CONSTRAINT OPTIMAL SELECTION TECHNIQUES (COSTs) FOR LINEAR PROGRAMMING

by

GOH SAITO

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2012

To all who brought me here.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervising professors Dr. H.W. Corley and Dr. Jay M. Rosenberger for providing support and guidance through weekly research meetings towards the dissertation work, while accommodating my work schedule. I would also like to thank the remaining members of the dissertation committee Dr. Victoria Chen and Dr. Erick Jones for their interest and taking the time to serve on the committee.

I am grateful for the fellow students of the COSMOS lab for allowing me to constantly run CPLEX on the workstation and Richard Zercher and Ann Hoang for IT assistance.

Finally, I would like to express my appreciation to my supervisors and coworkers at Alcon Research, Ltd. for their understanding and encouragement in my pursuit of the degree.

April 19, 2012

ABSTRACT

CONSTRAINT OPTIMAL SELECTION TECHNIQUES (COSTs)

FOR LINEAR PROGRAMMING

Goh Saito, Ph.D.

The University of Texas at Arlington, 2012

Supervising Professors: H.W. Corley, Jay M. Rosenberger

Simplex pivoting algorithms remain the dominant approach to solve linear programming (LP) because they have advantages over interior-point methods. However, current simplex algorithms are often inadequate for solving a large-scale LPs because of their insufficient computational speeds. This dissertation develops the significantly faster simplex-based, active-set approaches called Constraint Optimal Selection Techniques (COSTs). COSTs specify a constraint-ordering rule based on constraints' likelihood of being binding at optimality, as well as a rule for adding constraints. In particular, new techniques for adding multiple constraints in an active-set framework, and an efficient constraint-ordering rule for LP are proposed. These innovations greatly reduce computation time to solve LP problems.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Linear Programming Problem

Linear programming is a tool for optimizing numerous real world problems such as the allocation problem. Consider a general linear program (LP) as the following problem P

$$\text{maximize} \quad z = \mathbf{c}^{\mathrm{T}}\mathbf{x} \tag{1.1}$$

$$\text{subject to} \quad \mathbf{A}\mathbf{x} \leq \mathbf{b} \tag{1.2}$$

$$\mathbf{x} \geq \mathbf{0}, \tag{1.3}$$

where z represents an objective function for n variables

$$\mathbf{c}^{\mathrm{T}}\mathbf{x} = \begin{bmatrix} c_1 & c_2 & \cdots & c_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

and the expression (1.2) describes m rows of constraints for n variables

$$\begin{bmatrix} \mathbf{a}_1^{\mathrm{T}} \\ \mathbf{a}_2^{\mathrm{T}} \\ \vdots \\ \mathbf{a}_m^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \leq \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

A column vector of zeros of appropriate dimension according to context is written as $\mathbf{0}$. The dual of P is considered the standard minimization LP problem. We focus here on the maximization case.

Problems where $\mathbf{a}_i \geq \mathbf{0}$ and $\mathbf{a}_i \neq \mathbf{0}$, $\forall i = 1, \ldots, m$; $\mathbf{b} > \mathbf{0}$; and $\mathbf{c} > \mathbf{0}$ are called nonnegative linear programs (NNLPs). In general linear programs (LPs), the components of $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{c}$ are not restricted to be nonnegative numbers.

Though simplex pivoting algorithms and polynomial interior-point barrier-function methods represent the two principal solution approaches to solve problem P [1], there is no single best algorithm. For either method, we can always formulate an instance of P for which the method performs poorly [2]. Nevertheless, simplex methods remain the dominant approach because they have advantages over interior-point methods, such as efficient post-optimality analysis of pivoting algorithms, application of cutting-plane methods, and delayed column generation. However, current simplex algorithms are often inadequate for solving a large-scale LPs because of their insufficient computational speeds. In particular, emerging technologies require computer solutions in nearly real time for problems involving millions of constraints or variables. Hence faster techniques are needed. The COSTs of this dissertation represent a viable such approach.

1.2 Objectives of the Work

A unifying framework of algorithm termed Constraint Optimal Selection Technique (COST) was defined by Sung [3], and Corley and Rosenberger [4]. There are two classes of COSTs, Prior and Posterior. This dissertation focuses on Prior COSTs, which utilize global information obtained from P of (1.1)–(1.3) prior to solving relaxed LP subproblems of P. Specifically the focus is on improving and generalizing COSTs based on the RAD constraint selection metric involving Factor I and Factor II proposed by Sung [3]. The RAD for NNLPs is renamed NRAD in this dissertation.

1.3 Brief Description of COSTs

In COSTs presented in this dissertation, all constraints are initially ordered by a certain measure of their likelihood of being binding at optimality. Next, a bounded, relaxed subproblem P_0 of P is formed from (1.1), one or more sorted constraints of (1.2), a non-negativity requirement on variables (1.3), and possibly an artificial bounding constraint. For a subproblem P_r of a subsequent active-set iteration r, COSTs specify the particular constraints to be added from the ordered constraints. Active-set iterations continue until the solution to P_r is the optimal solution to P. In other words, this dissertation specifies an initial constraint-ordering rule as well as a rule for adding constraints.

1.4 Contributions

A new technique of adding multiple cuts is incorporated into the COST NRAD for NNLPs. The COST NRAD is then developed further to the COST GRAD for LPs. The key contributions of this dissertation are

(i) a bounding technique for the initial problem P_0 in which multiple constraints are added according to NRAD until each variable x_j is bounded (*multi-bound*),

(ii) a method of adding multiple cuts that bound every variable x_j in each active-set iteration (*multi-cut*), to enhance NRAD and efficiently solve the problem P,

(iii) development of a constraint selection metric involving Factor I and Factor II for general LP (*GRAD*) based on the insights gained from NRAD,

(iv) a technique for adding multiple constraints for each active-set iteration with general LP, extending the multi-bound and multi-cut techniques of (i) and (ii),

(v) creation of extensive sets of randomly generated large-scale NNLP and LP problems, and evaluation of COST methods on these problems.

1.5 Overview of the Dissertation

Chapter 2 presents a review of the relevant literature. A historical perspective, theoretical background, and the primal and dual simplex methods of linear programming are given. Chapter 3 describes COSTs and defines COST NRAD, COST GRAD, and the technique of adding multiple cuts. Chapter 4 gives the random NNLP and LP problems generated, experimental procedure, and results of computational evaluations. Finally, conclusions are stated in Chapter 5, and the Appendix gives example programming code.

CHAPTER 2

BACKGROUND

2.1 Introduction

The aspects of linear programming relevant to this research are summarized in this chapter.

2.2 Preliminaries

In P from Chapter 1, a *feasible point* is a point that satisfies all constraints (1.2). A set of all feasible points forms the *feasible region*. In (1.2), a set of points $\mathbf{x}$ that satisfies $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} \leq b_i$ defined by constraint i is a *half-space*. A *hyperplane* $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i$ divides a space into two half-spaces. The intersection of half-spaces defined by multiple constraints constitutes a *polyhedron*. A *polytope* is a bounded polyhedron. The problem P is *infeasible* if and only if the polyhedron defined by (1.2) is empty.

A set is *convex* if every line segment between two points is in the set. The intersection of convex sets is also convex. Since half-spaces are convex, the polyhedron defined by (1.2) is convex. A point in a convex set is said to be an *extreme point* if it does not lie in any open line segment between two points in the set. Extreme points cannot be expressed as a linear combination of other points in the set. Note that an extreme point of the polyhedron defined by (1.2) is a feasible point at an intersection of two or more linearly independent hyperplanes.

For P, a solution space is defined by k simultaneous equations $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i$, $\forall i = 1 \ldots k$ and n variables x_j, $\forall j = 1 \ldots n$ represented by the vector $\mathbf{x}$. Then $\mathbf{x}$ is a *basic solution* if $\mathbf{x}$ is obtained by setting $n - k$ variables equal to zero and solving k

equations. If $n = k$, P has only one solution but the majority of LP problems have $n > k$. If a basic solution also satisfies the non-negativity constraint (1.3), it is a *basic feasible solution.* A basic feasible solution is an extreme point of $\{\mathbf{Ax} \geq \mathbf{b},\, \mathbf{x} \geq \mathbf{0}\}$.

An optimal solution $\mathbf{x}^*$ of P is a basic feasible solution that maximizes the objective function (1.1). A constraint i is said to be *binding* at optimality if $\mathbf{x}^*$ satisfies $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i$.

2.3 Historical Perspective

A chronological list of key events on the development and research on linear programming is given below.

1717 — The idea of LP can be traced back to the concept of virtual velocity by Bernoulli.

1824 — Fourier generalized the idea of Bernoulli to the algebraic formulation utilizing inequalities and the geometric interpretation utilizing polyhedra. [5].

1910–1911 — de la Vallée Poussin presented an analogue of Fourier's method for solving LP [6].

1932 — A matrix structure was proposed by Leontief in the Interindustry Input-Output Model of the American Economy [7].

1936 — Motzkin published his dissertation on inequality systems [5].

1939 — A Soviet mathematician Kantorovich formulated LP, but his work was not known until the late 1950s [5].

1941 — Hitchcock authored a paper describing LP. The paper was not known to Dantzig [5].

1940s — There was an increase in interest in planning operation and optimization during wartime (1941-1945).

late 1940s — The development of computers was supported by the Pentagon.

1947 — G.B. Dantzig developed the simplex method and published his work "Programming in a Linear Structure" the following year [5].

1948 — A theorem known as the Fritz-John conditions is published. This theorem is principally related to nonlinear programming to develop later [5].

1950s — The field of mathematical programming emerged and network flow theory began to evolve.

1951 — Karush-Kuhn-Tucker conditions, which described necessary conditions for optimality were published. The field of non-linear programming began around this time.

1952 — Charnes, Cooper, and Mellon began utilizing a commercial application of LP in blending petroleum.

1954 — Frisch developed a nonlinear interior point method for solving LP [7].

1954 — First commercial grade software was developed by William Orchard-Hays of the Rand Corporation to solve linear programs, according to Dantzig [7].

1954 — Network flow theory's connection to graph theory was developed by Flood, Ford, and Fulkerson.

1955 — Large-scale methods were introduced by Dantzig's paper "Upper Bounds, Block Triangular Systems, and Secondary Constraints," leading to development of Dantzig-Wolfe decomposition and Benders Decomposition, and their applications in mixed integer programs and stochastic programming.

1955 — Beale and Dantzig independently proposed stochastic programming [7].

1956 — Network flow theory's connection to graph theory was developed by Hoffman and Kuhn [7].

late 1950s — Charnes and Cooper contributed to stochastic programming by their use of chance constraints [7].

1958 — Integer programming began when Gomory proposed an algorithm for generating cutting planes.

1962 — Benders published a dual method of Dantzig-Wolfe decomposition.

1960s — Application of duality was extended in nonlinear programming.

1960s — Stochastic programming was developed further by Wets [7].

1972 — Klee and Minty showed that at worst the simplex algorithm has exponential-time complexity [8].

1977 — Goldfarb proposed a steepest-edge rule for selecting the leaving variable at each dual simplex iteration, which led to more powerful implementation of dual simplex [9, 10].

1979 — Khachian developed an interior method using ellipsoids. The method was a polynomial-time algorithm for solving LP [11].

1980s — Stochastic programming was developed further by Birge [7].

1984 — Karmarkar [12] improved on the polynomial time algorithm of Khachian for solving LP.

1988 — Dual approach to interior method was shown by Renegar [7].

1988 — An optimization software CPLEX was first distributed [10].

to present — Development of variants of the simplex method and interior algorithms continues.

2.4 Primal-Dual Relationship

If the problem P defined by (1.1) – (1.3) is considered to be the primal problem, then the dual problem D is

$$\begin{aligned} \text{minimize} \quad & w = \mathbf{b}^{\mathrm{T}}\mathbf{y} && (2.1) \\ \text{subject to} \quad & \mathbf{A}^{\mathrm{T}}\mathbf{y} \geq \mathbf{c} && (2.2) \\ & \mathbf{y} \geq \mathbf{0}, && (2.3) \end{aligned}$$

where w represents an objective function for m variables

$$\mathbf{b}^{\mathrm{T}}\mathbf{y} = \begin{bmatrix} b_1 & b_2 & \cdots & b_m \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix},$$

and the expression (2.2) describes n rows of constraints for m variables

$$\begin{bmatrix} \mathbf{a}^{1\mathrm{T}} \\ \mathbf{a}^{2\mathrm{T}} \\ \vdots \\ \mathbf{a}^{n\mathrm{T}} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \geq \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix},$$

where $\mathbf{a}^j$ is a jth column of $\mathbf{A}$.

The important results of the relationship are weak duality, strong duality, and complementary slackness theorems.

Weak duality theorem says that the relationship between feasible solutions $\mathbf{x}$ and $\mathbf{y}$ is $\mathbf{b}^{\mathrm{T}}\mathbf{y} \geq \mathbf{c}^{\mathrm{T}}\mathbf{x}$. If $\mathbf{b}^{\mathrm{T}}\mathbf{y} = \mathbf{c}^{\mathrm{T}}\mathbf{x}$, then $\mathbf{x}$ and $\mathbf{y}$ are optimal solutions to P and D respectively.

The strong duality theorem observes that *a*) if either problem P or D has a feasible solution, the other problem also has a feasible solution; *b*) if either problem P or D has an unbounded solution, the other problem is infeasible; and *c*) it is possible that both problems P and D are infeasible.

Necessary and sufficient conditions for both $\mathbf{x}$ and $\mathbf{y}$ to be optimal are given by complementary slackness theorem. The conditions are

$$\mathbf{a}^{j\,\mathrm{T}}\mathbf{y} = c_j \quad \text{OR} \quad x_j = 0 \quad \forall j = 1, 2, \ldots, n$$

AND

$$\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i \quad \text{OR} \quad y_i = 0 \quad \forall i = 1, 2, \ldots, m.$$

Dual variables for binding constraints of P with an optimal solution $\mathbf{x}$ can be calculated by solving the following system of equalities

$$\mathbf{a}^{j\,\mathrm{T}}\mathbf{y} = c_j \text{ for } \{j \,|x_j > 0\}.$$

2.5 The Primal Simplex Method

The primal simplex method described below is generically known as the revised simplex method. This method was developed in 1954 by Dantzig and Orchard-Hays in "The product form for the inverse in the simplex method" [13, 14]. The revised simplex improves over the standard simplex method by finding a new solution at each iteration from the original data rather than referring to the dictionary, which gets updated at every simplex iteration.

Rewrite problem P with equality constraints by adding m slack variables

$$(\mathrm{P}') \qquad \text{maximize} \quad z = \mathbf{c}^{\mathrm{T}}\mathbf{x} \tag{2.4}$$

$$\text{subject to} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.5}$$

$$\mathbf{x} \geq \mathbf{0}. \tag{2.6}$$

Separate the basic and nonbasic columns to write (2.4) as

$$z = \mathbf{c}_B^{\mathrm{T}}\mathbf{x}_B + \mathbf{c}_N^{\mathrm{T}}\mathbf{x}_N, \tag{2.7}$$

and (2.5) as

$$\mathbf{B}\mathbf{x}_B + \mathbf{A}_N\mathbf{x}_N = \mathbf{b}. \tag{2.8}$$

Rearrange (2.8) to get

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{A}_N\mathbf{x}_N, \tag{2.9}$$

where $\mathbf{B}^{-1}\mathbf{b}$ represents the current basic solution $\mathbf{x}_B^*$. From primal-dual relationship,

$$\mathbf{y}^{\mathrm{T}} = \mathbf{c}_B^{\mathrm{T}}\mathbf{B}^{-1}. \tag{2.10}$$

Substitute (2.9) and (2.10) in (2.7) to obtain

$$z = \mathbf{c}_B^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{b} + \left(\mathbf{c}_N^{\mathrm{T}} - \mathbf{y}^{\mathrm{T}}\mathbf{A}_N\right)\mathbf{x}_N. \tag{2.11}$$

A positive coefficient for $\mathbf{x}_N$ is desired for a pivot column. When choosing an entering column from $\mathbf{A}_N$, find column j such that $\left(\mathbf{c}_N^{\mathrm{T}} - \mathbf{y}^{\mathrm{T}}\mathbf{A}_N\right)_j > 0$, where $(\mathbf{v})_j$ denotes jth element of vector $\mathbf{v}$. The variable x_j is the entering variable, and $\mathbf{a}^j$ is the entering column. Substituting the entering variable and entering column in (2.9) gives

$$\mathbf{x}_B = \mathbf{x}_B^* - x_j \mathbf{B}^{-1}\mathbf{a}^j. \tag{2.12}$$

The expression (2.12) shows that as the entering variable x_j is increased by t, $\mathbf{x}_B$ changes from $\mathbf{x}_B^*$ to $[\mathbf{x}_B^* - (t)\,(\mathbf{B}^{-1}\mathbf{a}^j)]$, eventually causing one of the variables in $\mathbf{x}_B$ to reach zero. The variable that first drops to zero becomes the leaving variable.

The revised simplex algorithm is summarized as follows.

1. Choose an initial feasible basis $\mathbf{B}$ for solving $\mathbf{y}^{\mathrm{T}}\mathbf{B} = \mathbf{c}_B^{\mathrm{T}}$. If a primal feasible basis not found, P' is infeasible.
2. Solve $\mathbf{y}^{\mathrm{T}}\mathbf{B} = \mathbf{c}_B^{\mathrm{T}}$.
3. Choose an entering or pivot column $\mathbf{a}^j$ of $\mathbf{A}_N$ such that $\left(\mathbf{c}_N^{\mathrm{T}} - \mathbf{y}^{\mathrm{T}}\mathbf{A}_N\right)_j > 0$. If no $\mathbf{a}^j$ exists, $\mathbf{x}_B$ is an optimal solution.
4. Find the largest t that satisfies $\mathbf{x}_B^* - t\mathbf{B}^{-1}\mathbf{a}^j \geq 0$. If no t exists, P' is unbounded.
5. Choose a leaving variable x_k such that $(\mathbf{x}_B^* - t\mathbf{B}^{-1}\mathbf{a}^j)_k = 0$ using t from the previous step.
6. Update $\mathbf{x}_B^*$ with $\mathbf{x}_B^* - t\mathbf{B}^{-1}\mathbf{a}^j$ except for the entering variable. Use t for the entering variable.
7. Update $\mathbf{B}$ by replacing the leaving column $\mathbf{a}^k$ with the entering column $\mathbf{a}^j$. Go to step 2.

2.6 The Dual Simplex Method

The dual simplex method was developed by Lemke [15] in 1954. While the steps in primal simplex move from a primal-feasible solution to another, the dual simplex starts with a primal-infeasible solution. The dual simplex keeps the dual-feasible basis, while searching for a primal feasible solution. Therefore the method is useful when a dual-feasible solution is easily found while a primal-feasible solution is not. Examples include sensitivity analysis where new constraints are introduced, and active-set methods where constraints are added iteratively to a relaxed problem. The dual simplex algorithm is summarized as follows.

1. Choose an initial dual-feasible basis $\mathbf{B}$ such that $\mathbf{c}_B^{\mathrm{T}}\mathbf{B} \geq \mathbf{0}$. If a feasible basis is not found, P' is infeasible.
2. Choose a leaving variable x_k such that $(\mathbf{x}_B)_k < 0$. If no such x_k exists, $\mathbf{x}_B$ is an optimal solution. Set p to the position number of x_k in $\mathbf{x}_B$.
3. Choose an entering column j such that

$$j \in \operatorname*{arg\,min}_{j \in N} \left[\frac{c_j}{\left(\mathbf{e}^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{A}_N\right)_j} \middle| \left(\mathbf{e}^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{A}_N\right)_j < 0 \right],$$

   where N is a set of original indexes belonging to nonbasic variables, $\mathbf{e}$ is the pth column of an identity square matrix. If no such j is found, the dual problem is unbounded and P' is primal-infeasible.
4. Update $\mathbf{x}_B^*$ with

$$\mathbf{x}_B^* - \left(\frac{x_k^*}{\left(\mathbf{e}^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{A}_N\right)_j} \right) \mathbf{B}^{-1}\mathbf{a}^j,$$

   except for the entering variable. Use

$$\left(\frac{x_k^*}{\left(\mathbf{e}^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{A}_N\right)_j} \right)$$

   for the entering variable.
5. Update $\mathbf{c}_N$ with

$$-\frac{c_j}{\left(\mathbf{e}^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{A}_N\right)_j}$$

for c_k, and

$$c_s + c_k\left(\mathbf{e}^{\mathrm{T}}\mathbf{B}^{-1}\mathbf{A}_N\right)_k$$

for the rest of the elements $c_s|\, s \in N \setminus \{j\}$.

6. Update $\mathbf{B}$ by replacing the leaving column $\mathbf{a}^k$ with the entering column $\mathbf{a}^j$. Go to step 2.

2.7 Large-scale Linear Programming

General strategies for solving large-scale linear programming include division of the problem into subproblems, delayed column generation, and delayed constraint generation. Dividing of the original problems into subproblems is possible when the constraints have a special structure. In a maximization problem P, the delayed column generation approach adds column j to a subproblem of P only after determining that column j's reduced cost ${\mathbf{a}^j}^{\mathrm{T}}\mathbf{y} - c_j$ is negative. For the delayed constraint generation approach, row i is added to a subproblem of P only after determining that the row's violation check $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} - b_i$ is positive. In particular, Dantzig-Wolve decomposition is a method that divides the original problem into subproblems and applies delayed column generation. Benders decomposition also forms subproblems but applies delayed constraint generation.

Cutting-plane methods are delayed constraint generation techniques applied to convex optimization problems but they may add constraints that are not part of the original constraint set (1.2). COSTs are cutting-plane methods utilizing only (1.2).

2.8 Active-set Methods

Active-set methods divide the system of inequalities in (1.2) into *operative* and *inoperative* sets. LP is solved iteratively by strategically updating each set. Constraints in inoperative set are ignored in an active-set iteration.

2.9 Related Literature

The COSTs are active-set methods in which a series of relaxations P_r, $r = 0, 1, 2, \ldots$, of P is formed by adding one or more violating constraints from set (1.2). Active-set approaches have been studied in the past, including those by Stone [16], Thompson et al. [17], Adler et al. [18], Zeleny [19], Myers and Shih [20], and Curet [21], with the term "constraint selection technique" used in Myers and Shih [20]. Adler et al. [18] added constraints randomly, without any selection criteria. Zeleny [19] added a constraint that was most violated by the problem P_r to form P_{r+1}. These methods are called SUB and VIOL here, respectively. Also, VIOL, which is a standard pricing method for delayed column generation in terms of the dual [22], is identical to the Priority Constraint Method of Thompson et al. [17]. In all of these approaches, constraints were added one at a time.

More recent work on constraint selection has focused on choosing the violated inoperative constraints considered most likely to be binding at optimality for the original problem P according to a particular constraint selection criterion. In the cosine criterion, the angle between normal vector $\mathbf{a}_i$ of (1.2) and normal vector $\mathbf{c}$ of (1.1) as measured by the cosine, $\mathrm{COS}\,(\mathbf{a}_i, \mathbf{c}) = \frac{\mathbf{a}_i^{\mathrm{T}}\mathbf{c}}{\|\mathbf{a}_i\|\|\mathbf{c}\|}$, was considered. Naylor and Sell [23][pp. 273–274], for example, suggested that a constraint with a larger cosine value may be more likely to be binding at optimality. Pan [24, 25] applied the cosine criterion to pivot rules of the simplex algorithm as the "most-obtuse-angle" rule. The cosine criterion has also been utilized to obtain an initial basis for the simplex algorithm by Trigos et al. [26], Junior and Lins [27]. Corley and Rosenberger [28], Corley et al. [4] chose for P_{r+1} a single inoperative constraint $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} \leq b_i$ of P_r violating $\mathbf{x}_r^*$ and having the largest $\cos(\mathbf{a}_i, \mathbf{c})$.

CHAPTER 3

CONSTRAINT OPTIMAL SELECTION TECHNIQUES (COSTs)

3.1 Introduction

In this chapter, the COST NRAD for nonnegative linear programs (NNLPs) is first developed, then the COST GRAD for LPs is developed loosely based on the intuition gained from NRAD.

3.2 NNLP

NNLP is a special case of P where $\mathbf{a}_i \geq \mathbf{0}$ and $\mathbf{a}_i \neq \mathbf{0}$, $\forall i = 1, \ldots, m$; $\mathbf{b} > \mathbf{0}$; and $\mathbf{c} > \mathbf{0}$. Useful properties of NNLPs include *a*) the origin $\mathbf{x} = \mathbf{0}$ is guaranteed to be a feasible point that minimizes the objective function; *b*) variable x_j is bounded if and only if $\mathbf{a}^j > \mathbf{0}$; and *c*) the upper bound on x_j is $\min\limits_{i=1,\ldots,m} \left\{ \frac{b_i}{a_{ij}} \middle| \, a_{ij} > 0 \right\} \forall j = 1, \ldots, n$.

3.3 Active-set Framework

An active-set framework utilized for COSTs begins with a relaxation of P, with initial bounding constraint(s). A bounded P_0 could be formed by adding a single artificial bounding constraint such as $\mathbf{1x} \leq M$ or $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M$ for sufficiently large M so as not to reduce the feasible region of P.

A series of relaxations P_r, $r = 0, 1, 2, \ldots,$ of P is formed by adding one or more violating constraints from set (1.2). The constraints that have been added are called *operative constraints*, while constraints that still remain in (1.2) are called *inoperative constraints*. Eventually a solution $\mathbf{x}_r^*$ of P is obtained when none of the inoperative constraints are violated, i.e., for no inoperative constraint i is $\mathbf{a}_i^{\mathrm{T}}\mathbf{x}_r^* - b_i > 0$. In

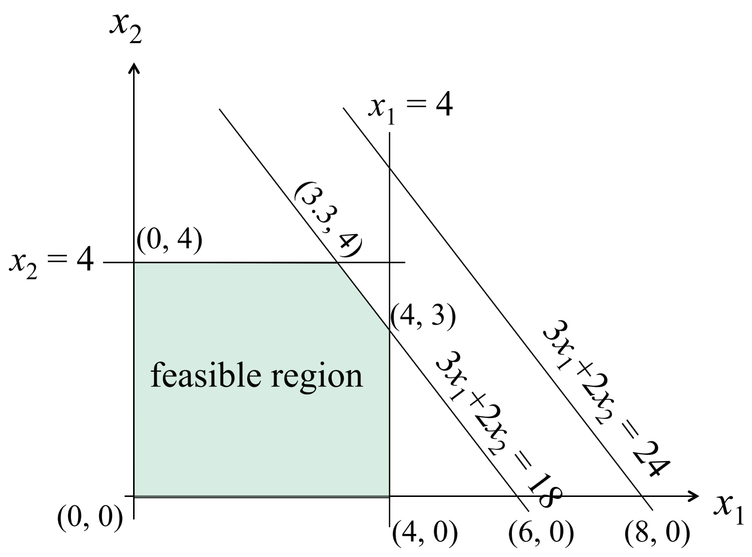

Figure 3.1. Geometric presentation of NNLP example (3.1).

the COSTs of this dissertation, the following is explored: *a*) the ordering of a set of inoperable constraints for possibly adding them to the current operable constraints; and *b*) the actual selection of a group of such constraints to be added at an iteration.

Take a simple 2-dimensional NNLP example presented in (3.1) below and Figure 3.1

$$\begin{aligned} \text{maximize} \quad & z = 3x_1 + 5x_2 \\ \text{subject to} \quad & x_1 \le 4 \\ & x_2 \le 4 \\ & 3x_1 + x_2 \le 18 \\ & 3x_1 + x_2 \le 24 \\ & x_1, x_2 \ge 0. \end{aligned} \tag{3.1}$$

Figure 3.2 shows that as the objective function $z = 3x_1 + 5x_2$ is increased,

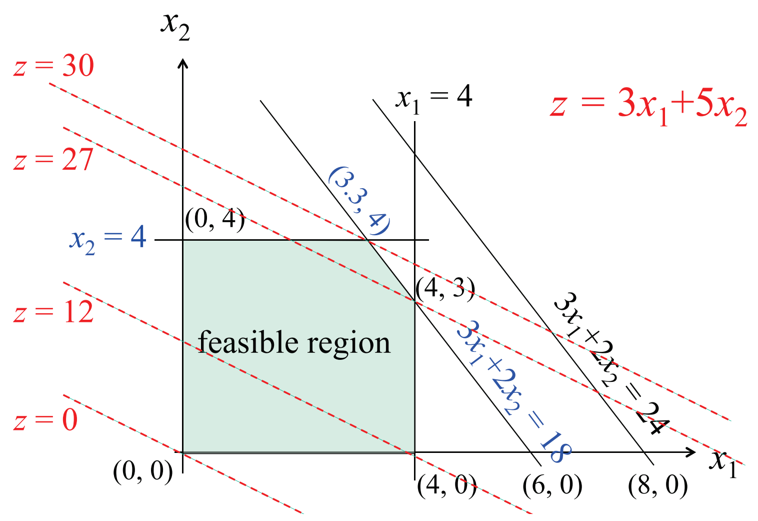

Figure 3.2. Binding constraints at optimality in NNLP example (3.1).

the optimal objective value of 30 is achieved at $\mathbf{x}^* = \begin{bmatrix} 3.3 & 4 \end{bmatrix}$, with the binding constraints $x_2 \leq 4$ and $3x_1 + x_2 \leq 18$.

From the work described in Section 2.9, the selection of constraints likely to be binding at optimality seems to be influenced by the following two geometric factors. Factor I is the angle that the constraint's normal vector $\mathbf{a}_i$ forms with the normal vector $\mathbf{c}$ of the objective function. Factor II is the depth of the cut that constraint i removes as a violated inoperative constraint of P_r. Figure 3.3 illustrates this observation. A COST NRAD incorporating both Factor I and Factor II is developed below.

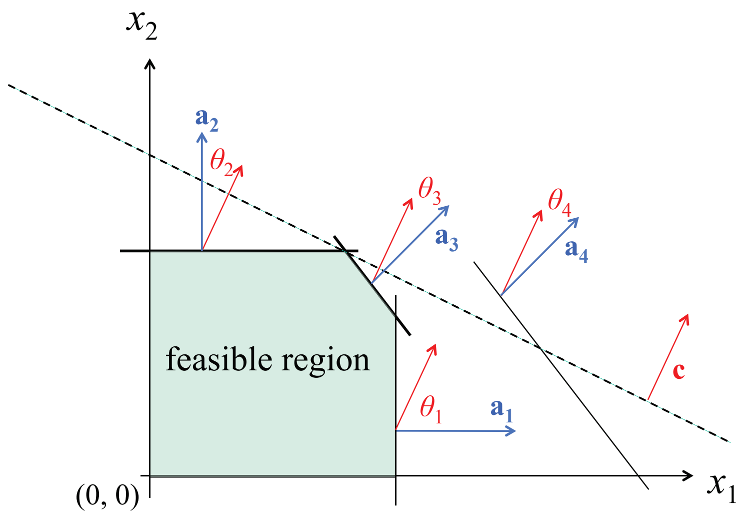

Figure 3.3. Factor I and Factor II in NNLP example (3.1).

3.4 The COST NRAD for NNLP

3.4.1 Constraint Selection Criterion

Define a constraint selection metric

$$\mathrm{NRAD}\left(\mathbf{a}_i, b_i, \mathbf{c}\right) = \frac{\mathbf{a}_i^{\mathrm{T}}\mathbf{c}}{b_i}. \tag{3.2}$$

Since the elements of $\mathbf{A}$ are nonnegative and the elements of $\mathbf{c}$ and $\mathbf{b}$ are strictly positive for NNLPs, NRAD > 0. In COST NRAD, one or more violating inoperative constraints that have the highest values of NRAD are selected to become operative constraints. In other words, NRAD seeks constraint $\mathbf{a}_{i^*}^{\mathrm{T}}\mathbf{x} \leq b_{i^*}$ such that $i^* \in \underset{i \notin OPERATIVE}{\arg\max} \left(\mathrm{NRAD}\left(\mathbf{a}_i, b_i, \mathbf{c}\right) \middle| \, \mathbf{a}_i^{\mathrm{T}}\mathbf{x}_r^* > b_i\right)$. Writing NRAD as

$$\frac{\mathbf{a}_i^{\mathrm{T}}\mathbf{c}}{b_i} = \frac{\left\|\mathbf{a}_i\right\|}{b_i}\frac{\mathbf{a}_i^{\mathrm{T}}\mathbf{c}}{\left\|\mathbf{a}_i\right\|\left\|\mathbf{c}\right\|}\left\|\mathbf{c}\right\| \propto \frac{\left\|\mathbf{a}_i\right\|}{b_i}\mathrm{COS}\left(\mathbf{a}_i, \mathbf{c}\right)$$

illustrates that it consists of a Factor I term $\text{COS}(\mathbf{a}_i, \mathbf{c})$ of Corley et al. [28] and a Factor II term $\frac{\|\mathbf{a}_i\|}{b_i}$.

3.4.2 NRAD in an Active-set Framework Utilizing Multiple Cuts

Since $\mathbf{a}^j \geq \mathbf{0}$ and $\mathbf{a}^j \neq \mathbf{0}\ \forall j = 1, \ldots, n$ assures boundedness for NNLPs, P_0 is bounded by adding multiple constraints from (1.2) in decreasing order of NRAD until $\mathbf{a}^j \geq \mathbf{0}$ and $\mathbf{a}^j \neq \mathbf{0}\ \forall j = 1, \ldots, n$ in P_0 (Step 1: *multi-bound*). A solution $\mathbf{x}_0^*$ to P_0 is found by the primal simplex method (Step 2). To form $\text{P}_r, r = 1, \ldots,$ multiple inoperative constraints that violate $\mathbf{x}_0^*$ are added in decreasing order of NRAD until $\mathbf{a}^j \geq \mathbf{0}$ and $\mathbf{a}^j \neq \mathbf{0}\ \forall j = 1, \ldots, n$, where $\mathbf{a}^j$ is the jth column newly added set of constraints (lines 8-15 of Step 3: *multi-cut*). Problem P_r is solved iteratively by the dual simplex method until there are no more violating inoperative constraints. The following pseudocode summarizes the steps for the COST NRAD.

Step 1 — Identify constraints to initially bound the problem.

```
1: $\mathbf{a}^* \leftarrow \mathbf{0}$
2: while $\mathbf{a}^* \not> \mathbf{0}$ do
3:     Let $i^* \in \underset{i \notin BOUNDING}{\arg\max}\ \text{NRAD}(\mathbf{a}_i, b_i, \mathbf{c})$
4:     if $\exists j |\ a_j^* = 0$ and $a_{i^*j} > 0$ then
5:         $BOUNDING \leftarrow BOUNDING \cup \{i^*\}$
6:     end if
7:     $\mathbf{a}^* \leftarrow \mathbf{a}^* + \mathbf{a}_{i^*}$
8: end while
```

Step 2 — Using the primal simplex method, obtain an optimal solution $\mathbf{x}_0^*$ for the initial bounded problem P_0

$$\text{maximize} \quad z = \mathbf{c}^{\mathrm{T}}\mathbf{x} \tag{3.3}$$

$$\text{subject to} \quad \mathbf{a}_i^{\mathrm{T}}\mathbf{x} \le b_i \quad \forall i \in BOUNDING \tag{3.4}$$

$$\mathbf{x} \ge \mathbf{0}. \tag{3.5}$$

Step 3 — Perform the following iterations until an optimal solution to problem P is found.

```
1: $r \leftarrow 0$
2: $STOP \leftarrow$ **false**
3: $OPERATIVE \leftarrow BOUNDING$
4: **while** $STOP =$ **false do**
5:     **if** $\mathbf{a}_i^{\mathrm{T}}\mathbf{x}_r^* \le b_i \ \forall i \notin OPERATIVE$ **then**
6:         $STOP \leftarrow$ **true** // $\mathbf{x}_r^*$ is an optimal solution to P.
7:     **else**
8:         $\mathbf{a}^* \leftarrow \mathbf{0}$
9:         **while** $\mathbf{a}^* \not> \mathbf{0}$ **and** $OPERATIVE \subset \left\{i \middle| \mathbf{a}_i^{\mathrm{T}}\mathbf{x}_r^* > b_i\right\}$ **do**
10:            Let $i^* \in \underset{i \notin OPERATIVE}{\arg\max} \ \mathrm{NRAD}\left(\mathbf{a}_i, b_i, \mathbf{c} \middle| \mathbf{a}_i{}^{\mathrm{T}}\mathbf{x}_r^* > b_i\right)$
11:            **if** $\exists j |\, a_j^* = 0$ **and** $a_{i^*j} > 0$ **then**
12:                $OPERATIVE \leftarrow OPERATIVE \cup \{i^*\}$
13:            **end if**
14:            $\mathbf{a}^* \leftarrow \mathbf{a}^* + \mathbf{a}_{i^*}$
15:        **end while**
16:        $r \leftarrow r + 1$
```

```
17:         Solve the following $P_r$ by the dual simplex method to obtain $\mathbf{x}_r^*$.
```

$$\begin{aligned} \text{maximize} \quad & z = \mathbf{c}^{\mathrm{T}}\mathbf{x} && (3.6) \\ \text{subject to} \quad & \mathbf{a}_i^{\mathrm{T}}\mathbf{x} \le b_i \quad \forall i \in OPERATIVE && (3.7) \\ & \mathbf{x} \ge \mathbf{0}. && (3.8) \end{aligned}$$

```
18:     end if
19: end while
```

3.4.3 Geometric Interpretation

A geometric representation of a constraint for problem P with two variables is shown in Figure 3.4. When P is an NNLP, the value of NRAD $= \mathbf{a}_i^{\mathrm{T}}\mathbf{c}$ is positive. Note that the distance between the origin and the intersection of the hyperplane $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i$ and the normal vector $\mathbf{c}$ is $\frac{b_i\|\mathbf{c}\|}{\mathbf{a}_i^{\mathrm{T}}\mathbf{c}}$. As depicted in Figure 3.5, NRAD may be interpreted as approximating the feasible region by nested spheres. The name RAD in NRAD (NNLP-RAD) comes form this observation. The figure also shows that the two binding constraints at optimality have the highest values of NRAD, 1.250 and 1.056.

However, since $\mathbf{a}_i$ may include zeros in certain components, the hyperplane may include some zeros in certain components, therefore, the hyperplane from a single constraint $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} \le b_i$ may not effectively form an angle in every dimension of $\mathbf{c}$. Consequently, adding multiple constraints to P_r in which there is at least one positive coefficient for each variable x_j will add cutting planes forming a geodesic-like dome cutting off the current $\mathbf{x}_r^*$ in a more efficient manner than a single cutting plane.

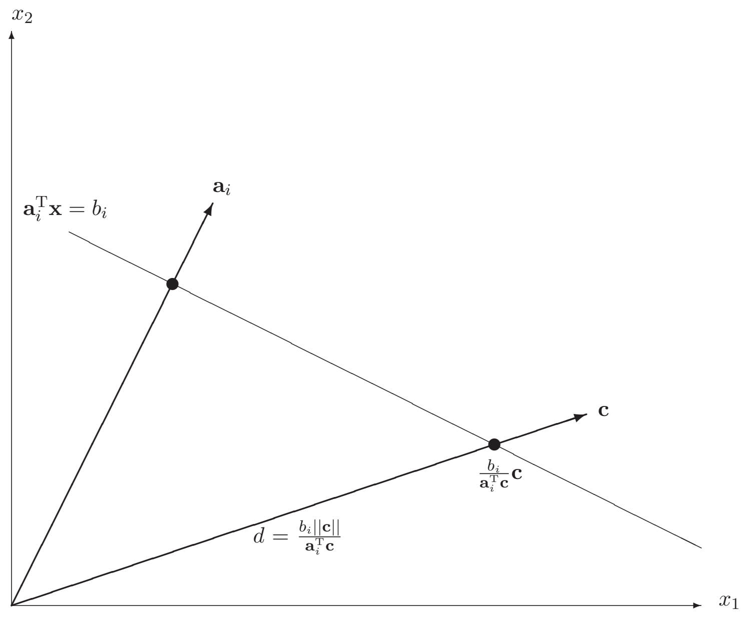
x_2
$\mathbf{a}_i$
$\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i$
$\mathbf{c}$
$\frac{b_i}{\mathbf{a}_i^{\mathrm{T}}\mathbf{c}}\mathbf{c}$
$d = \frac{b_i\|\mathbf{c}\|}{\mathbf{a}_i^{\mathrm{T}}\mathbf{c}}$
x_1

Figure 3.4. Geometric interpretation of NRAD.

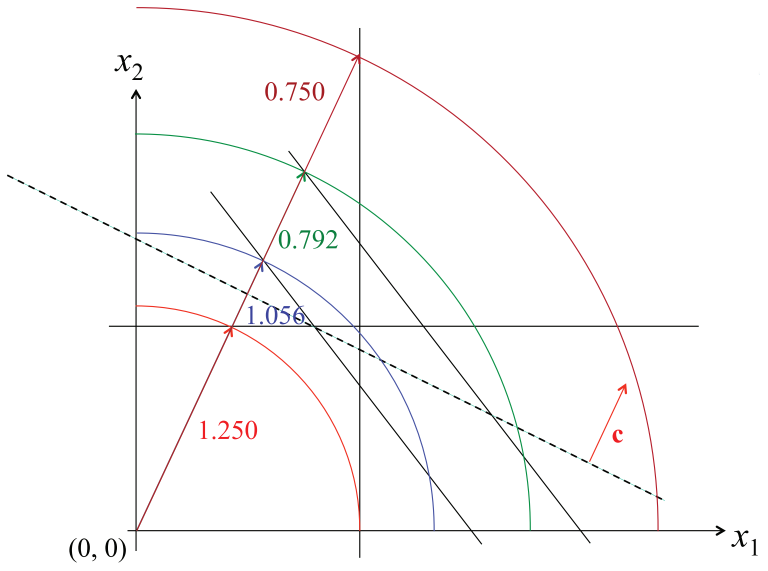

Figure 3.5. Geometric interpretation of NRAD in NNLP Example (3.1).

3.5 The COST GRAD for General LP

An active-set framework for solving general LPs will be analogous to that for NNLPs. However, GRAD (General-LP RAD) is not an immediate extension of NRAD since LP does not have some of the useful properties of NNLP problems. For example, the origin $\mathbf{x} = \mathbf{0}$ is no longer guaranteed to be feasible for LP problems. Moreover, the optimal solution $\mathbf{x}^*$ may not lie in the same orthant as the normal $\mathbf{a}_i$ to a constraint. We must thus modify NRAD for NNLP to GRAD for LP in order to emulate efficiently the underlying reasoning of NRAD based on Factors I and II.

3.5.1 Constraint Selection Criterion

Boundedness of NNLP could be assured by adding multiple constraints from (1.2) until no column of $\mathbf{A}$ is a zero vector. However, this is not the case for LP. Therefore an initial bounded problem P_0 is formed by adding a bounding constraint such as $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M$, along with some constraints from (1.2), as described in Section 3.5.2. P_0 is then solved to obtain an initial solution $\mathbf{x}_0^*$. P_{r+1} is generated by adding one or more inoperative constraints of P_r that maximize the constraint selection metric for LP among all inoperative constraints of P_r violating $\mathbf{x}_r^*$. Define this constraint selection metric as

$$\mathrm{GRAD}\left(\mathbf{a}_i, b_i, \mathbf{c}\right) = \sum_{j=1, c_j>0}^{n} \frac{a_{ij}c_j}{b_i^+} - \sum_{j=1, c_j<0}^{n} \frac{-a_{ij}}{b_i^+}, \tag{3.9}$$

where

$$b_i^+ = \begin{cases} b_i - \min\limits_{k=1,\ldots,m} [b_k] + \varepsilon, & \text{if } \min\limits_{k=1,\ldots,m} [b_k] < 0 \\ b_i, & \text{otherwise.} \end{cases} \tag{3.10}$$

Thus GRAD seeks $i^* \in \underset{i \notin OPERATIVE}{\arg\max} \left(\mathrm{GRAD}\left(\mathbf{a}_i, b_i, \mathbf{c}\right) \middle| \, \mathbf{a}_i^{\mathrm{T}}\mathbf{x}_r^* > b_i\right)$.

The first term in (3.9) is a quantity that invokes Factor I and Factor II analogous to NRAD, while the second term is a quantity that invokes Factor II. In (3.10), values of b_i are shifted by $\min\limits_{k=1,\ldots,m}[b_i]$ if the minimum value is negative. Hence b_i^+ is always positive, and each term in (3.9) contributes additively to the criterion. The GRAD (3.9) becomes the same as the NRAD (3.2) when $\mathbf{b} > \mathbf{0}$ and $\mathbf{c} > \mathbf{0}$. Therefore it could be utilized to effectively solve NNLPs as well. Although equality constraints are not considered here, it should be noted that equality constraints could be included in P_0.

Figure 3.4 depicted a two-dimensional example of an NNLP, always resulting in a positive value for $\mathbf{a}_i^{\mathrm{T}}\mathbf{c}$. However for LP, as described above in Section 3.5, the intersection of $\mathbf{c}$, drawn from the origin, and $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i$ may not necessarily lie in the feasible region. Furthermore, unlike in an NNLP, the origin may not be a feasible solution for LP. Therefore calculating values based upon the origin may provide irrelevant criteria.

The intuition for judging a general LP constraint's likelihood of being binding at optimality may be described as follows. Given an objective function $\max z = c_1x_1 + c_2x_2 + \ldots + c_nx_n$, observe that the objective is maximized when c_j and x_j are large. Hence, a larger value of c_j is more likely to yield a larger value of x_j. This relationship implies that the left-hand side of the constraint is likely to be larger for larger values of $\sum\limits_{j=1,c_j>0}^{n} a_{ij}c_j$. For a_j with $c_j < 0$, it is hard to predict the likely value of x_j in a solution. Consequently, we assume that the x_j in which $c_j < 0$ are all equally likely to be a nominal value of 1. The left-hand side is now $\sum\limits_{j=1,c_j>0}^{n} a_{ij}c_j + \sum\limits_{j=1,c_j<0}^{n} a_{ij}$. As for the right-hand side of the constraint, a small b_i makes a constraint more likely to be binding. We thus divide the left-hand side by b_i to measure the ith constraint's likelihood of being binding at optimality, resulting in $\sum\limits_{j=1,c_j>0}^{n} \frac{a_{ij}c_j}{b_i} + \sum\limits_{j=1,c_j<0}^{n} \frac{a_{ij}}{b_i}$, which is essentially GRAD.

GRAD can also be derived from NRAD by incorporating the term $\sum_{j=1,c_j>0}^{n} \frac{a_{ij}c_j}{b_i^+}$ that results in a higher GRAD value when a_{ij} and c_j take large positive values with a small b_i^+. This term considers only dimensions of j for which the components of $\mathbf{c}$ are positive, making the interpretation analogous to the NNLP case.

To incorporate dimensions of j for which the components of $\mathbf{c}$ are negative to NRAD, consider the following metric analogous to NRAD

$$\sum_{j=1,c_j>0}^{n} \frac{a_{ij}c_j}{b_i^+} - \sum_{j=1,c_j<0}^{n} \frac{a_{ij}c_j}{b_i^+}. \tag{3.11}$$

The second term in (3.11) makes sense from the point of view that the expression (3.11) results in a higher value when a_{ij} and c_j are both negative, and b_i is large. However, it is found in Section 4.5.2.1 that the constraint selection metric performed better when the second term was $\sum_{j=1,c_j<0}^{n} \frac{-a_{ij}}{b_i^+}$, as shown in (3.9). With this second term, RAD will maximize the intercept $\frac{b_i}{a_{ij}}$ of the hyperplane $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i$ and the axis for $\left\{x_j \,\middle|\, c_j < 0\right\}$.

3.5.2 GRAD in an Active-set Framework Utilizing Multiple Cuts

Boundedness for NNLP is guaranteed by adding multiple constraints from (1.2) ordered by decreasing value of NRAD until no column of $\mathbf{A}$ is a zero vector. Although this approach does not guarantee boundedness for LP, a generalization was found to be effective here.

For the COST GRAD, an initial bounded problem P_0 is formed by adding an artificial bounding constraint such as $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M$ and multiple constraints from (1.2) ordered by decreasing value of GRAD until all columns of $\mathbf{A}$ has at least one positive and at least one negative coefficient (Step 1). After an optimal solution to the initial bounded problem is obtained by the primal simplex method (Step 2), subsequent iterations are solved by the dual simplex method (Step 3). Moreover, after the solution of P_0 and each subsequent P_r, constraints are again added in groups. P_{r+1} is formed by selecting inoperative constraints in decreasing order of GRAD until a positive coefficient and a negative coefficient is included for each variable x_j (Step 3, lines 7–29). The following pseudocode depicts the COST GRAD with the new multi-cut technique.

Step 1 — Identify constraints to form the initial problem P_0.

```
1: for $i = 1 \to m$ do
2:   if $\exists j |\, a_{ij} > 0$ then
3:     $POSITIVE_a \leftarrow POSITIVE_a \cup \{i\}$
4:   end if
5:   if $\exists j |\, a_{ij} < 0$ then
6:     $NEGATIVE_a \leftarrow NEGATIVE_a \cup \{i\}$
7:   end if
8: end for
9: $\mathbf{a}^{*\text{positive}} \leftarrow \mathbf{0}$
10: $\mathbf{a}^{*\text{negative}} \leftarrow \mathbf{0}$
11: while $\mathbf{a}^{*\text{positive}} \not> \mathbf{0}$ and $\mathbf{a}^{*\text{negative}} \not> \mathbf{0}$ and $OPERATIVE \subset \{i\}$ do
12:   Let $i^* \in \underset{i \notin OPERATIVE}{\arg\max}\ \mathrm{GRAD}\left(\mathbf{a}_i, b_i, \mathbf{c}\right)$
13:   if $\exists j |\, a_j^{*\text{positive}} = 0$ and $a_{i^*j} > 0$ then
14:     $OPERATIVE \leftarrow OPERATIVE \cup \{i^*\}$
15:     if $OPERATIVE \subset POSITIVE_a$ then
16:       $\mathbf{a}^+ \leftarrow \left[a^+_{i^*1} \dots a^+_{i^*n}\right]$ where $a^+_{i^*j} = \begin{cases} 1, \text{if } a_{i^*j} > 0 \\ 0, \text{otherwise} \end{cases} \quad \forall j = 1 \dots n$
17:     else
18:       $\mathbf{a}^+ \leftarrow \mathbf{1}$ // case if there are no more constraints with $a_{ij} > 0$
```

```
19:         end if
20:     end if
21:     if $\exists j |\, a_j^{*\text{negative}} = 0$ and $a_{i^*j} < 0$ then
22:         $OPERATIVE \leftarrow OPERATIVE \cup \{i^*\}$
23:         if $OPERATIVE \subset NEGATIVE_a$ then
24:             $\mathbf{a}^- \leftarrow \left[a^-_{i^*1} \ldots a^-_{i^*n}\right]$ where $a^-_{i^*j} = \begin{cases} 1, \text{if } a_{i^*j} < 0 \\ 0, \text{otherwise} \end{cases} \quad \forall j = 1 \ldots n$
25:         else
26:             $\mathbf{a}^- \leftarrow \mathbf{1}$ // case if there are no more constraints with $a_{ij} < 0$
27:         end if
28:     end if
29:     $\mathbf{a}^{*\text{positive}} \leftarrow \mathbf{a}^{*\text{positive}} + \mathbf{a}^+$
30:     $\mathbf{a}^{*\text{negative}} \leftarrow \mathbf{a}^{*\text{negative}} + \mathbf{a}^-$
31: end while
```

Step 2 — Using the primal simplex method, obtain an optimal solution $\mathbf{x}_0^*$ for the initial bounded problem P_0

$$\text{maximize} \quad z = \mathbf{c}^{\mathrm{T}}\mathbf{x} \tag{3.12}$$

$$\text{subject to} \quad \mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M \tag{3.13}$$

$$\mathbf{a}_i^{\mathrm{T}}\mathbf{x} \leq b_i \quad \forall i \in OPERATIVE \tag{3.14}$$

$$\mathbf{x} \geq \mathbf{0}. \tag{3.15}$$

Step 3 — Perform the following iterations until an optimal solution to problem P is found.

```
1: $r \leftarrow 0$
```

```
2:  $STOP \leftarrow$ false
3:  while $STOP =$ false do
4:    if $\mathbf{a}_i^{\mathrm{T}}\mathbf{x}_r^* \leq b_i \ \forall i \notin OPERATIVE$ then
5:      $STOP \leftarrow$ true // $\mathbf{x}_r^*$ is an optimal solution to P.
6:    else
7:      $\mathbf{a}^{*\text{positive}} \leftarrow \mathbf{0}$
8:      $\mathbf{a}^{*\text{negative}} \leftarrow \mathbf{0}$
9:      while $\mathbf{a}^{*\text{positive}} \not> \mathbf{0}$ and $\mathbf{a}^{*\text{negative}} \not> \mathbf{0}$ and $OPERATIVE \subset \left\{ i | \, \mathbf{a}_i^{\mathrm{T}}\mathbf{x}_r^* > b_i \right\}$
        do
10:       Let $i^* \in \underset{i \notin OPERATIVE}{\arg\max} \ \mathrm{GRAD}\left(\mathbf{a}_i, b_i, \mathbf{c} | \, \mathbf{a}_i{}^{\mathrm{T}}\mathbf{x}_r^* > b_i\right)$
11:       if $\exists j | \, a_j^{*\text{positive}} = 0$ and $a_{i^*j} > 0$ then
12:         $OPERATIVE \leftarrow OPERATIVE \cup \{i^*\}$
13:         if $OPERATIVE \subset POSITIVE_a$ then
14:           $\mathbf{a}^+ \leftarrow \left[a_{i^*1}^+ \dots a_{i^*n}^+\right]$ where $a_{i^*j}^+ = \begin{cases} 1, \text{if } a_{i^*j} > 0 \\ 0, \text{otherwise} \end{cases} \quad \forall j = 1 \dots n$
15:         else
16:           $\mathbf{a}^+ \leftarrow \mathbf{1}$ // case if there are no more constraints with $a_{ij} > 0$
17:         end if
18:       end if
19:       if $\exists j | \, a_j^{*\text{negative}} = 0$ and $a_{i^*j} < 0$ then
20:         $OPERATIVE \leftarrow OPERATIVE \cup \{i^*\}$
21:         if $OPERATIVE \subset NEGATIVE_a$ then
22:           $\mathbf{a}^- \leftarrow \left[a_{i^*1}^- \dots a_{i^*n}^-\right]$ where $a_{i^*j}^- = \begin{cases} 1, \text{if } a_{i^*j} < 0 \\ 0, \text{otherwise} \end{cases} \quad \forall j = 1 \dots n$
23:         else
24:           $\mathbf{a}^- \leftarrow \mathbf{1}$ // case if there are no more constraints with $a_{ij} < 0$
```

```
25:           end if
26:         end if
27:         $\mathbf{a}^{*\text{positive}} \leftarrow \mathbf{a}^{*\text{positive}} + \mathbf{a}^{+}$
28:         $\mathbf{a}^{*\text{negative}} \leftarrow \mathbf{a}^{*\text{negative}} + \mathbf{a}^{-}$
29:       end while
30:       $r \leftarrow r + 1$
31:       Solve the following $\mathrm{P}_r$ by the dual simplex method to obtain $\mathbf{x}_r^*$.
```

$$\begin{aligned} \text{maximize} \quad & z = \mathbf{c}^{\mathrm{T}}\mathbf{x} && (3.16) \\ \text{subject to} \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M && (3.17) \\ & \mathbf{a}_i^{\mathrm{T}}\mathbf{x} \leq b_i \quad \forall i \in OPERATIVE && (3.18) \\ & \mathbf{x} \geq \mathbf{0}. && (3.19) \end{aligned}$$

```
32:     end if
33: end while
```

CHAPTER 4

COMPUTATIONAL EXPERIMENTS

4.1 Introduction

The COST NRAD and COST GRAD were compared with the CPLEX primal simplex method, the CPLEX dual simplex method, the polynomial interior-point CPLEX barrier method, as well as the previously defined constraint selection techniques SUB, COS and VIOL. NRAD, GRAD, SUB, COS, and VIOL utilized the CPLEX primal simplex solver to solve P_0 and the CPLEX dual simplex solver to solve each new relaxed problem P_{r+1}.

4.2 CPLEX Preprocessing

The CPLEX preprocessing parameters PREIND (preprocessing presolve indicator) and PREDUAL (preprocessing dual) had to be chosen appropriately. The default parameter settings of PREIND = 1 (ON) and PREDUAL = 0 (AUTO) were used for CPU times of the CPLEX primal simplex method, the CPLEX dual simplex method, and the CPLEX barrier method when comparing against the COSTs. No CPLEX preprocessing was implemented [used PREIND = 0 (OFF) and PREDUAL = −1 (OFF)] by the CPLEX primal simplex and dual simplex solvers as part of NRAD, GRAD, SUB, and VIOL.

The first parameter PREIND turns on and off the CPLEX's proprietary "preprocessing presolve" routine. The routine significantly decreases the computation time of CPLEX by reducing problem size in dimensions of m and n before the primal simplex, dual simplex method, and the barrier algorithms are applied.

The second parameter PREDUAL (preprocessing dual) has three settings: on, auto, and off. The parameter which determines if CPLEX takes the dual of the problem also greatly affects the computation time.

The default settings for the two preprocessing parameters are PREIND = 1 (preprocessing presolve ON) and PREDUAL = 0 (preprocessing dual AUTO). Without turning them off explicitly, CPLEX automatically alters the LPs. However, for the CPLEX barrier solver, the prosolve routine is an integral part of the CPLEX algorithm, because even with PREIND = 0 (OFF), the problem is altered with "restricted presolve."

Results presented in Section 4.5.1.1 show the effect of CPLEX preprocessing on CPU times.

4.3 Equipment

Comparisons of computational methods were performed with the IBM CPLEX 12.1 callable library on an Intel Core 2 Duo E8600 3.33GHz workstation with a Linux 64-bit operating system and 8 GB of RAM. Computational test results of Tables 4.4 through 4.14 were obtained by calling CPLEX commands from an application written in the programming language C. In these tables, each CPU time presented is an average computation time of solving five instances of randomly generated LP.

4.4 Problem Instances

4.4.1 NNLP

Three sets of randomly generated NNLPs were constructed. In the first set, NNLPs with 1,000 variables (n) and 200,000 constraints (m) were generated at various densities ranging from 0.005 to 1. Randomly generated real numbers between 1 and

Table 4.1. Randomly Generated NNLP Problem Set 1

Number of Variables	1,000
Number of Constraints	200,000
Range of a_{ij}	$1 \leq$ Random Real ≤ 5
Range of b_i	$1 \leq$ Random Real ≤ 10
Range of c_j	$1 \leq$ Random Real ≤ 10

	Average of 5 instances of LPs at each density					
Problem Instance	Density	Minimum number of nonzero a_{ij} in a constraint	Maximum number of nonzero a_{ij} in a constraint	Average number of nonzero a_{ij} in a constraint	Total number of nonzero a_{ij}	Number of binding constraints at optimality
1 – 5	0.005050	2.0	17.8	5.1	1,009,980	726.2
6 – 10	0.006019	2.0	19.8	6.0	1,203,860	717.0
11 – 15	0.007005	2.0	22.4	7.0	1,401,005	701.6
16 – 20	0.008001	2.0	23.8	8.0	1,600,161	673.6
21 – 25	0.009004	2.0	27.0	9.0	1,800,698	668.4
26 – 30	0.009999	2.0	27.4	10.0	1,999,898	665.2
31 – 35	0.020001	3.6	43.8	20.0	4,000,270	571.4
36 – 40	0.029999	10.0	57.6	30.0	5,999,746	513.4
41 – 45	0.040007	15.4	71.8	40.0	8,001,314	488.0
46 – 50	0.049997	22.6	85.2	50.0	9,999,341	460.4
51 – 55	0.060002	29.4	100.0	60.0	12,000,361	433.2
56 – 60	0.069995	37.4	110.0	70.0	13,998,950	422.6
61 – 65	0.080009	44.4	122.4	80.0	16,001,832	396.6
66 – 70	0.089990	51.2	134.2	90.0	17,998,055	384.0
71 – 75	0.099997	61.2	146.6	100.0	19,999,422	372.4
76 – 80	0.199993	146.6	261.6	200.0	39,998,674	310.8
81 – 85	0.300012	235.0	369.6	300.0	60,002,460	262.2
86 – 90	0.399997	333.4	472.0	400.0	79,999,452	233.0
91 – 95	0.499983	429.0	572.6	500.0	99,996,701	202.6
96 – 100	0.750009	684.6	809.0	750.0	150,001,816	158.0
101 – 105	1.000000	1,000.0	1,000.0	1,000.0	200,000,000	111.0

5, 1 and 10, 1 and 10 were assigned to elements of **A**, **b**, and **c** respectively. The number of nonzero a_{ij} in each constraint was binomially distributed $B(n, p = \text{density})$. Additionally, we required each constraint to have at least two nonzero a_{ij}, so that a constraint would not become a simple upper bound on a variable. At each of the 21 densities, 5 random LPs were generated. Table 4.1 summarizes the NNLPs generated for Set 1.

Set 2 was prepared using the same method as in Set 1, but with different parameters. Here n and m were increased to 5,000 and 1,000,000, respectively; and randomly generated real numbers between 1 and 100 were assigned to elements of **b**, and **c**. Since computer memory was limited to approximately 3×10^6 nonzero a_{ij}, 0.06

Table 4.2. Randomly Generated NNLP Problem Set 2

Number of Variables	5,000
Number of Constraints	1,000,000
Range of a_{ij}	$1 \leq$ Random Real ≤ 5
Range of b_i	$1 \leq$ Random Real ≤ 100
Range of c_j	$1 \leq$ Random Real ≤ 100

	Average of 5 instances of LPs at each density					
Problem Instance	Density	Minimum number of nonzero a_{ij} in a constraint	Maximum number of nonzero a_{ij} in a constraint	Average number of nonzero a_{ij} in a constraint	Total number of nonzero a_{ij}	Number of binding constraints at optimality
1 – 5	0.000508	2.0	12.4	2.5	2,540,864	3,641.0
6 – 10	0.000574	2.0	13.2	2.9	2,868,363	3,437.6
11 – 15	0.000650	2.0	14.4	3.3	3,249,509	3,244.0
16 – 20	0.000733	2.0	15.6	3.7	3,667,001	3,149.2
21 – 25	0.000822	2.0	17.0	4.1	4,108,983	3,054.2
26 – 30	0.000914	2.0	17.6	4.6	4,570,717	2,967.8
31 – 35	0.001009	2.0	18.8	5.1	5,047,736	2,891.2
36 – 40	0.002000	2.0	28.4	10.0	9,998,206	2,475.2
41 – 45	0.003000	2.0	38.4	15.0	15,001,505	2,227.6
46 – 50	0.004000	2.8	45.2	20.0	19,999,272	2,068.2
51 – 55	0.004999	5.4	52.6	25.0	24,996,828	1,936.4
56 – 60	0.006000	7.8	59.4	30.0	30,000,841	1,861.2
61 – 65	0.007001	10.0	68.6	35.0	35,003,724	1,766.2
66 – 70	0.008000	13.0	74.4	40.0	40,000,970	1,692.2
71 – 75	0.009001	17.6	79.8	45.0	45,002,765	1,630.8
76 – 80	0.009999	20.2	87.6	50.0	49,996,022	1,598.4
81 – 85	0.020000	56.8	150.6	100.0	99,999,172	1,288.2
86 – 90	0.030000	94.4	211.8	150.0	150,001,867	1,146.8
91 – 95	0.039999	137.2	271.6	200.0	199,995,504	1,028.6
96 – 100	0.050000	175.8	328.2	250.0	249,999,857	968.6
101 – 105	0.060002	224.4	388.4	300.0	300,008,499	901.6

was the maximum possible density allowed. Hence, much smaller density increments were chosen. Table 4.2 summarizes the NNLPs generated for Set 2.

The third set, in which the ratio m/n was varied from 200 to 1, is described below in Section 4.5.1.5.

4.4.2 LP

A set of 105 randomly generated LP was constructed. The LP problems were generated with 1,000 variables (n) and 200,000 constraints (m) having various densities ranging from 0.005 to 1. Randomly generated real numbers between 1 and 5, or between -1 and -5 were assigned to elements of $\mathbf{A}$. In order to assure that the ran-

Table 4.3. Randomly Generated General LP Problem Set

Number of variables	1,000
Number of constraints	200,000
Range of a_{ij}	$1 \leq$ random real ≤ 5, or $-5 \leq$ random real ≤ -1
Fraction of positive a_{ij}	0.5

	Average of 5 instances of LP at each density										
Problem instance	Density	Number of nonzero a_{ij} in a constraint			b_i			c_j			Number of binding constraints at optimality
	mean	min	mean	max	min	mean	max	min	mean	max	mean
1 – 5	0.00505	2.0	5.1	17.6	-2.5	1.0	4.4	-19.2	-1.0	16.0	836.0
6 – 10	0.00602	2.0	6.0	19.6	-2.4	1.0	4.3	-18.2	-0.9	15.1	834.2
11 – 15	0.00701	2.0	7.0	23.0	-2.2	1.0	4.2	-19.0	-0.9	13.7	827.6
16 – 20	0.00801	2.0	8.0	23.2	-2.0	1.0	4.1	-19.8	-1.0	14.6	812.4
21 – 25	0.00900	2.0	9.0	25.4	-1.8	1.0	3.9	-17.7	-1.1	14.8	803.6
26 – 30	0.01000	2.0	10.0	27.6	-1.8	1.0	3.8	-17.1	-0.9	13.1	807.0
31 – 35	0.02000	4.0	20.0	43.2	-1.6	1.0	3.6	-15.2	-1.0	12.9	754.0
36 – 40	0.03001	8.8	30.0	60.2	-1.3	1.0	3.3	-13.3	-1.0	11.6	723.0
41 – 45	0.04000	15.8	40.0	70.6	-1.3	1.0	3.2	-11.6	-1.1	9.5	694.4
46 – 50	0.04999	22.0	50.0	82.8	-1.2	1.0	3.2	-13.1	-0.9	10.0	696.0
51 – 55	0.06000	29.8	60.0	97.8	-1.0	1.0	3.0	-11.1	-1.0	9.7	659.8
56 – 60	0.07000	38.0	70.0	110.4	-1.1	1.0	3.1	-9.9	-1.0	9.4	664.8
61 – 65	0.08001	43.8	80.0	123.6	-1.0	1.0	2.9	-12.0	-1.0	9.3	647.4
66 – 70	0.08999	53.2	90.0	133.6	-0.9	1.0	2.9	-9.7	-1.0	8.2	630.0
71 – 75	0.10000	61.0	100.0	145.0	-0.9	1.0	2.9	-11.5	-1.0	9.2	633.6
76 – 80	0.20001	146.2	200.0	261.4	-0.8	1.0	2.8	-9.2	-1.0	8.1	578.6
81 – 85	0.30001	236.8	300.0	369.6	-0.6	1.0	2.6	-7.9	-1.0	5.6	546.6
86 – 90	0.40000	332.6	400.0	471.4	-0.6	1.0	2.6	-9.1	-1.0	6.8	530.0
91 – 95	0.50001	429.6	500.0	574.0	-0.7	1.0	2.6	-8.6	-1.0	6.6	514.0
96 – 100	0.75000	688.8	750.0	811.4	-0.6	1.0	2.5	-8.4	-1.0	5.8	472.2
101 – 105	1.00000	999.0	1,000.0	1,000.0	-0.6	1.0	2.6	-6.5	-1.0	5.1	432.6

domly generated LP had a feasible solution, a feasible solution $\mathbf{x}^*$ (not all elements of $\mathbf{x}^*$ were nonzero) was randomly generated to derive random $\mathbf{b}$, where $\mathbf{Ax}^* \leq \mathbf{b}$. Then a feasible solution $\mathbf{y}^*$ (having the same number of nonzero elements as $\mathbf{x}^*$) was randomly generated to derive random $\mathbf{c}$, where $\mathbf{y}^*\mathbf{A} \geq \mathbf{c}$. The ratio of the number of positive and negative elements of $\mathbf{A}$ was one. The number of nonzero a_{ij} in each constraint was binomially distributed $B(n, p = \text{density})$. Additionally, we required each constraint to have at least two nonzero a_{ij}, so that a constraint would not become a simple upper or lower bound on a variable. At each of the 21 densities, 5 random LP were generated. Table 4.3 summarizes the generated LP.

4.5 Computational Results

4.5.1 NNLP

Table 4.4 presents the results for the three CPLEX algorithms for the test problem Set 1. CPU times for NRAD are also shown for comparison. At all densities (ranged from 0.005 to 1), the CPU times for COST NRAD were faster than the CPLEX primal simplex, the CPLEX dual simplex, and the CPLEX barrier linear programming solvers. The average time for NRAD was 3.9 seconds, which was approximately 15 times faster than the fastest CPLEX time of 57.3 seconds by CPLEX primal simplex.

The test results for NNLP set 2 are presented in Table 4.5. CPU times of greater than 2,400 seconds are not reported. The cut-off was based on the approximate sum of the NRAD CPU times over all densities. CPU times for the COST NRAD were faster than the three CPLEX methods again. The average time for NRAD was 78.5 seconds, which was approximately 8 times faster than the fastest CPLEX time of 610.3 seconds by CPLEX primal simplex. NRAD was stable across all densities compared to CPLEX methods. The decrease in NRAD CPU times at densities above 0.02 reflects NRAD's increased discrimination in constraint selection at high density, which is further discussed in Section 4.5.1.3.

4.5.1.1 Influences of the Active-set Approach, COST NRAD, Multi-bound and Multi-cut

The COST NRAD, which incorporates the multi-bound and multi-cut techniques into the active-set approach, achieved significantly fast CPU times in Tables 4.4 and 4.5. The contribution of the NRAD constraint selection metric, active-set approach, and multiple cuts is examined in Tables 4.6 and 4.7.

Table 4.4. Comparison of Computation Times of CPLEX and COST NRAD Methods on NNLP Problem Set 1 (Random NNLP with 1,000 Variables and 200,000 Constraints, $a_{ij} = 1$ to $5, b_i = 1$ to $10, c_j = 1$ to 10)

	CPLEX Primal Simplex	CPLEX Dual Simplex	CPLEX Barrier	NRAD
Presolve	On	On	On	Off
Predual	Auto	Auto	Auto	Off
Density	CPU TIME[†](std. dev.), sec			
0.005056	6.8 (0.2)	50.0 (6.2)	2.7 (0.1)	2.1 (0.1)
0.006023	10.1 (0.8)	58.8 (5.5)	3.2 (0.1)	2.4 (0.1)
0.007014	12.6 (0.9)	87.0 (3.6)	4.2 (0.2)	2.7 (0.1)
0.007999	15.3 (0.8)	99.2 (6.1)	5.2 (0.4)	2.5 (0.1)
0.009007	18.6 (0.6)	113.4 (5.8)	7.3 (0.4)	2.8 (0.2)
0.010000	22.5 (1.1)	124.8 (7.1)	9.8 (0.5)	2.8 (0.2)
0.019991	40.8 (2.0)	195.5 (8.3)	36.9 (4.5)	3.1 (0.2)
0.029987	46.4 (2.3)	221.9 (6.7)	59.0 (6.4)	3.3 (0.4)
0.040024	50.9 (4.1)	238.1 (13.0)	82.6 (7.8)	3.4 (0.3)
0.050009	52.4 (3.9)	251.7 (25.8)	111.8 (15.7)	3.4 (0.3)
0.059995	58.1 (5.4)	237.0 (12.9)	134.4 (19.0)	3.2 (0.3)
0.069983	62.2 (3.4)	241.1 (15.8)	168.9 (17.5)	3.4 (0.2)
0.080011	64.9 (4.1)	255.0 (21.4)	207.7 (42.9)	3.3 (0.3)
0.089992	63.9 (4.0)	246.8 (15.8)	263.2 (62.2)	3.4 (0.4)
0.099989	68.3 (6.2)	277.4 (41.2)	308.3 (60.0)	3.3 (0.3)
0.199977	79.8 (5.9)	289.8 (5.5)	774.1 (83.1)	4.3 (0.4)
0.300018	87.1 (2.8)	339.0 (27.4)	1,547.2 (160.9)	4.9 (0.6)
0.399917	96.1 (3.6)	383.0 (27.0)	2,379.5 (246.4)	5.6 (0.5)
0.499921	97.4 (2.8)	427.9 (29.0)	3,551.3 (240.1)	6.7 (0.4)
0.750024	116.6 (4.6)	407.7 (25.1)	7,184.4 (486.0)	7.9 (1.0)
1.000000	132.4 (13.3)	315.6 (32.4)	10,628.8 (197.5)	8.1 (0.1)
Average (pooled standard deviation)	57.3 (4.5)	231.5 (19.5)	1,308.1 (144.2)	3.9 (0.4)

[†]Average of 5 instances of LPs at each density.

CPU times of CPLEX methods with and without preprocessing, SUB and NRAD are presented in Table 4.6. The data supports the significant role that CPLEX preprocessing plays in reducing computation time. SUB added one constraint per active-set iteration in the order that constraints appeared in the original problem, essentially resulting in a random active-set approach. The application of an active-set approach by itself was sufficient to achieve CPU times faster than some preprocessing-enabled CPLEX methods at high density and some without the presolve at lower density. The computation speed of SUB was quite stable across all densities. NRAD,

Table 4.5. Comparison of Computation Times of CPLEX and COST NRAD Methods on NNLP Problem Set 2 (Random NNLP with 5,000 Variables and 1,000,000 Constraints, $a_{ij} = 1$ to $5, b_i = 1$ to $100, c_j = 1$ to 100)

	CPLEX Primal Simplex	CPLEX Dual Simplex	CPLEX Barrier	NRAD
Presolve	On	On	On	Off
Predual	Auto	Auto	Auto	Off
Density	CPU TIME[†](std. dev.), sec			
0.000508	11.6 (0.6)	13.7 (1.6)	23.6 (1.2)	7.7 (0.2)
0.000574	28.9 (5.1)	30.4 (1.9)	37.3 (3.9)	12.3 (0.3)
0.000650	14.2 (0.9)	99.4 (1.6)	48.5 (0.5)	16.4 (0.6)
0.000733	21.3 (1.8)	169.3 (3.6)	62.6 (2.3)	22.9 (0.7)
0.000822	31.9 (3.6)	249.2 (13.6)	65.8 (3.3)	28.5 (3.2)
0.000914	40.4 (1.8)	335.8 (8.3)	74.5 (3.5)	35.2 (2.0)
0.001009	50.6 (1.6)	420.8 (13.2)	83.8 (3.8)	41.8 (4.3)
0.002000	179.2 (9.9)	1796.3 (100.0)	160.1 (8.5)	96.2 (3.2)
0.003000	249.9 (8.4)	‡	213.9 (14.6)	116.8 (7.9)
0.004000	305.3 (7.0)	‡	221.7 (38.3)	126.8 (8.1)
0.004999	363.2 (10.2)	‡	228.2 (27.7)	128.5 (11.7)
0.006000	423.1 (23.9)	‡	259.2 (35.2)	136.4 (13.3)
0.007001	469.8 (18.8)	‡	338.0 (21.3)	129.4 (7.8)
0.008000	523.2 (23.2)	‡	325.7 (22.6)	125.3 (10.8)
0.009001	560.8 (22.6)	‡	433.2 (47.9)	116.7 (4.2)
0.009999	619.2 (16.5)	‡	483.4 (51.8)	123.0 (12.2)
0.020000	1235.0 (89.4)	‡	1760.8 (409.4)	92.0 (8.4)
0.030000	1699.4 (73.3)	‡	‡	82.5 (3.0)
0.039999	1942.9 (76.7)	‡	‡	71.5 (4.8)
0.050000	1994.1 (197.7)	‡	‡	72.8 (2.0)
0.060002	2053.3 (60.9)	‡	out of memory	65.7 (5.5)
Average (pooled standard deviation)	610.3 (55.5)	n/a	n/a	78.5 (6.8)

[†]Average of 5 instances of LPs at each density.
[‡]Runs with CPU times $> 2,400$ seconds (total CPU time of NRAD for all densities) are not reported.

shown for reference, had the additional advantage of the constraint selection metric and multiple cuts.

In Table 4.7, two active-set methods utilizing SUB and NRAD are presented with various combination of single-cut, multi-bound and multi-cut. Comparing SUB and NRAD, the effect of sorting the constraints by the NRAD metric was about seven to ten-fold reduction in CPU time.

Table 4.6. Comparison of Computation Times to Illustrate the Effects of Applying an Active-set Method on NNLP Problem Set 1 (Random NNLP with 1,000 Variables and 200,000 Constraints, $a_{ij} = 1$ to $5, b_i = 1$ to $10, c_j =$ 1 to 10)

	CPLEX Primal Simplex		CPLEX Dual Simplex		CPLEX Barrier[†]	SUB[‡]	NRAD
Presolve	On	Off	On	Off	On	Off	Off
Predual	Auto	Off	Auto	Off	Auto	Off	Off
Density	CPU TIME[§], sec						
0.005056	6.8	644.7	50.0	818.8	2.7	212.9	2.1
0.006023	10.1	627.0	58.8	1,005.2	3.2	236.2	2.4
0.007014	12.6	629.3	87.0	963.6	4.2	260.1	2.7
0.007999	15.3	647.7	99.2	999.8	5.2	257.3	2.5
0.009007	18.6	677.9	113.4	949.5	7.3	276.3	2.8
0.010000	22.5	660.1	124.8	986.4	9.8	282.6	2.8
0.019991	40.8	601.3	195.5	1,113.7	36.9	287.9	3.1
0.029987	46.4	553.6	221.9	1,126.1	59.0	267.6	3.3
0.040024	50.9	489.2	238.1	921.6	82.6	266.3	3.4
0.050009	52.4	263.5	251.7	504.9	111.8	246.4	3.4
0.059995	58.1	262.4	237.0	433.2	134.4	228.9	3.2
0.069983	62.2	272.7	241.1	386.2	168.9	236.6	3.4
0.080011	64.9	265.7	255.0	366.6	207.7	219.7	3.3
0.089992	63.9	265.5	246.8	366.5	263.2	217.9	3.4
0.099989	68.3	276.6	277.4	362.0	308.3	210.8	3.3
0.199977	79.8	304.6	289.8	346.5	774.1	227.7	4.3
0.300018	87.1	322.7	339.0	323.8	1,547.2	235.0	4.9
0.399917	96.1	373.9	383.0	471.3	2,379.5	260.8	5.6
0.499921	97.4	420.2	427.9	263.6	3,551.3	260.1	6.7
0.750024	116.6	596.7	407.7	202.7	7,184.4	264.3	7.9
1.000000	132.4	1,731.5	315.6	131.1	10,628.8	221.0	8.1
Average	57.3	518.4	231.5	621.1	1,308.1	246.5	3.9

[†]Ran only with presolve on, because it cannot be completely turned off for the CPLEX barrier method.
[‡]One constraint was added per iteration r. $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M = 10^{10}$ was used as the bounding constraint.
[§]Average of 5 instances of LPs at each density.

For low density ($\leq$ 0.03) problems, the introduction of multiple cuts reduced the CPU times approximately 9% and 12% for SUB and RAD respectively. At high density, however, the multi-cut algorithm adds fewer constraints per active-set iteration r. Therefore the difference in CPU times between the single-cut and muti-cut methods gets smaller as the density increases. Further details on the number of constraints added by the methods are given in Section 4.5.1.4. Note that multi-cut SUB gets slower at high density by a much greater amount compared to multi-cut NRAD.

Table 4.7. Comparison of Computation Times to Illustrate the Effects of COST NRAD, Multi-bound and Multi-cut on NNLP Problem Set 1 (Random NNLP with 1,000 Variables and 200,000 Constraints, $a_{ij} = 1$ to $5, b_i = 1$ to $10, c_j = 1$ to 10)

	SUB				NRAD			
bound cut	single[†] single	multi single	single[†] multi	multi multi	single[†] single	multi single	single[†] multi	multi multi
Density	CPU TIME[‡], sec							
0.005056	212.9	216.7	10.5	11.5	25.2	25.2	2.7	2.1
0.006023	236.2	240.3	12.0	11.9	29.0	30.0	3.0	2.4
0.007014	260.1	274.2	12.8	14.1	33.5	33.8	2.9	2.7
0.007999	257.3	269.8	13.0	13.5	37.1	32.5	2.8	2.5
0.009007	276.3	287.7	14.1	14.1	38.7	35.8	3.1	2.8
0.010000	282.6	294.9	14.8	15.0	36.5	36.9	3.1	2.8
0.019991	287.9	283.6	16.4	17.2	38.2	39.2	3.4	3.1
0.029987	267.6	272.5	16.3	18.2	37.9	40.5	3.5	3.3
0.040024	266.3	267.5	16.7	17.7	33.6	34.2	3.4	3.4
0.050009	246.4	246.0	15.9	16.8	32.4	33.4	3.5	3.4
0.059995	228.9	227.6	15.1	15.7	27.9	28.5	3.2	3.2
0.069983	236.6	231.1	15.5	15.8	27.0	27.1	3.4	3.4
0.080011	219.7	216.0	15.2	15.2	24.2	24.2	3.3	3.3
0.089992	217.9	215.2	15.0	15.0	22.4	22.2	3.4	3.4
0.099989	210.8	207.8	14.6	14.6	21.1	21.0	3.5	3.3
0.199977	227.7	229.9	18.0	17.8	15.0	15.1	4.2	4.3
0.300018	235.0	240.2	21.7	21.8	11.3	11.3	4.8	4.9
0.399917	260.8	268.3	28.3	28.6	10.3	10.4	5.2	5.6
0.499921	260.1	267.9	34.1	34.0	8.9	8.9	6.4	6.7
0.750024	264.3	272.0	57.7	57.6	8.5	8.5	8.1	7.9
1.000000	221.0	228.0	227.5	227.1	8.1	8.1	8.1	8.1
Average	246.5	250.3	28.8	29.2	25.1	25.1	4.0	3.9

[†] $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M = 10^{10}$ was used as the bounding constraint.
[‡] Average of 5 instances of LPs at each density.

The observation is attributed to the fact that NRAD's sorting gets more efficient at high density, as explained further in Section 4.5.1.3.

The effect of applying multi-bound is observed in multi-cut NRAD at low density (e.g. 2.7 seconds vs. 2.1 seconds at density 0.005), though the effect is much smaller in magnitude than the other two factors.

4.5.1.2 Effects of Utilizing Factor I, Factor II, and Multiple Cuts

As discussed in Section 3.4.1, the parts of NRAD contributing to Factor I and Factor II are COS and $\frac{\|\mathbf{a}_i\|}{b_i}$ respectively. A metric based on Factor II can be defined

Table 4.8. Comparison of Computation Times of SUB, COS, F_{II}, and NRAD on NNLP Problem Set 1

	SUB		COS		F_{II}		NRAD	
	single cuts[†] [18]	multiple cuts for NNLP	single cuts[†] [28]	multiple cuts for NNLP	single cuts[†] [20]	multiple cuts for NNLP	single cuts[†]	multiple cuts for NNLP
Density	CPU TIME[‡], sec							
0.00506	212.9	11.5	113.9	6.4	28.7	2.4	25.2	2.1
0.00602	236.2	11.9	132.5	7.6	36.0	2.7	29.0	2.4
0.00701	260.1	14.1	130.2	7.7	41.1	3.2	33.5	2.7
0.00800	257.3	13.5	123.9	7.7	40.2	2.9	37.1	2.5
0.00901	276.3	14.1	130.9	8.1	43.7	3.2	38.7	2.8
0.01000	282.6	15.0	137.4	8.4	46.0	3.4	36.5	2.8
0.01999	287.9	17.2	115.9	8.2	49.9	3.8	38.2	3.1
0.02999	267.6	18.2	106.7	7.6	47.3	4.0	37.9	3.3
0.04002	266.3	17.7	90.3	7.0	44.4	4.1	33.6	3.4
0.05001	246.4	16.8	85.9	7.1	41.6	4.1	32.4	3.4
0.06000	228.9	15.7	73.9	6.6	36.1	4.0	27.9	3.2
0.06998	236.6	15.8	74.6	6.7	36.2	4.2	27.0	3.4
0.08001	219.7	15.2	65.3	6.5	31.8	4.0	24.2	3.3
0.08999	217.9	15.0	62.1	6.3	29.5	4.0	22.4	3.4
0.09999	210.8	14.6	62.8	6.6	27.7	4.1	21.1	3.3
0.19998	227.7	17.8	63.8	8.5	19.0	4.8	15.0	4.3
0.30002	235.0	21.8	70.6	11.0	14.6	5.4	11.3	4.9
0.39992	260.8	28.6	76.2	13.7	12.7	5.9	10.3	5.6
0.49992	260.1	34.0	81.0	16.9	10.8	6.8	8.9	6.7
0.75002	264.3	57.6	100.8	28.8	9.4	7.2	8.5	7.9
1.00000	221.0	227.1	102.7	102.6	8.4	8.4	8.1	8.1
Average	246.5	29.2	95.3	13.8	31.2	4.4	25.1	3.9
% of SUB [18]	–	11.8	38.7	5.6	12.7	1.8	10.2	1.6

[†]One constraint was added per iteration r. $\mathbf{c}^{\mathrm{T}}\mathbf{x} \le M = 10^{10}$ was used as the bounding constraint.
[‡]Average of 5 instances of LP at each density. Used CPLEX preprocessing parameters of Presolve = off and Predual = off.

as $F_{II}(\mathbf{a}_i, b_i) = \sum_{j=1}^{n} \frac{a_{ij}}{b_i}$. A single-cut application of F_{II} in an active-set framework was proposed by Myers et al. [20]. CPU times in Table 4.8 show the effects of utilizing Factor I, Factor II, and multiple cuts. Utilizing Factor I reduced the CPU times to 39% of SUB. Utilizing Factor II further reduced the CPU times to 13% of SUB. The application of both Factor I and Factor II achieved the reduction at 10%. Incorporating the multi-cut technique into the four methods further reduced the computation time by 6 to 8 folds compared to single-cut version of each method.

4.5.1.3 Efficient Sorting of Constraints by the COST NRAD

The COST NRAD belongs to *Prior* COST as defined by Sung [3], Corley and Rosenberger [4]. In Prior COST, all constraints are sorted accruing to the constraint selection metric before iteratively solving P_r for $\mathbf{x}_r^*$. Thus NRAD does not depend on $\mathbf{x}_r^*$ and does not need to be recalculated at each active-set iteration r.

Once the test problems were solved, the effectiveness of the prior sorting was evaluated by plotting the distribution of constraints that were binding at optimality against the NRAD sort order, as shown in Figure 4.1. At the highest density of 1, NRAD was most efficient in sorting the likely binding constraints to the top. Binding rows at optimality were found in the top 1.0% of the sorted list of constraints. The distribution becomes wider at lower densities, but binding rows were found in the top 6.5% of the list even in the worst case. The steepness of the curves in Figure 4.1 is notable as well. Across all densities, 95% of binding rows at optimality were found in the top 1.8%, and 99% of binding rows at optimality were found in the top 3.0% of the NRAD sorted constraints in the NNLP problem Set 1.

4.5.1.4 Number of Constraints Added

The active-set methods can also be compared by the number of constraints that methods add. In Table 4.9, the COST NRAD is compared with the constraint selection methods SUB and VIOL of Adler et al. [18] and Zeleny [19], respectively. The CPU times are provided as well for comparison. "Number of Constraints Added" represents the total added from (1.2), which includes constraints in both $BOUNDING$ and $OPERATIVE$ sets. SUB and VIOL used a single bounding constraint as done in previous work [18, 19]. NRAD utilized a multi-bound of Step 1 in the pseudocode.

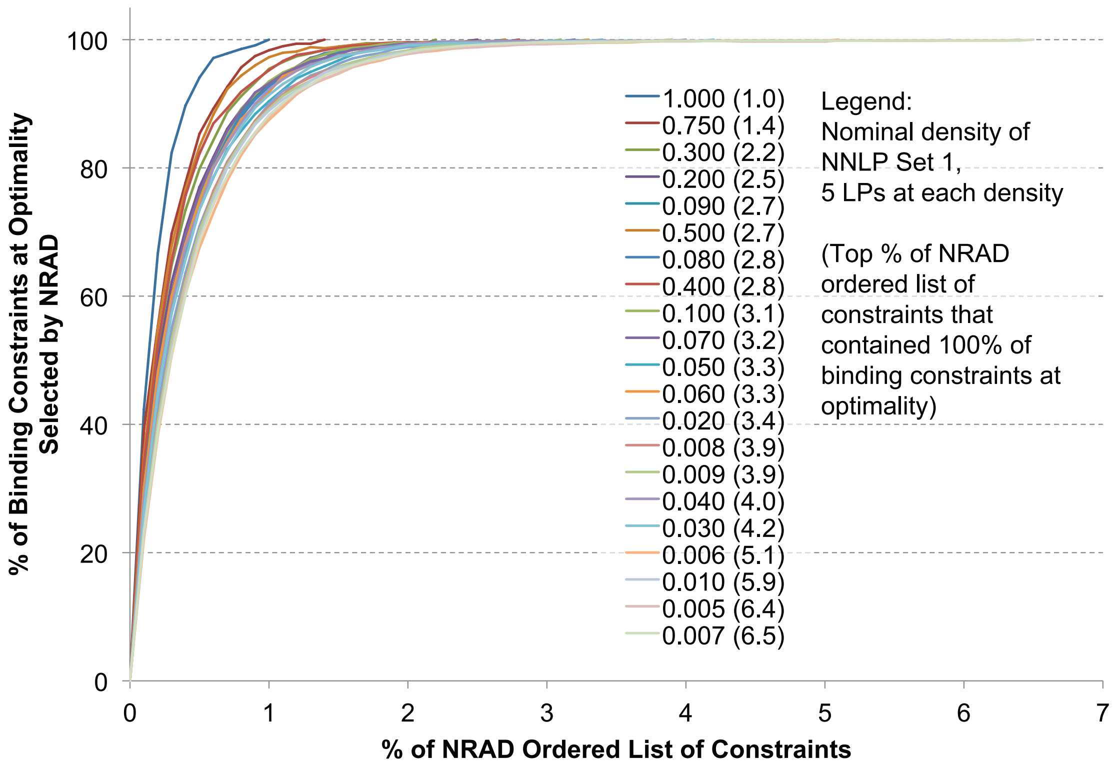

Figure 4.1. Binding constraints at optimality found in NRAD-sorted list of constraints on NNLP problem Set 1 (random NNLP with 1,000 variables and 200,000 constraints, densities 0.005 to 1), average of 5 instances of LPs at each density.

Sorting was implemented by sorting arrays of pointers to the structures holding row information using sort in ANSI C (Section A.5, line 10). Then a new block of memory was allocated to copy the row structure in the order of the NRAD sorted pointers (Section A.5, lines 17–43). This way, the steps in accessing the memory is minimized during the active-set iterations, for example in violation check calculations. For NNLP problem Set 1, the maximum CPU time required to calculate the NRAD values, for the array of pointers, and copy the block of memory were 1.6, 0.1 and 3.3 seconds respectively. Their average times over all densities were 0.3, 0.1, and 0.7 seconds respectively.

Average computation times for SUB and VIOL were 246.5 and 118.5 seconds respectively. They were faster than 518.4 and 621.1 seconds of CPLEX primal simplex and dual simplex without the preprocessing. This shows the efficiency of utilizing an active-set method itself for NNLP.

Among the three active-set methods, the COST NRAD which utilizes multiple cuts had the shortest CPU time by an order of magnitude. Since NRAD adds a group of constraints per iteration r, the total number of calls to the dual simplex algorithm was much lower at 33.2 times on average, when compared to 3818.6 and 942.3 times for SUB and VIOL respectively.

Although VIOL on average added about the same number of constraints as NRAD and added a higher percentage of constraints that were binding at optimality, the computation time was much slower. The reason is that VIOL depends on $\mathbf{x}_r^*$ thus makes use of *posterior* information. The CPU time required to calculate VIOL at each iteration r outweighed the benefit it gained from utilizing the local information $\mathbf{x}_r^*$. NRAD had an advantage as a prior COST.

The last set of columns labeled "% Reduction in # of Original Constraints at Method's Optimal Solution" shows the amount of constraints not added, or effectively eliminated by each method. The three active-set methods did well in this measure. Even the random SUB achieved 98.1% elimination on average.

4.5.1.5 Varying m/n Ratio

The third set of random NNLPs was generated to examine the effect of varying the dimensions of the matrix $\mathbf{A}$. Five problems were generated at each of the 21 densities at m/n ratios of 20, 2, and 1, resulting in additional 315 problem instances.

For all NNLP problems in the third set, i.e. problems with a square or narrow ($m \geq n$) $\mathbf{A}$ matrix, NRAD was faster than any of the CPLEX methods, except

Table 4.9. Comparison of Computation Times of COST NRAD and Non-COST Methods, SUB and VIOL on NNLP Problem Set 1 (Random NNLP with 1,000 Variables and 200,000 Constraints, $a_{ij} = 1$ to $5, b_i = 1$ to $10, c_j = 1$ to 10)

Density	CPU TIME[†], sec			Number of Added Constraints (and number of iterations r for RAD)[†]			% of Constraints Added by Method and also Binding at Optimality[†]			% Reduction in # of Original Constraints at Method's Optimal Solution[†]		
	SUB[‡]	VIOL[‡]	NRAD	SUB[‡]	VIOL[‡]	NRAD	SUB[‡]	VIOL[‡]	NRAD	SUB[‡]	VIOL[‡]	NRAD
0.005056	212.9	43.3	2.1	6,542.2	1,685.6	1,588.4 (7.4)	11.1	43.1	45.7	96.7	99.2	99.2
0.006023	236.2	49.0	2.4	6,385.8	1,647.8	1,551.8 (7.4)	11.2	43.3	45.9	96.8	99.2	99.2
0.007014	260.1	52.4	2.7	6,322.6	1,588.4	1,508.6 (7.0)	10.9	43.3	45.6	96.8	99.2	99.2
0.007999	257.3	51.8	2.5	6,058.6	1,503.0	1,440.2 (7.4)	11.1	44.8	46.7	97.0	99.2	99.3
0.009007	276.3	56.9	2.8	6,111.2	1,498.8	1,424.4 (8.0)	10.9	44.5	46.8	96.9	99.3	99.3
0.010000	282.6	59.1	2.8	5,927.2	1,462.6	1,396.2 (7.8)	11.0	44.6	46.8	97.0	99.3	99.3
0.019991	287.9	67.3	3.1	5,014.4	1,196.8	1,181.8 (9.6)	11.5	48.0	48.7	97.5	99.4	99.4
0.029987	267.6	75.8	3.3	4,405.6	1,078.6	1,074.6 (10.6)	11.9	48.7	48.9	97.8	99.5	99.5
0.040024	266.3	81.2	3.4	4,166.0	983.2	1,010.6 (12.6)	11.7	49.7	48.4	97.9	99.5	99.5
0.050009	246.4	88.3	3.4	3,810.0	928.0	971.4 (13.4)	12.5	51.2	48.9	98.1	99.5	99.5
0.059995	228.9	90.9	3.2	3,558.2	859.4	910.6 (14.0)	12.1	50.3	47.4	98.2	99.6	99.5
0.069983	236.6	99.2	3.4	3,441.2	832.2	886.0 (15.8)	12.2	50.6	47.5	98.3	99.6	99.6
0.080011	219.7	101.4	3.3	3,221.2	780.4	857.8 (16.8)	12.1	50.0	45.5	98.4	99.6	99.6
0.089992	217.9	107.3	3.4	3,107.0	753.0	819.8 (17.6)	12.6	51.9	47.7	98.4	99.6	99.6
0.099989	210.8	110.4	3.3	3,003.8	713.4	802.8 (18.2)	13.1	55.1	49.0	98.5	99.6	99.6
0.199977	227.7	166.9	4.3	2,323.2	569.8	693.2 (27.8)	13.4	54.8	45.0	98.8	99.7	99.7
0.300018	235.0	199.9	4.9	1,925.4	475.4	603.6 (35.0)	13.8	56.0	44.1	99.0	99.8	99.7
0.399917	260.8	227.1	5.6	1,686.2	415.4	556.0 (43.4)	13.7	55.6	41.5	99.2	99.8	99.7
0.499921	260.1	238.6	6.7	1,468.4	354.2	498.4 (50.4)	14.2	58.7	41.7	99.3	99.8	99.8
0.750024	264.3	275.9	7.9	1,042.2	278.0	395.2 (71.8)	15.5	58.3	41.0	99.5	99.9	99.8
1.000000	221.0	245.3	8.1	670.6	185.0	295.6 (294.6)	16.6	60.0	37.6	99.7	99.9	99.9
Average	246.5	118.5	3.9	3,818.6	942.3	974.6 (33.2)	12.5	50.6	45.7	98.1	99.5	99.5

[†]Average of 5 instances of LPs at each density.
[‡]One constraint was added per iteration r [18, 19]. $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M = 10^{10}$ was used as the bounding constraint.

for problems with a ratio of $m/n = 20$ and a density less than or equal to 0.009. In this small fraction of problems, the CPLEX barrier was minimally faster. The overall superior performance of NRAD is apparent across all densities and m/n ratios if plotted as shown in Figure 4.2. For problems P with short-and-wide $\mathbf{A}$ matrix ($m < n$), the dual of P with a corresponding dual version of NRAD can be used to optimize P. The dual form of NRAD is

$$j^* \in \underset{j \notin OPERATIVE}{\arg\min} \left(\frac{\mathbf{a}^{j\mathrm{T}}\mathbf{b}}{c_j} \middle| \mathbf{a}^{j\mathrm{T}}\mathbf{y}_r^* < c_j \right),$$

where $\mathbf{a}^j$ is the j^{th} column of $\mathbf{A}$.

4.5.2 LP

Computational results for the CPLEX primal simplex, dual simplex, and barrier solvers for the general LP set are presented in Table 4.11. CPU times for the COST GRAD with multi-cut, as well as the COST NRAD with multi-bound and multi-cut are shown for comparison. The CPU times for GRAD were faster than the CPLEX primal simplex, the CPLEX dual simplex, and the CPLEX barrier linear programming solvers at densities between 0.02 and 1. Between densities 0.005 and 0.01, CPLEX barrier was up to 4.0 times faster than GRAD. On average, GRAD was 7.0 times faster than NRAD and 14.6 times faster than the fastest CPLEX solver — dual simplex.

4.5.2.1 Influences of the COST GRAD and Multi-cut

In constructing a constraint selection metric for general LPs, a natural strategy might be to have the metric give priority to those constraints with either “as large positive a_{ij} and large positive c_j with small b_i as possible,” or “as small negative a_{ij}

Table 4.10. Comparison of Computation Times of CPLEX and COST NRAD Methods on Random NNLP Problem Set 3, Varying m/n Ratio (Random NNLP with $a_{ij} = 1$ to $5, b_i = 1$ to $10, c_j = 1$ to 10)

	NRAD				CPLEX Primal Simplex[†]				CPLEX Dual Simplex[†]				CPLEX Barrier[†]			
n	1,000	3,163	10,000	14,143	1,000	3,163	10,000	14,143	1,000	3,163	10,000	14,143	1,000	3,163	10,000	14,143
m	200,000	63,246	20,000	14,143	200,000	63,246	20,000	14,143	200,000	63,246	20,000	14,143	200,000	63,246	20,000	14,143
m/n	200	20	2	1	200	20	2	1	200	20	2	1	200	20	2	1
Nominal Density	CPU Time (sec), average of 5 instances of LPs at each density															
0.005	2.1	32.3	124.0	133.9	6.8	73.4	517.5	566.1	50.0	794.5	1,625.3	1,635.3	2.7	24.9	626.3	1,693.9
0.006	2.4	32.1	107.0	132.6	10.1	79.5	477.6	541.2	58.8	878.1	1,572.8	2,077.5	3.2	27.1	631.2	1,771.8
0.007	2.7	33.0	107.1	117.7	12.6	83.1	484.0	548.3	87.0	890.8	1,469.0	1,308.9	4.2	28.2	634.3	1,653.3
0.008	2.5	32.5	98.5	99.4	15.3	80.8	483.4	483.6	99.2	881.0	1,401.7	2,147.7	5.2	32.3	621.2	1,695.7
0.009	2.8	33.5	93.8	91.9	18.6	82.9	463.4	472.8	113.4	932.1	1,336.8	2,031.4	7.3	33.0	625.6	1,679.4
0.01	2.8	32.0	85.0	91.1	22.5	85.0	440.2	478.9	124.8	913.7	1,283.9	2,003.2	9.8	33.8	628.6	1,640.5
0.02	3.1	25.2	54.3	50.9	40.8	84.7	407.5	410.8	195.5	930.4	777.3	763.9	36.9	45.2	687.9	1,780.4
0.03	3.3	21.3	38.7	39.5	46.4	78.0	408.9	437.1	221.9	1,219.0	615.1	609.0	59.0	67.4	730.4	1,802.6
0.04	3.4	19.6	32.6	34.1	50.9	72.2	534.2	499.4	238.1	1,504.6	453.2	978.7	82.6	91.8	817.3	1,922.4
0.05	3.4	17.3	29.3	30.3	52.4	73.4	631.4	594.7	251.7	1,978.1	423.8	577.7	111.8	123.7	897.4	2,065.0
0.06	3.2	16.7	27.0	27.2	58.1	71.5	711.8	692.0	237.0	2,018.3	362.4	534.0	134.4	162.0	971.7	2,294.0
0.07	3.4	14.5	25.7	26.4	62.2	73.2	814.2	882.2	241.1	2,481.6	319.3	439.2	168.9	195.9	1,144.9	2,442.9
0.08	3.3	14.7	22.8	25.1	64.9	72.4	919.9	954.7	255.0	2,855.7	302.4	682.6	207.7	237.0	1,244.6	2,641.7
0.09	3.4	13.0	24.0	22.9	63.9	72.4	1,135.1	814.9	246.8	2,619.2	277.4	348.9	263.2	304.6	1,359.9	2,733.1
0.1	3.3	13.5	21.6	22.4	68.3	75.8	1,163.3	775.8	277.4	2,226.2	252.7	395.1	308.3	343.6	1,470.7	‡
0.2	4.3	10.8	19.5	20.6	79.8	75.9	919.2	620.4	289.8	1,415.1	189.3	352.0	774.1	1,104.5	‡	‡
0.3	4.9	11.1	18.9	19.9	87.1	75.2	831.3	558.7	339.0	1,267.8	148.9	271.1	1,547.2	2,440.0	‡	‡
0.4	5.6	11.4	19.1	19.1	96.1	85.4	793.9	552.3	383.0	1,309.5	139.0	228.8	2,379.5	‡	‡	‡
0.5	6.7	10.8	18.6	19.7	97.4	84.9	772.6	507.9	427.9	1,106.9	142.1	192.1	‡	‡	‡	‡
0.75	7.9	10.7	17.1	18.2	116.6	102.2	654.2	492.9	407.7	1,070.8	141.5	196.2	‡	‡	‡	‡
1	8.1	12.8	17.9	19.9	132.4	111.5	440.7	349.7	315.6	717.1	169.0	195.9	‡	‡	‡	‡
Average	3.9	19.9	47.7	50.6	57.3	80.6	666.9	582.6	231.5	1,429.1	638.2	855.7	n/a	n/a	n/a	n/a

[†]Presolve = ON, Predual = Auto
[‡]Runs with CPU times $> 3,000$ seconds are not reported.

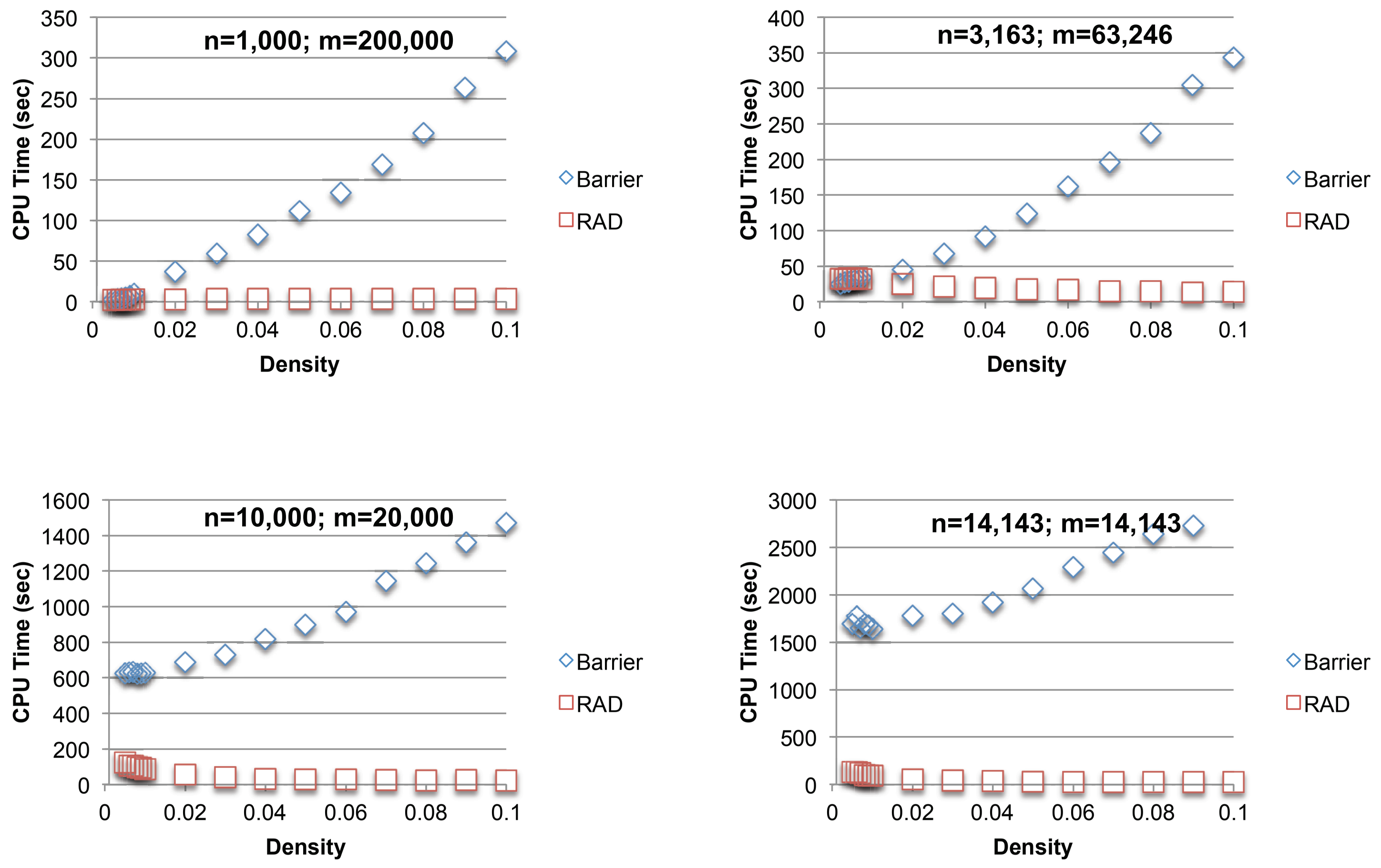

Figure 4.2. Graphical representation of CPU times for CPLEX barrier and COST NRAD presented in Table 4.10

Table 4.11. Comparison of Computation Times of CPLEX and COST GRAD Methods on General LP Problem Set

	CPLEX Primal Simplex	CPLEX Dual Simplex	CPLEX Barrier	GRAD with multiple cuts for LP	NRAD with multiple cuts for NNLP
Presolve	On	On	On	Off	Off
Predual	Auto	Auto	Auto	Off	Off
Density	CPU TIME[†](std. dev.), sec				
0.00505	41.1 (2.3)	21.6 (1.4)	2.4 (0.1)	9.4 (1.8)	15.0 (0.8)
0.00602	86.0 (8.2)	36.6 (1.1)	2.9 (0.1)	11.4 (1.8)	21.5 (4.7)
0.00701	134.8 (7.8)	49.3 (3.2)	4.6 (0.3)	15.3 (0.5)	27.4 (3.9)
0.00801	186.5 (11.1)	65.6 (4.3)	7.8 (0.4)	14.8 (2.3)	31.4 (3.6)
0.00900	215.3 (17.5)	84.8 (9.8)	9.3 (0.3)	13.4 (0.4)	33.9 (5.2)
0.01000	263.2 (18.1)	100.5 (11.9)	11.2 (0.2)	16.4 (1.8)	36.6 (5.3)
0.02000	412.0 (25.2)	223.3 (20.3)	27.4 (1.1)	25.8 (2.0)	69.1 (6.5)
0.03001	503.7 (46.3)	317.4 (28.9)	45.8 (1.3)	26.9 (1.9)	100.7 (8.5)
0.04000	575.2 (40.1)	389.2 (34.0)	64.9 (3.9)	27.6 (0.5)	103.6 (4.8)
0.04999	672.5 (94.3)	427.4 (37.2)	82.9 (3.9)	34.5 (2.2)	132.2 (7.0)
0.06000	718.3 (80.0)	497.9 (38.5)	99.9 (5.5)	32.5 (2.3)	130.4 (4.3)
0.07000	841.0 (119.0)	531.1 (44.2)	125.4 (9.8)	34.5 (0.8)	142.0 (4.7)
0.08001	835.0 (95.8)	570.5 (42.5)	145.0 (14.7)	32.5 (2.4)	150.7 (14.3)
0.08999	930.6 (111.0)	595.8 (50.0)	176.3 (27.0)	34.9 (1.2)	160.4 (10.3)
0.10000	1,029.9 (131.4)	627.8 (21.5)	203.1 (11.0)	36.2 (2.3)	175.2 (12.6)
0.20001	1,667.8 (293.8)	857.8 (68.5)	553.2 (21.2)	50.4 (3.7)	266.4 (18.9)
0.30001	1,939.7 (185.5)	1,016.9 (69.2)	1,097.9 (90.4)	55.8 (5.7)	273.0 (22.1)
0.40000	2,535.9 (857.5)	1,173.3 (117.7)	1,684.0 (124.8)	70.0 (6.4)	403.1 (48.0)
0.50001	2,383.3 (466.3)	1,421.3 (153.3)	2,480.3 (156.1)	79.4 (7.2)	460.1 (31.9)
0.75000	2,727.7 (287.1)	1,775.3 (168.8)	5,026.2 (217.4)	107.2 (13.8)	712.5 (62.9)
1.00000	2,972.7 (397.2)	1,904.8 (203.3)	8,463.6 (950.4)	138.5 (7.4)	2,600.9 (188.1)
Average (pooled standard deviation)	1,032.0 (257.0)	604.2 (78.3)	967.3 (218.3)	41.3 (4.5)	287.9 (45.9)

[†]Average of 5 instances of LP at each density.

and small negative c_j with large b_i as possible." However, when solving LP utilizing NRAD (3.2), the constraint selection metric is

$$\mathrm{NRAD}\left(\mathbf{a}_i, b_i, \mathbf{c}\right) = \sum_{j=1, c_j>0}^{n} \frac{a_{ij}c_j}{b_i} + \sum_{j=1, c_j<0}^{n} \frac{a_{ij}c_j}{b_i}, \tag{4.1}$$

if the terms for $c_j > 0$ and $c_j < 0$ are explicitly written out. The first term follows the general strategy, except when b_i becomes negative. The second term should be subtracted, instead of added, from the first term in order for the constraint selection metric to take a higher value when giving priority to "as small negative a_{ij} and small negative c_j with large b_i as possible." The b_i in the second term should also be positive for the metric to work additively.

To examine the effect of changing the form from NRAD (4.1) to GRAD, several intermediate variations are tested and presented in Table 4.12. The results utilizing SUB are also shown in the table for comparison. The first variation,

$$\mathrm{NRAD}_{variation1}\left(\mathbf{a}_i, b_i, \mathbf{c}\right) = \sum_{j=1, c_j>0}^{n} \frac{a_{ij} c_j}{b_i}, \tag{4.2}$$

is a version only considering the $c_j > 0$ term of (4.1). The test problems of Table 4.3 did not have any constraints with $b_i = 0$, therefore the case was not specially handled. The second version,

$$\mathrm{NRAD}_{variation2}\left(\mathbf{a}_i, b_i, \mathbf{c}\right) = \sum_{j=1, c_j>0}^{n} \frac{a_{ij} c_j}{b_i^+}, \tag{4.3}$$

is (4.2) with b_i^+, which was defined in (3.10). Variation 3,

$$\mathrm{NRAD}_{variation3}\left(\mathbf{a}_i, b_i, \mathbf{c}\right) = \sum_{j=1, c_j>0}^{n} \frac{a_{ij} c_j}{b_i^+} - \sum_{j=1, c_j<0}^{n} \frac{a_{ij} c_j}{b_i^+},$$

incorporates a subtractive term to (4.3). This version was previously described (3.11) but shown here again for clarity. Finally, GRAD is obtained by changing the second term of (3.11) to consider only the intercept, $\frac{b_i}{a_{ij}}$, of the hyperplane $\mathbf{a}_i^{\mathrm{T}}\mathbf{x} = b_i$ and the axis for x_j. For calculation of b_i^+, $\varepsilon = 10^{-10}$ was used for all results presented.

Results for SUB and NRAD from Table 4.12 show that the multi-cut method for LP reduced CPU times by 56 to 57% over the multi-cut method for NNLP, supporting the importance of having both positive and negative a_{ij} for every variable x_j in forming a set of cuts for each iteration of an active-set method for LP. The rest of the comparisons are made with those methods utilizing the multi-cut method for LP.

Table 4.12. Comparison of Computation Times to Illustrate the Effects of Muti-cut, NRAD and GRAD on General LP Problem Set

	Constraint Selection Metric[†]							
	SUB	SUB	NRAD	NRAD	NRAD Variation 1 (4.2)	NRAD Variation 2 (4.3)	NRAD Variation 3 (3.11)	GRAD
	multiple cuts for NNLP	multiple cuts for LP	multiple cuts for NNLP	multiple cuts for LP	multiple cuts for LP			
Density	CPU TIME[‡], sec							
0.00505	16.2	14.5	15.0	13.9	14.0	10.1	11.2	9.4
0.00602	16.8	15.1	21.5	15.7	15.5	10.8	13.0	11.4
0.00701	22.9	17.9	27.4	18.6	22.1	12.4	18.5	15.3
0.00801	30.9	19.3	31.4	21.0	24.2	13.7	20.7	14.8
0.00900	31.2	22.4	33.9	22.4	21.7	14.2	17.9	13.4
0.01000	31.8	28.6	36.6	28.8	25.8	15.4	19.7	16.4
0.02000	73.5	57.0	69.1	53.4	53.3	23.9	34.7	25.8
0.03001	98.2	79.6	100.7	73.5	66.7	27.9	38.3	26.9
0.04000	111.4	83.4	103.6	75.8	76.4	29.6	40.7	27.6
0.04999	144.6	100.1	132.2	87.3	90.1	38.7	49.9	34.5
0.06000	148.5	106.3	130.4	91.7	93.3	35.9	47.1	32.5
0.07000	160.0	109.4	142.0	93.8	108.5	39.6	52.2	34.5
0.08001	179.0	116.6	150.7	98.4	98.5	37.5	49.4	32.5
0.08999	191.0	122.8	160.4	102.8	111.5	38.8	53.2	34.9
0.10000	201.1	133.5	175.2	111.4	112.5	41.6	57.7	36.2
0.20001	312.7	196.1	266.4	166.6	176.6	57.5	83.3	50.4
0.30001	389.0	246.0	273.0	168.5	195.4	59.2	98.2	55.8
0.40000	566.8	344.5	403.1	238.0	254.7	74.6	119.8	70.0
0.50001	734.8	431.0	460.1	284.9	299.5	90.9	136.1	79.4
0.75000	1,158.4	614.8	712.5	379.7	430.7	113.0	186.2	107.2
1.00000	3,662.7	774.6	2,600.9	455.4	522.1	134.8	236.9	138.5
Average	394.4	173.0	287.9	123.9	134.0	43.8	65.9	41.3

[†]Used CPLEX preprocessing parameters of Presolve = off and Predual = off.
[‡]Average of 5 instances of LP at each density.

For densities between 0.005 and 0.09, SUB, NRAD and variation 1 (4.2) of NRAD performed about the same. Introducing the use of b_i^+ in (4.3), (3.9) and (3.11) significantly improved the CPU times over (4.2). Between (3.11) and (4.3), (4.3) which only considered the $c_j > 0$ term was faster. Going from (4.3) to GRAD, including a second term to utilize the intercept term for $c_j < 0$ improved the CPU time slightly more, 5.7% on average.

The COST GRAD with multiple cuts for LP was also tested on NNLP problem Set 1, as shown in Table 4.13. The results confirm that the methods perform equally

Table 4.13. Comparison of Computation Times of NRAD and GRAD on NNLP Problem Set 1

	CPU TIME[†], sec	
Density	COST NRAD	COST GRAD
0.00505	2.1	2.1
0.00602	2.4	2.5
0.00701	2.7	2.7
0.00800	2.5	2.6
0.00900	2.8	2.8
0.01000	2.8	2.8
0.02000	3.1	3.2
0.03000	3.3	3.4
0.04001	3.4	3.5
0.05000	3.4	3.5
0.06000	3.2	3.4
0.06999	3.4	3.6
0.08001	3.3	3.6
0.08999	3.4	3.7
0.10000	3.3	3.7
0.19999	4.3	4.9
0.30001	4.9	5.8
0.40000	5.6	6.9
0.49998	6.7	8.3
0.75001	7.9	10.3
1.00000	8.1	11.5
Average	3.9	4.5

[†]Average of 5 instances of LP at each density. Used CPLEX preprocessing parameters of Presolve = off and Predual = off.

for NNLP. Although the constraint selection metric becomes exactly the same between the two methods when running NNLP, slight increase in CPU times for the COST GRAD occurs because of the time it takes for the algorithm to determine whether the problem is an NNLP (pseudocode in Section 3.5.2, Step 1, lines 1 through 8). In case of solving NNLP with the GRAD, this check allows the multi-cut procedure to stop searching for negative a_{ij} if there are no constraints with negative a_{ij} in $INOPERATIVE$ set.

4.5.2.2 Number of Constraints Added

In Table 4.14, CPU times and the number of constraints added during computation of the test problems by GRAD (both single-cut and multi-cut versions) are compared with the constraint selection methods SUB and VIOL of Adler et al. [18]

and Zeleny [19], respectively. “Number of Constraints Added” reflects the number of constraints added in $OPERATIVE$ set, but not the artificial bounding constraint $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M$. To implement SUB and VIOL as in previous work, a single bounding constraint $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M = 10^{10}$ was used.

Although the single-cut SUB performed comparably with the CPLEX dual simplex with the default preprocessing parameter settings in solving NNLP, SUB is much slower when solving LP. However, as shown above in Table 4.12, the CPU times for SUB greatly improves to 173.0 seconds from 4515.6 seconds on average, faster than 604.2 seconds for the CPLEX dual simplex, once the multi-cut procedure is incorporated.

For the NNLP Set 1, the 25.1 seconds of the single-cut version of NRAD was faster than 118.5 second for VIOL on average, whereas for the general LP set, the 615.7 seconds of the single-cut version of GRAD was slower than 525.1 seconds for VIOL on average. However the COST GRAD, which incorporates multi-cut, outperformed VIOL with multi-cut. The respective times were compared at 41.3 seconds vs. 82.9 seconds on average.

In general, a method that makes use of *posterior* information such as VIOL adds fewer constraints and thus adds a higher percentage of binding constraints at optimality. But this comes at a cost of extra computation time required to rank the set of inoperative constraints at every iteration r. The data in Table 4.14 confirmed that single-cut VIOL added the fewest number of constraints (2,012 on average). The advantage of not re-sorting the constraints at every r for a *prior* method, i.e. GRAD, became apparent when multi-cut is applied. In multi-cut VIOL, violating inoperative constraints had to be re-sorted in descending order of violation at every iteration r.

Comparing the CPU times with and without the multiple cuts, the reduction in CPU times was greater for GRAD than in NRAD. For NRAD, the reduction was about six-fold (from 25.1 to 3.9 seconds) on average. The CPU times for GRAD reduced about 14-fold (from 615.7 seconds to 41.3 seconds).

Table 4.14. Comparison of Computation Times of COST GRAD and Non-COST Methods, SUB and VIOL on General LP Problem Set

Density	CPU TIME[†], sec						Number of Added Constraints (and number of iterations r for multi-cut methods)[†]						% of Constraints Added by Method and also Binding at Optimality[†]					
	SUB		VIOL		GRAD		SUB		VIOL		GRAD		SUB		VIOL		GRAD	
	SC[‡]	MC[‖]	SC[§]	MC[‖]	SC[¶]	MC[‖]	SC[‡]	MC[‖]	SC[§]	MC[‖]	SC[¶]	MC[‖]	SC[‡]	MC[‖]	SC[§]	MC[‖]	SC[¶]	MC[‖]
0.00505	246.0	14.5	120.1	15.1	149.1	9.4	8,041	7,363 (12.8)	2,911	7,002 (11.8)	6,133	6,129 (10.8)	10.4	11.4	28.7	11.9	13.6	13.6
0.00602	324.8	15.1	145.8	17.6	188.0	11.4	8,154	7,264 (12.8)	2,904	6,751 (12.4)	6,094	5,801 (11.0)	10.2	11.5	28.7	12.4	13.7	14.4
0.00701	408.7	17.9	166.7	18.5	223.4	15.3	8,205	7,193 (13.6)	2,834	6,393 (12.8)	6,014	5,638 (11.6)	10.1	11.5	29.2	12.9	13.8	14.7
0.00801	458.6	19.3	179.7	20.9	256.1	14.8	8,165	7,068 (14.2)	2,767	6,144 (12.4)	5,985	5,514 (11.4)	10.0	11.5	29.4	13.2	13.6	14.7
0.00900	545.4	22.4	193.4	23.0	296.0	13.4	8,129	7,001 (15.0)	2,685	6,031 (13.6)	5,845	5,387 (12.0)	9.9	11.5	29.9	13.3	13.7	14.9
0.01000	628.5	28.6	217.4	24.3	299.6	16.4	8,228	7,009 (16.0)	2,705	5,725 (13.4)	5,726	5,143 (12.0)	9.8	11.5	29.8	14.1	14.1	15.7
0.02000	1,463	57.0	302.2	31.5	491.9	25.8	7,974	6,597 (21.8)	2,341	4,423 (15.0)	5,208	4,465 (15.4)	9.5	11.4	32.2	17.0	14.5	16.9
0.03001	2,521	79.6	356.8	37.2	536.2	26.9	7,570	6,237 (26.4)	2,163	3,875 (17.4)	4,557	3,826 (16.6)	9.6	11.6	33.4	18.7	15.9	18.9
0.04000	2,552	83.4	346.9	36.7	600.2	27.6	7,347	6,024 (30.8)	2,055	3,496 (18.6)	4,386	3,685 (20.4)	9.5	11.5	33.8	19.9	15.8	18.8
0.04999	3,344	100.1	400.1	41.4	724.3	34.5	7,354	6,112 (36.8)	2,009	3,318 (20.8)	4,317	3,634 (22.4)	9.5	11.4	34.6	21.0	16.1	19.2
0.06000	3,346	106.3	378.5	38.8	644.2	32.5	7,067	5,864 (40.2)	1,860	3,028 (21.6)	3,949	3,326 (24.0)	9.3	11.3	35.5	21.8	16.7	19.8
0.07000	4,480	109.4	414.9	41.0	654.1	34.5	6,923	5,780 (44.4)	1,843	2,905 (23.2)	3,927	3,323 (26.0)	9.6	11.5	36.1	22.9	16.9	20.0
0.08001	4,357	116.6	426.4	40.1	708.3	32.5	6,851	5,737 (48.6)	1,803	2,724 (24.0)	3,600	3,035 (26.4)	9.5	11.3	35.9	23.8	18.0	21.3
0.08999	4,448	122.8	428.1	42.3	595.8	34.9	6,727	5,658 (52.8)	1,743	2,636 (25.6)	3,555	3,029 (28.8)	9.4	11.1	36.1	23.9	17.7	20.8
0.10000	4,379	133.5	449.2	45.1	643.3	36.2	6,730	5,681 (58.2)	1,734	2,563 (26.8)	3,500	2,992 (30.8)	9.4	11.2	36.5	24.7	18.1	21.2
0.20001	7,666	196.1	650.5	69.4	758.9	50.4	6,226	5,415 (97.2)	1,561	2,089 (37.8)	3,145	2,742 (50.0)	9.3	10.7	37.1	27.7	18.4	21.1
0.30001	7,499	246.0	758.1	93.8	806.6	55.8	5,814	5,137 (131.8)	1,393	1,775 (45.8)	2,787	2,467 (63.6)	9.4	10.6	39.2	30.8	19.6	22.2
0.40000	10,947	344.5	947.9	134.7	1,020	70.0	5,814	5,248 (176.6)	1,359	1,684 (55.6)	2,641	2,381 (79.6)	9.1	10.1	39.0	31.5	20.1	22.3
0.50001	11,929	431.0	1,104	178.6	1,050	79.4	5,764	5,225 (217.2)	1,318	1,573 (64.0)	2,482	2,264 (93.6)	8.9	9.8	39.0	32.7	20.7	22.7
0.75000	12,697	614.8	1,420	313.4	1,100	107.2	5,386	4,988 (315.2)	1,192	1,387 (83.6)	2,201	2,030 (127.4)	8.8	9.5	39.6	34.0	21.4	23.3
1.00000	10,585	774.6	1,621	478.0	1,183	138.5	5,055	4,725 (415.6)	1,070	1,245 (102.0)	2,040	1,913 (168.4)	8.6	9.2	40.4	34.8	21.2	22.6
Average	4,516	173.0	525.1	82.9	615.7	41.3	7,025	6,063 (85.6)	2,012	3,656 (31.3)	4,195	3,749 (41.1)	9.6	11.1	33.4	18.4	16.0	17.9

[†] Average of 5 instances of LP at each density.
[‡] One constraint was added per iteration r [18]. $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M = 10^{10}$ was used as the bounding constraint.
[‖] Multi-cut technique for LP was applied with $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M = 10^{10}$ as the bounding constraint.
[§] One constraint was added per iteration r [19]. $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M = 10^{10}$ was used as the bounding constraint.
[¶] One constraint was added per iteration r. $\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq M = 10^{10}$ was used as the bounding constraint.

CHAPTER 5
CONCLUSIONS

An efficient COST NRAD with multi-bound and multi-cut for NNLPs was developed, then generalized to a COST GRAD with multi-cut for LPs. A geometric interpretation of Factor I, Factor II, NRAD and multi-cut was given. An intuition behind the development of GRAD was also described.

The COST NRAD was tested on three sets of randomly generated large-scale NNLP problems. The results showed that the COST NRAD performed significantly better than the CPLEX primal simplex, dual simplex, and barrier solvers for NNLPs (maximization) with long-and-narrow **A** matrices with various densities, from very sparse to 1.

The COST GRAD was tested on a set of randomly generated large-scale general LP problems with $m \gg n$. For densities between 0.02 and 1, GRAD outperformed the CPLEX primal simplex, dual simplex and barrier solvers for LP (maximization) with long-and-narrow **A** matrices. The computational results showed that GRAD was not as effective as NRAD in selecting constraints that are likely to be binding at optimality. Further research may improve GRAD towards the efficiency that NRAD demonstrated with NNLP. Incorporating new techniques may also be of interest. In particular, incorporating a method to better approximate the feasible region for general LP, as well as simultaneously addressing both the primal and dual problems, could conceivably improve COST GRAD by adding both constraints and variables.

While many methods for solving large-scale LP involve preliminary problem reduction, the COST framework builds up the problem. Moreover, the COST NRAD and COST GRAD maintain the attractive features of the dual simplex method such as a basis, shadow prices, reduced costs, and sensitivity analysis [16]. They also retain the post-optimality advantages of pivoting algorithms useful for integer programming. As a practical matter, a definition and usage of the COSTs NRAD and GRAD are well described here and not proprietary, unlike the CPLEX preprocessing routines.

Furthermore, the utilization of local posterior information [4] obtained from each $\mathbf{x}_r^*$ in addition to the global GRAD information for constraints obtained prior to the active-set iterations is another area of exploration. It is conceivable that the rationale behind NRAD and GRAD could also lead to better integer programming cutting planes. Finally, it should be noted that a COST is a polynomial algorithm if the CPLEX barrier solver is used to solve each new subproblem with added constraints instead of the primal simplex or the dual simplex.

APPENDIX A
CODE EXAMPLE FOR THE COST NRAD

In this appendix, an example of ANSI C code for implementing the COST NRAD utilizing the CPLEX callable library is presented.

A.1 Definitions

The section contains definitions such as the headers to be loaded and the row structure.

```c
#include <cplex.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
#include <string.h>

#define PRINTDETAILS 0

int PREIND; //CPLEX presolve switch
int PREDUAL; //CPLEX presolve dual setting

double totallstime = 0;
double radsorttime = 0;
double radmemcopytime = 0;
double radcalctime = 0;

struct row {
  int nzcnt;
  int rmatbeg;
  int * rmatind;
  double * rmatval;
  int rmatspace;
  int surplus;
  double rhs;
  char sense;
  double rad;
  double criteria;
  double violation;
  int used;
};

struct row * myRowsRAD;
double *ub;
double *lb;
double *obj;

int PrepMyRows (char *, int *, int *, int *, int *);
int SortCopyRows(struct row **, struct row *, int, double *, double *);
int CostSolver (char *,struct row *, double *, double *, double *, int, int, int, int
    );
int compar(const void *, const void *);
```

A.2 Main

```c
int main (int argc, char * argv[])
{
```

```
clock_t ticks1, ticks99;

struct tm * timeinfo;
char currenttime[50];

char filenamewithpath[250] = "";

int i;
int j;
int k;
int nrows;
int ncols;
int objsense;
int nzcnt;

if (argc == 2) {
  sprintf(filenamewithpath, "%s", argv[1]);

  // turn off CPLEX preprocessing presolve
  PREIND = CPX_OFF;

  // turn off CPLEX preprocessing predual
  PREDUAL = -1;

}
else {
  printf("missing arguments\n");
  printf("arg[1] = lp file name\n");
  exit(1);
}

time ( &ticks1 );

FILE * fp1 = fopen("output.txt","a");
FILE * fp3 = fopen("taboutput.txt","r");
int fileexists;
if (fp3 == NULL) {
  fileexists = 0;
}
else {
  fileexists = 1;
  fclose(fp3);
}
FILE * fp2 = fopen("taboutput.txt","a");
printf ("\n=================================================\n");
printf ("=================================================\n");
fprintf (fp1, "\n=================================================\n");
fprintf (fp1, "=================================================\n");
printf ( "Run time is: %s", ctime (&ticks1));
fprintf (fp1, "Run time is: %s", ctime (&ticks1));
// write header line for taboutput
if (!fileexists) {
  fprintf (fp2, "start time\t");
  fprintf (fp2, "file name\t");
  fprintf (fp2, "opt method\t");
  fprintf (fp2, "RAD calc time\t");
  fprintf (fp2, "sort time\t");
  fprintf (fp2, "mem copy time\t");
  fprintf (fp2, "add constraints-to-bound time\t");
  fprintf (fp2, "number of constraints-to-bound\t");
  fprintf (fp2, "opt method used to solve bounded problem\t");
  fprintf (fp2, "opt time to solve bounded problem\t");
  fprintf (fp2, "1st Opt niter\t");
```

```
    fprintf (fp2, "Status\t");
    fprintf (fp2, "presolve indicator\t");
    fprintf (fp2, "preprocess dual indicator\t");
    fprintf (fp2, "LS time\t");
    fprintf (fp2, "number of added constraints\t");
    fprintf (fp2, "number of evaluations\t");
    fprintf (fp2, "total iterations\t");
    fprintf (fp2, "total phase 1 iterations\t");
    fprintf (fp2, "Number of COST iterations\t");
    fprintf (fp2, "objective\t");
    fprintf (fp2, "CPU time (total)[sec]\t");
    if (PRINTDETAILS) {
      fprintf (fp2, "Total CPU Time COST Simplex\t");
      fprintf (fp2, "Total CPU Time calc add sort rows\t");
    }
    fprintf (fp2, "end time\t");
    if (PRINTDETAILS) {
      fprintf (fp2, "COST opt iter num\t");
      fprintf (fp2, "CPU Time of COST simplex\t");
      fprintf (fp2, "CPU Time to calc sort add rows\t");
      fprintf (fp2, "Number of violations\t");
      fprintf (fp2, "Number of rows added per COST iter\t");
      fprintf (fp2, "Index i of the Last row added in cover x\t");
    }
    fprintf (fp2, "\n");
  }

  PrepMyRows(filenamewithpath, &nrows, &ncols, &objsense, &nzcnt);

  time (&ticks99);
  timeinfo = localtime (&ticks99);
  strftime (currenttime,50,"%c", timeinfo);
  printf ("\n%s\n", currenttime);
  fprintf (fp1, "\n%s\n", currenttime);
  fprintf (fp2, "%s\t", currenttime);
  printf("LP Problem (Primal) is: %s \n",filenamewithpath);
  fprintf(fp1, "LP Problem (Primal) is: %s \n",filenamewithpath);
  fprintf(fp2, "%s\t",filenamewithpath);

  printf("RAD ");
  fprintf(fp1, "RAD ");
  fprintf(fp2, "RAD\t");

  printf("Calc RAD (sec) %g ", radcalctime);
  fprintf(fp1, "Calc RAD (sec) %g ", radcalctime);
  fprintf(fp2, "%g\t",radcalctime);

  printf("Sort RAD (sec) %g ", radsorttime);
  fprintf(fp1, "Sort RAD (sec) %g ", radsorttime);
  fprintf(fp2, "%g\t",radsorttime);

  printf("Mem Copy RAD (sec) %g ", radmemcopytime);
  fprintf(fp1, "Mem Copy RAD (sec) %g ", radmemcopytime);
  fprintf(fp2, "%g\t",radmemcopytime);

  totallstime = radcalctime + radsorttime + radmemcopytime;

  fclose(fp1);
  fclose(fp2);
  CostSolver (filenamewithpath, myRowsRAD, ub, lb, obj, objsense, nrows, ncols, nzcnt
      );


  for (j = 0; j < nrows; j++) {
    free(myRowsRAD[j].rmatind);
    free(myRowsRAD[j].rmatval);
  }
  free(myRowsRAD);
  free(ub);
  free(lb);
  free(obj);

  fp1 = fopen("output.txt","a");
  printf ("\n==========================================\n");
  fprintf (fp1, "\n==========================================\n");
  fclose(fp1);

  return 0;
} /* end of Main */
```

A.3 PrepMyRows

The following procedure reads the LP problem into the row structure, calculates the constraint selection metric, sorts the pointers to rows then copies the sorted rows into a block memory.

```
int PrepMyRows
(char * lpfilename, int * nrows, int * ncols, int * objsense, int * nzcnt)
{
  int i;
  int j;
  double sum;
  double a_size;
  double c_size;
  double intercept;
  int status;
  clock_t ticks1, ticks2;
  double timediff, sorttime, memcopytime;

  CPXCENVptr myEnv;
  CPXLPptr myLP;

  /* variables for printing run info */
  int * param1;
  int * param2;
  struct tm * timeinfo;
  char currenttime[50];

  int *rmatbeg;
  int *rmatind;
  double *rmatval;
  int surplus;
  int rmatspace = 0;
  double *rhs;
  char *sense;

  // ************************
  // Start COSTs Methods Here
  // ************************
  myEnv = CPXopenCPLEX(&status);
  myLP = CPXcreateprob(myEnv, &status, "TestProb");
  status = CPXreadcopyprob(myEnv, myLP, lpfilename, "LP");
```

```
*nrows = CPXgetnumrows (myEnv, myLP);
*ncols = CPXgetnumcols (myEnv, myLP);
rmatbeg = (int *) calloc(*nrows, sizeof(int));
rhs = (double *) calloc(*nrows, sizeof(double));
sense = (char *) calloc(*nrows, sizeof(char));
obj = (double *) calloc(*ncols, sizeof(double));
ub = (double *) calloc(*ncols, sizeof(double));
lb = (double *) calloc(*ncols, sizeof(double));

/* Read the row structure in */
/* This two-step read is quicker and what is recommended by CPLEX */
/** First Read with rmatspace = 0, then surplus will equal to
 the negative of needed space **/
status
= CPXgetrows (myEnv, myLP, nzcnt, rmatbeg, NULL, NULL, rmatspace,
              &surplus, 0, *nrows-1);

rmatspace = -surplus;
rmatind = (int *) calloc(-surplus, sizeof(int));
rmatval = (double *) calloc(-surplus, sizeof(double));

/** Second, Read with the correct rmatspace = -surplus **/
status
= CPXgetrows (myEnv, myLP, nzcnt, rmatbeg, rmatind, rmatval, rmatspace,
              &surplus, 0, *nrows-1);

//READ right hand sides of the constraints
status = CPXgetrhs (myEnv, myLP, rhs, 0, *nrows-1);

//READ sense of the constraints
status = CPXgetsense (myEnv, myLP, sense, 0, *nrows-1);

struct row ** myRows_p = (struct row **) calloc(*nrows, sizeof(struct row *));
myRowsRAD = (struct row *) calloc(*nrows, sizeof(struct row));

for (i = 0; i < *nrows; i++) {
  myRows_p[i] = (struct row *) malloc(sizeof(struct row));
}
status = CPXgetub (myEnv, myLP, ub, 0, *ncols-1);
status = CPXgetlb (myEnv, myLP, lb, 0, *ncols-1);
status = CPXgetobj (myEnv, myLP, obj, 0, *ncols-1);
*objsense = CPXgetobjsen (myEnv, myLP);

// ********************************
// ********************************
// ******** Set up myRows *********
// ********************************
// ********************************
for (i = 0; i < *nrows; i++) {
  myRows_p[i]->rmatspace = *nzcnt;

  // Read rmatbeg into myRows
  myRows_p[i]->rmatbeg = rmatbeg[i];
  // Calculate nzcnt for each row and save in myRows
  if (i < *nrows-1) {
    // case for all rows except for the last row
    myRows_p[i]->nzcnt = rmatbeg[i+1] - rmatbeg[i];
  }
  else {
    // case for the last row
    myRows_p[i]->nzcnt = *nzcnt - rmatbeg[i];
  }

  // Allocate memeory for pointers to row matrix index and
```

```
    // row matrix values in myRows structure
    // using the non-zero count that was just calculated
    myRows_p[i]->rmatind
    = (int *) calloc(myRows_p[i]->nzcnt, sizeof(int));
    myRows_p[i]->rmatval
    = (double *) calloc(myRows_p[i]->nzcnt, sizeof(double));

    // Read the Row Matrix Index and
    // Row Matrix Values for each row
    for (j = 0; j < myRows_p[i]->nzcnt; j++) {
      myRows_p[i]->rmatind[j] = rmatind[j+rmatbeg[i]];
      myRows_p[i]->rmatval[j] = rmatval[j+rmatbeg[i]];
    }
    status = CPXgetrhs (myEnv, myLP, &(myRows_p[i]->rhs), i, i);
    status = CPXgetsense (myEnv, myLP, &(myRows_p[i]->sense), i, i);
  }

  CPXfreeprob(myEnv, &myLP);
  free(rmatbeg);
  free(rhs);
  free(sense);
  free(rmatind);
  free(rmatval);

  // *****************************
  // This prepares RAD as criteria
  // *****************************

  /****************** Clock Start ******************/
  ticks1=clock();

  if (*objsense == CPX_MAX) {
    for (i = 0; i < *nrows; i++) {
      sum = 0;

      for (j = 0; j < myRows_p[i]->nzcnt; j++) {
        // calculates ac
        sum += (double) myRows_p[i]->rmatval[j] * obj[myRows_p[i]->rmatind[j]];
      }
      // calculates ac/b
      myRows_p[i]->rad = (sum)/myRows_p[i]->rhs;
      myRows_p[i]->criteria = -myRows_p[i]->rad;

      if (myRows_p[i]->sense == 'E')
        myRows_p[i]->criteria -= 1000;
    }
  }
  else if (*objsense == CPX_MIN) {
    for (i = 0; i < *nrows; i++) {
      sum = 0;

      for (j = 0; j < myRows_p[i]->nzcnt; j++) {
        // calculates a*(-1)c = -ac
        sum += (double) myRows_p[i]->rmatval[j] * (-1) * obj[myRows_p[i]->rmatind[j
            ]];
      }
      // calculates -ac/b
      myRows_p[i]->rad = (sum)/myRows_p[i]->rhs;
      myRows_p[i]->criteria = -myRows_p[i]->rad;

      if (myRows_p[i]->sense == 'E')
        myRows_p[i]->criteria -= 1000;
    }
  }
  ticks2=clock();
  /****************** Clock End ******************/

  timediff = ((double)ticks2 -ticks1)/CLOCKS_PER_SEC;
  radcalctime = timediff;
  SortCopyRows(myRows_p, myRowsRAD, *nrows, &sorttime, &memcopytime);
  radsorttime = sorttime;
  radmemcopytime = memcopytime;

  for (i = 0; i < *nrows; i++) {
    free(myRows_p[i]->rmatind);
    free(myRows_p[i]->rmatval);
    free(myRows_p[i]);
  }
  free(myRows_p);

  return 0;
}
```

A.4 CostSolver

The code in this section applies the active-set method of the COST utilizing the multi-bound and multi-cut techniques on the sorted rows of constraints.

```
int CostSolver (char * lpfilename,
                struct row * myRows, double * ub, double * lb,
                double * obj, int objsense, int nrows, int ncols, int nzcnt)
{
  int status;
  int * param1;
  int * param2;
  clock_t ticks1, ticks2, // for clocking entire LSSolver
  ticks3, ticks4, // for clocking Simplex
  ticks5, ticks6, // for clocking violation check evaluation
  ticks7, ticks8, // for clocking adding constraint after violation check
  ticks9, ticks10, // for clocking addition of bounding constraints
  ticks11, ticks12, // for clocking multi-row criteria calc sort and add
  ticks13, ticks14, // not used now
  ticks99; // for getting the system time for file name creation
  double timediff;
  double timediffopt;
  double timediff1stopt;
  double timediffeval;
  double timediffaddconstraint;
  double timediffaddboundingconstraints;
  double timediffcalcsortadd;
  double timediffcalcsortaddtotal = 0;
  double timediffcostsimplex;
  double timediffcostsimplextotal = 0;

  struct tm * timeinfo;
  char currenttime[50];
  char currentyymmddHHMMSS[50] = "";

  int i;
  int j;
  int k;
  double objval;
  int rmatbeg = 0;
  double *x = (double *) calloc(ncols, sizeof(double));
```

```

int optimized = 0;
int optimizedsub = 0;
int violationfound = 0;
int addedconstraints = 0;
int newcoli = -1;
int nrowsaddedtocoverx;
int lastrowadded;
int nviolations;
int ncostopt = 0;
long int niter1stopt = 0;
long int phase1cnt1stopt = 0;
long int nevals = 0;
long int niter = 0;
long int nitertotal = 0;
long int phase1cnt = 0;
long int phase1cnttotal = 0;
int method1stopt;

// used for bounding variables
int * variableisbound = (int *) calloc(ncols, sizeof(int));
int boundvariablecount = 0;
int addthisrow = 0;
int numrowsadded = 0;

// used during violation check
double lhs;

/* variables for log file writing */
FILE * fp1 = fopen("output.txt","a");
FILE * fp2 = fopen("taboutput.txt","a");

// for sorting sub LP
struct row ** mySubRows_p = (struct row **) calloc(nrows, sizeof(struct row *));

CPXCENVptr myEnv = CPXopenCPLEX(&status);
CPXLPptr myLP = CPXcreateprob(myEnv, &status, "MCOSTProb");

status = CPXsetintparam(myEnv, CPX_PARAM_PREIND, PREIND);
status = CPXsetintparam(myEnv, CPX_PARAM_PREDUAL, PREDUAL);

/****************** Clock Start ******************/
ticks1 = clock();

status = CPXnewcols (myEnv, myLP, ncols, obj, lb, ub, NULL, NULL);
CPXchgobjsen (myEnv, myLP, objsense);
for (i = 0; i < nrows; i++) myRows[i].used = 0;

ticks9=clock();

status = CPXgetintparam(myEnv, CPX_PARAM_PREIND, &param1);
status = CPXgetintparam(myEnv, CPX_PARAM_PREDUAL, &param2);
for (i = 0; i < ncols; i++) variableisbound[i] = 0;
for (i = 0; i < nrows && boundvariablecount < ncols; i++) {
  for (k = 0; k < myRows[i].nzcnt; k++) {
    if (variableisbound[myRows[i].rmatind[k]] == 0) {
      variableisbound[myRows[i].rmatind[k]] = 1;
      boundvariablecount++;
      addthisrow = 1;
    }
  }// for k
  if (addthisrow) {
    status = CPXaddrows (myEnv, myLP, 0, 1, myRows[i].nzcnt,
                         &(myRows[i].rhs),
                           &(myRows[i].sense),
                           &rmatbeg, myRows[i].rmatind,
                           myRows[i].rmatval, NULL, NULL);
    myRows[i].used = 1;
    numrowsadded++;
    if (PRINTDETAILS) lastrowadded = i;
  }//if addthisrow
  addthisrow = 0;
}//for i

ticks10 = clock();

timediffaddboundingconstraints = ((double)ticks10 -ticks9)/CLOCKS_PER_SEC;
printf(
       "\nTime bounding constraints add (sec) %g num constraints needed to bound %d
           ",
       timediffaddboundingconstraints, numrowsadded);
fprintf(fp1,
        "\nTime bounding constraints add (sec) %g num constraints needed to bound %
            d",
        timediffaddboundingconstraints, numrowsadded);
fprintf(fp2,
        "%g\t%d\t",
        timediffaddboundingconstraints, numrowsadded);

while (!optimized) {
  // first iteration is the solving of initial bounded problem
  // clock, use dual simplex
  if (addedconstraints == 0) {

    ticks3=clock();

    // solve the initial bounded problem using primal simplex
    status = CPXprimopt (myEnv, myLP);

    ticks4=clock();

    timediff1stopt = ((double)ticks4 -ticks3)/CLOCKS_PER_SEC;
    timediffcostsimplex = timediff1stopt;
    niter1stopt = CPXgetitcnt(myEnv, myLP);
    phase1cnt1stopt = CPXgetphase1cnt(myEnv, myLP);
    method1stopt = CPXgetmethod (myEnv, myLP);

    switch (method1stopt) {
      case CPX_ALG_NONE:
        printf("\n1st Opt Method=None ");
        fprintf(fp1, "\n1st Opt Method=None ");
        fprintf(fp2, "None\t");
        break;
      case CPX_ALG_PRIMAL:
        printf("\n1st Opt Method=Primal ");
        fprintf(fp1, "\n1st Opt Method=Primal ");
        fprintf(fp2, "Primal\t");
        break;
      case CPX_ALG_DUAL:
        printf("\n1st Opt Method=Dual ");
        fprintf(fp1, "\n1st Opt Method=Dual ");
        fprintf(fp2, "Dual\t");
        break;
      case CPX_ALG_BARRIER:
        printf("\n1st Opt Method=Barrier  ");
        fprintf(fp1, "\n1st Opt Method=Barrier  ");
        fprintf(fp2, "Barrier\t");
        break;
    }
    printf(
           "1st Opt Time (sec) %g 1st Opt niter %d\n", \
           timediff1stopt, niter1stopt);
    printf(
           "PREIND %d PREDUAL %d\n", param1, param2);
    fprintf(fp1,
            "1st Opt Time (sec) %g 1st Opt niter %d\n", \
            timediff1stopt, niter1stopt);
    fprintf(fp1,
            "PREIND %d PREDUAL %d\n", param1, param2);
    fprintf(fp2,
            "%g\t%d\t\t%d\t%d\t",
            timediff1stopt, niter1stopt, param1, param2);
    if (PRINTDETAILS) {
      fprintf(fp2, "\n");
      for (k = 0; k < 30; k++) fprintf(fp2, "\t");
      fprintf(fp2, "-1\t\t\t\t%d\t%d\t\n", numrowsadded, lastrowadded);
    }
  } // if added constraints is 0

  status = CPXgetx (myEnv, myLP, x, 0, ncols-1);

  optimized = 1;
  if (PRINTDETAILS) {
    nrowsaddedtocoverx = 0;
    lastrowadded = 0;
    nviolations = 0;
  }

  if (PRINTDETAILS) ticks11 = clock();
  for (i = 0; i < ncols; i++) variableisbound[i] = 0;
  boundvariablecount = 0;
  for (i = 0; i < nrows && boundvariablecount < ncols; i++) {
    addthisrow = 0;
    if (!myRows[i].used) {
      nevals++;
      lhs = 0;
      // calculate lhs
      for (j = 0; j < myRows[i].nzcnt; j++) {
        lhs += x[myRows[i].rmatind[j]] * myRows[i].rmatval[j];
      }
      // calculate violation
      if (objsense == CPX_MAX) {
        myRows[i].violation = lhs - myRows[i].rhs;
        if (PRINTDETAILS) {
          if (myRows[i].violation > 1e-10) {
            nviolations++;
          }
        }
      }
      else if (objsense == CPX_MIN) {
        printf("note: not coded for minimization\n");
        getchar();
      }

      if ((myRows[i].violation > 1e-10) || (myRows[i].violation < - 1e-10 && myRows
          [i].sense == 'E')) {
        for (k = 0; k < myRows[i].nzcnt && !addthisrow ; k++) {
          if (variableisbound[myRows[i].rmatind[k]] == 0) {
            variableisbound[myRows[i].rmatind[k]] = 1;
            addthisrow = 1;
            optimized = 0;
            boundvariablecount++;
```

```
              if (PRINTDETAILS) {
                lastrowadded = i;
              }
            }
          }// for k
        }// if violated
      }// if not used
      if (addthisrow) {
        status = CPXaddrows (myEnv, myLP, 0, 1, myRows[i].nzcnt,
                             &(myRows[i].rhs),
                             &(myRows[i].sense),
                             &rmatbeg, myRows[i].rmatind,
                             myRows[i].rmatval, NULL, NULL);
        myRows[i].used = 1;
        nrowsaddedtocoverx++;
        addedconstraints++;
      }//if addthisrow
    }//for i
    if (PRINTDETAILS) {
      ticks12 = clock();
      timediffcalcsortadd = ((double)ticks12 -ticks11)/CLOCKS_PER_SEC;
      timediffcalcsortaddtotal += timediffcalcsortadd;
      for (k = 0; k < 30; k++) fprintf(fp2, "\t");
      fprintf(fp2, "%d\t%g\t%g\t%d\t%d\t%d\t\n", \
              ncostopt, timediffcostsimplex, timediffcalcsortadd, nviolations,
                  nrowsaddedtocoverx, lastrowadded);
    }
    if (addedconstraints != 0 && !optimized) {
      // always use dualopt for subsequent steps
      if (PRINTDETAILS) ticks3 = clock();
      status = CPXdualopt (myEnv, myLP);
      if (PRINTDETAILS) {
        ticks4 = clock();
        timediffcostsimplex = ((double)ticks4 -ticks3)/CLOCKS_PER_SEC;
        if (ncostopt > 0) timediffcostsimplextotal += timediffcostsimplex;
      }
      ncostopt++;
      niter = CPXgetitcnt(myEnv, myLP);
      nitertotal = nitertotal + niter;
      phase1cnt = CPXgetphase1cnt(myEnv, myLP);
      phase1cnttotal = phase1cnttotal + phase1cnt;
    }
  } //while (!optimized) loop
  ticks2=clock();  // part of totallstime
  /******************* Clock End *******************/

  status = CPXgetobjval (myEnv, myLP, &objval);


  timediff = ((double)ticks2 -ticks1)/CLOCKS_PER_SEC;
  totallstime = totallstime + timediff;
  printf(
         "COST Time (sec) %g added_constraints %d nevals %d total_iterations %d
             total_phase_1_iterations %d ncostopt %d objval %g\n",
         timediff, addedconstraints, nevals, nitertotal, phase1cnttotal, ncostopt,
             objval);
  fprintf(fp1,
          "COST Time (sec) %g added_constraints %d nevals %d total_iterations %d
              total_phase_1_iterations %d ncostopt %d objval %g\n",
          timediff, addedconstraints, nevals, nitertotal, phase1cnttotal, ncostopt,
              objval);
  if (PRINTDETAILS) fprintf(fp2, "\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t");
  fprintf(fp2,
          "%g\t%d\t%d\t%d\t%d\t%d\t%g\t",
          timediff, addedconstraints, nevals, nitertotal, phase1cnttotal, ncostopt,
              objval);
  printf(
         "Total CPU Time (sec) %g\n", totallstime);
  fprintf(fp1,
          "Total CPU Time (sec) %g\n", totallstime);
  fprintf(fp2,
          "%g\t", totallstime);
  if (PRINTDETAILS) {
    printf(
           "COST Simplex Time (sec) (total) %g\n", timediffcostsimplextotal);
    printf(
           "Time to calc sort and add multi-row COST (sec) (total) %g\n",
               timediffcalcsortaddtotal);
    fprintf(fp1,
            "COST Simplex Time (sec) (total) %g\n", timediffcostsimplextotal);
    fprintf(fp1,
            "Time to calc sort and add multi-row COST (sec) (total) %g\n",
                timediffcalcsortaddtotal);
    fprintf(fp2,"%g\t%g\t", timediffcostsimplextotal, timediffcalcsortaddtotal);
  }

  // System Time Print
  time (&ticks1);
  timeinfo = localtime (&ticks1);
  strftime (currenttime,50,"%c", timeinfo);
  printf ("%s\n\n", currenttime);
  fprintf (fp1, "%s\n\n", currenttime);
  fprintf (fp2, "%s", currenttime);
  fprintf (fp2, "\n");

  CPXfreeprob(myEnv, &myLP);
  free(x);

  CPXcloseCPLEX(&myEnv);
  fclose(fp1);
  fclose(fp2);

  return 0;
}// end of CostSolver
```

A.5 SortCopyRows

The routine below is called by PrepMyRows to sort and copy the row structure according to the constraint selection metric.

```
int SortCopyRows(struct row ** myRows_p, struct row * myRows, int nrows,
                 double * sorttime, double * memcopytime)
{
  clock_t ticks1, ticks2;
  double timediff;
  int i;

  /******************* Clock Start *******************/
  ticks1=clock();
  qsort(myRows_p, nrows, sizeof(struct row *), compar);
  ticks2=clock();
  /******************* Clock End ******************/

  timediff = ((double)ticks2 -ticks1)/CLOCKS_PER_SEC;
  *sorttime = timediff;

  /******************* Clock Start *******************/
  ticks1=clock();

  for (i = 0; i < nrows; i++) {
    int j;
    myRows[i].nzcnt = myRows_p[i]->nzcnt;
    myRows[i].rmatbeg = myRows_p[i]->rmatbeg;
    myRows[i].rmatind
    = (int *) calloc(myRows[i].nzcnt, sizeof(int));
    myRows[i].rmatval
    = (double *) calloc(myRows[i].nzcnt, sizeof(double));
    for (j = 0; j < myRows[i].nzcnt; j++) {
      myRows[i].rmatind[j] = myRows_p[i]->rmatind[j];
      myRows[i].rmatval[j] = myRows_p[i]->rmatval[j];
    }
    myRows[i].rmatspace = myRows_p[i]->rmatspace;
    myRows[i].surplus = myRows_p[i]->surplus;
    myRows[i].sense = myRows_p[i]->sense;
    myRows[i].rhs = myRows_p[i]->rhs;
    myRows[i].rad = myRows_p[i]->rad;
    myRows[i].criteria = myRows_p[i]->criteria;
    myRows[i].violation = myRows_p[i]->violation;
    myRows[i].used = myRows_p[i]->used;
  }

  ticks2=clock();
  /******************* Clock End *******************/

  timediff = ((double)ticks2 -ticks1)/CLOCKS_PER_SEC;
  *memcopytime = timediff;
  return 0;
}
```

A.6 Compar

The following code is utilized by the ANSI qsort called by the SortCopyRows procedure.

```
int compar(const void * a, const void * b)
{
  struct row ** rowa_p = (struct row **) a;
  struct row ** rowb_p = (struct row **) b;

  if ((*rowa_p)->criteria < (*rowb_p)->criteria) {
    return -1;
  }
  else if ((*rowa_p)->criteria > (*rowb_p)->criteria) {
    return 1;
  }
  else {
    return 0;
  }
}
```

APPENDIX B
CODE EXAMPLE FOR THE COST GRAD

In this appendix, an example of ANSI C code for implementing the COST GRAD utilizing the CPLEX callable library is presented. Since the code is analogous to that of the COST NRAD, the parts that differ from NRAD are described.

B.1 PrepMyRows

Calculation of GRAD criterion can be implemented by applying the code such as the following into the PrepMyRows routine presented in Section A.3.

```
double * newobj = (double *) calloc(*ncols, sizeof(double));
double newb;
double sum_plus;
double sum_minus;
// find min_b
min_b = 1e10;
for (i = 0; i < *nrows; i++) {
  if (myRows_p[i]->rhs < min_b) {
    min_b = myRows_p[i]->rhs;
  }
}
// determine new c
for (i = 0; i < *ncols; i++) {
  if (obj[i] > 0) {
    newobj[i] = obj[i];
  }
  else {
    newobj[i] = -1;
  }
}
if (*objsense == CPX_MAX) {
  for (i = 0; i < *nrows; i++) {
    if (min_b < 0) {
      newb = myRows_p[i]->rhs - min_b + epsilon1;
    }
    else {
      newb = myRows_p[i]->rhs;
    }
    sum_plus = 0;
    sum_minus = 0;
    for (j = 0; j < myRows_p[i]->nzcnt; j++) {
      if (obj[myRows_p[i]->rmatind[j]] > 0) {
        sum_plus += (double) myRows_p[i]->rmatval[j] * newobj[myRows_p[i]->
            rmatind[j]];
      }
      else {
        sum_minus += (double) myRows_p[i]->rmatval[j] * newobj[myRows_p[i]->
            rmatind[j]];
      }
    }
    sum_plus = sum_plus / newb;
    sum_minus = sum_minus / newb;
    myRows_p[i]->rad = sum_plus - sum_minus;
    myRows_p[i]->criteria1 = -myRows_p[i]->rad;
  }
}
free (newobj);
SortCopyRows(lpfilename, myRows_p, myRowsRAD, *nrows);
```

B.2 CostSolver

An example code for adding an artificial bounding constraint is shown below.

```
double rhs;
char sense;
double *bound = (double *) calloc(ncols, sizeof(double));
int *rmatind = (int *) calloc(ncols, sizeof(int));

  sense = 'L';
  rhs = 1e10;
  for (i = 0; i < ncols; i++) {
    bound[i] = obj[i];
  }
  status = CPXaddrows (myEnv, myLP, 0, 1, ncols, &rhs, &sense,
                   &rmatbeg, rmatind, bound, NULL, NULL);
```

To determine the initial set of constraints to add to form P_0, and to determine $POSITIVE_a$ and $NEGATIVE_a$, apply the following to the CostSolver presented in Section A.4.

```

int * addthisrow  = (int *) calloc(nrows, sizeof(int));
int * varisboundpos = (int *) calloc(ncols, sizeof(int));
int * varisboundneg = (int *) calloc(ncols, sizeof(int));
int boundposvariablecount = 0;
int boundnegvariablecount = 0;
int lastrowapos = 0; // last row where positive aij is found
int lastrowaneg = 0; // last row where negative aij is found

for (i = 0; i < ncols; i++) {
  varisboundpos[i] = 0;
  varisboundneg[i] = 0;
}

  for (i = 0; i < nrows && (boundposvariablecount < ncols ||
       boundnegvariablecount < ncols); i++) {
    for (k = 0; k < myRows[i].nzcnt; k++) {
      // cover if a>0
      if (myRows[i].rmatval[k] > 0 && !varisboundpos[myRows[i].rmatind[k]]) {
        boundposvariablecount++;
        varisboundpos[myRows[i].rmatind[k]] = 1;
        addthisrow[i] = 1;
        lastrowapos = i;
      }// if a > 0
      // cover if a<0
      if (myRows[i].rmatval[k] < 0 && !varisboundneg[myRows[i].rmatind[k]]) {
        boundnegvariablecount++;
        varisboundneg[myRows[i].rmatind[k]] = 1;
        addthisrow[i] = 1;
        lastrowaneg = i;
      }// if a < 0
    }// for k
  }// for i

  // search the rest of the rows to check
  // lastrowapos and lastrowaneg
  if (lastrowapos > lastrowaneg) i = lastrowapos;
  else i = lastrowaneg;
  while (i < nrows) {
    for (k = 0; k < myRows[i].nzcnt; k++) {
      if (myRows[i].rmatval[k] > 0) {
        lastrowapos = i;
      }
      else {
        lastrowaneg = i;
      }
    }
    i++;
   }
  for (i = 0; i < nrows; i++) {
    if (addthisrow[i]) {
        status = CPXaddrows (myEnv, myLP, 0, 1, myRows[i].nzcnt,
                             &(myRows[i].rhs),
                             &(myRows[i].sense),
                             &rmatbeg, myRows[i].rmatind,
                             myRows[i].rmatval, NULL, NULL);
        myRows[i].used = 1;
        numrowsaddedtobound++;
    }//if addthisrow
  }// for i
```

For determining cuts to add for $\mathrm{P}_r, r = 1, 2, \ldots,$ apply the following to the CostSolver presented in Section A.4.

```
    optimized = 1;

    for (i = 0; i < ncols; i++) {
      varisboundpos[i] = 0;
      varisboundneg[i] = 0;
    }
    for (i = 0; i < nrows; i++) addthisrow[i] = 0;
    boundposvariablecount = 0;
    boundnegvariablecount = 0;
    numconstraintstobeadded = 0;

    for (i = 0;
           i < nrows  &&
           ((boundposvariablecount < ncols && i <= lastrowapos) ||
            (boundnegvariablecount < ncols && i <= lastrowaneg))
      ; i++) {
      if (!myRows[i].used) {
        nevals++;
        lhs = 0;
        // calculate lhs
        for (j = 0; j < myRows[i].nzcnt; j++) {
          lhs += x[myRows[i].rmatind[j]] * myRows[i].rmatval[j];
        }
        // calculate violation
        if (objsense == CPX_MAX) {
          myRows[i].violation = lhs - myRows[i].rhs;
        }
        else if (objsense == CPX_MIN) {
          printf("didn't code this yet\n");
          getchar();
        }

  /******************************/
  /* violation check / multicut */
  /******************************/
      // EPSILON_VC has been defined earlier as some small number such as 1e-10
```

```
        if ((myRows[i].violation > EPSILON_VC) || (myRows[i].violation < - EPSILON_VC
            && myRows[i].sense == 'E')) {
          for (k = 0; k < myRows[i].nzcnt; k++) {
            // cover if a>0
              if (myRows[i].rmatval[k] > 0 && !varisboundpos[myRows[i].rmatind[k]]) {
                boundposvariablecount++;
                varisboundpos[myRows[i].rmatind[k]] = 1;
                addthisrow[i] = 1;
                optimized = 0;
              }
            // cover if a<0
              if (myRows[i].rmatval[k] < 0 && !varisboundneg[myRows[i].rmatind[k]]) {
                boundnegvariablecount++;
                varisboundneg[myRows[i].rmatind[k]] = 1;
                addthisrow[i] = 1;
                optimized = 0;
              }
          }// for k
        }// if violated
      } // end of if row has not been used
    }// end of for loop for i
    for (i = 0; i < nrows; i++) {
      if (addthisrow[i]) {
          status = CPXaddrows (myEnv, myLP, 0, 1, myRows[i].nzcnt,
                               &(myRows[i].rhs),
                               &(myRows[i].sense),
                               &rmatbeg, myRows[i].rmatind,
                               myRows[i].rmatval, NULL, NULL);
          myRows[i].used = 1;
          addedconstraints++;
      }//if addthisrow
    }//for loop for adding rows
```

REFERENCES

[1] M. J. Todd, "The many facets of linear programming," *Mathematical Programming*, vol. 91, pp. 417–436, 2002. [Online]. Available: http://dx.doi.org/10.1007/s101070100261

[2] J. M. Rosenberger, E. L. Johnson, and G. L. Nemhauser, "Rerouting aircraft for aircraft recovery," *Transportation Science*, vol. 37, no. 4, pp. 408–421, 2003. [Online]. Available: http://dx.doi.org/10.1287/trsc.37.4.408.23271

[3] T.-K. Sung, "Constraint optimal selection techniques (COSTs) for a class of linear programming problems," Ph.D. dissertation, The University of Texas at Arlington, August 2006.

[4] H. W. Corley and J. M. Rosenberger, "System, method and apparatus for allocating resources by constraint selection," US Patent US 8 082 549, December 20, 2011. [Online]. Available: http://patft1.uspto.gov/netacgi/nph-Parser?patentnumber=8082549

[5] R. Cottle, E. Johnson, and R. Wets, "George B. Dantzig (1914–2005)," *Notices of the AMS*, vol. 54, no. 3, pp. 344–362, 2007.

[6] M. Osborne and G. Watson, "On the best linear Chebyshev approximation," *The Computer Journal*, vol. 10, no. 2, pp. 172–177, 1967.

[7] G. B. Dantzig and M. N. Thapa, *Linear programming 1: introduction.* New York: Springer-Verlag, 1997.

[8] V. Klee and G. J. Minty, "How good is the simplex algorithm?" in *Inequalities*, O. Shisha, Ed. New York: Academic Press, 1972, vol. III, pp. 159–175.

[9] D. Goldfarb and J. Reid, “A practicable steepest-edge simplex algorithm,” *Mathematical Programming*, vol. 12, no. 1, pp. 361–371, 1977. [Online]. Available: http://dx.doi.org/10.1007/BF01593804

[10] R. Bixby, “Progress in linear programming,” *ORSA Journal on computing*, vol. 6, pp. 15–15, 1994. [Online]. Available: http://dx.doi.org/10.1287/ijoc.6.1.15

[11] L. G. Khachiyan, “A polynomial algorithm in linear programming,” *Doklady Adad. Nauk SSSR (translated as Soviet Mathematics Doklady)*, vol. 20, pp. 191–194, 1979.

[12] N. Karmarkar, “A new polynomial-time algorithm for linear programming,” in *Proceedings of the sixteenth annual ACM symposium on Theory of computing.* ACM, 1984, pp. 302–311.

[13] V. Chvátal, *Linear programming.* New York: W.H. Freeman and Company, 1983.

[14] G. Dantzig and W. Orchard-Hays, “The product form for the inverse in the simplex method,” *Mathematical Tables and Other Aids to Computation*, vol. 8, no. 46, pp. 64–67, 1954. [Online]. Available: http://dx.doi.org/10.2307/2001993

[15] C. Lemke, “The dual method of solving the linear programming problem,” *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 36–47, 1954. [Online]. Available: http://dx.doi.org/10.1002/nav.3800010107

[16] J. J. Stone, “The cross-section method: an algorithm for linear programming,” Rand Corporation Memorandum P-1490, 1958.

[17] G. L. Thompson, F. M. Tonge, and S. Zionts, “Techniques for removing nonbinding constraints and extraneous variables from linear programming problems,” *Management Science*, vol. 12, pp. 588–608, 1966. [Online]. Available: http://dx.doi.org/10.1287/mnsc.12.7.588

[18] I. Adler, R. Karp, and R. Shamir, "A family of simplex variants solving an $m \times d$ linear program in expected number of pivots steps depending on d only," *Mathematics of Operations Research*, vol. 11, pp. 570–590, 1986. [Online]. Available: http://dx.doi.org/10.1287/moor.11.4.570

[19] M. Zeleny, "An external reconstruction approach (ERA) to linear programming," *Computers & Operations Research*, vol. 13, no. 1, pp. 95–100, 1986. [Online]. Available: http://dx.doi.org/10.1016/0305-0548(86)90067-5

[20] D. C. Myers and W. Shih, "A constraint selection technique for a class of linear programs," *Operations Research Letters*, vol. 7, no. 4, pp. 191–195, 1988. [Online]. Available: http://dx.doi.org/10.1016/0167-6377(88)90027-2

[21] N. D. Curet, "A primal-dual simplex method for linear programs," *Operations Research Letters*, vol. 13, no. 4, pp. 223–237, 1993. [Online]. Available: http://dx.doi.org/10.1016/0167-6377(93)90045-I

[22] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*, 3rd ed. New York: John Wiley, 2005.

[23] A. W. Naylor and G. R. Sell, *Linear Operator Theory in Engineering and Science*. New York: Springer-Verlag, 1982.

[24] P.-Q. Pan, "Practical finite pivoting rules for the simplex method," *OR Spektrum*, vol. 12, no. 4, pp. 219–225, 1990. [Online]. Available: http://dx.doi.org/10.1007/BF01721801

[25] ——, "A simplex-like method with bisection for linear programming," *Optimization*, vol. 22, no. 5, pp. 717–743, 1991. [Online]. Available: http://dx.doi.org/10.1080/02331939108843714

[26] F. Trigos, J. Frausto-Solis, and R. R. Rivera-Lopez, "A simplex-cosine method for solving hard linear problems," *Advances in Simulation, System Theory*

and Systems Engineering, vol. 70X, pp. 27–32, 2002. [Online]. Available: http://lsog.tol.itesm.mx/public_html/files/SimplexCosine.pdf

[27] H. V. Junior and M. P. E. Lins, "An improved initial basis for the simplex algorithm," *Computers & Operations Research*, vol. 32, no. 8, pp. 1983–1993, 2005. [Online]. Available: http://dx.doi.org/10.1016/j.cor.2004.01.002

[28] H. W. Corley, J. M. Rosenberger, W.-C. Yeh, and T.-K. Sung, "The cosine simplex algorithm," *The International Journal of Advanced Manufacturing Technology*, vol. 27, no. 9, pp. 1047–1050, 2006. [Online]. Available: http://dx.doi.org/10.1007/s00170-004-2278-1

BIOGRAPHICAL STATEMENT

Goh Saito（齊藤 剛）was born in Fuchu-shi Tokyo, Japan. He received his B.S. and M.S. degrees in Chemical Engineering from Columbia University in 1996 and 1998, respectively, and his Ph.D. degree in Industrial Engineering from The University of Texas at Arlington in 2012. His current research interest is in the area of optimization.