Global Optimization of Nonconvex Factorable Programs with Applications to Engineering Design Problems

by

Hongjie Wang

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Industrial and Systems Engineering

APPROVED:

Dr. Hanif D. Sherali (Chairman)

Dr. Joel A. Nachlas

Dr. Subhash C. Sarin

May 20, 1998

Blacksburg, Virginia

Global Optimization of Nonconvex Factorable Programs with Applications to Engineering Design Problems

by

Hongjie Wang

Dr. Hanif D. Sherali, Chairman

Industrial and Systems Engineering

(ABSTRACT)

The primary objective of this thesis is to develop and implement a global optimization algorithm to solve a class of nonconvex programming problems, and to test it using a collection of engineering design problem applications. The class of problems we consider involves the optimization of a general nonconvex factorable objective function over a feasible region that is restricted by a set of constraints, each of which is defined in terms of nonconvex factorable functions. Such problems find widespread applications in production planning, location and allocation, chemical process design and control, VLSI chip design, and numerous engineering design problems. This thesis offers a first comprehensive methodological development and implementation for determining a global optimal solution to such factorable programming problems.

To solve this class of problems, we propose a branch-and-bound approach based on linear programming (LP) relaxations generated through various approximation schemes that utilize, for example, the Mean-Value Theorem and Chebyshev interpolation polynomials, coordinated with a *Reformulation-Linearization Technique* (RLT). The initial stage of the lower bounding step generates a tight, nonconvex polynomial programming relaxation for the

given problem. Subsequently, an LP relaxation is constructed for the resulting polynomial program via a suitable RLT procedure. The underlying motivation for these two steps is to generate a tight outer approximation of the convex envelope of the objective function over the convex hull of the feasible region. The bounding step is then integrated into a general branch-and-bound framework. The construction of the bounding polynomials and the node partitioning schemes are specially designed so that the gaps resulting from these two levels of approximations approach zero in the limit, thereby ensuring convergence to a global optimum. Various implementation issues regarding the formulation of such tight bounding problems using both polynomial approximations and RLT constructs are discussed. Different practical strategies and guidelines relating to the design of the algorithm are presented within a general theoretical framework so that users can customize a suitable approach that takes advantage of any inherent special structures that their problems might possess.

The algorithm is implemented in C++, an object-oriented programming language. The class modules developed for the software perform various functions that are useful not only for the proposed algorithm, but that can be readily extended and incorporated into other RLT based applications as well. Computational results are reported on a set of fifteen engineering process control and design test problems from various sources in the literature. It is shown that, for all the test problems, a very competitive computational performance is obtained. In most cases, the LP solution obtained for the initial node itself provides a very tight lower bound. Furthermore, for nine of these fifteen problems, the application of a local search heuristic based on

initializing the nonlinear programming solver MINOS at the node zero LP solution produced the actual global optimum. Moreover, in finding a global optimum, our algorithm discovered better solutions than the ones previously reported in the literature for two of these test instances.

Acknowledgment

This thesis is dedicated to my advisor Dr. Hanif D. Sherali. I would like to thank my parents and my wife Qing, for their love and support. I would also like to thank Dr. Joel A. Nachlas and Dr. Subhash C. Sarin for serving on the thesis committee.

Finally, I like to acknowledge the National Science Foundation for their support under Grant No. DMI-9521398.

Contents

1 Introduction **1**

1.1 Scope and Purpose 1

1.2 Problem Description 2

2 Literature Review **4**

2.1 Global Optimization 4

2.2 Factorable Programming 5

2.3 Reformulation-Linearization Technique 7

2.4 Range-Reduction and Constraint Filtering 8

3 Problem Structure and Assumptions **10**

4 Constructing Bounding Polynomials **15**

4.1 Mean-Value Theorem Based Approach 15

4.2 Chebyshev Interpolation Approach 19

5 A Branch-and-Bound Algorithm **25**

5.1 A Two-Step Procedure for Generating Relaxations 25

5.1.1 Generating a Lower Bounding Nonconvex Polynomial Programming Problem 26
5.1.2 Reformulation-Linearization Technique 27
5.2 A Branch-and-Bound Algorithm 30
5.2.1 Branching Rule 31
5.2.2 Algorithmic Statement 32
5.3 An Illustrative Example 38

6 Implementation 45
6.1 Design Issues 45
6.1.1 Pre-processing the Problem. 46
6.1.2 Algorithms for Bounding Polynomials 47
6.1.3 Constructing RLT Constraints Under Different Options 48
6.1.4 Range-Reduction 51
6.1.5 Constraint Filtering Techniques 51
6.1.6 Branch-and-Bound Framework 52
6.2 C++ Classes 52
6.2.1 Constraint Class 53
6.2.2 A Combination Class 60
6.2.3 Problem Class and Sparse Matrix Class 67
6.2.4 Classes for Branch-and-Bound 67

7 Computational Experience 69

8 Conclusions and Recommendations for Future Research 83
8.1 Issues for Future Research 85

Bibliography **88**

Appendix A Tutorial on the Branch-and-Bound Code **96**

Vita **112**

List of Figures

4.1 Bounding Polynomials via the Mean-Value Theorem 18
4.2 Bounding Polynomials via Chebyshev Interpolation Polynomials 22
5.1 Objective Function for the Illustrative Example. 39

List of Tables

7.1 Computational Results with $\epsilon = 10^{-6}$. 70

7.2 Computational Results with $\epsilon = 0.01$. 71

7.3 Computational Results with $\epsilon = 0.05$. 72

Chapter 1

Introduction

1.1 Scope and Purpose

The primary thrust of this thesis research is to design and implement a global optimization algorithm for a class of nonconvex factorable programming problems. This is a special class of nonconvex programming problems for which the objective function and constraints are defined in terms of general factorable functions. Such problems find numerous applications in diverse areas such as engineering design and control, computational biology, chemical process, network design, and stochastic programming. The algorithm presented in this thesis is a first general systematic approach to solve such a comprehensive class of problems.

We propose a branch-and-bound solution approach for this class of problems, based on linear programming relaxations generated through various approximation schemes that utilize, for example, the Mean-Value Theorem and Chebyshev interpolation polynomials, coordinated with a *Reformulation-*

Linearization Technique (RLT). Suitable branching rules that guarantee convergence to a global optimal solution are established. A software package in C++ has been written to implement the proposed algorithm, and test problems from various engineering applications are solved to evaluate the performance of this algorithm. The computational experience reported is very promising. The proposed LP relaxations are shown to produce very tight lower bounds, and the overall computational effort is quite modest. In two cases, we even found better solutions than the ones reported in the existing literature.

1.2 Problem Description

The class of nonconvex factorable programming problems that we consider in this thesis can be stated as follows.

$$\begin{aligned} \mathbf{FP}(\Omega): \quad & \text{Minimize} \quad f_0(x) \\ & \text{subject to} \quad f_i(x) \leq \beta_i \text{ for } i = 1 \dots, m, \\ & \qquad x \in \Omega = \{x : 0 \leq l_j \leq x_j \leq u_j < \infty, \text{ for } j \in N \equiv \{1, \dots, n\}\}, \end{aligned} \tag{1.1}$$

where

$$f_i(x) = \sum_{t \in T_i} \alpha_{it} \prod_{j \in J_{it}} f_{itj}(x_j) \equiv \sum_{t \in T_i} \alpha_{it} f_{it}(x) \text{ for } i = 0, 1, \dots, m. \tag{1.2}$$

Here, $f_i(x)$ is a nonconvex factorable function that is stated as a sum of terms $\alpha_{it} f_{it}(x)$, indexed by $t \in T_i$. For each $t \in T_i$, $f_{it}(x)$ is a product of univariate functions $f_{itj}(x_j)$ of x_j, indexed by $j \in J_{it} \subseteq N$. The name

factorable program denotes this latter property whereby each nonlinear term in the problem can be factored into a product of univariate functions.

The remainder of this thesis is organized as follows. Chapter 2 gives a brief literature review on several subjects that are relevant to our research. Chapter 3 delineates the structure and assumptions regarding the problem that are necessary for our procedure to be applicable. Chapter 4 presents two different methods for constructing bounding polynomial functions for use in the first step of the relaxation process. The branch-and-bound algorithm is developed in Chapter 5. Various aspects of the proposed procedure are illustrated via an insightful numerical example in this chapter. In Chapter 6, some practical issues, considerations, and implementation strategies are presented. Several C++ program modules that play a key role in the implementation are also discussed. The computational experience obtained on a set of fifteen practical engineering design and control problems is reported in Chapter 7. Finally, we conclude the thesis with some suggestions for future research in Chapter 8 and offer a tutorial on the software in the appendix.

Chapter 2

Literature Review

2.1 Global Optimization

Global optimization has received a great deal of attention recently, due to its widespread applications in the areas of engineering design, process control, finance, economics and risk management, among many others. Several examples of practical optimization problems are discussed in Pintér (1996). Motivated by such applications, a great number of algorithms have been developed to solve various classes of problems (see Horst and Tuy (1996)). Practical nonlinear problems are usually difficult to solve due to the presence of multiple optima. From the complexity point of view, global optimization problems belong to the class of NP-hard problems (see Vavasis (1995)). This means that as the input size of the problem increases, the computational time to solve the problem to optimality can be expected to grow exponentially. Designing effective and efficient algorithms for global optimization is one of the fastest growing and challenging research areas in mathematical

programming and engineering science.

2.2 Factorable Programming

The class of factorable nonconvex programming problems was first introduced by McCormick (1976). It is a very comprehensive class of problems that subsumes many traditional types of problems that have received special attention in the literature. For example, geometric programming problems can be converted into separable programming problems, a subset of factorable programs, via suitable exponential transformations as shown in Dembo (1978). The general nonconvex multiplicative programming problems studied in Konno and Kuno (1995) are amenable to solution with separable programming techniques. Separable concave programs that are investigated in Horst and Thoai (1996) and Shectman and Sahinidis (1996) are special cases of nonconvex factorable programming. The nonconvex separable resource allocation problem examined in Haddad (1996) also fits the description of our problem statement in Chapter 1.

Most deterministic approaches for nonconvex optimization focus on constructing convex relaxations, and then embedding these lower bounding problems within a branch-and-bound procedure. Consequently, considerable research effort has been devoted to developing exact or tight approximations to convex envelopes for nonconvex functions. To solve factorable programming problems, McCormick (1976) proposed a systematic and recursive process for constructing convex underestimators for the nonconvex terms in the objective function and the constraints, assuming that convex and concave en-

velopes over Ω are known for each defining univariate function in the problem. The resulting underestimating convex program was embedded in a successive partitioning or branch-and-bound process, and various conditions were developed under which global convergence is guaranteed. Al-Khayyal and Falk (1983) have derived an explicit closed-form formula for the convex envelope of a bivariate, bilinear function over a rectangle. Sherali and Alameddine (1990) apply RLT to characterize explicit convex envelopes of bivariate bilinear functions over more general D-polytopes. Sherali (1997) has also developed the convex envelopes of multilinear functions over the unit hypercube, as well as over other special discrete sets. Adjiman and Floudas (1996) establish procedures to derive rigorous convex underestimators for general twice-differentiable functions by finding valid lower bounds on the eigenvalues of the functions using interval Hessian matrices.

The class of polynomial programming problems is a special case of factorable programs that has received considerable attention. Floudas and Visweswaran (1990, 1995) propose a transformation process for converting any given polynomial program into an equivalent quadratic program, and Al-Khayyal *et al.* (1994) show how such a problem can alternatively be converted into an equivalent bilinearly constrained bilinear program. In either case, the resulting problem is solved via a branch-and-bound procedure that employs convex envelope based linear programming relaxations in order to compute lower bounds at each node of the enumeration tree. Sherali and Tuncbilek (1992) propose a *Reformulation-Linearization Technique (RLT)* for directly computing lower bounds for polynomial programming problems, and embed this strategy into a branch-and-bound algorithm. Special branch-

ing rules are designed to ensure the convergence to a global optimum. This approach has been recently extended by Sherali (1998) to handle polynomial programs having more general rational exponents on the variables. In this case, an additional step is introduced to construct a lower bounding polynomial program having integer exponents to which the RLT approach is applied in order to derive lower bounds. Suitable partitioning rules are designed to close the gap in these two levels of approximation and induce convergence to a global optimum. The research in this thesis is a direct and significant extension of these previous research efforts.

2.3 Reformulation-Linearization Technique

The most important component in a deterministic approach to global optimization such as the branch-and-bound method, is to obtain tight lower bounds via suitable LP or convex relaxations (see Sherali and Adams (1996)). Sherali and Adams (1990,1994) develop a *Reformulation-Linearization Technique* that is designed to generate a hierarchy of linear programming relaxations for mixed 0-1 integer programming problems. These relaxations span the spectrum from the usual continuous relaxation to the actual convex hull representation of the original feasible solution space. Typically, first or second levels of RLT are used to generate tight LP relaxations that are embedded within a general branch-and-bound procedure. This methodology has been applied to solve many different types of discrete optimization problems (see the survey in Sherali and Adams (1996)). Moreover, Sherali and Tuncbilek (1992) have extended this unique approach to the continuous

problem domain, in particular, to nonconvex global optimization of polynomial programming problems. Sherali and Alameddine (1991) further utilize RLT to solve the class of bilinear programming problems. The LP relaxation constructed via RLT gives a close approximation to the convex hull of feasible points of the original bilinear program. In some cases, it is proven that the approximations are exact. In Sherali and Tuncbilek (1995), RLT is used to devise a powerful algorithm for solving nonconvex quadratic programming problems. In addition to the LP relaxation, other kinds of RLT based cuts are generated so that the resulting convex relaxation yields very tight lower bounds. Various classes of RLT based valid inequalities are also presented in Sherali and Tuncbilek (1997) for univariate and multivariate polynomial programming problems.

2.4 Range-Reduction and Constraint Filtering

One technique that is often employed in modern implementations of branch-and-bound methods is range-reduction. This typically utilizes the best incumbent solution and other information regarding the unique structure of the problem to further restrict the search region for locating global optimal solutions. Ryoo and Sahindis (1996) and Sherali and Tuncbilek (1995) develop a series of techniques to generate valid inequalities that are used to reduce the size of the search space for various global optimization problems. (Ryoo and Sahindis have coined the term *branch-and-reduce* to highlight this approach to global optimization.) Another approach to reduce the number

of iterations is investigated in Epperly and Pistikopoulos (1997) and Sherali and Alameddine (1991), where the algorithm chooses only a subset of the original variables to perform branching on, while still retaining theoretical convergence to a global optimum.

In RLT based algorithms, constraint filtering is a common strategy that is used to control the size of the LP relaxations. Several practical constraint filtering schemes are discussed in Sherali and Tuncbilek (1995). More recently, Sherali, Smith and Adams (1997) have explored the possibility and effectiveness of the generation of reduced first-level representations for 0-1 mixed-integer programs that retain the tightness of the full first-level RLT relaxation, while significantly reducing the total number of RLT constraints.

Chapter 3

Problem Structure and Assumptions

The essence of the proposed algorithm lies in two particular levels of approximation and relaxations that are employed in its design, as mentioned in the introduction in Chapter 1. The special features of the approximating polynomials derived in the first of these two steps ensure in the limit that these polynomials converge to the original nonconvex factorable functions. The unique properties of RLT are used to drive the gaps between the LP relaxations and the intermediate approximating polynomial programs to zero. To facilitate this two-step process, the univariate functions that define the nonlinear factorable terms in the problem should satisfy some basic properties or assumptions. We point out here that these assumptions are delineated to mainly ease the theoretical exposition. In practice, as explained later in this chapter, several types of problem manipulation techniques can be used to effectively cast a given, more general, problem into the required form that

satisfies these assumptions.

Assumption 1. For each (i,t,j), $i \in \{0,1,\ldots,m\}$, $t \in T_i$ and $j \in J_{it}$, $f_{itj}(x_j) \in C^2$ (twice continuously differentiable) and is nonnegative on $\Omega_j = [l_j, u_j]$, and there exists a polynomial function $g_{itj}^{\Omega_j}(x_j)$, which might depend on Ω_j, such that $g_{itj}^{\Omega_j}(x_j) \geq 0$ and for all $x_j \in \Omega_j$,

$$\begin{cases} g_{itj}^{\Omega_j}(x_j) \leq f_{itj}(x_j) \text{ if } \alpha_{it} > 0, \\ g_{itj}^{\Omega_j}(x_j) \geq f_{itj}(x_j) \text{ if } \alpha_{it} < 0. \end{cases} \tag{3.1}$$

Moreover, in each case, the corresponding function $g_{itj}^{\Omega_j}(x_j)$ satisfies the following property.

Property 1. (This property is stated for the case of lower bounding approximations in (3.1). A symmetric property must hold true for upper bounding approximations.)

For any nested sequence $\{\Omega_j^k\} \to \bar{\Omega}_j$, where k is an index for the elements of the sequence, and where $\Omega_j^k = [l_j^k, u_j^k]$ and $\bar{\Omega}_j = [\bar{l}_j, \bar{u}_j]$, consider a corresponding sequence of lower bounding functions $g_{itj}^{\Omega_j^k}(x_j), x_j \in \Omega_j^k$, satisfying Assumption 1. Then, in case $\bar{l}_j < \bar{u}_j$, we must have as $k \to \infty$,

$$g_{itj}^{\Omega_j^k}(x_j) \to g_{itj}^{\bar{\Omega}_j}(x_j), \forall x_j \in \bar{\Omega}_j, \tag{3.2}$$

where $g_{itj}^{\bar{\Omega}_j}(x_j)$ is a lower bounding function for $f_{itj}(x_j)$ on $\bar{\Omega}_j$. On the other hand, if $\bar{l}_j = \bar{u}_j$, then, for any $\{x_j^k\} \to \bar{l}_j \equiv \bar{u}_j$, where $x_j^k \in \Omega_j^k$, $\forall k$, we must have as $k \to \infty$,

$$\{g_{itj}^{\Omega_j^k}(x_j^k)\} \to f_{itj}(\bar{l}_j). \tag{3.3}$$

Furthermore, to aid in our analysis, let us identify the following sets. Given $f_{itj}(x_j) : \Omega_j \to \mathbb{R}$, for $i \in \{1, \ldots, m\}$, $t \in T_i$, and $j \in J_{it}$, let

$$E_{itj} \subseteq \{x_j \in \Omega_j : g_{itj}^{\Omega_j}(x_j) = f_{itj}(x_j)\}, \tag{3.4}$$

where the set E_{itj} is composed as follows, in the stated order of preference:

$$E_{itj} = \begin{cases} \{l_j, u_j\} & \text{if this holds true} \\ \{\bar{x}_{itj}\} & \text{if } \exists\ \bar{x}_{itj} \text{ satisfying condition A below} \\ \emptyset & \text{if neither of the above possibilities hold true.} \end{cases} \tag{3.5}$$

Condition A requires that $\bar{x}_{itj} \in [l_j, u_j]$ is a function of l_j and u_j such that for any nested sequence of intervals that is obtained by repeatedly partitioning the bounding interval Ω_j at $\bar{x}_{itj}$ and picking one of the subintervals as the revised bounding interval, we have that the interval length approaches zero.

It is obvious that this assumption is only relevant for the non-polynomial functions $f_{itj}(x_j)$ in the problem. Notice that we have assumed above that $f_{itj}(x_j) \geq 0$ on $\Omega_j = [l_j, u_j]$, for each $i \in \{0, 1, \ldots, m\}, t \in T_i, j \in J_{it}$. In practice, this assumption is not very restrictive. If $f_{itj}(x_j)$ is not nonnegative, we can replace the term $f_{itj}(x_j)$ in the problem with $f_{itj}(x_j) + v_{itj} - v_{itj}$, where $v_{itj} \geq |\min_{x_j \in \Omega_j} f_{itj}(x_j)|$. This value v_{itj} is easy to obtain since $f_{itj}(x_j)$ is a univariate function. With this substitution, we can define a new function $\bar{f}_{itj}(x_j) = f_{itj}(x_j) + v_{itj}$ which is nonnegative. Moreover, since we are dealing with bounding univariate functions, the user can even employ graphical plots to prescribe appropriate bounding function modules in concert with the

methods of Chapter 4 below, in order to derive suitable tight approximations that satisfy the required assumptions. Actually, as will be evident from our analysis, the assumptions on the nonnegativity of $f_{itj}(x_j)$ and $g_{itj}^{\Omega_j}(x_j)$ on Ω_j for some such pair of functions are not needed when the corresponding term $t \in T_i$ for $i \in \{0, 1, \ldots, m\}$ is, for example, such that $|J_{it}| = 1$, or when $f_{it}(x)$ is comprised of the product of a single non-polynomial function $f_{itj}(x_j)$ and polynomial terms involving a different set of nonnegative variables. In addition, given that $f_{itj}(x_j) \geq 0 \ \forall j \in J_{it}$, the nonnegativity requirement on $g_{itj}^{\Omega_j}(x_j)$ can be relaxed for one index $j \in J_{it}$, in each term $t \in T_i$, for $i \in \{0, 1, \ldots, m\}$. In our numerical example of Chapter 5, we illustrate this point. One of the techniques used in that example involves the substitution of each non-polynomial function with a new variable. This technique is used in the implementation. As we shall see in Chapter 6, by employing such problem manipulations, apparent violations of the assumptions listed in this chapter are effectively resolved.

The essence of the foregoing assumption is that it presumes the existence of bounding polynomials $g_{itj}(x_j)$ having the stated desirable properties. Several methods can be readily designed to effectively generate such polynomials, and we present two such methods in Chapter 4 below. However, any other alternative method can just as well be used so long as the resulting polynomial functions satisfy Assumption 1, and the ensuing convergence argument will then hold true. With this unique feature, the algorithm can be considered as a general framework in which users can design and implement their own versions of key components according to the guidelines provided. This design reflects our desire to develop an algorithm that is general enough to be

applicable on a wide class of problems, yet at the same time, it is robust and flexible enough to permit specializations for particular types of problems.

Chapter 4

Constructing Bounding Polynomials

An important step in our algorithm is to construct appropriate bounding polynomial functions $g_{itj}^{\Omega_j}(x_j)$ so that Property 1 stated in Chapter 3, in particular, is satisfied. In this chapter, we provide two general methods for constructing such polynomial functions based, respectively, on the Mean-Value Theorem and on Chebyshev interpolation. We point out here that although the assumption $g_{itj}^{\Omega_j}(x_j) \geq 0$ is not guaranteed for these two methods, this can be readily rectified by applying the technique mentioned at the end of Chapter 3 to manipulate the problem to satisfy this assumption as necessary.

4.1 Mean-Value Theorem Based Approach

For any $(i, t, j), i \in \{0, 1, \ldots, m\}, t \in T_i, j \in J_{it}$, consider the function $f_{itj}(x_j)$ defined on $\Omega_j = [l_j, u_j]$. If this function is sufficiently smooth, according to

the general Mean-Value Theorem,we have that

$$\begin{aligned} f_{itj}(x_j) &= f_{itj}(x_0) + f'_{itj}(x_0)(x_j - x_0) + \ldots + \frac{1}{(2r-1)!} f_{itj}^{(2r-1)}(x_0)(x_j - x_0)^{(2r-1)} \\ &\quad + \frac{1}{(2r)!} f_{itj}^{(2r)}(\xi)(x_j - x_0)^{2r} \text{ for any } x_j \in \Omega_j, \end{aligned} \tag{4.1}$$

where $\xi = \lambda x_j + (1-\lambda)x_0$ for some $\lambda \in (0,1)$, and where we can select $x_0 = \frac{1}{2}(l_j + u_j)$ and use r equal to any desired positive integer.

Now, suppose that $m \leq f_{itj}^{(2r)}(x_j) \leq M$ for all $x_j \in \Omega_j$, where m and M are known. (This is always possible for a continuous function on a compact interval.) Then,

$$\begin{aligned} & f_{itj}(x_0) + f'_{itj}(x_0)(x_j - x_0) + \ldots \\ & + \frac{1}{(2r-1)!} f_{itj}^{(2r-1)}(x_0)(x_j - x_0)^{(2r-1)} + \frac{1}{(2r)!} m(x_j - x_0)^{2r} \end{aligned} \tag{4.2}$$

is a valid lower bounding polynomial for $f_{itj}(x_j)$ on Ω_j and

$$\begin{aligned} & f_{itj}(x_0) + f'_{itj}(x_0)(x_j - x_0) + \ldots \\ & + \frac{1}{(2r-1)!} f_{itj}^{(2r-1)}(x_0)(x_j - x_0)^{(2r-1)} + \frac{1}{(2r)!} M(x_j - x_0)^{2r} \end{aligned} \tag{4.3}$$

is a valid upper bounding polynomial for $f_{itj}(x_j)$ on Ω_j. According to the sign of α_{it} as in (3.1), we can choose the appropriate expression (4.2) or (4.3) as our bounding polynomial function $g_{itj}^{\Omega_j}(x_j)$.

The following lemma shows that the bounding polynomials developed in (4.2) and (4.3) satisfy the requirements stated in Chapter 3 and are admissible in our algorithm.

Lemma 1. *For any (i,t,j), consider a nested sequence of intervals $\{\Omega_j^k\} \to \bar{\Omega}_j$, where $\Omega_j^k = [l_j^k, u_j^k]$, $\forall k = 1,2,\ldots$, and $\bar{\Omega}_j = [\bar{l}_j, \bar{u}_j]$. For each k, select m_k such that $f_{itj}^{(2r)}(x_j) \geq m_k, \forall x_j \in \Omega_j^k$, and such that $\{m_k\}$ is monotone increasing (such a choice exists since $\{\Omega_j^k\}$ is a nested sequence). Then, by defining $g_{itj}^{\Omega_j^k}(x_j), x_j \in \Omega_j^k$, as in (4.2) with $x_0 \equiv (l_j^k + u_j^k)/2$ and $m \equiv m_k$, we have Property 1 in Chapter 3 holding true for the case of lower bounding functions. (A symmetric result holds true for the case of upper bounding functions.)*

Proof. Define $\bar{x}_0 \equiv (\bar{l} + \bar{u})/2$ and note that $\{x_0^k\} \to \bar{x}_0$. Since $\{m_k\}$ is a monotone increasing sequence that is bounded from above by $f_{itj}^{(2r)}(\bar{x}_0)$, we must have $\{m_k\} \to \bar{m}$ for some $\bar{m}$. Define $g_{itj}^{\bar{\Omega}_j}(x_j)$ as in (4.2) using $x_0 = \bar{x}_0$ and $m = \bar{m}$. By continuity, we have $g_{itj}^{\Omega_j^k}(x_j) \to g_{itj}^{\bar{\Omega}_j}(x_j)$ for every $x_j \in \bar{\Omega}_j$. Moreover, if $\bar{l}_j = \bar{u}_j$, then $g_{itj}^{\bar{\Omega}_j}(x_j)$ coincides with $f_{itj}(\bar{x}_0) = f_{itj}(\bar{l}_j)$. This completes the proof. □

In principle, any $x_0 \in \Omega_j$ can be used in (4.2) and (4.3) to construct valid bounding polynomials. To obtain a tight polynomial approximation, it is advisable to take x_0 as the mid-point of the interval Ω_j. In this case, it is trivial to show that the maximum error is estimated as follows:

$$\max_{x_j \in \Omega} |f_{itj}(x_j) - g_{itj}(x_j)| \leq \frac{2K}{(2r)!} \frac{(u_j - l_j)^{2r}}{2^{2r}}, \tag{4.4}$$

where $K = \max_{[l_j,u_j]} |f_{itj}^{(2r)}(x_j)|$. In practice, however, one may choose several different values of x_0 to construct alternative bounding polynomials to provide multiple supports for $f_{itj}(x_j)$.

Finally, we give a numerical example on constructing bounding polyno-

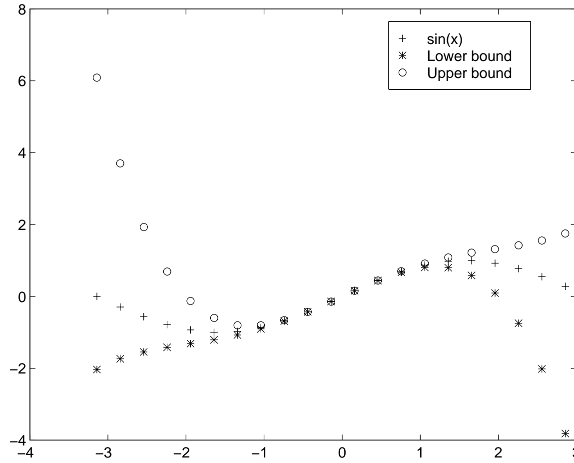

Figure 4.1: Bounding Polynomials via the Mean-Value Theorem

mial functions using the method described in this section. Consider the univariate function $f(x) = \sin(x)$, $x \in [-\pi, \pi]$. To construct 4th order bounding polynomials, we first expand $f(x)$ about the point $x_0 = 0$,

$$\begin{aligned} \sin(x) = & \quad \sin(0) + \cos(0)x - \frac{1}{2}\sin(0)x^2 + \frac{1}{3!}\cos(0)x^3 - \frac{1}{4!}\sin^{(4)}(\xi)x^4 \\ = & \quad x + \frac{1}{6}x^3 - \frac{1}{24}\sin^{(4)}(\xi)x^4, \end{aligned} \tag{4.5}$$

where $\xi \in (-\pi, \pi)$.

Since $|\sin^{(4)}(\xi)|$ is bounded by 1, we obtain the following lower and upper bounding polynomials, respectively. These are depicted in Figure 4.1.

$$x - \frac{1}{6}x^3 - \frac{1}{24}x^4, \quad x - \frac{1}{6}x^3 + \frac{1}{24}x^4. \tag{4.6}$$

4.2 Chebyshev Interpolation Approach

Suppose that we have constructed some polynomial function $p(x)$ based on a set of grid points in order to approximate a given univariate function $f(x)$ over the interval Ω, and let $E(x) = p(x) - f(x)$ be the associated error function. Then $p(x) - \max_{x\in\Omega}\{E(x)\}$ will be a valid lower bounding polynomial function for $f(x)$. To obtain tight lower envelopes, we naturally want the absolute values taken on by the error function to be as small as possible. An polynomial $p(x)$ is called the *best approximation* of $f(x)$ if the maximum absolute error is minimized. Unfortunately, it is not generally computationally tractable to derive best approximation polynomials. However, good approximations can be obtained by using *Chebyshev interpolation polynomials* (see Davis (1975) and Ueberhuber (1997)).

Given $f_{itj}(x_j) \in C^{r+1}, x_j \in \Omega_j = [l_j, u_j]$, let $P^{\Omega_j}(x_j)$ denote a Chebyshev interpolation polynomial of order r for $r \geq 1$ (see Volkov (1990), for example). This function can be generated as follows.

- Define Chebyshev points $\hat{x}_i = \frac{1}{2}(l_j + u_j) + \frac{1}{2}(u_j - l_j)\cos(\frac{(2i+1)\pi}{2r+2}), i = 0, 1, \ldots, r$.
- Derive the Chebyshev interpolation polynomial $P^{\Omega_j}(x_j)$ for $f_{itj}(x_j)$ corresponding to the grid points $\hat{x}_0, \hat{x}_1 \ldots, \hat{x}_r$. This can be done for example, by Newton's method (see Stoer and Burlirsch (1993)).
- Compute $T^{\Omega_j} = \frac{1}{(r+1)!}2^{-(2r+1)}(u_j - l_j)^{r+1} \max_{x_j\in[l_j,u_j]} |f_{itj}^{(r+1)}(x_j)|$.
- Let $g_{itj}^{\Omega_j}(x_j) = P^{\Omega_j}(x_j) - T^{\Omega_j}$ if $\alpha_{it} > 0$ and $g_{itj}^{\Omega_j}(x_j) = P^{\Omega_j}(x_j) + T^{\Omega_j}$ if $\alpha_{it} < 0$.

Then the following result holds true, which therefore assures (3.1).

Lemma 2.

$$\max_{x_j \in [l_j, u_j]} |f_{itj}(x_j) - P^{\Omega_j}(x_j)| \leq T^{\Omega_j} \equiv \frac{1}{(r+1)!} 2^{-(2r+1)} (u_j - l_j)^{r+1} \max_{x_j \in [l_j, u_j]} |f_{itj}^{(r+1)}(x_j)| \quad (4.7)$$

Proof. See Davis (1975) and Volkov (1990). □

The unique feature of using Chebyshev approximation is its known tight maximum error bound. Above, we have described one convenient procedure for deriving such approximations, but other efficient and elegant implementations are also available (see, for example, Ueberhuber (1997) and Press *et al.* (1992)). We now establish that this class of bounding polynomial functions is admissible for our proposed algorithm.

Lemma 3. *For any (i, t, j), and a nested sequence of intervals $\{\Omega_j^k\} \to \bar{\Omega}_j$, where $\Omega_j^k = [l_j^k, u_j^k]$, $\forall k = 1, 2, \ldots$ and $\bar{\Omega}_j = [\bar{l}_j, \bar{u}_j]$, define $g_{itj}^{\Omega_j^k}(x_j) \equiv P^{\Omega_j^k}(x_j) - T^{\Omega_j^k}$, $x_j \in \Omega_j^k$, $\forall k$. Then, we have Property 1 in Chapter 3 holding true for the case of lower bounding functions. (A symmetric results holds true for the case of upper bounding functions.)*

Proof. For each k, let $g_{itj}^{\Omega_j^k}(x_j)$ be the lower bounding polynomial generated using Chebyshev interpolation for $f_{itj}(x_j)$ on the interval $\Omega_j^k = [l_j^k, u_j^k]$. Let $P^{\Omega_j^k}(x_j)$ be the corresponding rth order interpolation polynomial, and denote by $X^{\Omega_j^k} = [\hat{x}_0^k, \hat{x}_1^k, \ldots, \hat{x}_r^k]$ the set of associated Chebyshev points from Ω_j^k as defined above. Then,

$$g_{itj}^{\Omega_j^k}(x_j) = P^{\Omega_j^k}(x_j) - T^{\Omega_j^k}, \quad (4.8)$$

where $T^{\Omega_j^k} = \frac{1}{(r+1)!} 2^{-(1+2r)} (u_j^k - l_j^k)^{r+1} \max_{x_j \in \Omega_j^k} |f_{itj}^{(r+1)}(x_j)|$.

Case 1: $\bar{l}_j \neq \bar{u}_j$. Let $P^{\bar{\Omega}_j}(x_j)$ denote the Chebyshev interpolation polynomial of $f_{itj}(x_j)$ on $\bar{\Omega}_j$ based on the associated Chebyshev points $X^{\bar{\Omega}_j} = [\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_r]$, and let $T^{\bar{\Omega}_j}$ be the corresponding error bound. Define

$$g_{itj}^{\bar{\Omega}_j}(x_j) = P^{\bar{\Omega}_j}(x_j) - T^{\bar{\Omega}_j}, \tag{4.9}$$

where $T^{\bar{\Omega}_j} = \frac{1}{(r+1)!} 2^{-(2r+1)} (\bar{u}_j - \bar{l}_j)^{r+1} \max_{x_j \in \bar{\Omega}_j} |f_{itj}^{(r+1)}(x_j)|$. By the definition of the grid points and the property of continuity, we have $X^{\Omega_j^k} \rightarrow X^{\bar{\Omega}_j}$. Hence, due to the uniqueness of the interpolation polynomial on a given set of points, we must have $P^{\Omega_j^k}(x_j) \rightarrow P^{\bar{\Omega}_j}(x_j)$ for all $x_j \in \bar{\Omega}_j$. Furthermore, let $M_k \equiv \max_{x_j \in \Omega_j^k} |f_{itj}^{(r+1)}(x_j)| = |f_{itj}^{(r+1)}(x_j^k)|$ for some $x_j^k \in \Omega_j^k$, and let $M \equiv \max_{x_j \in \bar{\Omega}_j} |f_{itj}^{(r+1)}(x_j)|$. Since $\{x_j^k\}$ is bounded, it has a convergent subsequence identified by $\{x_j^{k_s}\} \rightarrow z$ as $s \rightarrow \infty$, where $z \in \bar{\Omega}_j$. Moreover, since $\Omega_j^k \supseteqq \bar{\Omega}_j \ \ \forall k$, we have that,

$$M \leq \lim_{s \rightarrow \infty} |f_{itj}^{(r+1)}(x_j^{k_s})| = \lim_{s \rightarrow \infty} M_{k_s} = |f_{itj}^{(r+1)}(z)| \leq M. \tag{4.10}$$

Expression (4.10) implies that M is an accumulation point of $\{M_k\}$. Since $\{M_k\}$ is monotone decreasing, it is convergent. Hence, $\{M_k\} \rightarrow M$ and $T^{\Omega_j^k} \rightarrow T^{\bar{\Omega}_j}$. Thus, when $\bar{l}_j \neq \bar{u}_j$, we have $g_{itj}^{\Omega_j^k}(x_j) \rightarrow g_{itj}^{\bar{\Omega}_j}(x_j)$ for all $x_j \in \bar{\Omega}_j$, and so Property 1 holds true.

Case 2: $\bar{l}_j = \bar{u}_j$. Let $\{x_j^k\}$ be any sequence such that $x_j^k \in \Omega_j^k$ and $\{x_j^k\} \rightarrow \bar{l}_j$ as $k \rightarrow \infty$. By Lemma 2,

$$\begin{aligned} |f_{itj}(x_j^k) - g_{itj}^{\Omega_j^k}(x_j^k)| &\leq \max_{x_j \in \Omega_j^k} |f_{itj}(x_j) - g_{itj}^{\Omega_j^k}(x_j)| \\ &\leq \frac{2}{(r+1)!} 2^{-(2r+1)} (u_j^k - l_j^k)^{r+1} \max_{x_j \in \Omega_j^k} |f_{itj}^{(r+1)}(x_j)|. \end{aligned} \tag{4.11}$$

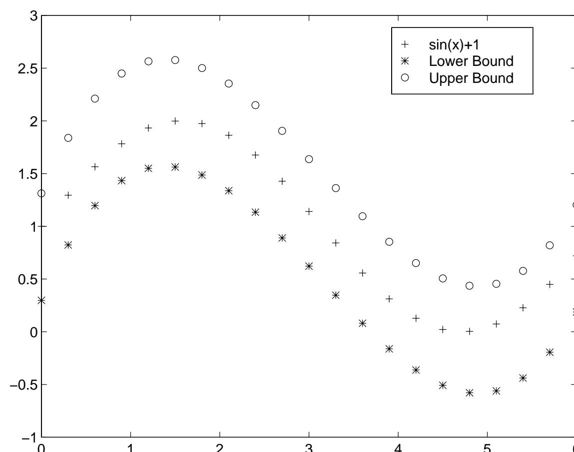

Figure 4.2: Bounding Polynomials via Chebyshev Interpolation Polynomials

Since $\max_{x_j \in \Omega_j^k} |f_{itj}^{(r+1)}(x_j)| \to \max_{x_j \in \bar{\Omega}_j} |f_{itj}^{(r+1)}(x_j)|$ and $u_j^k - l_j^k \to 0$ as $k \to \infty$,

$$\lim_{k\to\infty} g_{itj}^{\Omega_j^k}(x_j^k) = \lim_{k\to\infty} f_{itj}(x_j^k) = f_{itj}(\bar{l}_j). \tag{4.12}$$

This concludes the proof. □

To illustrate this method, we construct third-order bounding polynomials for $f(x) = \sin(x) + 1$ on $[0, 2\pi]$ using the Chebyshev interpolation polynomial based method. Following the procedure described in Section 4.2, we first compute the Chebyshev points to obtain $\{0.2391, 1.9394, 4.3438, 6.0440\}$. Using the Lagrange interpolation method, we derive the Chebyshev polynomial $p(x) = 0.0995x^3 - 0.9377x^2 + 2.026x + 0.8046$. Since $\max_{x\in[0,2\pi]} |\sin^{(4)}(x)| \leq 1$, this gives $T = 0.5073$. Thus, a valid lower bounding polynomial is $0.0995x^3 - 0.9377x^2 + 2.026x + 0.2973$ and a valid upper bounding polynomial is $0.0995x^3 - 0.9377x^2 + 2.026x + 1.3119$. They are depicted in Figure 4.2.

The two methods described in Sections 4.1 and 4.2 can be used to construct valid bounding polynomial functions for the original nonconvex functions. In general, the sets E_{itj} defined by (3.5) will be different for these two methods. If we use the Mean-Value Theorem based method, E_{itj} will be comprised of the midpoint of the interval Ω_j, while the Chebyshev interpolation approach will usually result in an empty set for E_{itj}.

For special classes of functions, other approximation methods can be preferably used. In particular, if $f_{itj}(x_j)$ is a polynomial function having a rational exponent, simple and effective methods have been developed in Sherali (1998) to generate valid bounding polynomial functions that can be readily verified to satisfy Assumption 1 and Property 1 of Chapter 3. Here, E_{itj} turns out to be given by either the first or the second case in (3.5).

The Mean-Value Theorem based approach provides a good approximation in a neighborhood of x_0, while the Chebyshev interpolation approach gives a more uniform approximation. The error estimate from the Chebyshev interpolation is smaller than the one from the Mean-Value Theorem based approach. Furthermore, the error of the Chebyshev interpolation is distributed more evenly on the entire interval. Consequently, it is highly likely that the actual maximum error of the Chebyshev interpolation will turn out to be less than the one from the approximation polynomial obtained by the Mean-Value Theorem based method (see Volkov (1990)) . In some cases, we might be able to apply more than one bounding method to simultaneously generate a manageable number of alternative relaxation constraints. For example, we could substitute a variable y_{itj} in place of the function $f_{itj}(x_j)$ and incorporate the explicit constraint $y_{itj} = f_{itj}(x_j)$ (written as two inequalities). To the

latter constraint(s), we can now apply multiple bounding methods to generate potentially tighter relaxations, while avoiding a combinatorial explosion of constraints. This is the strategy that we have adopted to pre-process the problem in the final implementation. It effectively relaxes the nonnegativity requirements on $f_{itj}(x_j)$ and its bounding polynomials (see Section 6.1.1).

Chapter 5

A Branch-and-Bound Algorithm

In this chapter, we present a branch-and-bound algorithm to solve the class of nonconvex factorable programming problems. There are three important components in this algorithm, the bounding procedure, the branching and partitioning rules, and the convergence property. They are discussed in detail in this chapter.

5.1 A Two-Step Procedure for Generating Relaxations

The proposed lower bounding relaxations are constructed over two steps. First, a tight nonconvex polynomial programming relaxation is created by replacing each non-polynomial term $f_{itj}(x_j)$ with an appropriate bounding polynomial $g_{itj}(x_j)$. Then, an LP relaxation is constructed for the resulting

polynomial program via a suitable RLT procedure. Thus, each node of the branch-and-bound tree contains these two levels of relaxations. The lower bounding and partitioning schemes are designed so that the gaps from these two levels of approximations approach zero in the limit, thereby ensuring convergence to a global optimum.

5.1.1 Generating a Lower Bounding Nonconvex Polynomial Programming Problem

Given the bounding polynomial function $g_{itj}^{\Omega_j}(x_j)$ for $f_{itj}(x_j)$ on $\Omega_j = [l_j, u_j]$, for each $i \in \{0, 1, \ldots, m\}, t \in T_i, j \in J_{it}$, let us define

$$\Phi_{it}^{R(\Omega)}(x) = \alpha_{it} \prod_{j \in J_{it}} g_{itj}^{\Omega_j}(x_j), \quad \text{and} \quad \Phi_i^{R(\Omega)}(x) = \sum_{t \in T_i} \Phi_{it}^{R(\Omega)}(x). \tag{5.1}$$

Thus, noting Assumption 1 in Chapter 3, we have

$$\begin{aligned} f_i(x) &\equiv \sum_{t \in T_i} \alpha_{it} f_{it}(x) = \sum_{t \in T_i} \alpha_{it} \prod_{j \in J_{it}} f_{itj}^{\Omega_j}(x_j) \\ &\geq \sum_{t \in T_i} \alpha_{it} \prod_{j \in J_{it}} g_{itj}^{\Omega_j}(x_j) \\ &= \sum_{t \in T_i} \Phi_{it}^{R(\Omega)}(x) \equiv \Phi_i^{R(\Omega)}(x), \forall i = 0, \ldots, m. \end{aligned} \tag{5.2}$$

Consequently, we derive the following polynomial programming relaxation of Problem $FP(\Omega)$.

$$\mathbf{FPR(\Omega)}: \quad \min\{\Phi_0^{R(\Omega)}(x) : \Phi_i^{R(\Omega)}(x) \leq \beta_i \quad \forall i = 1, \ldots, m, x \in \Omega\}. \tag{5.3}$$

Letting $v(P)$ generally denote the optimal objective function value of any problem P, we have the following result.

Lemma 4.

$$v(FPR(\Omega)) \leq v(FP(\Omega)). \tag{5.4}$$

Proof. Evident from (1.1), (5.2) and (5.3). □

Having constructed the polynomial problem $FPR(\Omega)$ in the form of (5.3), we now apply the RLT scheme of Sherali and Tuncbilek (1992) to generate a suitable linear programming relaxation. This process operates in two phases, as outlined in Sections 5.1.2 below.

5.1.2 Reformulation-Linearization Technique

RLT consists of two phases. The first phase augments the polynomial program by adding various types of RLT constraints. The second phase linearizes the reformulated problem via a variable substitution strategy.

Reformulation Phase

In the reformulation phase, we introduce valid RLT constraints and include them in $FPR(\Omega)$. In doing so, we first identify δ [1]as the highest order of any polynomial term in (5.3). Let $\bar{N} = \{N, \dots, N\}$ be δ replicates of N. We now compose all possible distinct constraints of the following type.

[1]In this thesis, δ is called the order of RLT. In principle, it is perfectly legitimate to let δ take a value that is higher than the highest order of any polynomial terms. In some cases, it is desirable to do so (see (Sherali and Tuncbilek (1995)).

$$F_\delta(J_1, J_2) \equiv \prod_{j\in J_1}(x_j - l_j)\prod_{j\in J_2}(u_j - x_j) \geq 0, \forall J_1 \cup J_2 \subseteq \bar{N}, |J_1 \cup J_2| = \delta, \quad (5.5)$$

where J_1 and J_2 might contain replicated elements, and where these replications are preserved in any union operations as well. The constraints in (5.5) are called the δth order *bound-factor product* RLT constraints, and are obtained by taking the products of the bounding factors $(x_j - l_j)$ and $(u_j - x_j)$, $j = 1, \ldots, n$, δ at a time, allowing repetitions. Additionally, we can also construct *constraint-factor product* RLT constraints. These valid inequalities can be obtained by taking suitable products of the polynomial constraint factors $\beta_i - \Phi_i^{R(\Omega)}(x) \geq 0$, with bound factors or with each other, as long as the degree of the resulting polynomial does not exceed δ. The newly generated implied RLT constraints are included to augment problem $FPR(\Omega)$, resulting in a reformulated lower bounding problem.

For the sake of simplicity, we only consider these basic RLT constraints in our current discussion. As we shall see, they are sufficient to guarantee theoretical convergence. Several other valid RLT constraints can be generated, however, to further tighten the LP relaxation. Ideas investigated in Sherali and Tuncbilek (1995), for example, can be readily extended and applied here as well. Also, more generalized constraint-factors that imply the bound-factors can be used to compose suitable RLT constraints that could be used in lieu of (5.5).

Linearization Phase

In this stage, we linearize the augmented polynomial programming problem by substituting

$$X_J = \prod_{j \in J} x_j, \quad \forall J \subseteq \bar{N}. \tag{5.6}$$

To achieve consistency, we assume that the indices in J are arranged in a nondecreasing order. Hence, for example, $x_1 x_2$ and $x_2 x_1$ will be replaced by the same new variable X_{12}. Furthermore, we identify $X_j \equiv x_j$ for all $j \in N$, and $X_\emptyset \equiv 1$.

For any $(i, t), i \in \{0, 1, \ldots, m\}, t \in T_i$, let $\Phi_{it}^{L(\Omega)}(x)$ denote the linearization of $\Phi_{it}^{R(\Omega)}(x)$ via (5.6), and let $\Phi_i^{L(\Omega)}(x) = \sum_{t \in T_i} \Phi_{it}^{L(\Omega)}(x)$. Hence, from (5.3) and (5.5), we obtain the following RLT linear programming relaxation.

$$\begin{aligned} \textbf{RLT}(\Omega) : \min \quad & \Phi_0^{L(\Omega)}(x) \\ \text{subject to} \quad & \Phi_i^{L(\Omega)}(x) \leq \beta_i \ \forall i = 1, \ldots, m, (21)_L, (VI)_L, x \in \Omega, \end{aligned} \tag{5.7}$$

where $(21)_L$ denotes the linearization of (5.5) under (5.6), and where $(VI)_L$ likewise denotes the linearization under (5.6) of any additional valid inequalities that might have been generated as previously mentioned in the Reformulation Phase.

Lemma 5 stated below records the fact that $v(RLT(\Omega))$ is indeed a lower bound on $v(FPR(\Omega))$ and hence on $v(FP(\Omega))$, and Lemma 6 reiterates an important property of RLT from Sherali and Tuncbilek (1992) based on the

presence of $(21)_L$ in (5.7), that will make it possible for us to design appropriate partitioning strategies for ensuring convergence to a global optimal solution.

Lemma 5. *$v[RLT(\Omega)] \leq v[FPR(\Omega)]$, and so, $RLT(\Omega)$ provides a lower bound on the original problem $FP(\Omega)$.*

Proof. Evident from Lemma 4 and by construction. □

Lemma 6. *Let $(\hat{x}, \hat{X})$ be any feasible solution to $RLT(\Omega)$. Suppose that $\hat{x}_p = l_p$ for some $p \in N$. Then $\hat{X}_{J \cup p} = l_p \hat{X}_J \quad \forall J \subseteq \bar{N}, 1 \leq |J| \leq \delta - 1$. Similarly, $\hat{x}_p = u_p$ implies that $\hat{X}_{J \cup p} = u_p \hat{X}_J \quad \forall J \subseteq \bar{N}, 1 \leq |J| \leq \delta - 1$.*

Proof. See Sherali and Tuncbilek (1992). □

5.2 A Branch-and-Bound Algorithm

The proposed algorithm is a branch-and-bound approach that is based on partitioning the set Ω into sub-hyperrectangles, each associated with a node of the branch-and-bound tree. Hence, at each stage s of the algorithm, suppose that we have a collection of active nodes indexed by $q \in Q_s$, say, each associated with a hyperrectangle $\Omega^q \subseteq \Omega$, $\forall q \in Q_s$. For each such node, we will have computed a lower bound LB_q via the solution of the linear program $RLT(\Omega^q)$, so that the overall lower bound on $FP(\Omega)$ at stage s is given by $LB(s) = \min\{LB_q : q \in Q_s\}$. Whenever the lower bounding solution for any node subproblem turns out to be feasible to $FP(\Omega)$, we update the upper bound or incumbent solution value v^*, if necessary. Additionally, a local search procedure that is initialized at the solution to the lower bounding

problem can be used as a heuristic to search for good quality feasible solutions. The branch-and-bound tree retains the set of active nodes that satisfy $LB_q < v^*$, $\forall q \in Q_s$, for each stage s. We now select an active node $q(s)$ that yields the least lower bound $LB(s) \equiv LB_{q(s)}$ among $q \in Q_s$, and partition its associated hyperrectangle into two sub-hyperrectangles. This branching rule is described below, and is critical to the theoretical convergence as well as the practical effectiveness of the algorithm.

5.2.1 Branching Rule

Consider any node subproblem identified by the hyperrectangle $\Omega' \subseteq \Omega$, and let $(\hat{x}, \hat{X})$ represent the solution obtained to its associated linear programming relaxation $RLT(\Omega')$. Determine a term (r, τ) such that

$$\alpha_{r\tau} f_{r\tau}(\hat{x}) - \Phi_{r\tau}^{L(\Omega')}(\hat{x}, \hat{X}) = \max_{i=0,1,\ldots m, t \in T_i} \{\alpha_{it} f_{it}(\hat{x}) - \Phi_{it}^{L(\Omega')}(\hat{x}, \hat{X})\}. \quad (5.8)$$

The selection of the branching variable x_p and the partitioning of Ω' is then performed using the following rule, where $\Omega' = \{x : x_j \in [l'_j, u'_j], \forall j \in N\}$.

- Partition $J_{r\tau}$ into the following three disjoint and collectively exhaustive subsets, based on the sets $E_{r\tau j}$ of Equation (3.5).

$$\begin{aligned} J_{r\tau}^1 &= \{j \in J_{r\tau} : E_{r\tau j} = \{l'_j, u'_j\}\} \\ J_{r\tau}^2 &= \{j \in J_{r\tau} : E_{r\tau j} = \{\bar{x}_{r\tau j}\}\} \text{ and} \\ J_{r\tau}^3 &= J_{r\tau} - J_{r\tau}^1 \cup J_{r\tau}^2. \end{aligned} \quad (5.9)$$

- Compute

$$\theta_j = \begin{cases} \min\{\hat{x}_j - l'_j, u'_j - \hat{x}_j\} & \text{if } j \in J^1_{r\tau} \\ \max\{|\hat{x}_j - \bar{x}_{r\tau j}|, \min\{\hat{x}_j - l'_j, u'_j - \hat{x}_j\}\} & \text{if } j \in J^2_{r\tau} \\ \frac{u'_j - l'_j}{2} & \text{if } j \in J^3_{r\tau} \end{cases} \tag{5.10}$$

and set

$$\tilde{x}_j = \begin{cases} \hat{x}_j & \text{if } j \in J^1_{r\tau} \\ \bar{x}_{r\tau j} & \text{if } j \in J^2_{r\tau} \\ \frac{u'_j + l'_j}{2}, & \text{if } j \in J^3_{r\tau}. \end{cases} \tag{5.11}$$

- Select

$$p \in \text{argmax}\{(u'_j - l'_j)\theta_j : j \in J_{r\tau}\} \tag{5.12}$$

and partition Ω' by subdividing the interval $[l'_p, u'_p]$ into $[l'_p, \tilde{x}_p]$ and $[\tilde{x}_p, u'_p]$.

5.2.2 Algorithmic Statement

Step 0: Initialization. Initialize by setting $x^* = \emptyset$, $v^* = \infty$, $s = 1, Q_s = \{1\}$, $q(s) = 1$, and $\Omega^1 = \Omega$. Solve $RLT(\Omega)$ and let $(\hat{x}, \hat{X})$ be the solution obtained of objective value $LB_1 = v[RLT(\Omega)]$. If $\hat{x}$ is feasible to $FP(\Omega)$ (perhaps after using some heuristic local search method or some Newton-Raphson iterations) update x^* and v^*. If $v^* \leq LB_1 + \epsilon$, where $\epsilon \geq 0$ is some

accuracy tolerance, then stop with x^* as the prescribed solution to $FP(\Omega)$. Otherwise, select a branching variable x_p according to the Branching Rule discussed in Section 5.2.1 and proceed to Step 1.

Step 1: Partitioning Step. Partition $\Omega^{q(s)}$ into two sub-hyperrectangles by splitting the interval for x_p as prescribed by the Branching Rule of Section 5.2.1. Replace $q(s)$ by these two new node indices in Q_s.

Step 2: Bounding Step. Solve the RLT linear programming relaxation for each of the two new nodes generated, and update the incumbent solution if possible, as in the initialization step.

Step 3: Fathoming Step. Fathom any non-improving nodes by setting $Q_{s+1} = Q_s - \{q \in Q_s : LB_q + \epsilon \geq v^*\}$. If $Q_{s+1} = \emptyset$, then stop. Otherwise, increment s by one and proceed to Step 4.

Step 4: Node Selection Step. Select an active node $q(s) \in$ argmin $\{LB_q : q \in Q_s\}$, and return to Step 1.

The following theorem asserts that the above branch-and-bound algorithm converges to a global optimum to Problem $FP(\Omega)$.

Theorem 1 (Convergence Result). *The algorithm of Section 5.2.2 (run with $\epsilon = 0$) either terminates finitely with the incumbent solution being optimal to $FP(\Omega)$, or else an infinite sequence of stages is generated such that along any infinite branch of the branch-and-bound tree, any accumulation point of the x-variable part of the linear programming relaxation solutions generated for the node subproblems solves $FP(\Omega)$.*

Proof. The case of finite termination is clear. Hence, suppose that an infinite sequence of stages is generated. Consider any infinite branch of

the branch-and-bound tree associated with a nested sequence of partitions $\{\Omega^{q(s)}\}$ for stage s in some index set S. Hence,

$$v[FP(\Omega)] \geq LB(s) \equiv v[RLT(\Omega^{q(s)})] \equiv \sum_{t \in T_0} \Phi_{0t}^{L(\Omega^{q(s)})}(x^{q(s)}, X^{q(s)}) \tag{5.13}$$

where for each node $q(s)$, $s \in S$, $(x^{q(s)}, X^{q(s)})$ denotes the optimal solution obtained for $RLT(\Omega^{q(s)})$. Moreover, let $[l^{q(s)}, u^{q(s)}]$ be the associated vectors of lower and upper bounds that define $\Omega^{q(s)}$. By taking any convergent subsequence if necessary, suppose that

$$\{x^{q(s)}, X^{q(s)}, l^{q(s)}, u^{q(s)}\}_S \to (x^*, X^*, l^*, u^*). \tag{5.14}$$

We must show that x^* solves Problem $FP(\Omega)$.

Now, over the infinite sequence of nodes $\{q(s), s \in S\}$, there exists a term (r, τ) for $\tau \in T_r, r \in \{0, 1, \ldots, m\}$, that is picked infinitely often via (5.8). Let $S_1 \subseteq S$ be the stages for which a partitioning is performed based on this term (r, τ) using the Branching Rule discussed in Section 5.2.1. Hence, we have from (5.8) that,

$$\begin{aligned}
&\alpha_{r\tau} \prod_{j \in J_{r\tau}} [f_{r\tau j}(x_j^{q(s)})] - \Phi_{r\tau}^{L(\Omega^{q(s)})}(x^{q(s)}, X^{q(s)}) \\
&\geq \alpha_{it} \prod_{j \in J_{it}} [f_{itj}(x_j^{q(s)})] - \Phi_{it}^{L(\Omega^{q(s)})}(x^{q(s)}, X^{q(s)}) \\
&\forall \quad i = 0, 1, \ldots, m, t \in T_i, \text{ for each } s \in S_1.
\end{aligned} \tag{5.15}$$

By the partitioning strategy, over the nested sequence of nodes $\{q(s), s \in S_1\}$, there exists some index $p \in J_{r\tau}$ that is selected infinitely often for partitioning according to

$$\theta_p^{q(s)}(u_p^{q(s)} - l_p^{q(s)}) \geq \theta_j^{q(s)}(u_j^{q(s)} - l_j^{q(s)}) \quad \forall j \in J_{r\tau}, \tag{5.16}$$

where $\theta_j^{q(s)}$ is computed as in (5.10) for node $q(s)$. Let $S_2 \subseteq S_1$ index the set of stages where this occurs.

Case (i): $p \in J^1{}_{r\tau}$. In this case, by (5.11), we partition the interval for x_p at $\tilde{x}_p \equiv x_p^{q(s)}$ for each $s \in S_2$. It is evident that

$$x_p^{q(s)} \in [l_p^{q(s)}, u_p^{q(s)}], \quad \text{while } x_p^{q(s)} \notin (l_p^{q(s')}, u_p^{q(s')}) \text{ for any } s' > s, s' \in S_2. \tag{5.17}$$

Hence, in particular, $x_p^* \in [l_p^*, u_p^*]$. We argue that x_p^* is not an interior point of $[l_p^*, u_p^*]$. If $x_p^* \in (l_p^*, u_p^*)$, then there is a $T \in S_2$ such that when $s \geq T, s \in S_2$, we have $x_p^{q(s)} \in (l_p^*, u_p^*)$. But $(l_p^*, u_p^*) \subseteq (l_p^{q(s)}, u_p^{q(s)})$ for all $s \in S_2$. This means that $x_p^{q(T)} \in (l_p^{q(s')}, u_p^{q(s')})$, for all $s' > T, s' \in S_2$, a contradiction to (5.17). Therefore, we must have that either $x_p^* = l_p^*$ or $x_p^* = u_p^*$. In either case, by (5.10), $\{\theta_p^{q(s)}\} \to 0$ as $s \to \infty$, $s \in S_2$. Thus, by (5.16), we either have $\{u_j^{q(s)} - l_j^{q(s)}\} \to 0$ or $\{\theta_j^{q(s)}\} \to 0$ for all $j \in J_{r\tau}$.

Consider any $j \in J_{r\tau}$. If $j \in J_{r\tau}^1$ and $\{\theta_j^{q(s)}\} \to 0$, then by (5.10),we will have $x_j^* = l_j^*$ or $x_j^* = u_j^*$. If $j \in J_{r\tau}^1$ and $\{u_j^{q(s)} - l_j^{q(s)}\} \to 0$, we will again have $x_j^* = l_j^* = u_j^*$. Hence, if $j \in J_{r\tau}^1$, x_j^* must equal at least one of the bounds l_j^*, u_j^*. Similarly if $j \in J_{r\tau}^2$ and $\{\theta_j^{q(s)}\} \to 0$, then by (3.5) and (5.10), $x_j^* = \lim_{s\to\infty} \bar{x}_{r\tau j}^{q(s)} = l_j^* = u_j^*$. If $j \in J_{r\tau}^2$ and $\{u_j^{q(s)} - l_j^{q(s)}\} \to 0$, we again have $x_j^* = l_j^* = u_j^*$. Finally, if $j \in J_{r\tau}^3$ and $\theta_j^{q(s)} \to 0$, by (5.10), we have $l_j^* = u_j^* = x_j^*$. Likewise, if $j \in J_{r\tau}^3$ and $\{u_j^{q(s)} - l_j^{q(s)}\} \to 0$, we again have that $x_j^* = l_j^* = u_j^*$.

To summarize, we have shown thus far that

- if $j \in J^1_{r\tau}$, then x^*_j equals one of the limiting interval end points l^*_j, u^*_j,
- if $j \in J^2_{r\tau} \cup J^3_{r\tau}$, then $x^*_j = l^*_j = u^*_j$.

Now, if $j \in J^1_{r\tau}$, they by definition, $f_{r\tau j}(x_j)$ coincides with $g^{\Omega_j^{q(s)}}_{r\tau j}(x_j)$ at the end points of the interval Ω_j. In case $l^*_j < u^*_j$, since x^*_j is equal to one of the end points, we have by Property 1 from Chapter 3 that as $s \to \infty$, $s \in S_2$,

$$g^{\Omega_j^{q(s)}}_{r\tau j}(x_j^{q(s)}) \to g^{\Omega_j^*}_{r\tau j}(x^*_j) = f_{r\tau j}(x^*_j). \tag{5.18}$$

Otherwise, $j \in J^1_{r\tau} \cup J^2_{r\tau} \cup J^3_{r\tau}$ such that $x^*_j = l^*_j = u^*_j$. By Property 1, we again have in this case that as $s \to \infty$, $s \in S_2$,

$$g^{\Omega_j^{q(s)}}_{r\tau j}(x_j^{q(s)}) \to f_{r\tau j}(x^*_j). \tag{5.19}$$

Therefore, noting (5.1), we conclude that

$$\alpha_{r\tau} \prod_{j \in J_{r\tau}} f_{r\tau j}(x^*_j) = \Phi^{R(\Omega^*)}_{r\tau}(x^*) \tag{5.20}$$

where

$$\Phi^{R(\Omega^*)}_{r\tau}(x^*) = \lim_{s \to \infty, s \in S_2} \alpha_{r\tau} \prod_{j \in J_{r\tau}} g^{\Omega_j^{q(s)}}_{r\tau j}(x_j^{q(s)}). \tag{5.21}$$

Furthermore, by Lemma 6, we have

$$\Phi^{R(\Omega^*)}_{r\tau}(x^*) = \Phi^{L(\Omega^*)}_{r\tau}(x^*, X^*). \tag{5.22}$$

Equations (5.20) and (5.22) imply the following important identity:

$$\alpha_{r\tau} \prod_{j \in J_{r\tau}} f_{r\tau j}(x^*_j) = \Phi^{L(\Omega^*)}_{r\tau}(x^*, X^*). \tag{5.23}$$

Consequently, as we let $s \to \infty$, $s \in S_2$, the left-hand side of (5.15) approaches zero. Hence, from the right-hand side of (5.15), we deduce that

$$\alpha_{it} \prod_{j \in J_{it}} [f_{itj}(x_j^*)] - \Phi_{it}^{L(\Omega^*)}(x^*, X^*) \leq 0 \quad \forall i, t. \tag{5.24}$$

Since $(x^{q(s)}, X^{q(s)})$ solves $RLT(\Omega^{q(s)})$, we must have

$$\beta_i \geq \sum_{t \in T_i} \Phi_{it}^{L(\Omega^{q(s)})}(x^{q(s)}, X^{q(s)}) \quad \forall i = 1, \ldots, m. \tag{5.25}$$

Taking limits as $s \to \infty, s \in S_2$, we have

$$\beta_i \geq \sum_{t \in T_i} \Phi_{it}^{L(\Omega^*)}(x^*, X^*). \tag{5.26}$$

From (5.24) and (5.26), we get

$$\sum_{t \in T_i} \alpha_{it} \prod_{j \in J_{it}} [f_{itj}(x_j^*)] \leq \beta_i, \forall i = 1, \ldots, m, \tag{5.27}$$

or that x^* is feasible to $FP(\Omega)$. Moreover, by (5.13) and (5.24), for $i = 0$, taking limits as $s \to \infty$, $s \in S_2$, we get

$$v[FP(\Omega)] \geq \sum_{t \in T_0} \Phi_{0t}^{L(\Omega^*)}(x^*, X^*) \geq \sum_{t \in T_0} \alpha_{0t} \prod_{j \in J_{0t}} f_{0tj}(x_j^*) \equiv f_0(x^*). \tag{5.28}$$

Thus from (5.27) and (5.28),we conclude that x^* solves $FP(\Omega)$.

Hence, for Case (i), we have established the convergence of the algorithm.

Case (ii): $p \in J_{r\tau}^2$. In this case, from (5.11), we split the interval $[l_p^{q(s)}, u_p^{q(s)}]$ at $\tilde{x}_p \equiv \bar{x}_{r\tau p}^{q(s)}, \forall s \in S_2$. By Condition A, this leads to $l_p^* = u_p^* = x_p^*$,

and therefore $\{\theta_p^{q(s)}\} \to 0$. Consequently, the remainder of the proof identically follows that for Case (i).

Case (iii): $p \in J_{r\tau}^3$. In this case, by (5.11), we simply sequentially bisect the interval $[l_p^{q(s)}, u_p^{q(s)}]$, and so, we must have $l_p^* = u_p^* = x_p^*$, or that again, $\{\theta_p^{q(s)}\} \to 0$. Hence the argument for this case is also similar to that for Case (i).

This completes the proof. $\square$.

5.3 An Illustrative Example

In this section, we present a numerical example to illustrate several features of the generic methodology that has been developed to solve problem $FP(\Omega)$. In particular, we demonstrate how the two bounding strategies discussed in Chapter 4 can be applied to generate different bounding polynomial programs. Also, we show how a problem that appears to violate the assumptions of Chapter 3 can be readily manipulated to satisfy the stated requirements by using some suitable transformations. Finally, we illustrate some techniques that can be used to further tighten the LP relaxations and the polynomial approximations.

Consider the following example adapted from Hock and Schittkowski (1981).

$$\begin{aligned} \text{Minimize} \quad & \sin\left(\frac{\pi x_1}{12}\right)\cos\left(\frac{\pi x_2}{16}\right) \\ \text{subject to} \quad & 4x_1 - 3x_2 = 0 \\ & 0 \leq x_1 \leq 20, 0 \leq x_2 \leq 20. \end{aligned} \tag{5.29}$$

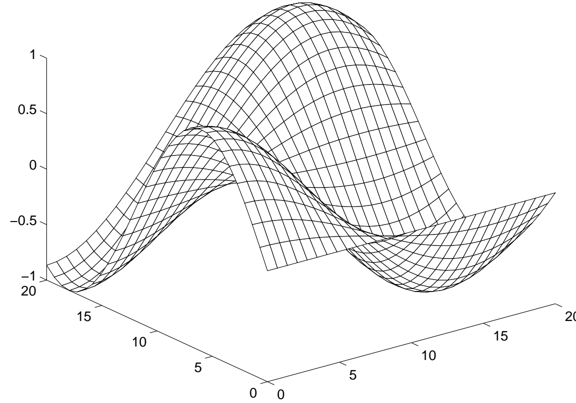

Figure 5.1: Objective Function for the Illustrative Example.

Figure 5.1 depicts the nonconvexity in the objective function. The problem has a unique global minimum at $x = [9, 12]$ with an optimal objective function value of -0.5.

Notice that the given form of the problem does not satisfy the assumption that requires $f_{itj}(x_j) \geq 0 \ \forall i, t, j$. This can be rectified by equivalently transforming the original problem into the following:

$$\begin{aligned} \text{Minimize} \quad & [\sin(\frac{\pi x_1}{12}) + 1][\cos(\frac{\pi x_2}{16}) + 1] - \sin(\frac{\pi x_1}{12}) - \cos(\frac{\pi x_2}{16}) - 1 \\ \text{subject to} \quad & 4x_1 - 3x_2 = 0 \\ & 0 \leq x_1 \leq 20, 0 \leq x_2 \leq 20. \end{aligned} \tag{5.30}$$

Next, let us illustrate the use of the Mean-Value Theorem method and the Chebyshev interpolation method in order to generate lower (upper) bounding polynomials $g_{itj}(x_j)$ as discussed in Chapter 4. For example, suppose that we want to construct second-order polynomial lower bounding functions for the

univariate functions $f_{0,1,1}(x_1) = \sin(\frac{\pi x_1}{12}) + 1$ and $f_{0,1,2}(x_2) = \cos(\frac{\pi x_2}{16}) + 1$ on the interval $[0, 20]$. The Chebyshev interpolation approach generates $g_{0,1,1}(x_1) = -0.0109x_1^2 + 0.1422x_1 + 0.4251$, and $g_{0,1,2}(x_2) = 0.0058x_2^2 - 0.2210x_2 - 1.9360$. On the other hand, by applying the Mean-Value Theorem Method with $x_0 = 10$, we obtain $g_{0,1,1}(x_1) = -0.0343x_1^2 + 0.4587x_1 + 0.3403$ and $g_{0,1,2}(x_2) = -0.0193x_2^2 + 0.2041x_2 + 0.5037$.

The sets E_{itj} defined by (3.5) are of different types for these two construction methods. For the Chebyshev interpolation method, we have $E_{0,1,1} = E_{0,1,2} = \emptyset$, whereas for the Mean-Value Theorem approach, since $x_0 = 10$, we obtain $E_{0,1,1} = E_{0,1,2} = \{10\}$.

Recall that in Chapter 3, we assumed that these bounding polynomial functions $g_{itj}(x_j)$ have the property that they are all nonnegative. However, the univariate concave function $g_{0,1,1}(x_1) = -0.0343x_1^2 + 0.4587x_1 + 0.3403$ generated by the Mean-Value Theorem method achieves its minimum on $[0, 20]$ at $x_1 = 20$, with $g_{0,1,1}(20) = -4.2057$. As suggested at the end of Chapter 3, this situation can be rectified by transforming the problem into an equivalent convenient format. We discuss two such transformations below.

First, using the fact that $g_{0,1,1}(x_1) \geq -5$, we have that the nonnegative function $g_{0,1,1}(x_1) + 5$ is a valid lower for $\sin(\frac{\pi x_1}{12}) + 6$. Hence, (5.30) is transformed into the following equivalent format.

$$\begin{aligned}
\text{Minimize} \quad & [\sin(\frac{\pi x_1}{12}) + 6][\cos(\frac{\pi x_2}{16}) + 1] - \sin(\frac{\pi x_1}{12}) - 6\cos(\frac{\pi x_2}{16}) - 6 \\
\text{subject to} \quad & 4x_1 - 3x_2 = 0 \\
& 0 \leq x_1 \leq 20, 0 \leq x_2 \leq 20.
\end{aligned} \tag{5.31}$$

Since $\sin(\frac{\pi x_1}{12}) + 6 \geq \bar{g}_{0,1,1}(x_1) \equiv (g_{0,1,1}(x_1) + 5) \geq 0$ and $\cos(\frac{\pi x_2}{16}) + 1 \geq g_{0,1,2}(x_2)$, we have that $[\bar{g}_{0,1,1}(x_1)] \times [g_{0,1,2}(x_2)]$ provides a valid lower bounding function for the first product term in the objective function. Notice that we need not require $g_{0,1,2}(x_2)$ to be nonnegative for this statement to hold true here. For the remaining two nonlinear terms in the objective function, we generate fourth-order lower bounding polynomial functions using the Mean-Value Theorem method to obtain $-\sin(\frac{\pi x_1}{12}) \geq -0.0002x_1^4 + 0.0052x_1^3 - 0.0226x_1^2 - 0.11x_1 - 0.4212$, and $-\cos(\frac{\pi x_2}{16}) \geq -0.00006x_2^4 + 0.0013x_2^3 - 0.0096x_2^2 + 0.2269x_2 - 1.6227$. The resulting lower bounding nonconvex polynomial programming problem is stated below.

$$\begin{aligned} \text{Minimize} \quad & p(x_1, x_2) \\ \text{subject to} \quad & 4x_1 - 3x_2 = 0 \\ & 0 \leq x_1 \leq 20, 0 \leq x_2 \leq 20 \end{aligned} \tag{5.32}$$

where $p(x_1, x_2) = -0.0002x_1^4 - 0.0004x_2^4 + 0.0007x_1^2x_2^2 + 0.005x_1^3 + 0.0078x_2^3 - 0.0069x_1^2x_2 - 0.0088x_1x_2^2 - 0.0398x_1^2 + 0.094x_1x_2 - 0.16x_2^2 + 0.121x_1 + 2.452x_2 - 13.47$. To generate the LP relaxation, we first recognize that (5.32) is a polynomial program of order $\delta = 4$, and therefore, we include all the fourth-order bound-factor product as well as constraint-factor product RLT constraints. For example, one such bound-factor product based RLT constraint has the form $[(x_1 - 0)^2(20 - x_1)(20 - x_2)]_L = [400x_1^2 - 20x_1^3 - 20x_1^2x_2 + x_1^3x_2]_L = 400X_{11} - 20X_{111} - 20X_{112} + X_{1112} \geq 0$, and one valid constraint-factor product based RLT constraint is $[(4x_1 - 3x_2)(x_1^2x_2)]_L = [4x_1^3x_2 - 3x_1^2x_2^2]_L = 4X_{1112} - 3X_{1122} = 0$. The problem is then solved by the branch-and-bound

algorithm outlined in Section 5.2. If we fathom according to the practical criterion

$$\text{LB} \geq \text{UB} - \epsilon \max\{1, |\text{UB}|\} \tag{5.33}$$

with $\epsilon = 0.05$, the problem is solved after enumerating a total of 49 nodes. The best solution obtained is $x_1 = 8.9063, x_2 = 11.8750$ with $f(x_1, x_2) = -0.4994$.

Note that for any bounding interval $[a_2, b_2]$ for x_2 in the branch-and-bound process, the constraint $4x_1 - 3x_2 = 0$ implies that $x_1 \in [(3/4)a_2, (3/4)b_2]$. We can explicitly use these revised bounds on x_1 to generate the RLT constraints (5.5), hence resulting in a potentially tighter relaxation. Resolving the problem with this revision, the branch-and-bound algorithm generates only 25 nodes, a significant reduction from the previous case. This type of range or bounding interval reduction can be further enhanced by sequentially minimizing and maximizing each variable in turn subject to the RLT relaxation constraints, plus a linearized form of an objective cut that requires the objective function to take on a value less than or equal to some known upper bound. Additionally, other valid RLT constraints can be included to strengthen the lower bounding problem. For example, $(x_1 - a)^2(x_2 - b)^2 \geq 0$ is a valid inequality for any values of a and b. Judiciously selected constraints of this type (see Sherali and Tuncbilek, (1997)) can be generated at the initial node, for example, and appended to the problem to tighten its relaxation. These types of issues are pursued in detail in Chapter 6 when we deal with computational aspects of this algorithm.

The second transformation we explore to deal with the nonnegativity

assumptions follows the discussion at the end of Chapter 4. Here, we first substitute y_1 for $\sin(\frac{\pi x_1}{12})$ and y_2 for $\cos(\frac{\pi x_2}{16})$ to derive the following equivalent problem

$$\begin{aligned}
\text{Minimize} \quad & y_1 y_2 \\
\text{subject to} \quad & y_1 = \sin(\frac{\pi x_1}{12}) \\
& y_2 = \cos(\frac{\pi x_2}{16}) \\
& 4x_1 - 3x_2 = 0 \\
& 0 \le x_1 \le 20, 0 \le x_2 \le 20, \\
& -1 \le y_1 \le 1, -1 \le y_2 \le 1.
\end{aligned} \tag{5.34}$$

Now, since there is no product term that requires a polynomial approximation, we are no longer concerned with the nonnegativity assumption. (Such a transformation is used to pre-process a problem in our final implementation of the algorithm.) Furthermore, as mentioned in Chapter 4, we can use different methods to construct multiple alternative bounding polynomials for the functions defined by y_1 and y_2. For example, we solved (5.34) by replacing the two nonlinear equality constraints by four equivalent inequalities, and used both the Mean-Value Theorem and Chebyshev interpolation methods to generate second-order polynomial approximations. This strategy proves to be very effective computationally. At the initial node, it produces a lower bound of -1, and with a tighter tolerance of $\epsilon = 0.01$, the problem was solved after enumerating only 9 nodes, producing a more accurate solution given by $x_1 = 9.0532, x_2 = 12.0709$, with $f(x_1, x_2) = -0.4998$. (With the previously used value of $\epsilon = 0.05$, this reformulation requires the enumera-

tion of only 6 nodes, yielding a solution $x_1 = 8.8618$ and $x_2 = 11.8158$, with $f(x_1, x_2) = -0.4987$.)

Chapter 6

Implementation

In this chapter, we discuss algorithmic issues and computational strategies regarding the implementation of the algorithm developed in Chapter 5. In the first section, we outline the overall design of the implementation. Topics such as problem pre-processing, RLT constraint generation, range-reduction, and constraint filtering are discussed. Several relevant C++ classes developed for this algorithm are examined in the second section.

6.1 Design Issues

There are three major elements in our proposed algorithm: bounding polynomial generation methods, RLT constraint construction strategies, and the branch-and-bound framework. These are the fundamental ingredients that ensure the validity of the algorithm. Other techniques such as range-reduction and constraint filtering can also be used to contribute toward the overall successful performance of the algorithm.

6.1.1 Pre-processing the Problem.

In the illustrative example in Section 5.3, we have seen that by suitably manipulating the original problem , we can circumvent apparent violations of the assumptions stated in Chapter 3. Furthermore, from our computational experience, it turns out that by adopting such a reformulation, not only is the implementation considerably simplified, but significant computational benefits accrue as well.

In general, at the pre-processing stage, we require that the problem is first transformed into a convenient form by replacing all the non-polynomial univariate functions with new variables. For example, consider the following problem.

$$\begin{aligned} \text{Minimize} \quad & \sin(x_3) + (x_1 - x_2)^2 - 1.5x_1 + 2.x_2 + 1 \\ \text{subject to} \quad & x_1 + x_2 = x_3 \\ & -1.5 \le x_1 \le 4, -3 \le x_2 \le 3. \end{aligned} \tag{6.1}$$

We transform it into the following equivalent problem.

$$\begin{aligned} \text{Minimize} \quad & x_4 + (x_1 - x_2)^2 - 1.5x_1 + 2.5x_2 + 1 \\ \text{subject to} \quad & x_1 + x_2 = x_3 \\ & x_4 - \sin(x_3) \ge 0 \\ & \sin(x_3) - x_4 \ge 0 \\ & -1.5 \le x_1 \le 4, -3 \le x_2 \le 3, -1 \le x_4 \le 1. \end{aligned} \tag{6.2}$$

After this transformation, we classify the constraints of the new problem

into four different categories: linear inequalities, equality constraints (linear and polynomial), polynomial inequalities, and equality constraints of the form $x_{n+i} = f(x_j)$. Here $f(x_j)$ is a non-polynomial univariate function in x_j and x_{n+i} is a newly defined variable. It is directly observed that after the initial transformation, the new objective function is simply a polynomial function. Appropriately implied bounds will be set for these newly defined variables.

There are three important advantages for such a reformulation. First, it transforms the problem into a new format that is easier to manage in a general implementation framework. Second, since there are no explicit products of non-polynomial functions, the nonnegativity requirements on the bounding polynomials are automatically dropped. Third, this permits a more systematic way to control the generation of RLT constraints that may involve products of different sets of original constraints.

6.1.2 Algorithms for Bounding Polynomials

In Chapter 4, we present two methods to generate valid bounding polynomials. One is based on the Mean-Value Theorem and the other is based on Chebyshev interpolation polynomials. The first approach is very straightforward while the implementation of the second method is more involved. The difficulty with the latter lies in the fact that we are interested in obtaining the coefficients of the approximating polynomial. This is rarely done in numerical analysis where one is usually concerned with only the values produced by the approximating polynomial at any given point. While algorithms to achieve this task exist, they usually suffer from instability, especially for higher order

polynomials. This instability is due to the inherent ill-conditioning of the matrix of coefficients (see Press *et al.* (1992)). Our implementation of this module is based on the algorithm given in Isaacson and Keller (1966). Other more sophisticated and better implementations are discussed in Dalquist and Bjorck (1974) and Stoer and Burlirsch (1993). Here, we emphasize the fact that, as a general rule, this module is assumed to be specified by the user for a particular problem instance. We optionally include this module as part of the developed package. The software, however, has a generic interface that allows the user to adopt any alternative valid bounding polynomial strategies.

6.1.3 Constructing RLT Constraints Under Different Options

The construction of RLT constraints is a very critical part of this algorithm. There are two issues that need to be considered here. On the one hand, we intend to generate a sufficient class of RLT constraints that involve both bound-factors and other structural constraints to ensure the tightness of the resulting LP relaxation. On the other hand, we hope to control the size of the resulting LP problem to keep it manageable. Our implementation always generates the basic bound-factor RLT constraints that are necessary for the theoretical convergence. In addition, we offer the user the flexibility to generate additional RLT constraints under different options to potentially improve the lower bounding LP relaxations.

There are two basic sets of RLT constraints that are always appended to the polynomial program at the reformulation phase. They are the bound-factor RLT constraints and the equality based RLT constraints. The pro-

cedure to construct bound-factor product RLT constraints is discussed in Section 5.1.2. To generate equality based RLT constraints of order δ, we multiply any original equality constraint with all the possible RLT variables, so long as the orders of the resulting polynomials do not exceed δ. For instance, suppose that we have a problem with two variables x_1, x_2 and two equality constraints $x_1 + x_2 = 1, \quad x_1x_2 + x_2 = 3$. The following list enumerates all the RLT constraints that are based on the above two equality constraints for $\delta = 3$.

$$\begin{aligned}
(x_1 + x_2 = 1) \times x_1 &\rightarrow x_{11} + x_{12} = x_1 \\
(x_1 + x_2 = 1) \times x_2 &\rightarrow x_{12} + x_{22} = x_2 \\
(x_1 + x_2 = 1) \times x_{11} &\rightarrow x_{111} + x_{112} = x_{11} \\
(x_1 + x_2 = 1) \times x_{12} &\rightarrow x_{112} + x_{122} = x_{12} \\
(x_1 + x_2 = 1) \times x_{22} &\rightarrow x_{122} + x_{222} = x_{22} \\
(x_1x_2 + x_2 = 3) \times x_1 &\rightarrow x_{112} + x_{12} = 3x_1 \\
(x_1x_2 + x_2 = 3) \times x_2 &\rightarrow x_{122} + x_{22} = 3x_2.
\end{aligned} \tag{6.3}$$

In essence, this process is a special case of constructing *constraint-factor product* mentioned in Section 5.1.2. Instead of multiplying equalities with bound factors, we simply multiply them with RLT variables, as the latter imply all possible constraint-factor RLT constraints that involve these equalities.

Additionally, we define three flag variables {*optionl, optionp, optionn*} that a user can set to signal the inclusion of additional valid RLT constraints.

1. *optionl*

- *optionl* = 1: In this case, each linear constraint is multiplied with any combination of $\delta - 1$ bound factors so that the resulting polynomials are of order δ.
- *optionl* = 2: In this case, all the linear constraints are considered as *special bound-factors* and they are grouped with the *basic bound-factors* to form the set of *general bound-factors.* A bound-factor product constraint is generated by taking the product of any δ of these general bound-factors.

2. *optionp*

- *optionp* = 1: Each polynomial of order $d < \delta$ is multiplied with any combination of $\delta - d$ basic bound-factors.
- *optionp* = 2: Each polynomial of order $d < \delta$ is multiplied with any combination of $\delta - d$ general bound-factors.

3. *optionn*

- *optionn* = 1: Each bounding polynomial of order $d < \delta$ is multiplied with any combination of $\delta - d$ basic bound-factors.
- *optionn* = 2: Each bounding polynomial of order $d < \delta$ is multiplied with any combination of $\delta - d$ general bound-factors.

The default set of RLT constraints ensures the convergence of the algorithm, in particular, by driving the gaps between the LP relaxations and the parent polynomial programs to zero. The other options provide the flexibility to judiciously include other valid constraints to enhance the LP relaxations.

Also, the user may choose to simply obtain a good lower bound at the initial node by including additional valid RLT constraints, and test this against a derived heuristic solution (without performing any branching steps).

6.1.4 Range-Reduction

During the branch-and-bound process, once a good incumbent solution is obtained, it is often effective to perform several steps of range-reduction by sequentially minimizing and maximizing each variable in turn, subject to the RLT relaxation constraints, plus a linearized objective function cut. Such a strategy is then embedded in a branch-and-cut algorithm. Other more advanced range-reduction techniques are discussed in detail in Tuncbilek (1994). In our implementation, we adopted the above simplest form of range-reduction and applied it only at the initial node.

6.1.5 Constraint Filtering Techniques

Sherali and Tuncbilek (1997) suggest several RLT constraint filtering schemes that are exploited in solving polynomial programming problems. More recently, Sherali, Smith and Adams (1997) exam the possibility of creating a reduced first-level representation via RLT in solving mixed 0-1 integer programming problems. Ideas in this study can be extended to the continuous domain in the future. In our preliminary version of the implementation, we do not consider any constraint filtering methods. However, proper interfaces are created to facilitate the future inclusion of such a technique into the model and the computational time is expected to reduce considerably.

6.1.6 Branch-and-Bound Framework

A successful implementation of a branch-and-bound algorithm relies heavily on the data structure, node storage strategies, and branching variable selection rules. Sherali and Myers (1985) have conducted elaborate computational experiments on node and branching variable selection and storage reduction strategies within a special breadth-and-depth enumeration combination approach. Sherali and Tuncbilek (1997) use such an approach in implementing an algorithm to solve polynomial programming problems. For simplicity, we adopt the traditional *best-first* approach. In this case, the program maintains a set of active nodes. At each iteration, the active node that has the least lower bound is selected for partitioning. To conserve the run-time memory consumption, we create a priority queue to store these active nodes. For each active node, only a minimal degree of information is kept. Once a node is selected from the queue, the associated subproblem is generated.

6.2 C++ Classes

The algorithm is implemented entirely in C++, a language that supports the *objected-oriented programming* (OOP). The C++ classes developed can be readily embedded into future RLT engines. The resulting product is a unified environment within which any RLT based algorithm can be easily implemented and tested. During the coding stage, we set two important goals to guide our development: computational efficiency and user friendliness. The code for the program is fine-tuned to reflect a good balance between these two sometimes conflicting interests. In this section, we present some

explanation on several important C++ program modules. There are many important details that comprise the implementation, only the most crucial aspects are described here.

6.2.1 Constraint Class

A very fundamental and important building unit in our algorithm is a *constraint*. We constantly need to store , manipulate and perform various arithmetic operations involving the problem constraints. From an OOP point of view, it is highly necessary to implement a class that encompasses the capacity to represent and manipulate such RLT constraints. In this section, we present such a class.

RLT Constraint and its Representation

In the current context, we define an *RLT constraint* as any polynomial constraint that is linearized via the variable substitution rules specified in Section 5.1.2. For simplicity and uniformity, we assume that each constraint is in the form $p(x) \geq 0$. Such a format proves to be convenient later when we consider constraint multiplications. Any RLT constraint can be uniquely represented by three entities: an array of subscripts, an array of coefficients, and an integer indicating the number of terms on the left-hand side of the constraint. [1] For example, the constraint $4x_1 - x_{23} + x_3 - 4 \geq 0$ can be described by two arrays and one integer.

[1] A more efficient data structure for this class would be a link list.

```
/* Three C statements describing a constraint */
int n_term=4;
long subs[4]={1,23,3,0};
double coefs[4]={4,-1,1,-4};
```

Therefore, the initialization constructor for the *Constraint* class takes three arguments,

```
/* Initialization Constructor  */
Constraint(int num_tt, double *coefss, long *subss);
Constraint example(4,coefs,subs); /*4x1-x23+x3-4 >=0 */
```

Any RLT constraint can be initialized by calling this constructor.

A *bound-factor* is a special type of RLT constraint that takes a very simple form, $x_i \geq a_i$ or $x_i \leq b_i$. Note that we can certainly construct such a constraint by using the initialization constructor itself. However, since it takes a simpler form, and we encounter bound-factors very frequently, it is justified to develop a constructor of its own.

```
/* Constructor for Bound Factors */
Constraint(int lu, double a, long sub);

/* lu=-1 if lower bound, 1 if upper bound. */
```

Thus, bound-factors $x_1 \geq 3.1$ and $x_3 \leq 4$ can be built by the following statements.

```
Constraint lbf(-1,3.1,1);  /* x1 >=3.1*/
Constraint ubf(1,4,2);  /* x2 <=4 */
```

Notice that the overloading facility of the C++ language allows us to define several functions having the same name but different signatures. The compiler will automatically decide which one to use according to the parameters passed.

Constraint Multiplications

The only major mathematical operation we require for the *Constraint* class is multiplication. The user can define additional operations such as addition and linear combinations. These additional operations may be useful for an RLT based algorithm in which a surrogate constraint of a given set of constraints is to be constructed. The inheritance feature of the C++ language will make such an extension painless. Consider the product $0.3x_{13} \times 0.4x_{24}$ to yield $0.12x_{1234}$. Notice the subscript of the new variable is obtained by taking the union of $\{1,3\}$ and $\{2,4\}$ (retaining duplications when present), and ranging the resulting four digits in a nondecreasing order. Such an operation is very similar to the merge step in *Merge Sort* (see Knuth (1973)). The following member function Subunion implements the union of two RLT subscripts.

```
/* a member function that generate the product subscript */
```

```
long SubUnion(long sub1, long sub2)
{
   long ans=0;
   int i=0;
   while(sub1>0 || sub2>0)
   {
      if((sub1 %10)>=(sub2%10))
      {
         ans=ans+(sub1%10)*pow(10,i);
         sub1/=10;
      }
      else
      {
         ans=ans+(sub2%10)*pow(10,i);
         sub2/=10;
      }
   }
}
```

With this member function defined, the implementation of constraint multiplication is trivial. To make the resulting module more user-friendly, we overload the operator $*$, so that the member function that carries out the general constraint multiplication takes the following form.

```
Constraint operator *(const Constraint & right);
```

Thus, a general constraint multiplication

$$(x_2 + 3 \geq 0) \times (x_{13} - x_3 \geq 0) = (x_{123} - x_{23} + 3x_{13} - 3x_3 \geq 0)$$

can be accomplished by the following C++ program segment.

```
long subs[2]={13,3};
double coefs[2]={1,-1};
Constraint b(2,coefs, subs);
Constraint a(-1,-3,2); // bound factor
Constraint prod=a*b;
```

For equality constraints, the multiplication is a little different. Instead of multiplying an equality with bound-factors or other RLT constraints, we simply multiply it with RLT variables. For example, we might perform the operation

$$(x_3 + x_{14} - 3 = 0) \times x_{22} \rightarrow (x_{223} + x_{1224} - 3x_{22} = 0).$$

The resulting constraint is also an equality having the same coefficients. The only thing that has been changed is the subscripts on the associated variables. In the *Constraint* class, we have included a member function to perform this special type of multiplication.

```
/* Member function handling equality multiplication.*/
```

```
Constraint Constraint::EQMult(long sub)
{
   Constraint b=*this;
   for (int i=0;i<num_t;i++)
   {
      b.subs[i]=SubUnion(b.subs[i],sub);
   }
   return b;
}
```

The header file of the *Constraint* class is listed below for reference.

```
//constt.h
//header file for the constraint class

#ifndef __constraint__
#define __constraint__
#include<assert.h>
class Constraint{
private:
int num_t;      //# of terms
double *coefs;    //coefficients
long *subs;   // subscripts
char Sign(double coef);
```

```
void AddTerm(double coef, long sub);
long SubUnion(long sub1,long sub2);


public:
// constructors and destructor
Constraint();
Constraint(int lu, double a, long sub);
Constraint(int num_tt, double *coefss, long *subss);
Constraint(const Constraint &other);
Constraint & operator=(const Constraint &other);
~Constraint();
// access functions
int NumTerms(){return num_t;};
long GetSubs(int j);
double GetCoefs(int j);
// arithmetic
Constraint operator *(const Constraint & right);
Constraint EQMult(long sub);
// testing and debugging functions
void Display(void);
};
#endif
```

6.2.2 A Combination Class

In generating RLT constraints, we often encounter the following operation.

Given n constraints, take the product of these constraints, k at a time, allowing repetition.

This occurs, for example, in generating all the bound-factor product RLT constraints. A *Combination* class is developed for this purpose.

A Combination Generator and its Algorithm

Given n numbers $\{1, 2, \ldots, n\}$ and any integer $k \leq n$, a combination generator generates a list of all the combinations of choosing k out of n numbers, allowing repetition. To make the list unique, we require that the resulting combinations are in a component-wise lexicographic order. More specifically, if $c_i = \{a_1^i, a_2^i, \ldots, a_k^i\}$ and $c_{i+1} = \{a_1^{i+1}, a_2^{i+1}, \ldots, a_k^{i+1}\}$ are respectively the ith and $(i+1)$th combinations in the list, we require that $a_1^j, a_2^j, \ldots, a_k^j$ are in a nondecreasing order for $j = i, i+1$ and c_{i+1} is lexicographically greater than c_i. For instance, when $n = 3, k = 2$, the list is

$$\underbrace{\{1,1\}, \{1,2\}, \{1,3\}, \{2,2\}, \{2,3\}, \{3,3\}}_{\text{six combinations}}.$$

To generate such a list of combinations, we inductively construct the next combination in the list, given the current one.

Let $c = \{a_1, a_2, \ldots, a_k\}$ be the current combination. The next combination in the list can be generated using the following steps.

- Scan through each component of c from right to left, and find the first element a_j such that $a_j < n$.

- The next combination is $\{a_1, \ldots, a_{j-1}, \underbrace{a_j + 1, \ldots, a_j + 1}_{k-j+1 \text{ elements}}\}$.
- If no such a_j exists, then the end of the list is reached.

We call the element a_j identified above as the *pivot element*, and let *pindex* denote the index of this pivot element. At each iteration, we update the index of the pivot element to avoid unnecessary checks. The algorithm thus proceeds as follows.

Step 1. Initialize with $pindex = k$, $More = 1$, $C = \{\underbrace{1, 1, \ldots, 1}_{k \text{ elements}}\}$, and proceed to the next step.
Step 2. If $More = 0$, stop; otherwise, go to Step 3.
Step 3. Let $j = pindex$ and $C = \{a_1, \ldots, a_{j-1}, a_j, \ldots, a_k\}$. The new combination is $C = \{a_1, \ldots, a_{j-1}, a_j + 1, \ldots, a_j + 1\}$. If $a_1 = n$, set $More = 0$ and go to Step 2. Otherwise, go to Step 4.
Step 4. If $a_j + 1 = n$, set $pindex = j - 1$; else, let $pindex = k$. Return to Step 2.

It can be shown that for any given n and k, there are a total of $\binom{n+k-1}{k}$ combinations. This important formula is used in several different places in this chapter.

The header file for this Combination class is as follows.

```
// file:        comb.h
// purpose:     the header file of the Combination class
//              This class will generate all
//              the k out of n combinations with repetition
```

```
#ifndef __comb__
#define __comb__

class Combination{
public:
   Combination(int nn, int kk);  // constructor
   ~Combination();           // destructor
   void Initialize(void);  // reset
   void Current(int *comb) ; // the current combination
   int Next(void);    // generate the next combination
   long Current2Num(void); // the current comb in number
private:
   int *combination;
   int n, k;
   int more;
   int pindex; // index of the pivot element
};
#endif
```

Generating Bound-Factor Product RLT Constraints

With the help of this *Combination* class, we can construct bound-factor product based RLT constraints. Suppose that we have four bound-factors

$$1 \leq x_1 \leq 2, 3 \leq x_2 \leq 4,$$

and we are interested in composing all possible RLT constraints by taking the product of them, 3 at a time. The program for this task is given below.

```
double bds[4]={1,3,2,4};
Constraint boundfactors[4];
// initialize the bound factors
for(int i=0;i<2;i++)
{
   boundfactors[i]=Constraint(-1,bds[i],i);
   boundfactors[i+2]=Constraint(1,bds[i+2],i);
}
// assume f is defined elsewhere
int num=f(4+3-1,3); // number of total combinations.
int indices[3];
Combination list(4,3); // 4 factors, take 3 at a time
Constraint product;
do
{
   list.Current(indices);
   product=boundfactors[indices[0]];
   for(int j=1;j<3;j++)
      product=product*boundfactors[indices[j]];
}
while(list.Next());
```

Here f(int i, int j) computes $\binom{i}{j}$. In this case, since there are 4 bound-factors and $\delta = 3$, the total number of bound-factor product RLT constraints is $\binom{4+3-1}{3} = 20$.

Indices of the RLT Variables

The combinations constructed from the corresponding combination generator are closely related to the subscripts of the RLT variables. For example, a combination $\{1,1,2\}$ corresponds to the subscript $\{112\}$, treated as the integer 112. The member function

```
long Current2Num(void)
```

transform a combination into an integer that represents the subscript of an RLT variable. Suppose that a problem has three original variables x_1, x_2, x_3 and we apply a second-order RLT. The list of all relevant RLT variables is as follows:

$$x_1, x_2, x_3, x_{11}, x_{12}, x_{13}, x_{22}, x_{23}, x_{33}.$$

Notice that the subscripts correspond to the list of combinations of choosing 1 out of 3 followed by combinations of choosing 2 out of 3, allowing repetitions. This relationship can be exploited to generate all the subscripts of RLT variables in a given problem. It is also easy to see, in light of this relationship, that the total number of RLT variables is given by

$$\sum_{i=1}^{\delta} \binom{n+i-1}{i} = \binom{n+\delta-1}{\delta}. \tag{6.4}$$

Here n is the number of original variables and δ is the order of the RLT.

The following program segment uses the combination class to construct SUBRLT[], an array of all possible RLT variables in a lexicographic order.

```
int n=2; // number of original vars.
int RLTorder=3;
int RLTnum=f(n+RLTorder, order)-1;// number of RLT vars;
long SUBRLT[RLTnum];
int j=0;
for(int i=1;i<RLTorder;i++)
{
   Combination list(n, i);
   do
   {
      SUBRLT[j]=l.Current2Num();
      j++;
   }
   while(l.Next());
}
```

It is conventional that the constraint set in an LP problem is represented as a matrix. (Notice the difference between representation and storage. The matrix is a conceptual representation. The constraints might be actually stored using different data structures.) Since there are new RLT variables

and constraints being defined during the process of the model construction, it is important to have a systematic way to assign column numbers to the variables in the final constraint matrix. With the ability to compose an array of all RLT variables via SUBRLT[], such a task becomes trivial. It can be shown that if we assign column numbers to RLT variables according to the lexicographic order of the subscripts of these RLT variables, then, $X_{a_1 a_2 \ldots a_k}$ corresponds to the column number

$$\binom{n+k-1}{k} - \sum_{i=1}^{k} \binom{n-a_i+k-i}{k-i+1} + \binom{n+k-1}{k-1}. \tag{6.5}$$

In our implementation, however, instead of using the above formula, we adopt a more practical approach. For any n (the number of original variables) and δ (the order of RLT), we compose SUBRLT[], an array of subscripts of all the possible RLT variables up to order δ. Since the subscripts in SUBRLT[] are sorted in a lexicographic order, the column number of any RLT variable can be determined by performing a binary search using the subscript of this RLT variable as the key word. This method avoids the repeated evaluation of formula (6.5), which is numerically expensive.

Finally, we describe briefly the method we use to build equality based RLT constraints. For convenience, we define the *order* of an RLT variable as the number of digits in its subscript. As mentioned in Section 6.1.3, to generate equality RLT constraints, we take the product of any original equality with all possible RLT variables so that the order of the resulting polynomial is less than or equal to δ. Suppose that the equality is of order s, where $s \leq \delta$. Then, the highest order of RLT variables by which this equality can be multiplied is $t = \delta - s$. The index of the first tth order RLT

variable in SUBRLT[] can be obtained by using the formula $I = \binom{n+t}{t} - 1$. To produce all the RLT constraints based on this equality, we simply multiply this equality with the first I elements in SUBRLT[].

6.2.3 Problem Class and Sparse Matrix Class

The *Problem* class is responsible for loading the data, generating the constraint set according to various user options, and feeding the resulting data via a proper interface to an LP or NLP solver. The LP solver we use is CPLEX 4.0, which requires information on the nonzero elements in the constraints. Since all the RLT constraints are generated at run-time, there is no obvious way to determine the number of nonzero elements in the constraints before the actual construction process. For this reason, an auxiliary sparse matrix class *SPM* is developed to hold the resulting RLT constraint set. The *SPM* class uses an array of link lists as the underlying data structure. Each link list corresponds to one column of the constraint matrix. The memory for the sparse matrix is dynamically allocated when a particular location is initialized with a nonzero element.

6.2.4 Classes for Branch-and-Bound

There are three classes developed for the general branch-and-bound framework. The *BranchBound* class implements a general branch-and-bound procedure. It is the only program module that the user needs to work with. Detailed information about this class can be found in the tutorial attached in the appendix. The *Odlist* class implements a priority queue. This queue is used by the branch-and-bound algorithm to store active nodes, and it is a

priority queue because whenever a new node is added to this queue, it is appended to the proper location according to the rank of its lower bound. The *BBNODE* class is another auxiliary module that is used to represent a node in the branch-and-bound tree. Finally, we have developed C++ interfaces for CPLEX 4.0 and MINOS 5.4 to simplify the use of these two solvers.

The classes we have discussed in this section are used in the implementation of our algorithm. However, they are all generic enough and independent to be incorporated into other applications, especially those that involve the application of the RLT.

Chapter 7

Computational Experience

For evaluating the proposed algorithm, we use a set of fifteen test problems from the literature. The algorithm is tested using different bounding polynomial strategies as well as diverse RLT constraint generation options. The overall performance of the algorithm is very encouraging. For problems whose optimal solutions are known, the algorithm terminates with the correct solutions with a very competitive effort. For two test problems, by detecting global optimal solutions, we actually found better solutions than the ones previously reported in the literature. In most cases, the LP solutions obtained at the initial node produced tight lower bounds and often provided warm starts for MINOS that resulted in an optimal solution being detected at node zero itself (for nine out of fifteen cases). In the following, we first summarize in three tables the computational results obtained on a Sun Ultra-1 station running SunOS 5.5. Then, detailed information on each individual problem is presented.

Table 7.1: Computational Results with $\epsilon = 10^{-6}$.

Problem No	Optimal Value	Initial LB	No. of Nodes	CPU time (seconds)
1	-1.90596	-2	17	1.31
2	-4.601308	-5.12	13	0.3
3	-16.73	-28.5	27	1.06
4	0	0	1	negligible
5	17.014	16.68	5	8.34
6	-2	-3	3	negligible
7	-1.9132	-12.3	41	18.32
8	$-\sqrt{3}$	-2.15	23	2.61
9	-4.5142	-5.78679	13	0.32
10	-3.1336	-3.5539	17	0.68
11	-13.40196	-14.0028	67	4.14
12	-30665.5387	-30673.086	3	0.75
13	-5.6848	-5.6848	1	13.41
14	-750	-750	1	3.58
15	-1.031628	-81	35	283.5

Table 7.2: Computational Results with $\epsilon = 0.01$.

Problem No	No. of Nodes	CPU time (seconds)
1	17	1.31
2	7	0.18
3	9	0.47
4	1	negligible
5	1	5.71
6	3	negligible
7	27	13.06
8	9	1.01
9	13	0.32
10	5	0.26
11	67	4.12
12	1	0.24
13	1	13.41
14	1	3.58
15	29	269

Table 7.3: Computational Results with $\epsilon = 0.05$.

Problem No	No. of Nodes	CPU time (seconds)
1	15	1.24
2	3	negligible
3	3	0.26
4	1	negligible
5	1	5.71
6	3	negligible
7	21	10.37
8	5	0.64
9	13	0.32
10	3	0.18
11	1	0.09
12	1	0.24
13	1	13.41
14	1	3.58
15	27	244

Problem 1

(Floudas and Pardalos (1992), pp. 215)

$$\begin{aligned} \text{Minimize} \quad & \sin(x) + \sin(\frac{2x}{3}) \\ \text{subject to} \quad & 3.1 \leq x \leq 20.4. \end{aligned}$$

This problem has three local minima and one global minimum. The global solution is $x^* = 17.0393, f^* = -1.90596$. For the two nonlinear terms, we construct fourth-order polynomials using the Chebyshev Interpolation Polynomial approach. A fourth-order RLT is applied. MINOS is called to find upper bounds as mentioned above. At the initial node, the LP relaxation yields a lower bound of -2. Using this LP solution, MINOS locates a local minimum of -1.1983 at the first node. The global minimum is found by MINOS at the eighth node. The branch-and-bound process terminates successfully after enumerating a total of 17 nodes. The total CPU time used is 1.31 seconds.

Problem 2

(Floudas and Pardalos (1992), pp. 215)

$$\begin{aligned} \text{Minimize} \quad & \sin(x) + \sin(\frac{10x}{3}) + \log(x) - 0.84x \\ \text{subject to} \quad & 2.7 \leq x \leq 7.5. \end{aligned}$$

The global optimal solution is $f^* = -4.601308$. For the three non-polynomial terms, second-order bounding polynomials based on the Mean-Value Theorem are constructed. For the resulting polynomial program, a second-order RLT is applied. The branch-and-bound algorithm enumerates totally 13 nodes and uses 0.3 CPU second.

Problem 3

(Floudas and Pardalos (1990), pp. 31)

$$\begin{array}{rl} \text{Minimize} & -12x_1 - 7x_2 + x_2^2 \\ \text{subject to} & -2x_1^4 + 2 - x_2 = 0 \\ & (0,0) \leq (x_1, x_2) \leq (2,3). \end{array}$$

The best known solution is $x^* = (0.71751, 1.470)$ with an objective function value of $f^* = -16.73889$. For this problem, we apply a fourth-order RLT and the correct solution is obtained after enumerating 27 nodes in 1.06 CPU seconds. The lower bound at the initial node is -28.5. After range-reduction, the same node yields a lower bound of -17.3974. For the branch-and-bound algorithm, we set *optionl*, *optionp* and *optionn* all to zero (see Section 6.1.3).

Problem 4

(Schittkowski (1987), pp. 32)

$$\begin{array}{rl} \text{Minimize} & 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \\ \text{subject to} & (0,0) \leq (x_1, x_2) \leq (3,3). \end{array}$$

This is the so-called *Banana* function. The optimal solution is $x^* = (1,1)$ with objective function value $f^* = 0$. In addition to the usual bound-factor product RLT constraints, we add two valid cuts, $(1-x_1)^2 \geq 0$ and $(x_2-x_1^2)^2 \geq 0$. The problem is solved at the initial node with negligible CPU time.

Problem 5

(Hock and Schittkowski (1981), pp. 92)

$$\begin{aligned} \text{Minimize} \quad & x_1x_4(x_1+x_2+x_3)+x_3 \\ \text{subject to} \quad & x_1x_2x_3x_4-25 \geq 0 \\ & x_1^2+x_2^2+x_3^2+x_4^2-40=0 \\ & 1 \leq x_i \leq 5, i=1,2,3,4. \end{aligned}$$

The optimal solution is $x^* = (1, 4.743, 3.82115, 1.37941)$, having an objective function value $f^* = 17.014$. A fourth-order RLT is applied to this problem. *optionl*, *optionp* are *optionn* are set to zero. The first node yields a lower bound of 16.6846. After range-reduction, the initial lower bound becomes 16.79806. The global optimal solution is found after enumerating 5 nodes in 8.34 CPU seconds with a tolerance level of $\epsilon = 10^{-6}$.

Problem 6

(Torn and Zilinskas (1989), pp. 185)

$$\begin{aligned} \text{Minimize} \quad & x_1^2+x_2^2-\cos 18x_1-\cos 18x_2 \\ \text{subject to} \quad & (-1,-1) \leq (x_1,x_2) \leq (1,1). \end{aligned}$$

The objective function in the above problem is called the *Rastrigin* function. It is a frequently used test problem in the Russian papers on global optimization. There are about 50 local minima in the feasible region. The global optimal solution is $x^* = (0,0)$ with objective value $f^* = -2$. For this problem, we replace $\cos 18x_1$ and $\cos 18x_2$ with x_3,and x_4 respectively. Bounding polynomials of fourth-order based on Chebyshev Interpolation are generated for these two nonlinear terms. Also, we use trivially implied bounds

for the new variables as $(-1,-1) \leq (x_3, x_4) \leq (1,1)$. Notice that the LP solutions from the relaxations would also be feasible to the original problem. Thus, this problem is solved without the help of MINOS. The initial node produces a lower bound of -3. After partitioning, both of the two children nodes of the initial node produce a lower bound of -2. The feasible solution from the left child gives an upper bound of -2. Thus, the problem is solved within 3 nodes and with negligible CPU time. This amazing performance is no surprise, since for the second and third nodes, the bound-factor RLT constraints actually produce a characterization of the convex envelope of the original objective function over the convex feasible region.

Problem 7

(McCormick (1976))

$$\begin{aligned} \text{Minimize} \quad & \sin(x_3) + (x_1 - x_2)^2 - 1.5x_1 + 2.5x_2 + 1 \\ \text{subject to} \quad & x_1 + x_2 = x_3 \\ & (-1.5, -3) \leq (x_1, x_2) \leq (4, 3). \end{aligned}$$

The global optimal solution is $x^* = (-0.5472, -1.5472)$, with an objective value of $f^* = -1.9132$. For this problem, we apply a fourth-order RLT. For the non-polynomial term $\sin(x_3)$, a fourth-order bounding polynomial based on Chebyshev Interpolation is constructed. The lower bound from node zero is -12.3. However, the second and third nodes yielded much tighter lower bounds of -5.94 and -3.92. The problem is solved after enumerating 41 nodes in 18.32 CPU seconds.

Problem 8

(Hock and Schittkowski (1981), pp. 30)

$$\begin{aligned} \text{Minimize} \quad & \log(1+x_1^2) - x_2 \\ \text{subject to} \quad & (1+x_1^2)^2 + x_2^2 - 4 = 0. \end{aligned}$$

The global optimal solution is $x^* = (0, \sqrt{3})$ with an objective value of $f^* = -\sqrt{3}$. To solve this problem, first, the univariate function $\log(1+x_1^2)$ is replaced by a new variable x_3. Then, second-order bounding polynomials are constructed for this term using the Mean-Value Theorem based approach. The order of RLT applied is four. The lower bound at node zero is -2.15. Using MINOS to find upper bounds, the problem is solved after enumerating 23 nodes in 2.61 CPU seconds.

Problem 9 (Structural Sensitivity Analysis)

(Floudas and Pardalos (1990), pp. 28)

$$\begin{aligned} \text{Minimize} \quad & x_1^{0.6} + x_2^{0.6} - 6x_1 - 4x_3 + 3x_4 \\ \text{subject to} \quad & x_2 - 3x_1 - 3x_3 = 0 \\ & x_1 + 2x_3 \le 4 \\ & x_2 + 2x_4 \le 4 \\ & x_1 \le 3, x_4 \le 1, x_i \ge 0, i = 1,2,3,4. \end{aligned}$$

The best solution reported is $x^* = (4/3, 4, 0, 0)$ with an objective function value of $f^* = -4.5142$. For the concave terms $x_1^{0.6}$ and $x_2^{0.6}$, bounding polynomials are constructed using the method presented in Sherali (1998). A second-order RLT is applied. The option *optionl* $= 2$ is used. Since all the constraints are linear, the LP solutions are also feasible to the original problem, and therefore, are used to update upper bounds. Problems 9, 10 and

11 are solved without using MINOS. The LP relaxation at node zero yields a lower bound of -5.78679 and $\bar{x} = (4/3, 4, 0, 0)$. This is actually a global solution. The problem is solved after enumerating 13 nodes in 0.36 CPU seconds.

Problem 10 (Structural Sensitivity Analysis)

(Floudas and Pardalos (1990), pp. 29)

$$\begin{aligned} \text{Minimize} \quad & x_1^{0.6} + 2x_2^{0.6} + 2x_3 - 2x_2 - x_4 \\ \text{subject to} \quad & x_2 - 3x_1 - 3 = 0 \\ & x_1 + 2x_3 \leq 4 \\ & x_2 + x_4 \leq 4 \\ & x_1 \leq 3, x_4 \leq 2 \\ & x_i \geq 0, i = 1, 2, 3, 4. \end{aligned}$$

The bounding method applied and the option of generating RLT constraints used for this problem are identical to the ones used in problem 9. The best solution reported in Floudas and Pardalos (1990) is $x^* = (4/3, 4, 0, 0)$, having an objective value of $f^* = -2.07$. This solution is obviously not feasible to the above problem. Our algorithm finds a global optimal solution $x^* = (0, 3, 0, 1)$,with $f^* = -3.13364$ for this problem after enumerating 17 nodes in 0.68 second. The solution prescribed in Floudas and Pardalos (1990) is actually an optimal solution to a problem identical to the above one, except that the first constraint is replaced by $x_2 - 3x_1 = 0$. For this case, the optimal objective value is -2.2168.

Problem 11 (Structural Sensitivity Analysis)

(Floudas and Pardalos (1990), pp. 29)

$$
\begin{aligned}
\text{Minimize} \quad & x_1^{0.6} + x_2^{0.6} + x_3^{0.4} + 2x_4 + 5x_5 - 4x_3 - x_6 \\
\text{subject to} \quad & x_2 - 3x_1 - 3x_4 = 0 \\
& x_3 - 2x_2 - 2x_5 = 0 \\
& 4x_4 - x_6 = 0 \\
& x_1 + 2x_4 \leq 4 \\
& x_2 + x_5 \leq 4 \\
& x_3 + x_6 \leq 6 \\
& (x_1, x_5, x_3) \leq (3, 2, 4) \\
& x_i \geq 0, \quad i = 1, 2, \ldots, 6.
\end{aligned}
$$

The best solution reported is $x^* = (0.67, 2, 4, 0, 0, 0)$, having an objective value of $f^* = -11.96$. Our algorithm finds a better solution $x^* = (0.16667, 2, 4, 0.5, 0, 2)$, with $f^* = -13.4019$. The initial node produces a lower bound of -14.00. The LP solution at node-zero is actually a global optimal solution. The branch-and-bound process generates a total of 67 nodes and the problem is solved to optimality in 4.14 seconds CPU time.

Problem 12

(Floudas and Pardalos (1990), pp. 23)

$$\begin{aligned}
\text{Minimize} \quad & 37.293239x_1 + 0.8356891x_1x_5 + 5.3578547x_3^2 - 40792.141 \\
\text{subject to} \quad & -0.0022053x_3x_5 + 0.0056858x_2x_5 + 0.0006262x_1x_4 - 6.665593 \leq 0 \\
& 0.0022053x_3x_5 - 0.0056858x_2x_5 - 0.0006262x_1x_4 - 85.334407 \leq 0 \\
& 0.0071317x_2x_5 + 0.0021813x_3^2 + 0.0029955x_1x_2 - 29.48751 \leq 0 \\
& -0.0071317x_2x_5 - 0.0021813x_3^2 - 0.0029955x_1x_2 + 9.48751 \leq 0 \\
& 0.0047026x_3x_5 + 0.0019085x_3x_4 + 0.0012547x_1x_3 - 15.699039 \leq 0 \\
& -0.0047026x_3x_5 - 0.0019085x_3x_4 - 0.0012547x_1x_3 + 10.699039 \leq 0 \\
& 78 \leq x_1 \leq 102, 33 \leq x_2 \leq 45, 27 \leq x_4 \leq 45 \\
& 27 \leq x_4 \leq 45, 27 \leq x_5 \leq 45.
\end{aligned}$$

The objective function and the constraints are defined by nonconvex quadratic terms. The best known solution is $x^* = (78, 33, 29.9953, 45, 36.7758)$, with an objective value of $f^* = -30665.5387$. In Tuncbilek (1994), a RLT based convex relaxation is used at the bounding step to solve this problem. The second-order convex RLT lower bound at node zero is -30765.086. We apply a third-order RLT (with $optionp = 1$) and obtain a better lower bound of -30673.04. Furthermore, the solution $x = (78, 33, 29.9878, 45, 36.7792)$ obtained for the LP relaxation is in a relatively close vicinity of an actual global optimal solution. This enables MINOS to locate a global solution at node zero. The branch-and-bound algorithm terminates successfully after enumerating 3 nodes.

Problem 13 (Flywheel Design)

(Siddall (1972))

$$\begin{aligned} \text{Minimize} \quad & -0.0201x_1^4x_2x_3^2/10^7 \\ \text{subject to} \quad & x_1^2x_2 \leq 675 \\ & x_1^2x_3^2 \leq 4190000 \\ & 0 \leq x_1 \leq 36, 0 \leq x_2 \leq 5, 0 \leq x_3 \leq 125. \end{aligned}$$

In Tuncbilek (1994), a global solution $x^* = (16.51, 2.477, 124)$ having $f^* = -5.6848$ is reported. Our algorithm applies a seventh-order RLT (with $optionp = 1$) and finds an alternative solution of $x^* = (35.8068, 0.526468, 50.71664)$. The problem is solved at the initial node.

Problem 14 (Haverly's Pooling Problem)

(Floudas and Pardalos (1990), pp. 61)

$$\begin{aligned} \text{Minimize} \quad & -9x_1 - 15x_2 + 6x_3 + 13x_4 + 10x_5 + 10x_6 \\ \text{subject to} \quad & x_7 + x_8 - x_3 - x_4 = 0 \\ & x_1 - x_7 - x_5 = 0 \\ & x_2 - x_8 - x_6 = 0 \\ & x_7x_9 + x_8x_9 - 3x_3 - x_4 = 0 \\ & x_7x_9 + 2x_5 - 2.5x_1 \leq 0 \\ & x_8x_9 + 2x_6 - 1.5x_2 \leq 0 \\ & x_1 \leq 100, x_2 \leq 200. \end{aligned}$$

The best known solution for this problem is $x^* = (0, 200, 50, 150, 0, 0, 0, 200, 1.5)$ with an objective value of -750. Applying a third-order RLT (with $optionp =$

1), the solution from the LP relaxation at the first node is feasible to the original problem. It produces an upper bound that equals the lower bound. Thus, the problem is solved after enumerating only one node.

Problem 15

(Dravnieks and Chinneck (1997))

$$\begin{aligned} \text{Minimize} \quad & 4x_1^2 - 2.1x_1^4 + x_1^6/3 + x_1x_2 - 4x_2^2 + 4x_2^4 \\ \text{subject to} \quad & -2x_1^4 + 2 - x_2 \geq 0 \\ & (x_2+2)^2 - x - x_1^2 \geq 0 \\ & \cos(x_1+1) + (x_1-1)^2 - x_2 - 6 \leq 6 \\ & -1 \leq x_1 \leq 4, -10 \leq x_2 \leq 10. \end{aligned}$$

The global optimal solution for this problem is $x^* = (-0.0898420, 0.712656)$ with an objective value of $f^* = -1.031628$. To solve this problem, we apply a sixth-order RLT. Eight valid nonnegativity restrictions of the following type are added

$$x_1^i x_2^j \geq 0, i, j, \text{ even}$$

to tighten the lower bounds. For the non-polynomial term $\cos(x_1 + 1)$, second-order bounding polynomials based on the Mean-Value Theorem are constructed. When generating RLT constraints, we set both *optionp* and *optionn* to 1. The lower bounds for the first three nodes are, respectively, -81.08, -16.96 and -3.1485. The problem is solved using 283 CPU seconds. The relatively greater computational time is due to the presence of high order terms and a large number of constraints, which results in large LP subproblems. An appropriate application of constraint-filtering techniques could be expected to improve the performance in this case.

Chapter 8

Conclusions and Recommendations for Future Research

In this thesis, we have explored a global optimization method for solving nonconvex factorable programming problems. Such problems abound in many domains of natural sciences, engineering, operations research, economics and social sciences. Traditional convex optimization algorithms are not adequate for solving these nonlinear problems due to the presence of multiple local optima and the lack of local criteria for deciding whether a local solution is globally optimal or not. From the complexity point of view, global optimization to nonconvex factorable programming is an NP-hard problem. Despite recent advances in developing new algorithms, there is still no prevailing method for this very general class of problems.

Sherali and Adams developed a *Reformulation-Linearization Technique*

(RLT), a ground-breaking approach in mathematical programming, which has been successfully applied in solving various combinatorial optimization problems. Later, Sherali and Tuncbilek extended this approach to solve the more general family of continuous, nonlinear polynomial programs. The research in this thesis is an important generalization of the previous research efforts.

Our proposed algorithm is also a branch-and-bound algorithm, but one that employs two levels of approximations, carefully crafted to yield a strong bounding mechanism. At the first level, the original nonconvex factorable functions are approximated by appropriately constructed nonconvex bounding polynomials. An LP relaxation is then generated for the resulting polynomial program via the application of RLT. Suitable partitioning rules are accordingly developed to drive the errors in these two levels of approximation simultaneously to zero, and therefore ensure the convergence of the process to a global optimum. In generating the LP relaxation, RLT is directly applied to the approximating polynomial program, without any intermediate transformation. For the first level approximation, we provide methods based on the Mean-Value Theorem and Chebyshev Interpolation, which are easy to implement and have empirically been shown to be effective. General assumptions, transformation strategies, and implementation guidelines are provided for each component of the algorithm. Based on these guidelines, users can develop their own versions of these components, and add other special techniques such as Lagrangian relaxation, range-reduction, and constraint filtering into the general algorithm. Therefore, this algorithm serves as a theoretical framework within which users can customize their applications to

take advantage of special structures that a particular problem may possess.

The software developed is implemented entirely in C++. It is fine-tuned to reflect the Objected-Oriented design and to maintain a good balance between efficiency and robustness. The class modules developed for this algorithm can be easily extended and adapted into future RLT engines. The resulting product is a comprehensive collection of data types, algorithms, and program modules that constitutes a unified environment within which the design, implementation, and testing of RLT based algorithms can be readily facilitated.

Finally, the algorithm is tested on a set of problems from various sources in the literature. The computational experience has been very competitive and promising.

8.1 Issues for Future Research

As mentioned in the beginning of this chapter, the primary focus of the research in this thesis is to develop the theoretical aspects of an algorithm for solving nonconvex factorable programming problems. There are many detailed issues that could be pursued in future research efforts to compose an efficient variant of this algorithm.

Sherali and Tuncbilek (1997) analyze the effectiveness of the inclusion of various additional valid RLT constraints in the bounding step. Special types of RLT constraints such as squared grid factor products, and squared Lagrangian interpolation polynomial based constraints can be incorporated in the initial bounding step to strengthen the tightness of the lower bounds.

Furthermore, instead of forming an LP relaxation, we can alternatively apply the *Reformulation-Convexification Technique* studied in Sherali and Tuncbilek (1995). At the bounding stage, a convex program is formulated by linearizing only the nonconvex terms and leaving the convex terms intact. Furthermore, other valid convex cuts and underestimators can be added to tighten the relaxation. By solving convex subproblems, we may potentially derive stronger lower bounds, although having to contend with convex, rather than simply linear relaxations. Computational experiments can be conducted to measure the tradeoff between these two approaches.

Constraint filtering schemes discussed in Sherali and Tuncbilek (1995) and Tuncbilek (1994) can be applied in conjunction with the basic framework of our algorithm. Computational improvements are expected. Ideas exemplified in Sherali, Smith and Adams (1997) prompt similar investigations in the continuous problem domain as well.

For the current implementation, we have employed the simplest form of a range-reduction strategy that is conducted at the initial node of the branch-and-bound tree. More general and more complex range-reduction techniques examined in Sherali and Tuncbilek (1995) and Ryoo and Sahinidis (1996) can be potentially embodied into our algorithm.

In Sherali, Adams and Driscoll (1996), a new and extended hierarchy of RLT relaxation is presented that provides a unifying framework for constructing a range of continuous relaxations for mixed 0-1 integer programming problems. The ideas discussed in that paper could also be exploited in the context of continuous global optimization. In particular, RLT constraints that imply simple bound-factors can be used to produce tighter LP or convex

relaxations. New discoveries in this area will enrich the theory of RLT and lead to more general and effective branching and partitioning rules for future branch-and-bound algorithms.

Bibliography

[1] Adjiman, C. S. and C. A. Floudas (1996), Rigorous Convex Underestimators for General Twice-Differentiable Problems, *Journal of Global Optimization* **9**, pp. 23-40.

[2] Al-Khayyal, F. A. J. E. Falk (1983), Jointly Constrained Biconvex Programming, *Mathematics of Operations Research* **8**, pp.273-286.

[3] Al-Khayyal, F. A., C. Larson, and T. Van Voorhis (1994), A Relaxation Method for Nonconvex Quadratically Constrained Quadratic Programs, Working Paper, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332.

[4] Audet, C., P. Hansen, B. Jaumard and G. Savard (1998), A Branch and Cut Algorithm for Nonconvex Quadratically Constrained Quadratic Programming, Working Paper, Ecole Polytechnique de Montreal.

[5] Bromberg, M. and T. Chang (1992), One Dimensional Global Optimization Using Linear Lower Bounds, in: C. A. Floudas and P. M. Pardalos (Eds.), *Recent Advances in Global Optimization*, Princeton University Press, pp. 200-220.

[6] Byrne, R. P. and I. D. Bogle (1995), Solving Nonconvex Process Optimization Problems Using Interval Subdivision Algorithms, in: I. E. Grossmann (Ed.), *Global Optimization in Engineering Design*, Kluwer Academic Publishers, pp. 155-174.

[7] Dalquist, G. and A. Björck (1974), *Numerical Methods*, Prentice-Hall, Englewood Cliffs, New Jersey.

[8] Davis, P. J. (1975), *Interpolation and Approximation*, Dover Publications, New York.

[9] Dembo, R. S. (1976), A Set of Geometric Programming Test Problems and their Solutions, *Mathematical Programming* **10**, pp. 192-213.

[10] Dembo, R. S. (1978), Current State of Art of Algorithms and Computer Software for Geometric Programming, *Journal of Optimization Theory and Applications* **26**, pp. 149-183.

[11] Dravnieks, E. W. and J. W. Chinneck (1997), Formulation Assistance for Global Optimization Problems, *Computers Operations Research* **24**, pp. 1151-1168.

[12] Epperly, T. G. and E. N. Pistikopoulos (1997), A Reduced Space Branch and Bound Algorithm for Global Optimization, *Journal of Global Optimization* **11**, pp. 287-311.

[13] Floudas, C. A. and P. M. Pardalos (1990), *A Collection of Test Problems for Constrained Global Optimization Algorithms*, Springer-Verlag, Berlin.

[14] Floudas, C. A. and V. Visweswaran (1990), A Global Optimization Algorithm (GOP) for Certain Classes of Nonconvex NLP's I. *Computers and Chemical Engineering,* **14**, pp. 1397-1417.

[15] Floudas, C. A. and P. M. Pardalos (1992) (Eds.), *Recent Advances in Global Optimization*, Princeton University Press.

[16] Floudas, C. A. and V. Visweswaran (1995), Quadratic Optimization, in: R. Horst and P. M. Pardalos (Eds.), *Handbook of Global Optimization, Nonconvex Optimization and its Applications*, Kluwer Academic Publishers, pp. 217-270.

[17] Haddad, E. (1996), Nonconvex Global Optimization of the Separable Resource Allocation Problem with Continuous Variables, in: C.A. Floudas and P.M. Pardalos (Eds.), *State of the Art in Global Optimization, Computational Methods and Applications*, Kluwer Academic Publishers, pp. 383-393.

[18] Hock, W. and K. Schittkowski (1981), *Test Examples for Nonlinear Programming Codes*, Springer-Verlag, Berlin.

[19] Horst, H. and H. Tuy (1996), *Global Optimization, Deterministic Approach*, Springer-Verlag, Berlin.

[20] Horst, H. and N. Thoai (1996), Global Minimization of Separable Concave Functions under Linear Constraints with Totally Unimodular Matrices, in: C. A. Floudas and P. M. Parlalos (Eds.), *State of Art in Global Optimization*, Kluwer Academic Publishers, pp. 35-45.

[21] Horst, R., P. M. Pardalos and N. V. Thoai (1995), *Introduction to Global Optimization*, Kluwer Academic Publishers, Netherlands.

[22] Isaacson, E., and H. B. Keller (1966), *Analysis of Numerical Methods*, John Wiley & Sons, New York.

[23] Knuth, D. E. (1973), *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, MA..

[24] Konno, H. and T. Kuno (1995), Multiplicative Programming Problems, in: R. Horst and P. M. Pardalos (Eds.), *Handbook of Global Optimization, Nonconvex Optimization and its Applications*, Kluwer Academic Publishers, pp. 369-405.

[25] Lamar, B. W. (1995), Nonconvex Optimization over a Polytope Using Generalized Capacity Improvement, *Journal of Global Optimization* **7**, pp. 127-142.

[26] McCormick, G. P. (1976), Computability of Global Solutions to Factorable Nonconvex Programs: Part I - Convex Underestimating Problems, *Mathematical Programming* **10**, pp. 147-175.

[27] Pintér, J. D. (1996), *Global Optimization in Action, Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications*, Kluwer Academic Publishers.

[28] Press, W. H, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery (1992), *Numerical Recipes in C*, Cambridge University Press, Cambridge.

[29] Ryoo, H. S. and N. V. Sahinidis (1996), A Branch-and-Reduce Approach to Global Optimization, *Journal of Global Optimization* **8**, pp. 107-138.

[30] Schittkowski, K. (1987), *More Test Examples for Nonlinear Programming Codes*, Springer-Verlag, Berlin.

[31] Shectman, J. P. and N. V. Sahihidis (1996), A Finite Algorithm for Global Minimization of Separable Concave Programs, in: C.A. Floudas and P.M. Pardalos (Eds.), *State of the Art in Global Optimization, Computational Methods and Applications*, Kluwer Academic Publishers, pp. 303-338.

[32] Sherali, H. D. and D. C. Myers (1985), The Design of Branch and Bound Algorithms for a Class of Nonlinear Integer Programs, *Annals of Operations Research* **5**, pp. 463-484.

[33] Sherali, H. D. and W. P. Adams (1990), A Hierarchy of Relaxations between the Continuous and Convex Hull Representations for Zero-One Programming Problems, *SIAM Journal on Discrete Mathematics* **3**, 411-430.

[34] Sherali, H. D. and A. Alameddine (1990), An Explicit Characterization of the Convex Envelope of a Bivariate Bilinear Function over Special Polytopes, *Annals of Operations Research* **25**, pp. 197-210.

[35] Sherali, H. D. and A. Alameddine (1991), A New Reformulation-Linearization Technique for Bilinear Programming Problems, *Journal of Global Optimization* **2**, pp. 379-410.

[36] Sherali, H. D. and C. H. Tuncbilek (1992), A Global Optimization Algorithm for Polynomial Programming Problems Using a Reformulation-Linearization Technique, *Journal of Global Optimization* **2**, pp. 101-112.

[37] Sherali, H. D. and W. P. Adams (1994), A Hierarchy of Relaxations and Convex Hull Characterizations for Mixed-Integer Zero-One Programming Problems, *Discrete Applied Mathematics* **52**, pp. 83-106.

[38] Sherali, H. D. and C. H. Tuncbilek (1995), A Reformulation-Convexification Approach for Solving Nonconvex Quadratic Programming Problems, *Journal of Global Optimization* **7**, pp. 1-31.

[39] Sherali, H. D., W. P. Adams and P. J. Driscoll (1996), Exploiting Special Structures in Constructing a Hierarchy of Relaxations for 0-1 Mixed Integer Problems, *Operations Research*, to appear.

[40] Sherali, H. D. and W. P. Adams (1996), Computational Advances in Using the Reformulation-Linearization Technique (RLT) to Solve Discrete and Continuous Nonconvex Programming Problems, *OPTIMA* **49**, pp. 1-7.

[41] Sherali, H. D. and C. H. Tuncbilek (1997), New Reformulation-Linearization/Convexification Relaxations for Univariate and Multivariate Polynomial Programming Problems, *Operations Research Letters* **21**, pp. 1-9.

[42] Sherali, H. D. (1997), Convex Envelopes of Multilinear Functions over a Unit Hypercubes and over Special Discrete Sets, *Acta Mathematica Vietnamica* **22**, pp. 245-270.

[43] Sherali, H. D., J. C. Smith and W. P. Adams (1997), Reduced First-Level Representation via the Reformulation-Linearization Technique: Results, Counter-Examples, and Computations, Working Paper, Department of ISE, Virginia Tech.

[44] Sherali, H. D. (1998), Global Optimization of Nonconvex Polynomial Programming Problems Having Rational Exponents, *Journal of Global Optimization* **12**, pp. 267-283.

[45] Siddall, J. N. (1972), *Analytical Decision Making in Engineering Design*, Prentice-Hall, Englewood Cliffs, N.J.

[46] Stephanopoulos, G. and A. W. Westerburg (1975), The Use of Hestenes's Method of Multipliers to Resolve Dual Gaps in Engineering System Optimization, *Journal of Optimization Theory and Applications* **15**, pp. 285-309.

[47] Stoer, J. and R. Burlirsch (1993), *Introduction to Numerical Analysis*, Springer-Verlag, Berlin.

[48] Torn, A. and A. Zilinskas (1989), *Global Optimization*, Spring-Verlag, Berlin.

[49] Tuncbilek, C. (1994), *Polynomial and Indefinite Quadratic Programming Problems: Algorithms and Applications*, Ph.D Dissertation, Department of ISE, Virginia Tech.

[50] Tuy, H. (1995), D. C. Optimization: Theory, Methods and Algorithms, in: R. Horst and P. M. Pardalos (Eds.), *Handbook of Global Optimiza-*

tion, Nonconvex Optimization and its Applications, Kluwer Academic Publishers, pp. 149-216.

[51] Ueberhuber, C. (1997), *Numerical Computation I, Methods, Software, and Analysis*, Springer-Verlag, Berlin.

[52] Vavasis, S. (1995), Complexity Issues in Global Optimization: A Survey, in: R. Horst and P.M. Pardalos (Eds.), *Handbook of Global Optimization, Nonconvex Optimization and its Applications*, Kluwer Academic Publishers, pp. 1-40.

[53] Visweswaran, V., and C. A. Floudas, Unconstrained and Constrained Global Optimization of Polynomial Functions in One Variable (1992), *Journal of Global Optimization* **2**, pp. 73-99.

[54] Volkov, E. A. (1990), *Numerical Methods*, Hemisphere Publishing, New York.

[55] Voorhis, T. V. and F. Al-Khayyal (1996), Accelerating Convergence of Branch-and-Bound Algorithms For Quadratically Constrained Optimization Problems, in: C.A. Floudas and P.M. Pardalos (Eds.), *State of Art in Global Optimization*, Kluwer Academic Publishers, pp. 267-284.

[56] Wolf, D. D. and Y. Smeers (1995), Optimal Dimensioning of Pipe Networks with Application to Gas Transmission Networks, *Operations Research* **44**, pp. 596-607.

Appendix

A Tutorial on the Branch-and-Bound Code

In this appendix, we present a tutorial on how to use the developed program to solve nonconvex factorable programming problems. As mentioned in Chapter 6, the *BranchBound* class is the only program module that a user has to work with. We will study the functions in this class in detail by working through an example. First, the header file of the *BranchBound* class is listed below.

```
#ifndef _bbound_
#define _bbound_
#include "bbnode.h"
#include "odlist.h"
#include "PXSWdata.h"
#include "problem.h"

class BranchBound{
```

```
private:
   int n;
   int order;
   double *bestsol;
   double bestsolvalue;
   double *l;
   double *u;
   int psizes[4];
   int loption, poption, noption;
   void SetIndex(BBNode node);
   int Fathom(BBNode node, double e);
   int objorder;
   int Bounding(BBNode & node, int flag, int index);
public:
   BranchBound(int nv, int orderRLT,
               double *lb, double *ub, int *size);
   void SetBestSolution(double *x, double v);
   void SetOptions(int op1,int op2, int op3);
   void SetObjOrder(int order);
   void BranchingVars(int i);
   void SolveInitialNode(int flag);
   void RangeReduction(void);
   void Enumerate(int iter, int rdtion, double eps);
};
#endif
```

We stress the fact that the problem is assumed to have been pre-processed as described in Section 6.1.1.

For each problem, a user needs to provide three modules. The first module is a header file named **PXSWdata.h**. It is imperative that the exact same name is used. This header file provides the input data of the problem. The following is a template for such a file.

```
#include "problem.h"
#include "cplexcpp.h"
#include <assert.h>
#include <iostream.h>

double PXSWocoefs[];
long PXSWosubs[];
int PXSWoterm;
// the above two constitutes the objective function
*******************************************
long PXSWlsubs[];
double PXSWlcoefs[];
int  PXSWlterms[];   // linear constraints
******************************************
long PXSWpsubs[];
double  PXSWpcoefs[];
int PXSWpterms[];     // polynomial constraints
```

```
****************************************
long PXSWesubs[];
double PXSWecoefs[];
int PXSWeterms[];    // equalities
****************************************
long PXSWnsubs[];
double PXSWncoefs[];
int PXSWnterms[];
// bounding polynomials
```

It is essential that all the variables are present in this file. Since they are global variables, the user should avoid using the same names in other parts of the program.

Each problem consists of an objective function and a set of constraints. The constraints are classified into four categories: linear, polynomial, equality, and general non-polynomial constraints. Each type of constraints is described by three variables. For example, PXSWlsubs[] stores the subscripts of all the terms of linear constraints. PXSWlcoefs[] stores the corresponding coefficients of these terms. PXSWlterms[] gives the number of terms in each constraint. The polynomial, equality, and non-polynomial constraints are specified in a similar fashion. For the objective function, since there is only one objective function, PXSWoterms is an integer, not an array. All the arrays are declared and initialized in PXSWdata.h with the exception of PXSWncoefs[]. This array is allocated for the coefficients of the bounding polynomials for the non-polynomial functions. Since these coefficients

change from node to node, they are determined during the run-time via a user supplied routine. However, the array still needs to be declared.

The second module that a user needs to provide is a routine to update the coefficients of the bounding polynomials. The following is a prototype of this routine.

```
/* routine to update the coefficients */

void UpdateBoundingPoly(double *l, double *u, double *coefs);
```

During the program execution, the array PXSWncoefs[] is passed into this routine to be updated at each node. The first two parameters in the above routine, i.e., l and u, are the lower and upper bounds for the original variables. Since they are passed as pointers, the user can change the content of l and u in the routine. This is especially useful in a situation where a certain variable x_k is defined in terms of other variables x_i, and x_j. When the bounds on x_i and x_j are revised for a new node, the user can update the bounds for x_k in the routine to potentially give a tighter RLT relaxation. Notice that coefs is just a formal parameter for the actual PXSWncoefs[] in PXSWdata.h. In implementing this routine, it is the user's responsibility to update this parameter in a way that is consistent with the way in which PXSWncoefs[] would be updated.

The third module that the user needs to supply is a function named UPPERBOUND. This routine is used to search for an incumbent solution. A prototype is given below.

```
double UPPERBOUND(double *x, int flag);
```

Here, x is the initial solution which will be overwritten by the final solution. The function value is returned. If the variable flag is set to 1, then, the initial solution is used as a warm start.

The *BranchBound* class provides a set of public member functions that are very straightforward to use. We discuss each of them in the sequel.

```
BranchBound(int nv, int orderRLT,
            double *lb, double *ub, int *size);
```

This is the constructor of the *BranchBound* class. It takes five parameters: the number of variables, the order of the RLT , the corresponding lower and upper bounds on the original variables, and the size of the problem. The size of the problem is given by an array of four integers, which respectively give the number of each type of constraints. These numbers should be consistent with the length of the arrays that are used to describe these constraints in the header file PXSWdata.h.

```
void SetBestSolution(double *x, double v);
```

This member function allows the user to optionally specify an advanced solution. If the user decides to perform a range-reduction, it is usually a good idea to set a good solution by calling this member function.

```
void SetOptions(int op1,int op2, int op3);
```

By using this member function, the user can control the RLT constraint generation process. Each of these three arguments takes the value $0, 1$, and 2. The meanings of these options are discussed in Section 6.1.3.

```
void SetObjOrder(int order){objorder=order;};
```

The user must input the order of the objective function by calling this member function.

```
void SolveInitialNode(int flag);
```

Upon being called, this member function will solve the initial node subproblem. The flag variable will signal whether a range-reduction should be performed first.

```
void RangeReduction(int i);
```

This member function performs a range-reduction on the variable x_i.

```
void BranchingVars(int i);
```

Suppose that $x_1, x_2, \ldots x_k$ are the variables present in the original problem. After the initial pre-processing, we add $x_{k+1}, \ldots x_n$ to replace the non-polynomial terms. All of the variables $x_1, x_2, \ldots x_n$ are considered "original" variables in the transformed problem. However, we may decide not to use $x_{k+1}, \ldots x_n$ as branching variables since they are defined in terms of the functions of the first k variables. Furthermore, among the first k variables, the user may choose to branch only on a subset of these variables. This member function sets the first i variables as branching variables.

```
void Enumerate(int iter, int rd, double eps)
```

This is the principal function in this class that executes the branch-and-bound process. The user may control the number of nodes the process generates before it terminates, and the tolerance by which a node is fathomed. Also, setting rd=1 indicates the execution of range-reduction at every five nodes.

Finally, we present an example. Consider the following problem.

$$\begin{aligned} \text{Minimize} \quad & \ln(1+x_1^2) - x_2 \\ \text{subject to} \quad & (1+x_1^2)^2 + x_2^2 - 4 = 0 \\ & (0,0) \le (x_1, x_2) \le (10,10). \end{aligned}$$

First, we substitute $\ln(1+x_1^2)$ with a new variable x_3. At any node, for $x_1 \in [a,b]$, x_3 is bounded by $[\ln(1+a^2), \ln(1+b^2)]$ since $\ln(z)$ is a monotone increasing function. Thus, the problem becomes to

$$\begin{array}{rl} \text{Minimize} & x_3 - x_2 \\ \text{subject to} & (1+x_1^2)^2 + x_2^2 - 4 = 0 \\ & x_3 - \ln(1+x_1^2) \geq 0 \\ & -x_3 + \ln(1+x_1^2) \geq 0 \\ & (0,0) \leq (x_1,x_2) \leq (10,10), 0 \leq x_3 \leq \ln(101). \end{array}$$

The second step is to set up the PXSWdata.h file.

```
#include "problem.h"
#include "cplexcpp.h"
#include <assert.h>
#include <iostream.h>

double PXSWocoefs[2]={1,1};
long PXSWosubs[2]={3,2};
int PXSWoterm=2;
//***************************

long PXSWlsubs[];
double PXSWlcoefs[];
int  PXSWlterms[];
//***************************
```

```
long PXSWpsubs[];
double  PXSWpcoefs[];
int PXSWpterms[];
long PXSWesubs[];

//*************************
long PXSWesubs[4]={0,1111,11,22};
int PXSWeterms[2]={4,0};
double PXSWecoefs[4]={-3,1,2,1};
//*************************
long PXSWnsubs[12]={0,1,11,111,1111,3,0,1,11,111,1111,3};
double PXSWncoefs[12];
int PXSWnterms[2]={6,6};
```

In this case, we construct fourth-order bounding polynomials to approximate the term $\ln(x_1^2+1)$. These bounding polynomials take the form

$$a_1x_1^4 + a_2x_1^3 + a_3x_1^2 + a_4x_1 + a_5 - x3 \geq 0$$
$$-b_1x_1^4 - b_2x_1^3 + b_3x_1^2 - b_4x_1 - b_5 + x_3 \geq 0.$$

There are a total of 12 terms. The subscripts of these terms are stored in the array PXSWnsubs[12]. The corresponding coefficients are to be determined by the routine UpdateBoundingPoly. There is no fixed order to range the terms in a constraint. However, the subscripts in PXSWnsubs[] should be in harmony with the coefficients in PXSWncoefs[].

The third step is to provide routines to update the coefficients of the bounding polynomials and to find feasible solutions. For this example, the bounding polynomials are built using the Chebyshev Interpolation method described in Chapter 4. The search for a feasible solution is accomplished by calling MINOS.

```
double fn(double a, double b, double x, int flag)
/* a function needed for using the Chebyshev interpolation */
{
   if (flag==0)
      return log(1+x*x);
   if ( flag==1)
      return log(1+a*a);
   if (flag==2)
      return log(1+b*b);
   if (flag==3)
      return
void UpdateBoundingPoly(double *l, double *u, double *coefs)
{
   double a=l[0];
   double b=u[0];  //a <=x1<=b
   ChebshevApp pxx(fn,4); // 4th order
   pxx.AppX(a,b,-1,coefs);
   pxx.AppX(a,b,1,coefs+6); // fill in coefs, 12 elements
   l[2]=log(1+a*a);  // update bounds for x3
```

```
   u[2]=log(1+b*b);
 }
 /* External subroutines to be called */



/* Results files subroutine */
extern "C" void mispec_ (int*, int*, int*, int*, int*);

/* Minoss */
extern "C" void minoss_(char*,int*,int*,int*,int*,int*,int*,
                       int*,int*,int*,double*,
           char*,double*,int*,int*,double*,
                       double*,int*,int*,int*,
           double*,double*,double*,int*,int*,
                       int*,int*,double*,double*,
                 double*,int*);



double UPPERBOUND(double *x, int flag)
{
  /*  int nwcore_ptr=3000;
    char* start1="Cold";
    char* start2="Warm";
    char names[40];
    int m=1;
```

```
int n=2;
int nb=m+n;
int ne=2;
int one=1;
int zero=0;
double dzero=0.0;
int nncon=1;
int nnobj=2;
int nnjac=2;
int iobj=0;
double objadd=0;
double xn[3];
int inform, ns, ninf, mincor;
double rc[6];
double obj,  sinf;
double z[nwcore];
int ispecs, iprint, isumm ;
int i;
int ha[2];
int ka[3];
int hs[6];
double pi[1];
double bl[3];
double bu[3];
double a[2];
```

```
inform=0;
for(i=0;i<n;i++)
  xn[i]=x[i];
pi[0]=0;
    ispecs=4;
    iprint=0;
    isumm=0;
ha[0]=ha[1]=1;
ka[0]=1;
ka[1]=2;
ka[2]=3;
bl[0]=0;
bl[1]=0;
bu[0]=5;
bu[1]=5;
bl[2]=0;
bu[2]=0;
for(i=0;i<n;i++)
  hs[i]=0;
for(i=n;i<nb;i++)
  hs[i]=0;
mispec_( &ispecs, &iprint, &isumm, &nwcore_ptr,&inform);
if(flag==0)
   minoss_ (start1,&m,&n,&nb,&ne,
      &one,&nncon,&nnobj,&nnjac,&zero,&dzero,
```

```
          names,a,ha,ka,bl,bu,&zero,&zero,hs,xn,pi,
           rc,&inform,&mincor,&ns,&ninf,&sinf,&obj,
                z,&nwcore_ptr);
    else
       minoss_ (start2,&m,&n,&nb,&ne,
         &one,&nncon,&nnobj,&nnjac,&zero,&dzero,
         names,a,ha,ka,bl,bu,&zero,&zero,hs,xn,pi,
         rc,&inform,&mincor,&ns,&ninf,&sinf,&obj,
              z,&nwcore_ptr);
      return obj;
}
```

Finally, a main function is set up to use the member functions from BranchBound class to solve this problem.

```
int main()
{
   int s[4]={0,0,0,1}; // one equality constraint
   double xl[3]={0,0,0};
   double xu[3]={10,10,10};
   double x[9]={0,0,0};
   double v=0;
   BranchBound bb(3,4,xl,xu,s);
   bb.SetBestSolution(x,v);
   bb.SetObjOrder(1);
```

```
  bb.SetOptions(0,0,0);
  bb.Printout();
  bb.Enumerate(100,0,0.000001);
}
```

We point out that the program is flexible enough to allow users to add additional cuts at each node. Notice that all the original constraints are given in the file PXSWdata.h. Some of the arrays in the PXSWdata.h are updated at each node to derive new constraints. To facilitate the inclusion of other polynomial constraints, the user can properly change the length of the arrays PXSWpcoefs[], PXSWpsubs[] and PXSWpterms[]. The values of these arrays can be dynamically updated in another user supplied routine.

Vita

Hongjie Wang was born in Shanghai, China on January 17, 1971. He graduated from Liberty University in Lynchburg, Virginia where he received a Bachelor of Science degree in Mathematics in 1995. He attended Virginia Polytechnic Institute and State University where he received a Master of Science degree in Industrial and Systems Engineering in May 1998.