1997

# A probabilistic method for cleaning contaminated systems of linear inequalities.

Halima M. El-Khatib
*University of Windsor*

# INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**UMI**

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

# A PROBABILISTIC METHOD FOR CLEANING CONTAMINATED SYSTEMS OF LINEAR INEQUALITIES

by

**HALIMA M. El-Khatib**

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor

Windsor, Ontario, Canada
1997

# Abstract

Mathematical programming (MP) problems can be viewed as abstractions of real-world situations. They consist of an objective function which needs to be maximized or minimized, subject to a set of constraints which defines a feasible region. The feasible region denoted by R, is often defined by a set of linear inequalities. For "real world" problems there can be thousands of inequalities and variables. A problem with such large systems is that there are often errors in formulating the constraints which may cause the feasible region to be empty. Another problem is that many of the constraints may be redundant. We define such systems as contaminated systems of linear inequalities.

This thesis develops the first method to simultaneously deal with infeasibility and redundancy. The new procedure is a probabilistic approach based on an equivalence to the set covering problem.

*In memory of my son, Jad*
*To my daughter, Samar*

# Acknowledgements

This thesis work could not have been accomplished without the support of many people. I would like to express my appreciation to R.J. Caron whose tremendous support, guidance, stimulating ideas, and review of this thesis report were invaluable to the successful completion of this thesis. I would like to thank Dr. R. A. Frost for his guidance and suggestions on my survey related to this thesis. Thanks to Dr. R. D. Kent, who has inspired me and has given me the guidance, encouragement and support required. I would also like to express my sincere thanks to Mr. Walid Mnaymneh and Mr. Sandeep Kamat for providing me the necessary technical support.

I would like to thank my husband Ali for his love, patience and support in helping make a dream come true.

Finally I would like to express my gratitude to all my teachers, fellow students and friends who have made this "journey" such a positive and rewarding experience for me.

# TABLE OF CONTENTS

# List of Figures

# List of Tables

# List of Equations

# Chapter 1 INTRODUCTION

## 1.1 Overview

Systems of linear constraints are used to model various phenomena. A model containing reasonable detail could involve thousands of variables and thousands of linear constraints. One problem with such a system is that it is often inconsistent. A system of constraints is inconsistent if it defines an empty feasible region. We might then wish to locate a minimal infeasible set. The system might then be made feasible by deleting or modifying the constraints belonging to this infeasibility set. The problem of locating a minimal infeasible set has been studied by many authors ( Chinneck and Dravnieks [14], Roodman [30], Van Loom [27], and Gleeson and Ryan [21]).

Another problem with a large system is that many of the constraints may be redundant. A constraint is redundant if it can be removed without affecting the feasible region defined by the original system. It is desirable to identify and remove redundant constraints in a pre-optimization phase. Not only does the elimination of redundant constraints reduce the computational effort required to solve an associated mathematical programming problem, but it also provides insight into the mathematical model represented by the system of linear constraints [25]. The problem of identifying redundant constraints has been studied by

many authors [2, 5, 7, 9, 20, 31, 36]. The two most popular approaches are the probabilistic methods [1] and the deterministic methods [25].

## 1.2 The Thesis Statement

Given a system of linear inequalities constraints. We hypothesize that both problems, locating a minimal infeasible set and removing redundancy, can be solved simultaneously with a probabilistic technique based upon Boneh's set covering equivalence [4]. We believe that this unified approach is important not only for theoretical reason, but also for practical reasons. A unified approach will likely lead to computational efficiencies.

## 1.3 Purpose of this Investigation

The purpose of the thesis work is to:

1. Show that the problem of locating a minimal infeasible set of linear inequalities has a set covering equivalence.

2. Show how both problems, locating a minimal infeasible set and removing redundancy, can be handled simultaneously using the set covering equivalence approach.

## 1.4 Objectives and Scope of the Thesis Work

The objectives and scope of the thesis work are as follows:

1. Develop an efficient method for implementing Boneh's set covering equivalence to simultaneously solve both problems: locating a minimal infeasible set and removing redundancy.
2. Program and test the proposed set-covering equivalence method to solve both problems: locating a minimal infeasible set and removing redundancy.

## 1.5 Organization of the Thesis

Chapter 2 describes some basic linear programming concepts and the related concepts of redundancy and infeasibility. Chapter 3 surveys the existing approaches for locating a minimal infeasible set. Chapter 4 surveys some of the existing approaches for removing redundancy. Chapter 5 presents the new method to solve both problems simultaneously. Chapter 6 describes the programming of the new method and its performance on test problems. Chapter 7 presents a summary of findings and conclusions. Chapter 7 also discusses recommendations and suggests possible future work.

# Chapter 2 PROBLEM DEFINITIONS AND NOTATION

## 2.1 The Linear Programming Problem

We consider the linear programming (LP) problem:

$$max \left\{ c^T x \mid Ax \leq b \right\}.$$

where c and x are n-vectors, A is an mxn matrix, and b is an m-vector. Letting $a_i^T$ represents the i-th row of A, with $b_i$ the i-th component of b, we can also write the LP as

$$max \left\{ c^T x \mid a_i^T x \leq b_i, i = 1, \ldots, m \right\}. \tag{2.1}$$

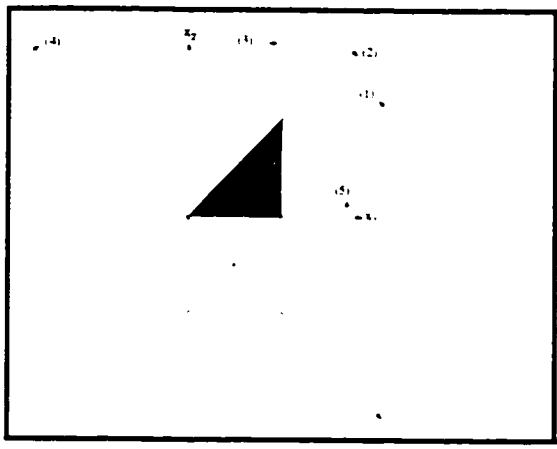
### 2.1.1 Feasible Region

The feasible region for the linear programming problem (2.1) is denoted by $R$ and is given by $R = \{x \in R^n \mid Ax \leq b\} = \left\{x \in R^n \mid a_i^T x \leq b_i, i = 1, \ldots, m\right\}$. We say that "$a_i^T x \leq b_i$" is the i-th constraint, and we denote the set of all constraints by G(I), that is,

$$G(I) = \left\{ a_i^T x \leq b_i \mid i \in I \right\}.$$

where I = {1,2, ..., m}. For any subset I′ of I, we have the set G(I′) and the corresponding region R′.

As an example, consider Figure 2.1 with the set of 5 linear constraints given in the right panel.



1. $a_1^T x = x_1 - x_2 \leq 8.$
2. $a_2^T x = -x_1 + x_2 \leq 0.$
3. $a_3^T x = x_1 \leq 8.$
4. $a_4^T x = -x_1 - x_2 \leq 0.$
5. $a_5^T x = -x_2 \leq 0.$

Figure 2.1 Feasible Region

In this and subsequent diagrams showing sets of constraints the feasible region is represented by the shaded region in the corresponding figure. The arrows on the constraint boundaries point to the half-space that satisfies the corresponding inequality.

## 2.1.2 Necessary and Redundant Constraints

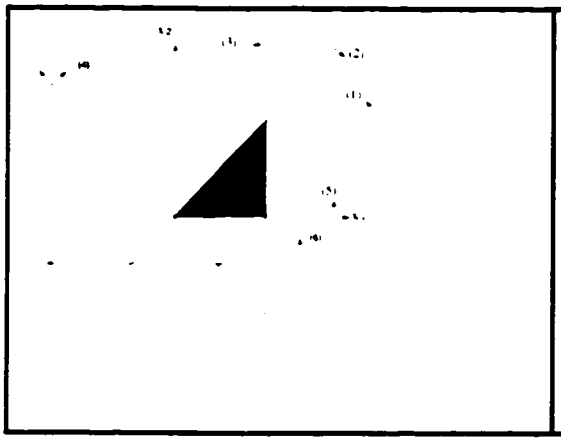
Let $I' = I\backslash\{k\}$. The k-th constraint is said to be redundant with respect to G(I) if $R = R'$, that is, if either $R' = R = \emptyset$ or if $R' \neq \emptyset$ and

$$max \left\{ a_k^T x \mid x \in R' \right\} \leq b_k. \tag{2.2}$$

The k-th constraint is said to be necessary with respect to G(I) if $R \neq R'$, that is, if there exist an $x' \in R'$ such that $a_k^T x' > b_k$.

***A constraint is redundant if removing it from the set causes no change to the feasible region. A constraint is necessary if removing it from the set causes a change to the feasible region. Note that with respect to a given set of constraints a constraint is either redundant or necessary, but never both.***

As an example, consider Figure 2.2 with the set of 6 linear constraints given in the right panel.



1. $a_1^T x = x_1 - x_2 \leq 8.$
2. $a_2^T x = -x_1 + x_2 \leq 0.$
3. $a_3^T x = x_1 \leq 8.$
4. $a_4^T x = -x_1 - x_2 \leq 0.$
5. $a_5^T x = -x_2 \leq 0.$
6. $a_6^T x = -x_2 \leq 4$

Figure 2.2 Redundant Constraint

We can see that constraints 1, 4 and 6 are redundant and that all others are necessary.

Acoording to Telgen [35] redundant constraints can be further classified depending upon whether or not (2.2) holds as an equality or inequality.

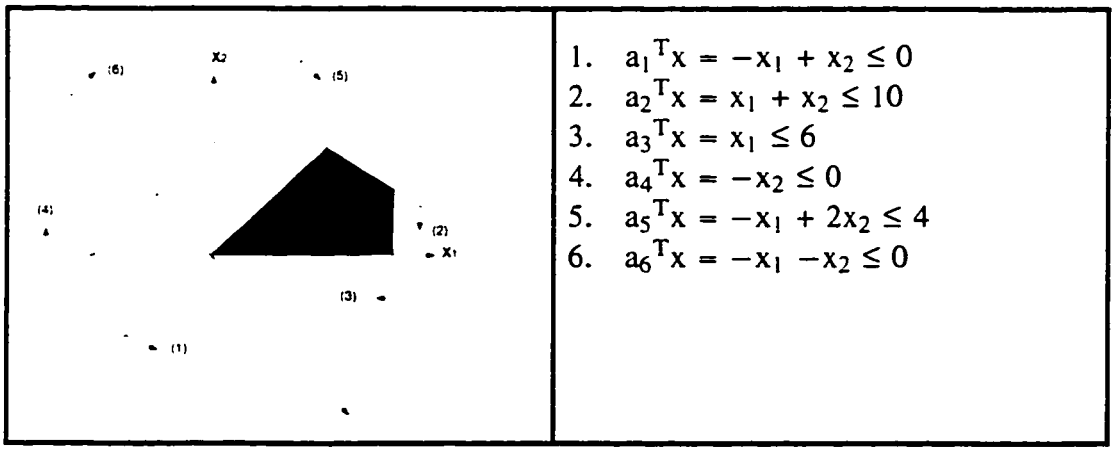
Let $I' = I\backslash\{k\}$. The k-th constraint is weakly redundant with respect to G(I) if $R' \neq \emptyset$ and

$$max\left\{a_k^T x \mid x \in R'\right\} = b_k. \tag{2.3}$$

Let I′ = I\{k}. The k-th constraint is strongly redundant with respect to G(I) if either R′ = R = ∅ or if R′ ≠ ∅ and

$$max\left\{a_k^T x \mid x \in R'\right\} < b_k. \tag{2.4}$$

As an example, consider Figure 2.3 with the set of 6 linear constraints given in the right panel. Constraints (1), (2), (3) and (4) are necessary with respect to G(I) and constraints (5) and (6) are redundant with respect to G(I). Furthermore, constraint (6) is weakly redundant and constraint (5) is strongly redundant. From Figure 2.3 we see that weakly redundant constraints "touch" the feasible region, and that strongly redundant constraints do not "touch" the feasible region.



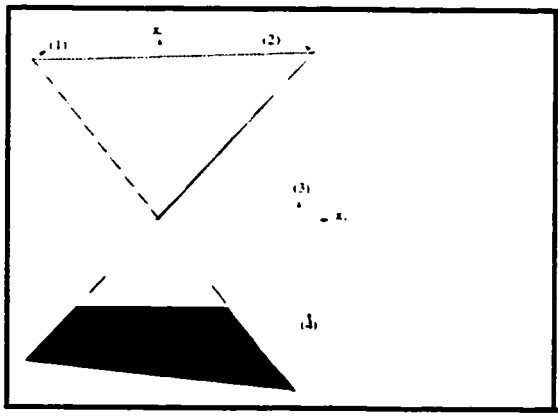

1. $a_1^T x = -x_1 + x_2 \leq 0$
2. $a_2^T x = x_1 + x_2 \leq 10$
3. $a_3^T x = x_1 \leq 6$
4. $a_4^T x = -x_2 \leq 0$
5. $a_5^T x = -x_1 + 2x_2 \leq 4$
6. $a_6^T x = -x_1 - x_2 \leq 0$

Figure 2.3 Weakly and Strongly Redundant Constraints

### 2.1.3 Infeasible Sets of Constraints

A set of constraints is said to be infeasible if it defines an empty feasible region R.

Consider Figure 2.4 with the set of 4 linear constraints.



1. $a_1^T x = -x_1 - x_2 \leq 0.$
2. $a_2^T x = x_1 - x_2 \leq 0.$
3. $a_3^T x = -x_2 \leq 0.$
4. $a_4^T x = x_2 \leq 4.$

Figure 2.4 Infeasible Set of Constraints

Let I = {1, 2, 3, 4}, I′ = {1, 2, 3}, I″= {4}. The upper shaded region is R′ and the lower is R″. We see that $R = R' \cap R'' = \emptyset$.

### 2.1.4 Implicit Equality Constraints

Implicit equalities are inequality constraints that are satisfied as equality in all solutions x ∈ R [35], that is, constraint k is an implicit equality with respect to G(I) if $a_k^T x = b_k$, ∀ x ∈ R.

### 2.1.5 Minimal Number of an Irreducible Set of Constraints

The total number of constraints in an irreducuble set of constraints determining R is minimal if and only if the set contains no redundant constraints and no implicit equalities [35].

### 2.1.6 Irreducible Inconsistent System

An irreducible inconsistent system, IIS, is a system with a minimal set of inconsistent constraints [14].

### 2.1.7 Theorem

If there are n variables in the original system of linear inequalities defining a linear programming problem, the maximum cardinality of any irreducible inconsistent system is n+1. PROOF([16], p. 24).

# Chapter 3 INFEASIBILITY

In this chapter we survey some of the existing approaches for identifying a minimal infeasible set of constraints.

## 3.1 A Survey of the Literature

The problem of detecting minimal infeasible sets has received some attention, and various mathematical approaches to post-infeasibility analysis have been given [21, 22, 23, 24, 27, 28, 30].

In 1980, Roodman [30] describes how to remove infeasibility when the phase 1 LP terminates with some of the artificial variables having nonzero values. Sensitivity analysis is used to find the minimum adjustment to the right hand sides of the corresponding constraints to achieve feasibility. This approach is not useful if the adjusted constraint is in fact correct while some other conflicting constraint is in error.

Greenberg [22, 23] described a set of heuristics which rely on tracing back through a series of manipulations of the model, such as removal of redundant constraints, bound tightening, path and cycle generation, and matrix analysis. These heuristics were not designed specifically for locating minimal infeasible sets of constraints, they often do so, but cannot guarantee such a result [14].

In 1983, Murty [28] described how to use the phase 1 LP solution to find a set of constraints which is causing infeasibility. The shadow prices of the phase 1 solution are used to relate possible causative constraints as is done by Roodman [30], and, in addition, the reduced costs for the original variables are used to implicate possible causative nonnegativity constraints. However, the indicated set may consist of a number of minimal infeasible sets, and no method is given for further localization [14] .

In 1981, Van Loom [27] presented a simplex variant and a set of necessary and sufficient conditions for the recognition of a minimal infeasible set. In this method, the search for the elements of the set is undirected, leaving no option but a combinatorially explosive exhaustive search [14]. The method also suffers from problem blow-up because equality constraints must be converted to a pair of inequality constraints and nonnegativity constraints must be explicitly added to the working constraint set. Greenberg and Murphy [24] point out that Van Loon's method could be extended to find minimal causative sets more efficiently by pivoting through alternative bases.

In 1990, Gleeson and Ryan [21] described a complete localization algorithm. They combine van Loon's result with a variant of Farkas Theorem [21] of the Alternative to obtain a polytope in which each vertex indexes the members of a minimal infeasible set of constraints. Gleeson and Ryan's method shares the drawbacks of Van Loon's method, but has the advantage of a directed and efficient

search [14].

Chinneck and Dravnieks developed an algorithm [14] for locating a minimal infeasible set of constraints. The algorithm is linear-programming based. A general description of the algorithm follows.

### 3.1.1 Locating Minimal Infeasible Constraint Sets in Linear Programming

Although various mathematical approaches to post-infeasibility analysis have been given previously, Chinneck and Dravnieks's algorithm[14], with their claim, is the first robust infeasibility localizer reported.

The approach to locate an infeasibility is to gradually eliminate constraints from the original set defining the problem until those remaining constitute a minimal infeasible set. This approach is called filtering the constraint set. Three basic filtering routines are developed. They are: deletion, elastic and sensitivity, which are then combined into a recommended integrated filtering algorithm.

Deletion filtering is the cornerstone, providing a positive identification of a single minimal set which causes the infeasibility. Given an LP having one or more irreducible inconsistent systems, deletion filtering operates by considering each functional constraint individually, as follows. Temporarily remove the constraint from the LP, then test the reduced LP for feasibility. If the reduced LP is infeasible, then remove the constraints permanently; if the reduced LP becomes

feasible, then return the constraints to the LP. Continue in this fashion until all of the constraints have been tested.

Elastic filtering speeds IIS localization by quickly eliminating non-IIS functional constraints from large models. It uses elastic programming and the fact that stretching constraints in an IIS sufficiently permits a feasible solution. The algorithm generates a series of LPs in which some of the functional constraints are elastic and some are nonelastic. After each LP is solved, any stretched constraints are reconverted to nonelastic form by removing the elastic variables. Enforced constraints are members of some IIS, since only IIS constraints are stretched. Enforcing functional constraints forces another elastic member of the IIS to stretch when the next LP is solved. When all of the members of an irreducibly inconsistent set of functional constraints (IISF) have been enforced, the next LP will be infeasible, then the method is halted. The output of the elastic filtering algorithm is the set of enforced constraints, which must contain at least one IISF.

Sensitivity Filtering uses the fact that stretching a constraint is equivalent to altering the right hand side (rhs) of the constraint. Sensitivity filtering is applied when an LP is infeasible. The phase 1 or elastic solution will show sensitivity to a small adjustement of the right hand sides (rhs's) of some of the IIS constraints, but never to the rhs of a non-IIS constraint. See reference [27] for a detailed description on how to use right hand side (rhs) manipulation of the constraints to recognize an IIS.

The three filtering algorithms can be combined in various ways to create efficient integrated algorithms to locate minimal infeasible constraints sets. Taking into consideration that infeasibility is discovered by solving a phase 1 LP, and that sensitivity filtering of the phase 1 solution is cheap [14], the authors, Chinneck and Dravnieks, assumed that this will be the first step in any integrated algorithm.

Depending on whether the goal is to identify a single IIS as quickly as possible, or to identify as many IIS's at reasonable cost, final filtering can proceed in a number of different ways. The authors suggested two ways to combine the algorithms described earlier. They are: Deletion/Sensitivity Filtering and Elastic/Sensitivity Filtering.

## 3.2 Conclusion

In this chapter we have reviewed some of the existing approaches for locating a minimal infeasible set. Most of these approaches are linear programming based methods and they deal with locating a minimal infeasible set only. Our method, the method given in chapter 5, differs from these existing approaches in that it is based on a set-covering equivalence method. It also differs from the existing approaches in that it simultaneously deals with redundancy and feasibility.

# Chapter 4 REDUNDANCY

## 4.1 Introduction

The first paper devoted entirely to redundancy was given by Boot [8]. Boot suggested checking the redundancy of an inequality by replacing the inequality sign of a constraint by a strict inequality in the reverse direction. The constraint is necessary if the resulting system is consistent. One disadvantage of the method is that a system of linear constraints has to be examined for feasibility in order to examine a constraint for redundancy.

In 1965, Zionts [37] gave some improvements upon the implementation of Boot's method, but not to the point where it achieved practical value. In addition, a number of other methods were developed that dealt with redundancy, among which the geometric vertex enumeration method is the most well-known. The geometric vertex enumeration method's essential characteristic is the establishment of a number of situations in which redundancy can be recognized immediately without further computations.

In 1971, Lisy [26] used the rules given by Zionts to identify all redundant constraints in systems of linear constraints. Gal [20] enlarged this approach by adding rules for situations in which constraints can be identified immediately as being nonredundant. Reference [25] presents a complete description of Gal's method. The methods are equivalent in that they classify constraints as redundant

or necessary; they produce results that are unconditionally correct; they perform iterations of an active set linear programming algorithm (for example, the simplex method [18]). Later Caron et. al [12] expanded the above methods by adding rules to deal with degeneracy.

As we have seen, the problem of identifying redundant constraints has been studied by many authors [25]. The two most popular approaches are the probabilistic hit-and run method [1] and linear-programming based methods [25]. In addition to these two approaches, there is the set-covering approach proposed by Boneh [4].

## 4.2 The Probabilistic Hit-and-Run Methods

The first hit-and-run method was the so-called Hypersphere Direction (HD) method was introduced by Boneh and Golan [6] for generating random points inside a feasible region defined by a system of linear inequality constraints and for classifying these constraints as either redundant or necessary.

In 1980, Telegen [34] suggested the coordinate direction (CD) method as an alternative to HD.

Smith [32] introduced a class of hit-and-run algorithms, which he called mixing algorithms, for generating random points in a feasible region R. Later in 1984, Smith [33] showed that if R is open and bounded then the sequence of iteration points of the HD algorithm converges to the uniform distribution over R.

In 1987, Berbee et al. [2] showed that if R is a convex polyhedron then the sequence of iteration points of the CD algorithm converges to the uniform distribution on R.

In 1986, Caron et al. [10] presented a class of hit-and-run algorithms made up of so-called continuous variants and discrete variants.

In 1990, Belisle et al. [17] considered a general class of hit-and-run algorithms where R is a bounded open subset of $R^n$.

The probabilistic Hit-and-Run methods are based on the following idea. If a randomly generated line intersects the interior of the feasible region R, then the end points of the feasible segment of that line identify necessary inequalities with probability one. The most well-known probabilistic method is the Hyperspheres Direction or HD method [3].

### 4.2.1 The HD Algorithm

An iteration (see table 4.1) of HD starts with a feasible interior point, say $x_j$. In the hit step, we first generate a random direction vector $s_j$. This is done by selecting a point uniformly distributed over the surface of the n dimensional hypersphere

$$H = \{x \in R^n : ||x|| = 1\}.$$

Together, $x_j$ and $s_j$ define a line in $R^n$ which passes through the interior of the feasible region R. We then determine the feasible segment of the line by

the calculation of the intersection points of the line with the boundaries of the constraints. The inequalities that are hit by the end points of the feasible line segment are said to have been detected, and are classified as necessary. In the run step we generate a new interior point $x_{j+1}$ from a uniform distribution over the feasible line segment. These steps are repeated until a stopping rule is satisfied.

Upon termination, all constraints that have not been detected are classified, possibly with error, as redundant.

(Input) : I, G(I), and $x_0 \in$ int (R).

(Initialization) : Set j = 1, $J_0 = \emptyset$.

(The Hit Step) : Generate a random vector $s_j$ uniformly distributed over the surface of the n dimensional hypersphere H.

(Calculate) :

a.

$$\sigma_{i_j} = \left(b_i - a_i^T x_j\right) / \left(a_i^T s_j\right)$$

$$\sigma_{u_j} = min\{\sigma_{i_j} \mid \sigma_{i_j} \geq 0\}$$

$$\sigma_{v_j} = max\{\sigma_{i_j} \mid \sigma_{i_j} \leq 0\}$$

b. $J_j = J_{j-1} \bigcup \{u,v\}$

(The Run Step) : Set j $\leftarrow$ j+1, $x_j \leftarrow x_{j+1}$ and check all stopping criteria. If termination is not due then go to the Hit Step.

(Solution) : Necessary inequalities are $J_j$

Table 4.1 The HD Hit-and-Run Algorithm.

The time complexity of a Hit-and-Run method is defined to be the time required to complete one iteration of the algorithm. According to reference [1] the Hit-and-Run method has time complexity O(mn).

## 4.3 The Set Covering (SC) Equivalence Approach

The basic idea of the Set-Covering equivalence for identifying redundant constraints has been suggested by Boneh [4].

The first step in the Set-Covering approach is to randomly generate points in $R^n$, and to each point assign a binary word of length m where m is the number of constraints. The k-th digit of the binary word is unity if and only if the k-th constraint is violated by the given point. The list of distinct binary words forms the rows of a set covering matrix. Any feasible solution to the set covering problem can be used to classify each constraint as either necessary or redundant. The approach can be summarized as follows:

(Input): I and G(I).

(Initialization) : Set i $\leftarrow$ 1.

(Generation of Observations) : Generate a point $x_i \in R^n$ either at random or by a specific strategy. For all j , j = 1, ..., m. Evaluate $a_j^T x_i$ and the corresponding binary word $e_i = (e_{i_1}, \ldots, e_{i_m})$ such that $e_{i_j}$ is unity if and only if $a_j^T x_i > b_j$.

(Acceptance-Rejection) : If $e_i$ is a new binary word then insert the binary word $e_i$ as a new row in the matrix E.

(Termination) : Set i $\leftarrow$ i+1 and check all stopping criteria. If termination is not due then go to (Generation of Observations).

Generate the set covering problem associated with the matrix E.

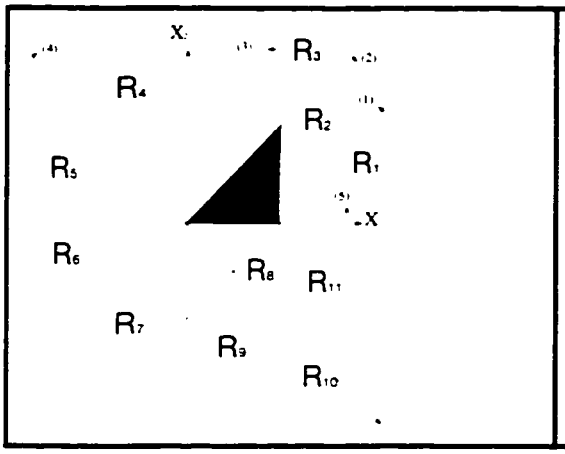
(Solution) : Solve the generated set covering problem. The solution of the generated set covering problem is an m-vector that can be used to classify each constraint as either redundant or necessary. If $Y^y = (y_1, \ldots, y_m)$ is a solution to the set covering problem, then if $y_i = 1$ implies that constraint i is necessary and $y_j = 0$ implies that constraint j is redundant.

Table 4.2 The Set-Covering Approach.

### 4.3.1 Example

The example presented here is small enough so that the results can be verified

by inspection. Consider the following figure with the set of five linear constraints.




1. $a_1^T x = x_1 - x_2 \leq 8.$
2. $a_2^T x = -x_1 + x_2 \leq 0.$
3. $a_3^T x = x_1 \leq 8.$
4. $a_4^T x = -x_1 - x_2 \leq 0.$
5. $a_5^T x = -x_2 \leq 0.$

Figure 4.5 A Set-Covering Example

The set of constraints defines a partition of $R^2$ into 12 distinct regions. Each region corresponds to a single binary word. The corresponding 12x5 matrix contains one row for each of these regions. The set covering matrix is

$$E = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Notice that E does not contain the zero binary word generated by region $R_{12}$. This word indicates a feasible region. It is not included in the set-covering problem as it does not contribute to the classification of constraint as redundant or necessary. Now the problem of finding a minimal feasible set or a minimal infeasible set can be formulated as a set covering problem and when formulated it reads as follows:

$$min\ cy$$

$$s.t.:\ Ey \geq 1$$

where c is taken to be a vector of ones throughout this thesis.

The number of distinct rows in E is 12. The number of nonzero entries in E is 27 and its density is therefore 27÷60 (which is approximately 50 %). It is evident from the figure that constraints (1) and (4) are redundant. The set of all feasible solutions of the set-covering problem in the example is: {11111, 01101, 10111, 11101} where 11111 is the trivial solution, and 01101 is the optimal solution which corresponds to an irreducible solution, where by irreducible solution we mean the solution is a minimal feasible set or a minimal infeasible set. The solution y = (0,1,1,0,1) indicates that constraints 2, 3, and 5 are necessary constraints.

## 4.4 The Main Theorem

As we have seen from the previous example, the set covering approach is

essentially a method of sampling words from the set of all possible binary words that correspond to the regions defined by a system of linear inequalities. It is also the interpretation of these binary words.

According to Boneh [4], if we consider all the constraints violated by an infeasible point $x_i \in R_n$. We can show that these constraints can not be a subset of the redundant constraints. More precisely we have the following theorem.

### 4.4.1 Theorem

If $x_j$ is an infeasible point in $R^n$ then at least one of the constraints violated by the point $x_j$ is necessary in any irreducible set of constraints.

Proof : Assume to the contrary that all the constraints violated by the point $x_i$ are redundant. According to the definition on page 5 (definition of redundant constraints) all these redundant constraints can be removed without altering the feasible region R. But when all these constraints are removed, the point $x_i$ becomes feasible (since it fulfills all the constraints which are not removed) while $x_i$ is initially assumed to be infeasible point, hence a contradiction which completes the proof.

Consequently, a solution to the set covering problem can be used to classify the constraints as redundant or necessary.

## 4.5 Conclusion

In this chapter we have reviewed some of the existing approaches for removing redundancy. These approaches deal with removing redundancy only. Our method, the method given in chapter 5, differs from these existing approaches in that it simultaneously deals with redundancy and feasibility.

# Chapter 5 A METHOD FOR CLEANING CONTAMINATED SYSTEMS OF LINEAR INEQUALITIES

## 5.1 Introduction

The method to be described in this chapter is designed to find a minimal representation of a system of linear inequalities that represents a feasible region to a mathematical programming problem. Fundamental to the method is a new technique to generate the set covering matrix associated with a given system of linear inequalities, and an equivalence of the set-covering approach to the infeasibility problem.

## 5.2 Exposition of the Approach

Basically the method cleans contaminated systems of linear inequalities by finding a feasible solution to the set-covering problem associated with the system under consideration. The approach to clean contaminated systems of linear inequalities is a new method to deal with both problems, detecting a minimal infeasible constraint set and identifying and removing redundant constraints, simultaneously. First, the approach tests the system under consideration for feasibility by examining the rows of the generated set covering matrix. A row of zeros in the set covering matrix corresponds to a region which fulfils all constraints, i.e., a non-empty feasible region. If the system is feasible, then

the solution of the associated set covering problem is used to identify a minimal feasible set. If the system is infeasible, i.e., if there is no row of zeros in the set covering matrix, then the solution of the associated set covering problem is used to identify a minimal infeasible set. The system might then be made feasible by deleting or modifying the constraints belonging to the minimal infeasible set.

## 5.3 A Probabilistic Set Covering Approach to the Minimal Representation Problem

Input : Given $a_i \in R^n$, i = 1, ..., m, $b_i \in R$.

Step 0: (Initialization):

a. Generate $x_0 = (x_{01}, ..., x_{0j}, ..., x_{0n})^T$ uniformly distributed in the n dimensional hypercube centered at the origin with side length 2, i.e., $x_{0j}$ uniformly distributed over (-1,+1), j = 1, ..., n.

b. Calculate the m-bit binary word, $e_0 = (e_{0_1}. .... e_{0_i}. .... e_{0_m})$ for $x_0$. Where

$$\begin{cases} e_{0_i} = 0. \ if \ a_i^T x_0 \leq b_i \\ e_{0_i} = 1. otherwise \end{cases}$$

c. Compute $e_0$'s equivalent decimal value, $d_0 = \sum_{s=1}^{m} e_{0_s}(2)^{m-s}$.

d. Initialize the binary search tree B, i.e.,insert $d_0$ in B.

e. Initialize the set covering matrix E, i.e., Set E= $[e_0]$.

f. Set k = 0.

Step 1: (Generation of Observations):

a. Generate a direction vector $s_k$ uniformly distributed over the surface of the n dimensional unit hypersphere centered at the origin, that is,

- Choose $z_{kj}$ to be N(0,1), j = 1, ..., n.
- Let $z_k = (z_{k1}, ..., z_{kn})^T$.
- Set $s_k = z_k/\|z_k\|$. Where $\|z_k\| = \sqrt{z_{k1}^2 + ... + z_{kn}^2}$.

b. Define $L(x_k, s_k) = \{x_k + \sigma s_k \mid \sigma \in R\}$.

c. Compute the distance from $x_k$ along $L(x_k, s_k)$ to the boundary of the i-th inequality $a_i^T x \leq b_i$. That is, for i = 1, ..., m calculate

$$\sigma_{ki} = \left(b_i - a_i^T x_k\right) / \left(a_i^T s_k\right)$$

d. Sort $\sigma_{ki}$'s , i.e., create an index set $\alpha_1, .., \alpha_m$ such that $\sigma_{k\alpha_1} \leq \sigma_{k\alpha_2} \leq ... \leq \sigma_{k\alpha_l} \leq 0 \leq \sigma_{k\alpha_{l-1}} \leq ... \leq \sigma_{k\alpha_m}$

e. Generate $\sigma_k$ uniformly distributed over $(\sigma_{k\alpha_1} - \epsilon, \sigma_{k\alpha_m} + \epsilon)$. Where $\epsilon = (\sigma_{k_{\alpha m}} - \sigma_{k_{\alpha 1}})/m$.[1]

f. Determine p such that $\sigma_{k\alpha_p} \leq \sigma_k \leq \sigma_{k\alpha_{p-1}}$

Step 2: (Acceptance-Rejection Step):

a. Generate the m-bit binary word $e_k^{\alpha_j}$ (j = 1, ..., m) associated with every region through which $L(x_k, s_k)$ passes. There are m such regions

[1] When we tested our algorithm we used $\epsilon$ to be uniformly distributed over (0,1). We did not have any reason for that choice, we now suggest $\epsilon = (\sigma_{k_{\alpha m}} - \sigma_{k_{\alpha 1}})/m$ as a better and more appropriate choice.

(excluding the region containing $x_k$ which has binary word $e_k$). Let $e_k^{\alpha_j}$ denote the binary word in the region along $L(x_k, s_k)$ determined by $\alpha_j$ and $\alpha_{j+1}$ and let $e_{k_i}^{\alpha_j}$ denote the i-th component of $e_k^{\alpha_j}$.

- For j = $l$+1, ..., m.
  - Initialize $e_k^{\alpha_j} = e_k^{\alpha_{j-1}}$ (for j = l+1, initialize $e_k^{\alpha_j} = e_k$).
  - If $e_{k_i}^{\alpha_{j-1}} = 0$, then set $e_{k_i}^{\alpha_j} = 1$ and do nothing because the newly generated binary word is redundant. See section 5.5.
    - If j = p, then set $e_p = e_k^{\alpha_j}$.
  - Else $e_{k_i}^{\alpha_{j-1}} = 1$, and set $e_{k_i}^{\alpha_j} = 0$.
    - If j = p, then set $e_p = e_k^{\alpha_j}$.
    - Check if $e_k^{\alpha_j}$ is a new binary word, compute $e_k^{\alpha_j}$'s equivalent decimal value, $d_{\alpha_j} = \sum_{s=1}^{m} e_{k_s}^{\alpha_j}(2)^{m-s}$.
    - Check if $d_{\alpha_j}$ is already in the binary search tree, B. If $d_{\alpha_j}$ is not already in the binary search tree, i.e., if $d_{\alpha_j}$ is inserted in the tree, then insert $e_k^{\alpha_j}$ as a new row in the set covering matrix E.

b. • For j = $l$, ..., 1.
  - Initialize $e_k^{\alpha_j} = e_k^{\alpha_{j-1}}$ (for j = $l$, initialize $e_k^{\alpha_j} = e_k$).

- If $e_{k_i}^{\alpha_j - 1} = 0$, then set $e_{k_i}^{\alpha_j} = 1$ and do nothing because the newly generated binary word is redundant. See section 5.5.

  - If j = p, then set $e_p = e_k^{\alpha_j}$.

- Else $e_{k_i}^{\alpha_j - 1} = 1$ and set $e_{k_i}^{\alpha_j} = 0$.

  - If j = p, then set $e_p = e_k^{\alpha_j}$.
  - Check if $e_k^{\alpha_j}$ is a new binary word, compute $e_k^{\alpha_j}$'s equivalent decimal value, $d_{\alpha_j} = \sum_{s=1}^{m} e_{k_s}^{\alpha_j}(2)^{m-s}$.
  - Check if $d_{\alpha_j}$ is already in the binary search tree, B. If $d_{\alpha_j}$ is not already in the binary search tree, i.e., if $d_{\alpha_j}$ is inserted in the tree, then insert $e_k^{\alpha_j}$ as a new row in the set covering matrix E.

c. Remove all newly redundant rows from the set covering matrix E. See Section 5.5 part 2.

Step 3: a. $x_{k+1} = x_k + \sigma_k s_k$. Where $s_k$ comes from step 1a and $\sigma_k$ come from step 1e.

b. Set the m-bit binary word, $e_k$, associated with $x_k$ to $e_p$, i.e., $e_k = e_p$. where $e_p$ comes from either step 2a or step 2b.

c. k = k+1.

d. Check all stopping criteria

- The total number of iterations (e.g. 1000 iterations).

- The number of iterations we did go without adding a new row to the set covering matrix.

If termination is not due, then go to step 1.

Step 4: Using the set covering matrix E, generate the associated SC problem.

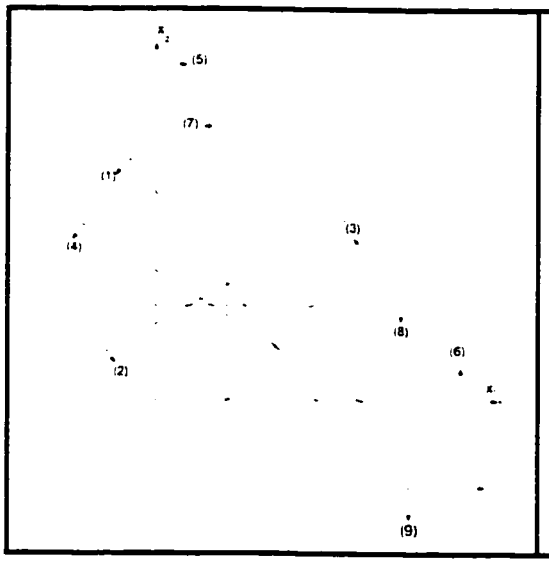
Step 5: (Solution): Find a solution to the generated SC problem using Chvatal's algorithm [15].

Step 6: Use the solution of SC problem to find a minimal representation to the original linear programming problem.

## 5.4 Example

In this section we present an example in order to clarify the method and terminology.

Consider Figure 5.6 with the set of nine linear constraints which defines an empty feasible region.




1. $a_1^T x = 2x_1 + x_2 \leq 7$
2. $a_2^T x = -x_1 + 2x_2 \leq 4$
3. $a_3^T x = x_1 - x_2 \leq 2$
4. $a_4^T x = x_1 + x_2 \leq 4$
5. $a_5^T x = -x_1 \leq 0$
6. $a_6^T x = -x_2 \leq 0$
7. $a_7^T x = x_1 \leq 2$
8. $a_8^T x = x_2 \leq 2.5$
9. $a_9^T x = x_2 \leq -2.5$

Figure 5.6

The values that will be used here are the actual values we obtained from testing our method using this example.

Step (0:a) Generate an initial point $x_0 = (-0.7879, -0.98520)^T$

Step (0:b) Calculate the m-bit binary word associated with $x_0$ we get $e_0$ = (0, 0, 0, 0, 1, 1, 0, 0, 1)

Step (0:c) Compute $e_0$'s equivalent decimal value, $d_0$, $d_0 = \sum_{s=1}^{m} e_{0s}(2)^{m-s}$. We get $d_0 = 25$

Step (0:d) Initialize the binary search tree B, i.e., insert $d_0 = 25$ in the binary search tree B.

Step (0:e) Initialize the set covering matrix E, i.e., Set E= $[e_0]$ = [0, 0, 0, 0, 1, 1, 0, 0, 1]

Step (0:f) Set k = 0.

In step 1: (Generation of Observation): we perform the following substeps:

Step (1:a) Generate a direction vector $s_0 = (-0.9506, -0.3103)^T$. As it can be seen in the follwoing figure (Figure 5.7) $x_0$ and $s_0$ defines the line $L(x_0 , s_0)$ that intersects the constraint boundaries.

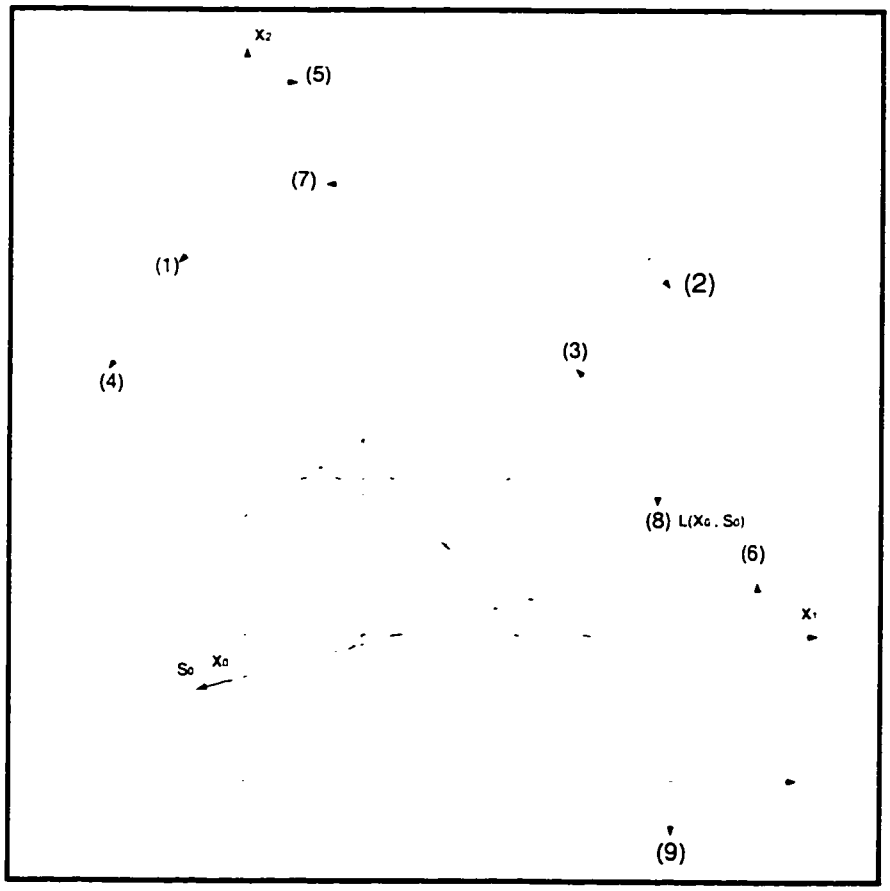



Figure 5.7

Step (1:c) Compute the distance from $x_k$ along $L(x_0, s_0)$ to the boundary of the i-th inequality

$a_i^T x \leq b_i$ for i = 1, .... m, i.e , calculate

$$\sigma_{ki} = \left(b_i - a_i^T x_k\right) / \left(a_i^T s_k\right).$$

We get : $\sigma_{01} = -4.3231$, $\sigma_{02} = 15.7028$, $\sigma_{03} = -2.8153$, $\sigma_{04} = -4.5783$, $\sigma_{05} = -0.8288$, $\sigma_{06} = -3.1748$, $\sigma_{07} = -2.9326$, $\sigma_{08} = -11.2315$ and $\sigma_{09} = 4.8818$.

Sort $\sigma_{ki}$'s in increasing order. We get the following: $\sigma_{08}$ = -11.2315, $\sigma_{04}$ = -4.5783, $\sigma_{01}$ = -4.3231, $\sigma_{06}$ = -3.1748, $\sigma_{07}$ = -2.9326, $\sigma_{03}$ = -2.8153, $\sigma_{05}$ = -0.8288, $\sigma_{09}$ = 4.8818 and $\sigma_{02}$ = 15.7028.

We have $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8, \alpha_9\} = \{8, 4, 1, 6, 7, 3, 5, 9, 2\}$ with $l = 7$ and $\alpha_l = 5$ and $l+1 = 8$ with $\alpha_{l+1} = 9$.

The constraint boundaries partition $L(x_0, s_0)$ into 10 regions (including the region that contains $x_0$, $R_3$ in the Figure 5.8). We want to sample a point from each region and we do this in two steps that corresponds to the (Acceptance-Rejection Step) in the algorithm.

Note: Figure 5.8 does not show the exact representation of the constraint set. We use this figure for illustrative purposes only.

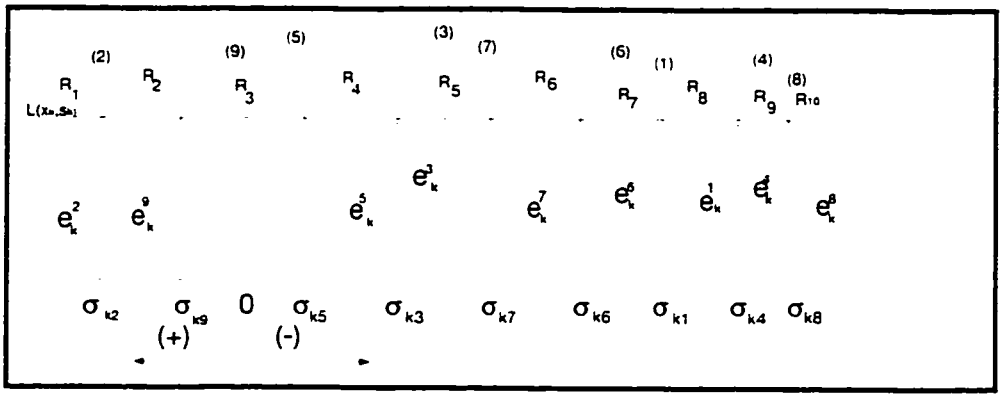



Figure 5.8

In step 2 (Acceptance-Rejection Step), we do the following: Starting with sorted $\sigma_{ki}$'s values, we divide $\sigma_{ki}$'s values into two subsets; positive $\sigma_{ki}$'s values and negative $\sigma_{ki}$'s values. We get the following positive $\sigma_{ki}$'s: $\sigma_{09}$ = 4.8818 and $\sigma_{02}$ = 15.7028 and the following negative $\sigma_{ki}$'s: $\sigma_{08}$ = -11.2315, $\sigma_{04}$ = -4.5783, $\sigma_{01}$ = -4.3231, $\sigma_{06}$ = -3.1748, $\sigma_{07}$ = -2.9326, $\sigma_{03}$ = -2.8153, $\sigma_{05}$ = -0.8288.

Now starting with $x_0$ in $R_3$ which has a binary word $e_k$ = (0, 0, 0, 0, 1, 1, 0, 0, 1) and moving in the negative direction, i.e., for i = $\alpha_7$, ..., $\alpha_1$. We perform the following:

- We cross the boundary of constraint 5 in going from $R_3$ to $R_4$. We set the binary word $e_k^5 = e_k$ = (0, 0, 0, 0, 1, 1, 0, 0, 1). Since $e_{k5}^5$ = 1 then we set $e_{k5}^5$ = 0 and the binary word to (0, 0, 0, 0, 0, 1, 0, 0, 1). In this case we compute $e_k^5$'s equivalent decimal value, $d_5$ = 9. We check if $d_5$ is already in the binary search tree, B. At this point the binary search tree

B contains 25. Hence $d_5 = 9$ is not in the binary search tree, i.e., it is inserted in the tree, then we insert the binary word $e_k^5$ as a new row in the set-covering matrix E.

- Now we cross the boundary of constraint 3 in going from $R_4$ to $R_5$. We set the binary word $e_k^3 = e_k^5 = (0, 0, 0, 0, 0, 1, 0, 0, 1)$. Since $e_{k,3}^3 = 0$ then we set $e_{k,3}^3 = 1$, and the binary word $e_k^3$ to (0, 0, 1, 0, 0, 1, 0, 0, 1) and do nothing because the newly generated binary word that corresponds to $R_5$ is redundant.

- Now we cross the boundary of constraint 7 in going from $R_5$ to $R_6$. We set the binary word $e_k^7 = e_k^5 = (0, 0, 1, 0, 0, 1, 0, 0, 1)$. Since $e_{k,7}^7 = 0$ then we set $e_{k,7}^7 = 1$ and the binary word $e_k^7$ to (0, 0, 1, 0, 0, 1, 1, 0, 1) and do nothing because the newly generated binary word that corresponds to region $R_6$ is redundant.

- Now we cross the boundary of constraint 6 in going from $R_6$ to $R_7$. We set the binary word $e_k^6 = e_k^7 = (0, 0, 1, 0, 0, 1, 1, 0, 1)$. Since $e_{k,6}^6 = 1$ then we set $e_{k,6}^6 = 0$, and the binary word $e_k^6$ to (0, 0, 1, 0, 0, 0, 1, 0, 1). In this case we compute $e_k^6$'s equivalent decimal value, $d_6 = 69$. We check if $d_6$ is already in the binary search tree,B. At this point the binary search tree B contains 25 and 9. Hence $d_6$ is not in the binary search tree, so $d_6$ is inserted in the binary search tree, then we insert the binary

word $e_k^6$ in the set-covering matrix E.

- Now we cross the boundary of constraint 1 in going from $R_7$ to $R_8$. We set the binary word $e_k^1 = e_k^6 = (0, 0, 1, 0, 0, 0, 1, 0, 1)$. Since $e_{k_1}^1 = 0$, then we set $e_{k_1}^1 = 1$, and the binary word $e_k^1$ to (1, 0, 1, 0, 0, 0, 1, 0, 1) and do nothing because the newly generated binary word that corresponds to $R_8$ is redundant.

- Now we cross the boundary of constraint 4 in going from $R_8$ to $R_9$. We set the binary word $e_k^4 = e_k^1 = (1, 0, 1, 0, 0, 0, 1, 0, 1)$. Since $e_{k_1}^4 = 0$, then we set $e_{k_1}^4 = 1$, and the binary word $e_k^4$ to (1, 0, 1, 1, 0, 0, 1, 0, 1) and do nothing because the newly generated binary word that corresponds to $R_9$ is redundant.

- Now we cross the boundary of constraint 8 in going from $R_9$ to $R_{10}$. We set the binary word $e_k^8 = e_k^4 = (1, 0, 1, 1, 0, 0, 1, 0, 1)$. Since $e_{k_8}^8 = 0$, then we set $e_{k_8}^8 = 1$, and the binary word $e_k^8$ to (1, 0, 1, 1, 0, 0, 1, 1, 1) and do nothing because the newly generated binary word that corresponds to $R_{10}$ is redundant.

Now starting with $x_0$ in $R_3$ which has a binary word $e_k$ = (0, 0, 0, 0, 1, 1, 0, 0, 1) and moving in the positive direction, i.e., for i = $\alpha_8$ and $\alpha_9$. We perform the following:

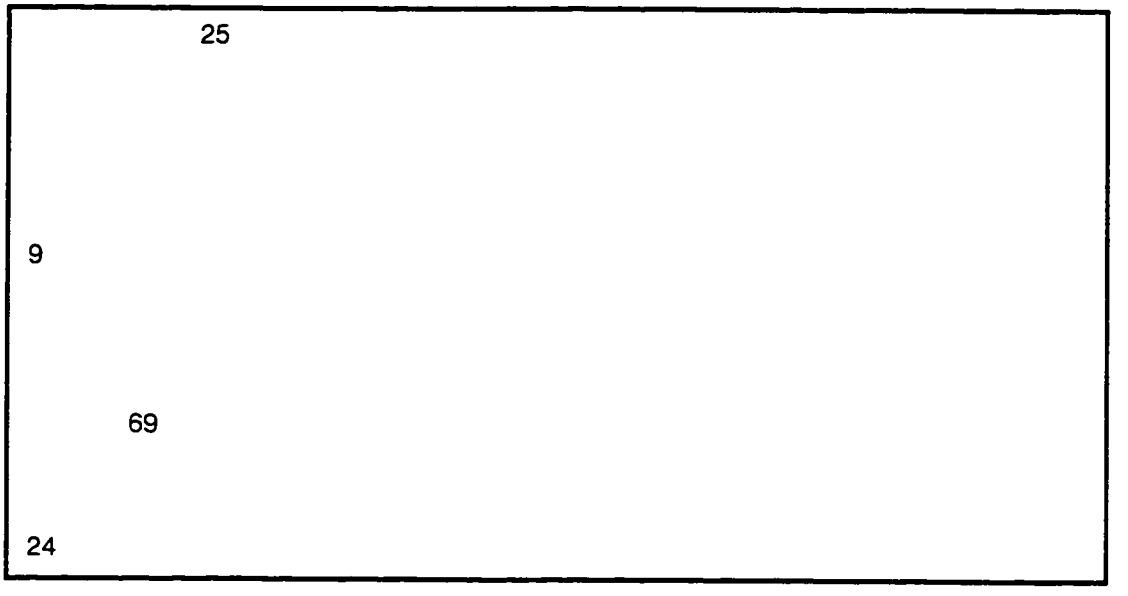
- We cross the boundary of constraint 9 in going from $R_3$ to $R_2$. We set the binary word $e_k^9 = e_k = (0, 0, 0, 0, 1, 1, 0, 0, 1)$ . Since $e_{k_9}^9 = 1$ , then we set $e_{k_9}^9 = 0$ and the binary word $e_k^9$ to (0, 0, 0, 0, 0, 1, 1, 0, 0), we compute $e_k^9$'s equivalent decimal value, $d_9 = 24$, and we check if $d_9$ is already in the binary search tree, B. At this point the binary search tree B contains 25, 9, and 69. Hence $d_9$ is not in the binary search tree. We insert it in the tree, then we insert the binary word $e_k^9$ in the set-covering matrix E.

- Now we cross the boundary of constraint 2 in going from $R_2$ to $R_1$. We set the binary word $e_k^2 = e_k^9 = (0, 0, 0, 0, 0, 1, 1, 0, 0)$. Since $e_{k_2}^2 = 0$, then we set $e_{k_2}^2 = 1$ and $e_k^2 = (0, 1, 0, 0, 0, 1, 1, 0, 0)$ and do nothing because the newly generated binary word that corresponds to region $R_1$ in Figure 5.8 is redundant.

After this iteration we have the following set-covering matrix :

$$E = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

and the following binary tree :




In step 3: We update k and $x_k$ and go back to step 1 (Generation of observation).

The method continues in this fashion until a stopping criteria is met (the number of iterations, 1000 iterations, was used as a stopping criteria in this excample). After 1000 iterations, the binary search tree B contains the following values: 25, 9, 1, 5, 3, 24, 17, 69, 37, 33, 35, 39, 76, 72, 73, 152, 145, 77, 131, 129, 88, 147, 357, 293, 179, 163, 295, 325, 332, 333, 364, 359, 419 and 423. The set-covering matrix appears as follow:

$$E = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0
\end{bmatrix}$$

**Comment**: We have 34 entries in the tree, but we have less then 34 rows in E. Once we have the set-covering matrix E, we remove redundant rows from the set-covering matrix. Then we generate the set-covering problem associated with E and find a solution to the generated set-covering problem. Using the

set-covering problem solution, the method outputs a minimal set of constraints that represents the original problem.

For this example, the output of the method was the following set of constraints {9, 6, 5, 8} which is not a minimal infeasible set.

## 5.5 Points To Rember When Using The Set-Covering Approach

1. While we are sampling points from the set of all possible points, what if a feasible point is found, i.e., if all $e_{k_i}^{a_j}$ are zero, for i = 1, ..., m. Then we stop the set-covering approach and we use the regular hit-and-run algorithm. In the hit-and-run algorithm we sample only those points with a single 1 in its corresponding binary word. Knowing this (By the theorem 4.4.1 ), we classify the corresponding constraints as necessary.
2. Every time we generate a new binary word $e_k^{a_j}$, (j = 1, ..., m). If a zero in position i is changed to one then we know the newly generated binary word is redundant. By redundant we mean the binary word is covered by an existing binary word. A binary word is covered by another if for i = 1, .., m, if $e_{k_i}^{a_{j-1}} > e_{k_i}^{a_j}$, where $e_{k_i}^{a_j}$ is the i-th entry in the j-th binary word . In our example, in moving from $R_8$ to $R_9$ we had $e_k^1 = (1.0.1.0.0.0.1.0.1)$ for $R_8$ and $e_k^4 = (1.0.1.1.0.0.1.0.1)$ for $R_9$. Note that the 4th bit in $e_k^1$ was changed from 0 to 1 to get $e_k^4$. If y is a set-covering solution with $(e_k^1)^T y \geq 1$, then it

follows that $\left(c_k^4\right)^T y \geq \left(c_k^1\right)^T y \geq 1$ as well. Thus $\left(c_k^4\right)^T y \geq 1$ is a redundant constraint in the set-covering problem, and it is removed.

## 5.6 Special Aspects of the Approach

The approach is a new method that can be used to identify redundancy and/or to identify a minimal infeasible set in a system of linear constraints. Given that it is a probabilistic approach it does not guarantee a minimal representation. This approach differs from the other existing approaches in that it does not require prior knowledge about the feasibility of the system. When our algorithm is finished, we have concluded with either:

1. A minimal infeasible set.
2. A minimal set of necessary constraints.

In the first case, we have as output an irreducible infeasible set. In fact, this conclusion could be wrong. However, we can now apply the IIS method of Chinneck and Dravnieks [14] on a much smaller constraint set. So our approach can be considered as a preprocessor.

In the second case, we have a feasible region. However, some necessary constraints may have been incorrectly classified as redundant. In spite of this, we solve the reduced LP problem. We then check the LP solution for feasibility with respect to the entire constraint set. If it is feasible, great. If not, we add back in the violated constraints and resolve using a dual simplex algorithm starting with

the infeasible "optimal solution". In this case our approach is a preprocessor for the LP code.

## 5.7 Time-Complexity

We did not do a time-complexity analysis of our set-covering approach. As was seen in previous chapter the approach iteself is a combination of more than one algorithm. Some of the algorithms have a well-known time-complexity such as the set-covering algorithm and the hit-and-run algorithm. We have chosen a version of the hit-and-run algorithm that has a per iteration time-complexity O(mn). As for the set-covering algorithm, it is well known that it is an NP-hard problem. For this reason we have chosen a heuristic algorithm to find a solution to the set-covering problem that has a linear time-complexity [15]. As for the other steps of the algorithm many of them involve matrix operations, such as multiplication, addition, inversion and subtraction, with a very well-known time complexity. In order to minimize the number of matrix operations we stored our data in PF format [11]. PF format is a packed format for storing equalities and inequalities. PF format uses the vectors A, and AINDEX. The float vector A has length AMAX and contains the non-zero constraint coefficients and the right hand sides. The constraints are stored sequentially. The integer vector AINDEX has length AMAX and contains the index of the pointers indicating the position in A

and AINDEX where data about the constraints begins and ends. As an example, consider the system given by:

1. $3x_1 + 4x_7 \leq 9$

2. $2x_1 + x_3 \leq 4$

3. $x_1 \leq 0$

Here we have AMAX = 8, and A, AINDEX as given below:

| $A$ | $AINDEX$ |
|---|---|
| 9 | $-3$ |
| 3 | 1 |
| 4 | 7 |
| 4 | $-6$ |
| 2 | 1 |
| 1 | 3 |
| 0 | $-8$ |
| 1 | 1 |

Since AINDEX(1) = $-3$ data on the first constraint ends at I = 3. Since AINDEX(4) = $-6$ data on the second constraint ends at I = 6. Since AINDEX(7) = $-8$. Therefore, we know that, for example, the data for constraint 3 is contained in position 7 to 8 of A and AINDEX. A(7) = 0, so the right-hand side for constraint 3 is 0. A(8) = $-1$ and AINDEX(8) = 1 so coefficient of $x_1$ in constraint 3 is 1.

# Chapter 6 PROGRAMMING THE METHOD

## 6.1 Introduction

The C computer language was selected for programming because of its widespread use and because of its powerful tools to create and maintain dynamic data structures. In programming the method, special care was taken to minimize the effects of any programming bias, such as memory allocation. Common subroutines were used where possible for steps that are common, such as binary search and quicksort algorithms.

$C^{++}$ is a superset of C, that is, programmers can use a $C^{++}$ compiler to compile C programs. Therefore the C language was used mostly to program the method and when it was necessary the $C^{++}$ language was used in order to allocate memory efficiently.

We present some details of the programming process used for the method. This will enable the reader to gain a better understanding of the specifics such as memory space requirement and the order of operations. We begin this discussion by presenting the general algorithm and then discuss the details of programming every subroutine associated with it.

## 6.2 Programming The Set Covering Approach For The Minimal Representation Problem.

The approach has been described in detail in the previous chapter. In general the approach attempts to clean, remove redundant constraints and/or identify a minimal infeasible set in a system of linear inequalities by generating the set-covering problem associated with it.

The algorithm consists of four basic steps:

1. Initialization
2. Generation of observations,
3. Acceptance-Rejection,
4. Termination, and
5. Solution.

In the following sections, we present details of the programming process used for each step.

### 6.2.1 Step 1: Initialization

The first step of the algorithm consists of three initialization steps

1. Generating an initial starting point, $x_0$. The point was chosen to be uniformly distributed over the n-dimensional hypercube centered at the origin with length 2. Knuth's suggestion ([29], p. 280) was implemented to generate $x_0$.

2. Calculate the m-bit binary word, $e_0$, associated with the initial starting point $x_0$. This was done by direct substitution.
3. Initialize the set-covering matrix, i.e., Set E = $[e_0]$.
4. Initialize the binary tree B, i.e., insert $e_0$'s equivalent decimal value in B.

### 6.2.2 Step 2: Generation of Observations

The second step of the algorithm consists of generating the observations points. The strategy used for generating the observations was developed by using the principles of the hit-and-run algorithm and it consists of the following steps

1. Generate a direction vector $s_k$ normaly distributed over the surface of the n dimensional hypersphere. The Box-Muller method ([29], p. 289) for generating random deviates with a normal (Gaussian) distribution was implemented to generate $z_{ki}$, such that $s_k = z_k/\|z_k\|$ and $z_k = (z_{k1}, ..., z_{kn})^T$.
2. Define $L(x_k, s_k) = x_k + \sigma s_k$, $\sigma \in R$.
3. Compute the intersection points of $L(x_k, s_k)$ with the boundary of the i-th inequality $a_i^T x \leq b_i$ for i = 1, ..., m, i.e , calculate

$$\sigma_{ki} = \left(b_i - a_i^T x_j\right) / \left(a_i^T s_k\right).$$

4. Using quicksort [19], we sort $\sigma_{ki}$'s , i.e., create an index set $\alpha_1, .., \alpha_m$ such that $\sigma_{k\alpha_1} \leq \sigma_{k\alpha_2} \leq ... \leq \sigma_{k\alpha_i} \leq ... \leq \sigma_{k\alpha_m}$

### 6.2.3 Step 3: Acceptance-Rejection Step

The acceptance-rejection step consists of two steps.

1. Compute $e_i$'s equivalent decimal value, $d_i$. This is done by a simple routine following the general rule for converting a binary number to decimal.
2. Check if $d_i$ is already in a binary search tree with no duplicate. If $d_i$ is inserted (details of inserting a node in a binary search tree will be given in the following section), then insert $e_i$ as a new row in the set covering matrix E.

#### 6.2.3.1 Inserting A Node In A Binary Search Tree

A special binary tree called a binary search tree with no duplicates is created. A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node. Figure 6.9 illustrates a binary search tree with 12 values. Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

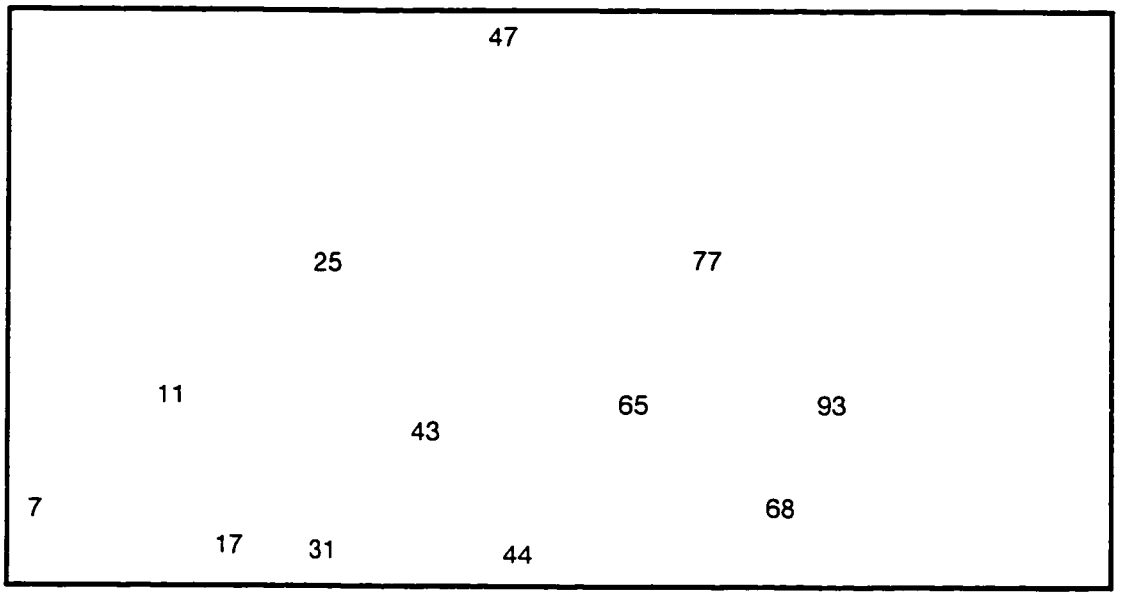



Figure 6.9 A Binary Search Tree

The steps for inserting a node in a binary search tree are as follows [19].

1. If *treePtr is NULL, create a new node. Call malloc, assign the allocated memory to *treePtr, assign to (*treePtr)->data the integer to be stored, assign to (*treePtr)->leftPtr and (*treePtr)->rightPtr the value NULL, and return to the caller (either main or a previous call to insertNode).
2. If the value of *treePtr is not NULL and the value to be inserted is less than (*treePtr)->data, function insertNode is called with the address of (*treePtr)->leftPtr. Otherwise, function insertNode is called with the address of (*treePtr)->rightPtr. The recursive steps continue until a Null pointer is found, then step 1 is executed to insert the new node.

### 6.2.4 Step 4: Termination

The third step of the algorithm is a decision step. It consists of whether we continue or terminate the algorithm. The termination step consists of three basic steps which are straightforward:

Check all stopping criteria such as

- The total number of iterations

While programming the approach the only stopping criteria that was used is the total number of iterations ( 1000 iterations). If termination is not due then

a. Update the necessary variables, $x_k$ and k

b. Go to step 2.

Else

Go to step 5, the solution step.

### 6.2.5 Step 5: The Solution Step

The final step of the algorithm consists of finding a minimal feasible solution to the set-covering problem generated in the previous steps. The problem is solved using Chvatal's algorithm [15]. A general description of the algorithm is presented in Appendix A

## 6.3 Performance Monitoring

In order to evaluate the performance of our method, we ran test problems and recorded the iteration number, a feasible point, if founded, and the set of necessary constraints and/or a minimal infeasible set that has been detected. This information was recorded in Table 6.4.3.

## 6.4 Test Problems

We evaluated the performance of the method on two types of problems:

1. Feasible problems.

2. Infeasible problems.

We now present the results in Table 6.4.3.

| Flie name | m | n | Feasible point found | After how many iterations | J | # of total iterations |
|---|---|---|---|---|---|---|
| h1.txt | 22 | 20 | yes | 46 | 13 | 1000 |
| h2.txt | 8 | 2 | yes | 1 | 6 | 1000 |
| h3.txt | 25 | 10 | yes | 862 | 7 | 1000 |
| h4.txt | 27 | 10 | no | N/A | 12 | 1000 |
| h5.txt | 24 | 8 | yes | 862 | 7 | 1000 |
| h6.txt | 27 | 10 | no | N/A | 12 | 1000 |
| h7.txt | 29 | 14 | yes | 836 | 12 | 1000 |
| h8.txt | 31 | 16 | yes | 246 | 12 | 1000 |
| h9.txt | 31 | 18 | yes | 460 | 10 | 1000 |
| h10.txt | 33 | 20 | yes | 906 | 16 | 1000 |
| h11.txt | 30 | 18 | yes | 38 | 13 | 1000 |
| h12.txt | 31 | 20 | yes | 38 | 18 | 1000 |
| h13.txt | 33 | 20 | yes | 38 | 18 | 1000 |
| h14.txt | 37 | 21 | yes | 38 | 27 | 1000 |
| h15.txt | 39 | 24 | yes | 38 | 21 | 1000 |
| h16.txt | 31 | 6 | no | N/A | 18 | 1000 |
| h17.txt | 27 | 15 | no | N/A | 12 | 1000 |
| s5.txt | 24 | 5 | yes | 133 | 7 | 1000 |
| s6.txt | 16 | 20 | no | N/A | 6 | 1000 |
| s11.txt | 22 | 15 | yes | 3 | 13 | 1000 |
| s12.txt | 12 | 5 | no | N/A | 2 | 1000 |
| s13.txt | 14 | 12 | no | N/A | 2 | 1000 |
| s14.txt | 28 | 25 | no | N/A | 16 | 1000 |
| s15.txt | 20 | 16 | no | N/A | 4 | 1000 |

Table 6.4.3 Results

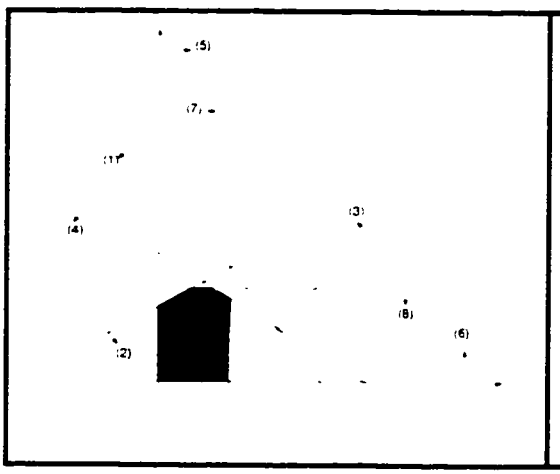
where m is number of constraints, n is the number of variables, and J is the cardinality of a minimal feasible/infeasible set.

As can be seen from Table 6.4.3, problems of different sizes were chosen. For the large size-problems we compared our results with the results in [11]. Our results were consistent with their results regarding the problems classification as

either infeasible or feasible. There was some inconsistency regarding, J. But we do not expect our results to be exactly the same as our algorithm is a probabilistic algorithm.

We choose small problems so we can verify our results by inspection. For those problems, feasible or infeasible, the results were accurate. As an illustration consider the following example (h2.txt in Table 6.4.3).

### 6.4.1 Examples



1. $a_1^T x = 2x_1 + x_2 \leq 7$
2. $a_2^T x = -x_1 + 2x_2 \leq 4$
3. $a_3^T x = x_1 - x_2 \leq 2$
4. $a_4^T x = x_1 + x_2 \leq 4$
5. $a_5^T x = -x_1 \leq 0$
6. $a_6^T x = -x_2 \leq 0$
7. $a_7^T x = x_1 \leq 2$
8. $a_8^T x = x_2 \leq 2.5$

Figure 6.10

As it can be seen from Figure 6.11, the system of linear inequalities defines a non-empty feasible region R. If we inspect the above figure we see that constraints (1) and (3) are redundant and constraints (2), (4), (5), (6), (7) and (8) are necessary. The result of our algorithm was as follows: After the first iteration, the algorithm reported that a feasible point is found and the original set-covering equivalence

approach was switched to the regular Hit-and-Run algorithm. After a 1000 iterations of the Hit-and-Run algorithm a minimum feasible set was reported as {2, 4, 5, 6, 7, 8} which is correct

Now consider the example given in Figure 5.7. As can be seen from the figure, the system of linear inequalities is inconsistent. The result of our algorithm was as follows: After the 1000–th iteration of the set-covering approach, no feasible point was found and a minimum infeasible set was reported as { 5, 6, 8, 9}.

If we examine Figure 6.11, the reduced set of constraints {5, 6, 8, 9} is still inconsistent. However, it is not a minimal infeasible set as constraint 5 could yet be removed.

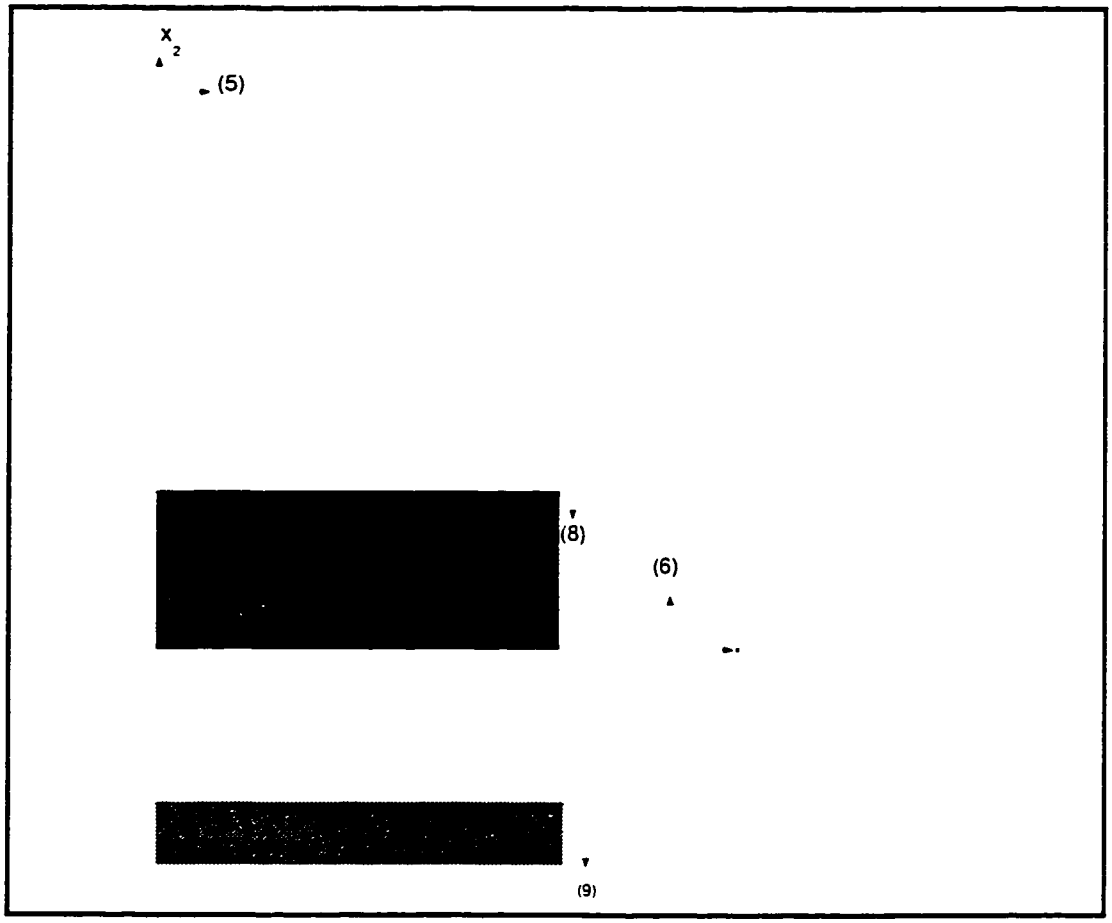



Figure 6.11

According to N. Chakravarti [13], the reduced set of constraints can be made feasible by removing one constraint. If we remove constraint (9), then the reduced set of constraints becomes feasible and we have a non-empty feasible region as it is shown Figure 6.12.

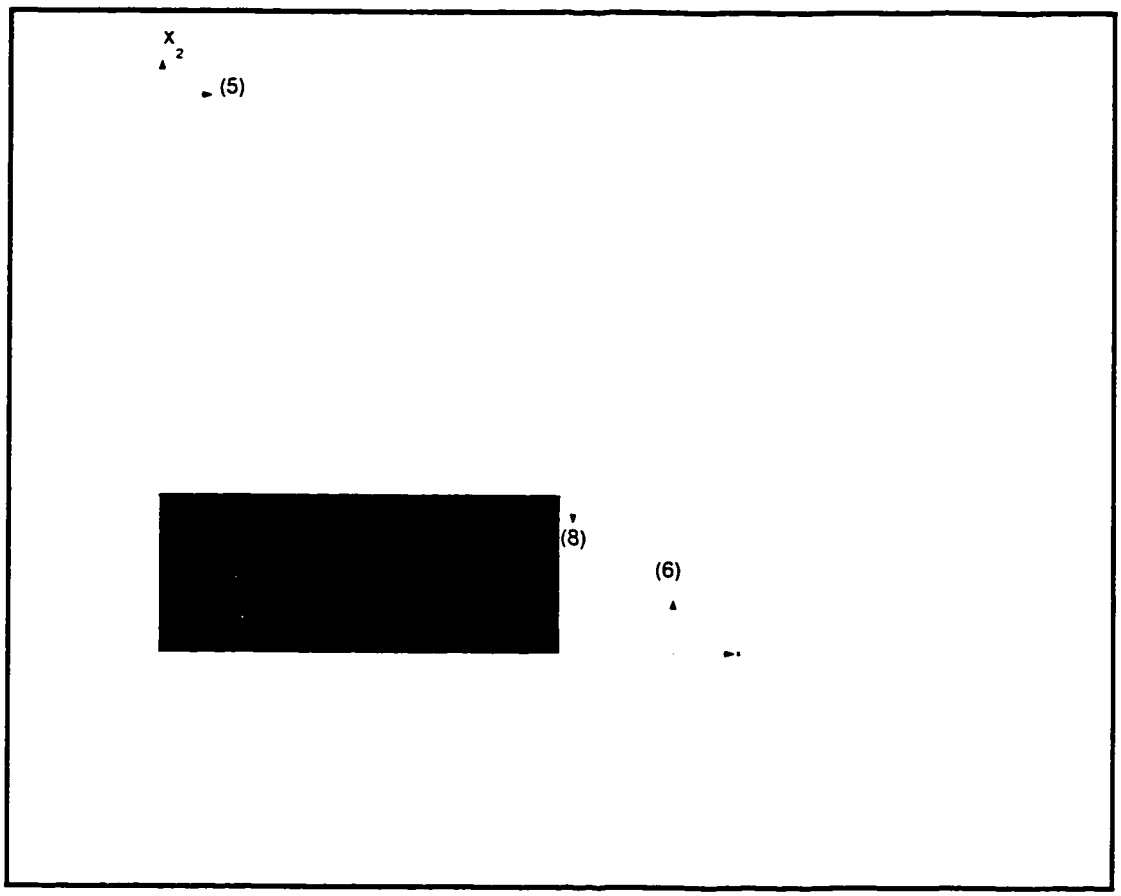



Figure 6.12

In these small examples it appears that the algorithm may be performing well, but true computational comparison with other methods awaits implementation.

# Chapter 7 CONCLUSION

## 7.1 Contributions of the Thesis Work and Completed Objectives

The first major contribution of this thesis work is that it has demonstrated that the minimum infeasible problem has a set covering equivalence.

The second major contribution of this thesis is the development of an efficient method for implementing Boneh's set covering equivalence to solve both problems: classifying linear inequalities as redundant or necessary and identifying minimal infeasible sets of linear inequalities. This thesis developed an efficient way to ensure that no duplicate binary words are inserted in the set-covering matrix. The thesis also provided a new approach to eliminate redundant observation points as they were generated. Details of this procedure have been explained in the previous chapter.

Finally, this thesis has provided computer-aided tool that can be used in conjunction with other existing deterministic algorithms to decide what can be done to clean a system of linear inequalities that represents a mathematical programming problem.

# BIBLIOGRAPHY


[1] Belisle, C. J. P., Boneh, A., and Caron, R. J. A uniformly recurrent linear hit-and-run algorithm. Research Report 0119/89, The University of Michigan. 1989.

[2] Berbee, H., Boender, C. C., Kan, A. R., Sheffer, C., Smith, R., and Telgen, T. Hit-and-run algorithms for the identification of nonredundant linear inequalities. *Mathematical Programming 37* (1987), 184–207.

[3] Boneh, A. Preduce-a probabilistic algorithm identifying redundancy by random feasible point generator (rfpg). In *M.H. Karwan and V. Lotfi and J. Telgen and S. Zionts*, redundancy in mathematical programming ed. Springer-Verlag, Berlin, 1983, pp. 108–134.

[4] Boneh, A. Identification of redundancy by a set-covering equivalence. In *Proceeding of the Tenth International Conference on Operational Research* (Washington, D.C., U.S.A., 1984), pp. 407–422.

[5] Boneh, A., and Caron, R. Constraints classification in mathematical programming. *Mathematical Programming 61* (1993), 61–74.

[6] Boneh, A., and Golan, A. Constraints redundancy and feasible region boundedness by a random feasible point generator (rfpg)". *paper presented at EURO III* (1979).

[7] Boneh, A., and Golan, A. Preduce- a probabilistic algorithm for identifying redundancy. In *Proceedings of the COAL Conference on Mathematical Programming Testing and Validation, Algorithms and Software* (Boulder, Colorado, Jan. 1981), National Bureau of Standards.

[8] Boot, J. C. G. On trivial and binding constraints in programming problems. *Management Science 8* (1962), 419–441.

[9] Caron, R., Hlynka, M., and McDonald, J. On the best case performance of hit and run methods for detecting necessary constraints. *Mathematical programming 54* (1992), 233–249.

[10]Caron, R., McDonald, J., Smith, R., and J.Telgen. Stopping rules for a class of random methods for detecting necessary linear inequality constraints. *Windsor Mathematics Report WMR 3* (1986), 1–17. Revised February, 1989. Submitted to Mathematical Programming.

[11]Caron, R. J., McDonald, J. F., and Pidgeon, R. A catalogue of lp test problems and systems of linear constraints. *Windsor Mathematics Report 16* (Nov. 1988).

[12]Caron, R. J., McDonald, J. F., and Ponic, C. M. A degenerate extreme point strategy for the classification of linear constraints as redundant or necessary. *Journal of Optimization Theory and Applications 62*, 2 (1989).

[13]Chakravarti, N. Some results concerning post-infeasibility analysis. *European Journal of Operational Research*, 73 (1994), 139–143.

[14]Chinneck, J. W., and Dravnieks, E. W. Locating minimal infeasible constraint sets in linear programs. *Operations Research Society of America (ORSA) 3*, 2 (Spring 1991), 157–168.

[15]Chvatal, V. A greedy heuristic for the set-covering problem. *Math. Operations Research 4*, 3 (1979), 233–235.

[16]Chvatal, V. *Linear Programming*. W.H. Freeman and Company, New York, 1983.

[17]C.J.P., B., H.E., R., and R.L., S. Hit-and-run algorithms for generating multivariate distributions. *Mathematical Operations Research 18* (1993), 255–266.

[18]Dantzig, G. *Linear Programming and Extensions*. Princeton University Press, Princeton, New Jersey, 1962.

[19]Deitel, H., and Deitel, P. *C How To Program*, second edition ed. Prentice-Hall, Inc, 1994.

[20]Gal, T. A method for determining redundant constraints. In *M.H. Karwan and V. Lotfi and J. Telgen and S. Zionts*, redundancy in mathematical programming ed. Springer-Verlag, 1983, pp. 36–52.

[21]Gleeson, J., and Ryan, J. Identifying minimally infeasible subsystems of inequalities. *ORSA Journal on Computing 2*, 1 (1990), 61–63.

[22]Greenberg, H. J. A functional description of analyze: A computer-assisted analysis system for linear programming models. *ACM Transactions on Mathematical Software 9*, 1 (1856).

[23]Greenberg, H. J. Diagnosing infeasibility for min-cost network flow models, part ii: Primal infeasibility. *IMA Journal of Mathematics Applied in Business and Industry 2* (1988), 1–12.

[24]Greenberg, H. J., and Murphy, F. H. Approaches to diagnosing infeasible linear programs. *ORSA Journal on Computing 3*, 3 (1991), 253–261.

[25]Karwan, M., Lotfi, V., Telgen, J., and Zionts, S. *Redundancy in Mathematical Programming*. Springer-Verlag, Berlin, 1983.

[26]Lisy, J. Metody pro nalezini redundantnich omezeni v ulohach linearniho programovani. *Ekonomicko Matematicky Obzor 7*, 3 (1971), 285–298.

[27]Loom, J. V. Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research 8* (1981), 283–288.

[28]Murty, K. G. *Linear Programming*. John Wiley and Sons, New York, 1983.

[29]Press, W. H., Vetterling, W. T., Flannery, B. P., and Teukolsky, S. A. *Numerical Recipes in C : the art of scientific computing*, second edition ed. Cambridge University Press, 1992.

[30]Roodman, G. M. Post-infeasibility analysis in linear programming. *Management Sciences 25*, 9 (Sept. 1980), 916–922.

[31]Rubin, D. S. Finding redundant constraints in sets of linear inequalities. In *M.H. Karwan and V. Lotfi and J. Telgen and S. Zionts*, redundancy in mathematical programming ed. Springer-Verlag, Berlin, 1983, pp. 60–67.

[32]Smith, R. L. Monte carlo procedures for generating random feasible solutions to mathematical programs. In *ORSA/TIMS Conference* (Washington D.C., 1980).

[33]Smith, R. L. Efficient monte carlo procedures for generating points uniformly distributed over bounded region. *Operations Research 32* (1984), 1296–1303.

[34]Telgen, J. Private communication with. A. Boneh, 1980.

[35]Telgen, J. Minimal representation of convex polyhedral sets. *Optimization Theory and Applications 38* (1982), 1–24.

[36]Telgen, J. Identifying redundancy in systems of linear constraints. In *M.H. Karwan and V. Lotfi and J. Telgen and S. Zionts*, redundancy in mathematical programming ed. Springer-Verlag, Berlin, 1983, pp. 53–59.

[37]Zionts, S. *Size Reduction Techniques of Linear Programming and Their Application*. PhD thesis, Carnegie Institute of Technology, 1965.

# APPENDIX A THE SET COVERING PROBLEM

## A.1 The Set Covering Problem

### A.1.1 Definitions and Notation

Consider a set I = {1,2, ...., m} and

a set P = { $P_1$, $P_2$, ..., $P_n$} where $P_j \subseteq I$, $j \in J = \{ 1, ..., n\}$.

A subset $J^* \subseteq J$ defines a cover of I if :

$$\bigcup_{j \in J^*} P_j = I$$

The set $J^*$ is refered to as a cover.

Let a cost $c_j > 0$ be associated with every $j \in J$. The cost of $J^* = \sum_{j \in J^*} c_j$.

The set covering problem is to find a cover of minimum cost and can be witten as Integer Linear Program (ILP):

$$min\ x_0 = \sum_{j=1}^{n} c_j x_j$$

$$\sum_{j=1}^{n} e_{ij} x_j \geq 1 \qquad i = 1, \ldots, m$$

$$x_j = 0, 1 \qquad j = 1, \ldots, n$$

where

$$x_j = \begin{cases} 1 & if\ j\ is\ in\ the\ cover \\ 0 & otherwise \end{cases}$$

$$e_{ij} = \begin{cases} 1 & if\ i \in P_j \\ 0 & otherwise \end{cases}$$

### A.1.2 Example

Let I = {1, 2, 3, 4, 5, 6, 7},

J = {1, 2, 3, 4},

P = {$P_1$, $P_2$, $P_3$, $P_4$},

$P_1$ = {1, 2, 3},

$P_2$ = {5, 6},

$P_3$ = {4} and

$p_4$ = {4, 5, 6, 7}.

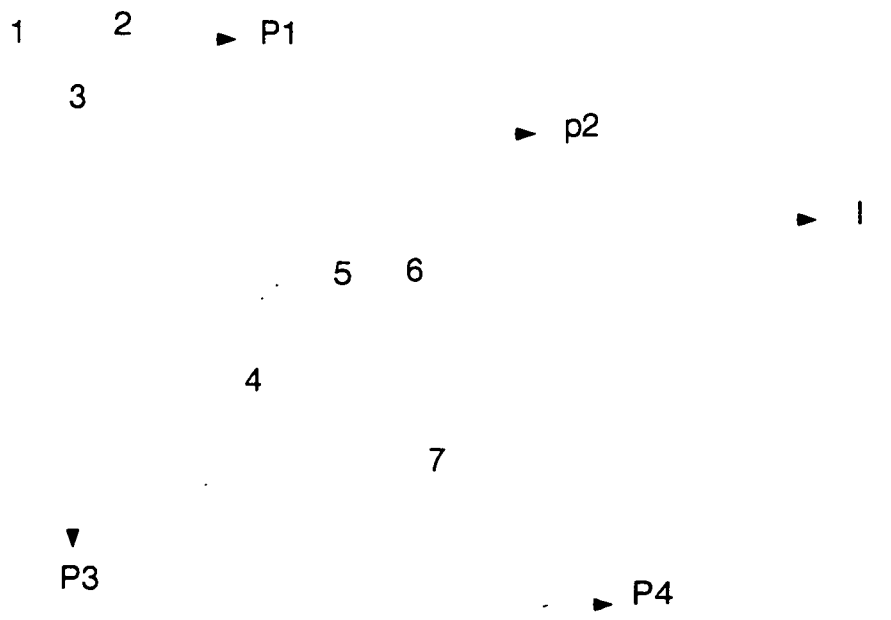
The matrix, E = [$e_{ij}$], appears as follows:

$$E = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$J^* = \{1, 4\} \subseteq J$ defines a cover of I since

$$\bigcup_{j \in J^*} P_j = I.$$

The above example can be represented by the following figure:




## A.2 A Greedy Heuristic Algorithm for the Set Covering Problem

The efficiency of an algorithm is a function of the number of units used to encode the input. We are interested in identifying algorithms guaranteed to run in time proportional to some polynomial in the number of units of input.

### A.2.1 Polynomial-Time Algorithm

An algorithm that runs in time proportional to some polynomial in the number of units of input is said to be a polynomial-time algorithm.

### A.2.2 Nondeterministic Algorithm

When an algorithm is faced with a choice of several options, and it has the power to "choose" any one then is said to be a nondeterministic algorithm.

### A.2.3 NP-Complete Problem

Any problem that can not be solved by a nondeterministic polynomial-time algorithms is said to be NP-complete.

Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. Heuristics are not infallible. They can sometimes lead to a suboptimal solutions or fail to find any solution at all.

The set-covering problem is known to be NP-complete. In view of this fact, the relative importance of heuristics for solving the set-covering problem increases. Several polynomial greedy heuristics for its solutions are known e.g. Boneh & Samuel and Chvatal. This thesis will use the following greedy heuristic algorithm due to V. Chvatal.

Step 0 : Set $J^* = \emptyset$.

Step 1 : If $P_j = \emptyset$ for all j then we stop and $J^*$ is a cover. Otherwise find an index k maximizing the ratio $|P_j|/c_j$ and proceed to step 2. Where $|P_j|$ denotes the cardinality of $P_j$, i.e., the number of elements that makes up $P_j$ .

Step 2 : Add k to $J^*$, replace each $P_j$ by $P_j/P_k$ and return to step 1. Where $P_j/P_k$ is the set $P_j$ with the elements of the set $P_k$ being removed.

The above algorithm does not deal with the case where there is more than one index k such that the ratio $|P_j|/c_j$ is maximized. If such case arises, the following rules will used :

1. If $\bigcup P_k = I$, then set $J^* = \bigcup k$ and stop.
2. If $\bigcup P_k \neq I$, then choose k = max{k's} to maximize the ratio $|P_j|/c_j$ and continue with step 2.

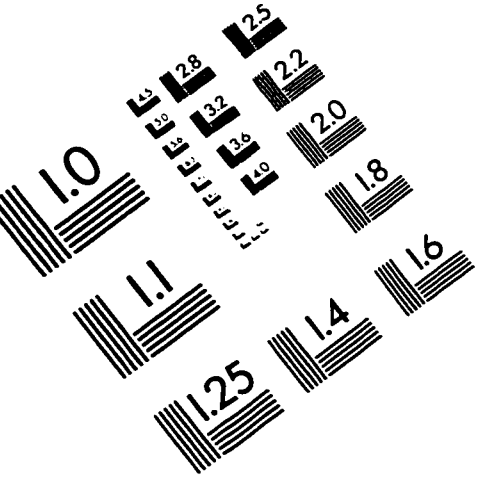
Also the above algorithm does not deal with the case where there is no cover, i.e, a feasible solution for the problem. If such case arises, i.e., if $P_j \neq 0$ for all j and there is no index k maximizing the ration $|p_j|/c_j$, then the algorithm is stoped in step 1.

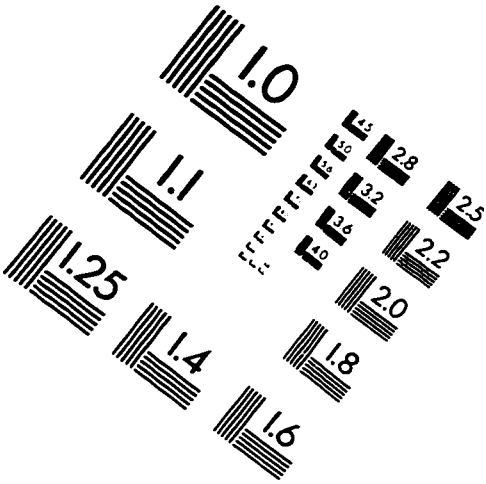
The ratio $|P_j|/c_j$ counts the number of points covered by $P_j$ per unit cost. Throughout this thesis a uniform cost $c_j = 1$ will be assumed.

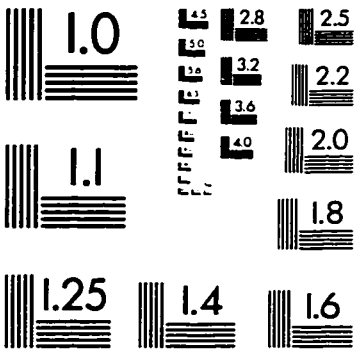
# VITA AUCTORS

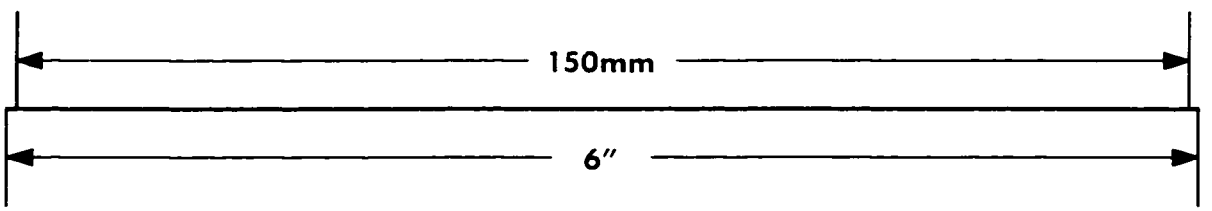
**Halima El-Khatib** was born in 1960 in **in Lebanon.** She graduated from **High School** in 1979. From there she went on to the **Post secondary school** where she obtained a B. Sc. in Computer Science in 1990 from University of Georgia. She is currently a candidate for the Master's degree in Computer Science at the University of Windsor and hopes to graduate in the Winter of 1997.

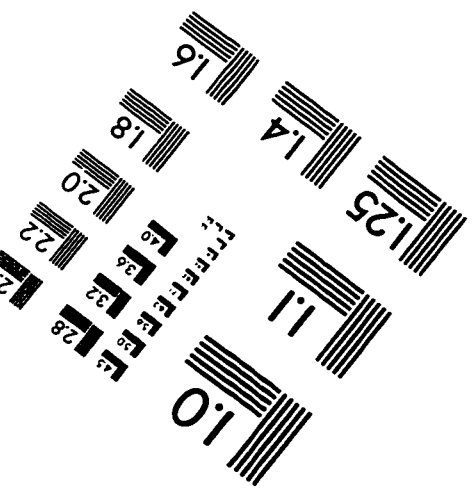
# IMAGE EVALUATION
# TEST TARGET (QA–3)




APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989