

**A Subgradient Algorithm for  
Nonlinear Integer Programming and  
Its Parallel Implementation**

*Zhiyun Wu*

**CRPC-TR91153  
May, 1991**

Center for Research on Parallel Com  
Rice University  
P.O. Box 1892  
Houston, TX 77251-1892



RICE UNIVERSITY

A SUBGRADIENT ALGORITHM FOR NONLINEAR  
INTEGER PROGRAMMING AND ITS PARALLEL  
IMPLEMENTATION

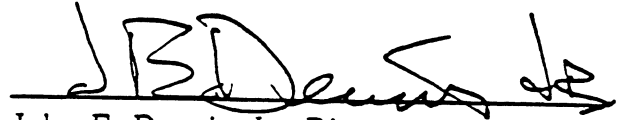
by

ZHIJUN WU

A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE

DOCTOR OF PHILOSOPHY

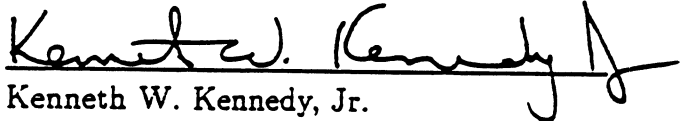
APPROVED, THESIS COMMITTEE:



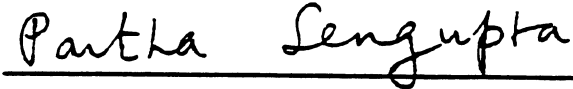
John E. Dennis, Jr., Director  
Noah Harding Professor of Mathematical  
Sciences



Robert E. Bixby, Co-Director  
Professor of Mathematical Sciences



Kenneth W. Kennedy, Jr.  
Noah Harding Professor of Computer  
Science



Partha Sengupta  
Staff Research Engineer  
Shell Development Company

Houston, Texas

May, 1991



# A SUBGRADIENT ALGORITHM FOR NONLINEAR INTEGER PROGRAMMING AND ITS PARALLEL IMPLEMENTATION

ZHIJUN WU

## ABSTRACT

This work concerns efficiently solving a class of nonlinear integer programming problems:  $\min \{f(x) : x \in \{0, 1\}^n\}$  where  $f(x)$  is a general nonlinear function. The notion of subgradient for the objective function is introduced. A necessary and sufficient condition for the optimal solution is constructed. And a new algorithm, called the subgradient algorithm, is developed. The algorithm is an iterative procedure searching for the solution iteratively among feasible points, and in each iteration generating the next iterative point by solving the problem for a local piecewise linear model of the original problem which is constructed with supporting planes for the objective function at a set of feasible points. Special continuous optimization techniques are used to compute the supporting planes. The problem for each local piecewise linear model is solved by solving an equivalent linear integer program. The fundamental theory for the new approach is built and all related mathematical proofs and derivations such as proofs for convergence properties, the finiteness of the algorithm, as well as the correct formulation of the subproblems are presented. The algorithm is parallelized and implemented on parallel distributed-memory machines.

The preliminary numerical results show that the algorithm can solve test problem effectively.

To implement the subgradient algorithm, a parallel software system written in EXPRESS C is developed. The system contains a group of parallel subroutines that can be used for either continuous or discrete optimization such as subroutines for  $QR$ ,  $LU$  and Cholesky factorizations, triangular system solvers and so on. A sequential implementation of the simplex algorithm for linear programming also is included. Especially, a parallel branch-and-bound procedure is developed. Different from directly parallelizing the sequential binary branch-and-bound algorithm, a parallel strategy with multiple branching is used for good processor scheduling. Performance results of the system on NCUBE are given.

## ACKNOWLEDGEMENTS

First, I would like to thank my thesis adviser, John Dennis, to whom I am particularly grateful for his assistance, encouragement and for believing in me and my abilities. It is he who introduced me to the fascinating area of numerical optimization and suggested this problem to me. With great patience, he spent much time discussing my work with me, and read my papers even when they were not written carefully. He showed interest in every bit of my progress and with his extensive research experience and background in numerical optimization, he gave me many ideas about what might be wrong in my work and where I could make improvements. Without his direction and push, I could hardly hope to have developed my algorithm and its theory.

Second, I would like to thank my thesis co-adviser, Robert Bixby, for his valuable advice on my thesis, especially for the amount of time he spent listening to my report and giving me suggestions. I learned most of my knowledge about linear programming and combinatorial optimization from him and enjoyed all his classes. He also taught me how to be both a good researcher and teacher.

I would like to thank my committee members, Kenneth Kennedy and Partha Sengupta, for caring about the work contained in this thesis. Their presence on my committee motivated me to work harder on the parallel implementation of the subgradient algorithm and on the possible application of my work in the optimization of the gas-pipeline network operation.

I would also like to thank Richard Tapia who accepted me into the graduate program and taught me the fundamental theory on optimization. Many ideas on constrained optimization in this thesis come from his class on optimization theory. I also thank Andrew Boyd for discussing the parallel branch-and-bound algorithm and I thank Guangye Li for understanding and caring about me and teaching me the parallel algorithm for solving triangular systems.

I have special thanks to my wife, Qiyang Fu, for her love, support and patience. She has been with me for almost 12 years since we met in college. She is definitely one of the people who will be delighted most to see me receive a Ph.D. degree.

Finally, this research is supported by the Center for Research on Parallel Computing at Rice University.



TO MY PARENTS



## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Problem . . . . .	1
1.2	Combinatorial and Algebraic Methods . . . . .	3
1.3	Our Approach . . . . .	7
1.4	Material to Follow . . . . .	9
<b>2</b>	<b>The Subgradient Algorithm</b>	<b>11</b>
2.1	A Necessary and Sufficient Condition . . . . .	11
2.2	The General Algorithm . . . . .	16
2.3	Convergence Properties . . . . .	19
<b>3</b>	<b>Computing Subgradients</b>	<b>26</b>
3.1	The Method for Simple Problems . . . . .	26
3.2	Mathematical Derivations . . . . .	31
3.3	Extension to General Problems . . . . .	42
<b>4</b>	<b>The Continuous Optimization Subproblems</b>	<b>44</b>

4.1	Nonlinear Least Square . . . . .	41
4.1.1	Backtracking . . . . .	41
4.1.2	$\epsilon$ -Approximation . . . . .	41
4.2	Nonlinear Constrained Optimization . . . . .	41
4.2.1	Structural BFGS Updating . . . . .	49
4.2.2	Solving Partially Positive Definite Systems . . . . .	50
<b>5</b>	<b>On Solving Integer Minimax Subproblems</b>	<b>53</b>
5.1	The General Branch and Bound Procedure . . . . .	53
5.2	Branching Strategies . . . . .	56
5.3	Lower and Upper Bounds . . . . .	57
<b>6</b>	<b>Parallelization and Numerical Experiments</b>	<b>59</b>
6.1	NIPACK System . . . . .	59
6.2	Basic Subroutines . . . . .	63
6.2.1	Communication Subroutines . . . . .	63
6.2.2	Matrix Computation Subroutines . . . . .	64
6.3	Parallelization for Computing Subgradients . . . . .	66
6.4	Parallelization for the Branch-and-Bound Procedure . . . . .	67
6.5	Preliminary Numerical Results . . . . .	71
<b>7</b>	<b>Summary and Further Research</b>	<b>82</b>

**Bibliography**

85

**Appendix A Print-out for an Example Problem**

91



# Chapter 1

## Introduction

### 1.1 The Problem

In this work, we are interested in solving a class of nonlinear integer programming problems:

$$\min f(x) \tag{1.1}$$

$$x \in B^n = \{0, 1\}^n$$

or its natural extension:

$$\min f(x) \tag{1.2}$$

$$x \in R^n \quad \text{integral}$$

where  $f : R^n \longrightarrow R$  is a general nonlinear function.

This class of problems have important theoretical and practical applications. For example, consider the problem that for any norm  $\| \cdot \|$ ,

$$\min \| b - Ax \| \tag{1.3}$$

$$x \in R^n \quad \text{integral}$$

where  $b \in R^m$  and  $A$  is an  $m \times n$  matrix with integer elements. This problem is called in integer programming the closest vector problem and has been proved to be

$\mathcal{NP}$ -hard even for simple norms such as  $l_2$  and  $l_\infty$  (see Van Emde Boas [1981], Lovás [1986], [1989]).

Another example is related to a class of more general problems, mixed-integer nonlinear programming problems, which have found their important applications recently in the gas pipeline network optimization (see Percell [1987]). It turns out that under some circumstances, problems of this class can be reduced to general nonlinear integer programming problems. For instance, an unconstrained mixed-integer nonlinear programming problem,  $\min \{g(x, y) : y \in R^m, x \in R^n, x \text{ integral}\}$ , can be formulated, under some appropriate assumptions, as the following problem:

$$\begin{aligned} \min \quad & f(x) \\ & x \in R^n \quad \text{integral} \end{aligned} \tag{1.4}$$

with  $f(x) = \min \{g(x, y) : y \in R^m\}$ .

If  $x$  is bounded, Problem 1.2 can be equivalently transformed into Problem 1.1. So we will concentrate only on solving Problem 1.1.

A lot of research have been done on solving Problem 1.1 since the end of the 1950's. In Section 1.2, we review three traditional approaches briefly. Section 1.3 contains an outline of our approach. Section 1.4 introduces the material to be discussed in the remainder of this thesis.



## 1.2 Combinatorial and Algebraic Methods

Several approaches to the solution of Problem 1.1 have been studied in the last 30 years. The main ones are enumeration, algebraic, and linearization approaches. They all also apply to nonlinear constrained 0 – 1 integer programming. Garfinkel and Nemhauser [1972] and Hansen, Jaumard and Mathon [1989] have general descriptions.

### Enumeration

The most simple way to solve Problem 1.1 is to enumerate all the feasible solutions and then choose the best, i.e., the one such that the objective value of the problem is the minimum. Unfortunately, this method can only work for very small problems for there can be  $2^n$  feasible solutions in total for an  $n$ -dimensional problem and it is impossible to compute them all in a reasonable time for  $n$  large.

A practically better way is to enumerate only a “heuristically” selected subset of feasible solutions and take the best solution in this subset as the approximation to the optimal solution. Here the solution obtained is only a local optimal solution. We can not guarantee it is also globally optimal.

A more intelligent and sophisticated enumeration scheme is referred to as the branch-and-bound procedure, or the divide-and-conquer method. In this method, the original problem is divided into several subproblems such that the problem can be solved if all the subproblems are solved. The process also is applied recursively to all subproblems. It terminates for some subproblem if either the optimal solution for the subproblem is found or the subproblem can be eliminated. The latter can be

determined by comparing the upper or lower bounds for the objective function with the optimal value for some other already solved subproblem. This method already is used for linear integer programming. But in most general cases, it may not be effective without efficient subproblem elimination procedures.

### Algebraic Method

The most studied algebraic method is the *Basic Algorithm* described in Hammer, Rosenberg and Rudeanu [1963], and Hammer and Rudeanu [1968].

The method exploits optimality conditions as follows. The objective function is first expressed in a multilinear form, i.e., a sum of distinct cross products of variables. Then the problem becomes

$$\min_{x \in \{0,1\}^n} f(x) = \sum_{k=1}^p c_k T_k$$

where

$$T_k = \prod_{j \in N_k} x_j, \quad N_k \subseteq N = \{1, 2, \dots, n\}, \quad k = 1, 2, \dots, p.$$

Let  $f_1$ , the function to be minimized, be written in the form

$$f_1(x_1, x_2, \dots, x_n) = x_1 \Delta_1(x_2, x_3, \dots, x_n) + \eta_1(x_2, x_3, \dots, x_n)$$

where the function  $\Delta_1$  and  $\eta_1$  do not depend on  $x_1$ . Clearly, there exists an optimal solution of  $f_1$ , say  $(x_1^*, x_2^*, \dots, x_n^*)$ , such that  $x_1^* = 1$  if and only if  $\Delta_1(x_2^*, x_3^*, \dots, x_n^*) < 0$ . This leads us to define a function  $\psi_1(x_2, x_3, \dots, x_n)$ , such that  $\psi_1(x_2, x_3, \dots, x_n) = \Delta_1(x_2, x_3, \dots, x_n)$  if  $\Delta_1(x_2, x_3, \dots, x_n) < 0$ , and  $\psi_1(x_2, x_3, \dots, x_n) = 0$  otherwise.

Assume that a polynomial expression of  $\psi_1$  has been obtained. Letting  $f_2 = \psi_1 + \eta_1$  reduces the original problem in  $n$  variables to the problem of minimizing  $f_2$ , which depends only on the  $n - 1$  variables  $x_2, x_3, \dots, x_n$ . Continuing this process for  $x_2, x_3, \dots, x_{n-1}$ , succesively, yields two sequences of functions  $f_1, f_2, \dots, f_n$  and  $\psi_1, \psi_2, \dots, \psi_{n-1}$ , where  $f_i$  depends on  $(n - i + 1)$  variables. An optimal solution  $(x_1^*, x_2^*, \dots, x_n^*)$  of  $f_1$  can then be traced back from the minimizer  $x_n^*$  of  $f_n$ , using recursion:

$$x_i^* = 1 \text{ if and only if } \psi_i(x_{i+1}^*, x_{i+2}^*, \dots, x_n^*) < 0 \quad (i = 1, 2, \dots, n - 1).$$

The efficiency of this procedure depends critically on how the polynomial expressions of  $\psi_1, \psi_2, \dots, \psi_{n-1}$  are obtained. Various methods to obtain these functions are proposed in Hammer and Rudeanu [1968], and Crama, Hansen and Jaumard [1990].

Crama, Hansen and Jaumard [1990] show that the algebraic method described above may work well for problems with special structures. But it is not promising for general problems due to the work for the algebraic manipulations and to the very large memory space usually required.

Moreover, to apply this method, the objective function is required to be in a multilinear form. This form, in principle, can be obtained for any 0–1 nonlinear function due to the fact that for any function  $g$ ,

$$g(x) = \sum_{T \in J^*} g(x^{(T)}) \prod_{j \in T} x_j \prod_{j \in J-T} (1 - x_j)$$

where  $T \subseteq J = \{1, 2, \dots, n\}$ ,  $J^*$  is the set of subsets of  $J$  and  $x^{(T)}$  is defined such that  $x_j^{(T)} = 1$  if  $j \in T$  and  $x_j^{(T)} = 0$  otherwise. However, the computation involved in achieving this formulation may be on the order of total enumeration of the  $2^n$  binary  $n$ -vectors. For problems with complicated objective functions such as Problem 1.3 and Problem 1.4, this can be as expensive as solving problems themselves.

### Linearization

In this approach, rules are invented to linearize the nonlinear objective function and then reduce the problem to a linear 0–1 integer program (see Danzig [1960] and Fortet [1959]). Generally, the problem is assumed to be in the multilinear form. Then it is linearized by replacing each product of variables by a new 0–1 variable and adding some linear constraints. Glover and Woolsey [1973], [1974] and Balas and Mazzola [1984] propose special rules to obtain the linearization with fewer new variables and constraints introduced.

The linearization approach seems successful for problems with a few products of variables, but the linearization becomes prohibitive both in terms of space and time when all or most possible products of variables are present and  $n$  increases. Hansen, Jaumard and Mathon [1989] indicate that aggregating constraints to obtain a more compact linearization also does not appear worthwhile, as the relaxations are then less tight than in the original linearization. Moreover, as in the algebraic method, this approach requires the objective function to be transformed into a multilinear form, which is difficult for complicated objective functions.

In conclusion, all three approaches for nonlinear 0–1 integer programming work both theoretically and practically for problems with special structures or problems of relatively small dimension. But for general problems, they all have difficulties. Directly applying the branch-and-bound procedure to a general problem usually produces too many subproblems. The algebraic and linearization approaches need the objective function to be transformed into a special polynomial form, which does not apply in general for problems such as Problem 1.3 and Problem 1.4.

### 1.3 Our Approach

In our approach, Problem 1.1 is considered for general cases. We are interested in solving problems with general or complicated objective functions such as Problem 1.3 and Problem 1.4.

First, instead of looking at the problem combinatorially or algebraically, we consider it as a nonsmooth problem over the set of all 0–1 integer points. We use the notion of subgradient from the theory of nonsmooth analysis and then construct a necessary and sufficient condition for the optimal solution, i.e.,

- *A feasible solution for the problem is optimal if and only if a subgradient of the objective function at this solution is equal to 0.*

We use this condition as one of our optimality testing criteria. It was not considered in the earlier approaches. For instance, in the branch-and-bound procedure, the optimal solution can not be determined until all necessary feasible solutions are enumerated.

Second, our algorithm searches for the solution iteratively among feasible points and in each iteration, it generates the next iterative point by solving the problem for a local piecewise linear model which is constructed with the supporting planes for the objective function at the set of iterative points already generated. The supporting planes are computed by using special continuous optimization techniques. The problem for the local piecewise linear model in each iteration is equivalent to an integer linear minimax problem, which can be solved with any standard method for linear integer programming.

Finally, our work also involves parallel computation. We implement the algorithm on parallel distributed-memory machines to explore the use of supercomputing tools in solving large and hard problems.

Our approach differs from, but also relates to, traditional approaches described in the previous section. In our approach, there is an enumeration mechanism embedded, but different from the general enumeration scheme, we do not apply it to the problem directly. We only apply it to locally constructed linear subproblems for which the enumeration procedure seems to be more effective.

We also try to approximate the objective function with a linearization technique. But we do this in a different way and for a different purpose, compared with the general linearization approach. In our approach, supporting planes are used to provide a piecewise linear approximation to the objective function. This approximation is used in each iteration of the algorithm for constructing the local model of the problem.

## 1.4 Material to Follow

There are certain aspects of our approach that require further discussion. For example, we have only hinted at how the idea in nonsmooth analysis can be introduced in solving our problem. But we have not given exact definitions for what we mean by “subgradient” and “supporting plane”, etc. Also, we have not described in detail how to compute subgradients and how to solve the linear integer minimax problem that happens to be a subproblem in our algorithm. In Chapter 2, we present a formal description of our algorithm. We also discuss the optimality testing criteria and convergence properties.

Computing subgradients is very important in our algorithm for both the construction of the local piecewise linear model and the optimality testing. We discuss this issue in Chapter 3. We first present an algorithm for simple problems, and then extend it to general cases. In our algorithm, the problem of computing subgradients is reduced to several continuous optimization subproblems. We present all related mathematical derivations and describe strategies for solving these subproblems. Chapter 4 contains more details about this.

In Chapter 5, we present an implicit enumeration procedure for solving the linear integer minimax subproblem generated at each iteration in our algorithm. While the enumeration scheme is relatively effective for linear integer programming, we still have difficulties in solving the linear integer minimax subproblems in general. We discuss several strategies to speed up the enumeration procedure.

In Chapter 6, we describe a parallel software system to implement our algorithm. The preliminary numerical results for testing our algorithm also are presented. Our system is written in EXPRESS C and it will run on several parallel distributed memory machines such as the NCUBE, SYMULT and MEIKO MK200. The system contains a group of parallel subroutines used for either continuous or discrete optimization such as subroutines for  $QR$ ,  $LU$  and Cholesky factorizations, triangular system solvers, a parallel branch-and-bound procedure and so on. Performance results of the system on the NCUBE are given.

Chapter 7 summarizes the material presented in Chapter 1 through Chapter 6. We also discuss possible extensions and improvements in both our algorithm for nonlinear integer programming and in our software system. Appendix A contains the print-out from solving an example problem via the use of our system.



## Chapter 2

### The Subgradient Algorithm

#### 2.1 A Necessary and Sufficient Condition

The algorithm we propose for Problem 1.1 will be called the subgradient algorithm because subgradient information is used in the main procedure of the algorithm. In this chapter, we give a formal description of the algorithm and present some of its mathematical properties.

Suppose we are given a nonlinear objective function  $f : R^n \rightarrow R$ . Consider its restriction  $f : B^n \rightarrow R$  denoted by  $f^r$ . It is a function over the discrete set of all 0–1 integer points, and therefore it is nondifferentiable, or in other words, nonsmooth. We treat Problem 1.1 as a nonsmooth problem by writing it as

$$\min f^r(x) \tag{2.1}$$

$$x \in B^n = \{0,1\}^n$$

We call  $f$  the continuous objective function and  $f^r$  the discrete objective function, and introduce the following definitions for  $f^r$ :

**Definition 2.1.1** A subgradient of  $f^r$  at  $\bar{x} \in B^n$  is a vector  $s \in R^n$  such that

$$s^T(x - \bar{x}) \leq f^r(x) - f^r(\bar{x}) \quad \forall x \in B^n.$$

**Definition 2.1.2** The subdifferential of  $f^r$  at  $\bar{x} \in B^n$  is the set of all subgradients of  $f^r$  at  $\bar{x}$  defined as:

$$\partial f^r(\bar{x}) = \{s \in R^n : s^T(x - \bar{x}) \leq f^r(x) - f^r(\bar{x}) \quad \forall x \in B^n\}$$

**Definition 2.1.3** A supporting plane of  $f^r$  at  $\bar{x} \in B^n$  is a hyperplane defined by

$$g_{\bar{x}}(x) = f^r(\bar{x}) + s_{\bar{x}}^T(x - \bar{x}) \quad \text{where } s_{\bar{x}}^T \in \partial f^r(\bar{x})$$

We say a supporting plane is “good” if it is tight as a bounding function. For example, in Figure 2.1, we say  $B$  is better than  $A$ , and  $C$  is the best. Also, given a subgradient, we can define a supporting plane and vice versa. So the notions of subgradient and supporting plane are related.

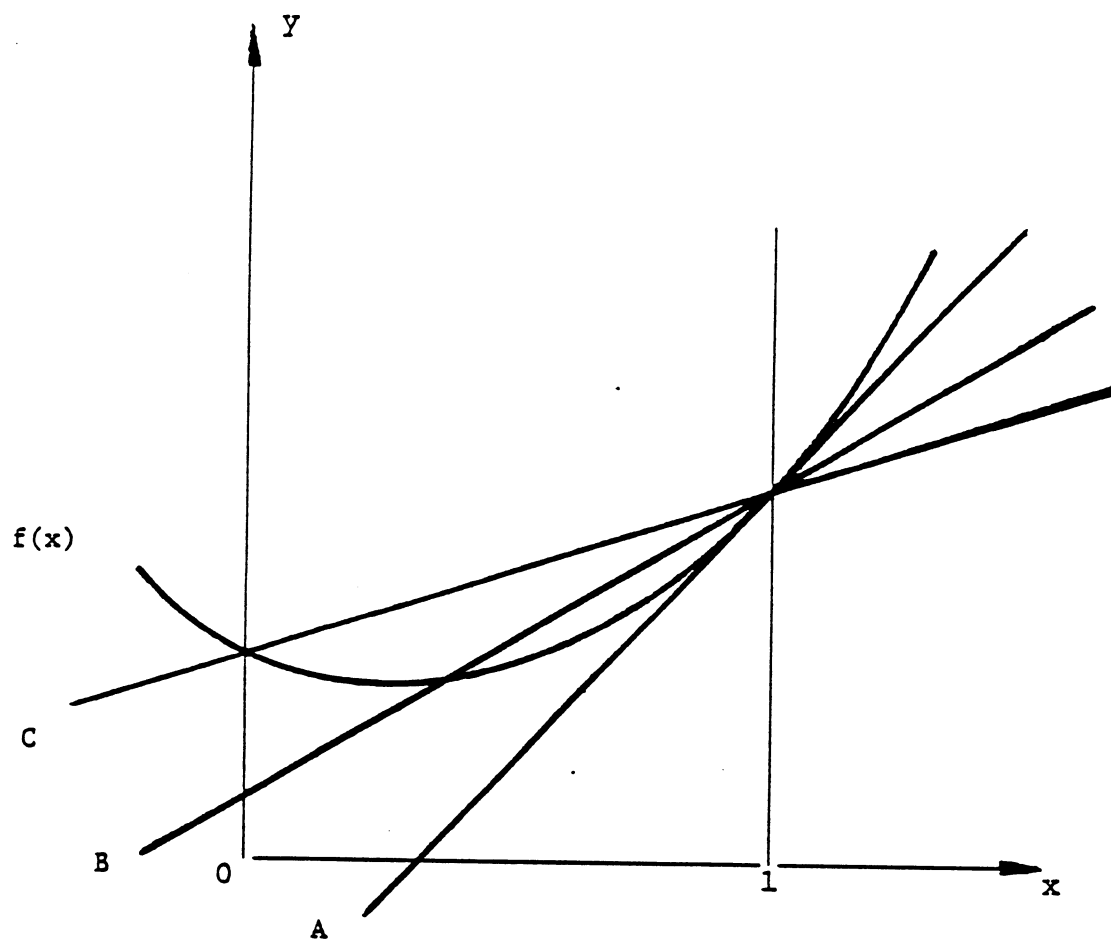


Figure 2.1 Simple examples for supporting planes.

With these definitions, we can obtain the following easy, but important, facts:

**Theorem 2.1.1** Suppose  $f$  is convex and differentiable, and let  $\nabla f(\bar{x})$  be the gradient of  $f$  at  $\bar{x}$ , then  $\nabla f(\bar{x}) \in \partial f^r(\bar{x}) \quad \forall \bar{x} \in B^n$ .

**Proof:** It suffices to show that for any  $\bar{x} \in B^n$ ,

$$\nabla f(\bar{x})^T(x - \bar{x}) \leq f(x) - f(\bar{x}) \quad \forall x \in B^n \quad (2.2)$$

For  $x = \bar{x}$ , inequality (2.2) holds obviously. So we only need to consider  $x \neq \bar{x}$ ,  $x \in B^n$ . Since  $f$  is differentiable, the directional derivative of  $f$  at  $\bar{x}$  in the direction of  $(x - \bar{x})$ , defined as

$$\lim_{\lambda \rightarrow 0} \frac{f(\bar{x} + \lambda(x - \bar{x})) - f(\bar{x})}{\lambda},$$

exists and equals  $\nabla f(\bar{x})^T(x - \bar{x})$ .

Since  $f$  is convex, for  $\lambda \in (0, 1]$ ,

$$\begin{aligned} f(x) - f(\bar{x}) &= \frac{\lambda f(x) + (1 - \lambda)f(\bar{x}) - f(\bar{x})}{\lambda} \\ &\geq \frac{f(\lambda x + (1 - \lambda)\bar{x}) - f(\bar{x})}{\lambda} \\ &= \frac{f(\bar{x} + \lambda(x - \bar{x})) - f(\bar{x})}{\lambda} \end{aligned}$$

which implies

$$\begin{aligned} f(x) - f(\bar{x}) &\geq \lim_{\lambda \rightarrow 0} \frac{f(\bar{x} + \lambda(x - \bar{x})) - f(\bar{x})}{\lambda} \\ &= \nabla f(\bar{x})^T(x - \bar{x}). \end{aligned}$$

□

**Theorem 2.1.2** The subdifferential  $\partial f^r(\bar{x})$  of  $f^r$  at  $\bar{x} \in B^n$  is a convex set.

**Proof:** For any  $\bar{x}$ , let  $s_1, s_2 \in \partial f^r(\bar{x})$ . We show that

$$\lambda s_1 + (1 - \lambda)s_2 \in \partial f^r(\bar{x}) \quad \text{for any } \lambda \in [0, 1].$$

Since  $s_1, s_2 \in \partial f^r(\bar{x})$ ,

$$s_1(x - \bar{x}) \leq f^r(x) - f^r(\bar{x}) \quad \forall x \in B^n, \text{ and}$$

$$s_2(x - \bar{x}) \leq f^r(x) - f^r(\bar{x}) \quad \forall x \in B^n$$

So, for any  $x \in B^n$  and  $\lambda \in [0, 1]$ ,

$$\begin{aligned} & (\lambda s_1 + (1 - \lambda)s_2)(x - \bar{x}) \\ &= \lambda s_1(x - \bar{x}) + (1 - \lambda)s_2(x - \bar{x}) \\ &\leq \lambda(f(x) - f(\bar{x})) + (1 - \lambda)(f(x) - f(\bar{x})) \\ &= f(x) - f(\bar{x}) \end{aligned}$$

which, by the definition for a subgradient, implies

$$\lambda s_1 + (1 - \lambda)s_2 \in \partial f^r(\bar{x}) \quad \text{for any } \lambda \in [0, 1].$$

□

**Theorem 2.1.3** A necessary and sufficient condition for  $x^* \in B^n$  to be the minimizer of  $f^r$  (and also  $f$ ) over  $B^n$  is  $0 \in \partial f^r(x^*)$ .

**Proof:** By the definition for a subgradient,  $0 \in \partial f^r(x^*)$  for  $x^* \in B^n$  if and only if

$$0(x - x^*) \leq f^r(x) - f^r(x^*) \quad \forall x \in B^n,$$

which exactly means

$$f^r(x^*) \leq f(x) \quad \forall x \in B^n.$$

□

Note that the subgradient of a function at a given point might not be unique. Typically there can be infinitely many. There is no general methods, especially for nonlinear nonsmooth functions, to compute the whole set of subgradients. This causes difficulties in using Theorem 2.1.3 to solve our problem. However, to determine whether or not there is a zero subgradient in a given subdifferential, some special techniques can be used that do not require the entire set. We will describe the method used in our algorithm in the following sections.

## 2.2 The General Algorithm

Algorithm 2.2.1 given below is an outline of the algorithm we propose to solve Problem 1.1. In this algorithm, the discrete form (2.1) of Problem 1.1 is considered. For simplicity, we will always refer to  $f^r$  as the objective function of the problem.

The algorithm carries out an iterative procedure from an initial guess  $x^{(0)}$  as follows. Let  $x^{(i)}$  be the current iterative point at the  $i$ th iteration. At the beginning of this iteration, if  $x^{(i)} = x^{(j)}$  for some  $j$  such that  $j < i$ , (or in other words,  $x^{(i)}$  has

been generated before in the whole procedure), or if the objective function  $f^r$  has a 0 subgradient at  $x^{(i)}$ , the algorithm stops and  $x^{(i)}$  is an optimal solution. Otherwise, a supporting plane  $g_{x^{(i)}}$  for the objective function at  $x^{(i)}$  is generated. All generated supporting planes  $g_{x^{(j)}}$  for  $j = 1, 2, \dots, i$  define a local piecewise linear model  $p(x) = \max_{1 \leq j \leq i} \{g_{x^{(j)}}(x)\}$  for the objective function. Then, the algorithm solves the 0–1 integer programming problem for this local piecewise linear model, takes the optimal solution for this local subproblem as the next iterative point  $x^{(i+1)}$  and goes to the next iteration.

There are two optimality testing criteria in our algorithm. One is the necessary and sufficient condition stated in Theorem 2.1.3. It is tested for an iterative point by looking to see if there is a supporting plane defined by a zero subgradient. This is done when the algorithm is generating supporting planes. The other test is if the algorithm repeats some iterative point. We will prove in the next section that whenever this happens, the repeated point is then an optimal solution and the algorithm can terminate. This criterion prevents cycling in the algorithm and guarantees that the algorithm terminates in finitely many steps.

Algorithm 2.2.1 {*A subgradient algorithm*}

0 {*Initialization*}

$T = \phi, H = \phi, i = 0$

pick up  $x^{(i)} \in B^n$

1 {*Iteration*}

do while  $i \leq m$

1.1 {*Optimality testing*}

if  $x^{(i)} \in T$  or  $0 \in \partial f^r(x^{(i)})$  is known then

$x^{(i)}$  is the optimal solution, stop

1.2 {*Generating supporting planes*}

$T = T \cup \{x^{(i)}\}$

$H = H \cup \{g_{x^{(i)}} : g_{x^{(i)}}(x) = f^r(x^{(i)}) + s_{x^{(i)}}^T(x - x^{(i)}), s_{x^{(i)}} \in \partial f^r(x^{(i)})\}$

1.3 {*Solving a linear integer minimax problem*}

find a solution  $x^{(*)}$  for

$\min_{x \in B^n} \{p(x) = \max \{g(x) : g \in H\}\}$

1.4 {*Updating*}

$i = i + 1$

$x^{(i)} = x^{(*)}$

end do

□



## 2.3 Convergence Properties

As described in the previous section, instead of solving the original nonlinear integer programming problem directly, our algorithm actually solves a sequence of linear integer programming subproblems. Each of them corresponds to an integer minimization problem for a piecewise linear function  $p$  defined by a group of supporting planes of the objective function. Let  $p^{(i)}$  and  $p^{(i+1)}$  denote the piecewise linear functions generated in the  $i$ th and  $(i+1)$ th iterations respectively, then,  $p^{(i+1)}$  is generated by adding one more supporting plane into  $p^{(i)}$ . We now prove some useful results for our algorithm.

**Theorem 2.3.1** Let  $p^{(i)}$  be the piecewise linear function constructed in the  $i$ th iteration of Algorithm 2.2.1, then for any  $i$ ,  $p^{(i)}(x) \leq f^r(x)$   $\forall x \in B^n$ .

**Proof:** As presented in Algorithm 2.2.1,

$$\begin{aligned} p^{(i)}(x) &= \max\{g(x) : g \in H\} \\ &= \max_{0 \leq j \leq i} \{g^{(j)}(x) : g^{(j)} \in H\} \end{aligned}$$

where  $g^{(j)} \in H$  is a supporting plane for the objective function  $f^r$  generated in the  $j$ th iteration of the algorithm. By the definition of a supporting plane,

$$g^{(j)}(x) = f^r(x^{(j)}) + (s^{(j)})^T(x - x^{(j)})$$

where  $x^{(j)} \in B^n$  is the  $j$ th iterative point and  $s^{(j)} \in \partial f^r(x^{(j)})$ . By the definition of subgradient,

$$(s^{(j)})^T(x - x^{(j)}) \leq f^r(x) - f^r(x^{(j)}) \quad \forall x \in B^n.$$

So,  $g^{(j)}(x) \leq f^r(x) \quad \forall x \in B^n$ . Since this is true for all  $j$ ,  $0 \leq j \leq i$ , we have also

$$\max_{0 \leq j \leq i} \{g^{(j)}(x) : g^{(j)} \in H\} \leq f^r(x) \quad \forall x \in B^n$$

which exactly means  $p^{(i)}(x) \leq f^r(x) \quad \forall x \in B^n$ .  $\square$

**Theorem 2.3.2** Let  $z^{(i)}$  and  $z^{(i+1)}$  be the optimal values of the linear integer minimax subproblems in the  $i$ th and  $(i+1)$ th iterations in Algorithm 2.2.1 respectively, then for any  $i$ ,  $z^{(i)} \leq z^{(i+1)}$ . If in addition the optimal solution  $x^{(i+1)}$  for the linear integer minimax subproblem in the  $i$ th iteration of the algorithm is unique and  $x^{(i+1)} \neq x^{(i+2)}$ , then  $z^{(i)} < z^{(i+1)}$ .

**Proof:** First prove  $z^{(i)} \leq z^{(i+1)}$ .

As defined in Algorithm 2.2.1,

$$\begin{aligned} z^{(i)} &= \min_{x \in B^n} p^{(i)}(x), \text{ and} \\ z^{(i+1)} &= \min_{x \in B^n} p^{(i+1)}(x). \end{aligned}$$

So, it suffices to show  $p^{(i)}(x) \leq p^{(i+1)} \quad \forall x \in B^n$ . By the definition of  $p^{(i)}$ , for any  $x \in B^n$ ,

$$p^{(i)}(x) = \max_{0 \leq j \leq i} \{g^{(j)}(x) : g^{(j)} \in H\}$$

$$\begin{aligned}
&\leq \max_{0 \leq j \leq i+1} \{g^{(j)}(x) : g^{(j)} \in H\} \\
&= p^{(i+1)}(x).
\end{aligned}$$

So,  $p^{(i)}(x) \leq p^{(i+1)}(x) \quad \forall x \in B^n$  and  $z^{(i)} \leq z^{(i+1)}$  follows immediately.

Now we prove  $z^{(i)} < z^{(i+1)}$  if the solution to the subproblem

$$\min_{x \in B^n} p^{(i)}(x) \tag{2.3}$$

is unique and  $x^{(i+1)} \neq x^{(i+2)}$ . The proof is by contradiction.

Suppose Problem (2.3) has a unique solution  $x^{(i+1)}$  and  $z^{(i)} = z^{(i+1)}$ . Since  $z^{(i)} = z^{(i+1)}$ ,  $p^{(i)}(x^{(i+1)}) = p^{(i+1)}(x^{(i+2)})$ . But

$$\begin{aligned}
p^{(i+1)}(x^{(i+2)}) &= \max_{0 \leq j \leq i+1} \{g^{(j)}(x^{(i+2)}) : g^{(j)} \in H\} \\
&= \max \{p^{(i)}(x^{(i+2)}), g^{(i+1)}(x^{(i+2)})\}.
\end{aligned}$$

So,  $p^{(i)}(x^{(i+1)}) \geq p^{(i)}(x^{(i+2)})$ .

But  $p^{(i)}(x^{(i+1)}) < p^{(i)}(x^{(i+2)})$  by the uniqueness of  $x^{(i+1)}$  and the fact that  $x^{(i+1)} \neq x^{(i+2)}$ . Contradiction! Therefore,  $z^{(i)} \neq z^{(i+1)}$ , and  $z^{(i)}$  can be only strictly less than  $z^{(i+1)}$  by the first argument of the theorem.  $\square$

**Theorem 2.3.3** Let  $T = \{x^{(j)} \in B^n, j = 1, \dots, i\}$  be a sequence of iterative points generated by Algorithm 2.2.1 before the algorithm starting the  $i$ th iteration. If  $\exists j < i$  such that  $x^{(j)} = x^{(i)}$ , then  $x^{(i)}$  must be a solution for Problem (2.1).

**Proof:** let  $j$  be the integer such that  $j < i$  and  $x^{(j)} = x^{(i)}$ .

As defined in the algorithm,  $x^{(i)}$  is the solution to the subproblem

$$\min_{x \in B^n} p^{(i-1)}(x).$$

So,

$$p^{(i-1)}(x^{(i)}) \leq p^{(i-1)}(x) \quad \forall x \in B^n.$$

By Theorem 2.3.1,

$$p^{(i-1)}(x) \leq f^r(x) \quad \forall x \in B^n.$$

Thus,

$$p^{(i-1)}(x^{(i)}) \leq f^r(x) \quad \forall x \in B^n$$

and particularly

$$p^{(i-1)}(x^{(i)}) \leq f^r(x^{(i)}).$$

But

$$\begin{aligned} p^{(i-1)}(x^{(i)}) &= \max_{0 \leq k \leq i-1} \{g^{(k)}(x^{(i)}) : g^{(k)} \in H\} \\ &\geq g^{(j)}(x^{(i)}) \\ &= g^{(j)}(x^{(j)}) \\ &= f^r(x^{(j)}) \\ &= f^r(x^{(i)}). \end{aligned}$$

Then we have  $f^r(x^{(i)}) = p^{(i-1)}(x^{(i)})$ ,

$$f^r(x^{(i)}) \leq f^r(x) \quad \forall x \in B^n,$$

and  $x^{(i)}$  is optimal for Problem (2.1).  $\square$

**Theorem 2.3.4** Algorithm 2.2.1 is finite.

**Proof:** It immediately follows from Theorem 2.3.3 and the fact that there are only finitely many distinct points  $x \in B^n$ .  $\square$

**Corollary 2.3.1** Let  $T = \{x^{(j)} \in B^n, j = 1, \dots, i\}$  be the sequence of iterates generated by Algorithm 2.2.1 up to the  $i$ th iteration. Let  $z^{(ji)} = f^r(x^{(ji)})$  be the minimal in  $T$ . Then for  $z^*$ , the optimal value of Problem (2.1), and  $z^{(i-1)} = p^{(i-1)}(x^{(i)})$ , the optimal value of the linear integer minimax subproblem in the  $(i-1)$ th iteration of the algorithm,

$$z^{(i-1)} \leq z^* \leq z^{(ji)},$$

and also

$$|z^{(ji)} - z^{(i)}| = 0 \quad \text{for } i \text{ sufficiently large.}$$

**Proof:** First we prove

$$z^{(i-1)} \leq z^* \leq z^{(ji)}.$$

By Theorem 2.3.1,

$$p^{(i-1)}(x) \leq f^r(x) \quad \forall x \in B^n.$$

So,

$$z^{(i-1)} = \min_{x \in B^n} p^{(i-1)}(x) \leq \min_{x \in B^n} f^r(x) = z^*$$

The second inequality follows since any feasible point  $x \in B^n$  yields an upper bound  $f^r(x)$  for the optimal value of  $f^r$ .

Now we prove

$$|z^{(ji)} - z^{(i-1)}| = 0 \quad \text{for } i \text{ sufficiently large.}$$

By Theorem 2.3.3 and Theorem 2.3.4, the algorithm stops when  $x^{(i)} = x^{(j)}$  for some  $j < i$  where  $x^{(i)}$  is the current iterative point. Then as in the proof for Theorem 2.3.3,

$$z^{(i-1)} = p^{(i-1)}(x^{(i)}) = f^r(x^{(i)}) = z^*.$$

Since now  $z^{(ji)} = z^*$ , then  $z^{(i-1)} = z^{(ji)}$ .  $\square$

An interesting observation from Corollary 2.3.1 is that after the  $i$ th iteration, we can get the best solution the algorithm is able to find in  $i + 1$  iterations. As in Corollary 2.3.1, if  $z^{(ji)}$  is the function value for this solution and  $z^*$  the optimal value, the error between the two values is bounded by

$$|z^{(ji)} - z^{(i-1)}|.$$

The bound is decreasing as  $i$  increases.

Finally, since it is not straightforward to test if there is a zero subgradient in applying Theorem 2.1.3 for the optimality testing in Algorithm 2.2.1, we state a

different, but equivalent, necessary and sufficient condition in the following theorem.

It turns out that this condition can be obtained more easily in our algorithm.

**Theorem 2.3.5** A necessary and sufficient condition for  $x^* \in B^n$  to be the minimizer of  $f^r$  (and also  $f$ ) over  $B^n$  is  $\exists s \in \partial f^r(x^*)$  such that

$$s_i \leq 0 \quad \forall i \text{ such that } x_i^* = 1 \quad (2.4)$$

and

$$s_i \geq 0 \quad \forall i \text{ such that } x_i^* = 0. \quad (2.5)$$

**Proof:** The necessity follows immediately from Theorem 2.1.3 and the fact that  $s = 0$  satisfies conditions (2.4) and (2.5). For the sufficiency, suppose  $\exists s \in \partial f^r(x^*)$  satisfying conditions (2.4) and (2.5). Then

$$s^T(x - x^*) \leq f^r(x) - f^r(x^*) \quad \forall x \in B^n$$

and it is easy to verify that

$$0 \leq s^T(x - x^*) \quad \forall x \in B^n.$$

So,

$$\begin{aligned} 0(x - x^*) &\leq s^T(x - x^*) \\ &\leq f^r(x) - f^r(x^*) \quad \forall x \in B^n \end{aligned}$$

and then  $0 \in \partial f^r(x^*)$ . By Theorem 2.1.3,  $x^*$  is a solution.  $\square$

## Chapter 3

### Computing Subgradients

#### 3.1 The Method for Simple Problems

In each iteration of Algorithm 2.2.1, if the iterate  $x^{(i)}$  is not optimal, the algorithm must generate a supporting plane at  $x^{(i)}$ :

$$g_{x^{(i)}}(x) = f^r(x^{(i)}) + s_{x^{(i)}}^T(x - x^{(i)}), \quad s_{x^{(i)}} \in \partial f^r(x^{(i)})$$

to update the local piecewise linear model of the objective function. To do this, the function value  $f^r(x^{(i)})$  and a subgradient  $s_{x^{(i)}} \in \partial f^r(x^{(i)})$  are required. Since  $\partial f^r(x^{(i)})$  is a set and not given explicitly, it is difficult to compute a nontrivial subgradient in  $\partial f^r(x^{(i)})$ .

Let us consider problems whose continuous objective functions are assumed to be convex and differentiable. For a problem of this class, by Theorem 2.1.1,  $\nabla f(x^{(i)})$ , the gradient of  $f$  at  $x^{(i)}$ , is contained in  $\partial f^r(x^{(i)})$ . So, a trivial way to choose  $s_{x^{(i)}}$  for each  $i$  is to set

$$s_{x^{(i)}} = \nabla f(x^{(i)}).$$

However, it happens that with this strategy, the piecewise linear function  $p$  generated in Algorithm 2.2.1 may not be a good bounding function of  $f^r$ . Geometrically, this is



because each function  $g_{x^{(i)}}$  with  $s_{x^{(i)}} = \nabla f(x^{(i)})$  might be too “steep” to be a “good” supporting plane for  $f^r$ . We should have a subgradient  $s_{x^{(i)}}$  better than  $\nabla f(x^{(i)})$  so that each  $g_{x^{(i)}}$  is as “flat” as possible.

For simplicity, let us write  $x^{(i)}$ ,  $s_{x^{(i)}}$  and  $g_{x^{(i)}}$  as  $\bar{x}$ ,  $s$  and  $g$  respectively, and  $g(x) = f^r(\bar{x}) + s^T(x - \bar{x})$ . Then our goal is to choose an  $s \in \partial f^r(\bar{x})$  such that  $g(x)$  is as “close” as possible to  $f^r(x)$  at all  $x \in B^n$ . The way we try to achieve this is to improve a given supporting plane successively, starting from the supporting plane with  $s = \nabla f(\bar{x})$ . That is,  $s$  is first set to  $\nabla f(\bar{x})$ . Then it is updated such that the corresponding supporting plane is “lifted” in the sense that it is “flatter”, or in other words, “closer” to  $f^r$ . The updated  $s$  remains a subgradient as long as the corresponding new hyperplane still supports  $f^r$ , i.e.,

$$g(x) \leq f^r(x) \quad \forall x \in B^n. \quad (3.1)$$

The “lifting” process continues until the best possible supporting plane is obtained. But the problem here is that we can not check the condition (3.1) directly because it involves  $2^n$  function evaluations. We proceed as follows.

For  $s \in R^n$ , consider  $S$  such that  $x \in S$  if  $f(x) \leq g(x)$ . We call  $S$  the projection set of  $s$  with respect to  $f$  at  $\bar{x}$ . Geometrically, if  $f$  is a convex function,  $S$  is the convex hull of the projection on  $R^n$  of the intersection of  $g$  and  $f$ . Now the condition (3.1) can be restated as follows:

- *For any  $s \in R^n$ ,  $s$  is a subgradient of  $f^r$  if and only if the interior of its projection set  $S$  does not contain 0–1 integer points.*

This statement implies that to update a subgradient  $s$  to a new  $s$  that remains a subgradient, we only need to maintain the corresponding  $S$  to be a special convex body that does not contain any interior points in  $B^n$ . In other words, given a new updated  $s$ , we can determine if  $s$  is a subgradient by checking if the corresponding  $S$  is a special integer lattice-free convex body. How to determine lattice-free convex bodies has been studied by quite a few people (see Grötschel, Lovász and Shrijver [1987], Kannan [1987] and Lovász [1986], [1989]), but a general algorithm to do so, if we can get one, can be expensive. This leaves us an interesting further research topic, which we will discuss in Chapter 7, for the improvement of our algorithm, but we take a different approach in the current version of our algorithm. What we really do is explained by a simple example shown in Figure 3.1. Generally speaking, we update the subgradient to improve the supporting plane in such a way that to check if the interior of a given  $S$  does not contain  $0-1$  points becomes more tractable and less expensive.

Now let us look at the example shown in Figure 3.1. The continuous objective function  $f$  is differentiable and strictly convex, and  $\bar{x}$  is such that  $\bar{x}_i = 1, \forall i$ . Let  $A = \{x \in R^n : x_i \geq 0, \forall i\}$ .

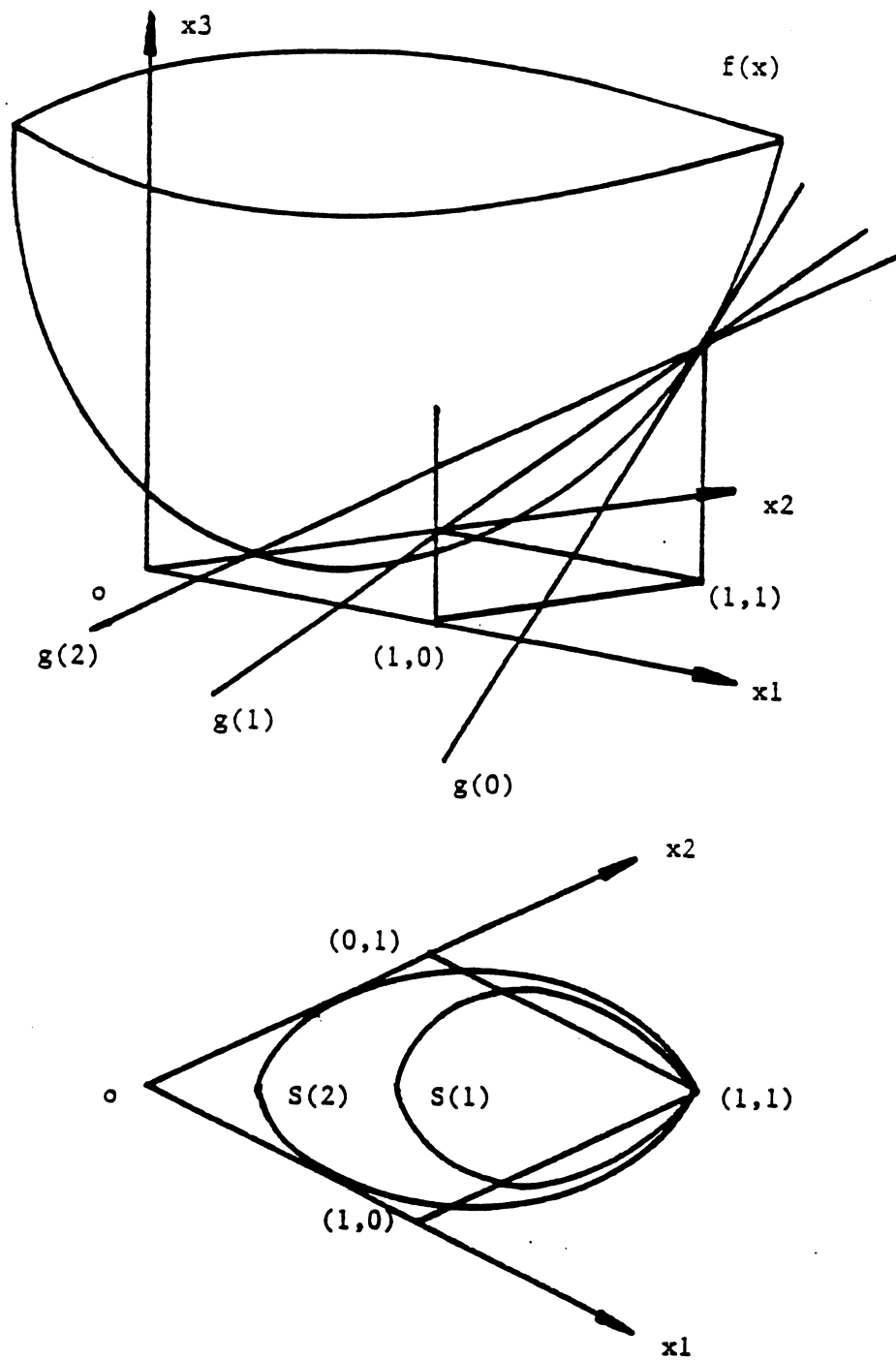


Figure 3.1 "Lifting" process for computing subgradients.

First, for  $s = \nabla f(\bar{x})$ , we have the supporting plane  $g_{(0)}$ . Updating  $s$  to “lift”  $g_{(0)}$  a little bit, we obtain the supporting plane  $g_{(1)}$  and projection set  $S_{(1)}$ . We know the updated  $s$  remains a subgradient as long as  $S_{(1)}$  does not contain 0–1 points in its interior. We maintain this property for  $S_{(1)}$  by keeping  $S_{(1)}$  inside of  $A$ . We observe that

- *For any projection set  $S$  obtained in the above “lifting” process, the interior of  $S$  does not contain 0–1 points as long as  $S$  is contained in  $A$ .*

To obtain better subgradients, we update  $s$  further until we get a supporting plane such that the corresponding projection set  $S$  hits the boundary of  $A$  (see  $g_{(2)}$  and  $S_{(2)}$  in Figure 3.1). The supporting plane finally obtained may not be the best. However, in each “lifting” step, to see if  $s$  still remains a subgradient, we only need to check if  $S$  is contained in  $A$ , which can be done more easily.

Let  $s$  be any updated subgradient with  $S$  the corresponding projection set. We measure the distance  $d_i$  between  $S$  and  $x_i = 0$ , the boundary of  $A$ , for each  $i$ . As we will prove in next section, if the continuous objective function  $f$  of the problem is differentiable and strictly convex, given each  $s$ , there is a group of uniquely determined  $d_i$ ’s, and therefore the function  $d = (d_1, d_2, \dots, d_n)$  is well defined. With this result, our process to compute subgradients can be formulated mathematically as solving an optimization problem:

$$\begin{aligned} \min \quad & \|d(s)\| \\ \text{st.} \quad & d_i(s) \geq 0 \quad i = 1, 2, \dots, n \end{aligned} \tag{3.2}$$

This problem can be attacked using some nonlinear continuous optimization methods. It is also not hard to solve in the sense that for our particular purpose, a good feasible solution is what we really want while the optimal solution is not that important.

Now there remains the problem of how to compute  $d(s)$  for each  $s$ . If  $f$  is differentiable and strictly convex, and  $S$  is closed and bounded, then we can compute extreme points of  $S$  along all  $x_i$  directions. Then  $d_i(s)$  can be obtained by calculating the distance between an extreme point of  $S$  along  $x_i$  direction and the boundary  $x_i = 0$  of  $A$ . The problem to compute an extreme point for  $S$  can be formulated as a continuous optimization problem with a linear objective function and nonlinear constraints. As we will describe in next chapter, this problem has special structures and it is not very expensive to solve.

Finally, the process for computing subgradients is used also in our algorithm for optimality testing. In each “lifting” step, if a subgradient  $s$  is obtained, then the algorithm checks for  $\bar{x}$  the necessary and sufficient condition stated in Theorem 2.3.5. If the condition is satisfied, the algorithm stops and  $\bar{x}$ , the current iterative point, is an optimal solution by Theorem 2.3.5.

## 3.2 Mathematical Derivations

The last section introduces the method we use to compute subgradients. In this section we give a formal description about the “lifting” process and discuss in detail the formulation of Problem (3.2).

We assume until next section that our problem can be formulated in such a way that its continuous objective function  $f$  is strictly convex. This, at least in principle, can be done for most problems as we will show in next section.

With this assumption, from a given point  $\bar{x} \in B^n$ , our goal is to compute the subgradient that defines a “good” supporting plane for  $f^r$  at  $\bar{x}$ .

Let  $g_{(0)}$  be the supporting plane of  $f^r$  at  $\bar{x}$  such that

$$g_{(0)}(x) = f^r(\bar{x}) + \nabla f(\bar{x})^T(x - \bar{x}).$$

**Definition 3.2.1** For any  $i > 0$ , let  $g_{(i-1)}$  be a supporting plane of  $f^r$  at  $\bar{x}$  and  $g_{(i)}$  the supporting plane of  $f^r$  obtained by updating the gradient of  $g_{(i-1)}$ . Then,  $g_{(i)}$  is said to be lifted from  $g_{(i-1)}$  if  $g_{(i)}(x) \geq g_{(i-1)}(x) \quad \forall x \in B^n$  and there exists at least one point  $x \in B^n$  such that  $g_{(i)}(x) > g_{(i-1)}(x)$ .

**Definition 3.2.2** For any  $s \in R^n$ , the following set

$$S = \{x \in R^n : f(x) \leq g(x)\}$$

is called the projection set of  $s$  on  $R^n$  with respect to the function  $f$  at  $\bar{x}$ , where  $g$  is defined such that  $g(x) = f^r(\bar{x}) + s^T(x - \bar{x})$ .

**Theorem 3.2.1** For any  $s \in R^n$  and convex function  $f$ , the projection set  $S$  of  $s$  with respect to  $f$  at  $\bar{x}$  is convex.

**Proof:** Let  $x, x' \in S$ . We show

$$\lambda x + (1 - \lambda)x' \in S \quad \forall \lambda \in [0, 1].$$

This follows immediately from

$$\begin{aligned}
 f(\lambda x + (1 - \lambda)x') &\leq \lambda f(x) + (1 - \lambda)f(x') \\
 &\leq \lambda g(x) + (1 - \lambda)g(x') \\
 &= g(\lambda x + (1 - \lambda)x'),
 \end{aligned}$$

where the first inequality is from the fact that  $f$  is convex, the second inequality is from that  $x, x' \in S$  and the equality holds because  $g$  is linear.  $\square$

**Theorem 3.2.2** For any  $s \in R^n$ ,  $s \in \partial f^r(\bar{x})$  if and only if

$$x \notin S^\circ \quad \forall x \in B^n$$

where  $S^\circ$  is the interior of the projection set  $S$  of  $s$  with respect to  $f$  at  $\bar{x}$ .

**Proof:** First we prove the sufficiency:

If  $x \notin S^\circ \quad \forall x \in B^n$ ,  $f(x) \geq g(x) \quad \forall x \in B^n$ , where  $g(x) = f^r(\bar{x}) + s^T(x - \bar{x})$ . This is equivalent to

$$f^r(x) \geq f^r(\bar{x}) + s^T(x - \bar{x}) \quad \forall x \in B^n.$$

So,  $s \in \partial f^r(\bar{x})$  by the definition of a subgradient.

Now we prove the necessity:

If  $s \in \partial f^r(\bar{x})$ ,

$$s^T(x - \bar{x}) \leq f^r(x) - f^r(\bar{x}) \quad \forall x \in B^n.$$

So,

$$g(x) \leq f^r(x) = f(x) \quad \forall x \in B^n,$$

implying that  $x \notin S^\circ \quad \forall x \in B^n$ .  $\square$

As introduced in the last section, computing a subgradient in our algorithm is done by a “lifting” process: starting from the supporting plane  $g_{(0)}$ , the subgradient is updated successively such that the supporting plane is lifted until a good enough supporting plane is obtained. Here, in each step of the process, the algorithm needs to check if the supporting plane is still valid. This is equivalent to checking if the gradient  $s$  for this supporting plane is still a subgradient. Theorem 3.2.2 gives a necessary and sufficient condition to check if  $s \in \partial f^r(\bar{x})$ . Although this condition can not be tested directly, the sufficient condition presented in the following theorem is used in our algorithm.

Define a constant vector  $\bar{c}$  such that each of its components  $\bar{c}_i = 1 - \bar{x}_i \quad i = 1, 2, \dots, n$ . Let  $A$  be a set of  $x \in R^n$  such that

$$x_i \leq \bar{c}_i \quad \forall i \text{ such that } \bar{c}_i = 1$$

and

$$x_i \geq \bar{c}_i \quad \forall i \text{ such that } \bar{c}_i = 0.$$

**Theorem 3.2.3** For any  $s \in R^n$ ,  $s \in \partial f^r(\bar{x})$  if  $S \subseteq A$ , where  $S$  is the projection set of  $s$  with respect to  $f$  at  $\bar{x}$ .



**Proof:** It is easy to see that  $\bar{x}$  is the only point in  $B^n$  contained in  $A^\circ$ , the interior of  $A$ . Since  $S \subseteq A$ ,  $\bar{x}$  is also the only possible point in  $B^n$  that can be contained in  $S^\circ$ , the interior of  $S$ . But  $\bar{x}$  is a boundary point of  $S$ . So,  $x \notin S^\circ \quad \forall x \in B^n$  and  $s \in \partial f^*(\bar{x})$  by Theorem 3.2.2.  $\square$

To verify if a given projection set  $S$  is contained in  $A$ , for  $S$ , a closed and bounded convex body, we look at the distance between  $S$  and the boundary of  $A$  along each direction  $x_i$ ,  $i = 1, 2, \dots, n$ . Let the distance be  $d_i$ ,  $i = 1, 2, \dots, n$ . Then  $S \subseteq A$  if and only if  $d_i$ 's are nonnegative. Let  $d = (d_1, d_2, \dots, d_n)$ . Then  $d$  depends on  $s \in R^n$  that defines the projection set  $S$ . The "lifting" process to compute a subgradient in our algorithm is formulated as the problem:

$$\begin{aligned} \min \quad & \| d(s) \| \\ \text{st.} \quad & d_i(s) \geq 0 \quad i = 1, 2, \dots, n, \end{aligned} \tag{3.3}$$

where  $d(s)$  is the distance vector that depends on  $s$ .

For any  $i = 1, 2, \dots, n$ , the distance  $d_i(s)$  is computed by first finding an extreme point of the convex body  $S$  along the  $x_i$  coordinate direction and then calculating the distance between this extreme point and  $x_i = \bar{c}_i$ , the boundary of  $A$ .

Problem (3.3) can be attacked by standard continuous optimization techniques and the work for finding an extreme point of a convex body in computing  $d_i(s)$  can be done by solving a problem

$$\begin{aligned}
\min \quad & x_i - 2\bar{c}_i x_i \\
\text{st.} \quad & x \in S,
\end{aligned} \tag{3.4}$$

or equivalently,

$$\begin{aligned}
\min \quad & x_i - 2\bar{c}_i x_i \\
\text{st.} \quad & f(x) - g(x) \leq 0,
\end{aligned} \tag{3.5}$$

where  $f$  is the continuous objective function of our problem and  $g$  is a linear function defined by  $g(x) = f^r(\bar{x}) + s^T(x - \bar{x})$ .

Problem (3.5) is not a very hard problem. Its objective function is linear and there is only one nonlinear constraint. As we will show below, the solution of this problem is unique and the first order necessary condition is also sufficient. Therefore the problem can be solved actually by solving only a system of nonlinear equations. More details about solving this problem as well as Problem (3.3) are discussed in the next chapter.

In the remainder of this section, we show that given any  $s$ , the function  $d(s)$  in Problem (3.3) exists and well defined. For each  $i$  and  $s$ , let  $y_i(s)$  be the optimal value of Problem (3.5) for  $s$ . Then it suffices to show that the function  $y$  with  $y = (y_1, y_2, \dots, y_n)$  always exists and well-defined for  $s$ .

For the simplicity, assume in the following statements that  $\bar{x}_i = 1 \forall i$ . Then Problem (3.5) is reduced to

$$\begin{aligned} \min \quad & x_i \\ \text{st.} \quad & f(x) - g(x) \leq 0 \end{aligned} \tag{3.6}$$

where  $f$  and  $g$  are defined as in Problem (3.5).

**Lemma 3.2.1** Given  $s \in \partial f^r(\bar{x})$  and its projection set  $S$  with respect to  $f$  at  $\bar{x}$ , if  $S$  is closed and bounded, Problem (3.6) to compute an extreme point of  $S$  along the  $x_i$  direction has a unique solution.

**Proof:** The existence can be proved by the fact that the objective function is continuous and  $S$  is closed and bounded.

Now we prove the uniqueness. The proof is by contradiction.

Suppose  $x^i$  and  $\hat{x}^i$  are both solutions of Problem (3.6), then  $z^i = \lambda x^i + (1 - \lambda)\hat{x}^i$  for any  $\lambda \in (0, 1)$  is also a solution, because  $\hat{x}_i^i = x_i^i$  and  $z_i^i = \lambda x_i^i + (1 - \lambda)\hat{x}_i^i = \lambda x_i^i + (1 - \lambda)x_i^i = x_i^i$ .

However, since  $f$  is strictly convex and  $x^i, \hat{x}^i \in S$ ,

$$\begin{aligned} f(z^i) &< \lambda f(x^i) + (1 - \lambda)f(\hat{x}^i) \\ &\leq \lambda g(x^i) + (1 - \lambda)g(\hat{x}^i) \\ &= g(z^i) \end{aligned}$$

which implies that  $z^i$  is an interior point of  $S$ . This is a contradiction to the fact that any solution of Problem (3.6) is an extreme point of  $S$ .  $\square$

**Lemma 3.2.2** For Problem (3.6), define Lagrangian function

$$l^i(x; u^i) = x_i + u^i(f(x) - g(x))$$

where  $u^i$  is a scalar. Then, for any  $i$ ,  $0 \leq i \leq n$ , a necessary and sufficient condition for  $x^i$  to be an optimal solution to Problem (3.6) is that  $\exists u^i \geq 0$  such that

$$\nabla_x l^i(x^i; u^i) = 0$$

$$u^i(f(x^i) - f(\bar{x}) - s^T(x^i - \bar{x})) = 0$$

$$f(x^i) - f(\bar{x}) - s^T(x^i - \bar{x}) \leq 0,$$

which, with  $\nabla_x l^i(x^i; u^i)$  written explicitly, is equivalent to

$$u^i(f'_{x_1}(x^i) - s_1) = 0$$

$$u^i(f'_{x_2}(x^i) - s_2) = 0$$

$$\vdots$$

$$u^i(f'_{x_{i-1}}(x^i) - s_{i-1}) = 0$$

$$1 + u^i(f'_{x_i}(x^i) - s_i) = 0 \tag{3.7}$$

$$u^i(f'_{x_{i+1}}(x^i) - s_{i+1}) = 0$$

$$\vdots$$

$$u^i(f'_{x_n}(x^i) - s_n) = 0$$

$$u^i(f(x^i) - f(\bar{x}) - s^T(x^i - \bar{x})) = 0$$

$$(f(x^i) - f(\bar{x}) - s^T(x^i - \bar{x})) \leq 0$$

**Proof:** The necessity follows directly from the first order necessary condition for a nonlinear constrained optimization problem.

Note the  $i$ th equation of (3.7) implies  $u^i > 0$  and  $\nabla_{\bar{x}}^2 l^i(x^i; u^i) = u^i \nabla^2 f(x^i)$  is positive definite. So, the necessary condition is also sufficient by the second order sufficient condition for a nonlinear constrained optimization problem.  $\square$

**Lemma 3.2.3** Given  $\hat{s} \in R^n$  and its projection set  $\hat{S}$  with respect to  $f$  at  $\bar{x}$ , let  $\hat{x}^i$  be an extreme point of  $\hat{S}$  along the  $x_i$  coordinate direction computed by solving Problem (3.6), then  $\exists$  a neighborhood  $N(\hat{s}, \epsilon)$  of  $\hat{s}$  and a function  $x^i : R^n \longrightarrow R^n$  continuous and differentiable in  $N(\hat{s}, \epsilon)$  such that

$$\hat{x}^i = x^i(\hat{s}).$$

**Proof:** Rewrite (3.7) in the following way:

$$\begin{aligned}
 F_1(x^i; u^i; s) &= u^i(f'_{x_1}(x^i) - s_1) = 0 \\
 F_2(x^i; u^i; s) &= u^i(f'_{x_2}(x^i) - s_2) = 0 \\
 &\vdots \\
 F_{i-1}(x^i; u^i; s) &= u^i(f'_{x_{i-1}}(x^i) - s_{i-1}) = 0 \\
 F_i(x^i; u^i; s) &= 1 + u^i(f'_{x_i}(x^i) - s_i) = 0 \\
 F_{i+1}(x^i; u^i; s) &= u^i(f'_{x_{i+1}}(x^i) - s_{i+1}) = 0 \\
 &\vdots
 \end{aligned} \tag{3.8}$$

$$F_n(x^i; u^i; s) = u^i(f'_{x_n}(x^i) - s_n) = 0$$

$$H(x^i; u^i; s) = (f(x^i) - f(\bar{x}) - s^T(x^i - \bar{x})) = 0$$

where the last equality holds because  $u^i > 0$  from the  $i$ th equality and

$$u^i(f(x^i) - f(\hat{x}) - s^T(x^i - \hat{x})) = 0.$$

So the inequality in (3.7) is removed.

Let  $F = (F_1, F_2, \dots, F_n, H)$ . Then (3.8) is equivalent to

$$F(x^i; u^i; s) = 0$$

By Lemmas 3.2.1, 3.2.2, given  $\hat{s}$ ,  $\exists \hat{x}^i$  and  $\hat{u}^i$  such that  $F(\hat{x}^i; \hat{u}^i; \hat{s}) = 0$ . And  $\hat{x}^i$ ,  $\hat{u}^i$  are also unique.

Differentiate  $F$  with respect to  $x^i$  and  $u^i$ ,

$$\nabla_{(x^i; u^i)} F^T = \begin{pmatrix} \nabla_{(x^i; u^i)} F_1 \\ \nabla_{(x^i; u^i)} F_2 \\ \vdots \\ \nabla_{(x^i; u^i)} F_n \\ \nabla_{(x^i; u^i)} H \end{pmatrix}$$

$$\begin{aligned}
&= \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} u^i & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_i} u^i & \dots & \frac{\partial^2 f}{\partial x_1 \partial x_n} u^i & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial^2 f}{\partial x_i \partial x_1} u^i & \dots & \frac{\partial^2 f}{\partial x_i \partial x_i} u^i & \dots & \frac{\partial^2 f}{\partial x_i \partial x_n} u^i & f'_{x_i} - s_i \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} u^i & \dots & \frac{\partial^2 f}{\partial x_n \partial x_i} u^i & \dots & \frac{\partial^2 f}{\partial x_n \partial x_n} u^i & 0 \\ 0 & \dots & f'_{x_i} - s_i & \dots & 0 & 0 \end{pmatrix} \\
&= \begin{pmatrix} u^i \nabla^2 f & \nabla h \\ \nabla h^T & 0 \end{pmatrix}
\end{aligned}$$

where  $\nabla h = (0, \dots, f'_{x_i} - s_i, 0, \dots, 0)^T$ .

Using the fact that  $\nabla^2 f$  is positive definite and  $\nabla h \neq 0$ , since  $f'_{x_i} - s_i = -1/u^i$ , it is not difficult to prove  $\nabla_{(x^i; u^i)} F$  is nonsingular at  $(\hat{x}^i; \hat{u}^i; \hat{s})$ . So by the implicit function theorem, the lemma is proved.  $\square$

From Lemmas 3.2.1, 3.2.2, 3.2.3, we can get the following theorem,

**Theorem 3.2.4** Given  $\hat{s} \in R^n$  and its projection set  $\hat{S}$  with respect to  $f$  at  $\bar{x}$ , let  $\hat{x}^i$  be an extreme point of  $\hat{S}$  along the  $x_i$  coordinate direction computed by solving Problem (3.6), and let  $y = (y_1, y_2, \dots, y_n)^T$  and  $\hat{y} = (\hat{x}_1^1, \hat{x}_2^2, \dots, \hat{x}_n^n)^T$ . Then,  $\exists$  a neighborhood  $N(\hat{s}, \epsilon)$  of  $\hat{s}$  and a function  $y : R^n \longrightarrow R^n$  continuous and differentiable in  $N(\hat{s}, \epsilon)$  such

that

$$\hat{y} = y(\hat{s}).$$

**Proof:** Simply set  $y_i(s) = x_i^i(s)$  as derived in Lemma 3.2.3.  $\square$

### 3.3 Extension to General Problems

Now, let us consider a problem (1.1) whose continuous objective function  $f$  is non-convex. One way to deal with this situation is given in the following algorithm. The idea here is to first perturb the function  $f$  with a penalty function to make it strictly convex, and then apply the method discussed in previous sections.

In Algorithm 3.3.1,  $q(x) = 0 \quad \forall x \in B^n$ , implying  $\tilde{f}$  and  $f$  agree on all  $x \in B^n$ . So, using  $\tilde{f}$  as the continuous objective function does not change the discrete objective function  $f^r$  and gives the same solution to our problem as using  $f$ . Since  $q$  is strictly convex,  $\tilde{f}$  is strictly convex when  $\rho$  is sufficiently large. Therefore, the method discussed in previous sections can apply after  $f$  is replaced by  $\tilde{f}$ .

Algorithm 3.3.1 shows theoretically that most problems can be reformulated so that their continuous objective functions are strictly convex. But in practice, choosing  $\rho$  in the algorithm appropriately is still a problem and requires further research.



**Algorithm 3.3.1** *{Computing subgradients for general problems}*

0 *{Define the function  $q : R^n \longrightarrow R$ }*

$$q(x) = \sum_{i=1}^n (x_i - 1/2)^{2k} - n/4^k \quad \text{for some } k$$

1 *{Choose  $\rho \geq 0$  and update  $f$ }*

$$\tilde{f}(x) = f(x) + \rho q(x)$$

where  $\rho = 0$  if  $f$  is strictly convex and  $\rho > 0$  otherwise.

2 *{Solve the problem with  $f$  replaced by  $\tilde{f}$ }*

...

□

## Chapter 4

# The Continuous Optimization Subproblems

### 4.1 Nonlinear Least Square

In this chapter, we discuss algorithmic details for solving two classes of subproblems introduced in last chapter for computing subgradients. They are formulated typically as in Problem 3.3 and Problem 3.5 respectively. If we choose the  $l_2$  norm, a problem of the first class becomes a nonlinear least square problem. Two methods are described in this section for this special class of nonlinear least square problems. Section 4.2 explores structures for the second class of problems and discusses its efficient solution.

#### 4.1.1 Backtracking

As discussed in the last chapter, we do not have to solve Problem 3.3 exactly. As a matter of fact, a “good” subgradient can be obtained by only solving either of following two relaxations of Problem 3.3:

1. keep the feasibility while making the objective function as small as possible but not necessarily optimal;
2. minimize the objective function while keeping the amount of infeasibility as small as possible but not necessarily zero.

In this section, we discuss how to solve the first relaxation. The method we use for the second relaxation is described in Section 4.1.2.

Algorithm 4.1.1 is used to solve the first relaxation. In the first step of the algorithm,  $s$  is set to an initial value. In the second step, a “better”  $s$  is computed by adjusting back and forth each component of  $s$ . If  $d(s) \geq 0$ ,  $s$  is adjusted such that the corresponding supporting plane can be “lifted”. Otherwise, some components of  $s$  are adjusted to “lower” the supporting plane. The algorithm can stop whenever a good enough feasible  $s$  is obtained. Furthermore, anytime  $s$  is infeasible, the algorithm actually starts a generalized bisection procedure which can eventually converge to a feasible  $s$ .

The disadvantage of the algorithm is that it may converge slowly and in a limited time, it can only provide a relatively “good” solution.

## Algorithm 4.1.1

```

0 {Initialization}

  for  $i = 1, \dots, n$  do

    set initial values for  $s_i$ ,  $\underline{s}_i$ , and  $\bar{s}_i$ 

1 {Updating}

  if  $d_i(s) \geq 0 \ \forall i$  then

    if  $\|d(s)\|$  small enough, stop

    for  $i = 1, \dots, n$  do

       $\underline{s}_i = s_i$ 

       $s_i = s_i + (\bar{s}_i - s_i)/2$ 

    else

      for  $\forall i$  such that  $d_i(s) < 0$  do

         $\bar{s}_i = s_i$ 

         $s_i = s_i - (s_i - \underline{s}_i)/2$ 

2 {Backtracking}

  goto 1

□

```

### 4.1.2 $\epsilon$ -Approximation

Another approach to Problem 3.3 is to solve the problem without considering the constraints. The problem becomes

$$\min \|d(s)\| \quad (4.1)$$

which, for the case of  $l_2$  norm, can be solved by using any standard methods for nonlinear least square problems (see Dennis and Schnabel [1983]).

Let  $s^*$  be the solution to Problem 4.1 and  $\epsilon = \|d(s^*)\|$  be the optimal value. Then we say  $s^*$  is an  $\epsilon$ -approximation to the solution for Problem 3.3 in the sense that it solves exactly the following problem:

$$\begin{aligned} \min \quad & \|d(s)\| \\ \text{st.} \quad & d_i(s) + \alpha\epsilon \geq 0 \quad i = 1, \dots, n \end{aligned} \quad (4.2)$$

where  $\alpha$  is a positive constant.

With this approximation, the total amount of infeasibility caused by  $s^*$  is always bounded by a quantity in order of  $O(\epsilon)$ . The smaller the  $\epsilon$ , the better  $s^*$  might be.

## 4.2 Nonlinear Constrained Optimization

In this section, we discuss two issues in solving the constrained optimization problems introduced in Section 3.2 for computing extreme points of a convex body. Typically, a problem of this kind is formulated as follows:

$$\begin{aligned} \min \quad & x_i \\ \text{st.} \quad & f(x) - g(x) \leq 0 \end{aligned} \tag{4.3}$$

where  $i = 1, 2, \dots$  or  $n$ , and  $f(x)$  and  $g(x)$  are as defined in Section 3.2.

Solving a constrained optimization problem can be expensive. But Problem 4.3 has its own structure. It is a problem with a linear objective function and there is only one nonlinear constraint. As shown in Section 3.2, this problem can be solved by equivalently solving a system of nonlinear equations:

$$F(x; u) = \begin{pmatrix} \nabla_x l(x; u) \\ h(x) \end{pmatrix} = 0 \tag{4.4}$$

where  $l(x; u)$  is the Lagrangian function of Problem 4.3,  $u$  is the Lagrangian multiplier and  $h(x) = f(x) - g(x)$ .

Note that the Jacobian of function  $F(x; u)$  is

$$\nabla F(x; u)^T = \begin{pmatrix} \nabla_x^2 l(x; u) & \nabla h(x) \\ \nabla^T h(x) & 0 \end{pmatrix} \tag{4.5}$$

and  $\nabla_x^2 l(x, u)$  is symmetric positive definite (see the proof for Lemma 3.2.2). So, to solve system (4.4), we can apply the Quasi-Newton method with structural BFGS updating (see Tapia [1988] and Dennis, Martinez and Tapia [1989]). Also, in computing the Newton step, we can take advantage of this special property to solve each linear system more efficiently. We discuss all these issues in the following sections.

#### 4.2.1 Structural BFGS Updating

By a secant method for solving the system of nonlinear equations:

$$F(x) = 0$$

where  $F : R^n \longrightarrow R^n$ , we mean the iterative procedure

$$Bs = -F(x)$$

$$x_+ = x + s$$

$$B_+ = B(x, s, y, B)$$

where  $s$  is the Quasi-Newton step,  $y = F(x_+) - F(x)$  and  $B_+$  is required to satisfy the secant equation

$$B_+s = y.$$

$B_+$  is the approximation to the first order information for  $F(x_+)$ . It is obtained by updating  $B$  with a process called a secant update. Among various kinds of secant updates, the BFGS update is in some sense the most effective one. However it requires the Jacobian of  $F(x)$  to be symmetric positive definite.

Often, in practice, a part of the first order information is available and we need only to approximate the remaining part. This kind of secant approximation is referred to as the structural secant update, for the special structure of the problem is taken into account. The structural BFGS update approximates the unknown part of the first order information using the BFGS update and computes the available part exactly.

Now let us consider system (4.4). Part of its first order information  $\nabla h(x)$  can be computed exactly, while  $\nabla_x^2 l(x; u)$  needs to be approximated. Since  $\nabla_x^2 l(x; u)$  is symmetric positive definite, we can apply to it the structural BFGS update. So, the secant method for solving system (4.4) can be formulated as the following iterative procedure:

$$\begin{aligned}
 &\text{Given } B = \begin{pmatrix} B_l & \nabla h(x) \\ \nabla h(x)^T & 0 \end{pmatrix}, \\
 &\text{solve } Bs = -F(x; u); \\
 &\text{set } (x_+; u_+) = (x; u) + s; \\
 &\text{set } B_{l+} = B_l + \frac{yy^T}{y^T s} - \frac{B_l s s^T B_l}{s^T B_l s}, \\
 &B_+ = \begin{pmatrix} B_{l+} & \nabla h(x_+) \\ \nabla h(x_+)^T & 0 \end{pmatrix}.
 \end{aligned}$$

#### 4.2.2 Solving Partially Positive Definite Systems

Note that in the secant method for solving system (4.4), there is a linear system to be solved in each iteration:

$$Bs = -F \tag{4.6}$$



where

$$B = \begin{pmatrix} B_l & \nabla h \\ \nabla h^T & 0 \end{pmatrix}$$

and  $B_l$  is symmetric positive definite. We can solve this system using the Cholesky factorization in the following way. Remember that  $\nabla h \in R^n$ .

First, let  $s = (x; \alpha)^T$  and  $-F = (y; \beta)^T$  where  $x, y \in R^n$  and  $\alpha$  and  $\beta$  are scalars.

Rewrite the system as

$$B_l x + \nabla h \alpha = y$$

$$\nabla h^T x = \beta.$$

Solve this system for  $x$  and  $\alpha$  as follows:

$$x = B_l^{-1}(y - \nabla h \alpha)$$

$$\alpha = \frac{\nabla h^T B_l^{-1} y - \beta}{\nabla h^T B_l^{-1} \nabla h}$$

This is equivalent to:

$$B_l a = \nabla h \tag{4.7}$$

$$B_l b = y \tag{4.8}$$

$$x = b - \nabla h \alpha \tag{4.9}$$

$$\alpha = \frac{\nabla h^T b - \beta}{\nabla h^T a} \tag{4.10}$$

Note that  $B_l$  is symmetric positive definite. So it can be decomposed with the Cholesky factorization.

System (4.6) can be solved by using formula (4.7) to (4.10). The total computation is in the order of  $O(n^3/6)$ , which is only half of the work for solving the system directly with  $LU$  factorization.

## Chapter 5

### On Solving Integer Minimax Subproblems

#### 5.1 The General Branch and Bound Procedure

In this chapter, we discuss how to solve the integer minimax subproblem in Algorithm

2.2.1. The problem is formulated as

$$\min_{x \in B^n} \{p^{(i)}(x) = \max \{g_{x(j)}(x), j = 0, \dots, i\}\} \quad (5.1)$$

and is equivalent to

$$\min \quad \eta \quad (5.2)$$

$$\text{st.} \quad \eta \geq g_{x(j)}(x) \quad j = 0, \dots, i \quad (5.3)$$

$$1 \geq x \geq 0, \quad x \text{ integral} \quad (5.4)$$

where  $g_{x(j)}$  is the  $j$ th linear supporting function generated by the algorithm and  $i$  indicates that the problem is the one in the  $i$ th iteration.

The problem above is a linear integer programming problem with only one continuous variable and can be solved with an enumeration procedure (see Balas [1970]).

Also, one may observe that  $p^{(i+1)}$  is generated by adding one more supporting plane to  $p^{(i)}$ , which implies that problems in the  $i$ th and  $i + 1$ th iterations are almost the

same except the problem in the  $i + 1$ th iteration has one more constraint. So, in solving the  $(i + 1)$ th problem, results from solving the  $i$ th problem can be used to reduce the total computation. In this section, we present a branch-and-bound procedure for the integer minimax problem. In the remaining sections, we discuss the branching strategy and the bounding process.

Consider the problem formulated in (5.2) to (5.4). We solve this problem with a special branch-and-bound procedure as shown in Algorithm 5.1.1. The general idea is the following. First, a relaxed problem; the problem without integrality constraints, is solved. If a 0–1 integer solution  $x$  is obtained, the algorithm terminates with  $x$  optimal. Otherwise  $x_j$  for some  $j$  is set to 1 or 0 and two corresponding subproblems are generated. Recursively, for any subproblem, again, the relaxed problem is solved. If the optimal value of the relaxed problem is larger than some upper bound for the optimal value of the original problem, the subproblem is eliminated and not considered any more. If a 0–1 integer solution is obtained, the solution is locally optimal to the original problem. Otherwise the subproblem is divided and two more subproblems are generated. The process goes until all subproblems are either eliminated or solved. Among all local solutions obtained, the optimal solution is the one that yields the smallest objective value.

Algorithm 5.1.1 *{Solving Integer Minimax Problems}*

0 *{Initialization}*

$p = (\wedge, \wedge, \dots, \wedge)$ ,  $P = \phi$ , push( $p, P$ )

$\underline{z}_p^{(i)} = -\infty$ ,  $\bar{z}^{(i)} = \min_{0 \leq j \leq i} \{f(x^{(j)})\}$

1 *{Iteration}*

do while  $P \neq \phi$

1.1 *{Problem selection and relaxation}*

solve  $p = \text{pop}(P)$

let  $\underline{z}_p^{(i)}$  be the optimal value

let  $x_p^{(i)}$  be the optimal solution

1.2 *{Prunning}*

if  $\underline{z}_p^{(i)} \geq \bar{z}^{(i)}$ , go to next loop

if  $x_p^{(i)}$  is integral,  $\bar{z}^{(i)} = \min(\bar{z}^{(i)}, \underline{z}_p^{(i)})$  and go to next loop

1.3 *{Branching}*

pick up  $x_j$  for some  $j$  with  $p_j = \wedge$

set  $p_j = 0$ , push( $p, P$ )

set  $p_j = 1$ , push( $p, P$ )

end do

2 *{Termination}*

the solution  $x_p^{(i)}$  for some  $p$  that yields  $\bar{z}^{(i)}$  is optimal

□

Define a relaxed problem as follows:

$$\begin{aligned}
 \min \quad & \eta \\
 \text{st.} \quad & \eta \geq g_{x^{(j)}}(x) \quad j = 0, \dots, i \\
 & 1 \geq x \geq 0
 \end{aligned} \tag{5.5}$$

In Algorithm 5.1.1,  $p = (p_1, p_2, \dots, p_n)$  represents the problem that is almost the same as the relaxed problem (5.5) except some  $x_j$ 's are set to 1 or 0, where

$$\begin{aligned}
 p_j &= 1 \quad \text{if and only if} \quad x_j \quad \text{is set to} \quad 1 \\
 p_j &= 0 \quad \text{if and only if} \quad x_j \quad \text{is set to} \quad 0 \\
 p_j &= \wedge \quad \text{otherwise}
 \end{aligned}$$

In Algorithm 5.1.1,  $P$  is a stack, and push and pop are standard stack manipulations.

## 5.2 Branching Strategies

In Algorithm 5.1.1, each time when a local solution is found, an upper bound for the optimal value is obtained. Let the upper bound be denoted by  $\bar{z}^{(i)}$ . Then since a solution that provides an objective value better than  $\bar{z}^{(i)}$  is always desired, the strategy to pick a branching variable is to try the variable that may reduce the current objective value if it is set to 1.

Write problem  $p$  in the following form:

$$\begin{aligned} \min \quad & \eta \\ \text{st.} \quad & \eta \geq b_j + a_{j1}x_1 + \dots + a_{jm}x_m \\ & j = 0, \dots, i \\ & 1 \geq x \geq 0, \end{aligned}$$

assuming  $p_k = \wedge \forall k = 1, \dots, m$ . Then a branching variable  $x_k$  for this problem is chosen such that  $k$  solves the problem:

$$\min_{1 \leq k \leq m} \{ \max_{0 \leq j \leq i} \{ b_j + a_{jk} \} \}.$$

### 5.3 Lower and Upper Bounds

In Algorithm 5.1.1, lower bounds for the optimal value are obtained by solving linear relaxed problems, and upper bounds are the objective values for local solutions obtained.

Linear relaxed problems can be solved by using the dual simplex method. Let  $p^{(i)}$  and  $p^{(i+1)}$  be the relaxed problems for the  $i$ th and  $i+1$ th integer minimax problems. As we mentioned before,  $p^{(i+1)}$  is the same as  $p^{(i)}$  except that it has one more constraint. The dual optimal basis for  $p^{(i)}$  is dual feasible for  $p^{(i+1)}$ . So,  $p^{(i+1)}$  can be solved with the dual optimal basis for  $p^{(i)}$  as its initial basis. Also, if  $P$  is a subproblem generated in Algorithm 5.1.1 and  $S$  is its subsubproblem,  $S$  and  $P$  have the same relationship as  $p^{(i+1)}$  and  $p^{(i)}$ :  $S$  is the same as  $P$  except it contains one more constraint  $x_j = 1/0$

for some  $j$ . Again, to solve  $S$ , the dual optimal basis for  $P$  can be used as the initial basis. In the simplex method, this is called the warm start. It generally speeds up the computation (see Bixby [1990] and Chvátal [1980]).



## Chapter 6

# Parallelization and Numerical Experiments

### 6.1 NIPACK System

NIPACK is a software system developed to implement the subgradient algorithm. The system is written in EXPRESS C and runs on parallel distributed-memory machines. It contains a group of parallel subroutines used for either continuous or discrete optimization. These include subroutines for  $LU$ ,  $QR$ , and Cholesky factorizations, triangular system solvers, the branch-and-bound procedure and so on. In this chapter, we discuss implementation issues such as parallel strategies and performance analysis. The preliminary numerical results for testing our algorithm are presented. Performance results of the system on the NCUBE are given also.

There are 10 program modules in NIPACK, each of which contains a group of subroutines used for a special purpose. Below, we describe briefly the function of each module. The relationship among all these modules is shown in Figure 6.1.

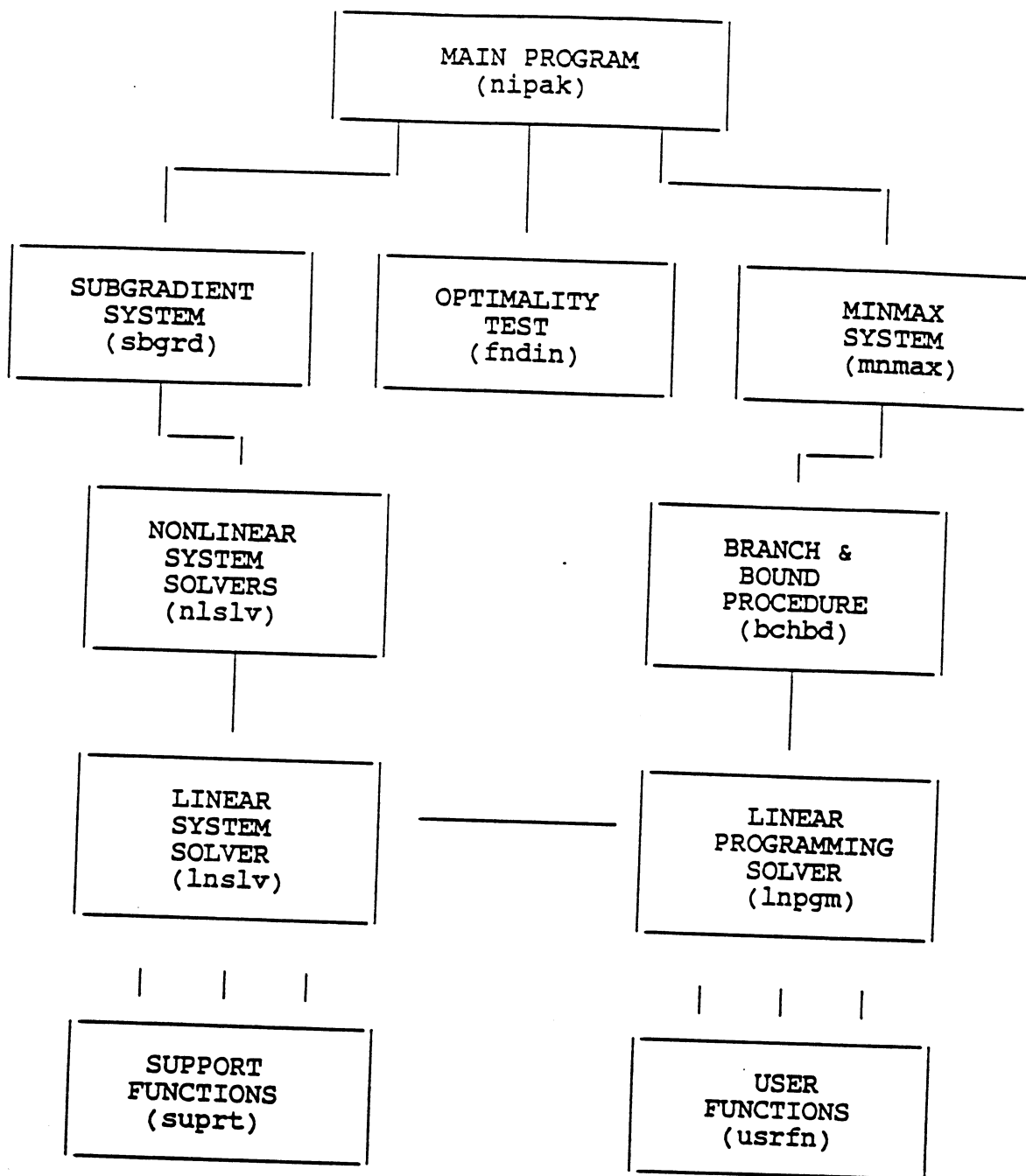


Figure 6.1 NIPACK system structure.

- <module 1> nipak: The main program for the subgradient algorithm. The program asks from users an initial guess, carries out the main loop of the subgradient algorithm, and outputs the optimal solution.
- <module 2> fndin: The program for optimality testing. This program tests if the current iterative point has been looked at before. If it has, the program reports to the main program, and an optimal solution is found.
- <module 3> sbgrd: The program for computing subgradients. The program carries out a "lifting" process, updating subgradients and finally reaching one that is good enough. It also reports to the main program when a subgradient is found that satisfies the necessary and sufficient condition. In this case, an optimal solution actually is determined.
- <module 4> nslsv: The program for solving a system of nonlinear equations. It is used to compute extreme points of a convex body. A Quasi-Newton method with structural *BFGS* updating is used.
- <module 5> lnslv: The program for basic matrix computation. Subroutines for *LU*, *QR*, and Cholesky factorizations as well as lower/upper triangular system solvers are contained.
- <module 6> mnmax: The main program for solving linear integer minimax subproblems. The program formulates the dual problem and calls a branch-and-bound procedure to find the optimal integer solution.

<module 7> bchbd: The program for branch-and-bound enumeration for linear integer programming. It is a recursive procedure, searching the optimal solution around a branching tree and recursively, in each node, solving a linear relaxed problem and generating subproblems.

<module 8> lnpgm: A linear programming solver. It is used in the branch-and-bound procedure for solving each linear relaxed problem. The program implements a simplex algorithm.

<module 9> suprt: Utility subroutines used in the whole system. All high level communication subroutines are contained such as global broadcasting, summation, concatenation, sorting, and finding the processor that contains the minimum element and so on.

<module 10> usrfn: The program used by users to define subroutines for user functions and their gradients.

Programs in NIPACK use  $p$  processors with  $p$  varying from 1 to  $n^2$ , where  $n$  is the problem dimension. All subroutines for matrix computation are restricted to use only less than or equal to  $n$  processors. In the parallel implementation of the subgradient algorithm, these subroutines are called for solving subproblems for which only less than or equal to  $n$  processors are assigned.

In NIPACK, except some I/O functions, only two machine-specific communication functions `exread` and `exwrite` are used. All other high level communication

functions are written by using only these two functions. So, the whole system is quite transportable.

NIPACK assumes in general a processor mesh embeded. The mesh is passed around to all subroutines. Each subroutine takes its own processors mapped from this mesh. For instance, in computing subgradients, there are  $n$  subproblems to be solved in parallel. Generally, the subroutine for solving each subproblem takes one column of processors from the processor mesh. Each subroutine for matrix computation uses only a processor ring, while each of communication subroutines should be efficient for a processor tree.

## 6.2 Basic Subroutines

In this section, we list a group of subroutines in NIPACK that are used frequently. They are divided into two classes, one for high level communications among processors, and the other for the basic matrix computation.

### 6.2.1 Communication Subroutines

The following subroutines are written for high level communications in NIPACK. Each of these subroutines is written in such a way that the communication cost is in the order of  $O(\log(p))$ , where  $p$  is the number of processors.

**<subroutine 1> expass(buf,n,src,num,ndlist):** Broadcasting  $n$  numbers in buf from processor src to num processors listed in ndlist.

<subroutine 2> `exmin(x,min,num,ndlist)`: Comparing `x` in all `num` processors listed in `ndlist` to find the minimum. `min` is set to the processor number that contains the minimum.

<subroutine 3> `exsum(buf,n,num,ndlist)`: Summing up numbers in `buf` among `num` processors listed in `ndlist`. `n` numbers in `buf` are used.

<subroutine 4> `excon(buf0,buf1,n,num,ndlist)`: Concatenating numbers in `buf0` among `num` processors listed in `ndlist`. `n` numbers in `buf0` are used. The concatenated array is sent to `buf1`.

<subroutine 5> `exorder(buf,order,num ndlist)`: Having numbers in `buf` sorted among `num` processors listed in `ndlist`. The order of processor numbers according to the sorted numbers in `buf` is set to `order`.

## 6.2.2 Matrix Computation Subroutines

Matrix computation on parallel distributed-memory machines has been studied extensively in recent years (see Bischof [1988], Geist and Heath [1985], Geist and Romine [1988], Golub and Van Loan [1989], Li and Coleman [1988] and Li and Coleman [1989]). We implemented for distributed-memory machines the parallel algorithms for *LU*, *QR*, and Cholesky factorizations as well as the Li and Coleman algorithm for solving triangular systems. They all are used in the implementation of the sub-gradient algorithm. But they can also be improved further for more general usage.

The algorithms here are basically those presented in Golub and Van Loan [1989]. A processor ring is used for each of these subroutines. The matrix is distributed by rows in a wrap fashion.

<subroutine 1> `lufac(A,ipvt,n,pmesh)`: A parallel  $LU$  factorization subroutine for matrix  $A$ . Both  $L$  and  $U$  are stored in  $A$ . The integer array `ipvt` is the pivoting vector. A row of processors in the processor mesh listed in `pmesh` is used.

<subroutine 2> `qrfac(A,Q,n,pmesh)`: A parallel  $QR$  factorization subroutine for matrix  $A$ . The triangular matrix  $R$  is stored in  $A$  and  $Q$  in  $Q$ . A row of processors in the processor mesh listed in `pmesh` is used.

<subroutine 3> `chfac(A,n,pmesh)`: A parallel  $R^T R$  factorization subroutine for matrix  $A$ . The triangular matrix  $R$  is stored in  $A$ . A row of processors in the processor mesh listed in `pmesh` is used.

<subroutine 4> `ltrsl(L,b,n,pmesh)`: A parallel lower triangular system solver.  $L$  is the lower triangular matrix. The array `b` is the right hand side. The solution is stored in `b`. A row of processors in the processor mesh listed in `pmesh` is used.

<subroutine 5> `utrsl(U,b,n,pmesh)`: A parallel upper triangular system solver.  $U$  is the upper triangular matrix. The array `b` is the right hand side. The

solution is stored in  $b$ . A row of processors in the processor mesh listed in  $p_{\text{mesh}}$  is used.

### 6.3 Parallelization for Computing Subgradients

As presented in Chapter 3 and Chapter 4, in each step of the “lifting” process for computing subgradients in our algorithm,  $n$  subproblems ( $n$  systems of nonlinear equations) need to be solved, where  $n$  is the problem dimension. The parallelization of this work is trivial: since all  $n$  subproblems are independent and have almost the same size, they can be solved in parallel with a good load balance and a low communication cost.

If there are  $p$  processors available with  $p$  dividing  $n$ ,  $n$  subproblems are solved in the following process:

```

for  $i = 1, \dots, n/p$  do
    solve, with processor  $j$ , the  $((i - 1)p + j)$ th problem,
    where  $0 \leq j \leq p$ .

```

If the number of processors  $p$  is greater than the problem dimension  $n$  with  $n$  dividing  $p$ ,  $p/n$  processors are assigned to each of  $n$  subproblems. In this case, the parallelization for solving each subproblem is involved also. Suppose  $p$  processors are arranged into a  $p/n$  by  $n$  process mesh. Then each processor column in the processor mesh is responsible for solving one of  $n$  subproblems.



## 6.4 Parallelization for the Branch-and-Bound Procedure

Branch-and-bound procedures are frequently used in integer programming. Work has been done to parallelize the procedure since the early 1980's (see Pruul [1988], Karp and Zhang [1988] and Rushmeier [1990]).

The simplest way to implement a parallel branch-and-bound procedure is to parallelize directly the sequential algorithm, or particularly for our purpose, the algorithm presented in Chapter 5. Generally, to do this, a global stack is kept in each processor to store all generated subproblems. Each time all processors are available, subproblems in the stack are selected for processors, one for each. Then, all processors solve their own subproblems, generate new subproblems, and update the global stack with new subproblems added. This process continues until all subproblems are solved, i.e., the stack becomes empty. This kind of parallelization requires frequently passing around data among all processors to update the global subproblem stack, which involves a large amount of communication. Expecially, when information such as the optimal basis in the parent node is requested, it is hard for a processor to find out where the information can be obtained.

A better processor scheduling is achieved in our implementation using a multiple branching strategy. Instead of using binary branching as described in the sequential algorithm in Chapter 5, in each step, the branch-and-bound procedure, with  $p$  processors, makes  $p$  branches, in other words, generates  $p$  subproblems, and solves

all of them immediately using  $p$  processors. As seen in below Algorithm 6.4.1, this branching strategy has a number of advantages for a parallel implementation.

1. Each processor does not have to keep a global subproblem stack. In each step of the procedure, each processor can find out easily the subproblem it will solve in next step. A local subproblem stack is kept for each processor, but the size is relatively small.
2. Each time after  $p$  subproblems are solved, the branching is made recursively by first, processor 1, second, processor 2, and so on and so forth. So, whenever  $p$  subproblems are generated in  $p$  processors, all processors know exactly which processor contains the problem that the  $p$  subproblems belong to. So, any information about the old problem can be delivered easily using a global broadcast.
3. While the strategy for choosing branching variables as described in Chapter 5 can still apply, a local best-first strategy can be used for selecting branching subproblems. Each time after  $p$  subproblems are solved, processors can be sorted according to the optimal values they have for the subproblems. Then the processor that has the smallest optimal value can branch first.
4. Finally, The algorithm can be coded into a recursive procedure. The programming structure is simple.

Algorithm 6.4.1 { *A Parallel Branch-and-Bound Algorithm* }

\* { *Initial Procedure* }

initialize  $p$ ,  $\underline{z}_p$ ,  $z$  and  $x$

solve  $p$

let  $\underline{z}_p$  be the optimal value

let  $x_p$  be the optimal solution

if  $x_p$  is integral then

$z = \min\{z, \underline{z}_p\}$

$x = x_p$

stop

push( $p, P$ )

branch-and-bound( $1, x, z$ )

pop( $P$ )

\* { *End of Initial Procedure* }

\* {Recursive Procedure}

branch-and-bound( $i, x, z$ )

  broadcast  $z_p$  from processor  $i$

  if  $z_p \geq z$ , return

  if processor  $\# = i$  then

    select branching variables

    broadcast branching variables

  generate subproblem  $p$

  solve  $p$

  let  $z_p$  be the optimal value

  let  $x_p$  be the optimal solution

  if  $x_p$  is integral then

$z = \min\{z, z_p\}, x = x_p$

  update  $z$  and  $x$  if necessary

  push( $p, P$ )

  for  $j = 1, \dots, \#$  of processors do

    branch-and-bound( $j, x, z$ )

  pop( $P$ )

\* {End of Recursive Procedure}

□

## 6.5 Preliminary Numerical Results

Test Problems:

Problem 1:

Objective function:  $\sum_{i=1}^n x_i^2 - 1.8 \sum_{i=1}^n x_i + 0.81n$

Optimal solution:  $x_i = 1, i = 1, \dots, n$

Problem 2:

Objective function:

$2 \sum_{i=1}^n x_i^2 + \sum_{i=1}^{n-1} x_i x_{i+1} - 2 \sum_{i=1}^{n/2} (1.9x_{2i-1} + 1.1x_{2i}) + 1.205n$

Optimal solution:  $x_{2i-1} = 1, x_{2i} = 0, i = 1, \dots, n/2$

Problem 3:

Objective function:  $\sum_{i=1}^n (x_i - 0.9)^{8/3}$

Optimal solution:  $x_i = 1, i = 1, \dots, n$

Problem 4:

Objective function:  $\sum_{i=1}^n x_i^2 - 0.8 \sum_{i=1}^n x_i + 0.16n$

Optimal solution:  $x_i = 0, i = 1, \dots, n$

### Notations:

dim: ==== Problem dimension.

igs: ==== Initial guess.

itr: ==== Total number of iterations.

fev0: ==== Total number of distinct function evaluations.

fev1: ==== Total number of function evaluations on all processors.

tcp: ==== Total computation time [milliseconds].

tcm: ==== Total communication time [milliseconds].

tt: ==== Total time [minutes:seconds].

pcs: ==== Total number of processors.

NIPACK0: ==== Early version of NIPACK.

NIPACK1: ==== Current version of NIPACK.

Test Problem: Problem 1.

Total # of Feasible Points: 256.

dim	igs	itr	tcm	tcp	tt	pcs
8	(0,0,0,0,0,0,0,0)	2	386	4961	0:11	8
8	(0,0,0,0,0,0,0,1)	7	12013	51918	1:59	8
8	(0,0,0,0,0,0,1,1)	3	1533	16709	0:40	8
8	(0,0,0,0,0,1,1,1)	3	3008	18248	0:46	8
8	(0,0,0,0,1,1,1,1)	2	358	4724	0:11	8
8	(0,0,0,1,1,1,1,1)	3	2558	21697	0:45	8
8	(0,0,1,1,1,1,1,1)	2	861	10268	0:24	8
8	(0,1,1,1,1,1,1,1)	7	11418	51393	1:56	8
8	(1,1,1,1,1,1,1,1)	1	193	2432	0:6	8
8	(0,1,0,1,0,1,0,1)	2	358	4727	0:12	8
8	(0,0,1,1,0,0,1,1)	2	385	4727	0:11	8
8	(0,1,1,1,0,1,1,1)	2	862	10270	0:23	8
8	(0,0,0,1,0,0,0,1)	3	1531	16713	0:42	8

Table 1 (from NIPACK0).

Test Problem: Problem 1.

Total # of Feasible Points: 256.

dim	igs	itr	tcm	tcp	tt	pcs
8	(0,0,0,1,0,0,0,1)	3	637	215392	4:4	1
8	(0,0,0,1,0,0,0,1)	3	3057	107645	2:24	2
8	(0,0,0,1,0,0,0,1)	3	3243	53736	1:32	4
8	(0,0,0,1,0,0,0,1)	3	1531	16713	0:42	8

Table 2 (from NIPACK0).

Test Problem: Problem 1.

Total # of Feasible Points: 65536.

dim	igs	itr	tcm	tcp	tt	pcs
16	(0,...,0)	2	189	323316	5:39	1
16	(0,...,0)	2	934	163497	2:55	2
16	(0,...,0)	2	985	83867	1:35	4
16	(0,...,0)	2	865	44048	0:57	8
16	(0,...,0)	2	751	24133	0:46	16

Table 3 (from NIPACK0).



Test Problem: Problem 1.

Total # of Feasible Points: 4294967296.

dim	igs	itr	tcn	tcp	tt	pcs
32	(0,...,0)	2	231	1826419	31:18	1
32	(0,...,0)	2	1147	927602	16:00	2
32	(0,...,0)	2	1203	478096	8:22	4
32	(0,...,0)	2	1041	253203	4:30	8
32	(0,...,0)	2	886	140831	2:36	16
32	(0,...,0)	2	798	84645	1:44	32
32	(1,...,1)	1	65	587234	10:4	1
32	(1,...,1)	1	335	294125	5:40	2
32	(1,...,1)	1	349	147571	2:35	4
32	(1,...,1)	1	298	74295	1:2	8
32	(1,...,1)	1	250	37658	0:43	16
32	(1,...,1)	1	220	19341	0:25	32

Table 4 (from NIPACK0).

Test Problem: Problem 2.

Total # of Feasible Points: 256.

dim	igs	itr	tcm	tcp	tt	pcs
8	(0,0,0,0,0,0,0,0)	3	473	264212	4:54	1
8	(0,0,0,0,0,0,0,0)	3	2634	134480	3:56	2
8	(0,0,0,0,0,0,0,0)	3	2727	68876	2:36	4
8	(0,0,0,0,0,0,0,0)	3	2673	41156	1:52	8

Table 5 (from NIPACK0).

Test Problem: Problem 2.

Total # of Feasible Points: 65536.

dim	igs	itr	tcm	tcp	tt	pcs
16	(0,1,...,0,1)	2	895	1378573	24:10	1
16	(0,1,...,0,1)	2	4273	693285	12:40	2
16	(0,1,...,0,1)	2	4521	348525	6:43	4
16	(0,1,...,0,1)	2	3993	177684	3:50	8
16	(0,1,...,0,1)	2	3478	92178	2:31	16

Table 6 (from NIPACK0).

Test Problem: Problem 3.

Total # of Feasible Points: 256.

dim	igs	itr	fev0	fev1	tcm	tcp	tt	pcs
8	(0,0,0,0,0,0,0,0)	2	3834	7920	2327	3150	0:6	16
8	(0,0,0,0,0,0,0,1)	2	3348	6948	2282	2905	0:7	16
8	(0,0,0,0,0,0,1,1)	2	3420	7092	2231	3076	0:7	16
8	(0,0,0,0,0,1,1,1)	2	3447	7146	2348	2846	0:6	16

Table 7 (from NIPACK1).

Test Problem: Problem 3.

Total # of Feasible Points: 65536.

dim	igs	itr	fev0	fev1	tcm	tcp	tt	pcs
16	(0,...,0,0,0,0,0,0)	2	60410	60911	13686	61648	1:18	16
16	(0,...,0,0,0,0,0,1)	2	52666	53176	30015	39485	1:12	16
16	(0,...,0,0,0,0,1,1)	2	47685	48195	18262	33927	0:54	16
16	(0,...,0,0,0,1,1,1)	3	56389	57154	20663	40687	1:9	16
16	(0,...,0,0,1,1,1,1)	2	41616	42126	20203	28811	0:55	16
16	(0,...,0,1,1,1,1,1)	2	37944	38454	17176	29135	0:48	16

Table 8 (from NIPACK1).

Test Problem: Problem 3.

Total # of Feasible Points: 4294967296 [dim=32]

Total # of Feasible Points: 18446744073709551616 [dim=64].

dim	igs	itr	fev0	fev1	tcm	tcp	tt	pcs
32	(0,...,0,1)	3	154341	1259280	130784	57475	3:10	256
32	(0,1,...,1)	2	124806	1014816	97930	50032	2:43	256
64	(0,...,0)	2	470210	1913600	71280	455065	9:9	256
64	(1,...,1)	1	220545	898560	32520	55679	1:46	256

Table 9 (from NIPACK1).

Test Problem: Problem 3.

Total # of Feasible Points: 4294967296.

dim	igs	itr	fev0	fev1	tcm	tcp	tt	pcs
32	(0,...,0)	2	122562	124608	2672	98302	3:10	32
32	(0,...,0)	2	122562	249216	20516	72184	2:43	64
32	(0,...,0)	2	122562	498432	32859	57247	9:9	128
32	(0,...,0)	2	122562	996864	43701	49960	1:46	256

Table 10 (from NIPACK1).

Test Problem: Problem 4.

Total # of Feasible Points: 16.

dim	igs	itr	tcn	tcp	tt	pcs
4	(0,0,0,0)	13	2459	12846	0:33	4
4	(0,0,0,1)	11	1772	10305	0:27	4
4	(0,0,1,0)	11	1770	10793	0:27	4
4	(0,0,1,1)	13	2339	12526	0:32	4
4	(0,1,0,0)	11	1679	10864	0:27	4
4	(0,1,0,1)	13	2229	12707	0:32	4
4	(0,1,1,0)	13	2257	12671	0:32	4
4	(0,1,1,1)	11	1742	10923	0:27	4
4	(1,0,0,0)	11	1729	10898	0:29	4
4	(1,0,0,1)	13	2178	12762	0:32	4
4	(1,0,1,0)	13	2170	12693	0:32	4
4	(1,0,1,1)	11	1734	10939	0:28	4
4	(1,1,0,0)	13	2248	12704	0:32	4
4	(1,1,0,1)	11	1739	10905	0:28	4
4	(1,1,1,0)	11	1769	10296	0:28	4
4	(1,1,1,1)	13	2630	13016	0:33	4

Table 11 (from NIPACK0).

### Comments on Numerical Results:

All results in Table 1 to Table 11 are obtained when running our system on the NCUBE.

As shown in Table 1 to Table 10, the test problems are solved effectively by the subgradient algorithm. The total number of iterations, which is equivalent to the number of feasible points the algorithm searched over, is relatively small compared with the total number of feasible points in  $B^n$ . Since improved methods for computing subgradients are used in NIPACK1, Table 8 to Table 10 show that the total number of iterations is even smaller using NIPACK1.

Problem 1 with  $n = 8$  is also tested with each initial guess in Table 1 when the gradient is used as the subgradient in each iteration of the algorithm. With this choice of subgradient, the algorithm was not able to find the optimal solution in a reasonable number of iterations. When the bound for the total number of iterations is set to 25, the algorithm simply run out of 25 iterations. This shows the significance of computing good subgradients by our lifting process.

In Table 8 to Table 10, the total number of function evaluations are given. In each iteration of the algorithm, extra function evaluations actually are taken for computing subgradients. So, it is necessary to see how many function evaluations in total are required. The number can not be too big, since actually  $2^n$  function evaluations suffice to solve the original problem. As shown in Table 8, Problem 3 for  $n = 8$  is solved with too many function evaluations. However, the total number of function

evaluations becomes relatively very small when  $n$  is large, compared with the total number of function evaluations required for enumerating all feasible solutions.

Parallel performance is satisfactory when the number of processors is less than or equal to the problem dimension (see Table 2 to Table 6). However, when the number of processors is much bigger than the problem dimension, the communication cost can be high and the parallel process is not quite efficient as shown in Table 10.

Problem 4 in Table 11 is supposed to be a bad example for our algorithm. The algorithm took many iterations without terminating at the optimal solution. This is because the algorithm failed to find a subgradient at the optimal solution that satisfies the necessary and sufficient condition. The algorithm eventually stopped when the optimal solution was repeated. This problem shows that the “lifting” process in our algorithm should be improved further such that the algorithm can have more chance to find a subgradient at the optimal solution that satisfies the necessary and sufficient condition.

## Chapter 7

### Summary and Further Research

We have presented a subgradient algorithm for nonlinear integer programming. Our approach is based on considering the problem as a nonsmooth problem and using subgradient information to linearize the nonlinear objective function. Our algorithm searches for the solution iteratively among feasible points, and in each iteration, it generates the next iterate by solving the problem for a local piecewise linear model constructed with supporting planes for the objective function at a set of iterates already generated. To determine if an iterate is optimal, two optimality testing criteria have been constructed. One of them is a necessary and sufficient condition for the optimal solution. It is derived from the theory of nonsmooth analysis and used in our algorithm effectively.

We have discussed how to compute subgradients for a special class of nonlinear nonsmooth functions in Chapter 3 and Chapter 4. All related mathematical derivations also are presented. In Chapter 5, we have extended the branch-and-bound method to solve the integer minimax subproblems in our algorithm.



We have also described a parallel software system that tests our approach on a class of nonlinear integer programming problems. The preliminary numerical results show that our algorithm can solve test problems effectively.

The parallel software system is developed on parallel distributed-memory machines and contains a group of parallel subroutines used for either continuous or discrete optimization such as subroutines for matrix factorizations, solving triangular systems, the branch-and-bound enumeration and so on. The parallelization of the algorithm has shown that the algorithm is promising in taking advantage of supercomputing tools to solve large and hard problems.

There are several aspects of our work that may be extended. Although we have shown in Chapter 3 that any problem can be formulated so that its continuous objective function is strictly convex, we have not yet given computationally a practical method to make a given function strictly convex. More work needs to be done.

Our algorithm can be extended for solving mixed-integer nonlinear programs. But we have not studied possible successes and limitations of applying the algorithm to this class of problems.

The parallel software system may also be extended. We would like to consider more general problems such as mixed-integer nonlinear integer programs. This requires the system to be able to handle problems with general nonconvex continuous objective functions and have subroutines for solving general nonlinear continuous optimization problems. Also, the parallel branch-and-bound procedure needs to be

improved further so that larger problems can be solved. The parallel strategy in our implementation can be considered in developing more general purpose branch-and-bound procedures.

The technique in Chapter 3 for computing “good” supporting planes for the objective function can be improved by replacing the “lifting” process with a procedure that can compute special integer lattice-free convex bodies. If such an “oracle” exists, the supporting plane can be “lifted” higher and the algorithm will be faster. The work for finding such an “oracle” has been considered in combining theories in both integer programming and the geometry of numbers (Lovász [1989]).

## Bibliography

- [1] A. V. Aho, J. E. Hopcroft and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.
- [2] S. G. Aki [1989]. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, Englewood Cliffs, NJ.
- [3] E. Balas and J. B. Mazzola [1984a]. *Nonlinear 0–1 Programming: I. Linearization Techniques*. Mathematical Programming 30, 1-21.
- [4] E. Balas and J. B. Mazzola [1984b]. *Nonlinear 0–1 Programming: II. Dominance Relations and Algorithms*. Mathematical Programming 30, 22-45.
- [5] D. P. Bertsekas and J. N. Tsitsiklis [1989]. *Parallel and Distributed Computation*. Prentice Hall, Englewood Cliffs, NJ.
- [6] C. H. Bischof [1988]. *QR Factorization Algorithms for Coarse Grain Distributed Systems*. Ph.D. Thesis, Dept. of Computer Science, Cornell Univ., Ithaca, NY.
- [7] R. E. Bixby [1987]. *Notes on Combinatorial Optimization*. Technical Report TR87-21, Dept. of Math. Sci., Rice Univ., Houston, TX.

- [8] R. E. Bixby [1990]. *Implementing the Simplex Method: The Initial Basis*. Technical Report TR90-32. Dept. of Math. Sci., Rice Univ., Houston, TX.
- [9] V. Chvátal [1980]. *Linear Programming*. W. H. Freeman and Company, New York, NY.
- [10] Y. Crama, P. Hansen and B. Jaumard [1990]. *The Basic Algorithm for Pseudo-Boolean Programming Revisited*.
- [11] G. B. Dantzig [1960]. *On the Significance of Solving Linear Programming Problems with Some Integer Variables*. The Rand Corporation, Document P1486.
- [12] J. E. Dennis, Jr., H. J. Martinez and R. A. Tapia [1989]. *A Convergence Theory for the Structured BFGS Secant Method with an Application to Nonlinear Least Squares*. Journal of Optimization Theory and Applications 61, 159-176.
- [13] J. E. Dennis, Jr. and R. B. Schnabel [1983]. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, NJ.
- [14] J. J. Dongarra, C. B. Moler, J. R. Bunch and G. W. Stewart [1979]. *LINPACK Users' Guide*. SIAM, Philadelphia.
- [15] P. van Emde-Boas [1981]. *Another NP-complete Partition Problem and the Complexity of Computing Short Vectors in a Lattice*. Report 81-04, Mathematical Institute, Univ. of Amsterdam, Amsterdam.

- [16] R. Fortet [1959]. *L'algèbre de Boole et ses Applications en Recherche Opérationnelle*. Cahiers du Centre d'Études de Recherche Opérationnelle 1(4), 5-36.
- [17] R. S. Garfinkel and G. L. Nemhauser [1972]. *Integer Programming*. John Wiley & Sons, Inc., New York, NY.
- [18] G. A. Geist and M. T. Heath [1985]. *Parallel Cholesky Factorization on a Hypercube Multiprocessor*. Report ORNL 6190, Oak Ridge Laboratory, Oak Ridge, TN.
- [19] G. A. Geist and C. H. Romine [1988]. *LU Factorization Algorithms on Distributed Memory Multiprocessor Architectures*. SIAM Journal on Scientific and Statistical Computing 9, 639-649.
- [20] F. Glover and E. Woolsey [1973]. *Further Reduction of Zero-One Polynomial Programs to Zero-One Linear Programming Problems*. Operations Research 21(1), 156-161.
- [21] F. Glover and E. Woolsey [1974]. *Converting the 0-1 Polynomial Programming problems to a 0-1 Linear Program*. Operations Research 22, 180-182.
- [22] G. H. Golub and C. F. Van Loan [1989]. *Matrix Computations*. The Johns Hopkins Univ. Press, Baltimore, MD.

- [23] M. Grötschel, L. Lovász and A. Schrijver [1987]. *Geometric Algorithm and Combinatorial Optimization*. Springer, New York, NY.
- [24] V. P. Gulati, S. K. Gupta, and A. K. Mittal [1981]. *Unconstrained Quadratic Bivalent Programming Problems*. European Journal of Operations Research 15, 121-125.
- [25] P. L. Hammer, I. Rosenberg and S. Rudeanu [1963]. *On the Determination of the Minima of Pseudo-Boolean Functions*. (in Romanian) Studii de Cercetari Matematica 14, 359-364.
- [26] P. L. Hammer and S. Rudeanu [1968]. *Boolean Methods in Operations Research and Related Areas*. Springer, New York, NY.
- [27] P. Hansen, B. Jaumard, V. Mathon [1989]. *Constrained Nonlinear 0-1 Programming*. RRR #47-89, RUTCOR, Rutgers Univ., New Brunswick, NJ.
- [28] R. Kannan [1987]. *Minkowski's Convex Body Theorem and Integer Programming*. Mathematics of Operations Research 12, 415-440.
- [29] R. M. Karp and Y. Zhang [1988]. *A Randomized Parallel Branch-and-Bound Procedure*. ACM Symposium on Theory of Computing, 290-300.
- [30] L. Lovász [1986]. *An Algorithmic Theory of Numbers, Graphs and Convexity*. CBMS-NSF Regional Conference Series in Applied Mathematics 50. SIAM, Philadelphia.

- [31] L. Lovász [1989]. *Geometry of Numbers and Integer Programming*. M. Iri and K. Tanabe (eds), *Mathematical Programming*, 177-201, KTK Scientific Publisher, Tokyo.
- [32] G. Li and T. Coleman [1988]. *A Parallel Triangular Solver for a Distributed-Memory Multiprocessor*. *SIAM Journal on Scientific and Statistical Computing* 9, 485-502.
- [33] G. Li and T. Coleman [1989]. *A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors*. *SIAM Journal on Scientific and Statistical Computing* 10, 382-396.
- [34] G. L. Nemhauser and L. A. Wolsey [1988]. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY.
- [35] J. M. Ortega and W. C. Rheinboldt [1970]. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, New York, NY.
- [36] Parasoft [1990]. *Express C User's Guide*. ParaSoft Corporation, Pasadena, CA.
- [37] P. M. Pardalos and J. B. Rosen [1987]. *Constrained Global Optimization: Algorithms and Applications*. Springer-Verlag, New York, NY.
- [38] P. B. Percell [1987]. *Steady-State Optimization of Gas Pipeline Network Operation*. Pipeline Simulation Interest Group Annual Meeting, Tulsa, OK.

- [39] E. Pruul, G. L. Nemhauser and R. Rushmeier [1988]. *Parallel Processing and Branch-and-Bound: A Historical Note*. Operations Research Letters.
- [40] R. Rushmeier [1990]. *Strategies for Parallel Integer Programming*. CRPC First Annual Research Symposium, Houston, TX.
- [41] J. F. Shapiro [1979]. *Mathematical Programming, Structures and Algorithms*. John Wiley & Sons, Inc., New York, NY.
- [42] R. Tapia [1977]. *Diagonalized Multiplier Methods and Quasi-Newton Methods for Constrained Optimization*. Journal of Optimization Theory and Applications 22, 135-194.
- [43] R. Tapia [1988]. *On Secant Updates for Use in General Constrained Optimization*. Mathematics of Computation 51, 181-202.
- [44] R. Tapia [1990]. *An Introduction to the Algorithms and Theory of Constrained Optimization*. Lecture Notes, Dept. of Math. Sci., Rice Univ., Houston, TX.



## Appendix A

### Print-out for an Example Problem

The Example Problem:

Objective function:  $\sum_{i=1}^8 x_i^2 - 1.6 \sum_{i=1}^8 x_i + 5.12$

Initial guess:

```
x[0]====0.000000
x[1]====0.000000
x[2]====0.000000
x[3]====0.000000
x[4]====0.000000
x[5]====0.000000
x[6]====0.000000
x[7]====1.000000
```

Optimal solution:

```
x[0]====1.000000
x[1]====1.000000
x[2]====1.000000
x[3]====1.000000
x[4]====1.000000
x[5]====1.000000
x[6]====1.000000
x[7]====1.000000
```

Output file [on NCUBE]:

Allocated 8 nodes, origin at 0, process id 5439.

Loading 85620 bytes

Loaded, starting ....

%

```

*****
|
|                                NIPACK VERSION 0
|
|                                04/01/1991
|
|                                DEPT OF MATH SCI
|                                RICE UNIV
|                                HOUSTON, TX 77251
|
*****

```

\*\*\*\*\* INPUT DATA : \*\*\*\*\*

\*\*\*\*\* DATA INPUT FINISHED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP BEGIN : \*\*\*\*\*

\*\*\*\*\* OPTIMALITY TESTING : \*\*\*\*\*

\*\*\*\*\* OPTIMALITY TESTING FINISHED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 1 : \*\*\*\*\*

\*\*\*\*\* COMPUTING A SUBGRADIENT : \*\*\*\*\*

SUBGRADIENT:

```

s[0]====-1.056250
s[1]====-1.056250
s[2]====-1.056250
s[3]====-1.056250
s[4]====-1.056250
s[5]====-1.056250
s[6]====-1.056250
s[7]====0.206250

```

\*\*\*\*\* COMPUTING SUBGRADIENT FINISHED ! \*\*\*\*\*

\*\*\*\*\* SOLVING INT MINMAX PROBLEM : \*\*\*\*\*

\*\*\*\*\* INT MINMAX PROBLEM SOLVED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 1 FINISHED ! \*\*\*\*\*

NEXT ITERATION POINT:

```
x[0]====1.000000
x[1]====1.000000
x[2]====1.000000
x[3]====1.000000
x[4]====1.000000
x[5]====1.000000
x[6]====1.000000
x[7]====0.000000
```

\*\*\*\*\* OPTIMALITY TESTING : \*\*\*\*\*

\*\*\*\*\* OPTIMALITY TESTING FINISHED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 2 : \*\*\*\*\*

\*\*\*\*\* COMPUTING A SUBGRADIENT : \*\*\*\*\*

SUBGRADIENT:

```
s[0]====0.100000
s[1]====0.100000
s[2]====0.100000
s[3]====0.100000
s[4]====0.100000
s[5]====0.100000
s[6]====0.100000
s[7]====-0.775000
```

\*\*\*\*\* COMPUTING SUBGRADIENT FINISHED ! \*\*\*\*\*

\*\*\*\*\* SOLVING INT MINMAX PROBLEM : \*\*\*\*\*

\*\*\*\*\* INT MINMAX PROBLEM SOLVED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 2 FINISHED ! \*\*\*\*\*

NEXT ITERATION POINT:

```
x[0]====1.000000
x[1]====0.000000
x[2]====1.000000
x[3]====1.000000
x[4]====1.000000
x[5]====0.000000
x[6]====1.000000
x[7]====1.000000
```

\*\*\*\*\* OPTIMALITY TESTING : \*\*\*\*\*

\*\*\*\*\* OPTIMALITY TESTING FINISHED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 3 : \*\*\*\*\*

\*\*\*\*\* COMPUTING A SUBGRADIENT : \*\*\*\*\*

SUBGRADIENT:

```
s[0]====-0.050000
s[1]====-0.996875
s[2]====-0.050000
s[3]====-0.050000
s[4]====-0.050000
s[5]====-0.996875
s[6]====-0.050000
s[7]====-0.050000
```

\*\*\*\*\* COMPUTING SUBGRADIENT FINISHED ! \*\*\*\*\*

\*\*\*\*\* SOLVING INT MINMAX PROBLEM : \*\*\*\*\*

\*\*\*\*\* INT MINMAX PROBLEM SOLVED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 3 FINISHED ! \*\*\*\*\*

NEXT ITERATION POINT:

```

x[0]====1.000000
x[1]====1.000000
x[2]====1.000000
x[3]====1.000000
x[4]====0.000000
x[5]====1.000000
x[6]====0.000000
x[7]====1.000000

```

\*\*\*\*\* OPTIMALITY TESTING : \*\*\*\*\*

\*\*\*\*\* OPTIMALITY TESTING FINISHED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 4 : \*\*\*\*\*

\*\*\*\*\* COMPUTING A SUBGRADIENT : \*\*\*\*\*

SUBGRADIENT:

```

s[0]====-0.050000
s[1]====-0.050000
s[2]====-0.050000
s[3]====-0.050000
s[4]====-0.996875
s[5]====-0.050000
s[6]====-0.996875
s[7]====-0.050000

```

\*\*\*\*\* COMPUTING SUBGRADIENT FINISHED ! \*\*\*\*\*

\*\*\*\*\* SOLVING INT MINMAX PROBLEM : \*\*\*\*\*

\*\*\*\*\* INT MINMAX PROBLEM SOLVED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 4 FINISHED ! \*\*\*\*\*

NEXT ITERATION POINT:

```

x[0]====0.000000
x[1]====1.000000

```

```

x[2]====1.000000
x[3]====0.000000
x[4]====1.000000
x[5]====1.000000
x[6]====1.000000
x[7]====1.000000

```

\*\*\*\*\* OPTIMALITY TESTING : \*\*\*\*\*

\*\*\*\*\* OPTIMALITY TESTING FINISHED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 5 : \*\*\*\*\*

\*\*\*\*\* COMPUTING A SUBGRADIENT : \*\*\*\*\*

SUBGRADIENT:

```

s[0]====-0.996875
s[1]====-0.050000
s[2]====-0.050000
s[3]====-0.996875
s[4]====-0.050000
s[5]====-0.050000
s[6]====-0.050000
s[7]====-0.050000

```

\*\*\*\*\* COMPUTING SUBGRADIENT FINISHED ! \*\*\*\*\*

\*\*\*\*\* SOLVING INT MINMAX PROBLEM : \*\*\*\*\*

\*\*\*\*\* INT MINMAX PROBLEM SOLVED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 5 FINISHED ! \*\*\*\*\*

NEXT ITERATION POINT:

```

x[0]====1.000000
x[1]====1.000000
x[2]====0.000000
x[3]====1.000000

```

```

x[4]====1.000000
x[5]====1.000000
x[6]====1.000000
x[7]====1.000000

```

\*\*\*\*\* OPTIMALITY TESTING : \*\*\*\*\*

\*\*\*\*\* OPTIMALITY TESTING FINISHED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 6 : \*\*\*\*\*

\*\*\*\*\* COMPUTING A SUBGRADIENT : \*\*\*\*\*

SUBGRADIENT:

```

s[0]====0.100000
s[1]====0.100000
s[2]====-0.775000
s[3]====0.100000
s[4]====0.100000
s[5]====0.100000
s[6]====0.100000
s[7]====0.100000

```

\*\*\*\*\* COMPUTING SUBGRADIENT FINISHED ! \*\*\*\*\*

\*\*\*\*\* SOLVING INT MINMAX PROBLEM : \*\*\*\*\*

\*\*\*\*\* INT MINMAX PROBLEM SOLVED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 6 FINISHED ! \*\*\*\*\*

NEXT ITERATION POINT:

```

x[0]====1.000000
x[1]====1.000000
x[2]====1.000000
x[3]====1.000000
x[4]====1.000000
x[5]====1.000000
x[6]====1.000000
x[7]====1.000000

```

\*\*\*\*\* OPTIMALITY TESTING : \*\*\*\*\*

\*\*\*\*\* OPTIMALITY TESTING FINISHED ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP ITERATION 7 : \*\*\*\*\*

\*\*\*\*\* COMPUTING A SUBGRADIENT : \*\*\*\*\*

SUBGRADIENT:

s[0]====-0.112500

s[1]====-0.112500

s[2]====-0.112500

s[3]====-0.112500

s[4]====-0.112500

s[5]====-0.112500

s[6]====-0.112500

s[7]====-0.112500

\*\*\*\*\* COMPUTING SUBGRADIENT FINISHED ! \*\*\*\*\*

\*\*\*\*\* OPTIMAL SOLUTION FOUND ! \*\*\*\*\*

\*\*\*\*\* MAIN LOOP END ! \*\*\*\*\*

OPTIMAL VALUE:

f(x)====0.320000

OPTIMAL SOLUTION:

x[0]====1.000000

x[1]====1.000000

x[2]====1.000000

x[3]====1.000000

x[4]====1.000000

x[5]====1.000000

x[6]====1.000000

x[7]====1.000000



TOTAL NUMBER OF ITERATIONS: 7

System 0:2 User 3:19

CUBIX: exit status 0

