

Branch and Bound Algorithm for
Binary Quadratic Programming with
Application in Wireless Network
Communications

by

Eun-Mee YoonAnn

Advisor: Professor Zhijun Wu

Department of Mathematics

Iowa State University

June, 2006

Contents

1. Introduction	1
1.1 Motivation	1
2. Linear Programming	3
2.1 Simplex Method	5
2.2 Integer Linear Programming	11
3. Nonlinear Programming	14
3.1 Quadratic Programming	14
3.2 Quadratic Integer Programming	18
4. Branch and Bound	20
4.1 Branching	21
4.2 Bounding	22
4.3 Node Selection	23
4.4 Branching Variable Selection	25
4.5 Branch and Bound Algorithm	26
5. Maximum Likelihood Detection	28
5.1 Multiple Input Multiple Output (MIMO)	28
5.2 Maximum Likelihood Detection	29
5.3 Branch and Bound in Matlab	30
5.4 An Example	32
5.5 More Experiments	37
6. Conclusion	39
Bibliography	40
Branch and Bound in Matlab	42

1. Introduction

Driven by the demand for increasingly sophisticated "anywhere anytime" connectivity, wireless communications have emerged as one of the largest and most rapidly growing sectors of the global telecommunications industry [6]. In modern communication systems, multiple users share a multi-access channel called a *Multiple Input Multiple Output* (MIMO) channel [7], where the information sent by different users is separated by using almost certain codes. The process of simultaneously estimating the information sent by multiple users is called *multiuser detection* (MUD) [8]. To estimate the information originally sent, a *Maximum Likelihood* (ML) problem is solved [8].

The Maximum Likelihood Detection is a *nondeterministic polynomial-time hard* (NP-hard) problem, for which there is no known algorithm that can find the optimal solution with polynomial-time complexity [9]. In a practical communication system, transmitted signals consist of either -1 or 1 . This discrete structure generally causes the Maximum Likelihood problem to be NP-hard [3]. Maximum Likelihood (ML) Detection is formulated as a *binary quadratic programming* (BQP) problem, which is a *combinatorial optimization problem* [8].

1.1 Motivation

A combinatorial optimization problem is solved by choosing the best combination out of all possible combinations. The most straightforward approach is to compute the optimal solution by enumerating all possible combinations. But, the computational burden for this approach is overwhelming. One of the popular detectors, called *Sphere*

Decoder (SD), offers ML Detection at a computational complexity that is polynomial in the average case [4]. M. Kisialiou and Z. Luo state that "Although Sphere Decoder offers excellent practical performance, its expected complexity is exponential in the problem size" [3,5].

Our conclusion is that we need an algorithm that can find the optimal solution without enumerating all possible combinations. This algorithm is *branch and bound*, [2] which enumerates only some of the possible combinations. Using the branch and bound algorithm, we can efficiently find the exact optimal solution [10].

In our work, we programmed Branch and Bound Algorithm in Malab and tested the program with several wireless communication problems, which we randomly generated. The results of experiment show in Chapter 5.

2. Linear Programming

Linear Programming (LP) [1] is a special case of mathematical programming.

In general, if c_1, c_2, \dots, c_n are real numbers, then a function f of real variables

x_1, x_2, \dots, x_n defined by $f(x_1, x_2, \dots, x_n) = c_1x_1 + c_2x_2 + \dots + c_nx_n = \sum_{j=1}^n c_jx_j$ is called

a *linear function*. If f is a linear function and if b is a real number, then the equation

$f(x_1, x_2, \dots, x_n) = b$ and the inequalities of the form $f(x_1, x_2, \dots, x_n) \leq b$ or $f(x_1, x_2, \dots, x_n)$

$\geq b$ are called *linear constraints* [12].

From an analytical perspective, a linear program is the problem of maximizing or minimizing a linear function subject to a finite number of linear constraints. Thus, a linear program is a problem that can be expressed as follows :

$$\begin{array}{ll} \text{maximize (or minimize)} & \sum_{j=1}^n c_jx_j \\ \text{subject to} & \sum_{j=1}^n a_{ij}x_j \leq b_i \quad (i = 1, 2, \dots, m) \\ & x_j \geq 0 \quad (j = 1, 2, \dots, n), \end{array} \quad (2.1)$$

where x is the vector of variables to be solved for, A is a matrix of known coefficients, and c and b are vectors of known coefficients.

The linear program (2.1) is in *standard form* [1]. There are two conditions for standard form: first is that all of the constraints are linear inequalities. Second is that $x_1, x_2, \dots, x_n \geq 0$. The latter constraints are called *nonnegativity constraints*. The linear function to be maximized or minimized in an linear programming problem is called the *objective function*. Thus, the goal of a linear program is to optimize the objective function subject to the constraints.

A region S that satisfies all the constraints of a linear programming is a *feasible region*. Numbers x_1, x_2, \dots, x_n in S are said to be a *feasible solution* of the problem. A feasible solution that minimizes (or maximizes) the objective function is called an *optimal solution*; the corresponding value of the objective function is called the *optimal value* of the problem [1].

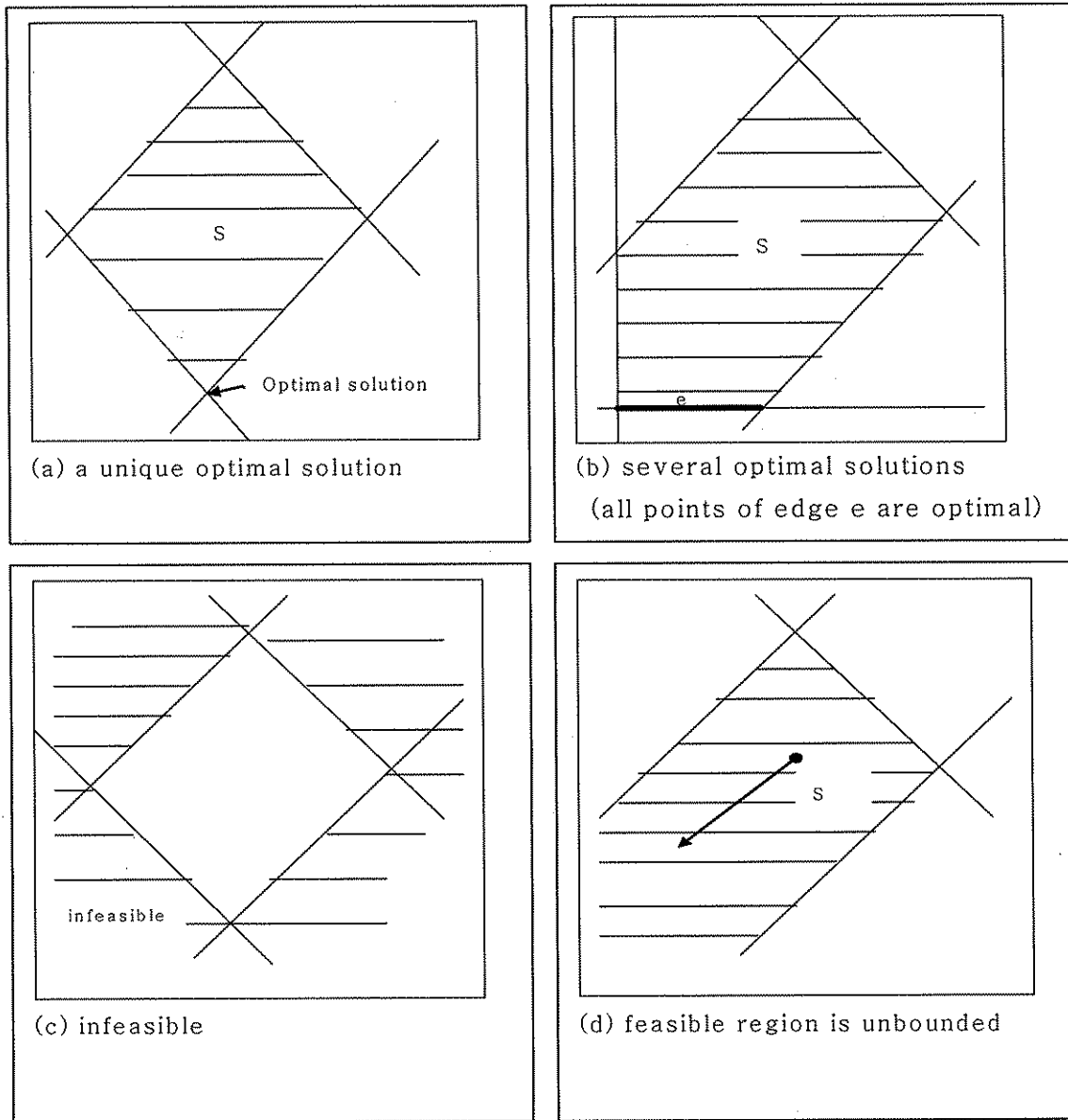


Figure 2.2 Different types of solutions to a linear program

Figure 2.2 shows the different types of solutions to a linear program [1]. Some LP problems have a unique optimal solution see Fig. 2.2(a); some problems have many different optimal solutions, as in Fig. 2.2(b); and others have no optimal solutions at all. The latter may occur for two different reasons. The first case is illustrated in Fig. 2.2(c), which shows an LP with no feasible solution. Such problems are called *infeasible*. On the other hand, some LP problems have feasible solutions, but none of them is optimal. Figure 2.2(d) shows this case, which is called *unbounded*.

Next, we give *The Fundamental Theorem of Linear Programming* [1]

Theorem 2.3 Every LP problem in standard form has the following three properties.

- (i) If it has no optimal solution, then it is either infeasible or unbounded.
- (ii) If it has a feasible solution, then it has a basic feasible solution.
- (iii) If it has an optimal solution, then it has a basic optimal solution.

Thus, every linear programming problem belongs to one of three categories: it has an optimal solution, is infeasible, or is unbounded.

Linear programming can be applied to a wide variety of optimization problems. It started in 1947 when G.B. Dantzig designed U.S. Air Force planning problems, which dealt with the logistics of supply chains and management of hundreds of thousands of items and people. The job provided the "real world" problems which linear programming would come to solve [11].

2.1 Simplex Method

In optimization problems, the *simplex algorithm* is the popular technique for numerical solution of the linear programming problems [12].

In general, given a problem

$$\begin{aligned}
 & \text{maximize} && \sum_{j=1}^n c_j x_j \\
 & \text{subject to} && \sum_{j=1}^n a_{ij} x_j \leq b_i \quad (i = 1, 2, \dots, m) \quad (2.4) \\
 & && x_j \geq 0 \quad (j = 1, 2, \dots, n),
 \end{aligned}$$

the first step is to introduce the *slack variables* $x_{n+1}, x_{n+2}, \dots, x_{n+m}$ and rewrite the problem as

$$\begin{aligned}
 x_{n+i} &= b_i - \sum_{j=1}^n a_{ij} x_j \quad (i = 1, 2, \dots, m) \\
 z &= \sum_{j=1}^n c_j x_j.
 \end{aligned} \tag{2.5}$$

In the framework of the simplex method, each feasible solution x_1, x_2, \dots, x_n of (2.4) is represented by $n + m$ nonnegative numbers x_1, x_2, \dots, x_{n+m} , with $x_{n+1}, x_{n+2}, \dots, x_{n+m}$, defined by (2.5). Next, we have to find an *initial basic feasible solution*: setting $x_1, x_2, \dots, x_n = 0$, we will get $x_{n+1} = b_1, x_{n+2} = b_2, \dots, x_{n+m} = b_m$ as the initial basic feasible solution and corresponding value $z = 0$. Here, $x_{n+1}, x_{n+2}, \dots, x_{n+m}$ is the set of *basic variables (basis)* and x_1, x_2, \dots, x_n are *nonbasic variables*.

The second step is testing the optimality of the basic feasible solution. We have to look at the objective function with the question: Can the value of z be increased by choosing different basic feasible solution? From the previous step, the objective value is $z = 0$. Thus, the positive coefficients among the nonbasic variables x_1, x_2, \dots, x_n are good candidates for entering the basis. We choose the entering variable that gives the maximum improvement to the objective value. This gives us an *adjacent basic feasible solution* [12].

The third step is selecting a variable to leave the basis. Once we have selected the variable x_j , where $\max c_j, (j = 1, 2, \dots, n)$, to enter the basis, we have to choose one variable to leave the basis, in order to improve the objective value. We would like to increase x_j as much as we can. Notice that as the entering variable, x_j , increases, the basic variables are decreased. Obviously, x_j can not be increased beyond a value that makes any of $x_{n+1}, x_{n+2}, \dots, x_{n+m}$ negative. Thus, the maximal allowable increase for is obtained by solving the system of inequalities :

$$x_{n+i} = b_i - a_{ij}x_j \geq 0 \quad (i = 1, 2, \dots, m), \quad \max c_j, (j = 1, 2, \dots, n) \quad (2.6)$$

$$\text{Hence, } x_j \leq \min \left\{ \frac{b_i}{a_{ij}} \right\}, \quad a_{ij} > 0 \quad (2.7)$$

The fourth step is to reset the constraints to the form of (2.5) and return to the second step. We will repeat this process until no further increase in the objective function z is possible.

In each iteration, the simplex method moves from some feasible solution x_1, x_2, \dots, x_{n+m} to another feasible solution $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_{n+m}$, which is better than the previous one in the sense that $\sum_{j=1}^n c_j \bar{x}_j > \sum_{j=1}^n c_j x_j$.

We will illustrate the simplex method with the following example:

$$\begin{aligned} \text{maximize} \quad & 5x_1 + 4x_2 + 3x_3 \\ \text{subject to} \quad & 2x_1 + 3x_2 + x_3 \leq 5 \\ & 4x_1 + x_2 + 2x_3 \leq 11 \\ & 3x_1 + 4x_2 + 2x_3 \leq 8 \\ & x_1, x_2, x_3 \geq 0. \end{aligned} \quad (2.8)$$

The first step is to introduce the slack variables x_4, x_5, x_6 and to write the problem in the form of (2.5)

$$\begin{aligned}
x_4 &= 5 - 2x_1 - 3x_2 - x_3 \\
x_5 &= 11 - 4x_1 - x_2 - 2x_3 \\
x_6 &= 8 - 3x_1 - 4x_2 - 2x_3 \quad (2.9) \\
z &= 5x_1 + 4x_2 + 3x_3 \\
x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0.
\end{aligned}$$

To find an initial basic feasible solution, we set the nonbasic variables x_1, x_2, x_3 equal to 0. Then, the initial feasible solution of (2.9) is $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 5, x_5 = 11, x_6 = 8$; this gives $z = 0$. In the first iteration, we try to increase the value of z by making one of the nonbasic variables x_1, x_2, x_3 positive. We choose the variable which has the largest coefficient in z . Increasing this variable makes z increase the most.

Thus, in (2.9), we see that x_1 is the chosen variable. The question is that how much can we increase x_1 (keeping $x_2 = x_3 = 0$ at the same time) and still maintain feasibility ($x_4, x_5, x_6 \geq 0$). The condition $x_4 = 5 - 2x_1 - 3x_2 - x_3 \geq 0$ implies $x_1 \leq \frac{5}{2}$; similarly, $x_5 \geq 0$ implies $x_1 \leq \frac{11}{4}$, and $x_6 \geq 0$ implies $x_1 \leq \frac{8}{3}$. By (2.7), the entering variable is x_1 . Increasing x_1 up to $\frac{5}{2}$, we get the next feasible solution, $x_1 = \frac{5}{2}, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 1, x_6 = \frac{1}{2}$; this gives $z = \frac{25}{2}$, which is an improvement over $z = 0$. In other words, x_1 is the entering variable and x_4 is the leaving variable. We now rewrite the system of equations for x_1 in terms of x_2, x_3, x_4 , following (2.5)

$$\begin{aligned}
x_1 &= \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \\
x_5 &= 1 + 5x_2 + 2x_4 \\
x_6 &= \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4 \quad (2.10) \\
z &= \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4 \\
x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0.
\end{aligned}$$

As in the first iteration, we try to increase the value of z by increasing the value of right-hand side variable. Note that increases in the value of x_2 or x_4 would the most,

decrease the value of z . We choose the variable which can increase z which is x_3 . From (2.10), with $x_2 = x_4 = 0$, the constraint $x_1 \geq 0$ implies $x_3 \leq 5$, $x_5 \geq 0$ imposes no restriction at all, and $x_6 \geq 0$ implies $x_3 \leq 1$. Thus, $x_3 \leq 1$ is the best we can do and new solution is $x_1 = 2, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 0$; this gives $z = 13$, which is an improvement over $z = 12.5$. Nothing that the entering variable is x_3 and the leaving variable is x_6 , the system becomes

$$\begin{aligned}
 x_3 &= 1 + x_2 + 3x_4 - 2x_6 \\
 x_1 &= 2 - 2x_2 - 2x_4 + x_6 \\
 x_5 &= 1 + 5x_2 + 2x_4 \\
 z &= 13 - 3x_2 - x_4 - x_6 \\
 x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0.
 \end{aligned}
 \tag{2.11}$$

For finding a variable, which increase the objective function, we look at the right-hand side of z in (2.11). However, there is no such variable because if we increase any of the right-hand side variables x_2, x_4, x_6 , we will make the value of z decrease. Therefore, we have solved the problem. The optimal solution is $x_1 = 2, x_3 = 1, x_5 = 1$; this gives $z = 13$.

The basic logic of the algorithm [12] is depicted in Figure 2.12. The algorithm starts with an initial basic feasible solution and tests its optimality. If the optimality condition is verified, then the algorithm terminates. Otherwise, the algorithm identifies an adjacent basic feasible solution, having a better objective value. The optimality of this new solution is tested again, and the entire scheme is repeated, until an optimal basic feasible solution is found. Because each time a new basic feasible solution is identified the objective value is improved and the set of basic feasible solution is finite, it follows that the algorithm will terminate in a finite number of iterations.

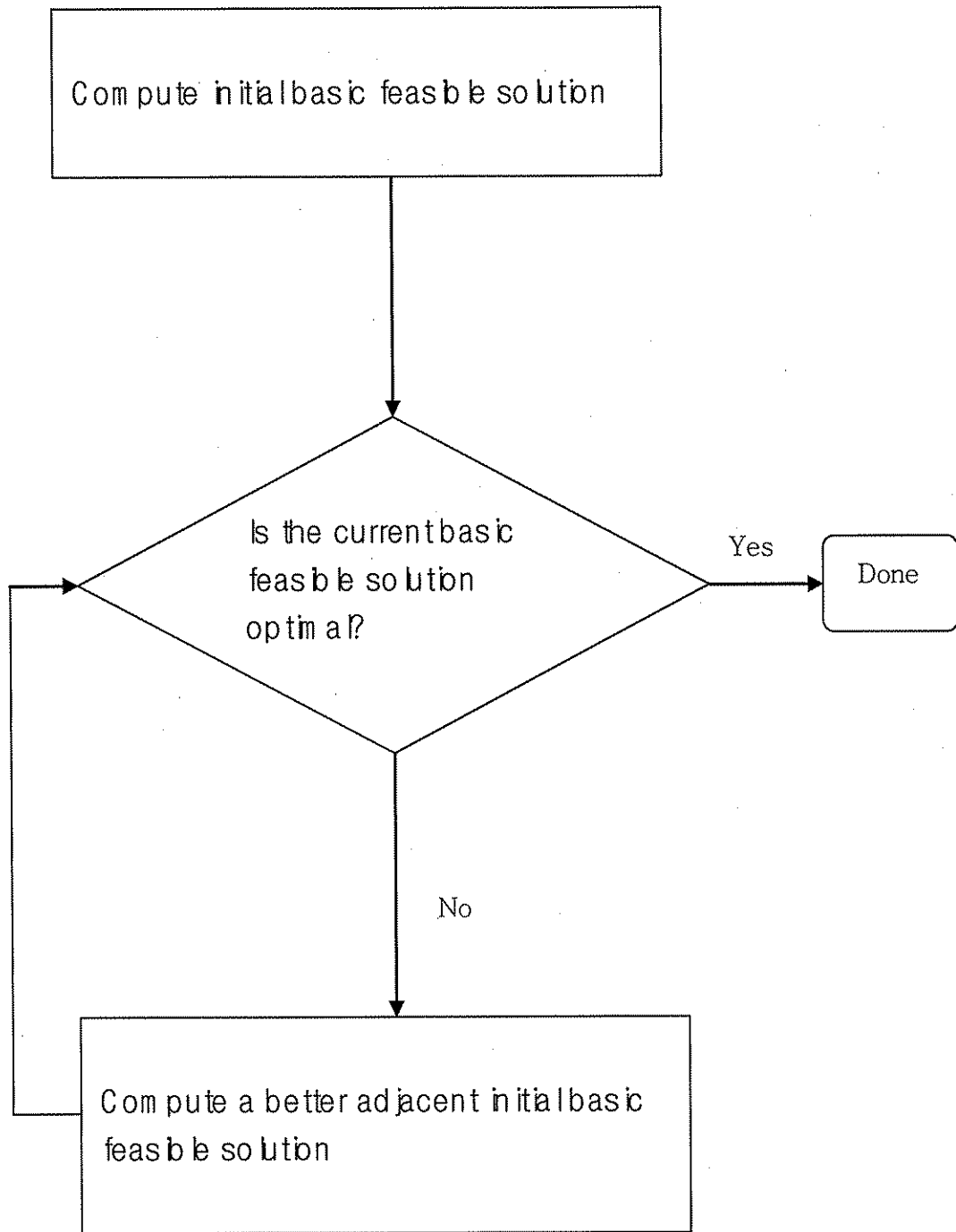


Figure 2.12: The basic Simplex logic

2.2 Integer Linear Programming

Integer Linear Programming (ILP) [11] uses a model similar to linear programming in that the objective function and constraint functions are linear. In integer programming, however, some or all the variables are required to be integers. If all variables are required to be integers, it is called a *pure integer programming* problem. If some variables are restricted to be integers and some are not then the problem is a *mixed integer programming* problem [13].

Integer programs often have the advantage of being more realistic than linear programming, but the disadvantage of being much harder to solve. The most widely used general-purpose techniques for solving integer programs use the solutions to a series of linear programs to manage the search for integer solutions and to prove optimality. Thus most integer programs is built upon linear programming. Linear and integer programming have provided the means for solving the diverse types of problems in planning, routing, scheduling, assignment, and design. Industries that make use of linear programming and its extensions include transportation, energy, telecommunications, and manufacturing of many kinds.

Because finding a solution of an integer program is based on linear programming, we look at the relationship between linear and integer programming.

Given an integer program

$$\begin{aligned} & \text{Minimize} && \sum_{j=1}^n c_j x_j \\ (IP) & \text{subject to} && \sum_{j=1}^n a_{ij} x_j \leq b_i && (i = 1, 2, \dots, m) \\ & && x_j \geq 0 \text{ and is an integer} && (j = 1, 2, \dots, n), \end{aligned} \tag{2.13}$$

there is an associated linear program called the *linear relaxation (LR)* formed by dropping the condition that x_j be an integer:

$$\begin{aligned}
 & \text{Minimize} && \sum_{j=1}^n c_j x_j \\
 (LR) & \text{subject to} && \sum_{j=1}^n a_{ij} x_j \leq b_i && (i = 1, 2, \dots, m) \\
 & && x_j \geq 0 && (j = 1, 2, \dots, n).
 \end{aligned} \tag{2.14}$$

Since LR is less constrained than IP, the following are immediate [14] :

- The optimal objective value for LR is less than or equal to the optimal objective for IP.
- If LR is infeasible, then so is IP.
- If LR is optimized by integer variables, then that solution is feasible and optimal for IP.

Thus, solving LR does give some information. It gives a *lower bound* for the optimal value of the IP problem. If we are lucky, we may find the optimal solution to IP; however, rounding the solution of LR will not in general give the optimal solution of IP.

The following capital budgeting example (2.15) shows that rounding does not necessarily give an optimal value. Suppose we wish to invest \$14,000. We have identified four investment opportunities. Into which investments should we place our money so as to maximize our total present value?

Plan	1	2	3	4	
Investment	\$5,000	\$7,000	\$4,000	\$3,000	(2.15)
Present Value	\$8,000	\$11,000	\$6,000	\$4,000	

As in linear programming, our first step is to decide on our variables. In this case,

we will use a 0 – 1 variable x_j ($j = 1, 2, 3, 4$) for each investment. If it is 1 then we will make investment x_j . If it is 0, we will not make the investment. This leads to the *Binary*

Integer Programming problem:

$$\begin{aligned} \text{Maximize} \quad & 8x_1 + 11x_2 + 6x_3 + 4x_4 \\ \text{subject to} \quad & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \quad (2.16) \\ & x_j \in \{0, 1\}, (j = 1, 2, 3, 4) \end{aligned}$$

Ignoring integrality constraints, the optimal linear programming solution is $x_1 = 1$, $x_2 = 1$, $x_3 = 0.5$, $x_4 = 0$ for a value of \$22,000. Unfortunately, this solution is not integral. Rounding x_3 down to 0 gives a feasible solution with a value of \$19,000, however, there is a better integer solution, $x_1 = 0$, $x_2, x_3, x_4 = 1$ for a value of \$21,000.

There are two common approaches to solve integer programming problems. One is *Cutting Plane Techniques* [14], in which cutting planes are used to add constraints to force integrality. The other is *branch and bound* [2, 14], which is based on dividing the problem into a number of smaller problems. All of these approaches involve solving a series of linear programs. Because branch and bound is the most effective technique [10], we will only look at the branch and bound in Chapter 4.

3. Nonlinear Programming

The goal of *Nonlinear Programming* (NLP) [13] is to find a solution that optimizes an objective function, possibly subject to constraints on the decision variables. It is different than linear programming in that the functions defining the objective function or specifying the constraints may be nonlinear functions of the decision variables. This makes the nonlinear programming model much more difficult to express because nonlinear functions can take on a such a wide variety of functional forms. It is also much more difficult to solve nonlinear programming problems.

3.1 Quadratic Programming

A *quadratic program* (QP) [11] is a nonlinear optimization problem with a quadratic objective function and linear constraints. The general quadratic program can be written as

$$\begin{aligned} \text{minimize} \quad & f(x) = \frac{1}{2}x^T Qx + cx \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0, \end{aligned} \tag{3.1}$$

where x is an $n \times 1$ vector, Q is an $n \times n$ symmetric matrix describing the coefficients of the quadratic terms, and c is an n - dimensional row vector describing the coefficients of the linear terms in the objective function. A is an $m \times n$ constant matrix and b is an m - dimensional column vector. We assume that a feasible solution exists and that the constraint region is bounded.

When the objective function $f(x)$ is strictly convex for all feasible points, the problem

has a unique *local minimum* which is also the *global minimum*. A sufficient condition to guarantee strict convexity is for Q to be *positive definite* [11].

We will examine the *Karush-Kuhn-Tucker* Conditions [8] for quadratic programming and see that they turn out to be a set of linear equalities and complementarity constraints. In this case, we can use simplex algorithm to find solutions.

The *Lagrangian* function for the quadratic program (3.1) is

$$L(x, \mu) = \frac{1}{2}x^T Qx + cx + \mu(Ax - b), \quad (3.2)$$

where μ is an m - dimensional row vector. The *Karush-Kuhn-Tucker* Conditions for a local minimum are given as follows [13] :

$$\begin{aligned} \frac{\partial L}{\partial x_j} &= x^T Q + c + \mu A \geq 0, & j &= 1, \dots, n & (a) \\ \frac{\partial L}{\partial \mu_i} &= Ax - b \leq 0, & i &= 1, \dots, m & (b) \\ \mu_i g_i(x) &= \mu(Ax - b) = 0, & i &= 1, \dots, m & (c) \\ x_j &= x \geq 0, & j &= 1, \dots, n & (d) \\ \mu_i &= \mu \geq 0, & i &= 1, \dots, m. & (e) \end{aligned} \quad (3.3)$$

To put on (3.3) (a) and (b) into a more manageable form, we introduce nonnegative surplus variables $y \in \mathbb{R}^n$ to the inequalities in (3.3) (a) and nonnegative slack variables $v \in \mathbb{R}^m$ to the inequalities in (b) to obtain the equations

$$Qx + c^T + A^T \mu^T - y = 0 \text{ and } Ax - b + v = 0$$

The KKT Conditions can now be written with the constraints moved to the right - hand side.

$$\begin{aligned}
Qx + A^T \mu^T - y &= -c^T & (a) \\
Ax + v &= b & (b) \\
x, \mu, y, v &\geq 0 & (c) \\
y^T x &= 0, \mu v = 0. & (d)
\end{aligned}
\tag{3.4}$$

We use the simplex algorithm to solve (3.4) (a) – (d).

The procedure for setting up the linear programming model follows [13].

- If any of the right-hand-side values are negative, multiply the corresponding equation by -1 .
- Add an artificial variable to each equation.
- Let the objective function be the sum of the artificial variables.
- Put the resultant problem into simplex form.

The goal is to find the solution to the linear program that minimizes the sum of the artificial variables. The entering variable will be the one whose reduced cost is the most negative in the objective function. At the conclusion of the algorithm, the vector x is the optimal solution and the vector μ is the optimal solution for the dual problems.

We will solve the following problem[13].

$$\begin{aligned}
\text{minimize } f(x) &= -8x_1 - 16x_2 + x_1^2 + 4x_2^2 \\
\text{subject to } x_1 + x_2 &\leq 5 \\
x_1 &\leq 3 \\
x_1, x_2 &\geq 0
\end{aligned}
\tag{3.5}$$

$$\text{where } c = [-8, -16], Q = \begin{bmatrix} 2 & 0 \\ 0 & 8 \end{bmatrix}, A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, b = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \mu = \begin{bmatrix} \mu_1 & \mu_2 \end{bmatrix}.$$

The linear constraints (3.4 a - d) take the following form:

$$\begin{aligned} 2x_1 + \mu_1 + \mu_2 - y_1 &= 8 \\ 8x_2 + \mu_1 - y_2 &= 16 \\ x_1 + x_2 + v_1 &= 5 \\ x_1 + v_2 &= 3 \end{aligned} \quad (3.6)$$

To create the appropriate linear program, we add artificial variables to each constraint and minimize their sum.

$$\begin{aligned} \text{Minimize } & a_1 + a_2 + a_3 + a_4 \\ \text{subject to } & 2x_1 + \mu_1 + \mu_2 - y_1 + a_1 = 8 \\ & 8x_2 + \mu_1 - y_2 + a_2 = 16 \\ & x_1 + x_2 + v_1 + a_3 = 5 \\ & x_1 + v_2 + a_4 = 3 \\ & x, \mu, y, v \geq 0, \quad y^T x = 0, \mu v = 0 \end{aligned} \quad (3.7)$$

Table 3.8 shows the iteration for the solution using the simplex algorithm which was explained in Chapter 1. The optimal solution to the original problem is $(x_1^*, x_2^*) = (3, 2)$.

Iteration	Basic variables	Solution	Objective value	Entering variable	Leaving variable
1	(a_1, a_2, a_3, a_4)	$(8, 16, 5, 3)$	32	x_2	a_2
2	(a_1, x_2, a_3, a_4)	$(8, 2, 3, 3)$	14	x_1	a_3
3	(a_1, x_2, x_1, a_4)	$(2, 2, 3, 0)$	2	μ_1	a_4
4	(a_1, x_2, x_1, μ_1)	$(2, 2, 3, 0)$	2	μ_2	a_1
5	(μ_2, x_2, x_1, μ_1)	$(2, 2, 3, 0)$	0	–	–

Table 3.8 : the iteration for the solution

3.2 Quadratic Integer Programming

Quadratic Integer Programming (QIP) [8] is a nonlinear program (NLP) which has a nonlinear objective function and integer variables. At first glance, quadratic integer programming looks similar to the quadratic programming problem described in (3.1). There is, however, one important difference: the optimization variables of QIP must be integral. When the integer variables are constrained to be 0 or 1, this problem is called a *Binary Quadratic Programming* (BQP) problem. The general binary quadratic programming program without constraints can be written as:

$$\begin{aligned}
 & \text{minimize} && f(x) = \frac{1}{2}x^T Qx + cx \\
 \text{(BQP)} & && x \in \{0, 1\}
 \end{aligned} \tag{3.9}$$

where x is an $n \times 1$ vector, Q is an $n \times n$ symmetric matrix describing the coefficients of the quadratic terms, and c is an n -dimensional row vector describing the coefficients of the linear terms in the objective function.

There is an associated quadratic program called the *quadratic relaxation* (QR)

created by relaxing the integer constraints to interval constraints:

$$\begin{array}{ll} \text{minimize} & f(x) = \frac{1}{2}x^T Qx + cx \\ \text{(QR)} & x \in [0, 1] \end{array} \quad (3.10)$$

The relationship between BQP and QR is the same as the relationship between an integer programming problem and a linear relaxation introduced in Chapter 3.

We will recall the important properties:

- The optimal objective value of a QR is less than or equal to the optimal objective for BQP.
- If the QR is infeasible, then so is BQP.
- If QR is optimized by integer variables, then that solution is feasible and optimal for the BQP.

Thus, solving QR does give some information. It gives a *lower bound* for the optimal value of the BQP problem. This property is used to solve BQP problem by the branch and bound algorithm. We will look at the branch and bound algorithm in the next chapter.

4. Branch - and - Bound

The Branch-and-Bound algorithm [2] is an approach developed for solving *discrete* and *combinatorial* optimization problems. Discrete optimization problems are problems in which the decision variables assume discrete values from a specified set; when this set is the set of integers, we have an integer programming problem. Combinatorial optimization problems are those for which we choose the best combination out of all possible combinations. Most combinatorial problems can be formulated as integer programs.

The major difficulty with these problems is that we do not have an optimality condition to check if a given feasible solution is optimal or not. For example, in linear programming we do have an optimality condition: we can check if there exists an improving feasible direction or not. There is no such global optimality condition for discrete or combinatorial optimization problems. In order to guarantee a given feasible solution's optimality we compare it with every other feasible solution.

The most straightforward approach is to compute the optimal solution by enumerating all possible combinations of the variables. However, the computational burden for this approach would become overwhelming. The conclusion from this discussion is that there is a need for an algorithm that can find the optimal solution without enumerating all possible combinations of the variables. One algorithm is called branch and bound, in which it is sufficient to explicitly enumerate only some of the possible combinations using a problem's relaxations and *branching* and *bounding* tools [2].

4.1 Branching

Branching tool is used to split the feasible region into several smaller feasible subregions. If there is a feasible set S , we can branch S into k smaller sets such that

$$S = \bigcup_{i=1}^k S_i \quad (4.1)$$

Because the partitioning is repeated recursively in each of the subregions, and all such subregions naturally form a tree structure, this is called a *search tree* or *branch-and-bound-tree*. An example of a binary search tree is given in Figure 4.2 [2]

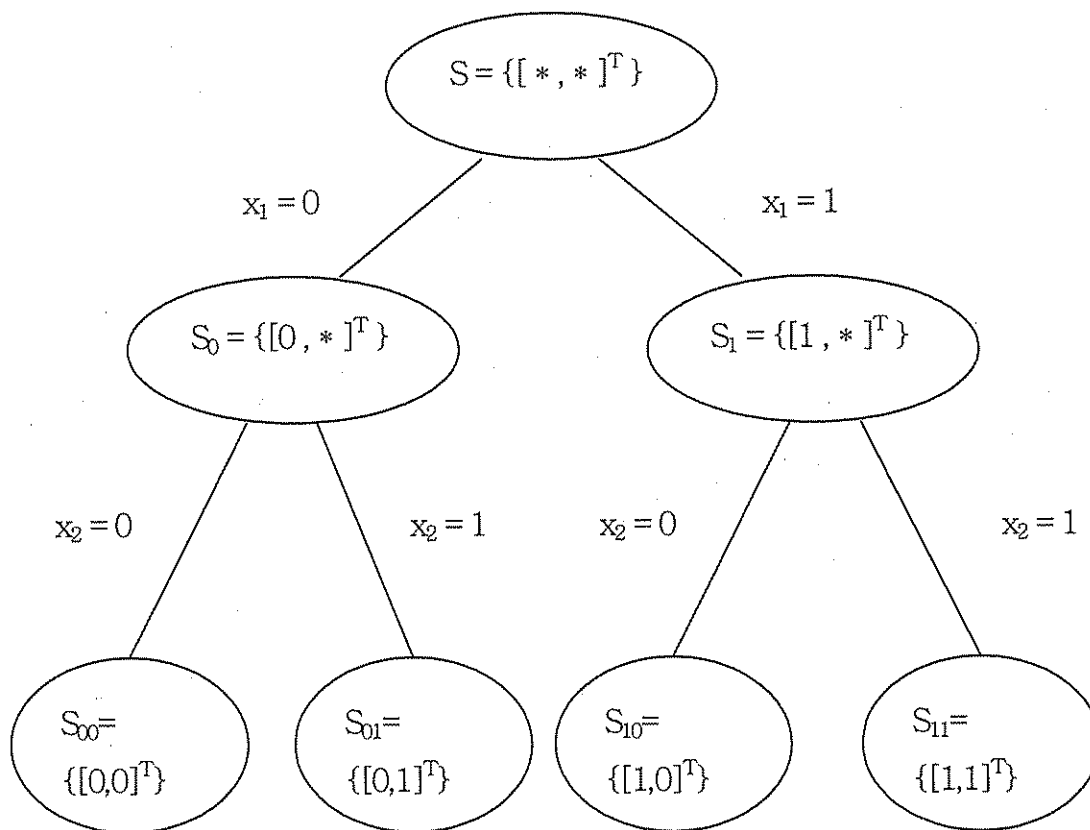


Figure 4.2 : A binary search tree

The tree originates from a binary quadratic programming problem:

$$\begin{array}{ll} \text{minimize} & f(x) = \frac{1}{2}x^T Qx + cx \\ \text{(BQP)} & x \in \{0,1\} \end{array} \quad (4.3)$$

Associated with this problem is the quadratic relaxation created by relaxing the integer constraints to interval constraints, $x_1, x_2 \in [0, 1]$.

The ellipses shown in Fig. 4.2 are called *nodes*. In each node, the corresponding feasible set S_i is shown. The symbol * is used to denote that this variable is between 0 and 1. The rows of nodes in the tree are called *levels*. The top node is called the *root node*. In the tree all nodes, except the nodes in the bottom of the tree, have two nodes connected to the lower side of the node. These two nodes are called the *children* of the node; any node above a child node is called a *parent* node.

4.2 Bounding

Bounding tool is used to compute lower and upper bounds for the optimal objective function value for the subproblems in the nodes. If a partial solution from a subregion is greater than or equal to the upper bound it is discarded from the search. This step is called *pruning*. Often, the bounds can be used to prune entire subtrees.

There exist three different possibilities for pruning a subtree with root node [2] :

1. Infeasibility
2. Optimality
3. Dominance

The first case occurs when the sub-problem has no feasible solution. The second case occurs when an optimal solution to the sub-problem is found. If the solution of a

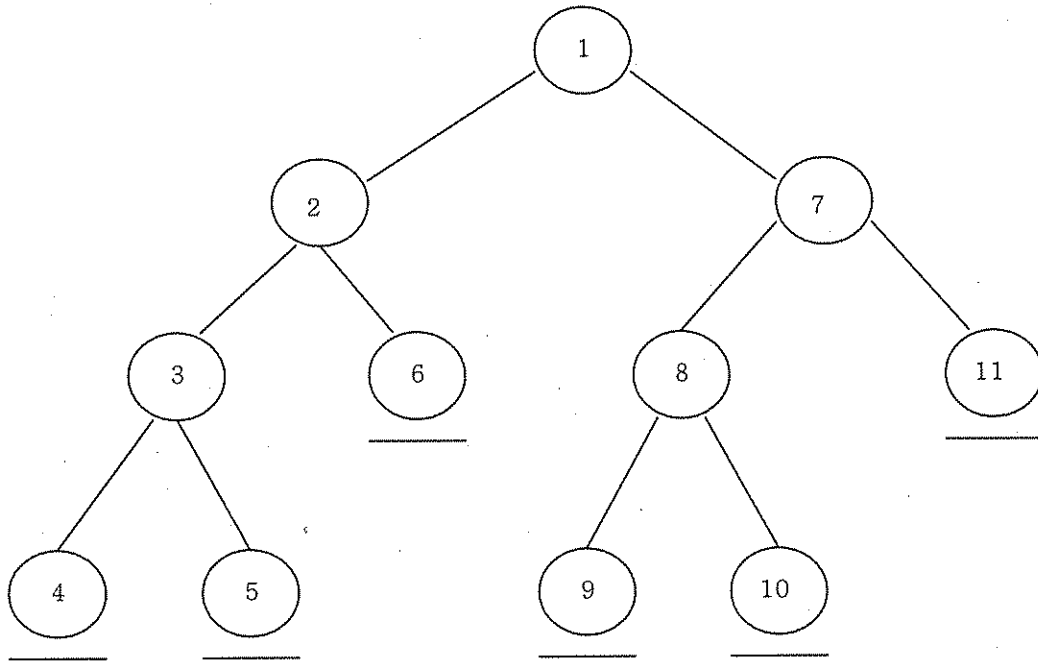
node is integral and the corresponding value is less than a global upper bound, \bar{z} , then \bar{z} should be updated to the value. The third case occurs when the solution of the sub-problem is no better than the current solution. If the value of node is greater than or equal to \bar{z} , then the node is pruned.

In the branch and bound method, there are several parameters and choices that may affect the performance drastically. Two important parameters are the choice of the next node to solve and the choice of the branch variable.

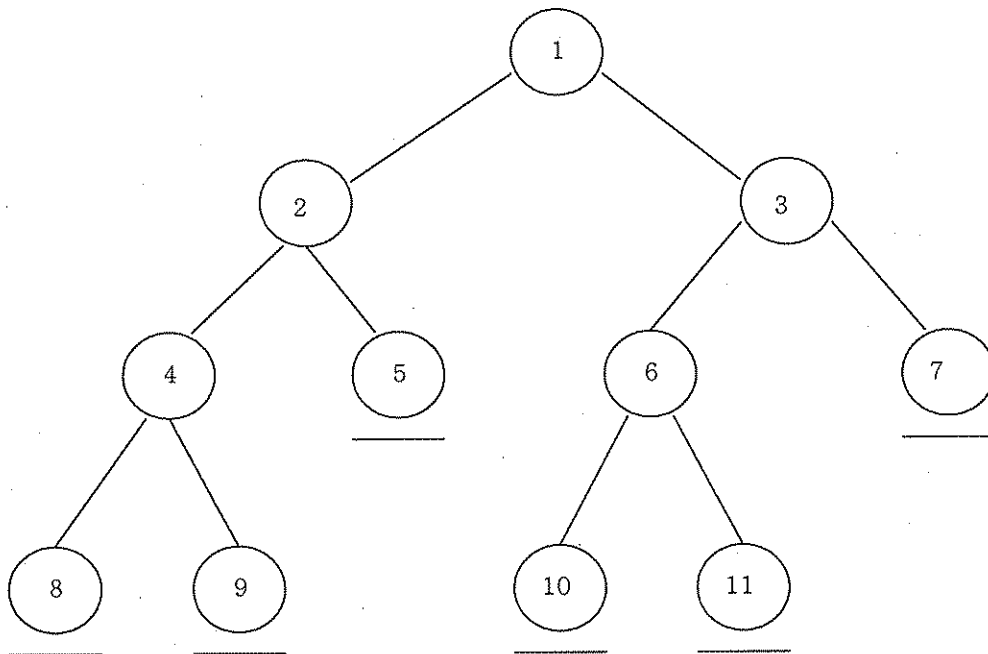
4.3 Node Selection

We will look at two strategies for selecting a next node [2]. The first is *Depth-first search plus backtracking*, which is also known as *last in, first out* (LIFO). In depth-first search, if the current node is not pruned, the next node considered is one of its two children. Backtracking means that when a node is pruned, we go back on the path from this node toward the root until we find the first node. An example of depth-first search plus backtracking with left-child-first is given in Figure 4.4 – (a). The nodes are numbered in the order in which they are considered. An underlined node is assumed to have been pruned.

The other one is *breadth-first search* (BFS). In this search algorithm all of the nodes at a given level are considered before any nodes at the next lower level. Figure 4.4 – (b) shows the BFS search tree. The number of nodes at each level of the search tree grows exponentially. Because the BFS algorithm enumerates all possible combinations for the variables, we will use depth-first search plus backtracking.



(a) Depth-First Search plus Backtracking



(b) Breadth-First Search

Figure 4.4 Search strategies for a next node [2]

4.4 Branching Variable selection

The next important parameter is how to select the next variable on which to branch. We will branch on the variable with a fractional value. If there is more than one fractional value, we have to make a decision. There is no standard rule for choosing the next variable on which to branch. Some may choose the fractional variable with the lowest or highest index. Others may choose the smallest or largest fraction value [8].

We will use a method that selects a variable which has a lowest function value determined by the quadratic function $\frac{1}{2}x^T Qx + cx$. We give an example in Figure 4.5. We get a feasible solution X from a quadratic relaxation. There are two fractional numbers in our solution x_1 and x_2 . Thus, we will branch on one of these. Choosing a branching variable, we have four subproblems setting $x_1 = 0$, $x_1 = 1$, $x_2 = 0$, and $x_2 = 1$. Now, we evaluate these subproblems using $\frac{1}{2}x^T Qx + cx$ and get the function values. Because the value 13 is the smallest value, we will branch on x_2 .

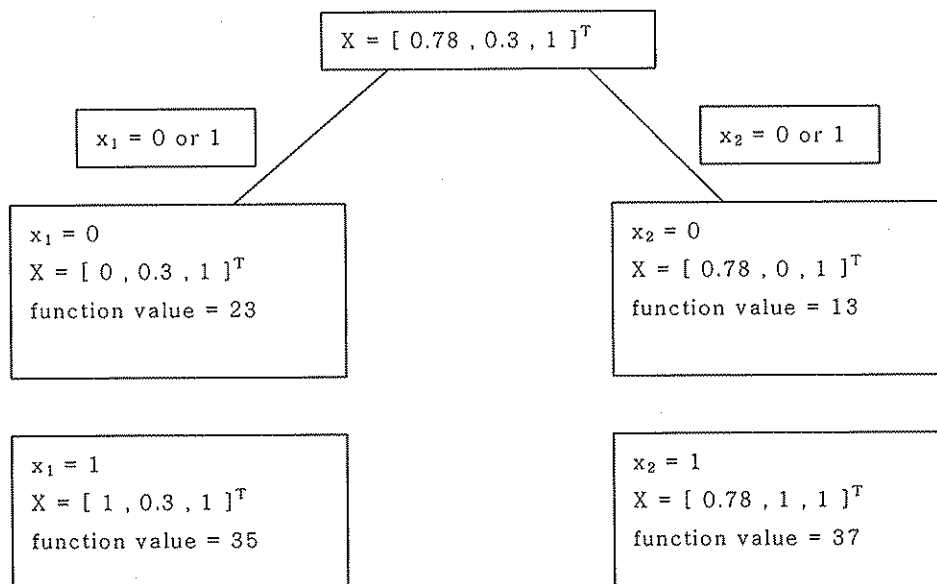


Figure 4.5 An example of selecting a branching variable.

4.5 Branch and Bound Algorithm

To simplify what follows, we make the following definition:

- P_i the optimization subproblem over the set S_i
- N_i the node containing P_i
- \bar{z} a global upper bound of the objective function value
- \underline{z} a global lower bound of the objective function value
- z^i the objective function value from sub-problem P_i
- x^i the feasible solution over the x^i

The branch and bound algorithm for a binary quadratic programming problem is shown in Figure 4.6 [2]

Step 1 (Initialization) :

Let the BQP problem (4.3) be stored as an element of a collection \mathcal{L} .

Set a lower and upper bound as $\underline{z} := -\infty$, $\bar{z} := +\infty$.

Step 2 (Termination test) :

If $\mathcal{L} = \emptyset$, stop the algorithm: the solution x that has an optimal value $z = \frac{1}{2}x^T Qx + cx$ is optimal.

Step 3 (Problem selection and relaxation) :

Select and delete a BQP problem from \mathcal{L} .

Solve its relaxation.

Let z^i be the value of the relaxation and let x^i be an solution of the relaxation.

If $\underline{z} := -\infty$, then \underline{z} should be updated to z^i .

Step 4 (Pruning) :

a. If $z^i \geq \bar{z}$, go to step 2

c. If $x^i \in \{1, 0\}$ and $z^i < \bar{z}$, the \bar{z} should be updated to z^i and go to step 2.

Otherwise go to step 5

Step 5 (Branching):

Let $\{S_{ij}\}_{j=1}^k$ be a division of S_i .

Add problem to \mathcal{L} , then go to step 2.

Figure 4.6 The branch and bound algorithm for a QBP problem.

5. Maximum-likelihood (ML) Detector

Driven by the demand for increasingly sophisticated "anywhere anytime" connectivity, wireless communications have emerged as one of the largest and most rapidly growing sectors of the global telecommunications industry [6]. In modern communication systems, multiple users share a multi-access channel called a *Multiple Input Multiple Output* (MIMO) channel [7], where the information sent by different users is separated by using almost certain codes. The process of simultaneously estimating the information sent by multiple users is called *multiuser detection* (MUD) [8]. To estimate the information originally sent, a *Maximum Likelihood* (ML) problem involving binary variables is solved. This requires the solution of a binary quadratic programming (BQP) problems [8].

5.1 Multiple Input Multiple Output (MIMO) channel

Consider the linear MIMO system diagram [14] shown in Figure 5.1. In the figure, x and x' are $M \times 1$ vectors, v and y are $N \times 1$ vectors, H is an $N \times M$ matrix. The components of x and x' are either -1 or 1 . We write this as $x, x' \in \{-1, 1\}^M$. The components of v and H are $N(0, 1)$.

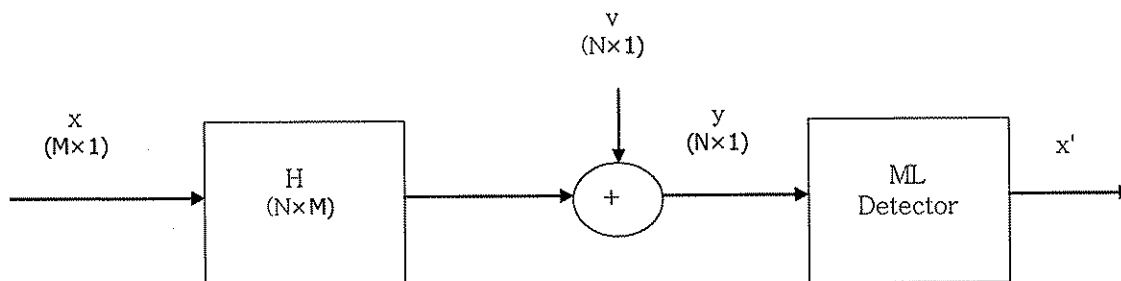


Figure 5.1 : A simplified linear MIMO communication system diagram showing the following discrete time signals: transmitted symbol vector x , channel matrix H , additive noise vector v , received vector y , and detected symbol vector x' .

To communicate over this channel, we are faced with the task of detecting a set of M transmitted symbols from a set of N observed signals. The vector x represents the transmitted signals. Each component of x is either -1 or 1 . The vector y represents the received signals. The $N \times M$ matrix H represents the channel matrix, which is assumed to be known at the receiver, and the noise is Additive White Gaussian Noise, which is represented by the $N \times 1$ vector v .

We associate with Fig. 5.1 the equation [3] :

$$y = Hx + n, \quad (5.2)$$

where $H = \sqrt{\frac{\rho}{m}} A$, ρ is the expected *signal-to-noise ratio* (SNR) at each receive antenna, m is the number of transmitters and A is the matrix of fading coefficients.

In modeling an MIMO channel it is convenient to think of the components of the transmitted vector x as 0 and 1 , not -1 and 1 .

5.2 Maximum Likelihood Detection

The ML Detection achieves the minimum error probability of detection. In general, ML Detector solves the following optimization problem: [3]

$$x_{ML} = \arg \max_{x \in \{0,1\}^M} P(y|x, A),$$

where $P(y|x, A)$ is the conditional probability density function of y and x_{ML} is the ML estimate of the transmitted signals.

The aim of ML detection is to find such a vector x that produces an output Hx which is closest to the received signal vector y [3], that is

$$\min_{x \in \{0,1\}^M} \|y - Hx\|^2. \quad (5.3)$$

This leads to the following optimization problem [8] :

$$\begin{aligned}
x_{ML} &= \arg \min_{x \in \{0,1\}^M} \|y - Hx\|^2, & (a) \\
&= \arg \min_{x \in \{0,1\}^M} (Hx - y)^T (Hx - y) & (b) \\
&= \arg \min_{x \in \{0,1\}^M} x^T H^T Hx - y^T Hx - x^T H^T y + y^T y & (c) \\
&\equiv \arg \min_{x \in \{0,1\}^M} x^T H^T Hx - 2y^T Hx & (d) \\
&= \arg \min_{x \in \{0,1\}^M} \frac{1}{2} x^T H^T Hx - y^T Hx & (e) \\
&= \arg \min_{x \in \{0,1\}^M} \frac{1}{2} x^T Qx + cx, \text{ where } Q = H^T H, c = -y^T H, & (f)
\end{aligned} \tag{5.4}$$

where x_{ML} minimizes the objective function $\|y - Hx\|^2$, $x \in \{0,1\}^M$.

If we expand the objective function shown in (a) we get (c). An equivalent form of (c) is shown in (e). We assume that $M \leq N$, and that H is of full rank M . Therefore, x_{ML} is the called ML Detection (MLD). This is a binary quadratic programming problem.

The MLD is a combinatorial optimization problem, which is solved by choosing the best combination out of all possible combinations. The most straightforward approach is to compute the optimal solution by enumerating all possible combinations. But, the computational burden for this approach is overwhelming. Our conclusion is that we need an algorithm that can find the optimal solution without enumerating all possible combinations. This algorithm is branch and bound, which is thoroughly described in Chapter 4.

5.3 Branch and Bound in Matlab

We generalize the branch and bound algorithm [2] from Fig. 4.6 in Figure 5.5.

$\underline{z} := -\infty, \bar{z} := +\infty$

```

Add  $P$  to  $\mathcal{L}$ 
while  $\mathcal{L} \neq \emptyset$     do
    Select and delete  $P_i$  from  $\mathcal{L}$ 
    Solve  $P_i^R \Rightarrow z^i$  and  $x^i$ 
    if  $\bar{z} := -\infty$ 
    then  $\bar{z} = z^i$     end if
    if  $z^i \geq \bar{z}$ 
    then pruning  $N_i$ 
    else if  $x^i \in \{1,0\}$  and  $z^i < \bar{z}$ 
    then  $\bar{z} = z^i$ 
    else
        Split  $S_i$  into  $S_{i0}$  and  $S_{i1}$ 
        Push  $P_{i0}$  and  $P_{i1}$  to  $\mathcal{L}$ 
    end if
end while

```

Figure 5.5 Branch and bound algorithm for binary quadratic programming problems

We programmed the branch and bound algorithm based on Fig. 5.5 using Matlab. The program is given in the Appendix. In the program, when we evaluate a quadratic relaxation we use “*quadprog*”, which is the built-in-solver in the optimization toolbox of Matlab. We create the binary quadratic problem shown in (5.4) by generating H and x randomly. Because $H \sim N(0, 1)$, we randomly generate the elements of H from standard normal distribution. We set $y = H * x$, Q is the symmetric matrix $H^T * H$, and $c =$

$-(y^T * H)$. The code is given in (5.6).

$$H = \text{randn}([m, m])$$

$$x = \text{rand}([m, 1])$$

$$y = H * x$$

$$Q = (H^T * H)$$

$$c = -(y^T * H)$$

Figure 5.6 The Matlab code for generating a wireless communication problem.

5.4 An Example of wireless communication problem

Fig. 5.6 gives a wireless communication problem in which we use ML detection to find a vector x that produces an output Hx which is closest to the received signal vector y . An example of wireless communication problem generated by Fig. 5.6 is shown in (5.7):

$$\begin{aligned} \text{(BQP)} \quad & \text{minimize} \quad f(x) = \frac{1}{2}x^T Qx + cx \\ & x_i \in \{0, 1\}, \quad i = 1, 2, 3, \end{aligned} \quad (5.7)$$

where x is an 3×1 vector, $Q = H^T H$,

$$\text{where } H = \begin{bmatrix} 1.0668 & -0.8323 & 0.7143 \\ 0.0593 & 0.2944 & 1.6236 \\ -0.0956 & -1.3362 & -0.692 \end{bmatrix}, \text{ and } Q = \begin{bmatrix} 1.1507 & -0.7427 & 0.9244 \\ -0.7427 & 2.5649 & 0.8078 \\ 0.9244 & 0.8078 & 3.6248 \end{bmatrix}$$

and $c = -(y^T * H)$, where $y = H * x$, where $x = \begin{bmatrix} 0.9355 & 0.9169 & 0.4103 \end{bmatrix}^T$

$$\text{and } c = \begin{bmatrix} -0.7747 & -1.9884 & -3.0926 \end{bmatrix}.$$

There is an associated quadratic program called the *quadratic relaxation* (QR) created by relaxing the integer constraints to interval constraints:

$$\begin{aligned} \text{(QR)} \quad & \text{minimize} && f(x) = \frac{1}{2}x^T Qx + cx \\ & && x \in [0, 1]. \end{aligned} \quad (5.8)$$

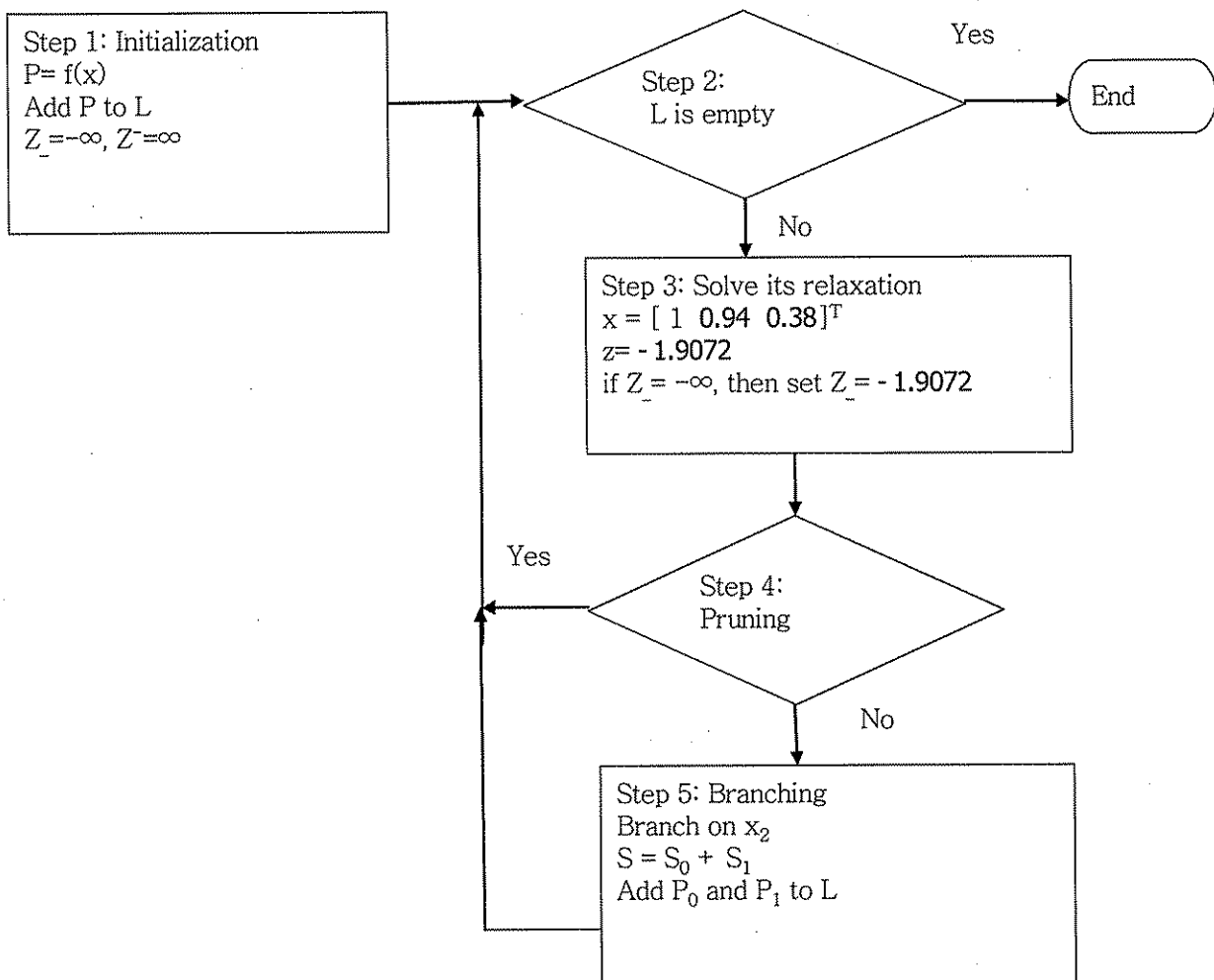


Figure 5.9 Solving the original problem P .

Figure 5.9 shows the first iteration of the problem. The solution from the

relaxation is $x_1 = 1, x_2 = 0.94, x_3 = 0.38$, with a value of -1.9072 , which is the lower bound. There are two fractional numbers in our solution. Choosing a branching variable, we set $x_2 = 0, x_2 = 1, x_3 = 0, x_3 = 1$ and evaluate these subproblems using $\frac{1}{2}x^T Qx + cx$. Because the subproblem with $x_2 = 1$ has the smallest value, we will branch on x_2 . Furthermore, we split S into S_0 and S_1 and store P_0 and P_1 to \mathcal{L}

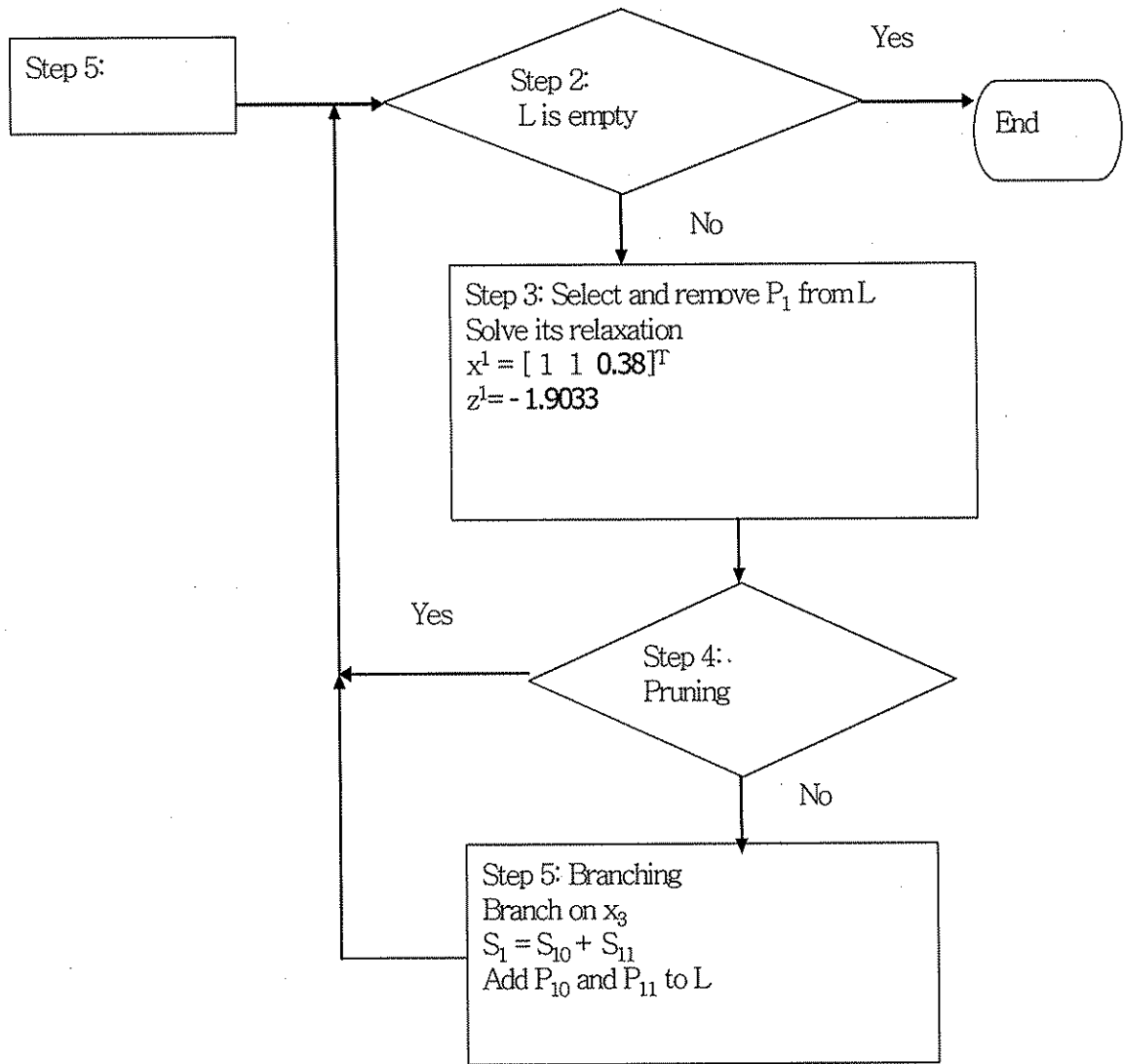


Figure 5.10 Solving the subproblem P_1 .

We select and delete P_1 from \mathcal{L} . The solution from the relaxation is $x_1 = 1,$

$x_2 = 1, x_3 = 0.38$, with a value of -1.9033 . We branch on x_3 because $x_3 = 0$ has the lowest function value. Furthermore, we split S_1 into S_{10} and S_{11} and store P_{10} and P_{11} to \mathcal{L} . Figure 5.10 shows this process.

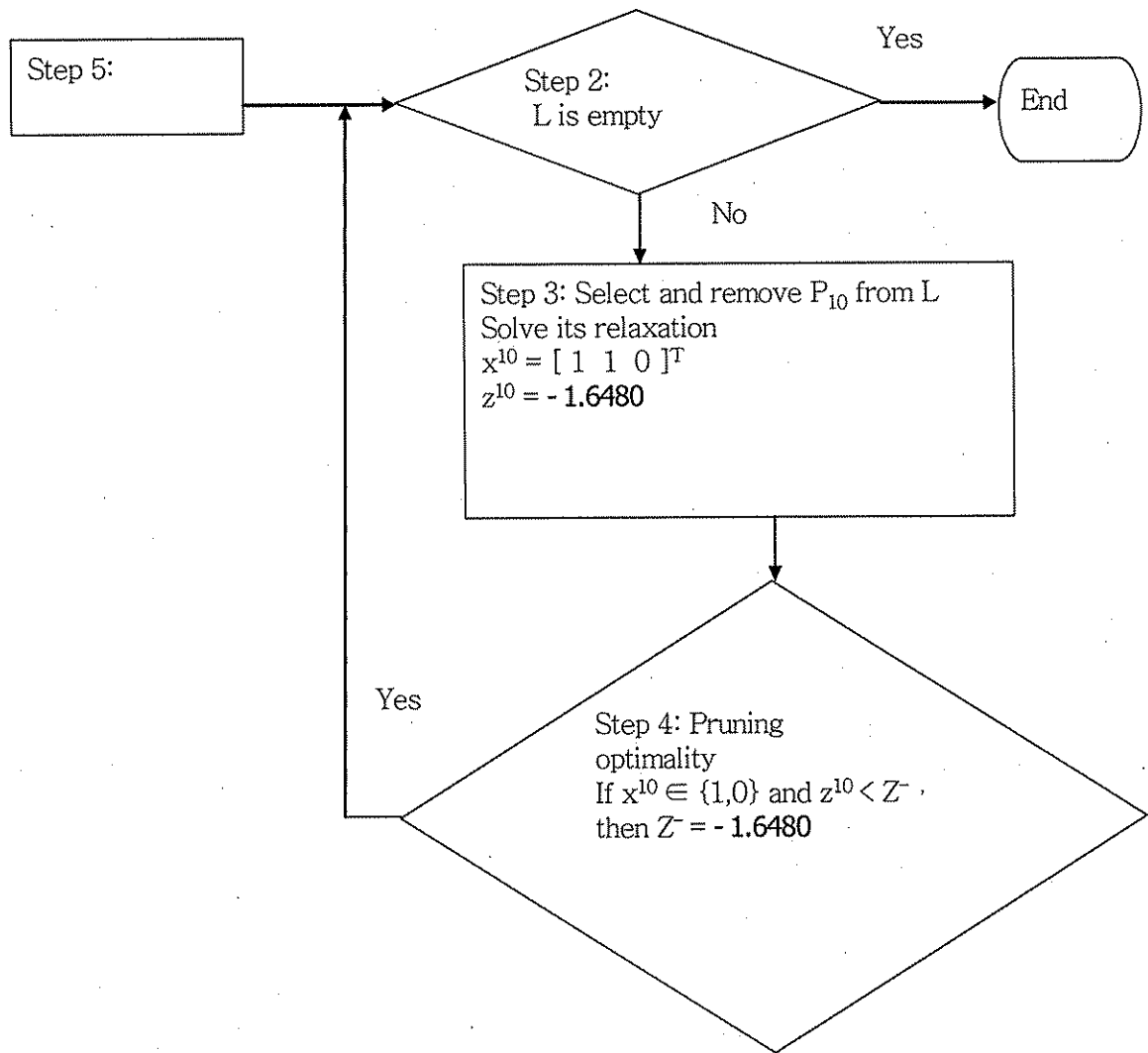


Figure 5.11 Solving the subproblem P_{10} .

Figure 5.11 shows the solution of subproblem P_{10} , which is the end of node N_{10} . The solution from the relaxation is $x = [1 \ 1 \ 0]^T$ with a value of -1.6480 . Because the solution is an integer and the value is the less than the upper bound, we prune this

node and go to step 2. Furthermore, we update the upper bound to -1.6480 .

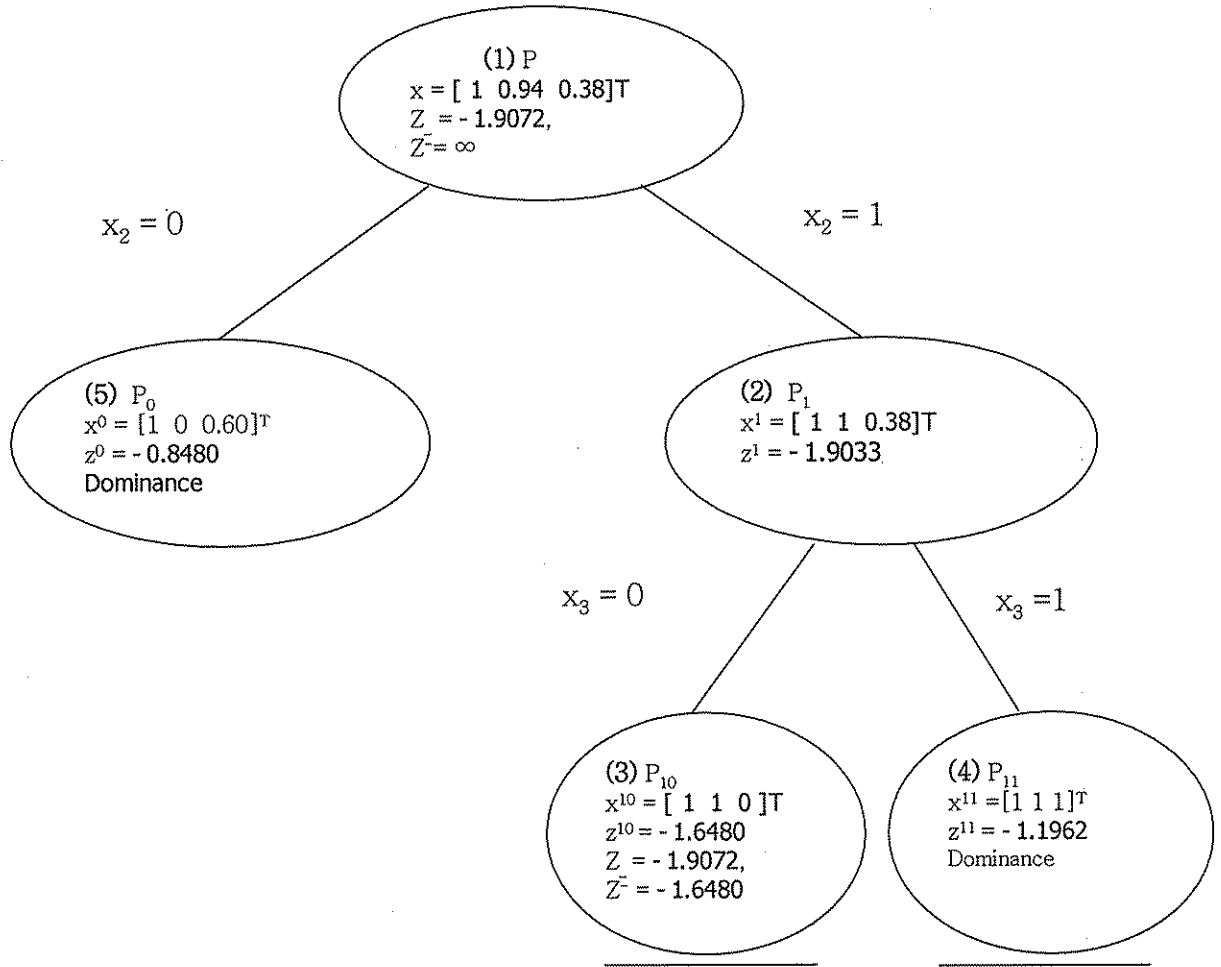


Figure 5.12 The branch and bound tree

We use Depth-First Search plus Backtracking strategy for selecting a next node.

Figure 5.12 shows the branch and bound tree for problem (5.7). The nodes are numbered in the order in which they are considered. An underlined node is assumed to have been pruned. After pruning N_{10} , we select and delete P_{11} from \mathcal{L} . The solution from the relaxation is $x = [1 1 1]^T$ with a value of -1.1962 . Because the value is greater than the upper bounds; so we prune this node.

Next, we select and delete P_0 from \mathcal{L} . The solution from the relaxation is $x = [1 \ 1 \ 0.6]^T$ with a value of -0.8480 . Because the value is greater than the upper bounds; so we prune this node. This completes the algorithm. Thus, we see that the optimal solution is $x = [1 \ 1 \ 0]^T$ with the value -1.6480 .

5.5 More Experiments

We randomly generated several wireless communication problems for each of $M = 20, 30,$ and 50 . We solved these problems using the branch and bound algorithm. Table 5.13 summarizes the result of this experiment. Specifically, it gives the average number of nodes visited and the average time needed to find the solution.

M	# of combinations (2^M)	# of nodes ($2^{M+1} - 1$)	# of nodes visited (average)	time to find solution (seconds)
20	1,048,576	2,097,151	69	2.457
30	1,073,741,824	2,147,483,647	123	12.6732
50	1,125,899,906,842,624	2,251,799,813,685,247	225	99.1875

Table 5.13 The result of this experiment.

We note that this is a limited experiment. We have not proved that this program gives the correct solution. However, we have some evidence of the correctness of our works. We have randomly generated several wireless communication problems (for $M = 6$). We solve these problems with our branch-and-bound algorithm. Our results agreed with the solution found with exhausted search, in which we chose the smallest value from 64 combinations. Therefore it appears that our program may give the optimal solution.

The test has been performed on an Intel Pentium M Processor 1.60 GHz with

752 Mb RAM running Microsoft Windows XP Professional Version 2005 and MatLab

6.1.0. The computational time was calculated using the Matlab command "*tfc*" and "*toc*".

6. Conclusion

Maximum Likelihood Detection is a binary quadratic programming problem, which is a combinatorial optimization problem, which is solved by choosing the best combination out of all possible combinations. But, the computational burden for this approach is overwhelming by increasing the problem size.. We used the branch and bound algorithm to solve this MLD problem. Branch and bound algorithm enumerates only some of the possible combination using quadratic relaxation and branching and bounding tools. Although there are other approaches for solving MLD problem, we chose the branch and bound algorithm because it is easy to solve and it gives the exact optimal solution.

We programmed the branch and bound algorithm using Matlab. The program is given in the Appendix. In this program, we use matlab comments, which are the built-in-solver in the optimization toolbox of Matlab. We generated the binary quadratic problems for each of $M = 20, 30,$ and 50 by generating H and x randomly. We solved these problems with our branch-and-bound algorithm. The result of this experiment showed in table 5.13. We have not proved that this program gives the correct solution. However, we have some evidence of the correctness of our works. We have randomly generated several wireless communication problems (for $M = 6$). We solve these problems with our branch-and-bound algorithm. Our resultss agreed with the solution found with exhausted search, in which we chose the smallest value from 64 combinations. Therefore it appears that our program may give the optimal solution.

Bibliography

- [1] Vasek Chvatal. *Linear Programming*. Freeman, 1983.
- [2] Laurence A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., 1988.
- [3] Mikalai Kisialiou and Zhi-Quan Luo. Performance Analysis of Quasi-Maximum-Likelihood Detector Based on Semi-Definite Programming. *IEEE International Conference on Volume 3*, Pages iii/433 – iii/436, March 2005.
- [4] Babak Hassibi and Haris Vikalo. On the Sphere-Decoding Algorithm I. Expected Complexity. *IEEE Transactions on Signal Processing*, volume 53, pages 2806 – 2818, August 2005.
- [5] Joakim Jalden and Bjorn Ottersten. An Exponential Lower Bound on the Expected Complexity of Sphere Decoding, *Proc. ICASSP '04*, vol. 4, pp. 393-396, 2004.
- [6] T. Rappaport, A. Annamalai, R. Buehrer, and W. Tranter. *Wireless Communications: Past events and a future perspective*. *IEEE Communications Magazine*, 40(5): 148-161, May 2002.
- [7] Lai-U Choi and Ross D. Murch. A Transmit Preprocessing Technique for Multiuser MIMO Systems Using a Decomposition Approach, *IEEE Transactions on Wireless Communications*, vol. 3, pp. 20 – 24, 2004.
- [8] Daniel Axehill. *Applications of Integer Quadratic Programming in Control and Communication*. Linkopings universitet, 2005.
- [9] Wing-Kin Ma, Timothy N. Davidson, Kon Max Wong, Zhi-Quan Luo, and Pak-Chung Ching. Quasi-Maximum-Likelihood Multiuser Detection Using Semi-Definite Relaxation With Application to Synchronous CDMA. *IEEE Transactions on Signal Processing*, volume 50, pages 912 – 922, April

2002.

- [10] Jens Causen. Branch and Bound Algorithms-Principles and Examples. University of Copenhagen, March 1999.
- [11] Wikipedia. the free encyclopedia. (<http://en.wikipedia.org/wiki>)
- [12] Spyros Reveliotis. An Introduction to Linear Programming and the Simplex Algorithm. (<http://www2.isye.gatech.edu/~spyros/LP/LP.html>)
- [13] Optimization Technology Center of Northwestern University and Argonne National Laboratory linear programming website. (<http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html>)
- [14] Michael A. Trick. Tutorial on Integer. (<http://mat.gsia.cmu.edu/orclass/integer/node1.html>)
- [15] Karen Su. Efficient Maximum Likelihood Detection For Communication Over Multiple Input Multiple Output Channels. Department of Engineering, University of Cambridge.

Appendix

Branch and Bound in Matlab

```
function [optimalX,optimalZ] = qq_42306(n)
tic

% generating H and x randomly

format short
H = rand([n,n]);
x = rand([n,1]);

% creating a wireless communication problem

y = H*x;
Q = (H'*H)
c = -(y'*H)
f = c;

% Basic set up

lb = zeros(n,1); % lower bound
ub = ones(n,1); % upper bound
Aeq = zeros(n,n); % a matrix showed the selection of the variables
beq = zeros(n,1); % a column vector showed the value of the 'x'.
optimalX = 0; % optimal solution
optimalZ = 0; % optimal value
fsolutionX = 0; % feasible solution
t = 0; % the number of non integers entries in 'x'.
xx = []; % a row vector specifying whether the entries of 'x'
           are integers or not (0 if integer; 1 if not)
ss = zeros(n,n); % a matrix used to show selected node
sinf = zeros(1,n); % a row vector used to select a branching variable
    for j = 1:1:n % setting the entries of 'sinf' as infinite
        sinf(1,j) = inf;
    end
a1=0;
```

% step 1 (Initialization)

```
Z_ = -inf;          % initial lower bound
Z  = inf;          % initial upper bound
```

%1st iteration

```
j = 1;
```

```
%called matlab build in program "quadprog", which returns a vector 'x'
and a value of the function 'z' that minimizes '1/2*x'*Q*x + f*x'
```

```
[x,z] = quadprog(Q,f,[],[],[],[],lb,ub);
```

```
% call subroutine "rd", which rounds 'x' to two decimal places
```

```
x = rd(x);
```

```
%called subroutine "makex", which determines how many non integer
entries in 'x'
```

```
[xx,t,Aeq,beq] = makex(x,xx,n,t,Aeq,beq);
```

```
s = sinf;          % a row vector used to select a branching variable
```

```
p = zeros(1,n);    % a row vector used to select a branching variable
```

```
if (t > 0)         % If there are non integer entries in 'x'
```

```
%called subroutine "evalqual", which do step 3 and step 5
```

```
[x,z,ss,Aeq,beq] = evalqual(j,x,xx,Q,f,n,Aeq,beq,lb,ub,s,p,ss);
```

```
Z_ = z;
```

```
else              % If all entries in 'x' is integral, we find optimal solution
```

```
and value
```

```
optimalX = x;
```

```
optimalZ = z;
```

```
end
```

```
[xx,t,Aeq,beq] = makex(x,xx,n,t,Aeq,beq);
```

```
if t == n
```

```
    w = 0;
```

```
else
```

```
    w = 1;
```

```
end
```

```
ww = 1;
```

```
while (max(max(ss(ww:n,1:n))) ~= 0) % step 2 (Termination Test)
```

```
    ww = 1+ w;
```

```
    j = j+1 % counting the numbers of nodes visited
```

```
    [xx,t,Aeq,beq] = makex(x,xx,n,t,Aeq,beq);
```

```
    ss
```

```
    Aeq
```

```
    beq
```

```
% step 3 (Problem Selection & Relaxation) & step 4 (Pruning)
```

```
    if (t == 0)
```

```
        if (Z == inf) % optimality
```

```
            '1 optimality'
```

```
            Z = z ; % updating the upper bound to z
```

```
            optimalZ = Z
```

```
            optimalX = x
```

```
        %called subroutine "goback", which do step 3 and step 4
```

```
        [x,z,ss,Aeq,beq,uu,ii,tarc,tarl,j]
```

```
        goback(n,j,w,Aeq,beq,s,ss,p,Q,f,lb,ub,optimalZ,optimalX,x,z);
```

```
        AAeq = Aeq;
```

```
        bbeq = beq;
```

```
    elseif ((Z ~= inf) & (z < Z)) % optimality
```

```
        '2 optimality'
```

```
        Z = z ; % updating the upper bound to z
```

```
        optimalZ = Z
```

```
        optimalX = x
```

```
        ss(tarc,tarl)=0
```

```
        Aeq(tarl,tarl)=0;
```

```
        beq(tarl,1)=0;
```

```
        [x,z,ss,Aeq,beq,uu,ii,tarc,tarl,j]
```

```
        goback(n,j,w,Aeq,beq,s,ss,p,Q,f,lb,ub,optimalZ,optimalX,x,z);
```

```
        AAeq = Aeq;
```

```

bbeq = beq;

elseif (z >= Z)      % dominance
'3 dominance'
    for a = 1:1:n
        for b = 1:1:n
            if Aeq(a,b) ~= AAeq(a,b)
                Aeq = AAeq;
                beq = bbeq;
            end
        end
    end
    end
    a1=0;
    ss(tarc,tarl)=0;
    Aeq(tarl,tarl)=0;
    beq(tarl,1)=0;
    for a = tarc+ 1:1:n
        for b = 1:1:n
            if ss(a,b) ~= 0
                ss(a,b) = 0;
                Aeq(b,b)=0;
                beq(b,1)=0;
            end
        end
    end
    end

    [x,z,ss,Aeq,beq,uu,ii,tarc,tarl,j] =
goback(n,j,w,Aeq,beq,s,ss,p,Q,f,lb,ub,optimalZ,optimalX,x,z);
    if (max(max(ss(w+ 1:n,1:n))) ~= 0)
        Aeq(tarl,tarl)=0;
        beq(tarl,1)=0 ;
    end
    AAeq = Aeq;
    bbeq = beq;
end

elseif (t > 0)

```

```

if (z >= Z)          % dominance
    '4 dominance'
    for a = 1:1:n
        for b = 1:1:n
            if Aeq(a,b) ~= AAeq(a,b)
                Aeq = AAeq;
                beq = bbeq;
            end
        end
    end
    end
    a1=0;
    ss(tarc,tarl)=0;
    Aeq(tarl,tarl)=0;
    beq(tarl,1)=0;
    for a = tarc+ 1:1:n
        for b = 1:1:n
            if ss(a,b) ~= 0
                ss(a,b) = 0;
                Aeq(b,b)=0;
                beq(b,1)=0;
            end
        end
    end
    end
    [x,z,ss,Aeq,beq,uu,ii,tarc,tarl,j] =
goback(n,j,w,Aeq,beq,s,ss,p,Q,f,lb,ub,optimalZ,optimalX,x,z)
    a1 = tarc
    if (max(max(ss(w+ 1:n,1:n))) ~= 0)
        Aeq(tarl,tarl)=0
        beq(tarl,1)=0
    end
    AAeq = Aeq;
    bbeq = beq;

% step 3 (Problem Selection & Relaxation) & step 5 (branching)
elseif (Z == inf)
    '5 branching'

```

```

[x,z,ss,Aeq,beq] = evalqual(j,x,xx,Q,f,n,Aeq,beq,lb,ub,s,p,ss);
elseif ((Z ~= inf) & (z < Z))
'6 branching'
for a = 1:1:n
    for b = 1:1:n
        if Aeq(a,b) ~= AAeq(a,b)
            Aeq = AAeq;
            beq = bbeq;
        end
    end
end
a1 = uu
while ((t > 0) & (z < Z))
    [x,z,ss,Aeq,beq,a1] =
evalqual2(a1,x,xx,Q,f,n,Aeq,beq,lb,ub,s,p,ss)
    [xx,t,Aeq,beq] = makex(x,xx,n,t,Aeq,beq)
end
end
end
end
optimal_solution = optimalX
optimal_value = optimalZ "
j=j-1
end
toc

```

```
function x = rd(x)           %rounding 'x' to two decimal places.
x = (round(100*x))*(1/100);
```

```
function [xx,t,Aeq,beq] = makex(x,xx,n,t,Aeq,beq)
% determining how many non integer entries in 'x'
% if 'x(i)' is 0 (or <0.0001) or 1 (or >0.9999) then indicating the entries of
'xx(i)' as "0"
% if 'x(i)' is non integer then indicating the entries of 'xx(i)' as "1"
% the initial value of t is 0 and t shows how many non integer entries in
'x'

xx=[];   t=0;
  for i = 1:1:n
    if ((x(i) == 1) | (x(i) > 0.9999)) % bigger than 0.9999 considered
as a value '1'
      xx(i) = 0;
      Aeq(i,i)=1;
      beq(i,1)=1;
    elseif (x(i) < 0.0001) % smaller than 0.01 considered as a value
'0'
      xx(i) = 0;
      Aeq(i,i)=1;
      beq(i,1)=0;
    else
      xx(i) = 1;
      t = t+ 1;
    end
  end
end
```

```

function [x,z,ss,Aeq,beq] = evalqual(j,x,xx,Q,f,n,Aeq,beq,lb,ub,s,p,ss)
% step 3 (Problem Selection & Relaxation) & step 5 (branching)
c = 0;
    for i = 1:1:n
        if xx(i) == 1
            X = x;                                q = [];
            X(i) = 0;
            q(i) = (1/2)*X'*Q*X+f*X;
            X(i) = 1;
            q(i+1) = (1/2)*X'*Q*X+f*X;
            if q(i) == q(i+1)    % if the value from "0" and "1" are
same
                s(1,i) = q(i);    % then picking the first one
                p(1,i) = 11;    % selecting the indication as "11"
(selecting).
            else    % if value from "0" and
"1" are differ,
                s(1,i) = min(q(i),q(i+1)); % then picking the small
value and saving in the matrix "s"
                if s(1,i) == q(i)
                    p(1,i) = 10;
                elseif s(i) == q(i+1)
                    p(1,i) = 11;
                end
            end
        end
    end
    [c,ii] = min(s);    Aeq(ii,ii) = 1;
    if p(1,ii) == 10
        beq(ii,1) = 0;
        ss(j,ii) = 10;
    elseif p(1,ii) == 11
        beq(ii,1) = 1;
        ss(j,ii) = 11;
    end
[x,z] = quadprog(Q,f,[],[],Aeq,beq,lb,ub)

```

```

function [x,z,ss,Aeq,beq,uu,i,tarc,tarl,j] =
    goback(n,j,w,Aeq,beq,s,ss,p,Q,f,lb,ub,minvalue,solutionX,x4,fval4)
% step 3 (Problem Selection & Relaxation) & step 4 (Pruning)
tarc=0;
tarl=0;
    for i = 1:1:n
        if ss(n,i) ~= 0
            ss(n,i) = 0
            Aeq(i,i) = 0
            beq(i,1) = 0
            j = j + 1
        end
    end
aaa = 0;
    for u = n-1:-1:1
        for I = 1:1:n
            if ss(u,i) ~= 0
                if aaa == 0
                    uu=u;
                    ii=i;
                    aaa = aaa + 1;
                end
            end
        end
    end
    if max(max(ss(w+1:n,1:n))) ~= 0
        if ss(uu,ii) == 10 % going back to previous node.
            beq(ii,1) = 1
        elseif ss(uu,ii) == 11
            beq(ii,1) = 0
        end
        tarc=uu
        tarl=ii
        [x,z] = quadprog(Q,f,[],[],Aeq,beq,lb,ub)
    end
end

```

```

function [x,z,ss,Aeq2,beq2,a1] =
    evalqual2(a1,x,xx,Q,f,n,Aeq2,beq2,lb,ub,s,p,ss)
    c = 0
    a1 = a1 + 1
    for i = 1:1:n
        if xx(i) == 1
            X = x;
            q = [];
            X(i) = 0;
            q(i) = (1/2)*X'*Q*X + f*X;
            X(i) = 1;
            q(i+1) = (1/2)*X'*Q*X + f*X;
            if q(i) == q(i+1)
                s(1,i) = g(i);
                p(1,i) = 11;
            else
                s(1,i) = min(q(i),q(i+1));
                if s(1,i) == q(i)
                    p(1,i) = 10;
                elseif s(i) == q(i+1)
                    p(1,i) = 11;
                end
            end
        end
    end
    [c,ii] = min(s)
    a1
    Aeq2(ii,ii) = 1
    if p(1,ii) == 10
        beq2(ii,1) = 0
        ss(a1,ii) = 10
    elseif p(1,ii) == 11
        beq2(ii,1) = 1
        ss(a1,ii) = 11
    end
    a2 = ii;    [x,z] = quadprog(Q,f,[],[],Aeq2,beq2,lb,ub)

```