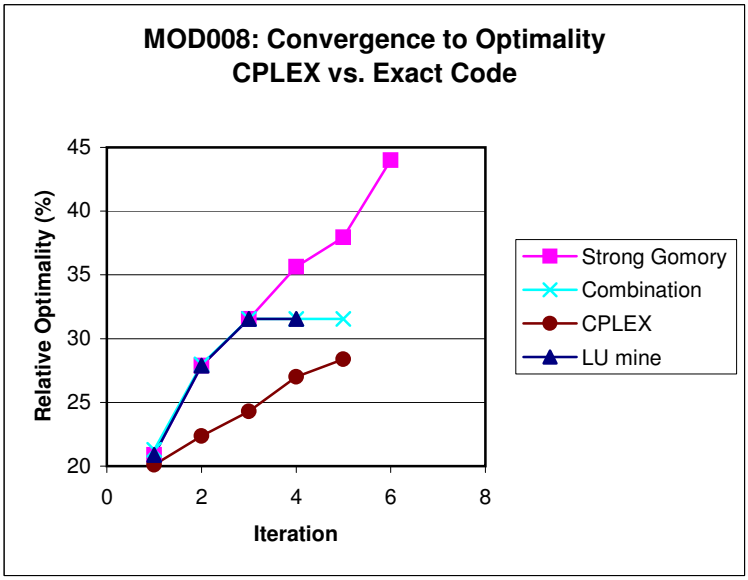

Figure 8.4: Results of CPLEX vs. Exact Code for MOD008

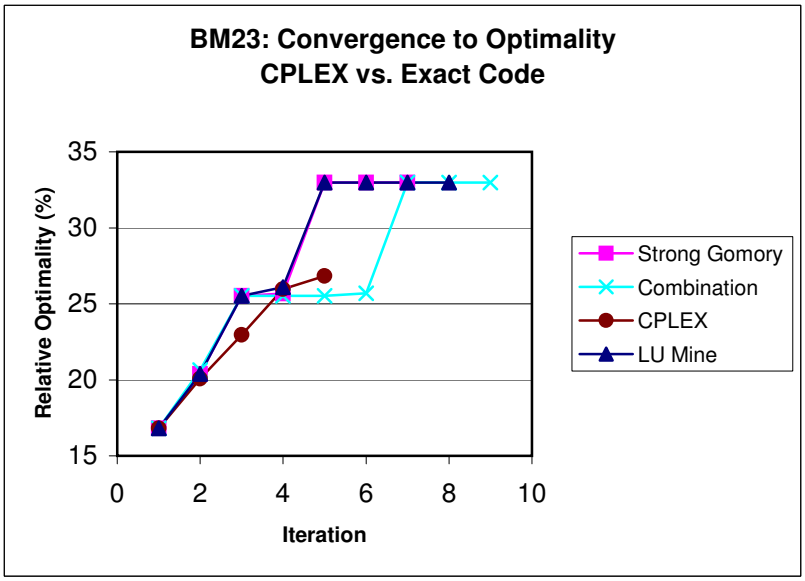

Figure 8.5: Results of CPLEX vs. Exact Code for BM23

Also, we have included the graph of the weak cuts. If we reduced the Gomory cutting planes to just 100 digits and perform the weakening described in **??** we get the results given in Figure 8.6. Notice that the weak cuts even to within 100 digits of accuracy is not enough to compete with the full version. For a cutoff of 300 digits it looks like it is the same as the full version, unfortunately it is not. The 300 digit cutoff is actually just below the full version but because of the representation of

floating-point numbers it is too hard to tell on the graph.

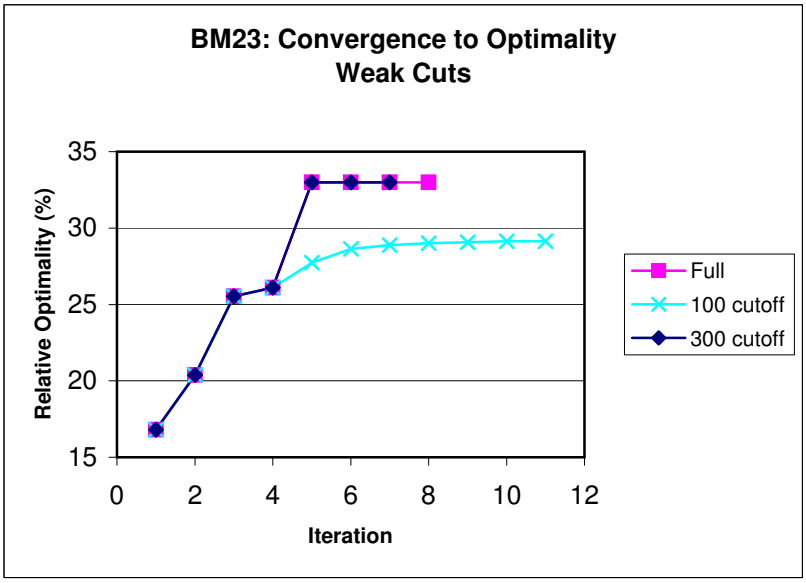

Figure 8.6: Results of Weak Cuts for BM23

Another interesting feature of this is the found in Figure 8.7 and Figure 8.9. Notice that the number of digits for the cutoff levels are below 300 for obvious reasons and the full version keeps growing exponentially. This seems reasonable since the fractions are growing in the tableau. Each dash represents the average length of the non-zero elements of each Gomory cutting plane added.

Since the length of the numbers in the Gomory cutting plane are growing, one would presume that the maximum y-value would grow also. The Maximum y-value when we are solving for the equation $By = A_{jp}$ in the dual. Note that the maximum y-value for each simplex iteration for when the 300 digit Gomory cutting planes are used are more than the full Gomory cutting planes as seen in Figure 8.11 and Figure 8.12.

8.5 P0033

P0033 is similar to BM23 in that it gets stuck in two places with objective value of 2925 and 2934 or 71.15% and 72.73% respectively for the LU code as shown in Figure 8.13. The CPLEX and the LU code are virtually the same for the first 5 iterations then all of the codes seem to trail off. The combination cuts do extremely

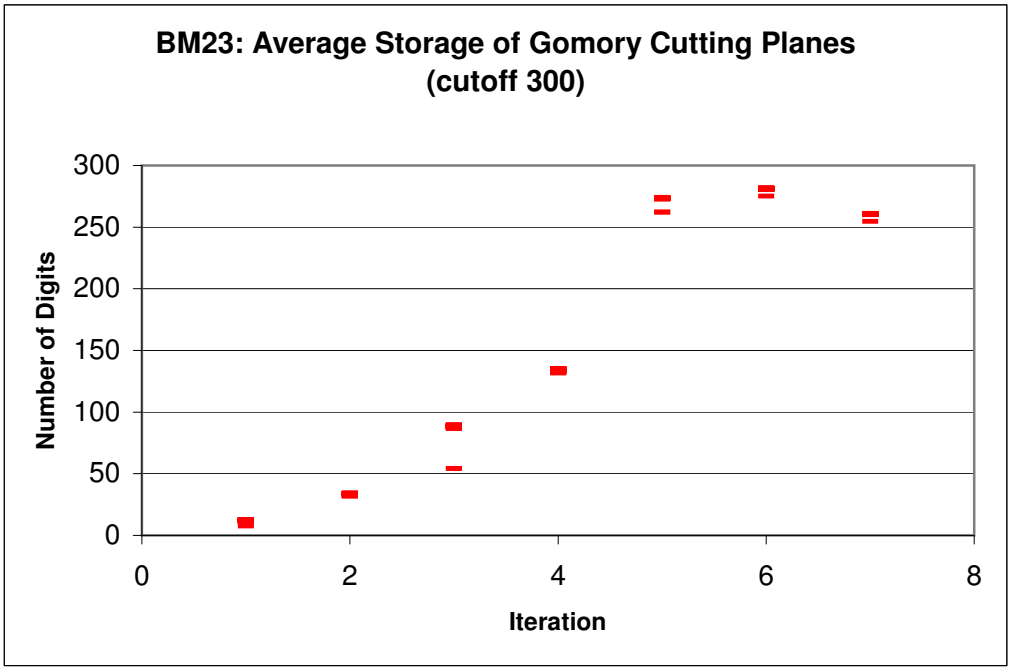

Figure 8.7: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem BM23 with Cutoff 300

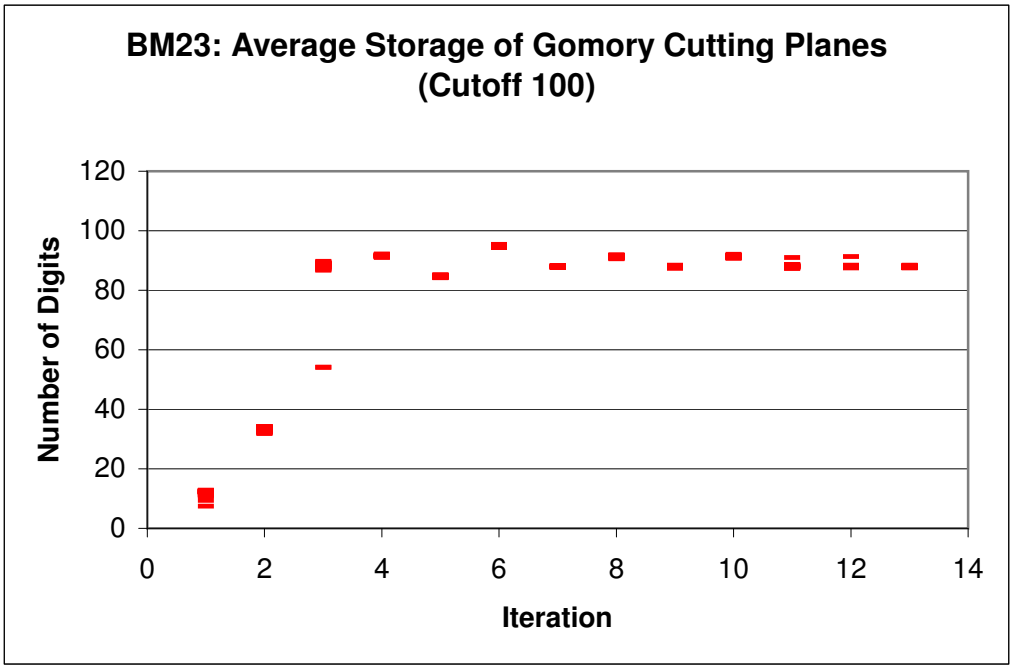

Figure 8.8: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem BM23 with Cutoff 100

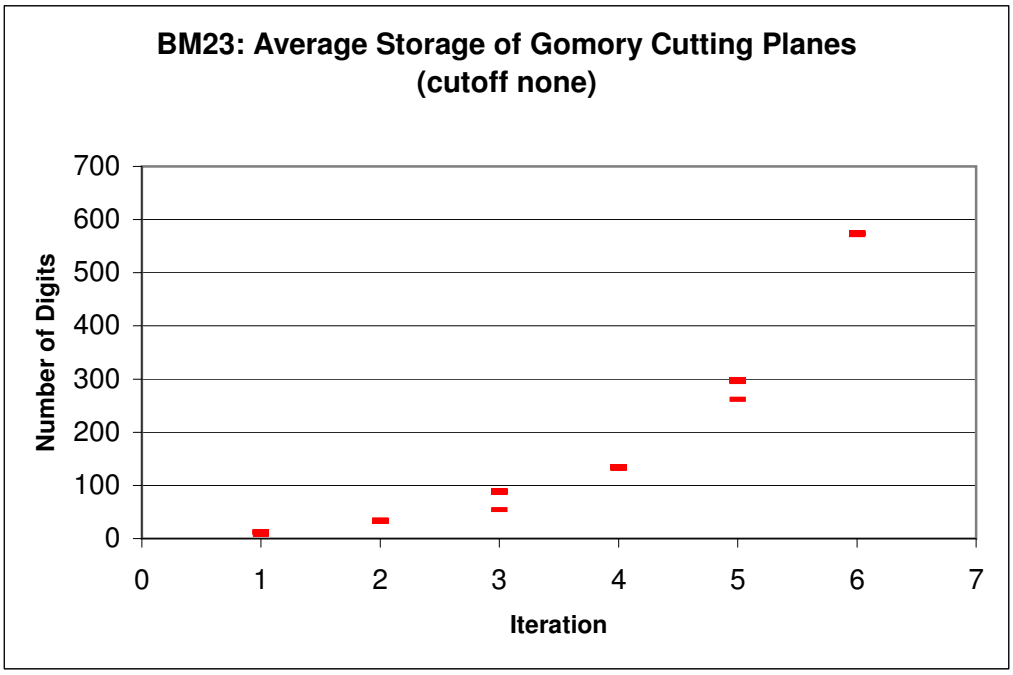

Figure 8.9: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem BM23 with no Cutoff

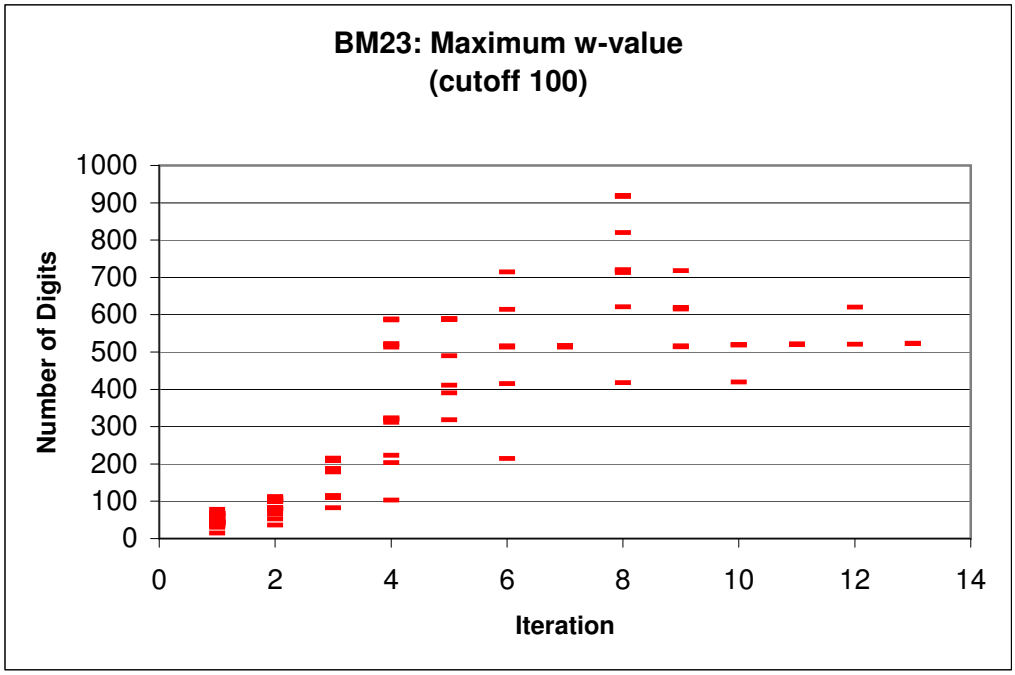

Figure 8.10: Maximum size of the element of w-vector for problem BM23

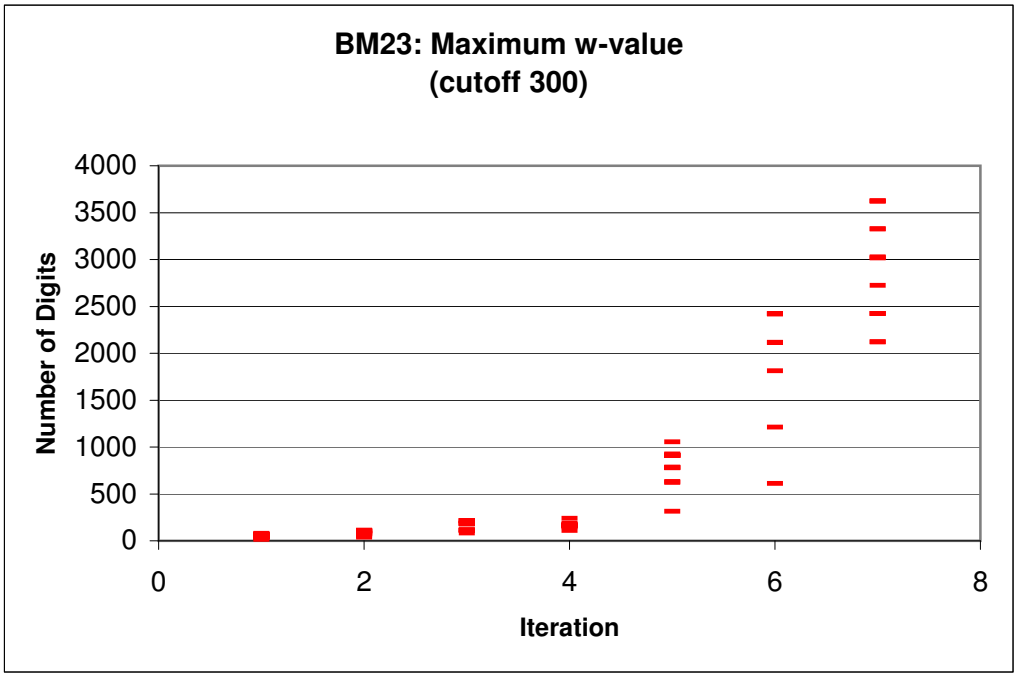

Figure 8.11: Maximum size of the element of w-vector for problem BM23

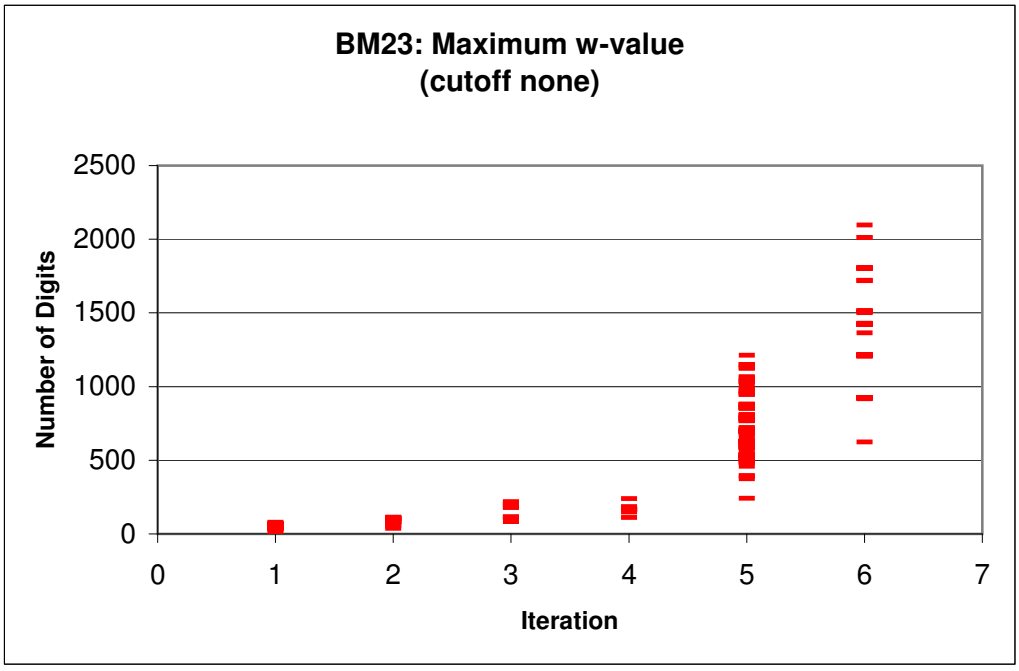

Figure 8.12: Maximum size of the element of w-vector for problem BM23

poorly for this situation. This problem has some degeneracy so we had to play around with the dropping of constraints. The default is to not drop any Gomory cutting planes for 3 Gomory iterations if the objective function has no change. We choose 3 because usually the tableau would just be too big after that point for this size of problem. If we are working with smaller problems then a default could be much higher.

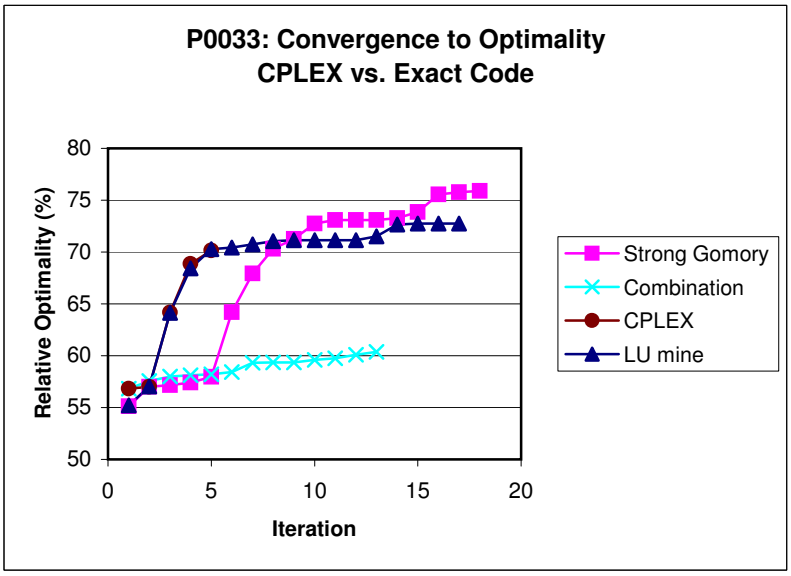

Figure 8.13: Results of CPLEX vs. Exact Code for P0033

We have included the results found in Figures 8.14 and 8.15. Again the 100 cutoff has a trail off that is uninteresting but notice the none cutoff one has a dip at about iteration 12. At iteration 12 is where there is a slight move away from the objective value that it was stuck on and then a lot of the cuts get dropped because of the lack of tightness in them.

Let us now look at Figures 8.16 and 8.17. Notice the dip at iteration 13 of Figure 8.17. The dip in the Gomory graph was at 12. The reason for this is that after iteration 12 all of those cutting planes are dropped and we had in some relatively small fractions at the 13th iteration. So the eta matrices do not get so big right at that iteration. However, we go back to exponential growth right afterwards.

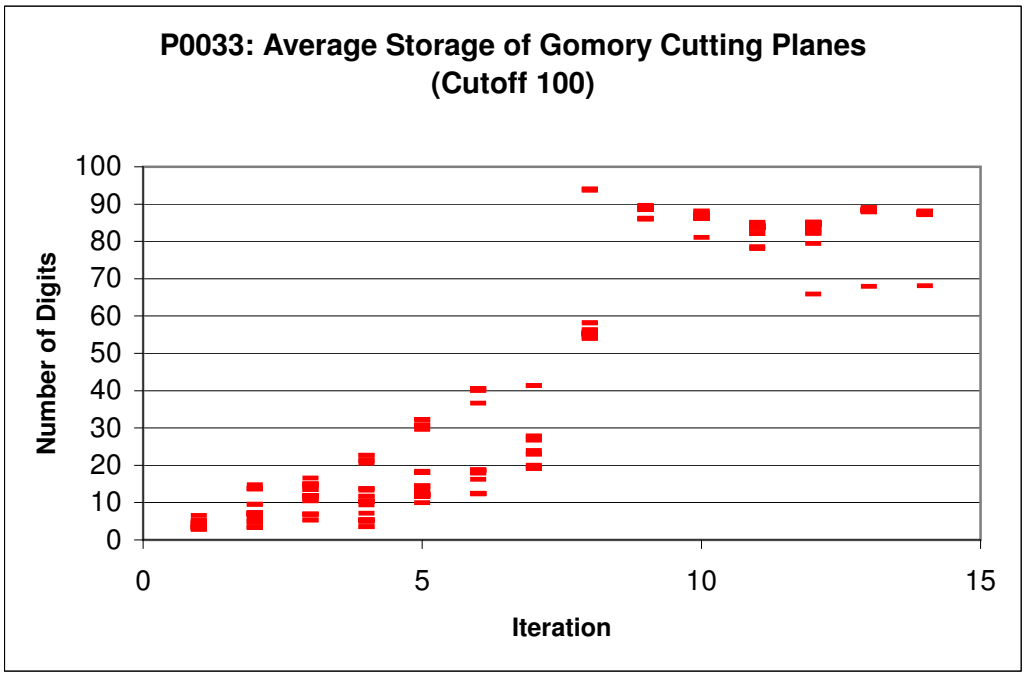

Figure 8.14: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem P0033

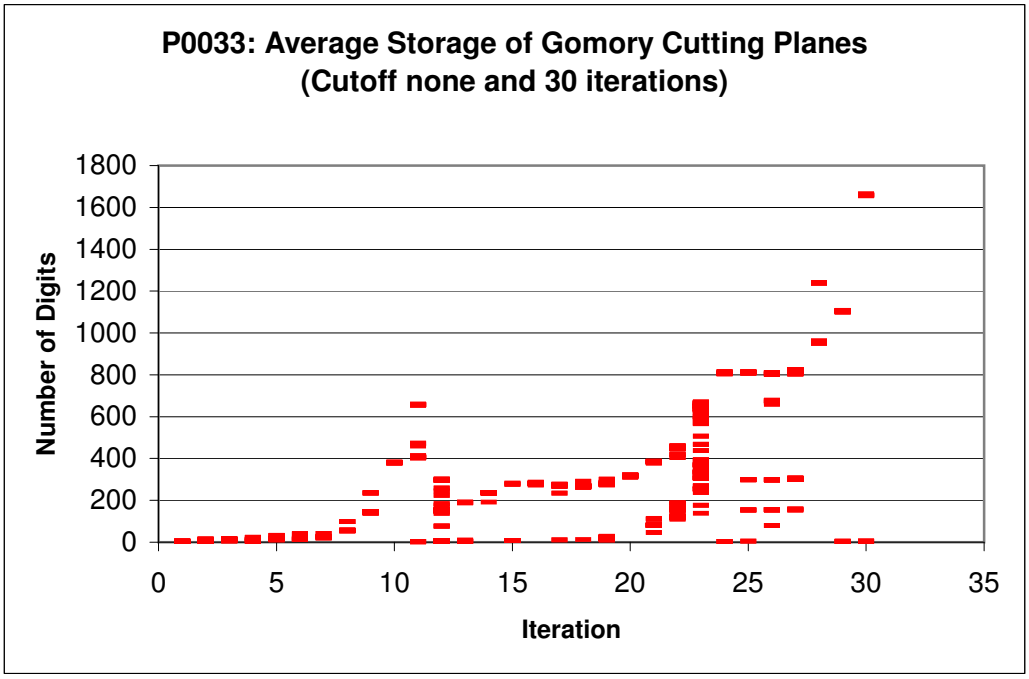

Figure 8.15: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem P0033

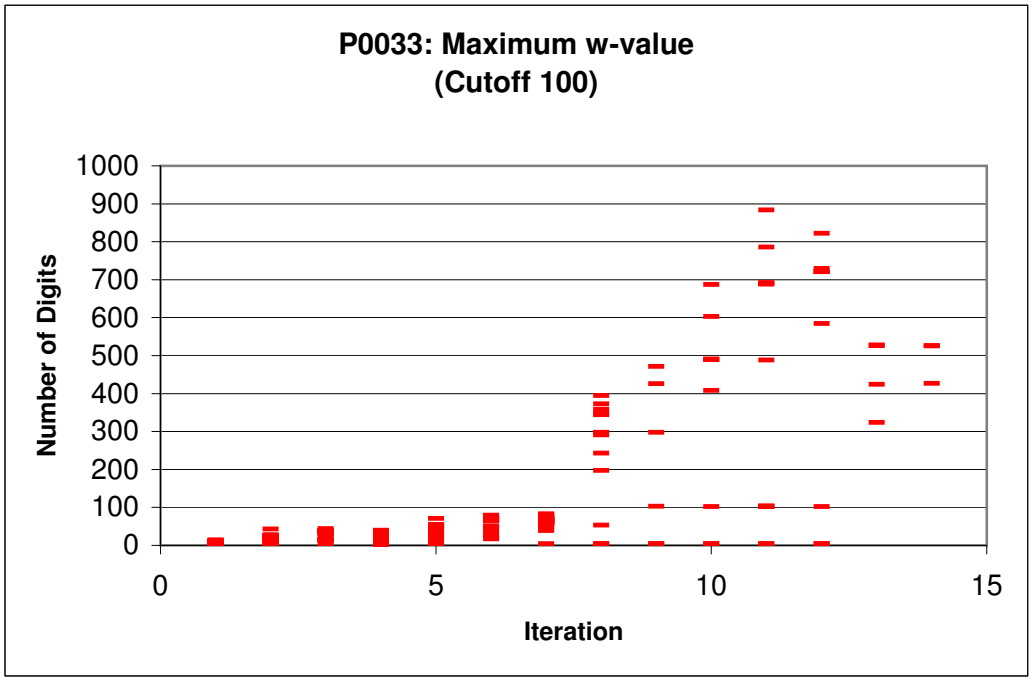

Figure 8.16: Maximum size of the element of w-vector for problem P0033

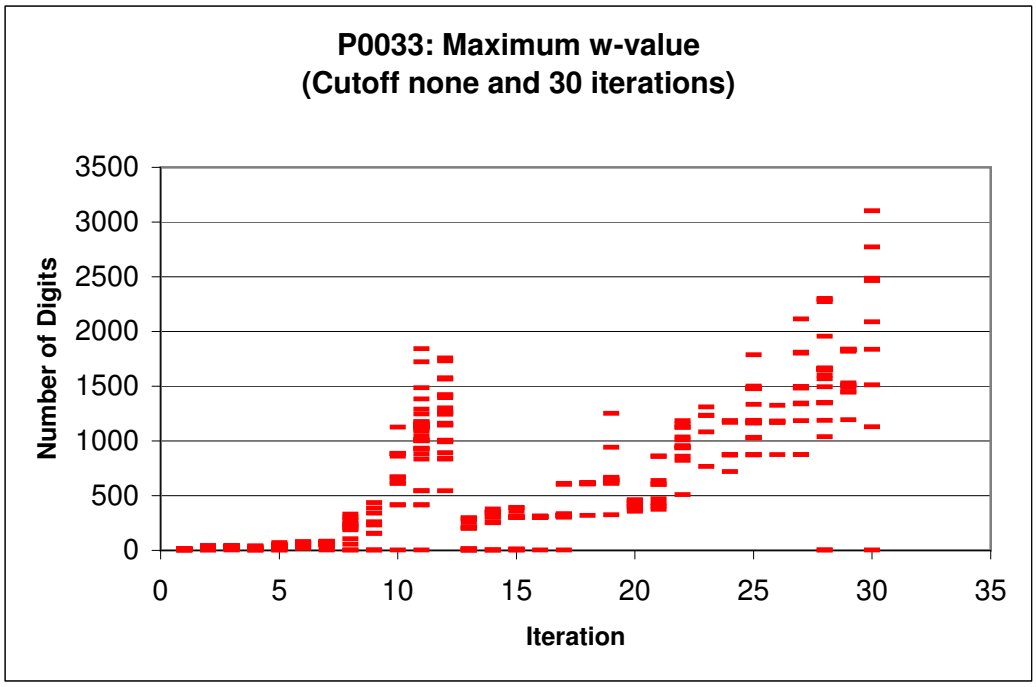

Figure 8.17: Maximum size of the element of w-vector for problem P0033

8.6 PIPEX

PIPEX gets stuck at both 781 and 782 objective values or 49.95% and 56.84% respectively for the LU code as shown in Figure 8.18. PIPEX is the only problem where CPLEX does better if a comparison is done with the final iteration of CPLEX. For example, CPLEX stops after 5 iterations for this problem and CPLEX is better at iteration 5 but eventually our code catches up and exceeds it. For all of the other problems our code is competitive or better than CPLEX when CPLEX stops.

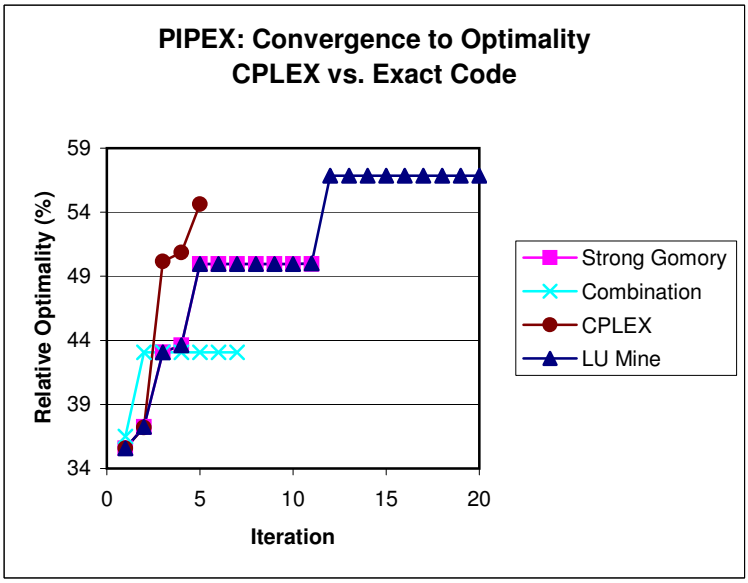

Figure 8.18: Results of CPLEX vs. Exact Code for PIPEX

8.7 LSEU

The ROP graph of LSEU is a classical example of what we usually would think that happens with cutting planes. This graph demonstrates the trailing off effect of cutting planes that is typical of cutting plane algorithms, see Figure 8.19 for details. All of the codes are within 1% ROP of each other up to iteration 5 where CPLEX bows out.

We have also plotted the weak cutting planes in Figure 8.20. It is exactly what we expect to happen with weaker cuts. The solution tends to die off.

We have included graphs of the average number of digits to store the numerator and denominator of the Gomory cutting planes. We used a cutoff of 50 digits and

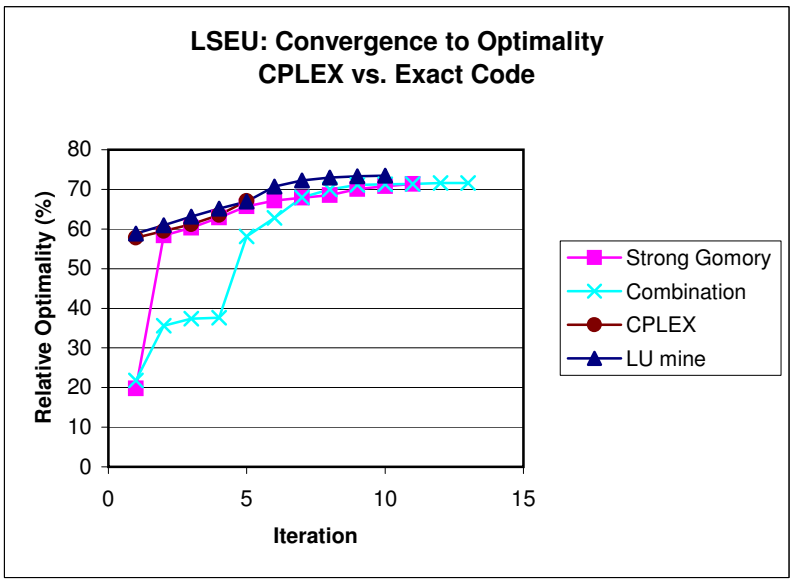

Figure 8.19: Results of CPLEX vs. Exact Code for LSEU

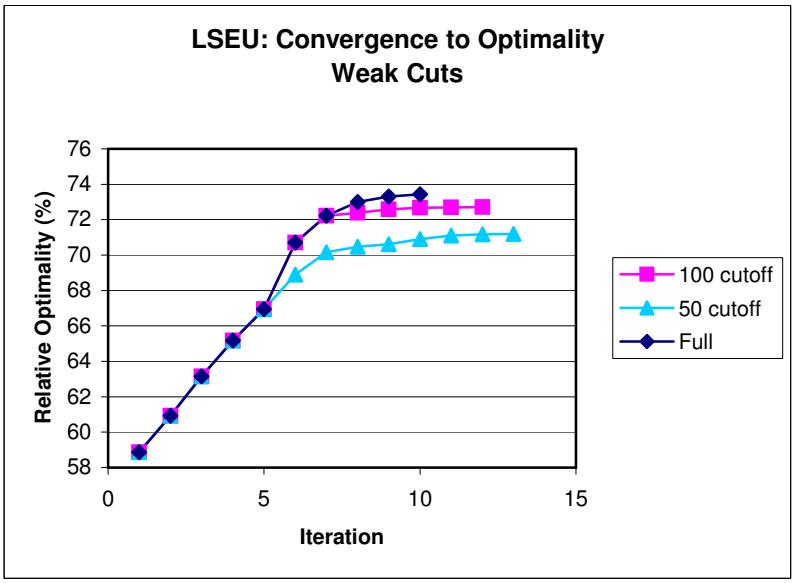

Figure 8.20: Results of Weak Cuts for LSEU

100 digits. Figures 8.21 and 8.22 are these graphs. Notice that in Figure 8.21 that the cutoff is 50 and the average is above 50. This could happen in two ways. First, if all of the Gomory cutting planes have 50 digits in it then the average will actually be 51 because there are 50 digits for the numerator and 1 digit for the denominator. Second, recall that the program used to find the size of a variable can be one larger than the actual size. This is surprising that this has occurred but it is possible.

Figure 8.23 displays the full version of the size of the Gomory cutting planes. Note the exponential growth of this graph where the 50 and 100 cutoff are much smaller. However, the weaker cuts just do not work as well as the full blown version. Not only do we have exponential growth in the number of cutting planes but also in the size of the cutting planes.

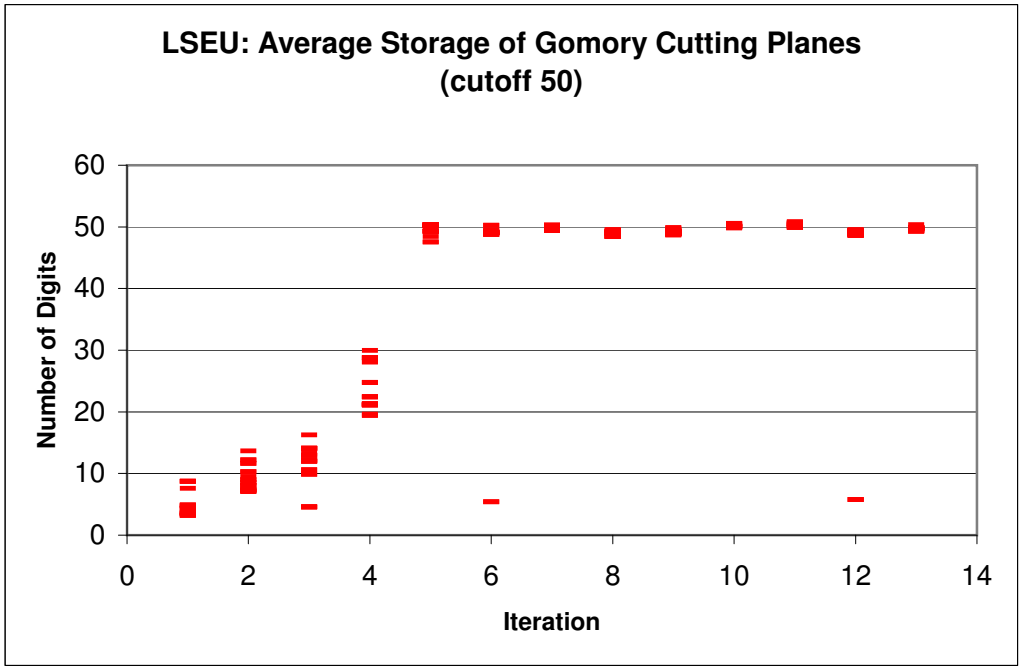

Figure 8.21: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem LSEU

Figures 8.24 and 8.25 show what happens to the size of the w-vector for LSEU for the 50 digit cutoff and 100 digit cutoff of the Gomory cutting planes. Notice the range of the 50 and the 100. The range is much large for the 100 digit cutoff. Interestingly enough, each of the graphs tend to level off after the reach their respective cutoff points. This is what we would hope would happen for all of the problems.

Now turn your attention to Figure 8.26. First note that we have switched to a logarithmic scale for the range. This is because the size of these values grow so large. This displays perfectly the exponential nature of exact arithmetic. The code slows down almost to a crawl. Mainly because all of the eta matrices has these huge fractions in them and it needs to solve systems of equations based on these eta matrices. With the size of the Gomory cutting planes, refactorizing the basis is

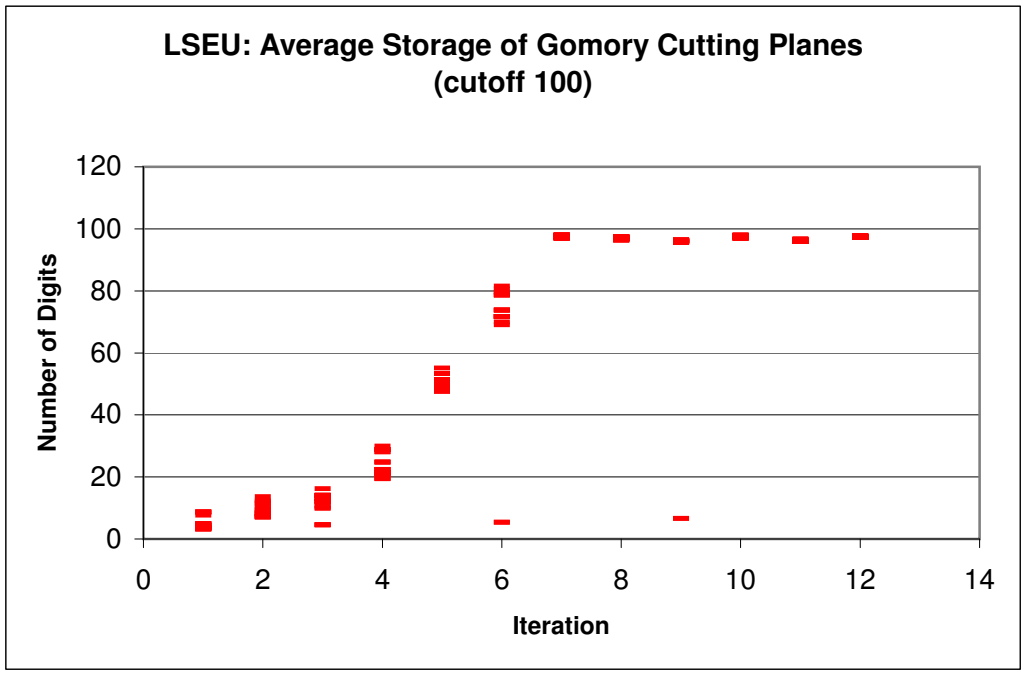

Figure 8.22: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem LSEU

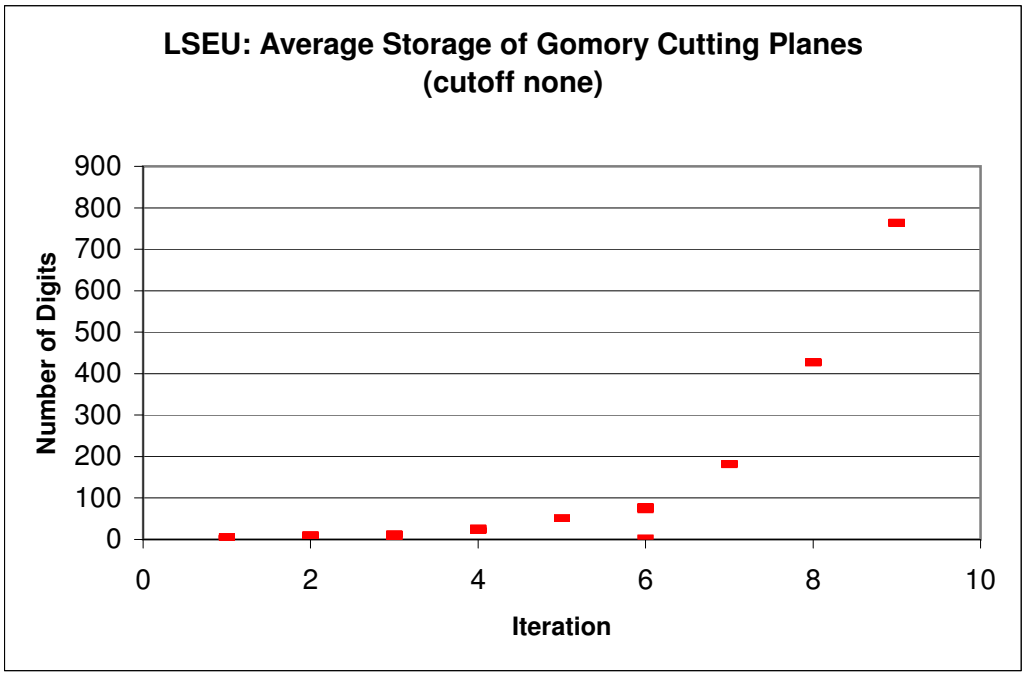

Figure 8.23: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem LSEU

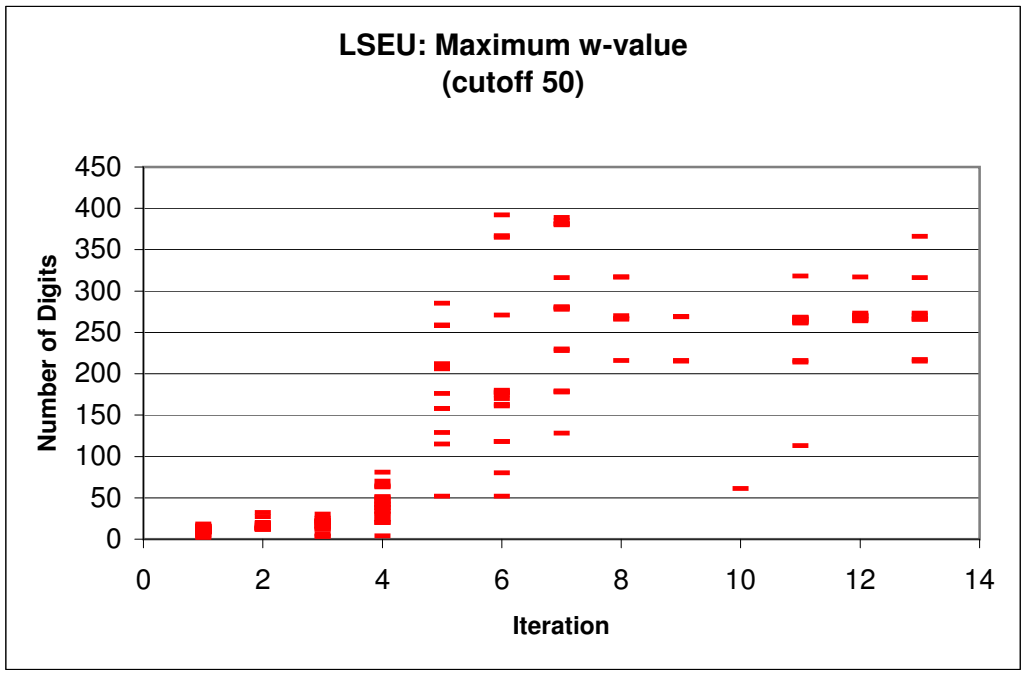

Figure 8.24: Maximum size of the element of w-vector for problem LSEU

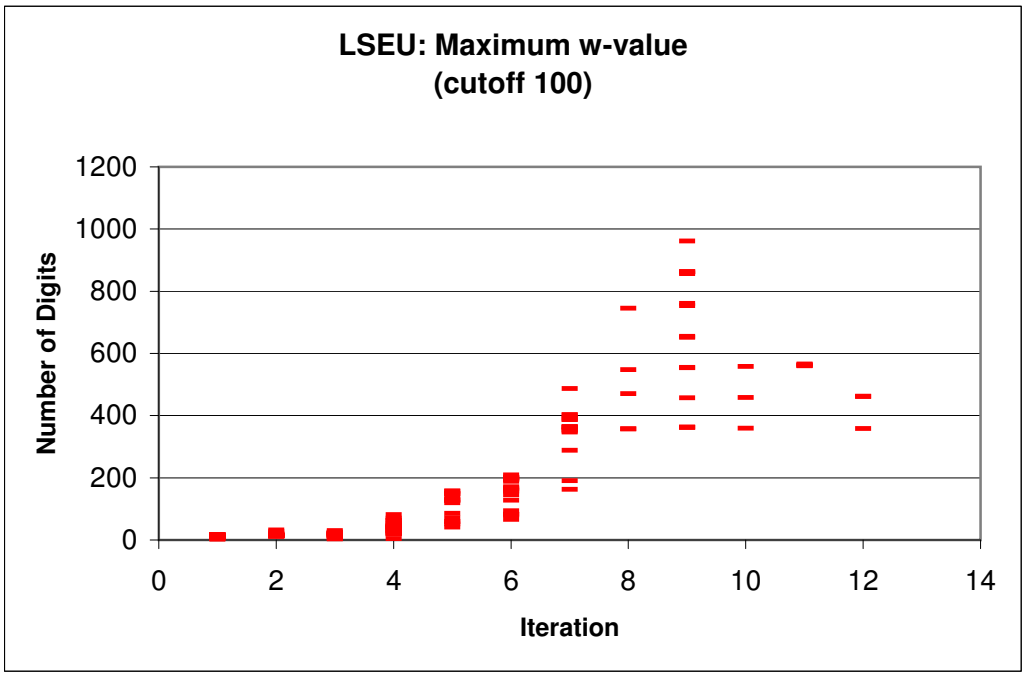

Figure 8.25: Maximum size of the element of w-vector for problem LSEU

probably not going to help either. This is probably the worst case scenario of what we want to avoid.

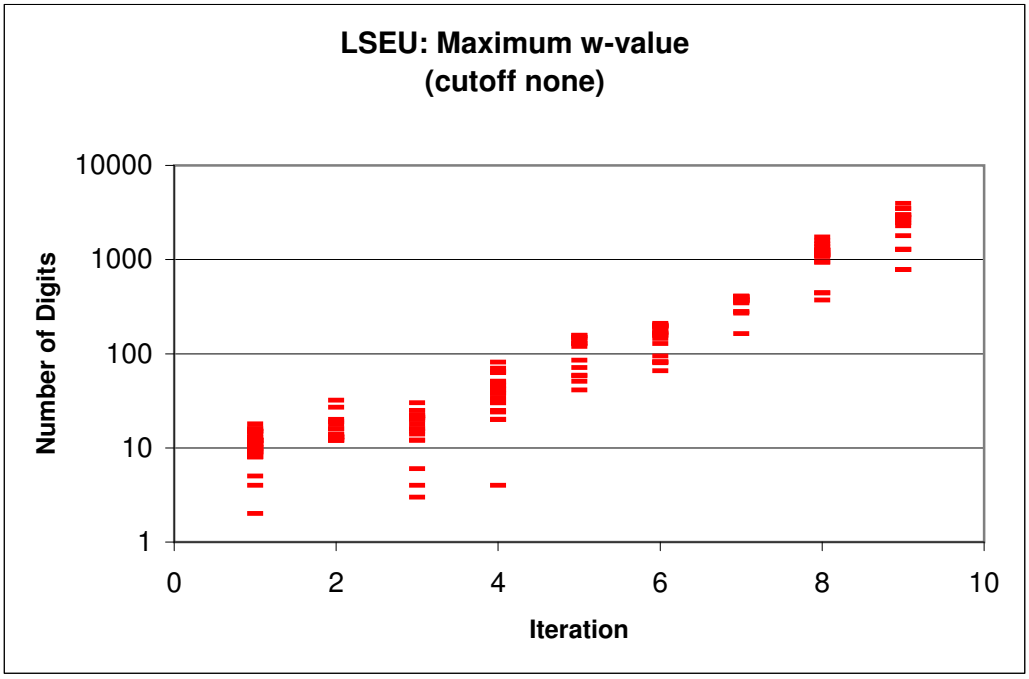

Figure 8.26: Maximum size of the element of w-vector for problem LSEU

8.8 P0201

Both versions of the old code could not get very far with this problem see Figure 8.27. However compared to CPLEX our three codes do extremely well. All of the codes, especially LU code, does much better than CPLEX. CPLEX does not even reach 15% of optimality where the LU code nearly reaches 45%.

Surprisingly, the size of the Gomory cutting planes stay very small see Figure 8.28. Also according to Figure 8.29, the numbers do not get large even in the solution of $Bw = A_{jp}$. However, CPLEX still does not do well even with these facts. CPLEX did get off to a very good start for this problem and then it tails off. The issue with this problem is not the size of the fractions, it is the number of Gomory cutting planes that get added to the tableau. In Figure 8.28, it looks like there is only one value of 2 for the Gomory iteration 1. However, there is actually 42 cutting planes all with that same average size. For this problem, it is probably useful to know the

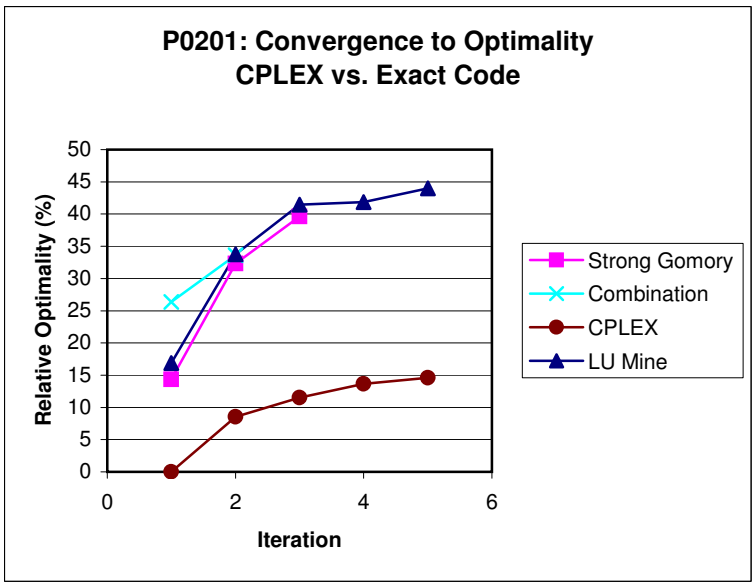

Figure 8.27: Results of CPLEX vs. Exact Code for P0201

number of Gomory cutting planes for each Gomory iteration (refer to Table 8.5). Notice that the number of dual simplex iterations is the same order or less than the number of Gomory cutting planes added. This is a common belief.

Iteration	Number of Gomory Planes	Dual Simplex Iterations
1	42	13
2	55	78
3	106	60
4	110	33
5	126	64

Table 8.5: Number of Gomory Cutting Planes for Problem P0201

8.9 P0282

This problem is similar to P0201 in that the Gomory cutting plane storage and the maximum w-value is small as in Figure 8.31 and Figure 8.32. We see the typical exponential growth in both of these graphs. None of the codes do extremely well as seen in Figure 8.30. Our three codes still outperform CPLEX but it is hardly anything to brag at.

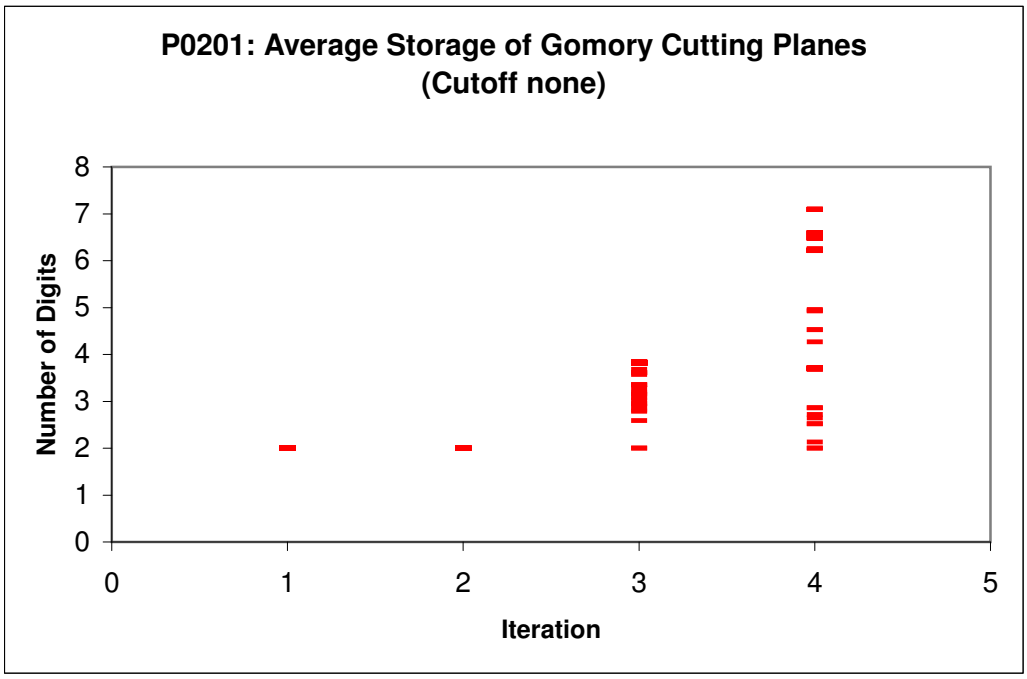

Figure 8.28: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem P0201

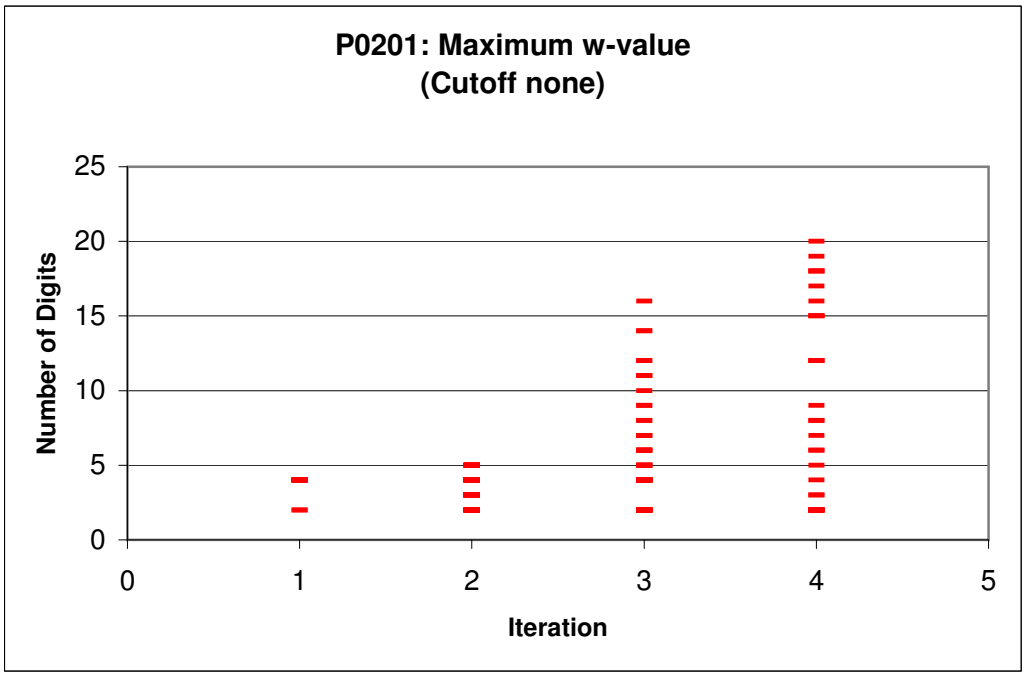

Figure 8.29: Maximum size of the element of w-vector for problem P0201

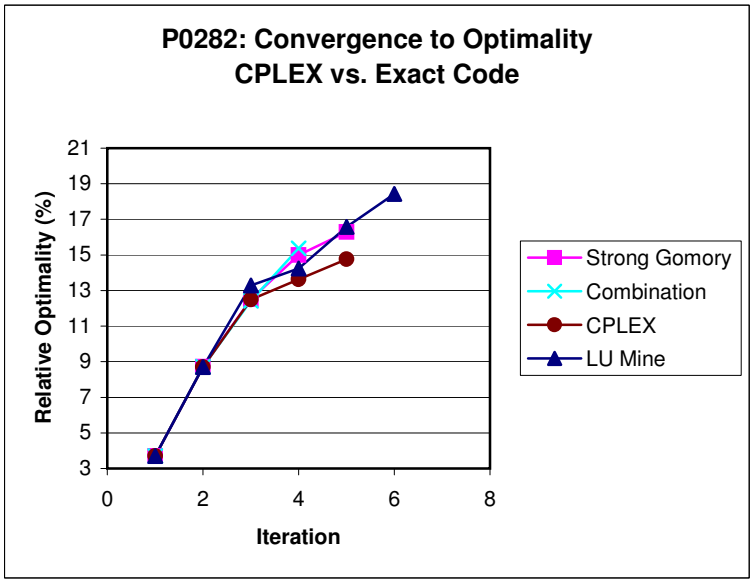

Figure 8.30: Results of CPLEX vs. Exact Code for P0282

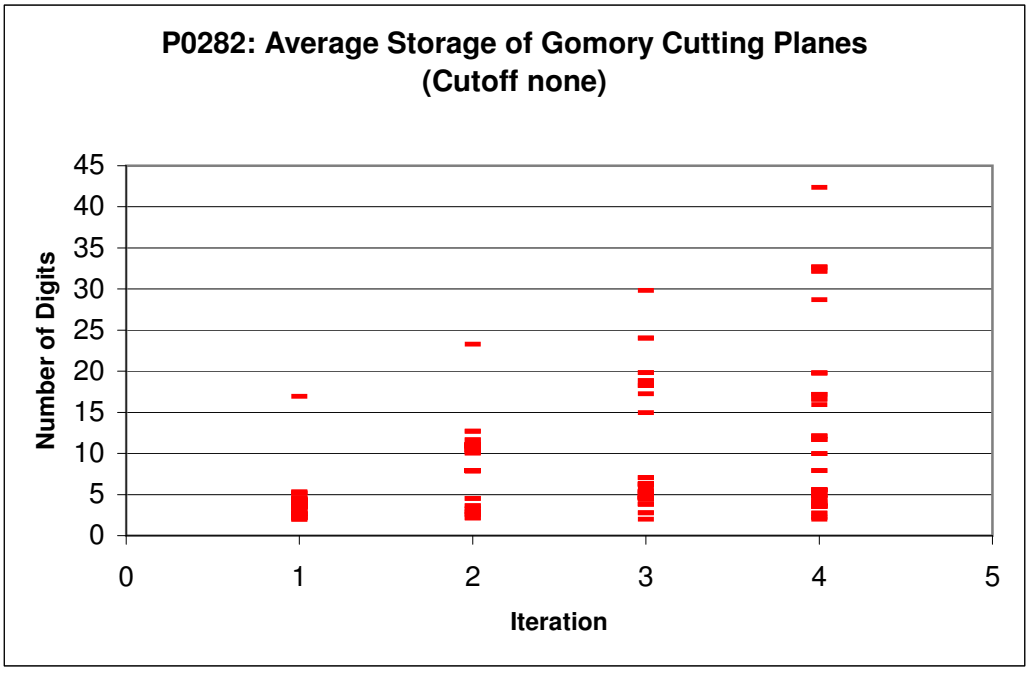

Figure 8.31: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem P0282

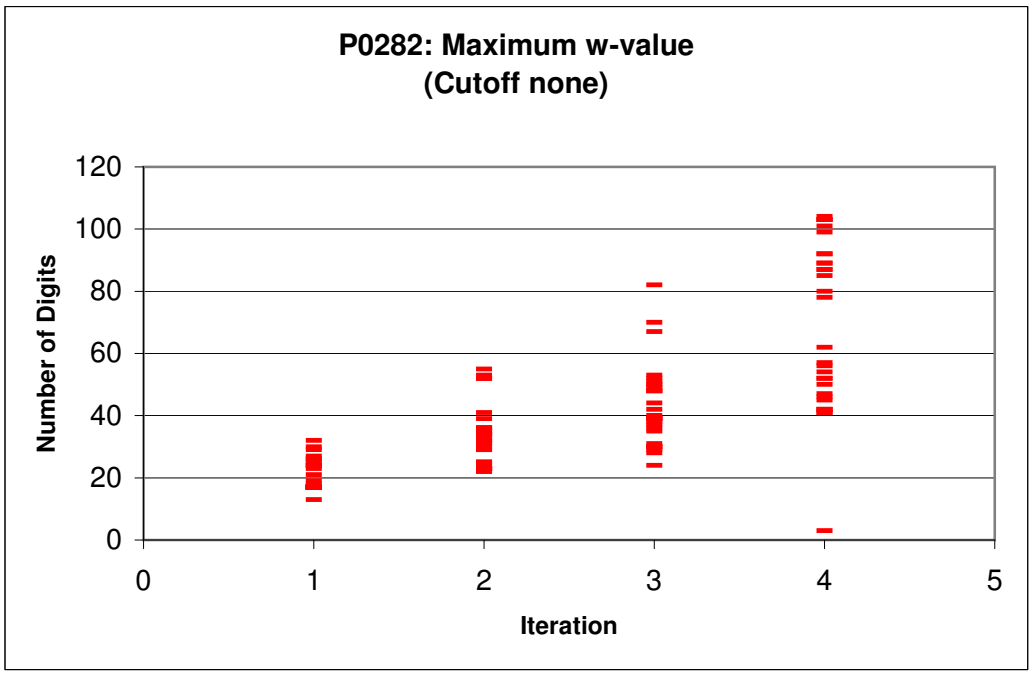

Figure 8.32: Maximum size of the element of w-vector for problem P0282

8.10 P0291

This problem is similar to LSEU in that it has the long tail off. All of the codes are very similar with ours slightly edging out CPLEX see Figure 8.33.

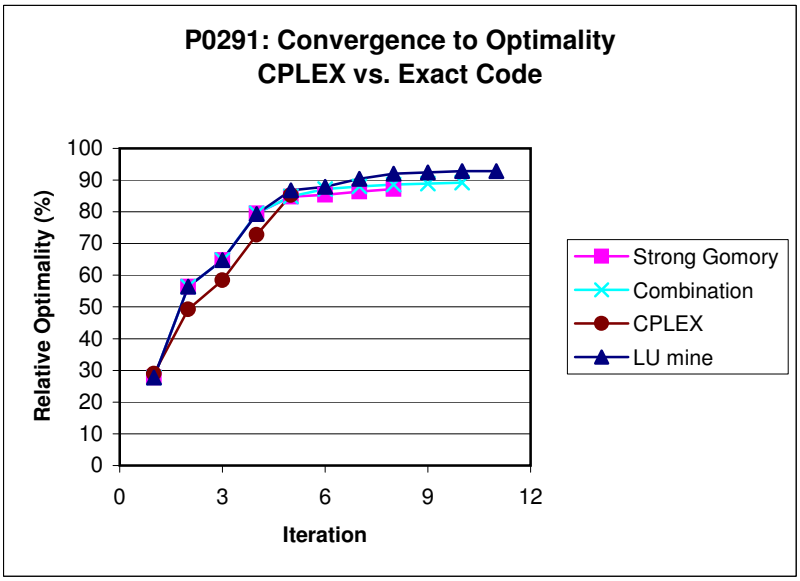

Figure 8.33: Results of CPLEX vs. Exact Code for P0291

We have include the later iterations of the weaker cuts in Figure 8.34. Notice that we do not have our typical situation. We have the cutoff of 50 doing better than the cutoff of 100. They are exactly the same up until iteration 6 when the cutoff 50 drops below cutoff of 100 but then it recovers and passes it. However, it still does not perform as well as the full blown version. We still need all of these digits of accuracy.

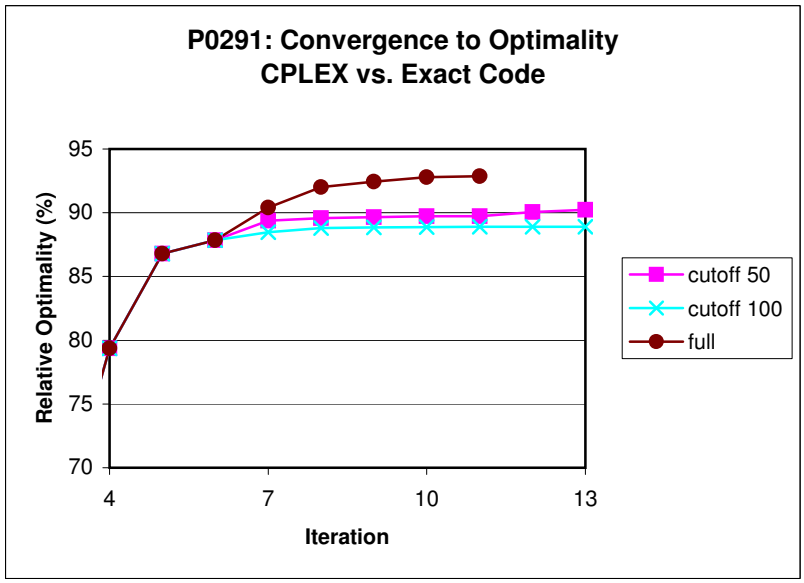

Figure 8.34: Results of Weak Cuts for P0291

Interestingly, we see a decrease in the maximum w-value in the cutoff 50 example (see Figure 8.38). Also, we see a levelling off in the cutoff 100 version (see Figure 8.39).

8.11 SENTOY

This problem is similar to P0282. All of the codes perform similarly as seen in Figure 8.41. None of them really shine. This is a hard problem because in its original format it is totally dense. It has 30 rows and 60 columns and 1800 non-zero elements. This is fairly rare for a problem of this size. With a totally dense system, the factorization and the sparse matrix storage does not help.

Again we have a levelling off of the graphs of the maximum w-vector in Figures 8.46 and 8.47, where we have exponential growth for the full version.

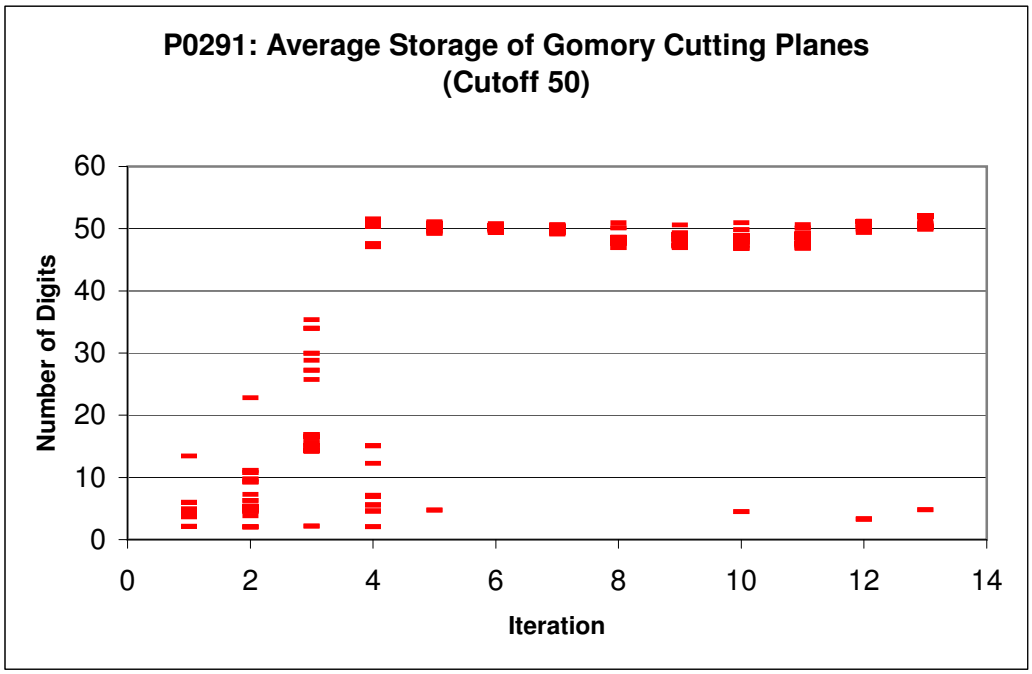

Figure 8.35: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem P0291

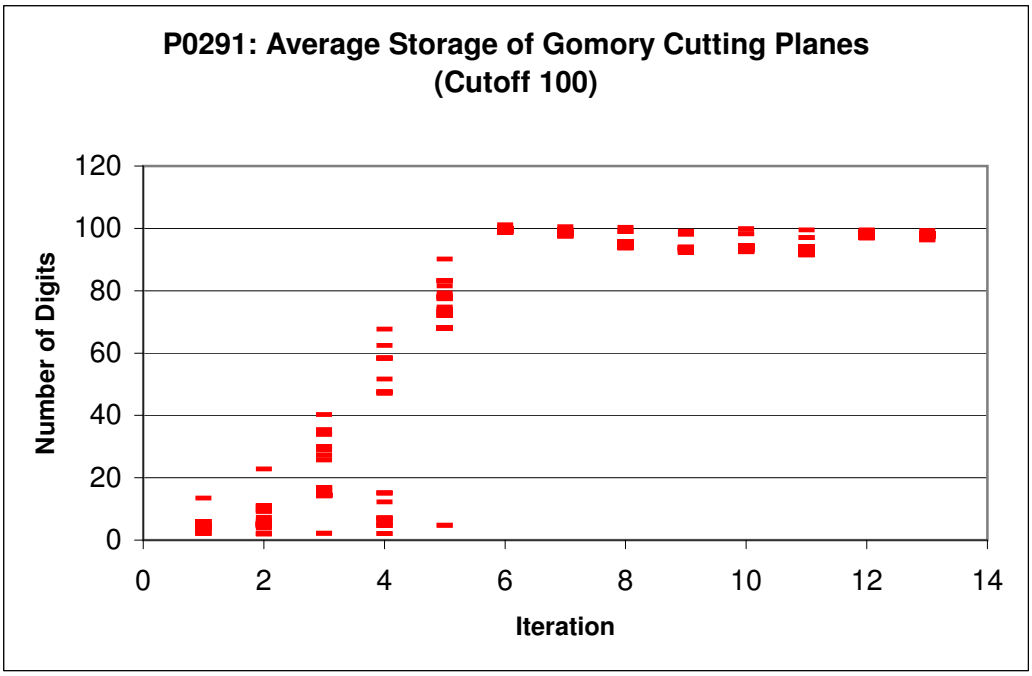

Figure 8.36: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem P0291

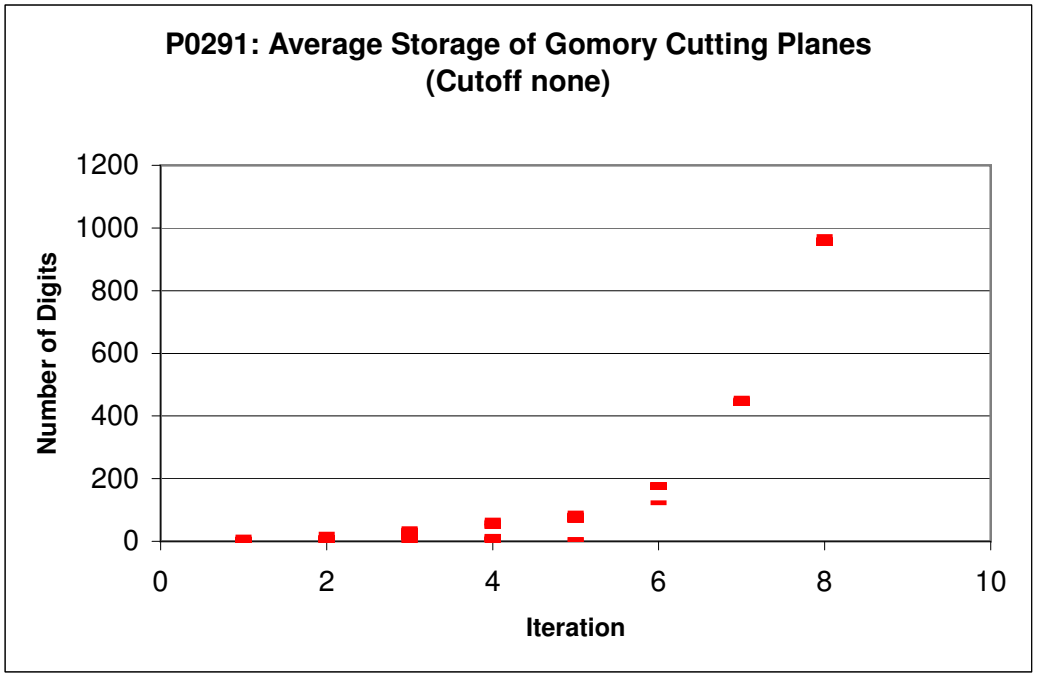

Figure 8.37: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem P0291

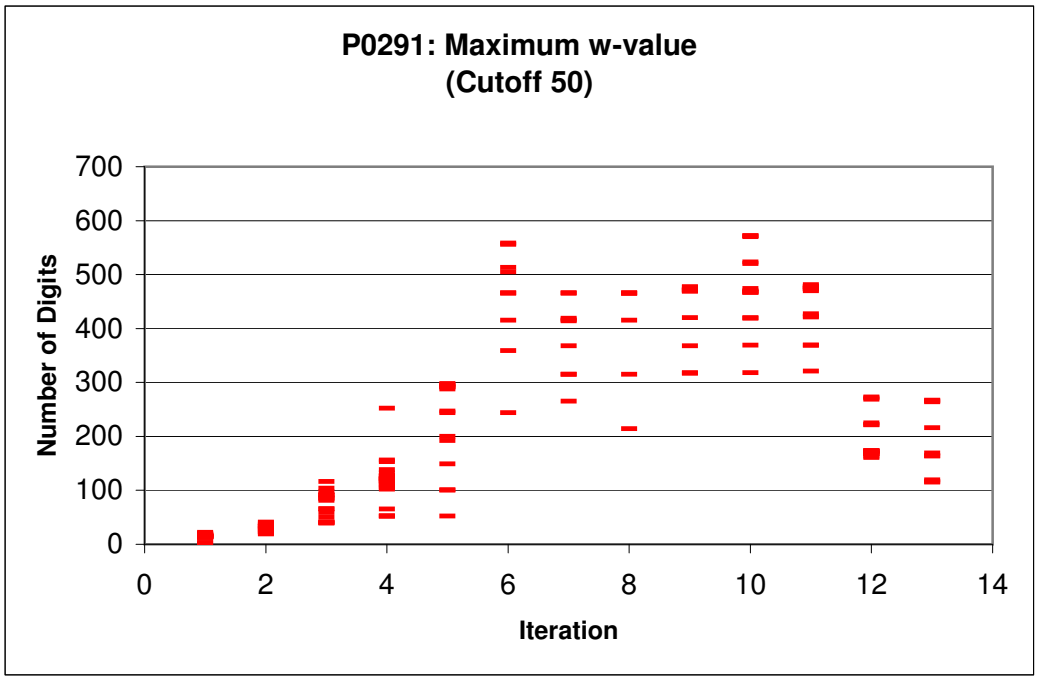

Figure 8.38: Maximum size of the element of w-vector for problem P0291

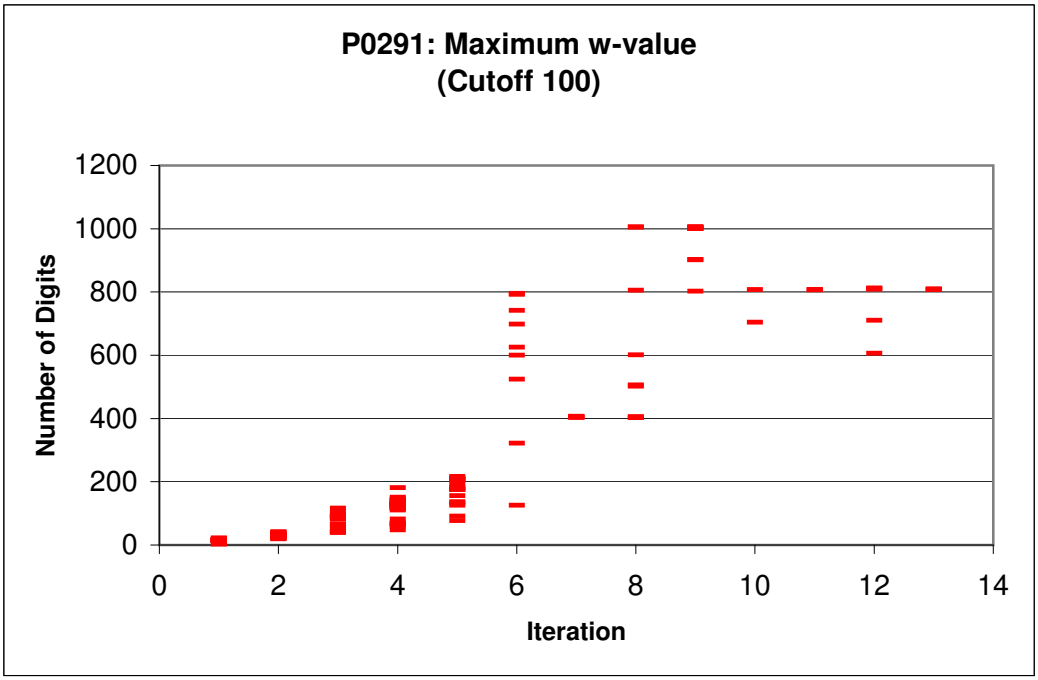

Figure 8.39: Maximum size of the element of w-vector for problem P0291

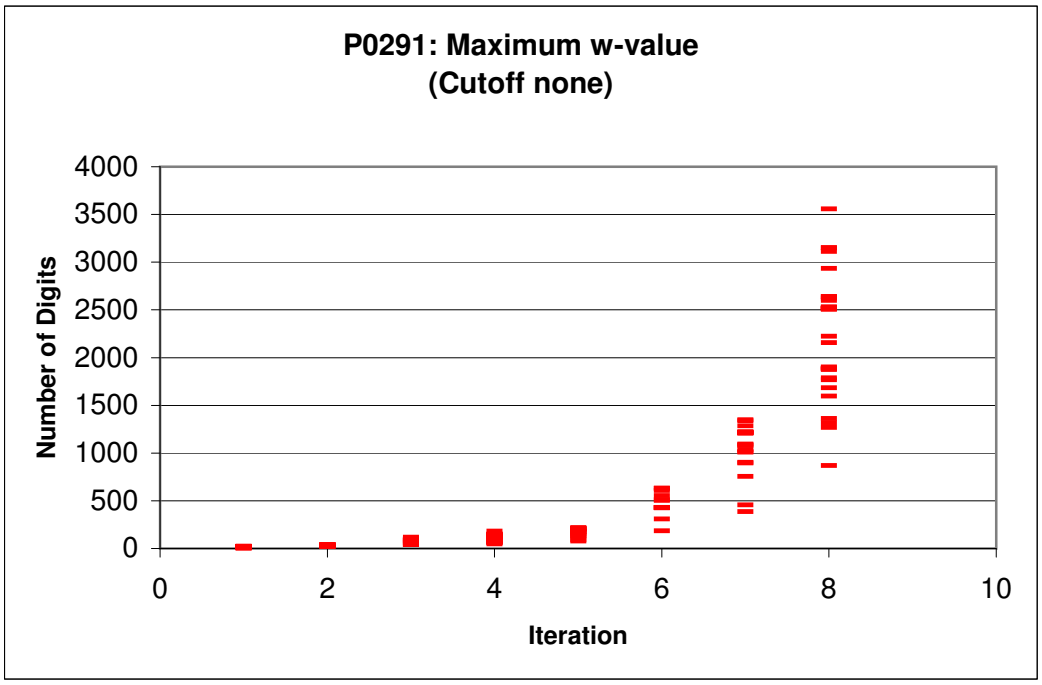

Figure 8.40: Maximum size of the element of w-vector for problem P0291

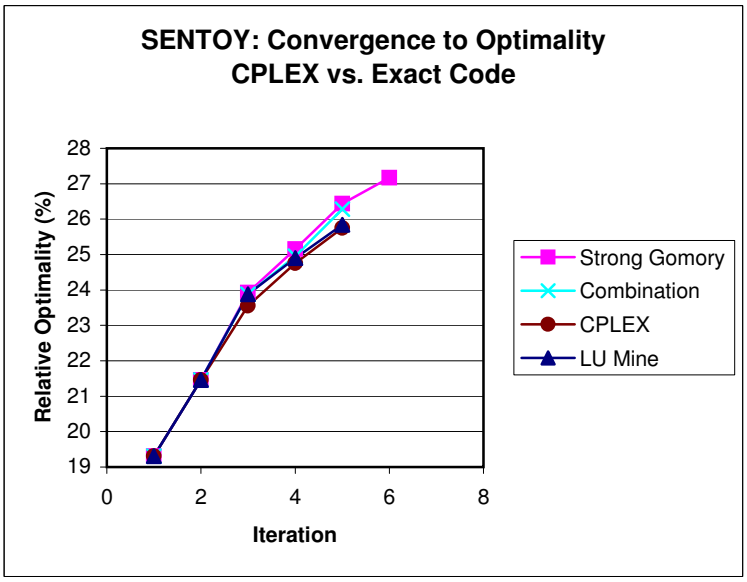

Figure 8.41: Results of CPLEX vs. Exact Code for SENTOY

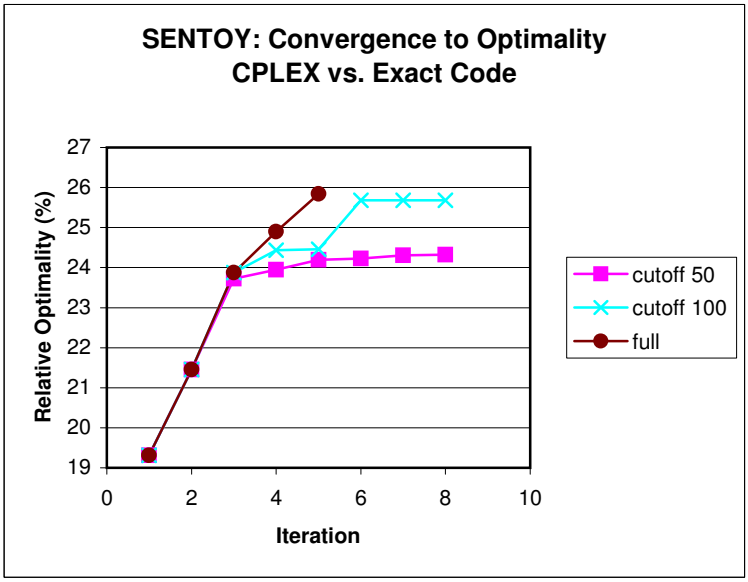

Figure 8.42: Results of Weak Cuts for SENTOY

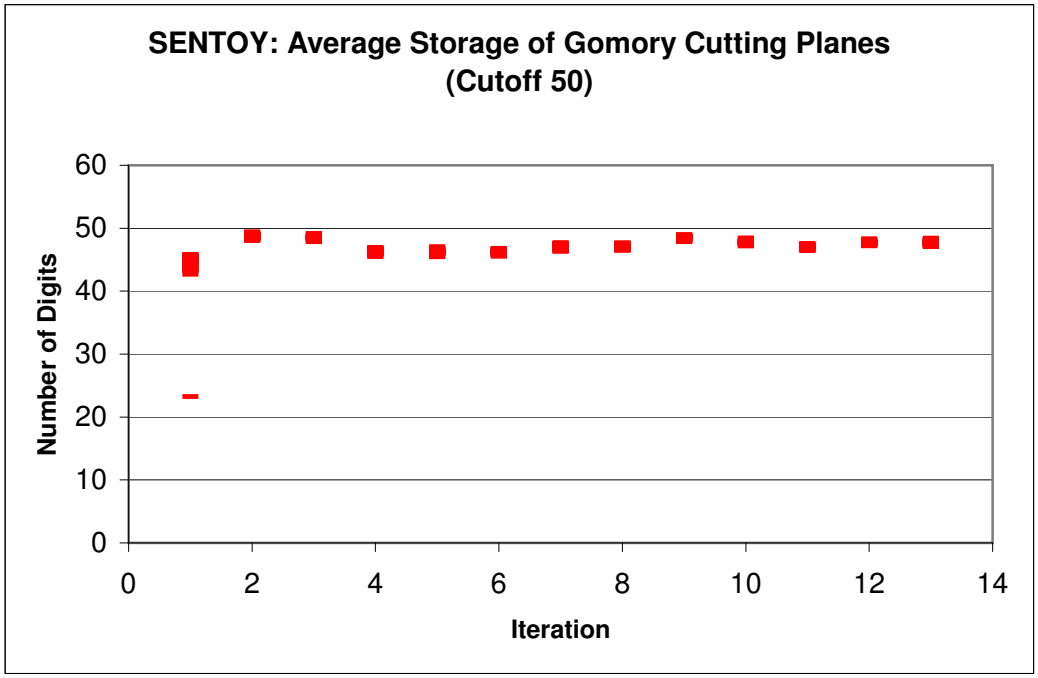

Figure 8.43: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem SENTOY

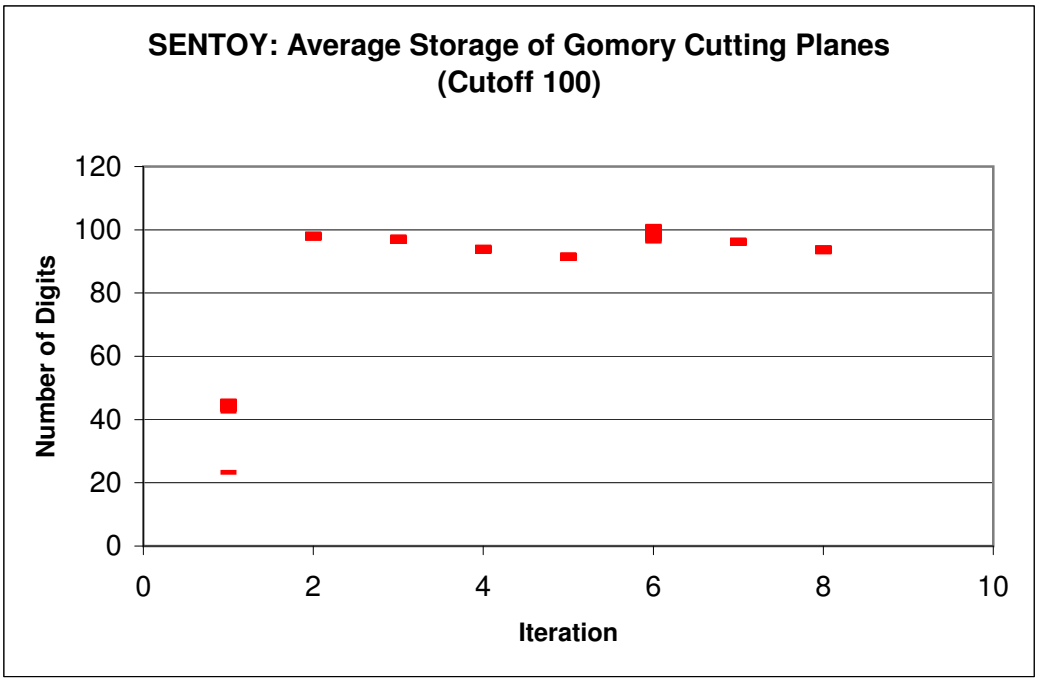

Figure 8.44: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem SENTOY

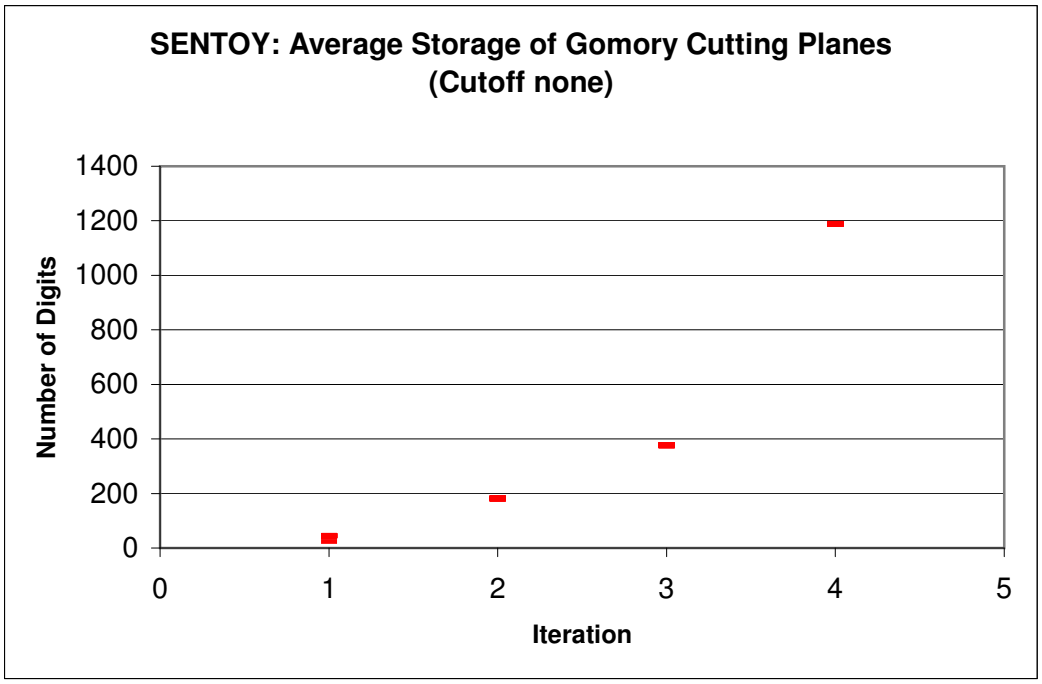

Figure 8.45: Average Digits for Storage of Non-Zeros elements of Gomory cutting planes for problem SENTOY

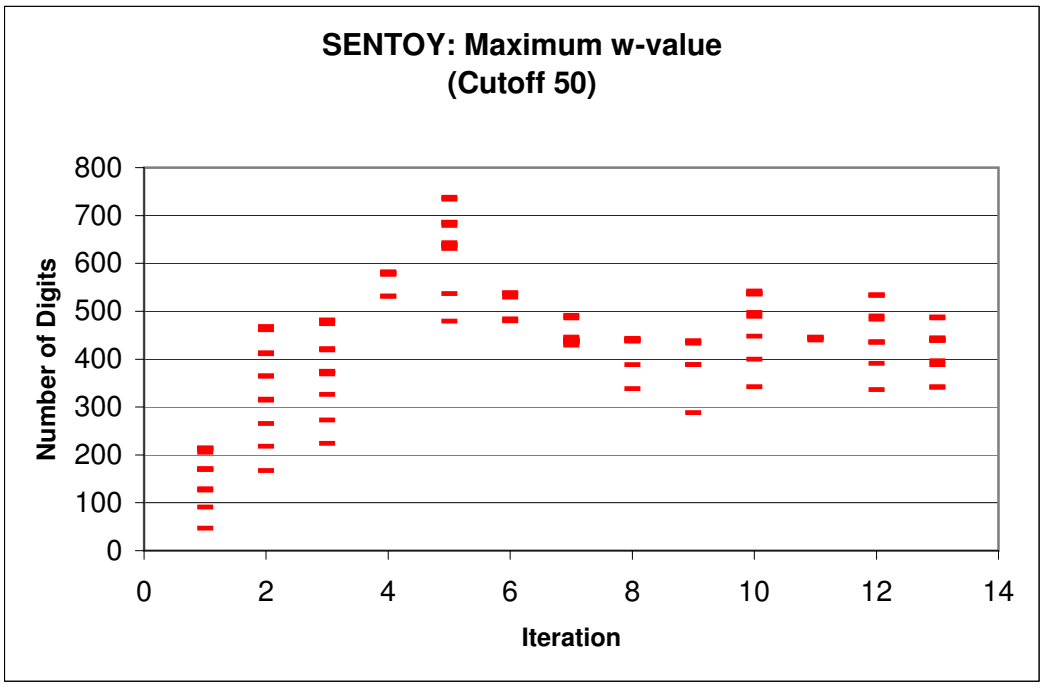

Figure 8.46: Maximum size of the element of w-vector for problem SENTOY

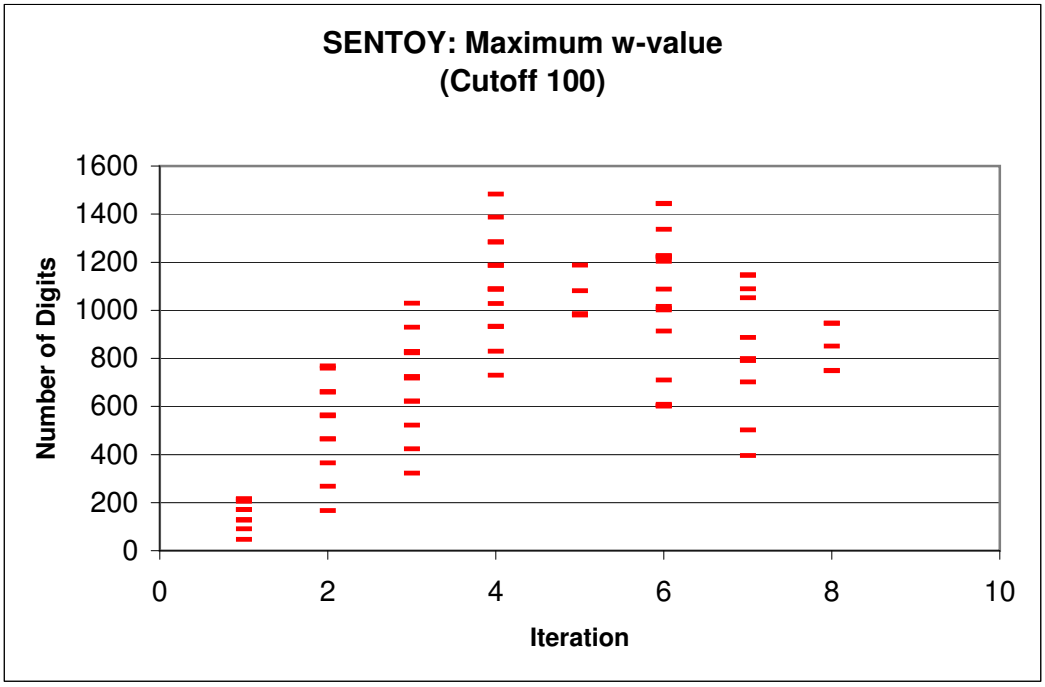

Figure 8.47: Maximum size of the element of w-vector for problem SENTOY

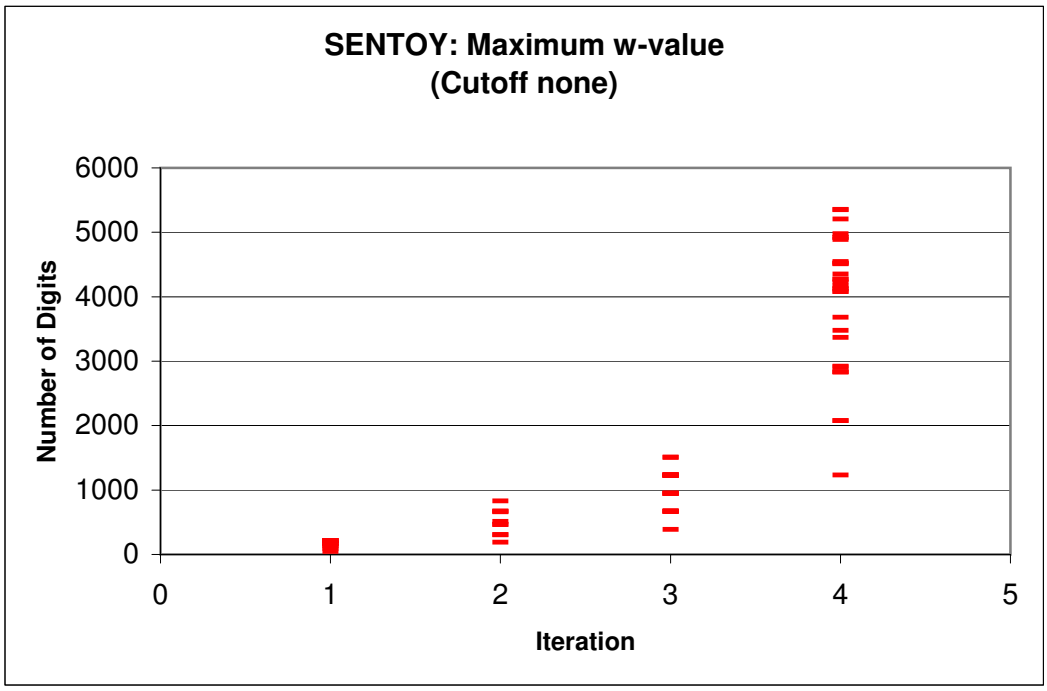

Figure 8.48: Maximum size of the element of w-vector for problem SENTOY

CHAPTER 9
CPLEX and Exact Arithmetic

We have also tried to implement code using the CPLEX callable library. Since the longest time is spent solving the LP relaxations in our exact code we use CPLEX to solve the LP relaxations and exact code to do the rest. So we have CPLEX solve the current LP and then we take the basis given by CPLEX and feed that into the exact Gomory cutting plane code that we wrote. Below is an outline of the algorithm:

Algorithm 1 *CPLEX/Exact Code Algorithm:*

1. *Use CPLEX to solve the LP relaxation.*
2. *Find the basis and form the LU factorization in exact arithmetic.*
3. *Find the Gomory cutting planes.*
4. *Add these cutting planes to the CPLEX tableau.*
5. *If optimal, stop.*
6. *Else go to 1.*

There are some considerations that must be taken in account. First, CPLEX and our code must have the same starting tableau. We must change the problem formulation in CPLEX. So, we add in all of the bound information directly to the tableau. We also need to change all of the inequalities to equalities. This way we can have access to all of the slack variables. We need to shut off the presolver and change the problem from an integer programming problem to a linear programming problem. Now we have the same tableau.

We put this LP formulation into CPLEX and we get a solution. We use the function *cpx_getbase* to get the basis information. CPLEX uses two vectors (*cstat* and $rstat$) to store all of the basis information. *cstat* is the column information so the j^{th} element of this vector is either basic, nonbasic at its lower bound, nonbasic at its upper bound, or free. $rstat$ is the row information so the i^{th} element of this vector is whether the slack/artificial/surplus variable associated with that row is basic, nonbasic at its lower bound, or nonbasic at its upper bound. One issue that

CPLEX has is that if there is a primal degenerate solution then it does not supply enough variables to fill the basis. So we must figure out the other variables that are basic. Since we have changed all of the constraints to equalities and explicitly put the bounds of all of the variables in, one would think that *rstat* would have all of these variables nonbasic at its lower bound since there are no slack/artificial/surplus variables. However, this is not the case. The exact number of variables that are missing from the basis can be found in *rstat*. So we took this row and put the "slack" variable associated with that row into the basis. This worked for every small example that we tried.

Once we have the solution and the basis, we now use our Gomory cutting plane code to find the cutting planes and add them to the LP. This caused a problem. CPLEX can only handle tableaux that are extremely well behaved. We have seen how quickly the numbers grow in the Gomory cutting plane so we must weaken the cuts so that we can add them to the tableau. The problem is that we had to weaken them to less than 8 digits. Therefore most of the cuts are so weak that they do not help at all. This part of the project was halted here because of this set back.

CHAPTER 10
Conclusion

Gomory cutting planes are incorporated into a lot of commercial solvers but they are not used to their full power. Using exact arithmetic is slower but it can solve more problems and usually yields lower iterations counts than floating point arithmetic. Exact arithmetic also allows for the possible exploitation of the alternate optima and generating cutting planes from all of these optima instead of just one at a time. We have found this to work extremely well for two dimensional cases. We have found that by using the alternate form of this alternate optima approach we can get away from problems that get stuck on the same objective value for long periods of time. Searching all of the other alternate optima might be overkill. The other types of cutting planes that CPLEX uses might be able to be expanded in the same way as Gomory cutting planes since they are just inequalities computed in a different manner. Exact arithmetic might not be as effective on those because in general they are usually already integral coefficients like the knapsack inequalities.

We have implemented many of the advancements that modern LP solvers have but in exact arithmetic found in [3]. For example, sparse matrix storage, using steepest edge pivot rules instead of the most negative reduced cost for solving the primal and the dual simplex, and using LU factorization with partial pivoting. Even with these advancements we are still only able to solve the smaller of the MIPLIB problems. Gomory cutting planes cause the size of the problems to become exponential. However, we have shown that using exact arithmetic has resulted in coming up with better cutting planes even with small problems. Hence we would not need to start branching algorithms so soon.

BIBLIOGRAPHY

[1] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Mathematical Programming*, 71:221–245, 1996.

[2] N. S. Asaithambi. *Numerical Analysis: Theory and Practice*. Saunders College Publishing, Fort Worth, 1995.

[3] R. E. Bixby. Solving real-world linear programs: a decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[4] R. E. Bixby and A. Martin. Parallelizing the dual simplex method. *INFORMS Journal on Computing*, 12(1):45–56, 2000.

[5] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2:103–107, 1977.

[6] V. Chvátal. *Linear Programming*. Freeman, New York, 1983.

[7] S. A. Cook. The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing Machinery*, pages 151–158, 1971.

[8] K. Devlin. *The Millennium Problems*. Basic Books, 2002.

[9] J. G. Ecker and M. Kupferschmid. *Introduction to Operations Research*. John Wiley, New York, 1986.

[10] J. G. Ecker and J. H. Song. Optimizing a linear function over an efficient set. *Journal of Optimization Theory and Applications*, 83:541–563, 1994.

[11] D. Goldfarb and J.K. Reid. A practicable steepest-edge simplex algorithm. *Mathematical Programming*, 12:361–371, 1977.

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

[13] R. E. Gomory. An algorithm for the mixed-integer problem. *Rand Corporation, Report RM-2597*, 1960. Never Published.

[14] T. Granlund. The GNU multiple precision arithmetic library, edition 4.1.2. Technical report, Swok, AB, Stockholm, Sweden, 2002.

[15] R. Johnson. *Linear Algebra*. Prindle, Weber, & Schmidt, 1967.

[16] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorthims*, volume 1. Addison Wesley, 2nd edition, 1973.

[17] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, 2nd edition, 1981.

[18] A. N. Letchford and A. Lodi. Strengthening Chvátal-Gomory cuts and Gomory fractional cuts. *Operations Research Letters*, 30(2):74–82, 2002.

[19] J. E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. In P. M. Pardalos and M. G. C. Resende, editors, *Handbook of Applied Optimization*, pages 65–77. Oxford University Press, January 2002.

[20] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley, New York, 1988.

[21] L. A. Wolsey. *Integer Programming*. John Wiley, New York, 1998.