IMPROVED BRANCH AND BOUND ALGORITHMS FOR INTEGER PROGRAMMING

By

Brian Borchers

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: Mathematical Sciences

Approved by the
Examining Committee:

_____
Professor John E. Mitchell, Thesis Adviser

_____
Professer Joseph G. Ecker, Member

_____
Professor George Habetler, Member

_____
Professor Carlton E. Lemke, Member

_____
Professor John Mittenthal, Member

Rensselaer Polytechnic Institute
Troy, New York

June 1992
(For Graduation August 1992)

IMPROVED BRANCH AND BOUND ALGORITHMS FOR INTEGER PROGRAMMING

By

Brian Borchers

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: Mathematical Sciences

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

Approved by the
Examining Committee:

_____
Professor John E. Mitchell, Thesis Adviser

_____
Professer Joseph G. Ecker, Member

_____
Professor George Habetler, Member

_____
Professor Carlton E. Lemke, Member

_____
Professor John Mittenthal, Member

Rensselaer Polytechnic Institute
Troy, New York

June 1992
(For Graduation August 1992)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to take this opportunity to thank my advisor, John Mitchell, for introducing me to the whole field of interior point methods and supervising the work that went into this thesis. I would also like to thank the members of the thesis committee, Joe Ecker, George Habetler, Carlton E. Lemke, and John Mittenthal. My fiancée Suzanne Bloodgood provided moral support and proofread the text of the thesis for spelling errors. Our cat Dustmop kept me company during long nights spent at the terminal.

This research was partially supported by ONR Grant number N00014-90-J-1714.

ABSTRACT

This thesis is an investigation into improved branch and bound algorithms for linear and nonlinear mixed integer programming. Heuristics are developed for determining when the solution to a subproblem will involve fractional variables. When this situation is detected, the algorithm branches early, creating new subproblems with the fractional variables fixed at integer values. In this way, we avoid the work of solving the current subproblem to optimality.

Branch and bound codes for mixed integer linear programming problems have traditionally used the simplex method to solve the linear programming subproblems that arise. Early detection of fractional variables is difficult with the simplex method, because a variable might become basic (and thus fractional) at the last iteration of the simplex method. In contrast, with an interior point method, variables converge steadily to their optimal values. This makes it relatively easy to detect fractional variables.

To be useful in a branch and bound algorithm, an interior point method must generate lower bounds, which are used to fathom subproblems, and primal solution estimates, which the heuristics use to detect a fractional solution. Most interior point methods satisfy these requirements, but we concentrate on the dual-affine method and the primal-dual path following method. We describe experimental codes based on both methods. Computational results are provided for a number of randomly generated problems as well as for problems taken from the literature. These results indicate that the early branching idea is very effective, but more work is needed to make the experimental branch and bound codes competitive with simplex based branch and bound codes.

The early branching idea is also used in a branch and bound code for mixed integer nonlinear programs. This code uses the sequential quadratic programming

method to solve the NLP subproblems. The SQP method generates estimates of the optimal solution at each iteration, which the early branching heuristics use. Computational results are provided for a number of sample problems taken from the literature. On some problems, the code with early branching uses 20% less CPU time than the code without early branching.

CHAPTER 1
Introduction and Historical Review

Mixed integer programming problems are optimization problems in which some of the decision variables are restricted to integer values while other decision variables may be continuous. The integer variables typically encode choices from a small set of options available to the decision maker. In many cases, these choices are binary "yes or no" decisions. Such decisions can be encoded by integer variables which are restricted to the values zero and one. The objective function and constraints can be linear or nonlinear, and nonlinear objective functions and constraints may be convex or nonconvex. We will restrict our attention to zero–one mixed integer linear programs and zero–one mixed integer nonlinear programs with convex objective functions and constraints.

Many optimization problems can be formulated as mixed integer linear programs with zero–one variables, including facility location problems, scheduling problems, set covering and partitioning problems, matching problems, and linear ordering problems. Salkin and Mathur provide many examples of zero–one integer linear programming models in their book [74]. Some applications of zero–one mixed integer nonlinear programs with convex constraints and objective functions include network design [43, 64], chemical process synthesis [25, 38, 52], product marketing [29], and capital budgeting [57, 65, 86].

There are a number of approaches to solving mixed integer linear programming problems, including branch and bound algorithms, implicit enumeration algorithms, cutting plane algorithms, Bender's decomposition, and group theoretic approaches. Of these methods, the most popular general purpose method is clearly branch and bound. Many commercially available codes, including MPSX/MIP, OSL, Sciconic, and Lindo implement branch and bound algorithms. The other approaches have

been somewhat successful in the research environment, but are much less widely used in practice. Much less work has been done on algorithms for mixed integer nonlinear programs. Approaches to solving these problems include branch and bound algorithms, generalized Bender's decomposition, and algorithms based on outer approximation.

1.1 The Branch and Bound Method

We will concentrate on branch and bound approaches to mixed integer linear and nonlinear programs. We restrict our attention to branch and bound algorithms for zero-one mixed integer programs, but the branch and bound method can also be applied to problems with integer variables that are not restricted to the values zero and one. The method was first described by Land and Doig in 1960 [55]. Dakin, Beale and Small, Driebeck, Tomlin, and others improved the basic algorithm in a number of ways [5, 15, 18, 26, 80]. There are many survey articles and textbooks that cover branch and bound algorithms for mixed integer linear programs, including [30, 56, 58, 72, 74]. Papers that discuss branch and bound algorithms for mixed integer nonlinear programs include [40–42].

The branch and bound algorithm for zero-one mixed integer programming works by enumerating possible combinations of the zero–one variables in a branch and bound tree. Each node of the tree is a continuous optimization problem, based on the original problem, but with some of the integer variables fixed at zero or one. The remaining zero–one variables are allowed to take on any value in the range between zero and one. The root of the tree is the original problem with all of the integrality constraints relaxed.

At each iteration, the algorithm selects a leaf node from the tree and solves the corresponding subproblem. There are four possible results. If the subproblem is infeasible, then any further restriction of the subproblem would also be infeasible,

so we can ignore all descendents of the current subproblem. If the optimal value of the current subproblem is larger than the objective value of a known integer solution, then no restriction of the current subproblem will give a better integer solution, and we can ignore all descendents of the current subproblem. This process of deleting the current subproblem and all of its potential children from the tree is called fathoming. If the optimal solution to the current subproblem satisfies all of the integrality constraints and has a lower objective value than any known integer solution, then we record the objective value of this integer solution. Finally, if none of the above conditions are satisfied, one of the zero–one variables is fractional at optimality. We create two new subproblems, with the variable alternately fixed at zero and one. The new subproblems are added to the branch and bound tree as the children of the current subproblem. We then select the next subproblem to solve. The algorithm stops when there are no more unfathomed subproblems in the branch and bound tree.

Figure 1.1 shows the branch and bound tree for the following sample problem

$$\begin{aligned} \min \quad & 4x_1 + 6x_2 & & \\ \text{subject to} \quad & 2x_1 + 2x_2 & \geq & \ 1 \\ & 2x_1 - 2x_2 & \leq & \ 1 \\ & x_1, x_2 & \in & \ \{0, 1\}. \end{aligned}$$

The algorithm begins by solving the LP relaxation of this problem. The optimal solution is $x_1 = 0.5$, $x_2 = 0$, with optimal objective value $z = 2$. Since x_1 is fractional at optimality, the algorithm creates two new subproblems with $x_1 = 0$ and $x_1 = 1$. The algorithm next solves the subproblem with x_1 fixed at zero. The solution to this subproblem is $x_1 = 0$, $x_2 = 0.5$, with objective value $z = 3$. Again, the algorithm creates two new subproblems, with x_2 fixed at zero and one. The algorithm attempts to solve the subproblem with x_2 fixed at zero and discovers that the subproblem is infeasible. It then solves the subproblem with x_2 fixed at 1. This

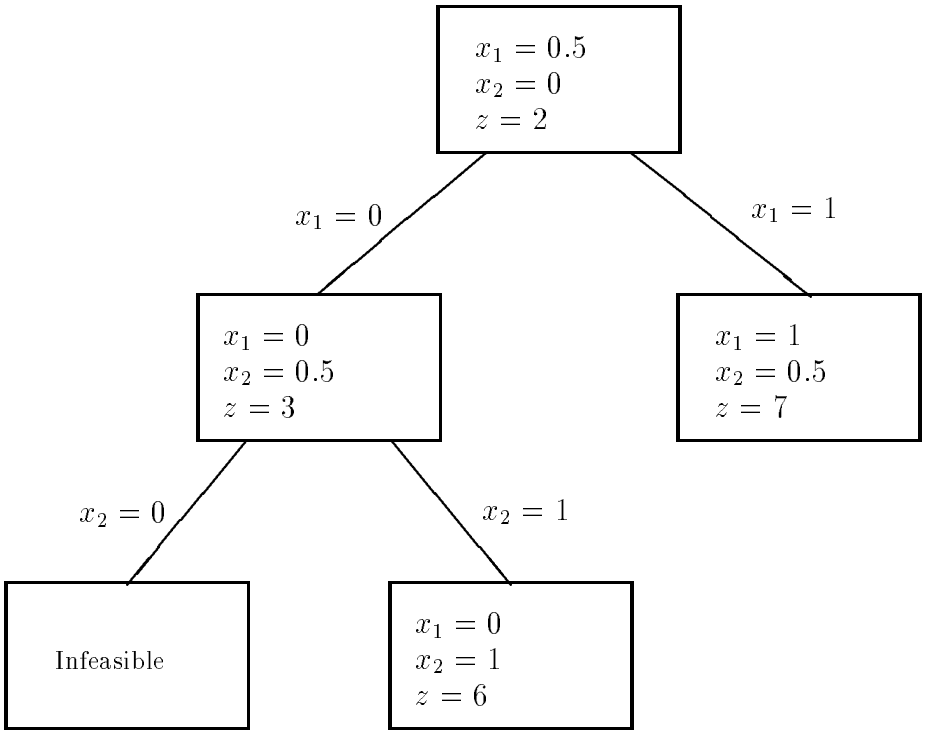

Figure 1.1: A branch and bound example.

subproblem has an integer optimal solution, with $x_1 = 0$, $x_2 = 1$, and optimal objective value $z = 6$. The algorithm then solves the subproblem with x_1 fixed at one. This subproblem has an optimal objective value of $z = 7$, and since this is larger than 6, the subproblem can be fathomed by bound. Thus the optimal solution is $x_1 = 0$, $x_2 = 1$, with objective value $z = 6$.

A flowchart of the basic branch and bound algorithm is shown in Figure 1.2. Although most of the steps in this algorithm are quite simple, some of the steps require additional explanation.

In step two, the algorithm picks the next subproblem. There are many options here, but most branch and bound algorithms rely on two basic strategies. In depth first search, the algorithm picks one of the most recently created subproblems. This results in subproblems with a large number of fixed variables, and these subproblems

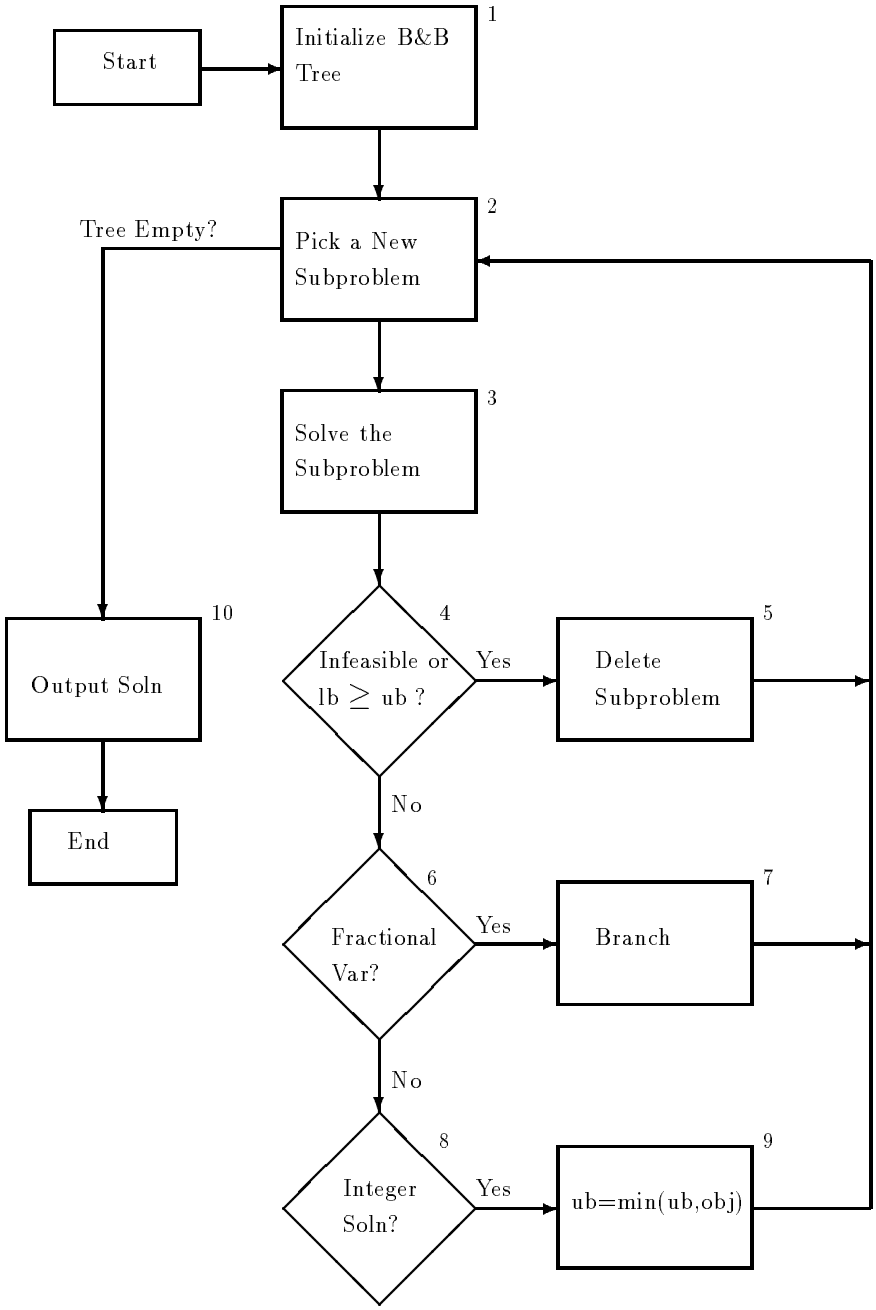

Figure 1.2: The branch and bound algorithm.

are more likely to have an integer solution. A second strategy, aimed at improving the lower bound on the optimal objective value as quickly as possible, is to pick the subproblem that has the lowest estimated objective value. There are many methods for estimating the optimal objective value of a subproblem, including the "penalties" proposed by Driebeck and Tomlin [18, 80] and "pseudocosts" described in [8].

In step three, any method for solving the continuous subproblems can be used. For mixed integer linear programs the simplex method has been very effective, but we will use interior point methods to solve the subproblems. For mixed integer nonlinear programs, any method for solving convex nonlinear programming problems can be used. Gupta and Ravindran used the Generalized Reduced Gradient (GRG) method [41]. Our experimental codes use the Sequential Quadratic Programming (SQP) method [24, 33].

In step six, we may find more than one fractional variable. The algorithm must decide which variable to branch on. One possibility is to pick the variable which is closest to an integer value. By branching on this variable and then solving the subproblem in which the variable has been rounded to the nearest integer value, we increase the chances of finding an integer solution. Another solution is to branch on the most fractional variable. The resulting subproblems are likely to have significantly larger optimal objective values than the parent subproblem. Pseudocosts and penalties can also be used to select the branching variable.

Implicit enumeration techniques are often used within a branch and bound algorithm to decrease the size of the branch and bound tree [72]. In implicit enumeration, the algorithm uses logical relationships implied by the linear constraints to fix variables at zero or one instead of explicitly enumerating combinations of the zero–one variables in the branch and bound tree. For example, if we are given the constraint $x_1 + x_2 \leq 1.5$, and x_1 has been fixed at one, then x_2 must be zero in any integer feasible solution. Implicit enumeration techniques can also be used to derive

lower bounds for subproblems. These techniques are used in IBM's Optimization Subroutine Library (OSL) [45].

Singhal, Marsten and Morin have developed a branch and bound code called ZOOM, in which the branching variables are selected before solving any of the subproblems [75]. This fixed order branch and bound scheme will be discussed further in Chapter 2.

In practice iterative algorithms are used to solve the subproblems. If an iterative algorithm computes lower bounds during the solution of a subproblem, these lower bounds can sometimes be used to fathom a subproblem by bound without solving the subproblem to optimality. For example, if the dual simplex method is used to solve an LP subproblem, and a dual solution is obtained that has a higher objective value than a known integer solution, then the current subproblem can be fathomed by bound. This method is used by a number of existing branch and bound codes [56].

We will develop branch and bound algorithms that use heuristics to detect zero–one variables which appear to be converging to fractional values. If the algorithm detects such a fractional variable, then it can branch early, without solving the current subproblem to optimality. A flowchart for the revised branch and bound algorithm is shown in Figure 1.3. After each iteration of the algorithm to solve the subproblem, we check to see if we can fathom the current subproblem by bound, or if there is a zero–one variable that appears to be converging to a fractional value. In the latter case, the algorithm branches without solving the subproblem to optimality.

If these heuristics err in declaring a variable fractional when it actually has an integer value at optimality, then the algorithm will waste time on the additional subproblems that are created. However, the algorithm will eventually find an optimal integer solution, since all possible integer solutions will still be enumerated.

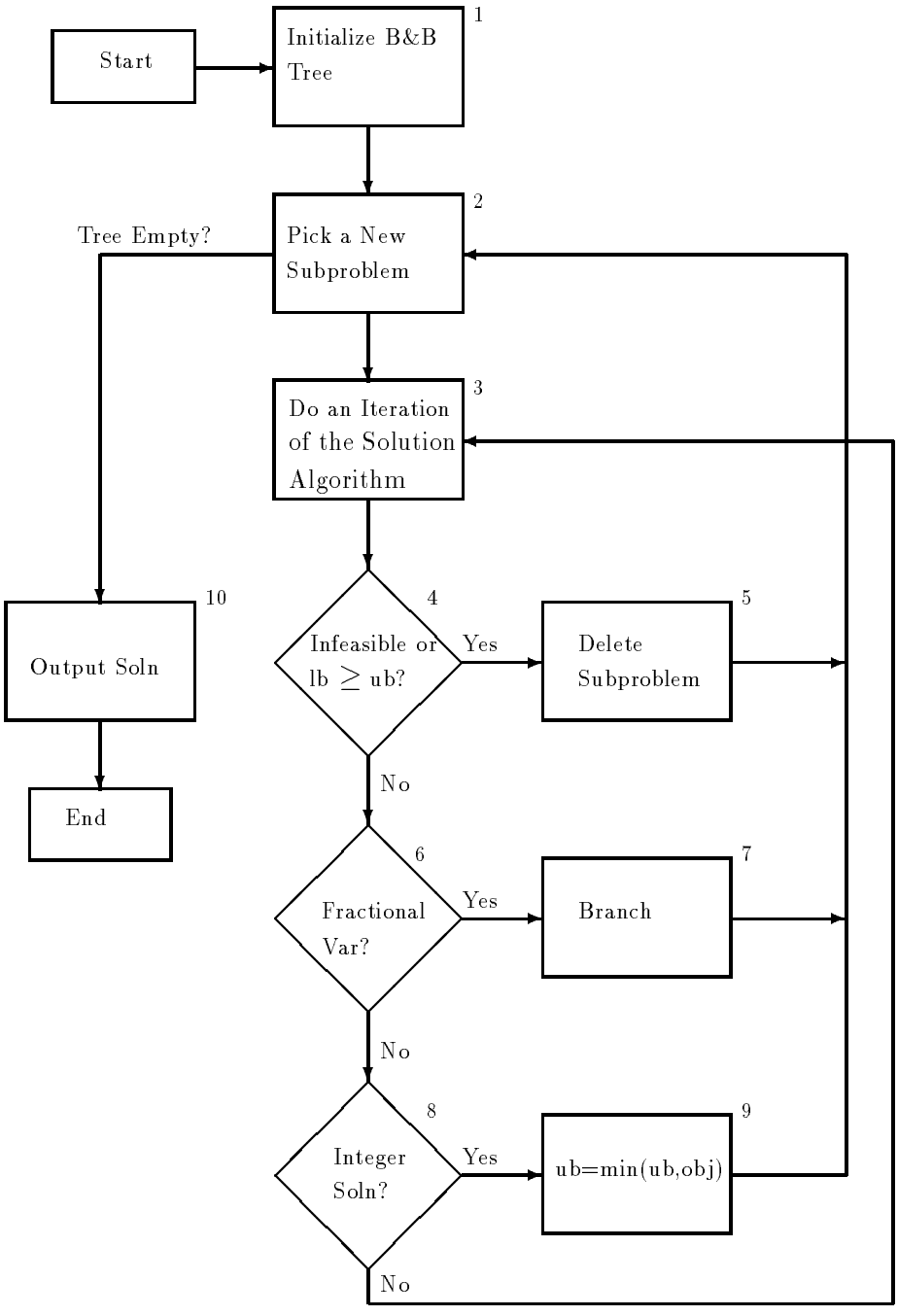

Figure 1.3: The revised branch and bound algorithm.

There are two other potential problems. The algorithm might branch early on a subproblem which is actually infeasible. Any further restrictions of the subproblem will also be infeasible, and the algorithm will eventually discover that each of these subproblems is infeasible. The algorithm might also branch early when the optimal value of the current subproblem is large enough to fathom the subproblem by bound. Again, the algorithm would waste time by doing this, but it would ultimately fathom all of the resulting subproblems.

In order to effectively apply the early branching idea, we need a method for solving the subproblems which gives good estimates of the optimal values of the zero–one variables, well before an optimal solution has been found. For linear problems, the commonly used simplex method is not a good choice, since a zero–one variable could become basic (and thus fractional) at the very last iteration of the simplex algorithm. Fortunately, interior point methods do give good estimates of the optimal solution relatively early in the solution process.

1.2 Interior Point Methods for Linear Programming

Recent interest in interior point methods dates back to 1984, when Narendra Karmarkar published a description of a new polynomial time algorithm for linear programming [50]. Karmarkar's work was of great practical and theoretical importance. Although Khachian had shown in 1979 that the ellipsoid method can be used to solve linear programming problems in polynomial time, the ellipsoid method was not successful in practice. Karmarkar's projective algorithm also had the desirable property of polynomial running time, but more importantly, versions of the algorithm were effective in practice.

Since the publication of Karmarkar's paper, the field of interior point methods for linear programming has grown quickly. A recent bibliography on interior point methods lists nearly a thousand papers [53]. There are a number of good survey

papers, including [34, 66, 77, 88]. Computational studies that indicate interior point methods can outperform the simplex method on various kinds of problems include [10, 13, 51, 60, 61].

We will use two interior point methods for linear programming. The first of these is the affine rescaling method, which Dikin described originally in the 1960's [16, 17]. Unfortunately, the algorithm was not developed into a practical method for solving linear programs at that time. This method was independently rediscovered in the 1980's as a variant of Karmarkar's algorithm [1, 4, 85]. We use a dual version of the affine rescaling method described in [67, 70]. The second method that we will use is the primal–dual path following method. This method is based on the sequential unconstrained minimization technique of Fiacco and McCormick [23]. We use a version of the algorithm developed by Lustig, Marsten, and Shanno in [14, 63, 68].

One reason that interior point methods had previously not been successful was the lack of good software for solving the systems of linear equations that arise in interior point methods. The matrices of these systems of equations often have far more zero entries than nonzero entries, and software can take advantage of this by saving only the nonzero entries in the matrix. Efficient codes for solving these sparse systems were first developed in the late 1970's. Our interior point codes make use of routines for solving sparse systems of equations from the Yale Sparse Matrix Package (YSMP) and IBM's extended scientific subroutine library (ESSL) [21, 44]. The books by George and Liu and Duff, Erisman, and Reid describe techniques for solving sparse systems of equations [19, 31].

Figure 1.4 shows the sequence of dual objective values generated by the dual–affine method in solving a sample problem. Note that the lower bounds given by these dual objective values are very good after only a few iterations, while 27 iterations are needed to solve the problem to optimality. Figure 1.5 shows the

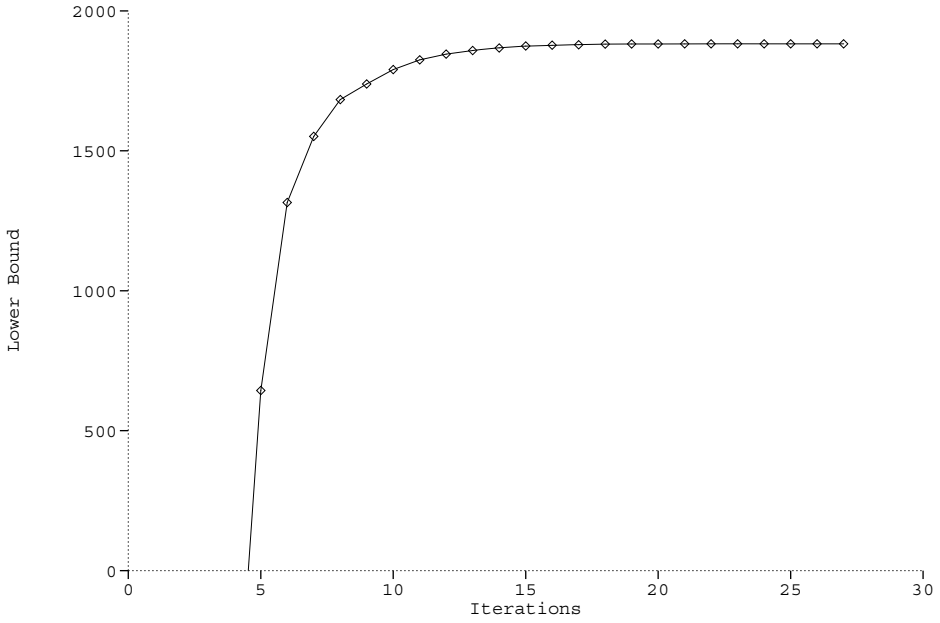

Figure 1.4: Lower bounds for a sample problem.

sequence of estimates of x_9 for the same problem. It's clear after about 15 iterations that x_9 will have a fractional value at optimality. In this example, the dual–affine method quickly produces useful lower bounds and estimates of the optimal values of the zero–one variables. As we shall see in Chapters 2 and 3, these properties of interior point methods are a very important factor in the performance of branch and bound codes based on interior point methods.

Throughout the thesis, we make use of the following notation. Upper case letters such as A are used to denote matrices. A_j refers to the jth column of the matrix A. Lower case letters are used to represent vectors and scalars. We will use $DIAG(x)$ to denote the diagonal matrix whose diagonal elements are the elements of the vector x. Frequently, we will associate a vector such as w with a diagonal matrix such as $W = DIAG(w)$. We sometimes use bold face type to distinguish vectors from scalars. We denote the jth element of a vector v by v_j. Superscripts

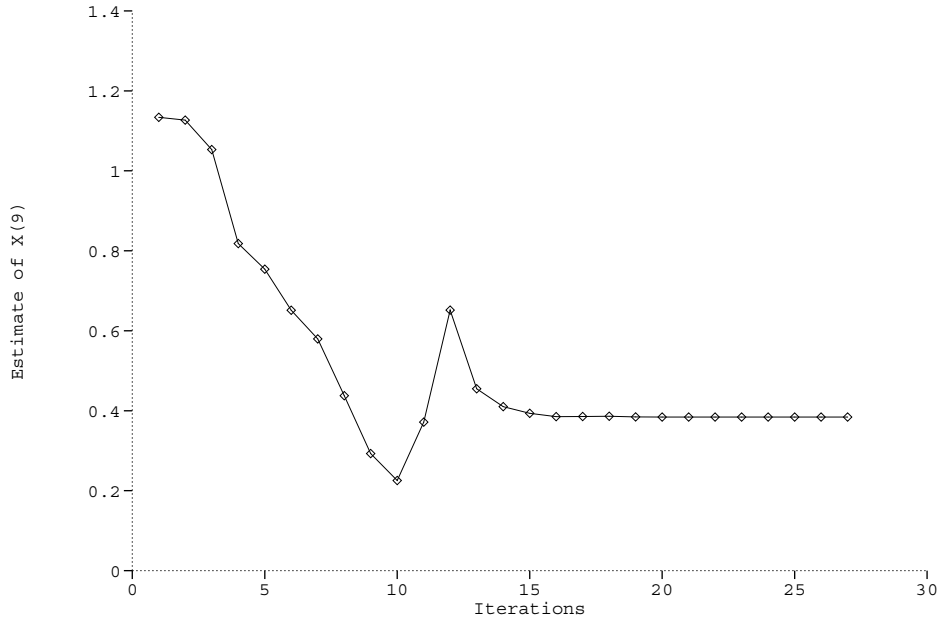

Figure 1.5: Estimates of x_9 for a sample problem.

are used to denote the iteration count within algorithms. Thus x^k would be used for the primal solution at the kth iteration of an algorithm. We use $|| x ||$ to denote the 2-norm of the vector x. We will also need the 1-norm, denoted by $|| x ||_1$.

The remainder of this thesis is organized as follows: In Chapter 2, we describe a branch and bound code for mixed integer linear programs that uses the dual–affine method to solve its LP subproblems. Computational results for four sets of sample problems are presented. In Chapter 3, we describe a branch and bound code for mixed integer linear programming problems that uses the primal–dual interior point method. Computational results for the four sets of sample problems from Chapter 2 and from three new sets of sample problems are given. In Chapter 4, we describe a branch and bound code for convex mixed integer nonlinear programs and present computational results from a number of sample problems. Our conclusions are presented in Chapter 5.

CHAPTER 2

Using the Dual–Affine Method in a Branch and Bound Code for Mixed Integer Linear Programming Problems

In this chapter, we describe two experimental branch and bound codes that use the dual–affine method to solve the LP subproblems. These codes use the early branching idea in an attempt to improve performance.

The remainder of this chapter is divided into sections. In sections 2.1 and 2.2, we describe the interior point method that we use to solve the subproblems. We develop procedures for warm starting the dual affine method and detecting fractional variables in sections 2.3 and 2.4. The branch and bound algorithm is described in section 2.5. Two strategies for selecting subproblems to be solved are described in section 2.6. In section 2.7, we describe a number of computational details that have affected the performance of the experimental codes. In section 2.8, we describe four sets of sample problems. Our computational results are presented in section 2.9.

2.1 The Dual–Affine Method

In this section, we present the dual–affine interior point method for solving linear programs. The method presented here is described in [67, 70].

We begin with a pair of primal and dual problems

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.1}$$

and

$$\begin{aligned} \max \quad & b^T y \\ \text{subject to} \quad & A^T y + z = c \\ & z \geq 0. \end{aligned} \tag{2.2}$$

We will assume that the constraint matrix A has full row rank. Consequently, in any feasible solution, y is determined by

$$y = (AA^T)^{-1}(Ac - Az)$$

The dual–affine method begins with an initial dual feasible solution y^0, z^0, with $z^0 > 0$. At each iteration, the algorithm rescales the problem so that z moves to e, the vector of all ones. In this rescaled problem, the algorithm moves in a direction that retains feasibility and improves the objective function. This direction is obtained by projecting the gradient of the objective function onto the null space of the constraints. The algorithm moves almost as far as possible in this direction without losing nonnegativity of z. This process is repeated until a convergence criterion is met.

At the kth iteration of the algorithm, we have a dual solution y^k, z^k. Let $Z = DIAG(z^k)$. Then let

$$\bar{z} = Z^{-1}z$$

$$\bar{A} = AZ^{-1}$$

and

$$\bar{c} = Z^{-1}c.$$

Now, the scaled problem is

$$\begin{array}{rrcl} \max & b^T y & & \\ \text{subject to} & \bar{A}^T y + \bar{z} & = & \bar{c} \\ & \bar{z} & \geq & 0. \end{array} \tag{2.3}$$

Here $\bar{z}^k = e$ is the current solution, and

$$y = (\bar{A}\bar{A}^T)^{-1}(\bar{A}\bar{c} - \bar{A}\bar{z}).$$

We can rewrite the scaled problem as

$$\begin{array}{lrcl} \max & b^T(\bar{A}\bar{A}^T)^{-1}(\bar{A}\bar{c} - \bar{A}\bar{z}) & & \\ \text{subject to} & \bar{A}^T(\bar{A}\bar{A}^T)^{-1}(\bar{A}\bar{c} - \bar{A}\bar{z}) + \bar{z} & = & \bar{c} \\ & \bar{z} & \geq & 0. \end{array} \tag{2.4}$$

We can ignore the constant term in the objective function and rewrite the equality constraints to get

$$\begin{array}{lrcl} \max & -b^T(\bar{A}\bar{A}^T)^{-1}\bar{A}\bar{z} & & \\ \text{subject to} & (I - \bar{A}^T(\bar{A}\bar{A}^T)^{-1}\bar{A})\bar{z} & = & (I - \bar{A}^T(\bar{A}\bar{A}^T)^{-1}\bar{A})\bar{c} \\ & \bar{z} & \geq & 0. \end{array} \tag{2.5}$$

Let

$$d\bar{z} = -\bar{A}^T(\bar{A}\bar{A}^T)^{-1}b$$

This is the gradient of the objective function of the scaled problem. The direction $d\bar{z}$ is already in the null space of the constraints, since

$$(I - \bar{A}^T(\bar{A}\bar{A}^T)^{-1}\bar{A})\bar{A}^T(\bar{A}\bar{A}^T)^{-1}b = 0.$$

In the original unscaled problem, this direction is equivalent to

$$dz = -A^T(AZ^{-2}A^T)^{-1}b \tag{2.6}$$

In order to retain dual feasibility, we must have $dz = -A^T dy$, so

$$dy = (AZ^{-2}A^T)^{-1}b \tag{2.7}$$

Note that since $b^T dy = b^T(AZ^{-2}A^T)^{-1}b$, $b^T dy$ is always positive. Thus the algorithm shows monotonic improvement at each step.

To determine the maximum possible step length, we solve the one-variable LP

$$\begin{array}{lrcl} \max & \alpha_d & & \\ \text{subject to} & z^k + \alpha_d dz & \geq & 0. \end{array} \tag{2.8}$$

We can now compute z^{k+1} and y^{k+1} using

$$z^{k+1} = z^k + 0.90\alpha_d dz$$

and

$$y^{k+1} = y^k + 0.90\alpha_d dy.$$

The factor of 0.90 ensures that z will remain strictly positive. In practice, it is better to compute

$$y^{k+1} = y^k + 0.90\alpha_d dy$$

and

$$z^{k+1} = c - A^T y^{k+1}.$$

This helps to ensure that the new solution will be dual feasible.

To find primal solutions, we'll solve the following least squares problem at each iteration of the algorithm

$$\begin{aligned} \min \quad & \textstyle\sum_{i=1}^{n} (x_i z_i)^2 \\ \text{subject to} \quad & Ax = b \end{aligned}$$

Any solution to this problem satisfies the primal constraints $Ax = b$. If we assume nondegeneracy, when z is dual optimal, the complementary slackness conditions ensure that we will find exactly one solution to the least squares problem with $\sum_{i=1}^{n} (x_i z_i)^2 = 0$. This will be an optimal primal solution. Unfortunately, when z is not optimal, the solution to this least squares problem may have elements that are less than zero. Thus the solutions to this least square problem will not always be primal feasible.

Let

$$x^{k+1} = -Z^{-2} dz$$

We claim that x^{k+1} solves the least squares problem. To see this, first multiply A by x^{k+1}.

$$\begin{aligned} Ax^{k+1} &= AZ^{-2}A^T dy \\ &= AZ^{-2}A^T(AZ^{-2}A^T)^{-1}b \\ &= b \end{aligned}$$

Next, compute the gradient of the objective function of the least squares problem, and project it onto the null space of the constraints. The gradient is $2Z^2x$. The projection onto the null space of the constraints is

$$\begin{aligned} p &= (I - A^T(AA^T)^{-1}A)2Z^2x^{k+1} \\ &= -2(I - A^T(AA^T)^{-1}A)dz \\ &= 2(I - A^T(AA^T)^{-1}A)A^T dy \\ &= 0 \end{aligned}$$

Since the projection of the gradient is zero, $x^{k+1} = -Z^{-2}dz$ is the optimal solution to the least squares problem.

An alternative procedure for finding primal solutions is described in [37, 79]. It has the advantage of generating truly feasible primal solutions. However, this method requires additional computations that increase the computational work per iteration. Furthermore, we sometimes found that the one-variable LP that occurs in this method was unsolvable because of numerical difficulties. So we have continued to use the simpler procedure for finding primal solutions. In practice, we have had no trouble finding good primal solutions using the technique described in this section.

There have been several proofs of convergence for primal and dual versions of the affine scaling algorithm [4, 17, 36, 81, 82, 84, 85]. Unfortunately, all of these proofs are based on assumptions of nondegeneracy or restricted step lengths. For example, the most recent paper by Tsuchiya and Muramatsu [82] shows that the algorithm converges when the step length is limited to 2/3 of the distance to the boundary of the feasible region. This paper also shows that the optimal primal and

dual solutions will be points in the interior of the optimal faces. This ensures that the optimal solution will have strict complementarity. The consequences of this will be discussed in section 2.4. Although our algorithm does not use a limited step length, the algorithm does converge in practice.

2.2 The Dual–Affine Method for Problems with Upper Bounds

The dual–affine method of the previous section has been extended to problems with upper bounds on the primal variables by Marsten, Saltzman, Shanno, Pierce, and Ballintijn [67]. In this section we derive the dual–affine method for problems with bounded primal variables and discuss ways of obtaining primal solutions.

We consider problems of the form

$$\begin{aligned} (P) \qquad \min \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \\ & x \leq u. \end{aligned}$$

Again, A is an m by n matrix, c is an n by 1 vector, and b is a m by 1 vector. For simplicity, we require that all the variables have explicit upper bounds. This restriction will be relaxed in Chapter 3.

The LP dual of this problem is

$$\begin{aligned} (D) \qquad \max \quad & b^T y - u^T w \\ \text{subject to} \quad & A^T y - w + z = c \\ & w, z \geq 0. \end{aligned}$$

We will reformulate these problems so that they fit into the framework of the previous section. Let

$$\hat{b} = \begin{bmatrix} b \\ -u \end{bmatrix}, \quad \hat{c} = \begin{bmatrix} c \\ 0 \end{bmatrix}, \quad \hat{x} = \begin{bmatrix} x \\ s \end{bmatrix}, \quad \hat{y} = \begin{bmatrix} y \\ w_{slack} \end{bmatrix}, \quad \hat{z} = \begin{bmatrix} z \\ w \end{bmatrix}$$

and

$$\hat{A} = \begin{bmatrix} A & 0 \\ -I & -I \end{bmatrix}$$

Our problems (P) and (D) are clearly equivalent to

$$\begin{aligned} (\hat{P}) \qquad \min \quad & \hat{c}^T\hat{x} \\ \text{subject to} \quad & \hat{A}\hat{x} = \hat{b} \\ & \hat{x} \geq 0 \end{aligned}$$

and the dual problem

$$\begin{aligned} (\hat{D}) \qquad \max \quad & \hat{b}^T\hat{y} \\ \text{subject to} \quad & \hat{A}^T\hat{y} + \hat{z} = \hat{c} \\ & \hat{z} \geq 0. \end{aligned}$$

Note that the equality constraints in $(\hat{D})$ imply that $w = w_{slack}$.

We can now apply the method of the previous section to problems $(\hat{P})$ and $(\hat{D})$. The equation $d\hat{y} = (\hat{A}\hat{Z}^{-2}\hat{A})^{-1}\hat{b}$ can be written as

$$\begin{pmatrix} AZ^{-2}A^T & -AZ^{-2} \\ -Z^{-2}A^T & Z^{-2} + W^{-2} \end{pmatrix} \begin{bmatrix} dy \\ dw \end{bmatrix} = \begin{bmatrix} b \\ -u \end{bmatrix} \tag{2.9}$$

Solving the above system of equations, we obtain

$$dw = (Z^{-2} + W^{-2})^{-1}(Z^{-2}A^T dy - u).$$

Substituting into the equations, we get

$$AZ^{-2}A^T dy = b + AZ^{-2}(Z^{-2} + W^{-2})^{-1}(Z^{-2}A^T dy - u).$$

So

$$A(Z^{-2} - Z^{-2}(Z^{-2} + W^{-2})^{-1}Z^{-2})A^T dy = b - AZ^{-2}(Z^{-2} + W^{-2})^{-1}u.$$

At this point, we need two easy identities

$$Z^{-2}(Z^{-2}+W^{-2})^{-1} = W^2(Z^2+W^2)^{-1}$$

and

$$Z^{-2} - Z^{-2}(Z^{-2}+W^{-2})^{-1}Z^{-2} = (Z^2+W^2)^{-1}.$$

Let

$$D^2 = (Z^2+W^2)^{-1}.$$

Then the equations for dy and dw become

$$dy = (AD^2A^T)^{-1}(b - AW^2D^2u)$$

and

$$dw = W^2D^2A^Tdy - Z^2W^2D^2u.$$

Finally, we must have $A^Tdy - dw + dz = 0$ to maintain dual feasibility, so we compute

$$dz = dw - A^Tdy.$$

Primal solutions can be obtained by

$$\hat{x} = -\hat{Z}^{-2}d\hat{z}$$

In terms of x and s, this is

$$x = -Z^{-2}dz$$

and

$$s = -W^{-2}dw.$$

However, since we want to ensure that $x + s = u$, it is simpler to compute

$$s = u - x.$$

The dual–affine method for problems with upper bounds on the primal variables is summarized in the following algorithm.

Algorithm 1

Given an initial dual feasible solution w, y, z, with $w, z > 0$, repeat the following iteration until some convergence criterion has been met.

1. *Let* $W = DIAG(w)$ *and* $Z = DIAG(z)$.
2. $D^2 = (Z^2 + W^2)^{-1}$
3. $dy = (AD^2A^T)^{-1}(b - AW^2D^2u)$
4. $dw = W^2D^2A^T dy - Z^2W^2D^2u$
5. $dz = dw - A^T dy$
6. $x = -Z^{-2}dz$
7. *Find the maximum step size* α_d *that ensures* $z, w \geq 0$. *If the step size is unbounded, the primal problem is infeasible and the dual is unbounded.*
8. $y = y + .90\,\alpha_d\,dy$
9. $w = w + .90\,\alpha_d\,dw$
10. $z = c - A^T y + w$

2.3 Starting and Stopping the Dual–Affine Method

Because the primal problem has explicit upper bounds on all of its variables, we can use the following formula, taken from [67], to generate an initial dual solution with $w, z > 0$. Let

$$y = \frac{\|c\|}{\|A^T b\|} b \tag{2.10}$$

If $c_j - A_j^T y \geq 0$ then let

$$w_j = \beta,\ z_j = c_j - A_j^T y + \beta \tag{2.11}$$

Otherwise, let

$$z_j = \beta,\ w_j = A_j^T y - c_j + \beta \tag{2.12}$$

where β is some positive constant. The experimental codes use $\beta = 50 \mid\mid c \mid\mid_1 /n$.

Each time we start work on a new subproblem in the branch and bound algorithm, we will need an initial dual feasible solution. Since a child subproblem is derived from its parent subproblem by fixing a primal variable at zero or one, the last dual solution of the parent subproblem will automatically be feasible for the child subproblem. However, some elements of z and w in this solution may be very small. The dual–affine method performs poorly when it is started from a point close to one of the boundaries of the feasible set. For this reason, we "center" the dual solution. By simultaneously increasing w_i and z_i by equal amounts, the dual solution is moved away from the boundary of the set of dual solutions so that the dual–affine method can proceed without difficulty. In our experimental codes, we add $\gamma = \mid\mid c \mid\mid_1 /(40n)$ to each element of w and z that is smaller than γ before starting work on a new subproblem. The choice of γ has consequences that will be discussed further in sections 2.4 and 2.7.

In step seven, the dual problem is considered unbounded if dz and dw are both nonnegative. In practice this seldom happens. Instead, because of numerical difficulties in the calculation of dz and dw, some elements of dz and dw that should be positive but near zero are really slightly negative. When this happens, the algorithm generates a sequence of dual solutions with steadily increasing objective values. Thus we stop the algorithm and declare the problem primal infeasible if no optimal solution has been found after 50 iterations.

Since the x iterates are not necessarily primal feasible, they do not provide upper bounds on the optimal objective value. Thus we cannot use the duality gap

to detect convergence. In our dual–affine code, we use the convergence criterion

$$\frac{|v_{n+1} - v_n|}{\max(1, |v_n|)} < \epsilon \tag{2.13}$$

where v_n is the dual objective value at the nth iteration. Our experimental codes use the value $\epsilon = 10^{-6}$. This has worked well in practice, although it provides no absolute guarantee that the current dual solution is near optimal.

Because the dual constraints involve $-w + z$, it is possible to simultaneously adjust w_i and z_i without losing dual feasibility. By simultaneously decreasing w_i and z_i, we decrease $u^T w$ and thus improve the dual objective value. Of course, this process is limited by the constraints $w \geq 0$ and $z \geq 0$. This method is used in the experimental codes to generate a good lower bound after each iteration of the dual–affine algorithm. The lower bound is given by

$$lb = \max(lb, b^T y - u^T(w - \bar{w})) \tag{2.14}$$

where $\bar{w}_i = \min(w_i, z_i)$.

2.4 Detecting Fractional Variables

To branch early, we need a heuristic for detecting zero–one variables which will be fractional at optimality. This heuristic does not need to be 100% accurate, but it should be able to find most fractional variables early in the process of solving an LP subproblem.

Our heuristic is based on the complementary slackness conditions. If x^* and y^*, w^*, z^* are optimal primal and dual solutions, and x_i^* is strictly between its upper and lower bounds, then $w_i^* = z_i^* = 0$. Our heuristic simply looks for zero–one variables with $w_i < \epsilon$ and $z_i < \epsilon$, where ϵ is a small positive constant. The choice of ϵ is highly dependent on the scaling of the dual problem (D). We must choose an ϵ which is smaller than any likely nonzero value of w_i^* or z_i^*. Our experimental codes use $\epsilon = || \ c \ ||_1 / (50n)$. This is typically small compared to the nonzero elements of

z^* and w^*. It is also slightly smaller then the centering parameter γ, so we can be sure that z_i and w_i will start off with values above the cutoff for declaring a variable fractional.

We assume that the optimal solution will have strict complementarity, so that we will not have $x_i^* = z_i^* = w_i^* = 0$. This is a consequence of Tsuchiya's work on global convergence of the affine scaling algorithm with restricted step length. Tsuchiya's work does not apply to our algorithm, because we take long steps. However, we have not encountered any problems with this assumption in practice.

If there are several fractional variables at optimality, the heuristic is unlikely to detect all of them during the same iteration of the dual–affine method. In order to detect as many fractional variables as possible, we wait until

$$\frac{|v_{n+1} - v_n|}{\max(1, |v_n|)} < 0.05$$

before searching for any fractional variables. This also increases the chance that we will fathom the current subproblem by lower bound instead of branching to create two new subproblems.

In the two experimental codes, a solution is declared integer and optimal when successive values of the dual objective differ by less than one part in 1,000,000, all primal variables are within one part in 1,000,000 of their upper and lower bounds, and the zero–one variables are within 0.0001 of zero or one. On the other hand, if successive values of the dual objective differ by less than 5%, x_i is a zero–one variable, w_i and z_i are less than ϵ, and x_i is between 0.0001 and 0.9999, then x_i is considered fractional and the code will branch early.

2.5 The Branch and Bound Algorithm

A flowchart of our branch and bound algorithm is shown in figure 2.1. This algorithm is similar to the branch and bound algorithm described in the introduction.

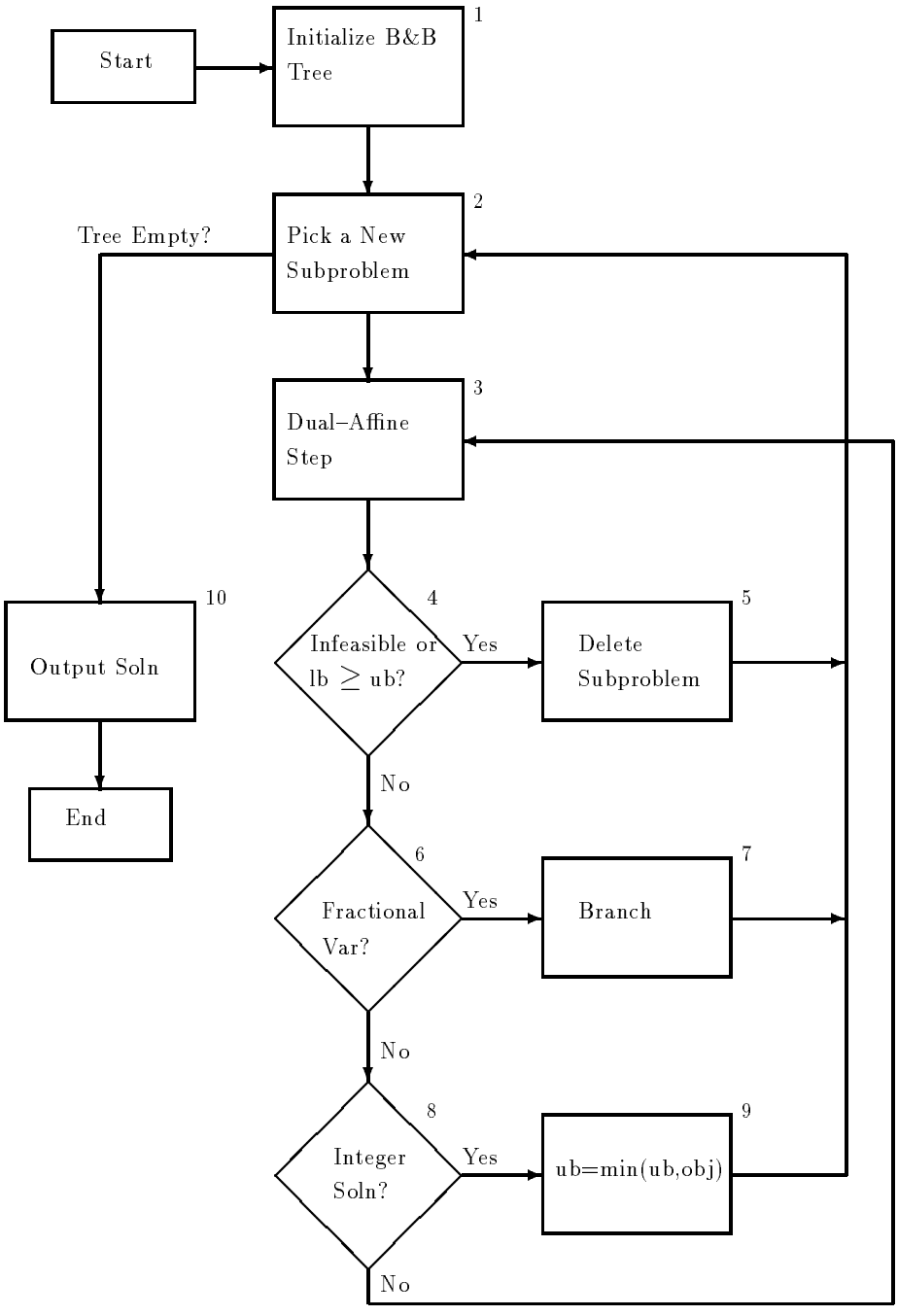

Figure 2.1: The branch and bound algorithm.

There are three important variables in the flowchart. The variable lb holds the best known lower bound on the objective value of the current subproblem. This bound comes from (2.14). The variable ub holds the objective value of the best known integer solution. This is used in step four to fathom the current subproblem if lb (a lower bound for the current subproblem) is larger than ub. The upper bound ub is updated in step nine, each time a new integer solution is found. The variable obj holds the optimal objective value of the current subproblem. This is used in step nine to update ub.

In step one, we initialize the branch and bound tree by creating a single node corresponding to the LP relaxation of the problem. At the same time, we find an initial dual feasible solution using the procedure of section 2.3 and set ub to $+\infty$.

In step two, a variety of strategies can be used to pick the next subproblem from the branch and bound tree. The two methods that we have implemented in our experimental code are described in section 2.6. When we have solved every subproblem in the branch and bound tree, the branch and bound procedure is finished and we can output the solution in step ten.

In step three, we perform a dual–affine step, as described in section 2.2. In order to do this, we need a dual feasible solution. The initial dual feasible solution for the first subproblem was calculated in step one. Each time we start work on a new subproblem, we can get a dual feasible solution from the parent subproblem. This solution is then centered as described in section 2.3. After the dual–affine step, we compute a new lower bound on the objective value of the current subproblem.

In steps four, six, and eight, we check for any of the conditions under which we will stop work on the current subproblem. If the lower bound lb is larger than ub, then we can delete the current subproblem and ignore all of its children in step five. In step six, we check for zero–one variables that appear to be fractional at optimality. If we detect a fractional variable, then we can branch to create two new

subproblems in step seven. In step eight, if we are at optimality and the current solution is integral, then we go on to step nine and compute a new upper bound.

The heuristics for detecting a fractional solution may err in declaring a variable fractional when it actually has an integer value at optimality or in declaring a variable fractional when the subproblem is actually infeasible. In either case, the algorithm will create two new subproblems instead of immediately fathoming the current subproblem or finding an integer solution. As noted in Chapter 1, the algorithm will eventually recover from either of these mistakes.

2.6 Branching Strategies

Two versions of this basic algorithm were coded. These two experimental codes differ in the manner in which they choose branching variables and the next subproblem to work on.

The first version of the code uses a depth first search strategy to choose subproblems until an integer solution has been found. During this early phase of the algorithm, the code branches on the least fractional variable, first rounding to the nearest integer value. The goal of this phase of the algorithm is to find an integer solution. The resulting upper bound can then be used to fathom subproblems.

Once an integer solution has been found, the algorithm enters a second phase in which it branches on the most fractional variable. The goal in this phase of the algorithm is to prune the branch and bound tree as quickly as possible. During this second phase of the algorithm, the code branches on the most fractional variable. In this phase, the code picks the remaining subproblem with the lowest estimated objective value. These estimates are based on pseudocosts, similar to those described in [56, 72]. Pseudocosts are heuristic estimates of the lower bound for a subproblem based on previous experience.

Each time a subproblem is solved to optimality, we record the difference in optimal objective values between the subproblem and its parent. A pseudocost is calculated for each integer variable by averaging the increase in objective value that occurs when that variable is fixed at zero or one. These pseudocosts are then added to the optimal objective value of a parent subproblem to get the estimated optimal values of its children. The estimated optimal values are used in choosing the next subproblem in step two of the branch and bound algorithm. As the branch and bound tree grows, the pseudocosts should become better estimates.

The second version of the experimental code uses a fixed order branch and bound strategy. In this strategy, used by the ZOOM system [75], the branching order is determined before solving any subproblems. As a consequence, all subproblems at the same level of the tree have the same fixed variables. Since fixing a primal variable corresponds to deleting a dual constraint, all subproblems at the same level in the tree have the same dual feasible sets.

After every fifth dual–affine iteration we use the current dual solution to evaluate the dual objective for each subproblem that is at the same level as the current subproblem. Sometimes a dual solution for the current subproblem is also a good dual solution for another subproblem with the same fixed variables. If the solution is good enough, it may even eliminate the other subproblem from consideration. This is called an "indirect hit" by the developers of the ZOOM system.

The fixed order branch and bound version of the code uses a depth first search strategy until it has found an integer solution. From then on, it picks the subproblem with the smallest lower bound. This lower bound is obtained by simply evaluating the dual objective function with the parent subproblem's optimal solution. If x_j is fixed at zero, then this is simply $b^T y - u^T w$. If x_j is fixed at one, then the lower bound is $(b - A_j)^T y - u^T w$. The fixed order branch and bound code does not make use of pseudocosts.

The fixed order branch and bound version of the program uses two level branching instead of single level branching. That is, instead of branching on one variable to create two subproblems, it branches on two variables to create four subproblems. This corresponds to the case $K = 2$ and $L = 1$ in [75]. In our experimental code, the branching order is based on the absolute values of the cost coefficients of the integer variables. Those variables with the largest cost coefficients come first in the branching order.

2.7 Computational Details

The algorithm maintains one large data structure, the tree of subproblems. Each leaf node in the tree is a record containing a list of variables that have been fixed at zero or one and the best known dual solution for that subproblem. This dual solution consists of $m + n$ numbers. (We store only y and w. The values of z can be easily calculated.) In contrast, a simplex based branch and bound code would normally save only a list of the variables that had been fixed at zero or one and a list of the variables in the current basis. Thus our codes use somewhat more storage than a simplex based code.

A simplex based code must refactor the basis each time it switches to a new subproblem. As we shall see later this is quite costly. The simplex based code could avoid this work by storing a factored basis for each subproblem. However, a factored basis requires $O(m^2)$ data storage, which is potentially far more expensive than the $O(m + n)$ storage required by the experimental codes.

The current versions of the codes handle up to 9,000 primal variables and 1,000 constraints, so each record takes up over 80,000 bytes of storage. Thus the entire tree can require many megabytes of storage.

The most difficult part of the computation is calculating dy in the dual–affine step. As long as the constraint matrix A is sparse and has no dense columns, the

matrix AD^2A^T should also be sparse. The experimental codes take advantage of this sparsity by saving the matrix in sparse form and making use of routines from the Yale Sparse Matrix Package [21] to factorize the matrix. After reading in the problem, the structure of AD^2A^T is calculated and the matrix is reordered by the minimum degree ordering routine of YSMP. The reordered matrix is then symbolically factorized. During each iteration of the algorithm, information gathered during the symbolic factorization is used in the numerical factorization of the matrix. The resulting Cholesky factors are then used in the computation of dy.

If the matrix A has dense columns, which would completely fill in AD^2A^T, they are handled separately with an update procedure based on the Sherman-Morrison-Woodbury formula. This formula is derived in [35]. Its application in an interior point method is described in [14]. Since none of our sample problems have any dense columns, this procedure has not been exercised.

Execution time profiling shows that much of the CPU time used by the two experimental codes is spent on factoring AD^2A^T and solving the resulting triangular systems of equations. Most of the remaining CPU time is spent on multiplying AD^2A^T and computing dw, dz, and x. For problems with more complicated structure, solving for dy should take a larger share of the CPU time. The most likely source of performance improvements would be better sparse matrix routines.

Note that the matrix A must have full row rank so that AD^2A^T can be factored, even after variables have been removed from the problem. This could become a problem if too many variables were fixed at zero or one. The difficulty could be resolved by removing rows from the matrix A as they become redundant. However, the experimental codes have no provision for removing such redundant rows.

One solution to this problem has been proposed in [70]. Whenever a zero pivot is detected during the factorization of AD^2A^T, the zero element is altered so as to give the matrix full rank, while setting the free variable to an arbitrary small value.

We have implemented this solution in our dual–affine code, but this has not been extensively tested, since none of our sample problems requires it.

There is a fundamental numerical problem in the dual–affine scheme, in that the matrix AD^2A^T becomes badly scaled as the solution approaches optimality. In particular, if x_i is a zero–one variable which is fractional at optimality, then z_i and w_i must go to zero to satisfy complementary slackness. Thus $D^2_{i,i}$ can become very large. Meanwhile, if x_j is at either its upper or lower bound, then $w_j^2 + z_j^2$ remains relatively large, and $D^2_{i,i}$ will remain small. When this happens, D^2 has some large elements and some small elements. This causes AD^2A^T to be badly scaled. This becomes very troublesome if we do not center the dual solution at the start of each subproblem, since if w_i or z_i is small when we start work on a subproblem it can be driven down to a very small value during the solution of the subproblem. In some computational runs without centering, we obtained values of w_i and z_i as small as 1.0×10^{-70}.

To deal with this problem, we modified the dual–affine algorithm slightly. At each iteration, we compute the square root of the sum of the squares of the elements of D^2. For each element in D^2 that is smaller than 1.0×10^{-25} times this norm, we fix the element at zero. This removes the tiny elements of D^2, and fixes the ill-conditioning of the matrix AD^2A^T. As a consequence, dw_i is zero, and $dz_i = -(A^T dy)_i$. Because of the way dw and dz are calculated, this does not result in a loss of dual feasibility. In terms of the primal solution, $D^2_{i,i}$ is very small only when x_i is at its upper or lower bound. Thus our procedure corresponds to fixing primal variables at their upper or lower bounds for an iteration when the complementary slackness conditions show that the variables must be non-basic.

If a subproblem is primal degenerate, then it may happen that D^2 will have fewer than m large entries as we approach optimality. In this case, AD^2A^T will become numerically singular. This problem can be handled by our modified version of YSMP, which arbitrarily restores full rank to the matrix.

As we noted earlier, there is also a numerical problem in detecting subproblems that are infeasible. When the algorithm works on an infeasible subproblem, it will generate a sequence of dual solutions whose objective values grow without bound, but it is unlikely that the unboundedness of the dual problem will be detected. The experimental codes deal with this difficulty in two ways. First, once a feasible integer solution has been found, infeasible subproblems will be fathomed by bound after only a few iterations. Second, if more than 50 iterations are needed to solve a subproblem, that subproblem is abandoned as being primal infeasible and dual unbounded.

Several changes were made to the basic algorithm to improve its performance. Primal solutions are not calculated in the dual–affine step unless the dual solution is close to optimality. In step ten of the dual–affine method, we use the formula

$$z = c - A^T y + w$$

only on every fifth iteration, and we use the simpler formula

$$z = z + .90\,\alpha_d\,dz$$

for all other iterations. The first formula ensures that the new dual solution satisfies the constraints $A^T y - w + z = c$, but it takes more effort to compute than the second formula. In this way, small violations of dual feasibility caused by the simpler formula are corrected before they can become large.

2.8 Sample Problems

We developed four sets of sample problems for computational testing. Each sample problem is a capacitated facility location problem with m facilities and n customer locations. Some of our problems are randomly generated, while others are taken from the literature. The capacitated facility location problem can be written as

$$\min \sum_{i=1}^{m} f_i x_i + \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} y_{ij}$$

$$\begin{aligned}
\text{subject to} \quad \textstyle\sum_{j=1}^{n} y_{ij} &\leq Cap_i x_i && \text{for } i = 1, ..., m \\
\textstyle\sum_{i=1}^{m} y_{ij} &= d_j && \text{for } j = 1, ..., n \\
x_i &\in \{0, 1\} && \text{for all } i \\
y_{ij} &\leq d_j && \text{for all } i,j \\
y_{ij} &\geq 0 && \text{for all } i,j
\end{aligned}$$

Here x_i is 0 if facility i is closed and 1 if facility i is open, y_{ij} is the flow from facility i to customer j, Cap_i is the capacity of facility i, f_i is the fixed cost of opening facility i, c_{ij} is the cost of shipping from facility i to customer j, and d_j is the customer j's total demand.

In the randomly generated test problems, f_i is uniformly distributed between 50 and 150, d_j is uniformly distributed between 1 and 10, and c_{ij} is uniformly distributed between 4 and 6. The Cap_i are given by $r_i \sum_{j=1}^{n} d_j/m$, where r_i is uniformly distributed between 1 and 2.

The first three sets of problems are randomly generated problems. Of the randomly generated problems, the problems in problem set one have 10 facilities and 400 customers, the problems in problem set two have 20 facilities and 400 customers, and the problems in problem set three have 10 facilities and 800 customers.

The fourth set of problems are a standard set of capacitated facility location problems taken from the literature. These problems were originally developed as

Problem Set	rows	columns	0–1 variables	density		
				A	AD^2A^T	L
1	410	4020	10	0.49%	5.00%	5.29%
2	420	8040	20	0.48%	9.31%	9.74%
3	810	8010	10	0.25%	2.56%	2.70%
4	66	832	16	2.97%	19.9%	26.5%

Table 2.1: Sample problem statistics.

uncapacitated problems by Kuehn and Hamburger in [54]. They used a table of railroad distances between fifty cities and the populations of the cities to generate uncapacitated facility location problems with up to 25 warehouses and 50 customers. Later, the same data was used by Akinc and Khumawala to generate capacitated problems in [2].

Our fourth set of problems is the problem set IV in the paper by Akinc and Khumawala. The problems are available through the OR-Library [7]. These problems have 16 potential warehouse locations, and 50 customer locations. The capacity of each warehouse is 5,000, while the total demand is 58,268. Thus the capacity constraints are fairly tight. The cost of opening a warehouse is \$7,500 in problem one, \$12,500 in problem two, \$17,500 in problem three, and \$25,000 in problem four. These costs are quite small compared to the total shipping costs at optimality.

Table 2.1 gives statistics on the size and density of the problems in these four problem sets. Under density, we give the density of the original constraint matrix A, the density of AD^2A^T, and the density of the Cholesky factor L. Note that in problem sets one, two, and three there is only a small increase in density during the factorization of AD^2A^T. In problem set four, the increase in density is more significant.

2.9 Computational Results

In this section we describe our computational results. In addition to seeing how fast our experimental codes are, there are a number of other questions about the performance of the experimental codes that we would like to answer. How effective are the early branching heuristics? How effective is our procedure for warm-starting the dual–affine method? How much storage is used by the experimental codes? In order to test the effectiveness of our experimental codes, we ran the experimental codes and a program using IBM's Optimization Subroutine Library subroutines [45, 87] on the four sets of problems described in the previous section.

All computations were performed on an IBM 3090–200S under the AIX/370 operating system. Computations were done in double precision. The experimental codes were written in Fortran and compiled with the FORTRANVS version 2 compiler and VAST2 preprocessor [46, 47]. Full optimization and vectorization was specified with the flags VEC and OPT(3). Times were measured with the CPUTIME subroutine of the Fortran run-time library. Measured CPU time does vary by as much as 5%, depending on system load and other random factors. These CPU times exclude the time required to read in the problems and to output solutions.

For the OSL runs, the EKKMSLV routine from release 2 of OSL was used. Default settings were used for the OSL code. The optimal objective function values and solutions from OSL were given to seven significant digits. Results for the experimental codes are also given to seven digits, although these codes were only designed to give answers good to about six significant digits.

For each set of problems, we first solved the LP relaxation of the problems using the OSL routine EKKSSLV and using the dual–affine method. These results are presented in Appendix A. We then solved the mixed integer linear programming problems with the two experimental codes and with the OSL code. For both the OSL and experimental runs we give the CPU time, number of iterations, number

of subproblems solved, and the optimal objective value. Detail results are shown in Tables 2.2 through 2.13.

In all cases, the experimental codes produced the same integer solutions as the OSL routines. The optimal objective values generally matched to at least six significant digits, and often matched to seven digits. Additional accuracy could be achieved at the cost of slightly more iterations. (Recall that only the subproblems that result in integer solutions need to be solved to optimality.)

Figure 2.2 shows the number of subproblems solved by the two experimental codes, relative to the number of subproblems solved by OSL. The experimental fixed order branch and bound code typically solved slightly fewer subproblems than OSL, while the experimental branch and bound code without fixed order branching typically solved more subproblems than OSL. The good results for the fixed order branch and bound code reflect the large number of subproblems that this code was able to fathom without actually solving. The experimental codes appear to have been effective in choosing branching variables and subproblems so as to minimize the size of the branch and bound tree.

Figure 2.3 shows the CPU time used by the two experimental codes to solve the four sets of problems, relative to the CPU time used by OSL. The fixed order branch and bound code is consistently faster than OSL on the first three sets of problems, while the experimental code without fixed order branching is faster than OSL on problem sets one and three. The relatively poor performance of the experimental branch and bound code on problem set two can be explained by the large number of subproblems that were solved. The poor performance of the experimental codes on problem set four can largely be explained by the difficulty that the dual–affine code had in solving the LP relaxations of these problems. The data in Appendix A show that OSL's simplex method was about three times faster than the dual–affine method in solving the LP relaxations of these problems.

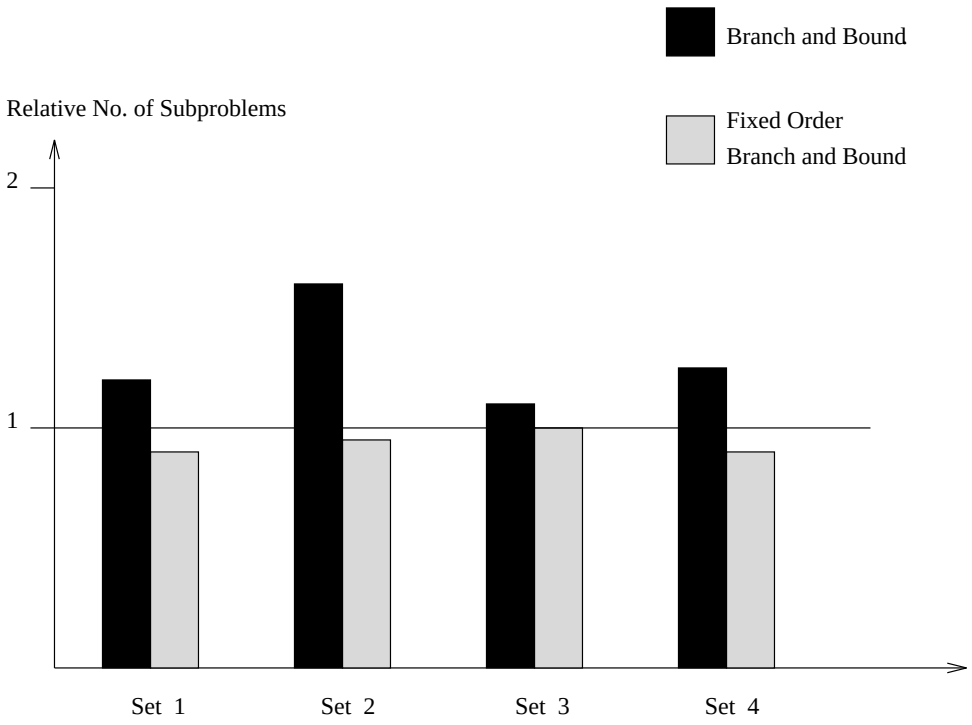

Figure 2.2: Subproblems solved relative to OSL.

One interesting aspect of the performance of OSL on these problems is that OSL slows down in terms of the number of iterations per second in the branch and bound algorithm. For example, OSL performs over 600 iterations per second in solving the LP relaxations of the problems in problem set one, but it performs fewer than 100 iterations per second in solving the mixed integer programs. A possible explanation for this is the need to refactorize the basis each time OSL begins work on a new subproblem. Another possible explanation is the time used in attempts to simplify the problem by implicit enumeration.

Another important measure of the performance of the experimental codes is the amount of storage required to hold the branch and bound tree. The experimental codes were compiled so that up to 240 megabytes of storage was available for the branch and bound tree. However, only about 70 megabytes were actually required

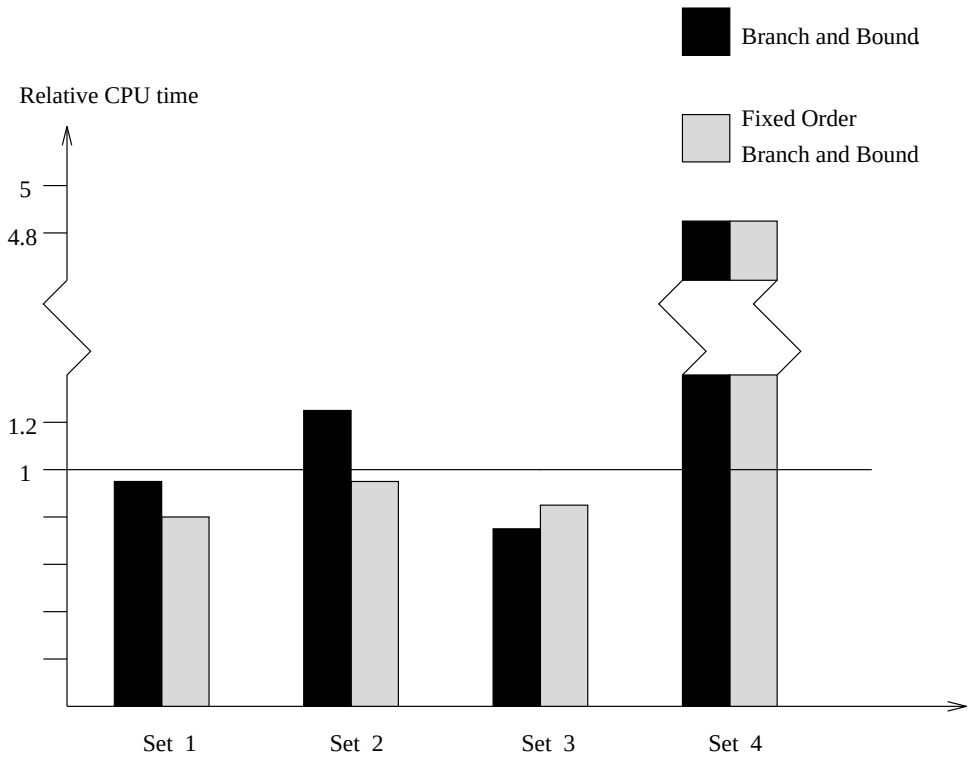

Figure 2.3: CPU time, relative to OSL.

by the largest problem. For the fixed order branch and bound solution of the first problem in set two, the branch and bound tree had 838 leaf nodes at its largest size. Since each leaf node required about 80,000 bytes of storage, the entire tree required about 70 megabytes of storage. The experimental branch and bound code needed a maximum of only 32 megabytes of storage. This happened on problem one of problem set one, where the branch and bound code required a maximum of 403 leaf nodes. In contrast, the OSL code used about 2 megabytes of storage, in direct access files, on most of the problems. Since the experimental branch and bound code without fixed order branching was much more efficient in terms of storage than the fixed order code, we have concentrated on the version of the code without fixed order branching in the remainder of this section.

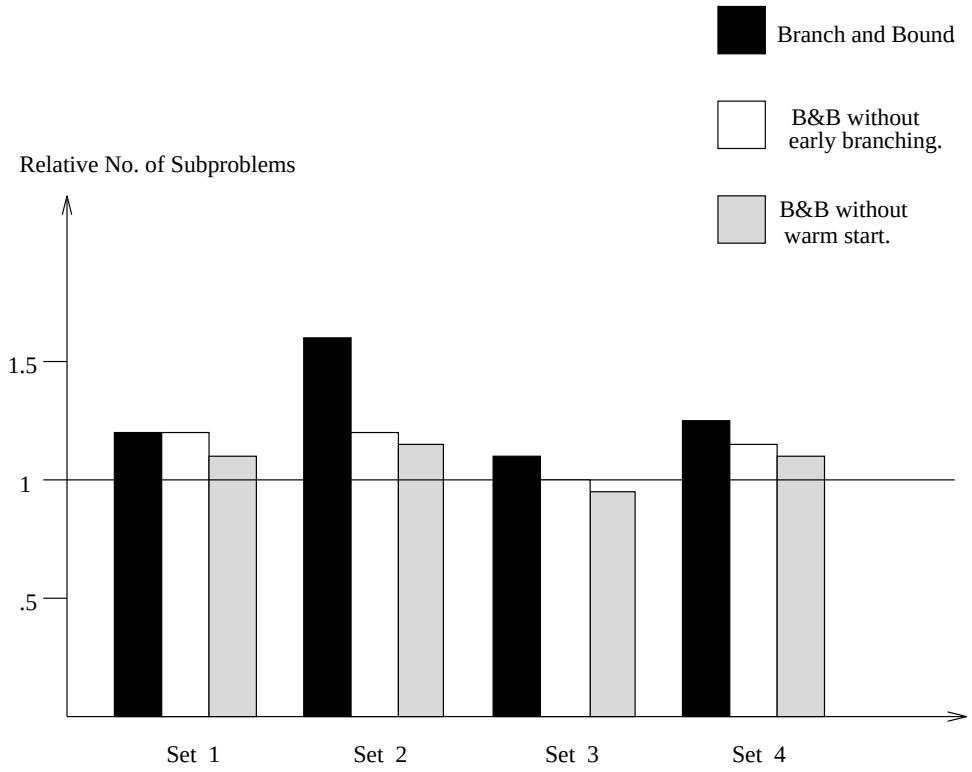

Figure 2.4: Subproblems solved, relative to OSL.

To test the effectiveness of the early branching heuristics and our warm–start procedure, we developed two more versions of the experimental code. The first of these codes solves each subproblem to optimality instead of branching early. With this code we can determine if the early branching heuristics are causing the code to solve extra subproblems. The second version of the code uses the procedure of section 2.3 to find a new dual solution for each subproblem instead of using the warm–start procedure. With this code, we can determine the effectiveness of the warm start procedure. Since the procedure for finding an initial dual solution generates solutions with large w and z values, we modified the code to search for fractional variables only after successive dual objective values are within 0.05% of each other. This gives the code time to drive the dual variables down to small

values so that fractional variables can be detected. Results for these new codes are presented in Tables 2.14 through 2.21.

Figure 2.4 shows the number of subproblems solved by the experimental branch and bound codes in problem sets one through four, relative to the number of subproblems solved by OSL. All three codes generally solved slightly more subproblems than OSL. The experimental code with early branching solved slightly more subproblems than the other two codes on most of the problem sets. However, on problem set two, the experimental code with early branching was significantly worse than the other two experimental codes. It appears that the early branching heuristics were performing poorly on problem set two.

Figure 2.5 shows the average number of iterations per subproblem for the experimental branch and bound codes on the four problem sets. The experimental code with early branching and the warm-start procedure used fewer iterations per subproblem than the other two codes. Of the experimental codes, the code without the warm-start procedure had the worst performance. These results show that the early branching heuristics and the warm-start procedure are effective in reducing the number of iterations per subproblem.

Figure 2.6 shows the CPU time used by the three experimental codes, relative to the CPU time used by OSL. Since the three versions of the experimental code solved roughly the same number of subproblems for each problem set, it is reasonable to expect that the number of iterations per subproblem will have the largest effect on the CPU times. The experimental code with early branching is consistently faster than the other experimental codes, even in cases where it solved more subproblems. This shows that the decreased number of iterations per subproblem is very worth while when compared to the slightly larger number of subproblems solved.

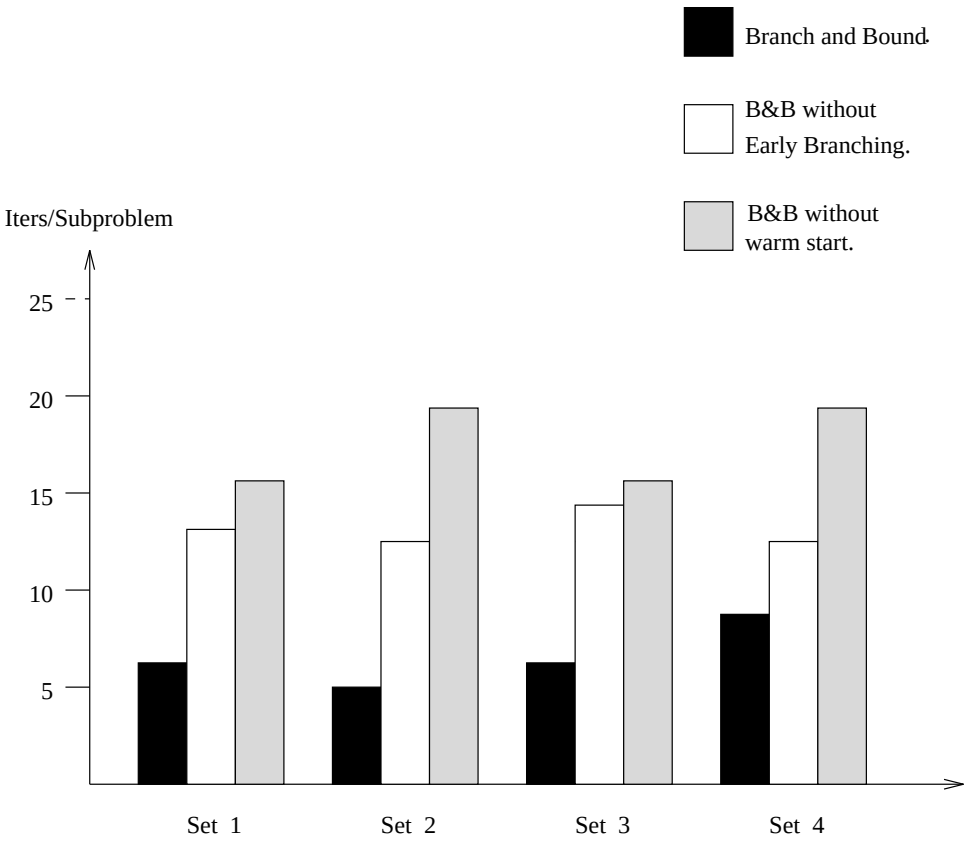

Figure 2.5: Iterations per subproblem.

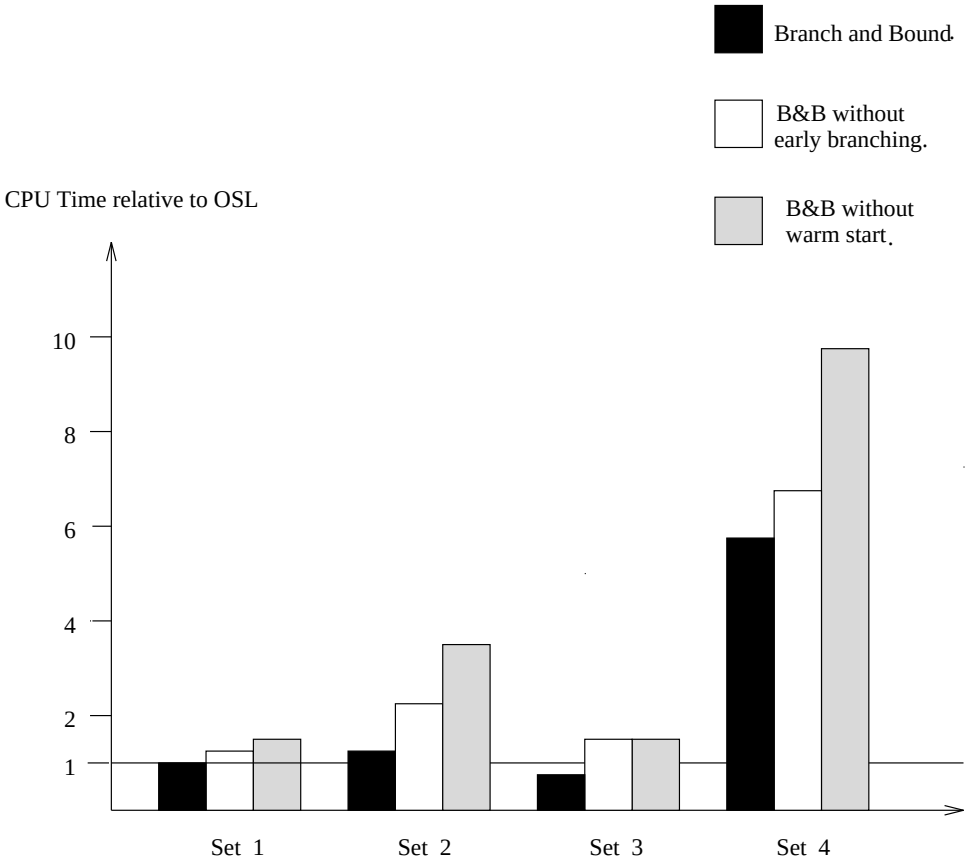

Figure 2.6: CPU time, relative to OSL.

Problem	CPU Time	Iterations	Subproblems	Objective
1	86.7	7663	146	10080.44
2	78.1	6983	114	9996.212
3	150.2	13061	237	10014.74
4	45.3	4079	65	10202.27
5	52.1	4572	90	9925.541

Table 2.2: OSL results for problem set one.

Problem	CPU Time	Iterations	Subproblems	Objective
1	70.8	958	161	10080.44
2	55.9	759	131	9996.214
3	133.9	1817	292	10014.73
4	35.5	484	70	10202.27
5	53.4	720	119	9925.539

Table 2.3: Branch and bound results for problem set one.

Problem	CPU Time	Iterations	Subproblems	Objective
1	67.7	911	167	10080.43
2	55.6	752	121	9996.214
3	99.2	1333	192	10014.73
4	30.6	412	57	10202.27
5	41.0	550	77	9925.539

Table 2.4: Fixed order branch and bound results for problem set one.

Problem	CPU Time	Iterations	Subproblems	Objective
1	611.4	27729	549	10024.90
2	273.2	12510	233	10319.08
3	361.4	16626	271	10414.04
4	368.0	16613	348	10512.00
5	413.2	18402	411	10187.25

Table 2.5: OSL results for problem set two.

Problem	CPU Time	Iterations	Subproblems	Objective
1	627.5	4148	853	10024.90
2	192.2	1278	213	10319.08
3	252.0	1670	325	10414.04
4	475.3	3162	603	10511.99
5	566.9	3763	657	10187.25

Table 2.6: Branch and bound results for problem set two.

Problem	CPU Time	Iterations	Subproblems	Objective
1	541.1	3300	545	10024.90
2	128.4	816	118	10319.08
3	208.8	1320	214	10414.04
4	350.7	2150	326	10511.99
5	343.1	2185	363	10187.25

Table 2.7: Fixed order branch and bound results for problem set two.

Problem	CPU Time	Iterations	Subproblems	Objective
1	267.6	12901	163	19709.22
2	133.0	6674	69	18784.34
3	326.4	15440	234	19518.01
4	371.6	17643	254	19542.94
5	454.8	21518	314	18933.19

Table 2.8: OSL results for problem set three.

Problem	CPU Time	Iterations	Subproblems	Objective
1	180.5	1222	186	19709.22
2	49.6	334	49	18784.33
3	242.3	1635	257	19518.00
4	276.3	1868	279	19542.93
5	333.3	2257	334	18933.19

Table 2.9: Branch and bound results for problem set three.

Problem	CPU Time	Iterations	Subproblems	Objective
1	211.3	1401	149	19709.22
2	87.2	576	57	18784.33
3	297.4	1931	217	19518.00
4	293.2	1935	274	19542.94
5	359.1	2374	345	18933.18

Table 2.10: Fixed order branch and bound results for problem set three.

Problem	CPU Time	Iterations	Subproblems	Objective
1	2.5	343	77	1040444
2	2.1	357	55	1098000
3	1.8	375	42	1153000
4	1.7	391	37	1235500

Table 2.11: OSL results for problem set four.

Problem	CPU Time	Iterations	Subproblems	Objective
1	10.7	683	77	1040444
2	7.7	494	55	1098000
3	7.8	504	61	1153000
4	6.4	414	45	1235500

Table 2.12: Branch and bound results for problem set four.

Problem	CPU Time	Iterations	Subproblems	Objective
1	7.5	478	44	1040444
2	8.1	516	45	1098000
3	8.2	524	42	1153000
4	8.7	561	42	1235500

Table 2.13: Fixed order branch and bound results for problem set four.

Problem	CPU Time	Iterations	Subproblems	Objective
1	149.1	2081	160	10080.44
2	116.8	1630	129	9996.214
3	24.9	3464	278	10014.73
4	84.8	1183	91	10202.27
5	111.6	1556	115	9925.539

Table 2.14: Problem set one without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	1131.1	7576	647	10024.91
2	340.9	2279	172	10319.08
3	434.4	2910	239	10414.04
4	724.4	4866	382	10512.00
5	1140.1	7672	614	10187.25

Table 2.15: Problem set two without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	431.1	2953	206	19709.22
2	146.8	1001	74	18784.34
3	421.0	2884	197	19518.00
4	490.1	3366	240	19542.94
5	662.6	4573	321	18933.19

Table 2.16: Problem set three without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	11.6	747	63	1040444
2	11.6	747	61	1098000
3	9.5	615	46	1153000
4	10.1	650	48	1235500

Table 2.17: Problem set four without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	179.0	2417	155	10080.44
2	129.2	1735	114	9996.214
3	30.0	4037	273	10014.73
4	86.3	1168	73	10202.27
5	114.8	1550	99	9925.539

Table 2.18: Problem set one without warm start.

Problem	CPU Time	Iterations	Subproblems	Objective
1	1705.5	11287	563	10024.91
2	439.2	2880	143	10319.08
3	546.7	3586	205	10414.04
4	1125.6	7382	401	10512.00
5	1850.0	12115	615	10187.25

Table 2.19: Problem set two without warm start.

Problem	CPU Time	Iterations	Subproblems	Objective
1	472.9	3120	197	19709.22
2	131.9	865	53	18784.34
3	449.5	2998	189	19518.00
4	521.8	3506	219	19542.94
5	748.9	5021	317	18933.19

Table 2.20: Problem set three without warm start.

Problem	CPU Time	Iterations	Subproblems	Objective
1	22.1	1410	71	1040444
2	18.3	1166	58	1098000
3	12.7	807	43	1153000
4	10.6	675	35	1235500

Table 2.21: Problem set four without warm start.

CHAPTER 3

Using the Primal–Dual Method in a Branch and Bound Code for Mixed Integer Linear Programming Problems

In this chapter, we describe an experimental branch and bound code that uses the primal–dual interior point method to solve its LP subproblems. The primal–dual method is described in sections 3.1 and 3.2. In section 3.3 we describe a new heuristic for detecting fractional variables. We discuss warm starting the primal–dual method in section 3.4. Some implementation issues are discussed in section 3.5. The branch and bound algorithm is presented in section 3.6. New test cases and computational results are presented in sections 3.7 and 3.8.

3.1 The Primal–Dual Method

In this section, we present the primal–dual interior point method, originally described in [14, 63, 68]. This method can be used to solve problems of the form

$$
\begin{array}{lrcl}
(P) & \min & c^T x & \\
& \text{subject to} \quad Ax & = & b \\
& x & \geq & 0 \\
& x & \leq & u.
\end{array}
$$

Again, A is an m by n matrix, c is an n by 1 vector, and b is an m by 1 vector. In this chapter, we do not require variables to have explicit upper bounds. If a variable x_i has no upper bound, we write $u_i = \infty$. We introduce a vector of slack variables, s, to make the constraints $x \leq u$ equality constraints. We then have the equality constraints $x + s = u$, and the inequalities, $s \geq 0$. If x_i has no upper bound, we write $s_i = \infty$.

The dual of this problem can be written as

$$\begin{array}{lll} (D) \qquad \max & b^T y - u^T w & \\ \text{subject to} & A^T y - w + z & = \quad c \\ & w, z & \geq \quad 0 \end{array}$$

The primal–dual method can be thought of as an example of the sequential unconstrained minimization method described by Fiacco and McCormick in [23]. At each iteration, the algorithm attempts to minimize the augmented objective function

$$c^T x - \mu \sum_{i=1}^{n} \ln x_i - \mu \sum_{i=1}^{n} \ln s_i$$

subject to the constraints

$$\begin{aligned} Ax &= b \\ x + s &= u \\ x, s &\geq 0. \end{aligned}$$

The constraint $x_i + s_i = u_i$ is ignored and the $\ln s_i$ term is not included in the barrier function if x_i has no upper bound.

Fiacco and McCormick have shown that as the barrier parameter μ is reduced from an initial positive value to zero, the optimal solution to the augmented problem approaches an optimal solution $(x^*, s^*, y^*, w^*, z^*)$ to (P) and (D). The curve defined by solutions $(x(\mu), s(\mu), y(\mu), w(\mu), z(\mu))$ is called the central trajectory. Figure 3.1 illustrates this idea. The primal–dual method simply follows the curve to an optimal solution.

In order to complete our development of the algorithm, we need a procedure for solving the augmented problem, an initial solution, a procedure for adjusting μ, and termination criteria.

The augmented problem is solved by finding the first order necessary conditions for the augmented problem, and then using iterations of Newton's method to move toward the solution of the first order necessary conditions. In the following, we use e

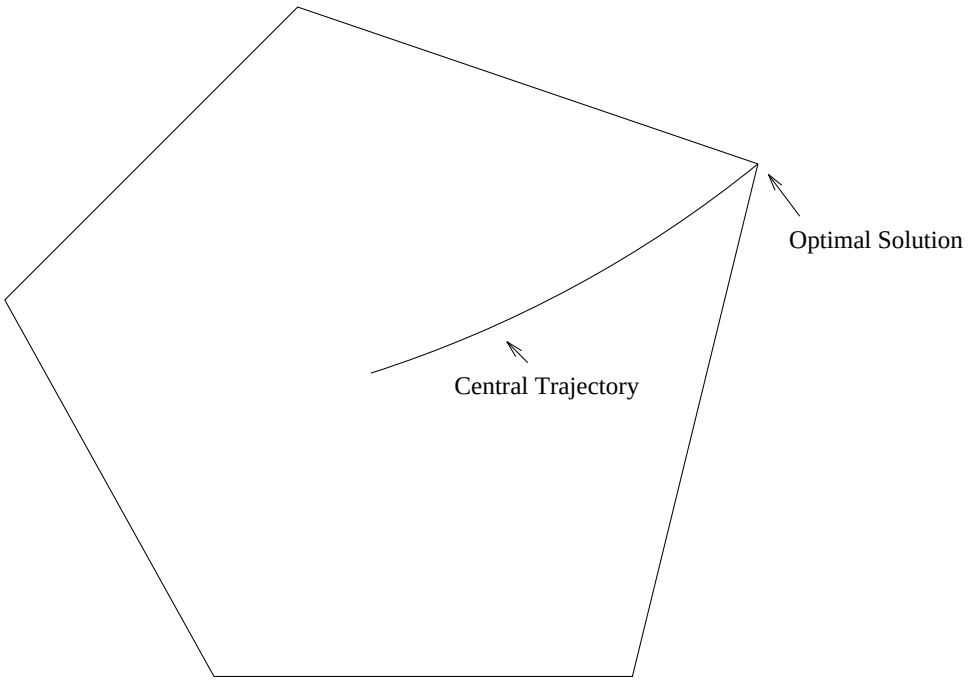

Figure 3.1: The central trajectory.

to represent a vector of all ones, and capital letters, such as W, to represent diagonal matrices in which the diagonal elements are the elements of a vector denoted by a lower case letter, such as w. The first order necessary conditions for the augmented problem can be written as

$$\begin{aligned} Ax &= b \\ x + s &= u \\ A^T y - w + z &= c \\ XZe &= \mu e \\ SWe &= \mu e \\ x, s, w, z &\geq 0. \end{aligned} \tag{3.1}$$

Applying Newton's method to this system of equations, we get

$$\begin{aligned} A\Delta x &= b - Ax \\ \Delta x + \Delta s &= 0 \\ A^T\Delta y - \Delta w + \Delta z &= c - A^T y - w + z \\ Z\Delta x + X\Delta z &= -XZe + \mu e \\ W\Delta s + S\Delta w &= -SWe + \mu e. \end{aligned} \tag{3.2}$$

Here we have assumed that we always have $x + s = u$, and that our current point always has $x, s, w, z > 0$, so that the current solution satisfies the nonnegativity constraints. However, we have not assumed that the current solution satisfies all of the equality constraints. In fact, this version of the primal dual method constantly moves towards primal and dual feasibility as well as optimality.

We need to solve these equations for the primal and dual Newton steps, Δx, Δs, Δy, Δz, and Δw. Multiply the last two equations by X^{-1} and S^{-1} respectively and subtract to get

$$\Delta z - \Delta w = -Ze + We - \mu(S^{-1} - X^{-1})e - X^{-1}Z\Delta x + S^{-1}W\Delta s.$$

Since $\Delta x = -\Delta s$, we have

$$\Delta z - \Delta w = -Ze + We - \mu(S^{-1} - X^{-1})e - X^{-1}Z\Delta x - S^{-1}W\Delta x.$$

Let

$$\rho(\mu) = \mu(S^{-1} - X^{-1})e - (W - Z)e$$

and

$$\Theta = (X^{-1}Z + S^{-1}W)^{-1}.$$

Note that Θ is a diagonal scaling matrix. Then

$$\Delta z - \Delta w = -\rho(\mu) - \Theta^{-1}\Delta x.$$

By substituting into the third equation, we can reduce the original system of equations to

$$\begin{aligned} A^T\Delta y &= c - A^T y - w + z + \rho(\mu) + \Theta^{-1}\Delta x. \\ A\Delta x &= b - Ax. \end{aligned}$$

We can write this system of equations in matrix form as

$$\begin{pmatrix} \Theta^{-1} & A^T \\ A & 0 \end{pmatrix} \begin{bmatrix} \Delta x \\ -\Delta y \end{bmatrix} = \begin{bmatrix} -(c - A^T y - w + z + \rho(\mu)) \\ b - Ax \end{bmatrix} \tag{3.3}$$

This system of equations is symmetric, but it is not positive definite. One approach to implementing the primal–dual algorithm is to solve this indefinite system of equations. We can then find Δs, Δz, and Δw from

$$\begin{aligned} \Delta s &= -\Delta x \\ \Delta z &= X^{-1}(-XZe + \mu e - Z\Delta x) \\ \Delta w &= S^{-1}(-SWe + \mu e - W\Delta s). \end{aligned} \tag{3.4}$$

Another approach to implementing the algorithm is to solve for Δy, using

$$(A\Theta A^T)\Delta y = (b - Ax) + A\Theta((c - A^T y - z + w) + \rho(\mu)). \tag{3.5}$$

Note that the matrix $A\Theta A^T$ is symmetric and positive definite. We can then solve for Δx with

$$\Delta x = \Theta(A^T\Delta y - \rho(\mu) - (c - A^T y - z + w)) \tag{3.6}$$

and we can find Δs, Δz, and Δw as before.

Kathryn Turner has considered the merits of these two approaches in [83]. The indefinite system of equations is larger than the positive definite system of equations. However, when we compute $A\Theta A^T$, the resulting matrix can be much denser than the original matrix A. In particular, if any column of A is completely nonzero, $A\Theta A^T$ will be completely filled in. Thus the indefinite system of equations typically has better sparsity than (3.5). We discuss these two methods of solving the equations further in section 3.5.

Ordinarily, in using Newton's method, we would now calculate the next x, s, y, z, and w by

$$\begin{aligned} x &= x + \Delta x \\ s &= s + \Delta s \\ y &= y + \Delta y \\ w &= w + \Delta w \\ z &= z + \Delta z. \end{aligned}$$

However, since we want to maintain nonnegativity of x, s, w, and z, we may have to use a smaller step size. To this end, we calculate

$$\begin{aligned} &\max && \alpha_p \\ &\text{subject to} && x + \alpha_p \Delta x \geq 0 \\ &&& s + \alpha_p \Delta s \geq 0 \\ &&& \alpha_p \leq 1/0.9995 \end{aligned} \tag{3.7}$$

and

$$\begin{aligned} &\max && \alpha_d \\ &\text{subject to} && w + \alpha_d \Delta w \geq 0 \\ &&& z + \alpha_d \Delta z \geq 0 \\ &&& \alpha_d \leq 1/0.9995. \end{aligned} \tag{3.8}$$

We then let

$$\begin{aligned} x &= x + .9995\alpha_p \Delta x \\ s &= s + .9995\alpha_p \Delta s \\ y &= y + .9995\alpha_d \Delta y \\ w &= w + .9995\alpha_d \Delta w \\ z &= z + .9995\alpha_d \Delta z. \end{aligned} \tag{3.9}$$

Thus if a step of length one is possible, the algorithm will take the full step. If a shorter step is required, the algorithm will take a step that is small enough to ensure that x, s, w, and z will remain strictly positive.

For an initial solution, we use the method described in [63]. First we compute the quantities

$$\begin{aligned} \xi_1 &= \max\left(\tfrac{|b^T Ae|}{\|Ae\|^2}, 1\right) \\ \xi_2 &= 1 \\ \xi_3 &= 0.1\xi_1. \end{aligned} \tag{3.10}$$

We then compute an initial solution with

$$\begin{aligned} x_i^0 &= \min\left(\xi_1, \tfrac{u_i}{2}\right) \\ s_i^0 &= u_i - x_i^0 \\ y_i^0 &= 0 \\ z_i^0 &= \begin{cases} |c_j|, & |c_j| \geq \xi_2 \\ \xi_2, & |c_j| \leq \xi_2. \end{cases} \end{aligned} \tag{3.11}$$

If x_i has an upper bound, we let

$$w_i^0 = \max\left(z_i^0 - c_i, \xi_2\right). \tag{3.12}$$

We consider the current solution feasible if

$$\frac{\|b - Ax\|}{1 + \|x\|} < 10^{-8} \tag{3.13}$$

and

$$\frac{\|c - A^T y + w - z\|}{1 + \|y\| + \|w\| + \|z\|} < 10^{-8}. \tag{3.14}$$

Instead of finding the minimum of the current augmented objective function and then adjusting μ, we adjust μ before each Newton's method iteration. If the current solution is feasible, then we let

$$\mu = \frac{c^T x - (b^T y - u^T w)}{\phi(n)} \tag{3.15}$$

where

$$\phi(n) = \begin{cases} n^2, & n \leq 5{,}000 \\ n^{1.5}, & n > 5{,}000. \end{cases}$$

If the current solution is not feasible, then we compute

$$\delta_p = M \frac{\|b - Ax\|}{\max(10^{-12}, \|b - Ax^0\|)}$$

and

$$\delta_d = M \frac{\|c - A^T y + w - z\|}{\max(10^{-12}, \|c - A^T y^0 + w^0 - z^0\|)}$$

where M is a constant given by

$$M = \rho\phi(n) \max\left(\max|c_i|, \max|b_j|\right).$$

We then compute μ by

$$\mu = \frac{\max(c^T x - (b^T y - u^T w), 0) + \delta_p + \delta_d}{\phi(n)}. \tag{3.16}$$

This procedure keeps μ large while the solution is still infeasible, but allows μ to decrease quickly once a feasible solution is found. By keeping μ large when the current solution is infeasible, we increase the possibility of making large steps towards the solution of the current augmented problem. If we can make a step of length 1, then a single step should give a new solution which is feasible. The parameter ρ can be adjusted to improve the performance of the algorithm. Small values of ρ emphasize improvement of the duality gap over improvement in feasibility, while large values of ρ emphasize improvement in feasibility. Our experimental codes use a value of $\rho = 0.1$.

We terminate the primal–dual algorithm and declare the current solution optimal when the current solution is primal and dual feasible, that is satisfying conditions (3.13) and (3.14), and when

$$\frac{|c^T x - (b^T y - u^T w)|}{1 + |b^T y - u^T w|} < 10^{-8}. \tag{3.17}$$

This ensures that the solution is primal and dual feasible and that the duality gap is small relative to the objective value.

If the linear programming problem is primal or dual infeasible, then the algorithm will go into a loop without ever finding a feasible solution. For this reason, it is advisable to stop the algorithm after some large number of iterations. In this case we have no way of determining why the algorithm has failed to converge to an optimal solution. However, in our branch and bound code it is reasonable to assume that the initial LP relaxation will be primal and dual feasible. As a result, all of the other subproblems in the branch and bound tree will be at least dual feasible, and we need only detect LP subproblems which are dual unbounded and primal infeasible.

In theory, we would know that a dual subproblem was unbounded if the current dual solution was dual feasible, $\Delta w \geq 0$ and $\Delta z \geq 0$. As with the dual–affine method in chapter 2, this does not work well in practice so we have developed an alternative procedure for detecting infeasible subproblems.

After each primal–dual iteration we compute directions $\Delta\bar{w}$ and $\Delta\bar{z}$ as follows. If x_i has an explicit upper bound, then compute

$$\Delta\bar{w}_i = \Delta w_i - \min(\Delta w_i, \Delta z_i)$$

and

$$\Delta\bar{z}_i = \Delta z_i - \min(\Delta w_i, \Delta z_i).$$

If x_i is an unbounded variable, then compute

$$\Delta\bar{w}_i = 0$$

and

$$\Delta\bar{z}_i = \Delta z_i.$$

If the previous dual solution was feasible, then the directions Δy, Δz and Δw are feasible directions, and by construction, the directions Δy, $\Delta\bar{z}$ and $\Delta\bar{w}$ are also feasible directions. If $b^T\Delta y - u^T\Delta\bar{w} > 0$, then Δy, $\Delta\bar{w}$ and $\Delta\bar{z}$ give us a

direction in which the dual solution is feasible and improving. If $\Delta\bar{w}$ and $\Delta\bar{z}$ are nonnegative, then we can move as far as we wish in this improving direction, and the dual problem is unbounded. This procedure for detecting infeasible subproblems is used in the experimental branch and bound codes described in this chapter. The procedure typically detects infeasible subproblems within less than five interations from a warm start.

3.2 The Primal–Dual Method and Complementary Slackness

Güler and Ye have shown in [39] that the optimal solution found by many interior point methods, including the primal–dual method of McShane, Monma, and Shanno, has strict complementary slackness. That is, given the primal and dual problems

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{3.18}$$

and

$$\begin{aligned} \max \quad & b^T y \\ \text{subject to} \quad & A^T y + z = c \\ & z \geq 0 \end{aligned} \tag{3.19}$$

the primal–dual method will produce an optimal solution (x^*, y^*, z^*) in which

$$x_i^* = 0 \Leftrightarrow z_i^* \neq 0.$$

Furthermore, Güler and Ye show that there is a constant ξ, such that

$$\frac{\min(XZe)}{x^T z} \geq \xi \tag{3.20}$$

at each iteration of the primal–dual method.

Unfortunately, Güler and Ye do their analysis with a version of the primal–dual method which does not include upper bounds. However, we can rewrite our

primal and dual problems (P) and (D) without explicit upper bounds. Let

$$\hat{b} = \begin{bmatrix} b \\ -u \end{bmatrix}, \quad \hat{c} = \begin{bmatrix} c \\ 0 \end{bmatrix}$$

Then let

$$\hat{x} = \begin{bmatrix} x \\ s \end{bmatrix}, \quad \hat{y} = \begin{bmatrix} y \\ w \end{bmatrix}, \quad \hat{z} = \begin{bmatrix} z \\ w_{slack} \end{bmatrix}$$

and

$$\hat{A} = \begin{bmatrix} A & 0 \\ -I & -I \end{bmatrix}$$

Our problems (P) and (D) are clearly equivalent to

$$\begin{aligned} (\hat{P}) \qquad \min \quad & \hat{c}^T x \\ \text{subject to} \quad & \hat{A}x = \hat{b} \\ & \hat{x} \geq 0 \end{aligned}$$

and the dual problem

$$\begin{aligned} (\hat{D}) \qquad \max \quad & \hat{b}^T \hat{y} \\ \text{subject to} \quad & \hat{A}^T \hat{y} + \hat{z} = \hat{c} \\ & \hat{z} \geq 0. \end{aligned}$$

Note that the equality constraints in $(\hat{D})$ imply that $w = w_{slack}$.

When we apply the primal–dual method of [68] to the problems $(\hat{P})$ and $(\hat{D})$, we get the necessary conditions

$$\begin{aligned} \hat{A}\hat{x} &= \hat{b} \\ \hat{A}^T \hat{y} + \hat{z} &= \hat{c} \\ \hat{X}\hat{Z}e &= \mu e \\ \hat{x}, \hat{z} &\geq 0. \end{aligned}$$

By inspection, we can see that these are exactly the necessary conditions given in (3.1). Thus the primal–dual method of McShane, Monma, and Shanno applied to

problems $(\hat{P})$ and $(\hat{D})$ is exactly equivalent to the primal–dual method developed in the previous section.

In terms of our original problems (P) and (D), the analysis of Güler and Ye shows that

$$x_i^* = 0 \Leftrightarrow z_i^* \neq 0$$

and

$$s_i^* = 0 \Leftrightarrow w_i^* \neq 0.$$

We also have that

$$\frac{\min(\min(XZe), \min(SWe))}{x^T z + s^T w} \geq \xi \tag{3.21}$$

at each iteration of the primal–dual method. This result has consequences which will be discussed further in sections 3.3 and 3.8.

3.3 Detecting Fractional Variables

A number of researchers have developed heuristics and sufficient conditions for detecting optimal basic and non-basic variables in interior point methods for linear programming [22, 73, 76, 78, 89]. When all of the optimal basic variables have been identified, it is very easy to compute an optimal basic solution. If this can be done well before the interior point method reaches optimality, then an interior point method that uses such indicators to find an optimal basic solution will have improved performance.

In our branch and bound algorithm for integer programming, we are interested in finding the zero–one variables which will be basic (and thus fractional) at optimality. This problem is similar to the problem that other researchers have worked on, except that we do not need an absolute guarantee that a variable is basic, and we only need to look at the zero-one variables. Once we have detected such a variable, we can proceed to branch without solving the current subproblem to optimality. We have considered three approaches to detecting fractional variables.

Our first approach, described in Chapter 2, used the complementary slackness conditions to detect fractional variables. If x_i has a fractional value at optimality, then both w_i and z_i must go to zero in order to meet the complementary slackness condition. Our experimental code simply examined the values of w_i and z_i. If both dual variables were below a specified tolerance, than x_i was declared a fractional variable. Since the primal–dual method finds an optimal solution that satisfies strict complementarity, if both z_i^* and w_i^* are zero, then x_i^* must be nonzero.

Our second approach used linear regression to fit a straight line to the last three values of x_i, w_i, and z_i. A variable will most likely be fractional at optimality if the slopes of these three lines are such that the x_i line stays between 0 and 1 for several iterations, while the lines for both w_i and z_i drop below zero within a few iterations.

We have also developed a third approach, based on Tapia's indicator [22]. The first order necessary conditions for the augmented problem include the equations

$$\begin{aligned} Z\Delta x + X\Delta z &= -XZe + \mu e \\ W\Delta s + S\Delta w &= -SWe + \mu e. \end{aligned}$$

Consequently,

$$\begin{aligned} Z(x+\Delta x) + X(z+\Delta z) &= XZe + \mu e \\ W(s+\Delta s) + S(w+\Delta w) &= SWe + \mu e. \end{aligned}$$

If we multiply the first of these equations by $X^{-1}Z^{-1}$, multiply the second equation by $S^{-1}W^{-1}$, and add the two equations together, then we get

$$X^{-1}(x+\Delta x)+Z^{-1}(z+\Delta z)+S^{-1}(s+\Delta s)+W^{-1}(w+\Delta w) = 2e+\mu(X^{-1}Z^{-1}+S^{-1}W^{-1})e$$

We will assume that as we approach optimality, $\mu(X^{-1}Z^{-1}+S^{-1}W^{-1})e$ goes to zero. Furthermore, the step lengths α_p and α_d approach one as we approach optimality. In the limit, $x^k+\Delta x \to x^{k+1}$, $s^k+\Delta s \to s^{k+1}$, $w^k+\Delta w \to w^{k+1}$, and $z^k+\Delta z \to z^{k+1}$. Thus

$$\frac{x_i^{k+1}}{x_i^k} + \frac{z_i^{k+1}}{z_i^k} + \frac{s_i^{k+1}}{s_i^k} + \frac{w_i^{k+1}}{w_i^k} \to 2. \tag{3.22}$$

If x_i is a zero–one variable which is fractional at optimality, then s_i is also fractional at optimality. We know that Δx_i and Δs_i approach zero, so

$$\lim \frac{x_i^k + \Delta x_i}{x_i^k} = \lim \frac{x_i^{k+1}}{x_i^k} = 1$$

and

$$\lim \frac{s_i^k + \Delta s_i}{s_i^k} = \lim \frac{s_i^{k+1}}{s_i^k} = 1.$$

This leaves no room for the remaining terms on the left hand side of (3.22), so we must have

$$\lim \frac{z_i^{k+1}}{z_i^k} = 0, \quad \lim \frac{w_i^{k+1}}{w_i^k} = 0.$$

If x_i is a zero–one variable which is zero at optimality, then $s_i^* = 1$, $w_i^* = 0$, and $z_i^* \neq 0$. We take advantage here of the fact that the optimal solution obtained by the primal–dual method will satisfy strict complementary slackness. Now,

$$\lim \frac{s_i^{k+1}}{s_i^k} = 1, \quad \lim \frac{z_i^{k+1}}{z_i^k} = 1$$

and

$$\lim \frac{x_i^{k+1}}{x_i^k} = 0, \quad \lim \frac{w_i^{k+1}}{w_i^k} = 0.$$

Similarly, if x_i^* is one, then

$$\lim \frac{x_i^{k+1}}{x_i^k} = 1, \quad \lim \frac{w_i^{k+1}}{w_i^k} = 1$$

and

$$\lim \frac{s_i^{k+1}}{s_i^k} = 0, \quad \lim \frac{z_i^{k+1}}{z_i^k} = 0.$$

This analysis relies on the assumption that the last term on the right hand side of (3.22) goes to zero. This is not true of our primal–dual method, but in practice, the term $\mu(X^{-1}Z^{-1} + S^{-1}W^{-1})e$ is small. First note that

$$\mu = \frac{c^T x - (b^T y - u^T w)}{\phi(n)} = \frac{x^T z + s^T w}{\phi(n)}.$$

We have that

$$\mu(X^{-1}Z^{-1} + S^{-1}W^{-1})e \leq \frac{2\mu}{\min(\min(XZe), \min(SWe))}e.$$

So, by (3.21),

$$\mu(X^{-1}Z^{-1} + S^{-1}W^{-1})e \leq \frac{2\mu}{\xi(z^T x + s^T w)}e$$

and

$$\mu(X^{-1}Z^{-1} + S^{-1}W^{-1})e \leq \frac{2(x^T z + s^T w)}{\xi\phi(n)(z^T x + s^T w)}e$$

and

$$\mu(X^{-1}Z^{-1} + S^{-1}W^{-1})e \leq \frac{2}{\xi\phi(n)}e.$$

This last upper bound is small enough to make the Tapia indicator work well.

In our experimental code, we use the Tapia indicator in the following way. Let

$$T_0 = \frac{x_i^{k+1}}{x_i^k} + |1 - \frac{z_i^{k+1}}{z_i^k}|$$

and

$$T_1 = \frac{s_i^{k+1}}{s_i^k} + |1 - \frac{w_i^{k+1}}{w_i^k}|.$$

If x_i is fractional at optimality, then T_0 and T_1 will both approach 2. If not, then T_0 and T_1 should both approach 0. We look for relatively large values of T_0 and T_1. We also insist that $x \geq 0.0001$ and $x \leq 0.9999$. In particular, if $T_0 \geq .9$, $T_1 \geq .9$, $0.0001 \leq x \leq 0.9999$, the current solution is dual feasible and within 10% of primal feasibility, and the duality gap is smaller than 5%, then we declare the variable x_i fractional.

There are two reasons for waiting until the duality gap is smaller than 5%. First, we increase the chance of fathoming the current subproblem by lower bound. Second, we are more likely to detect all of the variables that will be fractional at optimality.

This scheme for detecting fractional variables has a significant advantage over the scheme used in Chapter 2. The Tapia indicator is scale invariant, while the

ϵ parameter in Chapter 2 is very dependent on the scaling of the problem. The scheme in Chapter 2 needs a centering parameter that is larger than ϵ, while the Tapia indicator does not depend on centering at all.

3.4 Warm starting the Primal–Dual Method

How will we restart the primal–dual method when we switch to a new subproblem? We could use formulas 3.10 though 3.12 to find an initial primal–dual solution. However this makes no use of the information obtained by solving the parent subproblem. In contrast, a branch and bound code based on the simplex method can begin with the optimal solution to the parent subproblem and very quickly find a solution to the new subproblem.

In our dual–affine branch and bound code, we took the last dual solution to the parent subproblem and added ϵ to w and z to center the dual solution. Since fixing a primal variable at zero or one did not cause a loss of dual feasibility, the new solution was automatically dual feasible. By centering the dual solution, we avoided starting the dual–affine algorithm near a face of the dual feasible set, where the dual–affine method performs poorly.

In our new primal–dual branch and bound code, we would like an initial solution that is both primal and dual feasible. We would also like it to be relatively close to the central trajectory for the new subproblem. Unfortunately, the last solution to the parent subproblem will no longer be primal feasible for the child subproblem, and there is no obvious way to correct this infeasibility. Furthermore, the primal–dual method performs poorly when started from a solution near the boundary of the feasible region. As a result, the last solution to the parent subproblem is not a good starting point.

Lustig, Marsten, and Shanno have examined the problem of restarting the primal–dual method with a changed objective function in [62]. Their scheme for

restarting the primal–dual method involves adjusting primal variables or dual slacks that have small values. In this approach, if x_i or z_i is smaller than ξ, then the variable is set to ξ, where ξ is a parameter that they set to 0.001. Their method then attempts to adjust μ so as to speed up convergence.

Our approach is similar. If x_i or s_i is less than ξ, then we set the variable to ξ, and adjust the slack as needed. If w_i or z_i is less than ξ, and x_i has an upper bound, then we add ξ to both w_i and z_i. This helps to retain dual feasibility. If x_i has no upper bound and z_i is less than ξ, then we add ξ to z_i. Our code uses $\xi = 0.1$. The method of section 3.1 is used to calculate μ after this centering. However, we also reset ρ to 0.01 when we restart the primal–dual method.

3.5 Implementing the Primal–Dual Method

The method outlined in the previous sections is summarized in Algorithm 2.

Algorithm 2

Given an initial primal solution x, s with $x, s > 0$ and $x + s = u$, and an initial dual solution w, y, z, with $w, z > 0$, repeat the following iteration until the convergence criterion (3.17) has been met.

1. *Let $W = DIAG(w)$, $Z = DIAG(z)$, $X = DIAG(x)$, and $S = DIAG(s)$.*
2. *Compute μ using (3.15) or (3.16).*
3. *Compute $\Theta = (X^{-1}Z + S^{-1}W)^{-1}$ and $\rho(\mu) = \mu(S^{-1} - X^{-1})e - (W - Z)e$.*
4. *Compute Δx and Δy using either (3.3) or (3.5) and (3.6).*
5. *Compute Δs, Δw, and Δz using (3.4).*
6. *Find α_p and α_d by solving (3.7) and (3.8).*
7. *Use (3.9) to update x, s, y, w, and z.*

We need to pay some attention to primal variables which have no upper bound. Whenever x_i has no upper bound, we always keep $w_i = 0$ and $S_{i,i}^{-1} = 0$. In step 7, $\Delta w_i = 0$ if x_i has no upper bound.

Computationally, step 4 is the most difficult step in the primal–dual algorithm. We have implemented this step in three different ways.

Our first implementation used the Yale Sparse Matrix Package [21] to solve equation (3.5). As with the dual–affine code, the primal–dual code can find the nonzero structure of $A\Theta A^T$ and then reorder the matrix so as to reduce the fill-in incurred during LU factorization. The code then performs a symbolic factorization of the matrix that can be reused during each iteration of the algorithm. The only work that needs to be done during an iteration of the algorithm is the numerical factorization and solution of the resulting triangular systems of equations.

We have also implemented the primal–dual method using the DGSF and DGSS subroutines from IBM's Extended Scientific Subroutine Library [44] to solve (3.5). The DGSF routine performs minimum degree ordering and numerical factorization of the matrix $A\Theta A^T$. Unlike YSMP, the DGSF routine does not do the symbolic factorization of $A\Theta A^T$ once and then reuse the symbolic factorization in later iterations of the algorithm. Instead, DGSF does the factorization in one step, repeating the entire factorization process each time a system of equations is to be solved. This makes it possible for DGSF to factor indefinite matrices using pivoting to maintain stability. The DGSS routine does the work of solving the resulting triangular systems of equations. The ESSL routines are more efficient than the YSMP routines on relatively dense problems. For this reason we have chosen to use the ESSL routines on problems with relatively dense constraint matrices.

As in the dual–affine method, we can run into trouble if $A\Theta A^T$ is singular. This will happen if A does not have full row rank. The matrix may also become numerically singular if the optimal solution is primal degenerate. If the optimal

solution is primal degenerate, fewer than m primal variables have $0 < x_i < u_i$ at optimality. In this case, Θ will have many small elements, corresponding to variables at their upper or lower bounds, and fewer than m large entries, corresponding to variables that are strictly between their bounds. The large entries will dominate, making $A\Theta A^T$ numerically singular.

We have modified the YSMP routines to handle this problem as described in Chapter 2. However, the ESSL routines could not be modified to handle this situation since the source code is not available. For the sample problems that we have solved using the ESSL routines, this was not a problem.

The DGSF routine can also be used to factor the indefinite matrix in (3.3). Our third implementation of the primal–dual method used the ESSL routines to solve the indefinite system of equations. For problems where m was relatively large compared to n, this version of the primal–dual code worked quite well. However, for the problems in this thesis, which have far more columns then rows, the $m + n$ by $m + n$ indefinite system of equations (3.3) was far larger than the m by m positive definite system (3.5). For these problems, the improved sparsity of the indefinite system of equations could not compensate for the larger size of the indefinite system.

As with the positive definite system of equations, the indefinite system will become singular if A does not have full row rank or if the optimal solution is primal degenerate. Since the ESSL routines cannot solve such a singular system of equations, this approach cannot be applied to a problem in which the matrix becomes singular.

3.6 The Experimental Branch and Bound Code

We developed an experimental branch and bound code based on the branch and bound code of Chapter 2, but using the primal–dual method, heuristics for detecting fractional variables and centering scheme described in sections 3.1 through

3.4. A flowchart of the algorithm is given in Figure 3.2. This algorithm is very similar to the algorithm described in Chapter 2. However, a number of details have been changed.

Instead of storing a dual solution at each node of the branch and bound tree, we must now store a primal and dual solution at each node. In particular, we save x, y, and w. The remainder of the solution (s and z) can easily be recalculated when it is needed. As a result, the nodes in the branch and bound tree are about twice as large as in the dual–affine code described in Chapter 2.

Since the fixed order branch and bound scheme used excessive amounts of storage, we decided to restrict our attention to the conventional branch and bound scheme. In this version of the code, as in Chapter 2, the algorithm uses a depth-first search strategy until an integer solution has been found. During this early phase of the algorithm, the code branches on the least fractional variable. The code then chooses the subproblem in which the fractional variable has been rounded to the nearest integer value. Once an integer solution has been found, the algorithm switches over to a strategy of picking the subproblem with the best (smallest) estimated objective value. These estimates are based on pseudocosts as in Chapter 2. In this phase of the algorithm, we continue to branch on the most fractional variable.

In step four, the algorithm uses a lower bound lb obtained in one of two ways. An initial lower bound for the current subproblem is inherited from its parent subproblem. After each iteration of the primal–dual method the lower bound is updated with

$$lb = \max(lb, b^T y - u^T(w - \bar{w}))$$

where $\bar{w} = min(w, z)$. However, this can only be done when the current solution is dual feasible. The test described in section 3.1 is used to detect infeasible subproblems.

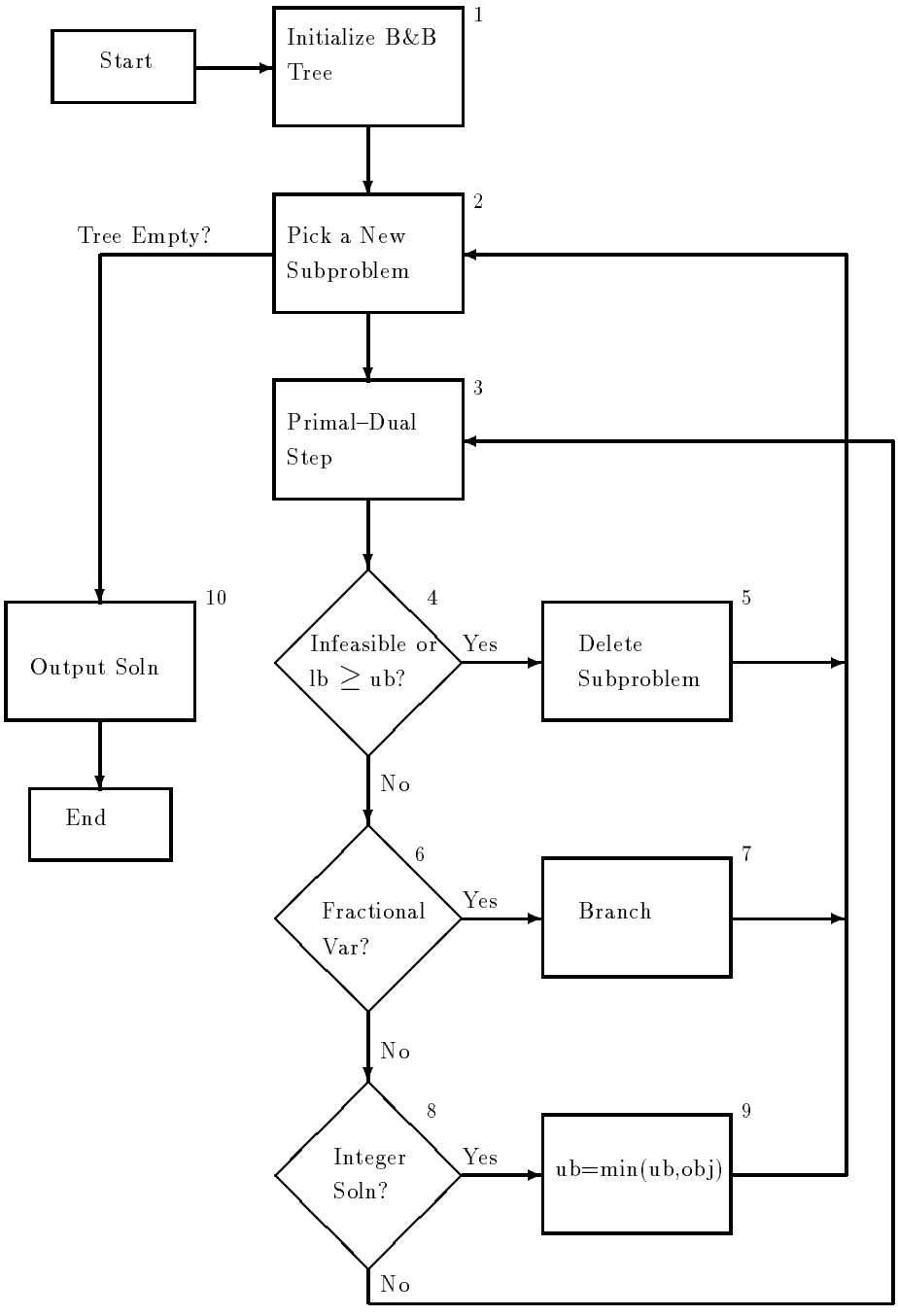

Figure 3.2: The branch and bound algorithm.

3.7 More Sample Problems

We collected six new sets of sample problems in addition to the sample problems described in section 2.9. These sets of problems differ from the earlier problems in that they have some unbounded variables.

AT&T Bell Laboratories supplied the problems in problem set five. The problems differ only in the right hand side, which is changed so that the problems become progressively harder. Although the original problem set consists of four problems, we have only been able to solve the first two problems in the problem set to optimality. Each problem is written as a mixed integer linear program with 198 constraints and 1927 variables, of which 25 are 0–1 variables. The constraint matrices have 7502 nonzero entries.

The problems in sets six and seven are set covering problems of the form

$$\begin{array}{lrcl} (SCP) & \min & c^T x & \\ & \text{subject to} & Ax & \geq \quad e \\ & & x & \in \quad \{0,1\}. \end{array}$$

Here A is an m by n zero–one matrix, and e is the vector of all ones. These problems were taken from Beasley's OR-Lib [7]. The problems were randomly generated using a procedure described in [3, 6]. After adding slack variables, the problems in set six have 300 rows and 3300 columns, with a 1.86% dense constraint matrix. The problems in set seven have 400 rows and 4400 columns with a 1.84% dense constraint matrix. The optimal values for these problems are taken from [6].

The final three problems are taken from the MIPLIB collection of mixed integer linear programming problems [11]. Problem eight is the problem khb05250 from MIPLIB. After adding slack variables and removing fixed variables, this problem has 101 constraints, 1324 variables, of which 24 are zero–one variables. Problem nine is the problem mod011 from MIPLIB. This problem has 4480 constraints, 11070 variables, and 96 zero–one variables. Problem ten is the problem misc06

from MIPLIB. This problem has 820 constraints, 2404 variables, and 112 zero-one variables. These problems were selected because they were large problems with a relatively small number of zero-one variables. Most of the other problems in MIPLIB are either very small, have many zero-one variables, or have integer variables that are not restricted to the values zero and one.

Because of the relatively large size of problem nine and because there are many zero–one variables in this problem, the problem is very difficult to solve. We have been unable to solve the mod011 problem with OSL or with our experimental branch and bound code. In one run, OSL used over 18,000 CPU seconds and solved over 50,000 subproblems without solving mod011 to optimality. Because of this, we ran the branch and bound codes only until the first integer solution was found.

Table 3.1 gives statistics on the size and density of the problems in each set. For each problem we also give the density of the matrix $A\Theta A^T$ and of the Cholesky factor L after minimum degree ordering. The density for L includes the diagonal entries.

Problem Set	rows	columns	0–1 variables	density		
				A	$A\Theta A^T$	L
1	410	4020	10	0.49%	5.00%	5.29%
2	420	8040	20	0.48%	9.31%	9.74%
3	810	8010	10	0.25%	2.56%	2.70%
4	66	832	16	2.97%	19.9%	26.5%
5	198	1927	25	1.97%	32.5%	46.6%
6	300	3300	3000	1.86%	70.1%	98.3%
7	400	4400	4000	1.84%	79.7%	99.3%
8	101	1324	24	1.96%	26.9%	31.5%
9	4480	11070	96	0.05%	0.16%	0.26%
10	820	2404	112	0.33%	1.15%	3.37%

Table 3.1: Sample problem statistics.

We solved the LP relaxations of these problems using OSL's primal–dual method, our experimental primal–dual LP solver, and OSL's simplex method. The results of these runs are presented in Appendix A.

3.8 Computational Results

Both OSL and the experimental branch and bound code were used to solve the four sets of problems described in Chapter 2 and the new sets of problems described in section 3.7. As in chapter 2, all computational runs were done in double precision. The computations were performed under AIX/370 on an IBM 3090–200S. The experimental codes were written in Fortran and compiled with the FORTRANVS compiler and VAST2 preprocessor [46, 47]. Full optimization and vectorization was specified with the compiler options OPT(3) and VEC. As in Chapter 2, the OSL results were obtained with the EKKMSLV routine from release 2 of OSL. Times were measured with the CPUTIME subroutine of the Fortran run-time library. These CPU times exclude the time required to read in the problems and to output solutions.

In order to examine the effectiveness of the early branching heuristics, we developed a version of the experimental branch and bound code in which all subproblems were solved to optimality before branching. A version of the experimental code was also developed to test the effectiveness of the warm start procedure. This version of the code generated a new initial solution for each subproblem instead of centering the last solution to the parent subproblem.

Computational results for problem sets one through five and eight through ten are presented in Tables 3.3 through 3.22 and 3.23 through 3.26. For each of the problems, we report the CPU time, number of iterations, number of subproblems solved, and the optimal value. Computational results for problem sets six and seven are presented later in this section.

Several factors affect the performance of the experimental branch and bound codes relative to OSL. Although CPU time is important, we will also consider the number of subproblems solved and the number of iterations per subproblem.

Figure 3.3 shows the number of subproblems solved by each of the experimental branch and bound codes relative to the number of subproblems solved by OSL. Generally, the experimental codes solved slightly more subproblems than OSL. From these results it does not appear that the early branching heuristics were causing the code to branch early on subproblems that had integer solutions or were infeasible. In general, OSL has better methods of choosing branching variables and selecting subproblems to solve. In some cases, it appears that the implicit enumeration techniques used by OSL were able to substantially decrease the number of subproblems solved. In other cases, the implicit enumeration techniques do not appear to have been helpful. Furthermore, relatively small differences between the three experimental branch and bound codes sometimes led to very large differences in the number of subproblems solved.

Figure 3.4 shows the number of iterations per subproblem used by the experimental codes on problem sets one through five and eight through ten. With the exception of problem set nine, the experimental code with early branching and warm start used between five and eight iterations per subproblem on each of the problem sets. The code without early branching and the code without warm start consistently used more iterations per subproblem. Since early branching and the warm start procedure tend to decrease the number of iterations per subproblem without increasing the number of subproblems solved, these results indicate that the early branching heuristics and warm start procedure are very effective in reducing the total number of iterations needed to solve these problems.

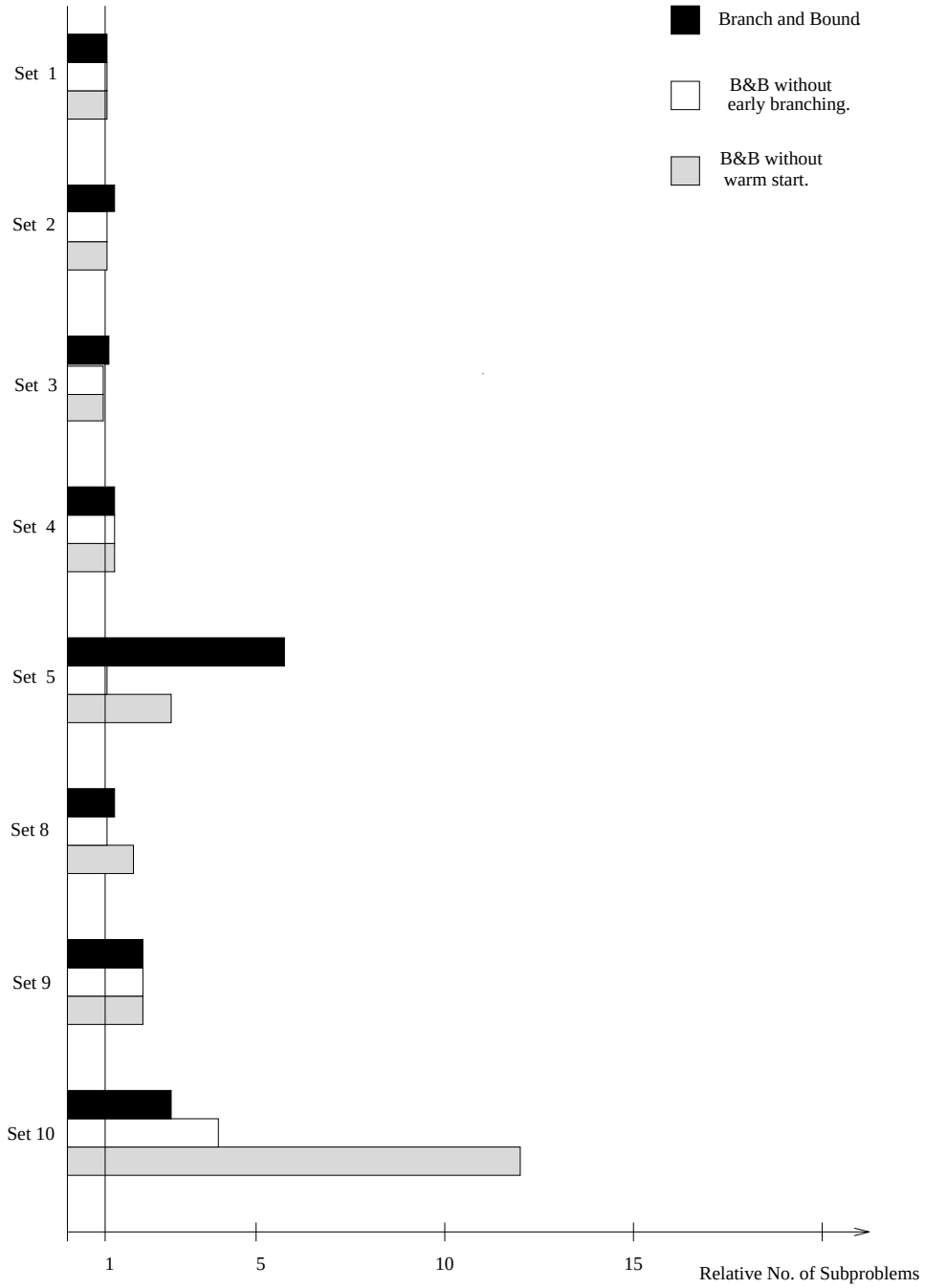

Figure 3.3: Subproblems solved relative to OSL.

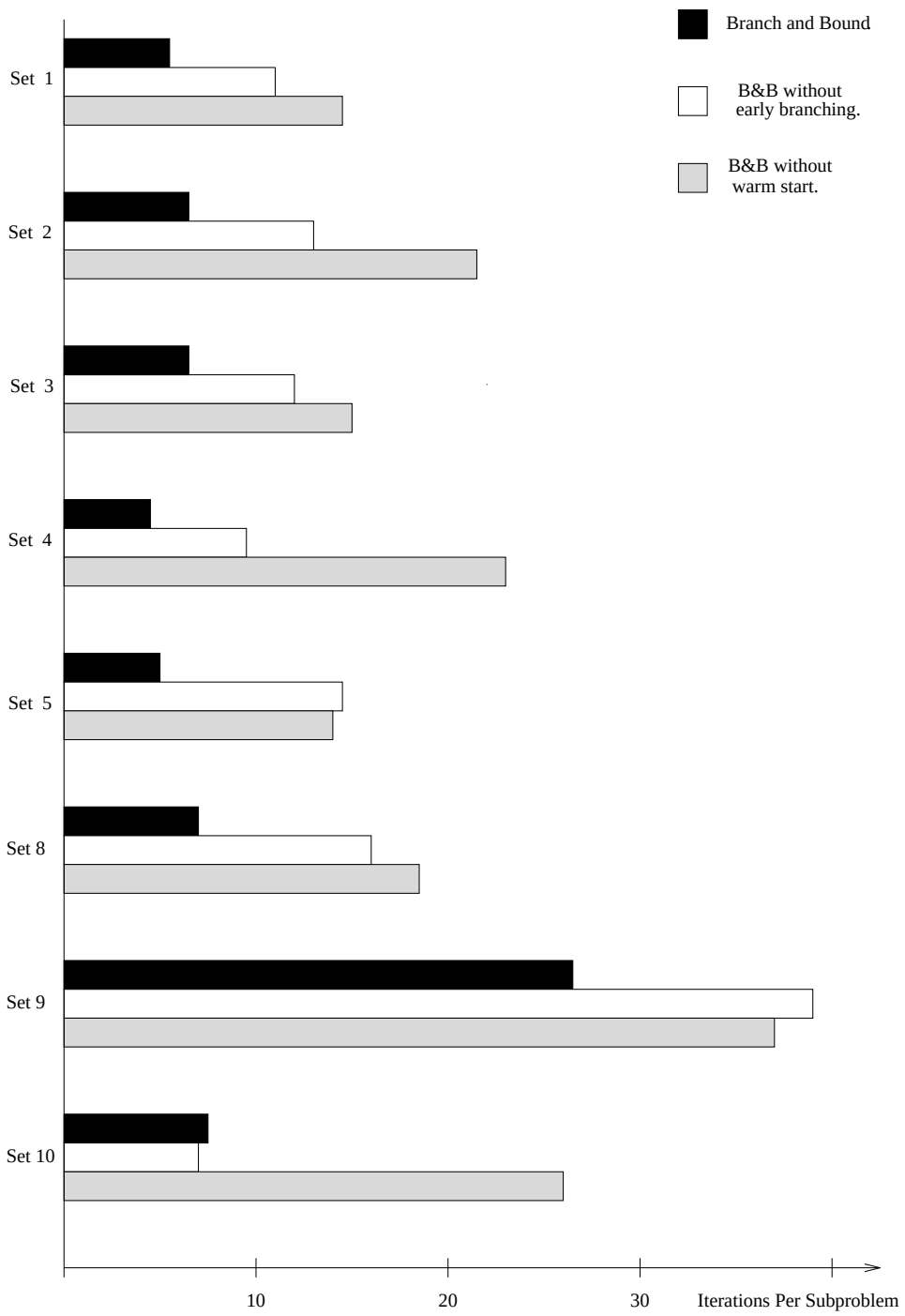

Figure 3.4: Iterations per subproblem.

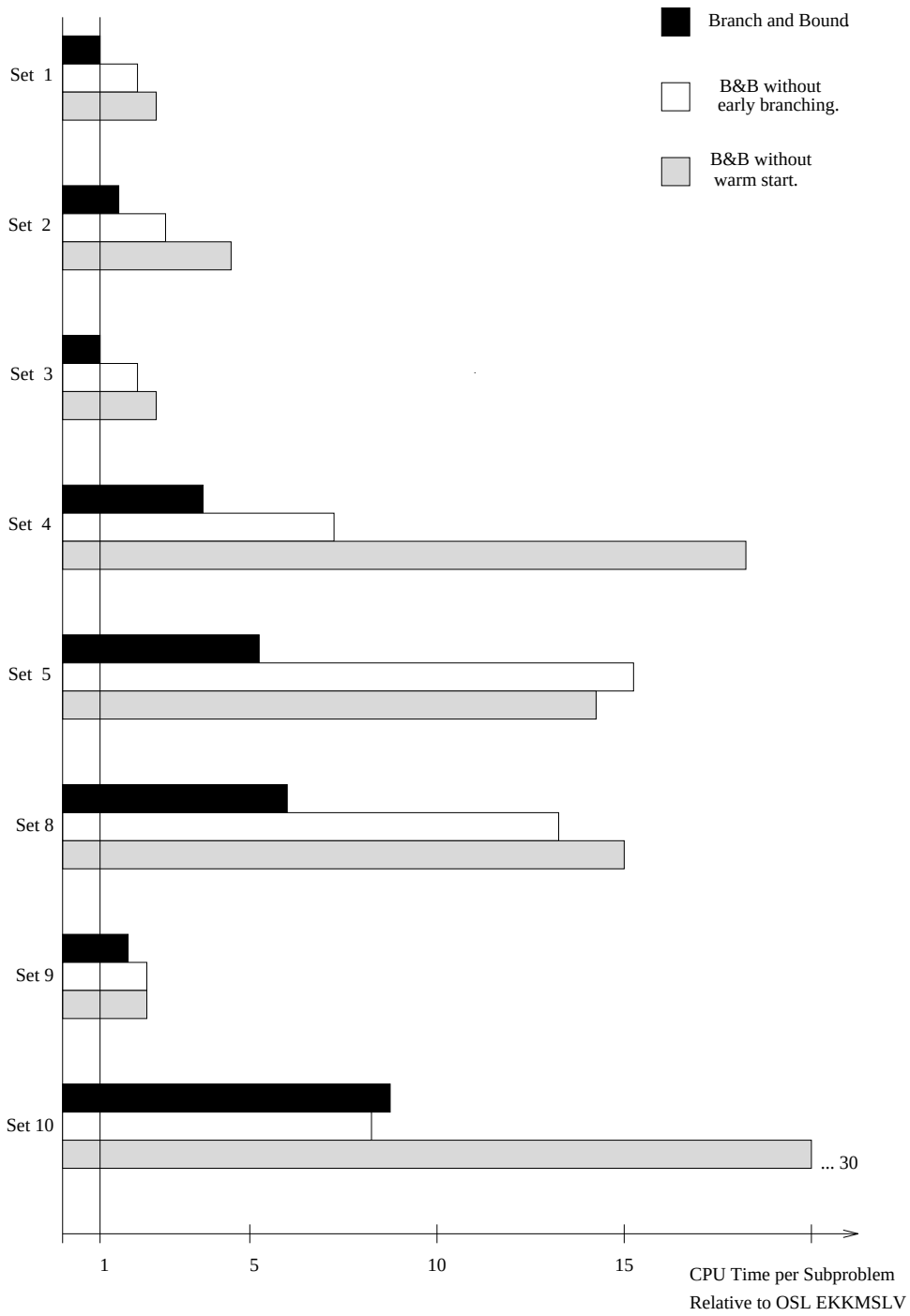

Figure 3.5: CPU time per subproblem, relative to OSL.

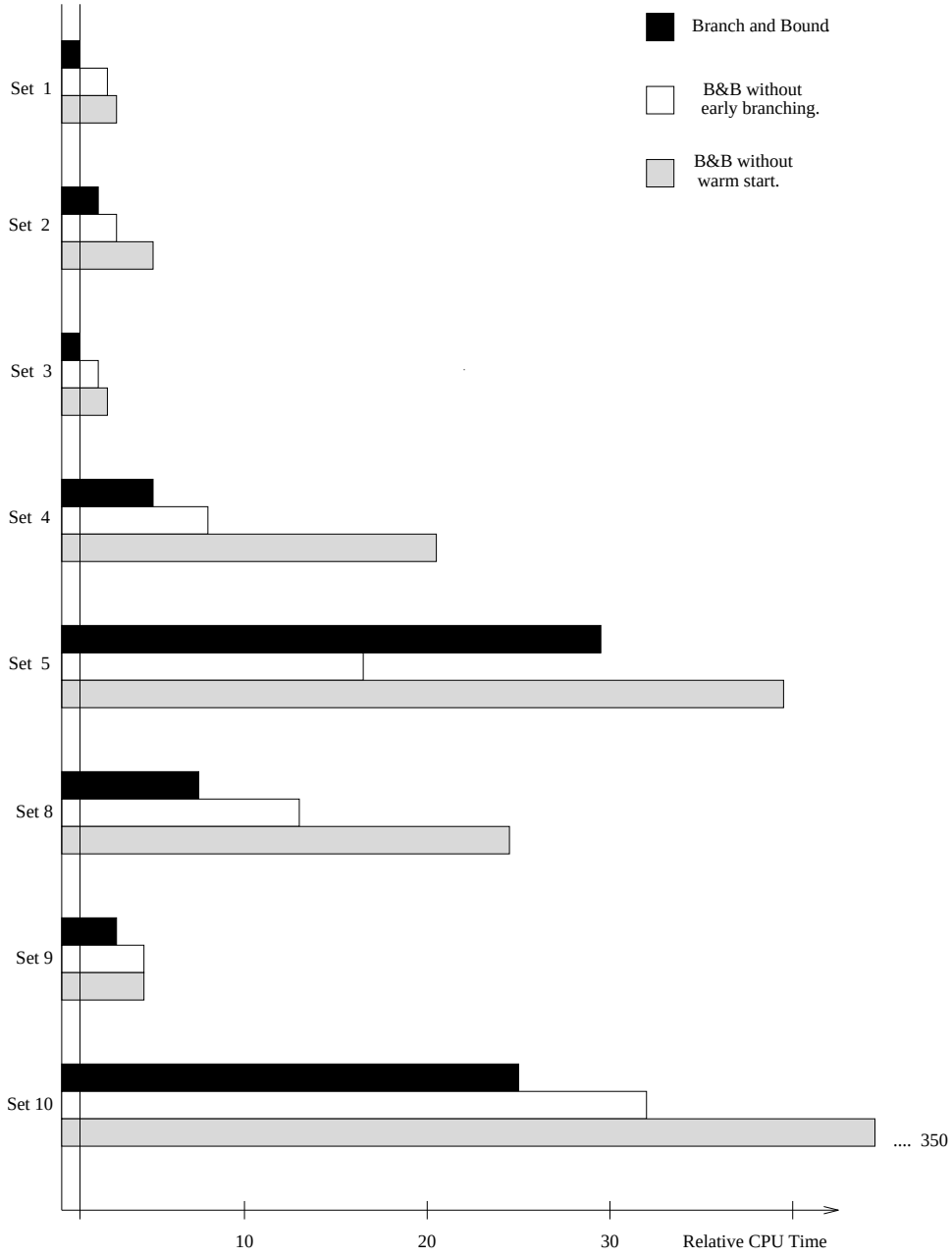

Figure 3.6: CPU times relative to OSL.

Since the experimental codes are solving only slightly more subproblems than OSL, the performance of the interior point method on the subproblems is the key issue. Although early branching and the warm start procedure are very helpful, the CPU time per subproblem is still higher for the experimental code. Figure 3.5 illustrates this. On problem sets one, two, three, and nine, the experimental code with early branching and warm start is comparable to EKKMSLV in CPU time per subproblem. On problem sets four, five, eight, and ten, OSL clearly dominates. Note that on problem ten, the experimental code without warm start used about 30 times more CPU time per subproblem than OSL. On this problem, the procedure for finding an initial primal–dual solution did not seem to work well.

The number of subproblems solved and the CPU time per subproblem combine to determine the performance of the experimental branch and bound codes. Figure 3.6 shows the total CPU time used by the three versions of the experimental code relative to the CPU time used by OSL on problem sets one through five and eight through ten. The experimental codes were consistently slower than OSL, particularly on problem sets five and ten.

The poor performance of the experimental branch and bound code on problem set four can largely be explained by the relatively small size of these problems. Because the problems were relatively small, OSL's simplex implmentation was about three times faster in solving the LP relaxations of these problems than our experimental primal–dual code.

In problem set five, the very poor performance of the experimental code can be explained by a combination of two factors. First, these problems are relatively small and dense. As a result, the experimental primal–dual LP solver took about six times as much CPU time to solve the LP relaxations of the problems as OSL's simplex code. Also, the experimental branch and bound code solved about six times as many subproblems as OSL's branch and bound code. Since the version of the

experimental code without early branching solved about as many subproblems as OSL, it appears that the experimental branch and bound code made some very poor choices in its selection of branching variables.

In problem eight, the experimental branch and bound codes solved roughly the same number of subproblems as OSL. However, OSL's simplex method was about six times faster than the experimental primal–dual code in solving the LP relaxation of this problem. This was probably because problem eight is relatively small and dense.

In problem nine, the branch and bound codes were run only until an integer solution had been found, not until an optimal solution had been found. Since the EKKMSLV routine uses rounding and implicit enumeration, it was able to find an integer solution after only 25 subproblems, while the experimental codes all required about 50 subproblems to find an integer solution. However, the solutions found by the three versions of the experimental code were all better than the integer solution found by OSL.

In problem ten, the implicit enumeration and rounding features of OSL were extraordinarily successful. EKKMSLV was able to solve this problem with only 228 subproblems, even though the problem has 112 zero–one variables. Furthermore, the routine used fewer simplex iterations to solve the mixed integer programming problem than EKKSSLV used to solve the LP relaxation of the problem. As in problem set five, the number of subproblems solved by the experimental branch and bound codes varied. This time, the code with early branching and warm start had the best performance of the three experimental codes. The code without the warm start procedure performed very poorly on this problem, using about 350 times as much CPU time as OSL. This can be explained by the very large number of subproblems solved, and the large number of iterations needed to solve each subproblem.

For most of these problems, OSL used about twenty times as many iterations to solve the LP relaxation as it did to solve an average subproblem in the branch and bound phase. This shows that the OSL code makes very good use of warm starts. In contrast, the experimental branch and bound code used three to five times as many iterations to solve the LP relaxations as it used to solve an average subproblem. However, the work that goes into a simplex iteration is not constant. In the branch and bound routine EKKMSLV, OSL executes simplex iterations at a considerably slower rate than in the LP solver EKKSSLV. Table 3.2 shows this effect. In contrast, the experimental codes perform primal–dual iterations at the same rate in the LP solver and in the branch and bound codes. In general, OSL slows down by a factor of about four in EKKMSLV. This might be caused by the need to refactorize the working basis for each new LP subproblem. The implicit enumeration techniques used by EKKMSLV might also play a part in this. Problem sets six and seven are an exception to the rule. On these problems, OSL took many iterations to solve the initial LP relaxation and then a very few iterations to solve the remaining 20 subproblems in the branch and bound tree. As a result, the count of iterations per second for EKKMSLV is based almost entirely on the iterations used to solve the initial LP relaxation.

Since the problems in problem sets six and seven have so many variables, it was not always possible to solve these problems to optimality. Instead, we set up both OSL and the experimental branch and bound code to solve 20 subproblems and report the best lower bound that had been achieved after solving the 20 subproblems. For these runs, it makes little sense to present statistics on the number of subproblems solved. Furthermore, since the solution of the initial LP relaxation can take a significant portion of the total solution time, statistics on the average number of iterations per subproblem and CPU time per subproblem are not particularly useful.

Problem Set	EKKSSLV Iterations/Second	EKKMSLV Iterations/Second
1	530	90
2	161	47
3	196	49
4	769	175
5	248	94
6	68	63
7	33	37
8	431	80
9	126	40
10	204	60

Table 3.2: Simplex iterations per second in EKKSSLV and EKKMSLV.

Computational results for problem sets six and seven are presented in Tables 3.27 through 3.30. For each of these runs we give the CPU time, number of iterations, best lower bound, and the optimal solution. Note that OSL found an optimal solution to problem 6-4. In general, the lower bounds obtained by OSL were slightly better than the lower bounds obtained by the experimental code.

In order to get good lower bounds for these problems, we modified the experimental code to wait until the duality gap was below 0.01% before searching for fractional variables. This gave better lower bounds, at the expense of more iterations per subproblem. OSL uses rounding heuristics to attempt to find an integer solution at each node of the branch and bound tree. These integer solutions provide upper bounds that can be used to fathom subproblems. The experimental branch and bound code was also modified with rounding heuristics that attempted to find integer solutions. However, these heuristics were not very successful at finding good integer solutions.

One interesting feature of the problems in problem sets six and seven is that all of the cost coefficients are drawn from a relatively small set of integers. As a result, many columns have equal cost, and this leads to LP subproblems which have many

optimal solutions. By its nature, the simplex method will pick out one basic feasible solution as the optimal solution of an LP. As noted in section 3.2, Güler and Ye have shown in [39] that the primal–dual method will converge to an optimal solution in the interior of the optimal face. Thus there will be more fractional variables in the solution found by the primal–dual method than in the solution found by the simplex method.

We conjectured that this effect would hurt the performance of the experimental branch and bound code. To test this, we randomly perturbed the cost coefficients of the problems in problem set six. For each cost coefficient, a random number uniformly distributed between -0.25 and 0.25 was added to the cost coefficient. We solved the perturbed problems to optimality using OSL. Although OSL was able to solve all of the problems, it is unlikely that the experimental codes could solve all of these problems to optimality since the experimental codes do not have OSL's rounding heuristics. These results are presented in Tables 3.31 and 3.32.

On problems four and five, OSL was able to find optimal solutions without solving more than 20 subproblems. The experimental code found the optimal solution to problem four. It appears that perturbing the objective function made these problems easier for both codes to solve. The lower bounds obtained by the experimental code are very close to the lower bounds obtained by OSL. On problems two, three, and four, the two codes found bounds that were roughly equal. On problem one, the experimental code found a better lower bound, while OSL found a better lower bound on problem five. Thus it appears that the two codes were equally effective in finding good lower bounds for these perturbed problems.

Problem	CPU Time	Iterations	Subproblems	Objective
1	86.7	7663	146	10080.44
2	78.1	6983	114	9996.212
3	150.2	13061	237	10014.74
4	45.3	4079	65	10202.27
5	52.1	4572	90	9925.541

Table 3.3: OSL results for problem set one.

Problem	CPU Time	Iterations	Subproblems	Objective
1	115.9	947	167	10080.44
2	80.6	658	119	9996.214
3	163.5	1329	261	10014.73
4	48.4	394	69	10202.27
5	69.2	563	97	9925.540

Table 3.4: Branch and bound results for problem set one.

pProblem	CPU Time	Iterations	Subproblems	Objective
1	210.7	1761	155	10080.44
2	161.6	1355	122	9996.214
3	352.5	2946	275	10014.73
4	99.3	832	73	10202.27
5	139.3	1167	103	9925.540

Table 3.5: Problem set one without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	263.7	2270	155	10080.44
2	188.6	1641	112	9996.214
3	439.6	3774	275	10014.73
4	124.8	1072	70	10202.27
5	182.4	1566	103	9925.540

Table 3.6: Problem set one without warm start.

Problem	CPU Time	Iterations	Subproblems	Objective
1	611.4	27729	549	10024.90
2	273.2	12510	233	10319.08
3	361.4	16626	271	10414.04
4	368.0	16613	348	10512.00
5	413.2	18402	411	10187.25

Table 3.7: OSL results for problem set two.

Problem	CPU Time	Iterations	Subproblems	Objective
1	1168.1	4719	728	10024.91
2	574.6	2323	342	10319.08
3	316.8	1292	200	10414.04
4	957.6	3918	580	10512.00
5	965.5	3944	567	10187.25

Table 3.8: Branch and bound results for problem set two.

Problem	CPU Time	Iterations	Subproblems	Objective
1	1838.9	7735	601	10024.91
2	505.6	2129	165	10319.08
3	608.2	2564	209	10414.04
4	1233.0	5186	407	10512.00
5	1958.3	8235	630	10187.25

Table 3.9: Problem set two without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	3069.6	13223	601	10024.91
2	842.7	3637	165	10319.08
3	913.4	3929	209	10414.04
4	1985.8	8573	409	10512.00
5	3214.8	13847	634	10187.25

Table 3.10: Problem set two without warm start.

Problem	CPU Time	Iterations	Subproblems	Objective
1	267.6	12901	163	19709.22
2	133.0	6674	69	18784.34
3	326.4	15440	234	19518.01
4	371.6	17643	254	19542.94
5	454.8	21518	314	18933.19

Table 3.11: OSL results for problem set three.

Problem	CPU Time	Iterations	Subproblems	Objective
1	342.7	1367	219	19709.23
2	97.8	389	61	18784.34
3	318.1	1268	199	19518.01
4	388.7	1554	247	19542.94
5	557.9	2230	355	18933.19

Table 3.12: Branch and bound results for problem set three.

Problem	CPU Time	Iterations	Subproblems	Objective
1	613.4	2551	207	19709.23
2	165.3	685	55	18784.34
3	567.0	2366	198	19518.01
4	634.4	2669	220	19542.94
5	936.0	3939	326	18933.19

Table 3.13: Problem set three without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	730.7	3124	207	19709.23
2	197.6	844	55	18784.34
3	684.1	2922	195	19518.01
4	770.9	3298	218	19542.94
5	1133.8	4856	322	18933.19

Table 3.14: Problem set three without warm start.

Problem	CPU Time	Iterations	Subproblems	Objective
1	2.5	343	77	1040444
2	2.1	357	55	1098000
3	1.8	375	42	1153000
4	1.7	391	37	1235500

Table 3.15: OSL results for problem set four.

Problem	CPU Time	Iterations	Subproblems	Objective
1	13.6	426	99	1040444
2	10.4	332	68	1098000
3	9.3	298	58	1153000
4	7.7	246	54	1235500

Table 3.16: Branch and bound results for problem set four.

Problem	CPU Time	Iterations	Subproblems	Objective
1	22.4	762	85	1040444
2	18.5	632	69	1098000
3	13.6	462	47	1153000
4	12.0	411	39	1235500

Table 3.17: Problem set four without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	65.1	2299	91	1040444
2	49.8	1763	73	1098000
3	27.6	971	47	1153000
4	23.0	808	39	1235500

Table 3.18: Problem set four without warm start.

Problem	CPU Time	Iterations	Subproblems	Objective
1	1.7	237	4	8693988
2	101.4	9442	670	8693905

Table 3.19: OSL results for problem set five.

Problem	CPU Time	Iterations	Subproblems	Objective
1	66.2	400	45	8693989
2	2959.6	18707	3695	8693905

Table 3.20: Branch and bound results for problem set five.

Problem	CPU Time	Iterations	Subproblems	Objective
1	59.7	349	19	8693988
2	1638.1	10058	704	8693905

Table 3.21: Problem set five without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
1	103.5	632	45	8693989
2	3974.0	25308	1819	8693905

Table 3.22: Problem set five without warm start.

Problem	CPU Time	Iterations	Subproblems	Objective
8	384.0	30782	7482	106940226
9	207.9	8266	26	-48163071
10	35.4	2175	228	12850.86

Table 3.23: OSL results for problem sets eight through ten.

Problem	CPU Time	Iterations	Subproblems	Objective
8	2909.1	65995	9444	106940226
9	664.8	1298	49	-53046481
10	877.2	4725	644	12850.86

Table 3.24: Branch and bound results for problem sets eight through ten.

Problem	CPU Time	Iterations	Subproblems	Objective
8	5102.3	120127	7495	106940226
9	887.0	1752	49	-53046482
10	1141.0	6357	899	12850.86

Table 3.25: Problem set eight through ten without early branching.

Problem	CPU Time	Iterations	Subproblems	Objective
8	9334.8	225175	12184	106940226
9	923.2	1837	50	-50798305
10	12535.1	71900	2759	12850.86

Table 3.26: Problems eight through ten without warm start.

Problem	CPU Time	Iterations	Best Lower Bound	Optimal Value
1	79.2	5235	250.9	253.0
2	75.3	4784	249.1	252.0
3	78.0	4893	229.4	232.0
4	73.0	4472	234.0	234.0
5	73.3	4574	235.9	236.0

Table 3.27: OSL results for problem set six.

Problem	CPU Time	Iterations	Best Lower Bound	Optimal Value
1	206.4	324	247.6	253.0
2	207.3	322	247.9	252.0
3	201.3	317	229.4	232.0
4	177.3	278	233.1	234.0
5	190.8	298	235.8	236.0

Table 3.28: Branch and bound results for problem set six.

Problem	CPU Time	Iterations	Best Lower Bound	Optimal Value
1	216.1	8106	225.6	227.0
2	207.4	7373	215.7	219.0
3	191.7	7105	236.0	243.0
4	206.0	7647	218.0	219.0
5	209.7	7852	214.0	215.0

Table 3.29: OSL results for problem set seven.

Problem	CPU Time	Iterations	Best Lower Bound	Optimal Value
1	462.7	340	224.5	227.0
2	490.5	363	214.3	219.0
3	470.9	351	236.0	243.0
4	455.2	341	216.7	219.0
5	485.5	363	212.7	215.0

Table 3.30: Branch and bound results for problem set seven.

Problem	CPU Time	Iterations	Best Lower Bound	Optimal
1	211.7	330	249.2	252.5
2	201.2	314	245.6	247.9
3	198.4	313	229.9	232.3
4	139.0	221	229.5	229.5
5	182.8	291	230.1	230.5

Table 3.31: Branch and bound results for perturbed problem set six.

Problem	CPU Time	Iterations	Best Lower Bound	Optimal
1	78.5	4660	248.3	252.5
2	81.2	4923	245.6	247.9
3	72.7	4585	229.9	232.3
4	68.0	4197	229.5	229.5
5	59.0	3726	230.5	230.5

Table 3.32: OSL results for perturbed problem set six.

CHAPTER 4
Solving Mixed Integer Nonlinear Programming Problems

In this chapter, we consider zero–one mixed integer nonlinear programming problems of the form:

$$
\begin{array}{lrll}
(MINLP) & \min & f(\mathbf{x},\mathbf{y}) & \\
& \text{subject to} & \mathbf{g}(\mathbf{x},\mathbf{y}) & \leq \quad 0 \\
& & \mathbf{x} & \in \quad \{0,1\}^m \\
& & \mathbf{y} & \leq \quad \mathbf{u} \\
& & \mathbf{y} & \geq \quad \mathbf{l}
\end{array}
$$

Here $\mathbf{x}$ is a vector of m zero–one variables, $\mathbf{y}$ is a vector of n continuous variables, and $\mathbf{u}$ and $\mathbf{l}$ are vectors of upper and lower bounds for the continuous variables $\mathbf{y}$. The objective function f and the constraint functions $\mathbf{g}$ are assumed to be convex.

In this chapter, we describe a branch and bound algorithm which uses the sequential quadratic programming method to solve the subproblems, and which uses heuristics to determine when to split a subproblem into two new subproblems. In section 4.1, we describe our approach to solving the NLP subproblems. Heuristics for detecting fractional zero–one variables are discussed in section 4.2. A lower bounding procedure based on Lagrangean duality is discussed in section 4.3. In section 4.4 we present our branch and bound algorithm. Computational results are given in section 4.5.

4.1 Solving the Subproblems

At each stage of the branch and bound algorithm, we must solve a nonlinear programming problem of the form:

$$\begin{array}{lrcl} (NLP) & \min & f(\mathbf{x},\mathbf{y}) & \\ & \text{subject to} & \mathbf{g}(\mathbf{x},\mathbf{y}) & \leq & 0 \\ & & \mathbf{x} & \leq & \mathbf{e} \\ & & \mathbf{x} & \geq & 0 \\ & & \mathbf{y} & \leq & \mathbf{u} \\ & & \mathbf{y} & \geq & \mathbf{l} \end{array}$$

where some of the zero–one variables in $\mathbf{x}$ may have been fixed at zero or one. Since this is a convex problem, we know that if it is feasible, any local minimum is also a global minimum. Furthermore, the Lagrangean for this problem is defined as:

$$L(\mathbf{x},\mathbf{y},\lambda) = f(\mathbf{x},\mathbf{y}) + \lambda^T \mathbf{g}(\mathbf{x},\mathbf{y})$$

where $\mathbf{x}$ and $\mathbf{y}$ are still subject to upper and lower bounds, and the Lagrange multipliers λ are restricted to nonnegative values. Since the problem is convex, the Lagrangean has a stationary point at $\mathbf{x}^*, \mathbf{y}^*, \lambda^*$, where $\mathbf{x}^*, \mathbf{y}^*$ is an optimal solution to (NLP), and λ^* is a set of optimal Lagrange multipliers, and λ_i is non-zero only if $\mathbf{g}(\mathbf{x}^*, \mathbf{y}^*)_i = 0$.

The method of sequential quadratic programming, described in [24, 33], attempts to find this stationary point and solve (NLP) by solving a sequence of quadratic programs with linear constraints. These quadratic subproblems are of the form:

$$\begin{array}{lrcl} (QP) & \min & \nabla f^T \mathbf{p} + \frac{1}{2}\mathbf{p}^T(W)\mathbf{p} & \\ & \text{subject to} & A\mathbf{p} & = & -\mathbf{c} \end{array}$$

Here W is the Hessian of the Lagrangean with respect to the variables $\mathbf{x}$ and $\mathbf{y}$, the rows of A are the gradients of the active constraints, and $\mathbf{c}$ is the vector of values of the active constraints.

It can be shown that an optimal solution to (QP) gives a direction $\mathbf{p}$ which is identical to the direction (in the $\mathbf{x}$ and $\mathbf{y}$ variables) given by Newton's method applied to the problem of finding a stationary point of the Lagrangean. Furthermore, the Lagrange multipliers for (QP) are the Lagrange multipliers that would be obtained by an iteration of Newton's method applied to the problem of finding a stationary point of the Lagrangean. Thus one iteration of the SQP method is equivalent to an iteration of Newton's method searching for the stationary point of the Lagrangean.

Our code uses a routine called E04VCF from the NAG library [71] to solve (NLP). This routine uses a version of the SQP method in which the solution to (QP) is used as a search direction. The routine picks a step size that minimizes an augmented Lagrangean "merit function."

The E04VCF routine calls user supplied subroutines that compute the objective function, constraint functions, and their gradients. The E04VCF subroutine will execute until it reaches a limit on the number of iterations, it finds an optimal solution, or it encounters an error condition. If the routine has reached the limit on iterations, the user can restart the routine from where it left off.

4.2 Heuristics for Detecting Fractional Solutions

Since the sequential quadratic programming algorithm generates estimates of the variables $\mathbf{x}$, we can examine them after each iteration to see if a variable is converging to a fractional value. This leads to heuristics for determining early in the solution of the subproblem that the optimal solution will have fractional zero–one variables.

Our experimental code examines the values of the zero–one variables after every second iteration of the sequential quadratic programming algorithm. (The E04VCF routine won't stop after just one iteration. It must run for at least two

iterations at a time.) If the current solution is not feasible with respect to both the linear and non-linear constraints, then no determination is made. If the current solution is feasible, the value of x_i is between 0.001 and 0.999, the difference between the previous value of x_i and the new value of x_i is less than a small tolerance ϵ, and the new value of x_i is closer to the old value of x_i than it is to either 0 or 1, then the code declares x_i fractional. The current version of the code uses a tolerance of 0.1, which was chosen after experimenting with several different values of ϵ.

The experimental code uses a very simple rule for selecting the branching variable. Of the variables which are considered fractional, the experimental code chooses the most fractional variable (that is, the variable which has value closest to 0.5) to branch on. In choosing the next subproblem to work on, the experimental code picks the subproblem with the smallest estimated optimal value. This estimate is simply the objective value of the parent subproblem if it was solved to optimality, or the objective value at the last iteration if the parent subproblem was not solved to optimality.

4.3 Generating Lower Bounds

In the branch and bound algorithm, when an early decision to break the current subproblem into two new subproblems is made, a lower bound on the values of the two new subproblems is required. This bound can be used later to fathom the subproblems if a better integer solution is found.

Given a set of Lagrange multipliers λ, we can use Lagrangian duality to find a lower bound for the optimal value of the current subproblem. This lower bound

is given by solving the nonlinear programming problem:

$$
\begin{aligned}
(DUAL) \qquad \min \quad & f(\mathbf{x},\mathbf{y}) + \lambda^T \mathbf{g}(\mathbf{x},\mathbf{y}) \\
\text{subject to} \quad & \mathbf{x} \leq \mathbf{e} \\
& \mathbf{x} \geq 0 \\
& \mathbf{y} \leq \mathbf{u} \\
& \mathbf{y} \geq \mathbf{l}
\end{aligned}
$$

where again, some of the zero–one variables may be fixed at zero or one.

This is a nonlinear programming problem with simple bound constraints that can easily be solved by a quasi-Newton method. Furthermore, if the multipliers λ are close to optimality, the minimum value of (DUAL) should be close to the optimal value of (NLP). However, this lower bounding procedure is likely to fail to give an improved lower bound if the Lagrange multipliers are not well chosen. For this reason, our algorithm doesn't attempt to use this lower bounding procedure until the Lagrange multipliers have had time to converge to their optimal values. If the lower bounding scheme fails, our algorithm will continue to work on the current subproblem until lower bounding succeeds or until an optimal solution has been found.

Our experimental code uses the NAG routine E04KBF to solve (DUAL). The initial guess is simply the current estimated solution $\mathbf{x}$ and $\mathbf{y}$ to the current subproblem. The Lagrangean function and its gradient are calculated automatically from the user supplied objective and constraint functions.

4.4 The Branch and Bound Algorithm

A flowchart of our branch and bound algorithm for solving mixed integer nonlinear programs is given in figure 4.1. This algorithm is identical in structure to the algorithms described in Chapter 2 and Chapter 3. However, there are several significant differences in the details of the algorithm.

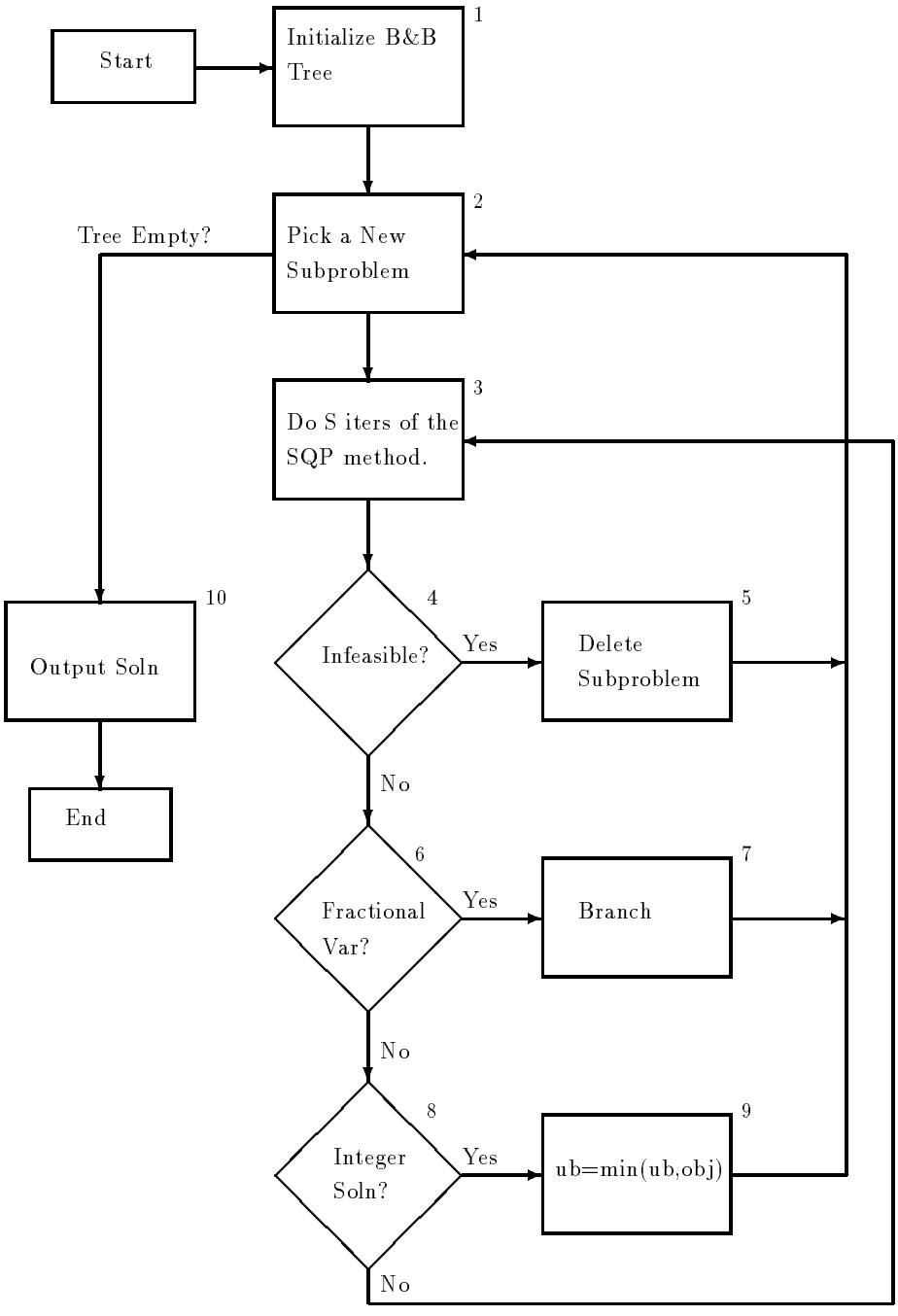

Figure 4.1: The branch and bound algorithm for MINLP.

In step two, we use a depth first strategy to choose the next subproblem until an integer solution has been found. From then on, we pick the subproblem with the smallest estimated lower bound. These estimates are based on the best known solution to the parent subproblem.

In step three, the algorithm performs S steps of the SQP algorithm. S is a parameter that can be adjusted. If we set S to a very large value, then the algorithm will solve each subproblem to optimality within S steps or detect infeasibility, but it will never use the heuristics to detect a fractional variable. This is a conventional branch and bound algorithm for (MINLP). If S is set to a smaller value, the heuristics will come into play. Because of a restriction in the E04VCF subroutine, we were only able to use values of S that were greater than one. Our experimental code uses the smallest possible value $S = 2$.

In step four of our algorithm, we have to determine whether the current subproblem is infeasible. The E04VCF routine will automatically detect infeasibility with respect to the linear constraints. However, if a subproblem is infeasible with respect to one or more of its nonlinear constraints, the E04VCF routine does not automatically detect the condition. Instead, the routine will return with an indication that it stopped after it could not find an improved point. This could be because the problem was infeasible, or it could be that the SQP routine was simply having trouble finding a feasible point. In this situation, we examine the current solution. If the current solution is close to feasibility (within 10%), then we restart the SQP algorithm in hopes of eventually finding a feasible solution. If the SQP algorithm fails a second time, we declare the subproblem infeasible.

Our lower bounding procedure is computationally expensive, and since it is usually not effective until the current solution is close to optimality, we do not perform the lower bounding procedure in step four.

In step six, we must first check that the current solution is feasible. If the current solution is within 1% of feasibility with respect to each of the linear and non-linear constraints, then we assume that the current subproblem is feasible. We can then check the values of the x_i to determine whether any variables are converging to fractional values.

Once we have decided to branch in step seven, it is important to get good lower bounds for the two new subproblems. We may already have a lower bound for the optimal objective value of the current subproblem. If the objective value of the current solution is within 2% of the current lower bound, then there is little point in trying to compute a better lower bound. In this case, the two new subproblems simply inherit the lower bound from the parent subproblem. If no lower bound for the current subproblem is available or the lower bound is not good enough, then we must use the procedure of section 4.3 to compute a lower bound for the current subproblem. This lower bound is then inherited by the two new subproblems.

As with the algorithms for mixed integer linear programming, this algorithm could err in considering a variable fractional when it is not. It could also make the mistake of declaring a subproblem feasible and a variable fractional when in fact the subproblem is infeasible. However, this process is limited by the finite size of the tree. Eventually, all of the integer variables would be fixed at zero or one, and the SQP method would correctly determine the feasibility or infeasibility of the subproblem. In practice, we have not had any significant problems with these early branching heuristics.

The experimental code makes use of three user supplied subroutines. The first routine, SETUPP, is used to initialize the continuous relaxation of the problem. SETUPP returns the total number of variables, number of integer variables, a matrix for any linear constraints, and feasibility tolerances for the linear and nonlinear constraints. A second routine, OBJFUN, calculates the objective function and its

gradient. The third routine, CONFUN, calculates the non-linear constraint functions and their gradients. These routines are then linked with the experimental code to create a program for solving the actual problem.

4.5 Computational Results

We tested the experimental code on a number of problems taken from the literature. Sample problems one through four were taken from a paper by Duran and Grossman [20]. The first three of these problems come from the chemical engineering problem of designing a chemical processing system. The fourth problem comes from the area of product marketing [29]. The third and fourth problems also appear in [49]. Our fifth sample problem is example problem five from a paper by Floudas, Aggarwal, and Ciric [25]. (Other problems in this paper were non-convex.) Our sixth sample problem is example four from a paper by Kocis and Grossman [52]. This problem is a convex version of a chemical process design problem that was non-convex in its original formulation. The seventh problem is a topological network design problem which is described in detail in Appendix B. A summary of the problems is presented in Table 4.1.

Problem	Zero–One Variables	Continuous Variables	Linear Constraints	Nonlinear Constraints
1	3	3	4	2
2	5	6	11	3
3	8	9	19	4
4	25	5	5	25
5	4	3	4	5
6	24	22	72	1
7	10	100	41	0

Table 4.1: Characteristics of the sample problems.

Two versions of the experimental code were developed. In the first version, the heuristics and lower bounding procedure are used after every other iteration of the SQP method. In the second version, the heuristics are not used. These codes were run under AIX on an IBM 3090–200S. The total number of SQP iterations, total number of subproblems solved, and CPU times were recorded for each code on each problem. The CPU times were measured with the CPUTIME subroutine of the Fortran run-time library. Unfortunately, our experience has been that these CPU times are only accurate to about 5%, since repeated runs of the code show slight but consistent variations in the CPU time. This could be caused by an inaccuracy in the timing routines or by cache effects. These computational results are presented in Tables 4.2 and 4.3.

Problem	SQP Iterations	Subproblems Solved	CPU Time (Seconds)	Optimal Value
1	21	5	0.1	6.00976
2	41	9	0.3	73.0353
3	75	15	1.0	68.0097
4	461	81	28.6	8.06487
5	25	7	0.2	4.57958
6	123	17	8.5	285507
7	2862	145	84.4	15.6086

Table 4.2: Computational results for the code without heuristics.

Note that the optimal objective values are given to six significant digits. The tolerances in E04VCF were set so that no constraint would be violated by more than 1.0×10^{-6} and so that the relative error in the optimal objective function would be smaller than 1.0×10^{-8}. Except for problem four, the optimal objective values found by the two experimental codes are identical to six significant digits. In problem four, the difference is in the sixth digit. In several cases, there are slight differences between the two codes in how many subproblems were solved. This

Problem	SQP Iterations	Subproblems Solved	CPU Time (Seconds)	Optimal Value
1	21	5	0.1	6.00976
2	37	9	0.3	73.0353
3	70	15	1.1	68.0097
4	461	81	28.7	8.06486
5	25	7	0.2	4.57958
6	90	15	6.6	285507
7	2238	137	67.8	15.6086

Table 4.3: Computational results for the code with heuristics.

can be explained by slightly different choices of the branching variables that were sometimes made by the two codes.

The code with heuristics worked about as well as the code without heuristics on the smaller problems. This is not surprising, since on these problems, the SQP method needed an average of 3 to 4 iterations to solve a subproblem, and there was little opportunity for the heuristics to come into play. On the more difficult problems 6 and 7, the SQP algorithm required more iterations to solve each subproblem, and the heuristics could be used successfully. For example, on problem 7, the code without heuristics used an average of 20 iterations on each subproblem that it solved, while the code with heuristics used an average of only 16 iterations per subproblem.

The paper by Paules and Floudas [49] includes computational results for problems three and four which can be compared to our results. These problems were solved by both generalized Bender's decomposition (GBD) and an outer approximation algorithm on an IBM–3090 computer similar to the computer used in our study. The optimal objective values and solutions reported by our codes agree with the solutions reported by Paules and Floudas with some differences in the fifth digit. However, Paules and Floudas only report their results to five digits, and it is not clear what tolerances were used in their code. On problem three, GBD required

16.35 CPU seconds while the outer approximation algorithm required 7.45 CPU seconds. In contrast, our branch and bound codes needed only about 1 CPU second to solve the problem. On problem four, GBD required over 800 CPU seconds to solve the problem. The outer approximation algorithm required between 16.09 and 25.63 CPU seconds depending on the starting point. Both of our codes solved the problem in about 29 CPU seconds.

CHAPTER 5
Discussion and Conclusions

In this chapter, we assess the results of the computational experiments described in Chapters 2, 3, and 4. The questions to be answered include:

- Is early branching effective in reducing the number of iterations per subproblem in the algorithms for mixed integer programs?
- Does early branching lead to an increase in the number of subproblems solved in these algorithms?
- How effective is fixed order branching in a branch and bound algorithm for mixed integer linear programs?
- Can branch and bound codes for mixed integer linear programming based on an interior point method compete with similar codes based on the simplex method?
- Is early branching worthwhile in the branch and bound code for mixed integer nonlinear programs?

We will also discuss directions for future research and ways in which the performance of the experimental codes could be improved.

The computational results in Chapters 2 and 3 clearly show that early branching substantially reduces the number of iterations per subproblem in the experimental branch and bound codes for mixed integer linear programs. Without early branching, a branch and bound code based on an interior point method for linear programming probably would not be competitive with a simplex based code. The computational results in Chapter 4 are less convincing. On most of the sample

problems, the average number of iterations per subproblem was very small (less than five). The early branching heuristics had little opportunity to detect fractional variables as the code solved these subproblems. On problems six and seven, the average number of iterations per subproblem was higher, and the early branching heuristics were able to function. However, the early branching heuristics were only able to reduce the number of iterations per subproblem by about 20% on these problems. Thus the early branching heuristics were not as effective in the code for mixed integer nonlinear programming problems as they were in the codes for linear problems.

The computational results in Chapters 2 and 3 show that early branching does not generally lead to an increase in the number of subproblems solved by the experimental branch and bound codes for mixed integer linear programs. In many cases, the experimental branch and bound code with early branching solved fewer subproblems than the experimental code without early branching. However, the poor performance of the experimental code on the problems in problem set five is a cause for concern. Furthermore, OSL consistently solved fewer subproblems than the experimental branch and bound codes. Improvements in the heuristics for detecting fractional zero–one variables, the heuristics for choosing branching variables, and the implementation of implicit enumeration techniques could alleviate this problem. The computational results in Chapter 4 also show that early branching does not lead to an increase in the number of subproblems solved by the branch and bound code for mixed integer nonlinear programming problems.

The computational results for the fixed order branch and bound code in Chapter 2 show that on our capacitated facility location problems, the fixed order branching strategy did reduce the number of subproblems solved and the total CPU time used by the experimental branch and bound code. However, the fixed order branch and bound code was never more than 20% faster than the branch and bound code

with conventional branching. Furthermore, the fixed order branching code used far more virtual storage than the conventional branch and bound code. Thus fixed order branching was not judged to be very successful in our computational testing.

The computational results in Chapters 2 and 3 show that our experimental branch and bound codes for mixed integer linear programs are quite capable of solving large zero–one mixed integer linear programs. The optimal solutions found by our codes consistently match the solutions found by OSL. However, our codes were not as fast as OSL in solving these problems. On problem sets one, two, and three, where our interior point codes could solve the LP relaxations in roughly the same time it took OSL's simplex method to solve the linear programs, and where the number of subproblems solved by the experimental branch and bound codes was similar to the number of subproblems solved by EKKMSLV, our branch and bound codes were competitive with OSL's branch and bound code. On the other problem sets, where our interior point codes could not solve the linear programs as quickly, or where EKKMSLV solved fewer subproblems than the experimental branch and bound codes, OSL's branch and bound code was clearly superior. Our experimental branch and bound code was about three times slower than OSL on the problems in sets six and seven. However, OSL's primal–dual predictor–corrector code solves the LP relaxations of these problems about three times faster than our primal–dual code. Thus it seems likely that with a better implementation of the primal–dual method, our experimental branch and bound code could compete with OSL on the problems in problem sets six and seven.

It seems that with substantial improvements in the implementation of the primal–dual method, improvements in the heuristics for detecting fractional variables, and the implementation of implicit enumeration techniques, the experimental branch and bound code could be improved to the point that it would compete with

OSL. However, such a code would only be competitive on problems where the interior point method was faster than the simplex method at solving the LP relaxations. Such problems would be relatively large, since the simplex method is dominant for smaller problems.

The computational results in Chapter 4 show that on some problems, early branching can reduce the number of SQP iterations per subproblem without increasing the total number of subproblems solved. However, this only happened in two of the seven test problems, and the number of iterations per subproblem was only reduced by about 20%. This gives us some reason to hope that early branching might be effective in larger problems. Further computational testing is needed to determine whether early branching is generally effective.

There are several areas in which our research can be extended:

- Which interior point method works best in a branch and bound code for MILP problems?
- Can we find a better warm start procedure for any of the interior point methods?
- Can we improve on the heuristics used to detect fractional zero–one variables?
- The experimental branch and bound code for nonlinear problems should be tested on larger sample problems.
- What other methods for solving the NLP subproblems can be used? This could effect the efficiency of the heuristics for early detection of fractional variables.

The major contribution of this thesis is the early branching idea. The computational results in Chapters 2, 3, and 4 provide evidence for the effectiveness of early branching. A second contribution of this thesis is the computational testing of

interior point methods in a branch and bound algorithm for zero–one mixed integer linear programming problems. Although the experimental branch and bound code was not competitive with OSL, there is substantial room for improvement in the performance of the experimental code.

LITERATURE CITED

[1] I. Adler, M. G. C. Resende, G. Veiga, and N. K. Karmarkar. An implementation of Karmkarkar's algorithm for linear programming. *Mathematical Programming*, 44(3):297–335, 1989.

[2] Umit Akinc and Basheer M. Khumawala. An efficient branch and bound algorithm for the capacitated warehouse location problem. *Management Science*, 23(6):585–594, February 1977.

[3] E. Balas and A. Ho. Set covering algorithms using cutting planes, heuristics, and subgradient optimisation: A computational study. *Mathematical Programming Study*, 12:37–60, 1980.

[4] E. R. Barnes. A variation of Karmarkar's algorithm for solving linear programming problems. *Mathematical Programming*, 36:174–182, 1986.

[5] E. Beale and R. Small. Mixed integer programming by a branch and bound technique. In *Proceedings of the Third IFIP Congress*, pages 450–451, 1965.

[6] J. E. Beasley. An algorithm for set covering problems. *European Journal of Operational Research*, 31:85–93, 1987.

[7] J. E. Beasley. OR-library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[8] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and D. Vincent. Experiments in mixed integer linear programming. *Mathematical Programming*, 1:76–94, 1971.

[9] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice-Hall, 1987.

[10] R. E. Bixby, J. W. Gregory, I. J. Lustig, R. E. Marsten, and D. F. Shanno. Very large–scale linear programming : A case study in combining interior point and simplex methods. Technical Report J–91–07, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA, May 1991. Technical Report SOR 91–08, School of Engineering and Applied Science, Dept. of Civil Engineering and Operations Research, Princeton University, Princeton, NJ 08544, USA, May 1991. Technical Report RRR 34–91, RUTCOR Center of Operations Research, Rutgers University, New Brunswick, NJ 08903, USA, 1991.

[11] Robert E. Bixby, E. Andrew Boyd, and Ronni R. Indovina. MIPLIB: A test set of mixed integer programming problems. *SIAM News*, 25(20):16, March 1992.

[12] Robert R. Boorstyn and Howard Frank. Large-scale network topological optimization. *IEEE Transactions on Communications*, 25:29–47, 1977.

[13] W. Carolan, J. Hill, J. Kennington, S. Niemi, and S. Wichmann. An empirical evaluation of the KORBX algorithms for military airlift applications. *Operations Research*, 38:240–248, 1990.

[14] In Chan Choi, Clyde L. Monma, and David F. Shanno. Further developments of a primal-dual interior point method. *ORSA Journal on Computing*, 2(4):304–311, 1990.

[15] R. Dakin. A tree search algorithm for mixed integer programming problems. *Computer Journal*, 8:250–255, 1965.

[16] I. I. Dikin. Iterative solution of problems of linear and quadratic programming. *Doklady Akademii Nauk SSSR*, 174:747–748, 1967. Translated in : *Soviet Mathematics Doklady*, 8:674–675, 1967.

[17] I. I. Dikin. On the convergence of an iterative process. *Upravlyaemye Sistemi*, 12:54–60, 1974. (In Russian).

[18] N. J. Driebeck. An algorithm for the solution of mixed integer programming problems. *Management Science*, 12:576–587, 1966.

[19] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[20] Marco A. Duran and Ignacio E. Grossmann. An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming*, 36:307–339, 1986.

[21] S. C. Eisenstat, M.C. Gurshy, M.H. Schultz, and A.H. Sherman. The Yale Sparse Matrix Package, I. The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18:1145–1151, 1982.

[22] A. S. El-Bakry, R. A. Tapia, and Y. Zhang. A study of indicators for identifying zero variables in interior-point methods. Technical Report TR91-15, Rice University, June 1991.

[23] Anthony V. Fiacco and Garth P. McCormick. *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*. SIAM, Philadelphia, 1990.

[24] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, second edition, 1987.

[25] C. A. Floudas, A. Aggarwal, and A. R. Ciric. Global optimum search for nonconvex NLP and MINLP problems. *Computers and Chemical Engineering*, 13(10):1117–1132, 1989.

[26] J. J. H. Forest, J. P. H. Hirst, and J. A. Tomlin. Practical solution of large mixed integer programming problems with Umpire. *Management Science*, 20(5):736–773, January 1974.

[27] J. J. H. Forrest and J. A. Tomlin. Implementing interior point linear programming methods in the optimization subroutine library. *IBM Systems Journal*, 31(1):26–38, 1992.

[28] J. J. H. Forrest and J. A. Tomlin. Implementing the simplex method for the optimization subroutine library. *IBM Systems Journal*, 31(1):11–25, 1992.

[29] B. Gavish, D. Horsky, and K. Srikanth. An approach to the optimal positioning of a new product. *Management Science*, 29(11):1277–1297, 1983.

[30] A. M. Geoffrion and R. E. Marsten. Integer programming algorithms: A framework and state-of-the-art survey. *Management Science*, 18:465–491, 1972.

[31] Alan George and Joseph W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[32] Mario Gerla and Leonard Kleinrock. On the topological design of distributed computer networks. *IEEE Transactions on Communications*, 25:48–60, 1977.

[33] Philip E. Gill, Walter Murray, and Margaret Wright. *Practical Optimization*. Academic Press, New York, 1981.

[34] D. Goldfarb and M. J. Todd. Karmarkar's projective scaling algorithm. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science*, pages 141–170. North Holland, Amsterdam, The Netherlands, 1989.

[35] Gene Golub and Charles Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, second edition, 1989.

[36] C. C. Gonzaga. Convergence of the large step primal affine–scaling algorithm for primal nondegenerate linear programs. Technical Report ES–230/90, Dept. of Systems Engineering and Computer Science, COPPE Federal University of Rio de Janeiro, 21941 Rio de Janeiro, RJ, Brazil, September 1990.

[37] Clovis C. Gonzaga. On lower bound updates in primal potential reduction methods for linear programming. *Mathematical Programming*, 52(2):415–428, 1991.

[38] I. E. Grossmann. Mixed-integer programming approach for the synthesis of integrated process flow-sheets. *Computers and Chemical Engineering*, 9:463–482, 1985.

[39] O. Güler and Yinyu Ye. Convergence behavior of some interior–point algorithms. Working Paper, 1991.

[40] Omprakash K. Gupta and A. Ravindran. Nonlinear integer programming algorithms: A survey. *OPSEARCH*, 20(4):189–206, 1983.

[41] Omprakash K. Gupta and A. Ravindran. Branch and bound experiments in convex nonlinear integer programming. *Management Science*, 31(12):1533–1546, 1985.

[42] Pierre Hansen. Methods of nonlinear 0-1 programming. In P. L. Hammer, E. J. Johnson, and B. H. Korte, editors, *Discrete Optimization II*, Annals of Discrete Mathematics. North Holland, New York, 1979.

[43] H. H. Hoang. Topological optimization of networks: A nonlinear mixed integer model employing generalized benders decomposition. *IEEE Transactions on Automatic Control*, 27:164–169, 1982.

[44] IBM, Kingston, NY. *Engineering and Scientific Subroutine Subroutine Library Guide and Reference*, November 1988.

[45] IBM, Kingston, NY. *IBM Optimization Subroutine Library Guide and Reference*, August 1990.

[46] IBM, Kingston, NY. *VAST-2 for VS Fortran: User's Guide*, April 1990.

[47] IBM, Kingston, NY. *VS Fortran Version 2, Language and Library Reference, Release 5*, 1990.

[48] Newspaper Enterprise Association Inc. *The World Almanac and Book of Facts*. Scripps Howard Company, 1990.

[49] Granville E. Paules IV and Christodoulos A. Floudas. APROS: Algorithmic development methodology for discrete-continuous optimization problems. *Operations Research*, 37(6):902–915, 1989.

[50] N. K. Karmarkar. A new polynomial–time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[51] N. K. Karmarkar and K. G. Ramakrishnan. Computational results of an interior point algorithm for large scale linear programming. *Mathematical Programming*, 52(2):555–586, 1991.

[52] Gary R. Kocis and Ignacio E. Grossman. Global optimization of nonconvex mixed-integer nonlinear programming (MINLP) problems in process synthesis. *Industrial & Engineering Chemistry Research*, 27:1407–1421, 1988.

[53] E. Kranich. Interior point methods for mathematical programming : A bibliography. Discussion Paper 171, Institute of Economy and Operations Research, FernUniversität Hagen, P.O. Box 940, D–5800 Hagen 1, West–Germany, May 1991. The (actual) bibliography can be accessed electronically by sending e–mail to '*netlib@research.att.com*' with message '*send index from bib*'.

[54] A. A. Kuehn and M. J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 9(9):643–666, July 1963.

[55] A. Land and A. Doig. An automatic method of solving discrete programming problems. *Econometrika*, 28(3):497–520, 1960.

[56] A. Land and S. Powell. Computer codes for problems of integer programming. In P. L. Hammer, E. J. Johnson, and B. H. Korte, editors, *Discrete Optimization II*, Annals of Discrete Mathematics. North Holland, New York, 1979.

[57] D. J. Laughhunn. Quadratic binary programming with applications to capital-budgeting problems. *Operations Research*, 18(3):454–461, 1970.

[58] E. L. Lawler and D. E. Wood. Branch and bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[59] I. J. Lustig, R. E. Marsten, and D. F. Shanno. On implementing Mehrotra's predictor–corrector interior point method for linear programming. Technical Report SOR 90–03, School of Engineering and Applied Science, Dept. of Civil Engineering and Operations Research, Princeton University, Princeton, NJ 08544, USA, April 1990. To appear in *SIAM Journal on Optimization.*

[60] I. J. Lustig, R. E. Marsten, and D. F. Shanno. Computational experience with a primal–dual interior point method for linear programming. *Linear Algebra and Its Applications*, 152:191–222, 1991.

[61] I. J. Lustig, R. E. Marsten, and D. F. Shanno. Interior method vs. simplex method : Beyond *netlib*. *COAL Newsletter*, 19:41–44, August 1991.

[62] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno. Starting and restarting the primal-dual interior point method. Technical Report SOR 90-14, Princeton University, October 1990.

[63] Irvin J. Lustig, Roy E. Marsten, and David F. Shanno. Computational experience with a primal–dual interior point method for linear programming. *Linear Algebra and Its Applications*, 152:191–222, 1991.

[64] T.L. Magnanti and R. T. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18(1):1–55, February 1984.

[65] J. C. T. Mao and B. A. Wallingford. An extension of Lawler and Bell's method of discrete optimization with examples from capital budgeting. *Management Science*, 15(2):51–60, 1968.

[66] R. E. Marsten, R. Subramaniam, M. J. Saltzman, I. J. Lustig, and D. F. Shanno. Interior point methods for linear programming: Just call Newton, Lagrange, and Fiacco and McCormick! *Interfaces*, 20(4):105–116, 1990.

[67] Roy E. Marsten, Matthew J. Saltzman, David F. Shanno, George S. Pierce, and J. F. Ballintijn. Implementation of a dual affine interior point algorithm for linear programming. *ORSA Journal on Computing*, 1(4):287–297, 1989.

[68] Kevin A. McShane, Clyde L. Monma, and David Shanno. An implementation of a primal-dual interior method for linear programming. *ORSA Journal on Computing*, 1(2):70–83, 1989.

[69] S. Mehrotra. On the implementation of a (primal–dual) interior point method. Technical Report 90–03, Dept. of Industrial Engineering and Management Science, Northwestern University, Evanston, IL 60208, USA, March 1990. Revised June 1990.

[70] Clyde L. Monma and Andrew J. Morton. Computational experience with a dual affine variant of Karmarkar's method for linear programming. *Operations Research Letters*, 6(6):261–267, December 1987.

[71] NAG. *FORTRAN Library Manual Mark 13: Volume 3*. Numerical Analysis Group, Oxford, 1988.

[72] R. Gary Parker and Ronald L. Rardin. *Discrete Optimization*. Academic Press, Boston, 1988.

[73] Mauricio G. C. Resende and Geraldo Veiga. A dual affine scaling algorithm for minimum cost network flow. Manuscript, 1990.

[74] Harvey M. Salkin and Kamlesh Mathur. *Foundations of Integer Programming*. North-Holland, New York, 1989.

[75] Jaya Singhal, Roy E. Marsten, and Thomas L. Morin. Fixed order branch and bound methods for mixed-integer programming: The ZOOM system. *ORSA Journal on Computing*, 1(1):44–51, 1989.

[76] R. A. Tapia and Yin Zhang. An optimal–basis identification technique for interior point linear programming algorithms. *Linear Algebra and Its Applications*, 152:343–363, 1991.

[77] M. J. Todd. Recent developments and new directions in linear programming. In M. Iri and K. Tanabe, editors, *Mathematical Programming : Recent*

Developments and Applications, pages 109–157. Kluwer Academic Press, Dordrecht, The Netherlands, 1989.

[78] Michael J. Todd. Improved bounds and containing ellipsoids in Karmarkar's linear programming algorithm. *Mathematics of Operations Research*, 13:650–659, 1988.

[79] Michael J. Todd and Bruce P. Burrell. An extension of Karmarkar's algorithm for linear programming using dual variables. *Algorithmica*, 1:409–424, 1986.

[80] J.A. Tomlin. An improved branch and bound method for integer programming. *Operations Research*, 19:1070–1075, 1971.

[81] T. Tsuchiya. Global convergence of the affine scaling methods for degenerate linear programming problems. *Mathematical Programming*, 52(2):377–404, 1991.

[82] T. Tsuchiya and M. Muramatsu. Global convergence of a long–step affine scaling algorithm for degenerate linear programming problems. Technical Report 423, The Institute of Statistical Mathematics, 1992.

[83] Kathryn Turner. Computing projections for the Karmarkar algorithm. *Linear Algebra and Its Applications*, 152:141–154, 1991.

[84] R. J. Vanderbei and J. C. Lagarias. I. I. Dikin's convergence result for the affine-scaling algorithm. In Jeffrey C. Lagarias and Michael J. Todd, editors, *Mathematical Developments Arising from Linear Programming*. American Mathematical Society, Providence, RI, 1991.

[85] R. J. Vanderbei, M.S. Meketon, and B. A. Freedman. A modification of Karmarkar's linear programming algorithm. *Algorithmica*, 1:395–407, 1986.

[86] H. M. Weingartner. Capital budgeting of interrelated projects. *Management Science*, 12(7):485–516, 1966.

[87] G. G. Wilson and B. D. Rudin. Introduction to the IBM optimization subroutine library. *IBM Systems Journal*, 31(1):4–10, 1992.

[88] M. H. Wright. Interior methods for constrained optimization. Numerical Analysis Manuscript 91–10, AT & T Bell Laboratories, Murray Hill, NJ 07974, USA, November 1991.

[89] Y. Ye. A build–down scheme for linear programming. *Mathematical Programming*, 46(1):61–72, 1990.

APPENDIX A
Benchmarks for the Primal–Dual Code

In this appendix, we describe a number of benchmarks which show the relative performance of our dual–affine interior point code, our primal–dual interior point code, the primal–dual interior point code in IBM's Optimization Subroutine Library, and the simplex code in OSL. The articles by Forrest and Tomlin provide descriptions of the OSL implementations of the simplex method and the primal–dual interior point method [27, 28].

For the problems in problem sets one through three, we used the version of our experimental primal–dual code that uses the YSMP routines. For the problems in problem sets four through seven, we used a version of the primal–dual code that uses the ESSL subroutines. We have found that the ESSL routines have better performance than the YSMP routines on relatively dense problems. For the problems in sets eight through ten, which are relatively sparse, we again used the YSMP routines.

The OSL codes and the experimental primal–dual code were tested on the LP relaxations of the test problems in problem sets one through ten. The dual–affine code was tested on the problems in problem sets one through four. The results are presented in tables A.1 through A.4. For each run, we give the number of iterations, CPU time, and the optimal objective value. Figure A.1 summarizes these results.

For the OSL simplex runs, the routine EKKSSLV was used. DEVEX pricing was turned off, because this gave faster solutions. It should be noted that the CPU time and number of simplex iterations needed to solve the linear programming problems was often different from the time used by the EKKMSLV routine to solve the initial LP relaxation in the branch and bound tree. For example, EKKSSLV required 8579 iterations and 251.2 CPU seconds to solve the LP relaxation of problem 7-1.

However, EKKMSLV was able to solve the LP relaxation and 19 other subproblems in 8106 iterations and 216.1 CPU seconds. This might be caused by the supernode processing feature of EKKMSLV (which uses implicit enumeration techniques to fix zero-one variables at zero or one.) Another factor might be differences in the simplex algorithm used by EKKMSLV.

The OSL codes both gave optimal values to seven digits. The experimental primal–dual code was also designed to give optimal values to seven digits, while the experimental dual–affine code was designed to give optimal values to roughly six digits. In most cases, the experimental primal–dual code and the two OSL codes agree on the optimal objective value. In the few cases where there are discrepancies, the differences are in the seventh digit. The experimental dual–affine code is somewhat less accurate, but most of the dual–affine results are good to six digits.

The OSL simplex code is faster than either of the primal–dual codes on problem sets one, two, four, five, and eight. However, the simplex code is much slower than the primal–dual codes on problem sets six and seven. The simplex code used a very large number of iterations on these problems, so it appears that the problems were degenerate.

Of the two primal–dual codes, the OSL code is generally faster than the experimental primal–dual code. This is particularly evident in problem sets six and seven, where the OSL code is approximately three times as fast as the experimental primal–dual code. The OSL primal–dual code is also about twice as fast as the experimental primal–dual code on problems eight and ten. The OSL primal–dual code makes use of a predictor–corrector scheme which has been shown to reduce the number of primal–dual iterations [59, 69]. On the problems in problem sets one through seven, OSL's predictor–corrector method used 20-50% fewer iterations than our code. The remaining difference in performance can be be attributed to better sparse matrix linear algebra routines in the OSL code.

An important factor in the performance of the sparse matrix linear algebra code is the structure of the matrix $A\Theta A^T$. Figures A.2 through A.10 show the pattern of non-zero elements in the matrices for three of the sample problems. For each problem, we show the matrix before minimum degree ordering, after minimum degree ordering, and after Cholesky factorization. In problem 1-1, the matrix $A\Theta A^T$ is quite sparse, and there is very little fill-in after minimum degree ordering and Cholesky factorization. In problem 5-1, the matrix is much more dense, but minimum degree ordering is still effective in reducing the fill-in during factorization. However, in problem 6-1, the original matrix $A\Theta A^T$ is quite dense, and even with minimum degree ordering, the fill-in is nearly complete after Cholesky factorization.

For problems such as 6-1, where the matrix is nearly completely dense, it makes more sense to use dense matrix techniques, especially for the lower right hand corner of the matrix. OSL's primal-dual implementation automatically switches over to a dense matrix factorization routine when the density of the matrix exceeds 70%. The ESSL and YSMP routines continue to use algorithms for sparse matrices even on very dense matrices, and as a result, performance suffers.

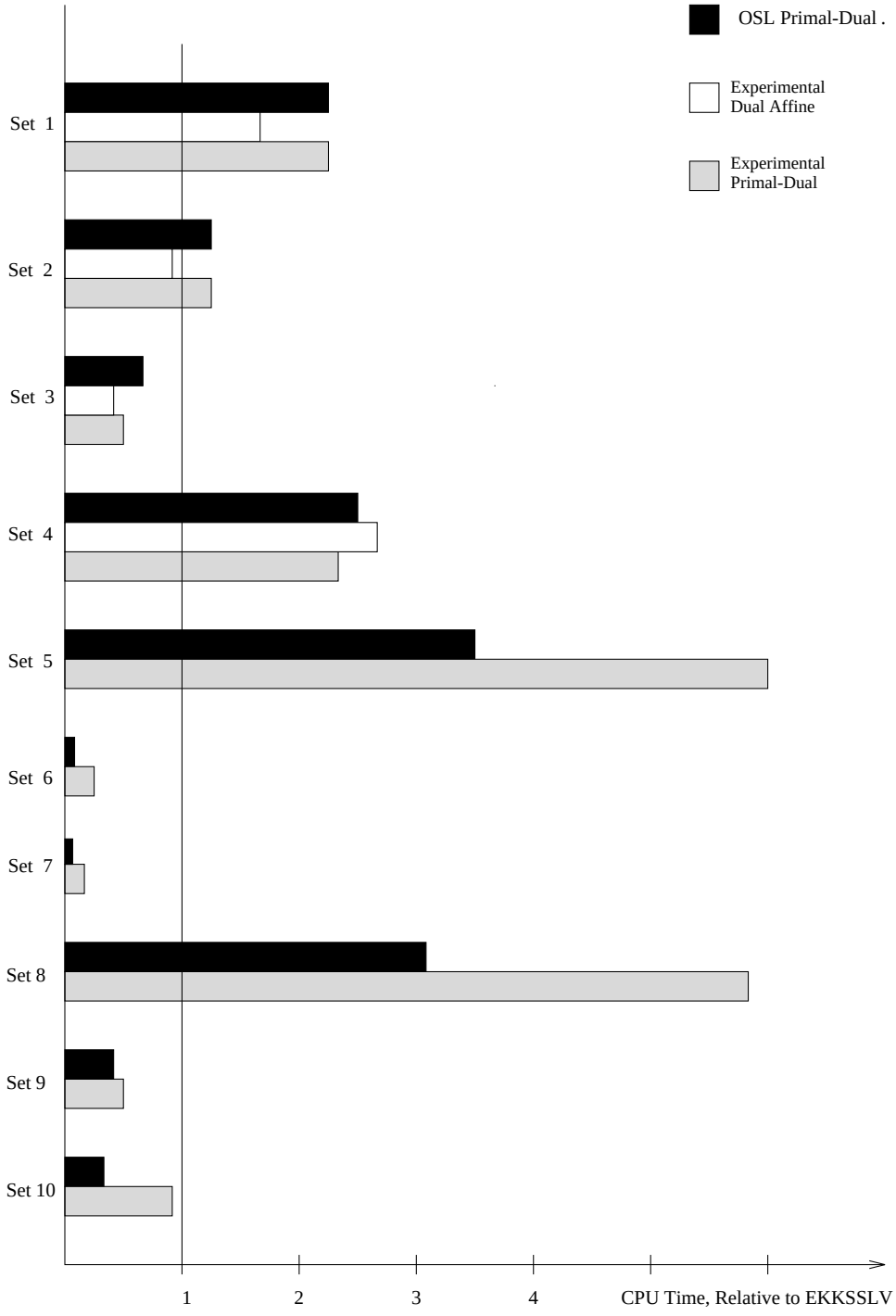

Figure A.1: LP results for problem sets one through ten.

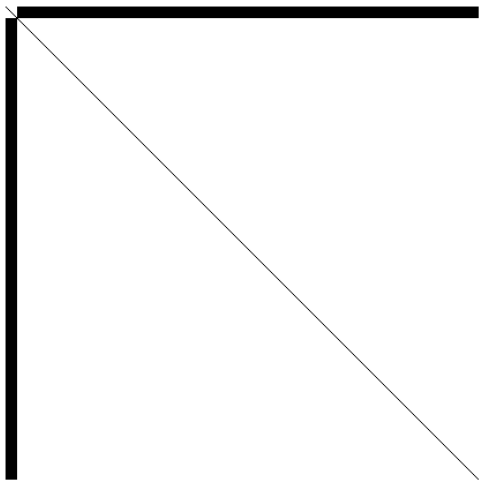

Figure A.2: Problem 1-1 before minimum degree ordering.

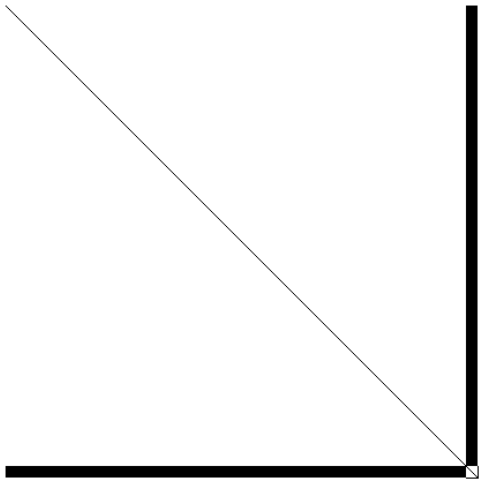

Figure A.3: Problem 1-1 after minimum degree ordering.

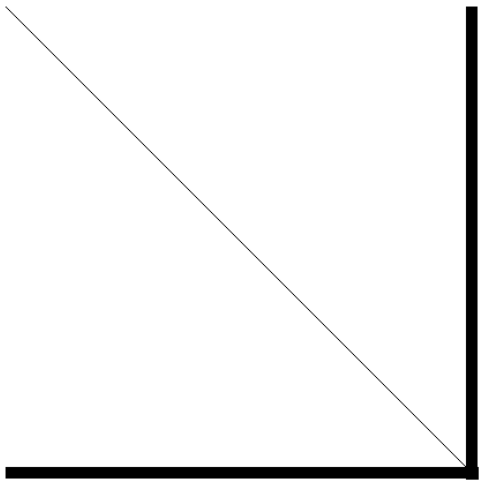

Figure A.4: The Cholesky factors for problem 1-1.

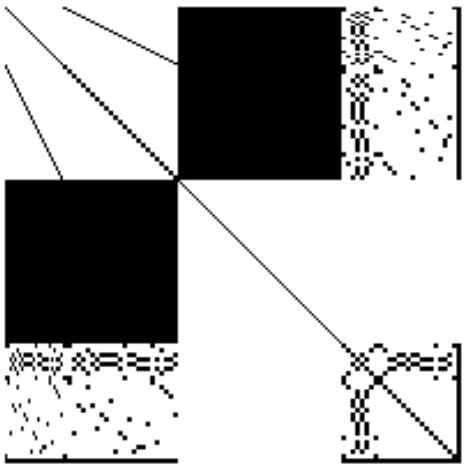

Figure A.5: Problem 5-1 before minimum degree ordering.

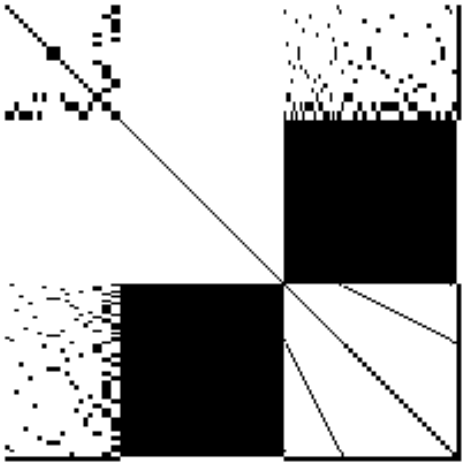

Figure A.6: Problem 5-1 after minimum degree ordering.

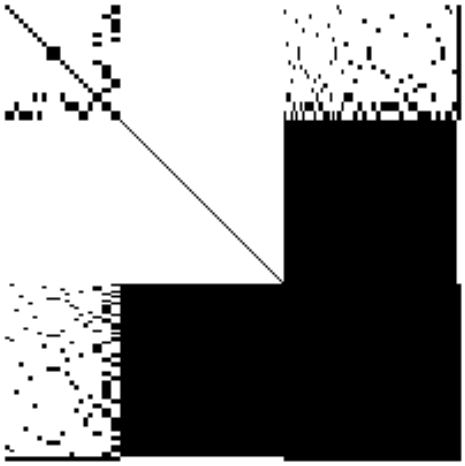

Figure A.7: The Cholesky factors for problem 5-1.

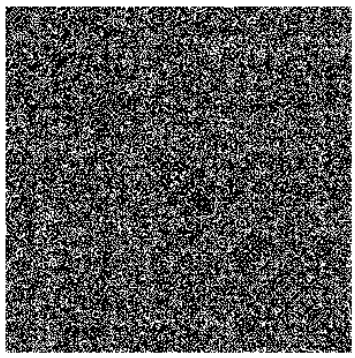

Figure A.8: Problem 6-1 before ordering.

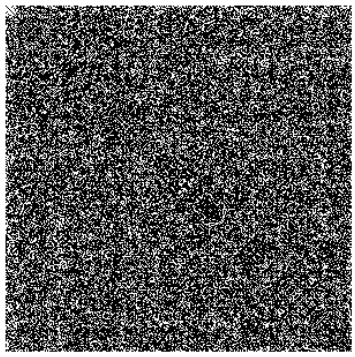

Figure A.9: Problem 6-1 after minimum degree ordering.

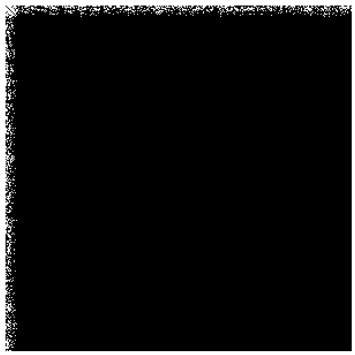

Figure A.10: The Cholesky factors for problem 6-1.

Problem	Iterations	CPU time	Objective
1-1	841	1.3	9930.746
1-2	862	1.5	9886.191
1-3	857	1.4	9865.063
1-4	874	1.5	10105.79
1-5	834	2.4	9802.896
2-1	991	5.7	9971.261
2-2	952	5.3	10286.82
2-3	954	6.8	10379.20
2-4	1042	7.0	10462.02
2-5	951	5.6	10138.69
3-1	1686	8.5	19459.09
3-2	1747	9.1	18641.96
3-3	1718	9.0	19252.27
3-4	1683	8.1	19259.10
3-5	1674	8.6	18638.69
4-1	146	0.2	1018152
4-2	152	0.2	1071420
4-3	154	0.2	1124688
4-4	163	0.2	1204590
5-1	155	0.6	8613441
5-2	142	0.6	8528637
6-1	4905	71.0	246.8368
6-2	6011	94.6	247.4964
6-3	5094	73.0	228.0000
6-4	5356	80.0	231.3968
6-5	5073	71.1	234.8889
7-1	8579	251.2	223.8010
7-2	8235	257.0	212.8475
7-3	8208	253.9	234.5829
7-4	8545	261.4	213.8483
7-5	7844	241.5	211.6365
8	112	0.26	95919464
9	6247	49.6	-62121983
10	2201	10.8	12841.69

Table A.1: OSL simplex solutions of the LP relaxations.

Problem	Iterations	CPU time	Objective
1-1	12	2.2	9930.746
1-2	15	2.7	9886.191
1-3	14	2.6	9865.063
1-4	16	2.9	10105.79
1-5	13	2.4	9802.896
2-1	23	8.0	9971.261
2-2	21	7.3	10286.82
2-3	23	8.0	10379.20
2-4	22	7.6	10462.02
2-5	23	7.9	10138.69
3-1	15	5.6	19459.09
3-2	19	6.9	18641.96
3-3	15	5.6	19252.27
3-4	13	4.9	19259.10
3-5	15	5.5	18638.69
4-1	14	0.5	1018152
4-2	14	0.5	1071420
4-3	13	0.5	1124688
4-4	13	0.5	1204590
5-1	13	1.8	8613441
5-2	20	2.4	8528637
6-1	19	7.2	246.8368
6-2	19	7.4	247.4964
6-3	18	7.0	228.0000
6-4	19	7.3	231.3968
6-5	17	6.8	234.8889
7-1	20	14.4	223.8010
7-2	20	14.2	212.8475
7-3	21	14.7	234.5829
7-4	19	13.8	219.8483
7-5	21	14.8	211.6365
8	14	0.8	95919464
9	27	21.5	-62121983
10	27	4.1	12841.69

Table A.2: OSL primal-dual solutions of the LP relaxations.

Problem	Iterations	CPU time	Objective
1-1	15	2.3	9930.746
1-2	19	2.7	9886.189
1-3	18	2.6	9865.065
1-4	20	2.9	10105.79
1-5	14	2.2	9802.896
2-1	29	7.4	9971.266
2-2	28	7.1	10286.83
2-3	30	7.6	10379.20
2-4	31	7.9	10462.01
2-5	33	8.3	10138.68
3-1	18	4.8	19459.09
3-2	20	5.3	18641.96
3-3	17	4.6	19252.27
3-4	18	4.8	19259.10
3-5	17	4.6	18638.69
4-1	19	0.5	1018152
4-2	19	0.5	1071420
4-3	19	0.5	1124688
4-4	20	0.6	1204590
5-1	18	3.0	8613441
5-2	25	4.2	8528637
6-1	30	19.0	246.8368
6-2	28	17.6	247.4964
6-3	24	15.1	228.0000
6-4	28	17.5	231.3968
6-5	26	16.2	234.8889
7-1	28	36.8	223.8010
7-2	29	38.1	212.8475
7-3	32	42.3	234.5829
7-4	25	33.0	213.8483
7-5	32	42.1	211.6365
8	22	1.5	95919464
9	47	25.5	-62121983
10	47	9.1	12841.69

Table A.3: Experimental primal-dual solutions of the LP relaxations.

Problem	Iterations	CPU time	Objective
1-1	22	1.7	9930.738
1-2	25	1.9	9886.189
1-3	27	2.0	9865.065
1-4	25	1.9	10105.79
1-5	22	1.7	9802.896
2-1	33	5.2	9971.266
2-2	34	5.4	10288.83
2-3	34	5.5	10379.20
2-4	37	5.9	10462.01
2-5	36	5.7	10138.68
3-1	24	3.7	19459.09
3-2	28	4.3	18641.96
3-3	24	3.7	19252.26
3-4	24	3.7	19259.10
3-5	23	3.6	18638.69
4-1	27	0.5	1018151
4-2	27	0.4	1071419
4-3	27	0.5	1124687
4-4	27	0.5	1204589

Table A.4: Experimental dual–affine solutions of the LP relaxations.

APPENDIX B
A Small Network Design Problem

The seventh example problem in chapter 4 is a small network design problem for a data communications network involving five cities (New York, Los Angeles, Chicago, Philadelphia, and Houston.) The distances between these cities and the populations of the cities were obtained from the 1990 World Almanac and Book of Facts [48]. The total flow between pairs of cities was made proportional to the product of the populations. These flows are given in Table 4. The cost of a link between two cities was made equal to the distance between the two cities in miles. These distances are given in Table 5.

City	New York	Los Angeles	Chicago	Houston	Philadelphia
New York	0	0.592	0.547	0.314	0.298
Los Angeles	0.592	0	0.245	0.141	0.134
Chicago	0.547	0.245	0	0.130	0.124
Houston	0.314	0.141	0.130	0	0.071
Philadelphia	0.298	0.134	0.124	0.071	0

Table B.1: Flows between cities.

City	New York	Los Angeles	Chicago	Houston	Philadelphia
New York	0	2786	802	1608	100
Los Angeles	2786	0	2054	1538	2706
Chicago	802	2054	0	1067	738
Houston	1608	1538	1067	0	1508
Philadelphia	100	2706	738	1508	0

Table B.2: Distances between cities.

The problem is to choose a set of links between cities and a routing of the data such that the total queueing delay is minimized subject to a budget constraint.

Problems of this type are discussed in [9, 12, 32]. To formulate this problem, we number the nodes from 1 to 5, and Let x_{ij} be 0 if the arc (i,j) is in the network and 1 if the arc is not in the network. We'll assume that the communications links are bidirectional, so we will always have $x_{ij} = x_{ji}$. Let f_{ij} be the flow between nodes i and j. Let f_{ij}^s be the flow on arc (i,j) of data destined for node s.

Each link will have a nominal capacity of 1.0 in each direction. However, since the queueing delay would be infinite if the flow was at full capacity, we'll introduce a capacity constraint of 90%. This can be written as:

$$\sum_{s=1}^{n} f_{ij}^s \leq 0.9 x_{ij}$$

There are 20 of these constraints, with one for each direction on each link.

Let F_i^s be the total flow of data from node i to node s for $i \neq s$. The network flow constraint at node i for data destined for node s can be written as:

$$\sum_{\text{arcs (i,j)}} f_{ij}^s - \sum_{\text{arcs (j,i)}} f_{ji}^s = F_i^s$$

For our five city problem, there are 20 such constraints.

Finally, we have a budget constraint:

$$\sum_{i<j} c_{ij} x_{ij} \leq 8000$$

Here c_{ij} is the mileage between cities i and j. This constraint limits the network to a total of 8000 miles of links. (Without it, the optimal solution would involve every possible link.)

The objective is to minimize the total queueing delay in the network. This delay can be approximated by the sum over all arcs of the queueing delay on an individual arc, which is given by $D(f_{ij}) = f_{ij}/(1 - f_{ij})$, where $f_{ij} = \sum_{s=1}^{n} f_{ij}^s$ is the total flow on arc (i, j), and 1 is the normalized capacity of the link. Unfortunately, this function has a singularity which could cause numerical problems. To avoid this,

we use the function:

$$D(f_{ij}) = \begin{cases} f_{ij}/(1 - f_{ij}) & \text{if } f_{ij} \leq 0.9 \\ 9 + 100(f_{ij} - 0.9) + 1000(f_{ij} - 0.9)^2 & \text{Otherwise} \end{cases}$$

This alternative function matches the original function for all feasible flows (where $f_{ij} \leq 0.9$), and is continuous and twice differentiable for all values of f_{ij}.

Thus the problem can be written as:

$$\begin{array}{lrcll} (NET) & \min & \sum_{\text{arcs (i,j)}} D(f_{ij}) & & \\ & \text{subject to} & \sum_{s=1}^{n} f_{ij}^s & \leq & 0.9x_{ij} \quad \text{for all arcs (i,j)} \\ & & \sum_{\text{arcs (i,j)}} f_{ij}^s - \sum_{\text{arcs (j,i)}} f_{ji}^s & = & F_i^s \quad \text{for all } i \neq s \\ & & \sum_{i<j} c_{ij}x_{ij} & \leq & 8000 \\ & & f_{ij}^s & \geq & 0 \quad \text{for all } i, j, s \\ & & x_{ij} & = & x_{ji} \quad \text{for all } i \neq j \\ & & x_{ij} & \in & \{0, 1\} \quad \text{for all } i \neq j \end{array}$$

In this form, the problem has 10 zero-one variables, 100 continuous variables, 41 linear constraints, and a nonlinear objective function. The optimal solution has a total queueing delay of 15.6086. It uses links between New York and all four of the other cities, a link between Houston and Chicago, and a link between Houston and Los Angeles. This solution uses 7901 miles of the 8000 mile budget.