1

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

National Library
of Canada

Bibliothèque nationale
du Canada

0-315-22226-3

Canadian Theses Division

Division des thèses canadiennes

Ottawa Canada
K1A 0N4

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

- Please print or type — Ecrire en lettres moulees ou dactylographier

Full Name of Author — Nom complet de l'auteur

AWAN RATTANAVINYOO

Date of Birth — Date de naissance

14/03/57

Country of Birth — Lieu de naissance

THAILAND.

Permanent Address — Residence fixe

534/57-58 SURESANAT RD. BANGPRAKOK BANGKOK THAILAND 10140.

Title of Thesis — Titre de la these

An Integer Linear Programming problem with a fixed number.
within Polynomial time.

University — Université

CARLETON University

Degree for which thesis was presented — Grade pour lequel cette these fut presentee

Master degree (Information and System Sciences).

Year this degree conferred — Année d'obtention de ce grade

Name of Supervisor — Nom du directeur de these

Prof. Brian Montmer

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film

The author reserves other publication rights and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission

L'autorisation est, par la presente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette these et de prêter ou de vendre des exemplaires du film

L'auteur se réserve les autres droits de publication, ni la these ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur

9/01/85.

A. Rattany

Date

Signature

**AN ALGORITHM FOR SOLVING AN INTEGER LINEAR PROGRAMMING PROBLEM
WITH A FIXED NUMBER OF VARIABLES
IN POLYNOMIAL TIME**

by

A. RATTANAVINYOO

A thesis submitted to the Faculty
of Graduate Studies and Research in
partial fulfillment of the requirements
for the degree of Master of Science

Department of Mathematics and Statistics

Carleton University

Ottawa, Ontario

December 1984

The undersigned recommend to the
Faculty of Graduate Studies and
Research acceptance of the thesis

**AN ALGORITHM FOR SOLVING AN INTEGER LINEAR PROGRAMMING PROBLEM
WITH A FIXED NUMBER OF VARIABLES
IN POLYNOMIAL TIME**

submitted by A. Rattanavinyoo, B.Sc.
in partial fulfillment of the requirements
for the degree of Master of Science

Thesis Supervisor

Chairman, Department of Mathematics
and Statistics

Carleton University

ABSTRACT

This thesis begins by giving background information on Integer Linear Programming (ILP) and Lattices including the methods of solving an ILP problem such as *Rounding Method*, *Branch and Bound Method* and *Cutting Plane Method*.

An ILP problem is an NP Complete problem. There is no known algorithm to solve an ILP problem which has a running time bounded by a polynomial function of the input length. H.W. Lenstra has defined an algorithm to solve an ILP problem for a fixed number of variables. The complexity of his algorithm is bounded by a polynomial function of the input length.

The details of Lenstra's algorithm are described in this thesis. His algorithm based on Geometry of Numbers, [Lattices]. The idea of the algorithm is to transform the ILP problem into an equivalent one and also transform the integer points into lattice points. This transformed problem has the following additional property: either the existence of the integer solution in the feasible region of the ILP problem is obvious or it is known that the last coordinate of the integer solution belongs to an interval whose length is bounded by a constant only depending on the number of variables. An important part of Lenstra's algorithm is a Basis Reduction process for lattices which is of independent interest.

ACKNOWLEDGEMENT

I would like to take this opportunity to thank Professor B. Mortimer for his encouragement and all the valuable advice during the span of this thesis. Also I would like to thank my family and my friends for their support.

satisfies the integer requirements (1.2), it is an optimum solution of the ILP and we are done. On the other hand, if one or more of the x_j , ($1 \leq j \leq n$), are non-integer, $\bar{x}$ is not feasible for the ILP.

One may be tempted to conclude that the optimal solution of the ILP can be obtained by rounding off $\bar{x}$ to a neighbouring integer solution. This rounding off method may be a practical method to use, and may give an excellent approximation to the optimum integer solution. This is clearly the case when the solution values are large numbers. Suppose, for example, an integer programming problem in three variables attacked via the simplex algorithm, yields the following solution:

$$x_1 = 326.2, \quad x_2 = 1074.9, \quad x_3 = 195.5.$$

It seems a fair assumption that the value of the objective function in an optimum integer programming solution to this problem would not be significantly different from that produced by the following solution, obtained by rounding

$$x_1 = 326, \quad x_2 = 1075, \quad x_3 = 196.$$

Truncating the fractional part of non-integer answers may also work. Consider the programming problem in two variables given in *Figure 1.1*.

If both x_1, x_2 are to be integers, the optimum solution is clearly $x_1 = x_2 = 1$, and f is maximized at f_1 (as shown in *Figure 1.1*). By truncating the fractional part of the non-integer solution, we obtain an optimum integer solution.

However, rounding and truncating will sometimes get us into trouble. For example, if we change the second constraint in problem 1.1, we can obtain such a problem as in Problem 1.2.

As a linear programming problem, the optimum solution is $x_1 = 1.87, x_2 = 1.87$. Clearly from *Figure 1.2*, if the problem were altered by the added requirement that both x_1 and x_2 must be

INTRODUCTION

Integer linear programming (ILP) is the following optimization problem.

$$\text{maximize } cx$$

$$\text{subject to } Ax \leq b$$

$$x \geq 0, \text{ integers,}$$

where A is an $(m \times n)$ -matrix with integral coefficients, b is an m -row vector with integral coefficients, m, n are positive integers, and c is a cost function. The problem is to find an n -vector x with integral coordinates $x_j, (1 \leq j \leq n)$, satisfying the inequalities $Ax \leq b$ and maximizing cx . Since the ILP problem is a NP-Complete problem, there is no known algorithm for finding a solution vector x of this problem which has a running time that is bounded by a polynomial function of the length of the data. This length may (details in Chapter 2) be defined to be $nm \log(a + 2)$ where a denotes the maximum of the absolute values of the coefficients of A and b .

Consider any ILP problem. In the case $n = 1$, it is trivial to design a polynomial algorithm for the solution of the problem. For $n = 2$, Hirschberg and Wong [8] and Kannan [9] have given polynomial algorithms in special cases which we shall discuss later in Chapter 4. In this thesis, we will present the algorithm defined by H.W. Lenstra. He shows that for any fixed value of n there exists a polynomial time algorithm for the solution of the ILP problem. The running time of the algorithm is bounded by a polynomial function of the length of the input.

In fact, his algorithm solves the linear inequality problem instead of the ILP problem. Since solving an ILP problem is no more difficult than solving a system of linear inequalities. We can develop a solution

to the ILP problem from this algorithm in the following way. First, solve the ILP problem by relaxing the integer constraints (this can be done by using simplex method or an ellipsoid method). Secondly, apply this algorithm to the ILP problem, the result of this algorithm being to decide whether there is at least one integer point in the feasible region. If there is one then we can reduce the problem to a new ILP problem with a smaller feasible region. A new ILP problem is constructed from the original constraints and a new constraint which is $cx \geq cz_1$ where z_1 is $(x_1 + z_2)/2$ and x_1 is the optimal solution for the ILP problem by relaxing the integer constraints, z_2 is any integer point in the feasible region of the original ILP found by applying this algorithm, and c is a cost function. This is a binary search algorithm, repeat step 2 and solve for an integer solution of the new problem (if there exists one); otherwise, construct a new problem with the original constraints and two more constraints which are $cx \geq cz_1$ and $cx \leq cz_2$, and solve for an integer point of this problem. Continue recursively, until we conclude that there are no more integer points in the new feasible region then we terminate with the last integer point as the optimal solution for the ILP problem. If there does not exist any integer point in the original problem (no such z_2) then terminate with the conclusion that the ILP problem has no solution.

The complexity of solving an ILP problem by the above method is also bounded by a polynomial in the length of the input. To see this; in step 1, it is clear that it can be done within a time bounded by a polynomial in the length of the input. For step 2, the number of times that we search for integer points between x_1 and x_2 is at most $\log(x_1 - x_2)$. Thus there are at most $\log(x_1 - x_2)$ iterations of the algorithm to find any integer point in the feasible region. This algorithm can be proved (later in this thesis) to have a running time that is bounded by a polynomial function of the length of the input.

Our algorithm is described in Section 3.1. We want to find an integer point satisfying the system

$Ax \leq b$. We will transform the problem into a new problem by a non-singular linear transform (r), and transform Z^n to a lattice $\mathcal{L}$ by r . Now we have an equivalent problem such that $A'y \leq b$, where $A' = Ar^{-1}$, solve for $y \in \mathcal{L}$, if there exists one. In this new formulation the algorithm proceeds as follows: either the existence of a vector $y \in \mathcal{L}$ is obvious, which mean that there exists a vector $y = y_1 u_1 + \dots + y_n u_n$, where $y_1, \dots, y_n$ are integers, and $u_1, \dots, u_n$ is a basis for $\mathcal{L}$, satisfying $A'y \leq b$, or it is known that the last coordinate of any such y belongs to an interval whose length is bounded by a constant only depending on n . In the latter case, the problem is reduced to a bound number of lower dimensional problems

Before the transformation can be made, we need, first, to remodel the convex set $\{x \in \mathbb{R}^n : Ax \leq b\}$ and construct the linear transformation r ; this algorithm is given in *Step 1* and *Step 2* of the algorithm in Section 2.1. Secondly, we will change a given basis for a lattice ($\mathcal{L}$) into one satisfying the inequality $\prod_{i=1}^n |b_i| \leq c_2 d(\mathcal{L})$, where $b_1, \dots, b_n$ is a basis for $\mathcal{L}$, c_2 is a constant only depending on n , and $d(\mathcal{L})$ is the determinant of $\mathcal{L}$. This latter algorithm is known as the basis-reduction process for n -dimensional lattices. It is described in Chapter 3.

We let K be the convex set defined by the system $Ax \leq b$, so that $K = \{x \in \mathbb{R}^n : Ax \leq b\}$. Our problem is to find an integer vector in K , that is, an integer solution to $Ax \leq b$. We apply an appropriate r to K such that rK contains a regular simplex (this is a better shape, because rK is equilateral), and also apply an appropriate r to Z^n forming a lattice ($\mathcal{L}$). Now, we can decide whether rK contains a lattice point, i.e. if $rK \cap \mathcal{L} \neq \emptyset$. To do so, we need to get a basis of $\mathcal{L}$ such that $\prod_{i=1}^n |b_i| \leq c_2 d(\mathcal{L})$, as mentioned before. With all these preparations, we get a new problem with additional properties, and we can decide whether K contains an integer point. At the end of Chapter 2,

we will show that this can be done in a time bounded by a polynomial in the length of the input for a fixed value of n .

As P. Van Emde Boas observed, it follows directly from *Leitner's* main result that the ILP problem with a fixed value of m is also polynomial solvable, the detail is in Section 4.1.

Chapter 1

BASIC CONCEPTS

In this Chapter, we would like to introduce some concepts and some definitions used throughout this thesis, and also mention integer linear programming.

1.1 Integer linear programming (ILP) problems

An integer linear programming problem is :

$$\text{maximize } cz$$

$$\text{subject to } Ax \leq b \quad (1.1)$$

$$x \geq 0$$

$$x \text{ an integer vector.} \quad (1.2)$$

where A is a matrix of order $m \times n$. The problem obtained by relaxing the integer requirement (1.2) is known as the associated linear programming problems (LP). The following are methods of solving ILP problems.

1.1.1 Rounding off and truncating method

Solve the associated LP, by the simplex method for example. If its optimum solution,

$$z = (z_1, z_2, \dots, z_n).$$

satisfies the integer requirements (1.2), it is an optimum solution of the ILP and we are done. On the other hand, if one or more of the x_j , ($1 \leq j \leq n$), are non-integer, $\bar{x}$ is not feasible for the ILP.

One may be tempted to conclude that the optimal solution of the ILP can be obtained by rounding off $\bar{x}$ to a neighbouring integer solution. This rounding off method may be a practical method to use, and may give an excellent approximation to the optimum integer solution. This is clearly the case when the solution values are large numbers. Suppose, for example, an integer programming problem in three variables, attacked via the simplex algorithm, yields the following solution:

$$x_1 = 326.2, \quad x_2 = 1074.9, \quad x_3 = 195.5.$$

It seems a fair assumption that the value of the objective function in an optimum integer programming solution to this problem would not be significantly different from that produced by the following solution, obtained by rounding :

$$x_1 = 326, \quad x_2 = 1075, \quad x_3 = 196.$$

Truncating the fractional part of non-integer answers may also work. Consider the programming problem in two variables given in *Figure 1.1*.

If both x_1, x_2 are to be integers, the optimum solution is clearly $x_1 = x_2 = 1$, and f is maximized at f_2 (as shown in *Figure 1.1*). By truncating the fractional part of the non-integer solution, we obtain an optimum integer solution.

However, rounding and truncating will sometimes get us into trouble. For example, if we change the second constraint in problem 1.1, we can obtain such a problem as in *Problem 1.2*.

As a linear programming problem, the optimum solution is $x_1 = 1.87, x_2 = 1.87$. Clearly from *Figure 1.2*, if the problem were altered by the added requirement that both x_1 and x_2 must be

Problem 1.1 :

$$\text{maximize } f = x_1 + 3x_2$$

$$\text{subject to } x_1 \leq 1.87$$

$$x_2 \leq 1.87.$$

Figure 1-1:

integers, truncating would get us into trouble, the optimum integer solution is not that corresponding to the point (1,1) but rather to the point (2,1).

Efficiency of the rounding off and truncating method

This rounding off is a procedure that can be used in some simple integer programming models, and its constraints should be in the model so that it is easy to decide whether feasibility is preserved by rounding off a non-integer value of a variable to the next larger integer or the nearest integer lower than

Problem 1.2

$$\text{maximize } f = x_1 + 3x_2$$

$$\text{subject to } x_2 \leq 1.87$$

$$60x_1 + 28x_2 \leq 165$$

it.

This method can also give us trouble if a feasible neighbouring integer point of a solution is far away from the optimum integer solution. This method is not suitable for any ILP problems whose values of an optimum solution of the problem are likely to be small.

We begin by solving the linear programming problem that results if constraint 3 is eliminated. Using the simplex algorithm of linear programming, we obtain $x_1 = 1.87, x_2 = 1.87$, and the upper bound (UB) of this problem is $x_1 + 3x_2 = (1.87) + 3(1.87) = 7.5$. We associate this solution with node 1 in Figure 1.4.

We now introduce constraint 3, and partition the set of further constrained problems into those in which $x_1 \leq 1$ on the one hand and those in which $x_2 \geq 2$ on the other hand; that is, we investigate values of x_1 on either side of $x_1 = 1.87$. We represent these two classes of problems as nodes 2 and 3 respectively, in Figure 1.4.

We now solve these two problems using the simplex algorithm, and produce the solutions associated with nodes 2 and 3 in Figure 1.4. The resulting objective function values constitute upper bounds on all further constrained problems in their respective classes, as shown in Figure 1.4; and the LP's below any node have a feasible set contained in the feasible set of the root. That is to say, examining nodes 2, 3, we conclude that the objective function of nodes 2 and 3 cannot be greater than 7.5. For node 2, we also conclude that the objective of any linear program in which $x_1 \leq 1$ and in which x_2 is not only constrained to be greater than or equal to zero, but is further constrained to be an integer, cannot be greater than 6.61. Similarly, referring to node 3, we see that the objective function of any linear programming problem in which $x_1 \geq 2$ and in which x_2 is only constrained to be greater than or equal to zero, but is further constrained to be an integer, cannot be greater than 7.4.

We repeat the process for nodes 4, 5, ..., 8. From Figure 1.4, the problem represented by node 5 is not feasible, then we terminate from that branch. Finally, we get an optimum solution as shown in node 8, such that $x_1 = 3, x_2 = 1$ and the upper bound is 6.

Figure 1-4:

1.1.3 Cutting planes method

Let $\bar{x}$ be an optimum solution of the associated LP and suppose $\bar{x}$ is non-integer. A cutting plane

is defined to be a hyperplane that strictly separates $\bar{x}$ from the set of integer feasible solutions of this problem, that is, it is a hyperplane with $\bar{x}$ on one side of it ($\bar{x}$ cannot lie on the hyperplane) and all the integer feasible solutions of the problem either on the hyperplane or on the other side of it. Figure 1.5 contains an illustration of a cutting plane. The associated LP has four constraints and $\bar{x}$ is an optimum solution for it. The convex hull of all the integer feasible solutions is the polytope outlined with dashed lines. The vector $\bar{x}$ is on one side of the cutting plane and all the integer feasible solutions are on its other side.

Figure 1.5:

The cutting plane method proceeds as follows :

- a) Let the current problem be the current associated LP. Solve the current problem. It can be solved by the simplex method, starting with an optimum solution of the previous problem.

If the current problem is infeasible, the original integer problem is also infeasible. Terminate.

If the optimum solution of the current problem satisfies the integer requirements, it is an optimum solution of the original integer program. Terminate.

If the optimum solution of the current problem violates some integer restrictions, go to b)

- b) Use the cut generation method, from a) consider a row appearing in the final simplex tableau. A rule for choosing a row from the final simplex tableau is to choose a row with the largest fractional part in the non-integer solution value of the variables. We can generate the new constraint (a cutting plane) from this row. Replace all the coefficients in the constraint equation under consideration by the smallest non-negative numbers that are congruent to these coefficients, and then state that the resulting expression must be greater than (less than) or equal to the fractional part of the constant on the right hand side of the equal sign. Two numbers are congruent if their difference is an integer, zero being considered an integer. Thus, $27/14$ is congruent to $13/14$, $-3/5$ is congruent to $2/5$. Add this new constraint to the current problem. Go to a).

The method is repeated until it terminates. We will only consider cutting plane methods satisfying the following property: the method must terminate either with the conclusion that the problem is infeasible or with an optimum integer solution in a finite number of steps.

An example of ILP problem using a cutting plane method

Problem 1.4

$$\begin{array}{ll}
 \text{maximize} & 3x_1 + 3x_2 \\
 \text{subject to} & 2x_1 + 4x_2 \leq 10 \\
 & 5x_1 + 4x_2 \leq 12
 \end{array}$$

$$x_1, x_2 \geq 0, \text{integers.}$$

conical form :

$$\text{maximize} \quad 3x_1 + 3x_2$$

$$\text{subject to} \quad 2x_1 + 4x_2 + s_1 = 10$$

$$5x_1 + 4x_2 + s_2 = 12$$

$$x_1, x_2, s_1, s_2 \geq 0, \text{integers.}$$

Invoking the simplex procedure, we produce the final tableau in the table 1.1.

x_1	x_2	s_1	s_2	
0	1	5/12	-1/6	13/6
1	0	-1/3	1/3	2/3

Table 1-1: Solution $x_1 = \frac{2}{3}, x_2 = 2\frac{1}{6}$

Choose the second constraint to produce a new constraint (a cutting plane), since $x_1 = 2/3$ is the variable with the largest fractional part in the non-integer solution value. From the second constraint equation we produce the cut

$$0x_1 + 0x_2 + \frac{2}{3}s_1 + \frac{1}{3}s_2 \leq \frac{2}{3}$$

$$2s_1 + s_2 \leq 2.$$

Solving for s_1 and s_2 in the original constraint equations and substituting, we transform this to

$$3x_1 + 4x_2 \leq 10 \quad (\text{cut 1 in Figure 1.6}).$$

Add the new constraint (new cut) to the current problem, and solve via the simplex procedure. Since the first constraint is no longer binding (from Figure 1.6), we drop it. We produce the final tableau in Table 1.2.

x_1	x_2	s_3	s_4	
1	0	$1/2$	$-1/2$	1
0	1	$-3/8$	$5/8$	$7/4$

Table 1-2:

From the second constraint equation we produce the cut

$$\frac{5}{8}s_3 + \frac{5}{8}s_4 \geq \frac{3}{4} \text{ or } 5x_1 + 5x_2 \leq 13 \text{ (cut 2 in Figure 1.6)}$$

Adding the new cut to the current problem and solving via the simplex procedure, we produce the final tableau in Table 1.3.

x_1	x_2	s_3	s_4	s_7	
1	0	1	0	-4/5	8/5
0	0	1	1	-8/5	6/5
0	1	-1	0	1	1

Table 1-3:

From the first constraint equation, we produce the cut

$$s_7 \geq 3 \text{ or } x_1 + x_2 \leq 2 \text{ (cut 3 in Figure 1.6).}$$

Repeating the process, finally we get an optimal solution of $x_1 = 2, x_2 = 0$.

Efficiency of cutting plane method

Even though it can be proven that the cutting plane algorithm solves integer programs in a finite number of steps, this finite number can be very large. For simple problems, this algorithm requires a very large number of steps before termination, and in most other practical problems, this method is not suitable.

1.2 Some definitions

Lattice (Λ)

Let $a_1, \dots, a_n$ be linearly independent real vectors in n -dimension real Euclidean space, so that the only set of numbers $t_1, \dots, t_n$ for which $t_1 a_1 + \dots + t_n a_n = 0$ is $t_1 = \dots = t_n = 0$. The set of all points

$$X = u_1 a_1 + \dots + u_n a_n, \quad (1.3)$$

with integral $u_1, \dots, u_n$ is called the lattice with basis $a_1, \dots, a_n$. Since $a_1, \dots, a_n$ are linearly independent, the expression of any vector X in (1.3) with $u_1, \dots, u_n$ is unique.

The basis is not uniquely determined by the lattice. Let a'_i be the points

$$a'_i = \sum_j v_{ij} a_j, \quad (1.4)$$

where v_{ij} are any integers with

$$\det(v_{ij}) = \pm 1, \quad (1.5)$$

then

$$a_i = \sum_j w_{ij} a'_j, \quad (1.6)$$

with integral w_{ij} . It follows easily that the set of points (1.3) is precisely the set of points

$$u_1 a'_1 + \dots + u_n a'_n,$$

where $u'_1, \dots, u'_n$ are integers, that is $a_1, \dots, a_n$ and $a'_1, \dots, a'_n$ are bases of the same lattice. On

substituting (1.4) in (1.6) and making use of the linear independence of the a_i , we have :

$$\sum w_{ij}v_{jl} = \begin{cases} 1, & \text{if } i=l; \\ 0, & \text{otherwise.} \end{cases}$$

Hence, $\det(w_{ij})\det(v_{jl}) = 1$, note that $\det(w_{ij}) = \text{determinant of } [w_{ij}]_{1 \leq i,j \leq n}$ so each of $\det(w_{ij})$ and $\det(v_{jl})$ must be ± 1 , that is (1.5) holds as required.

Determinant of a lattice, $d(\Lambda)$

Let $a_1, \dots, a_n$ and $b_1, \dots, b_n$ be bases of the same lattice, so that they are related by (1.4), (1.5)

Then we have

$$\det(b_1, \dots, b_n) = \det(v_{ij})\det(a_1, \dots, a_n) = \pm \det(a_1, \dots, a_n).$$

Hence, if we define $d(\Lambda)$, the determinant of the lattice (Λ) , as $|\det(a_1, \dots, a_n)|$ then $d(\Lambda)$ is independent of the particular choice of basis, and $d(\Lambda) > 0$, since $a_1, \dots, a_n$ are linearly independent.

Lattices under transformation

Let the linear transformation $X = \alpha x$ be given by

$$X_i = \sum_{j=1}^n \alpha_{ij} x_j \quad (1 \leq i \leq n) \quad (1.7)$$

where $X = (X_1, \dots, X_n)$, $x = (x_1, \dots, x_n)$, are corresponding points in the transformation and α_{ij} are real numbers such that $\det(\alpha) = \det(\alpha_{ij}) \neq 0$. This is a non-singular linear transformation.

Let Λ be a lattice and denote by $\alpha\Lambda$ the set of points αx , $x \in \Lambda$. If $b_1, \dots, b_n$ is a basis for Λ , then the general point $b = u_1 b_1 + \dots + u_n b_n$, where $u_1, \dots, u_n$ are integers, of Λ has the transform

$$\alpha b = \alpha(u_1 b_1 + \dots + u_n b_n) = u_1 \alpha b_1 + \dots + u_n \alpha b_n.$$

Hence, $\alpha\Lambda$ is a lattice with basis $\alpha b_1, \dots, \alpha b_n$, and

$$d(\alpha\Lambda) = |\det(\alpha b_1, \dots, \alpha b_n)| = |\det \alpha| |\det(b_1, \dots, b_n)| = |\det \alpha| d(\Lambda).$$

Inequality of Hadamard

Lemma 1 Let $a_1, \dots, a_n$ be n -dimensional vectors. Then

$$|\det(a_1, \dots, a_n)| \leq |a_1| \dots |a_n|.$$

Proof

If the determinant is 0, there is nothing to prove. Suppose that $a_1, \dots, a_n$ are linearly independent

We construct a sequence of vectors c_j ($1 \leq j \leq n$) such that

$$c_i c_j = 0 \quad (i \neq j). \quad (1.8)$$

This is just a scalar product of two vectors, and

$$a_i = t_{i1}c_1 + \dots + t_{i,n-1}c_{n-1} + c_i, \quad (1.9)$$

for some real numbers t_{ij} , hence

$$c_1 = a_1.$$

and define c_i recursively,

$$c_i = a_i - \sum_{j < i} (a_i c_j) |c_j|^{-2} c_j.$$

By (1.8), (1.9) we have

$$\begin{aligned} |a_i|^2 &= a_i a_i \\ &= t_{i1}^2 |c_1|^2 + \dots + t_{i,n-1}^2 |c_{n-1}|^2 + |c_i|^2 \\ &\geq |c_i|^2. \end{aligned} \quad (1.10)$$

Since,

$$c_1 = a_1 \text{ and}$$

$$c_2 = a_2 - (a_2 c_1) |c_1|^{-2} c_1 = a_2 - k_{11} a_1,$$

where $k_{11} = (a_2 c_1) |c_1|^{-2}$,

$$c_3 = a_3 - (k_{21} a_1 - k_{22} c_2) = a_3 - (k_{21} + k_{22} k_{11}) a_1 - k_{22} a_2.$$

where $k_{21} = (a_3 c_1) |c_1|^{-2}$, and $k_{22} = (a_3 c_2) |c_2|^{-2}$. From above, we can see that c_i is a linear combination of a_i . Hence

$$\det(a_1, \dots, a_n) = \det(c_1, \dots, c_n). \quad (1.11)$$

Regarding the $c_1, \dots, c_n$ in $\det(c_1, \dots, c_n)$ first as rows and then as columns and multiplying the two determinants together, we have

$$\{\det(c_1, \dots, c_n)\}^2 = \det\{c_i c_j\} = \prod_{i=1}^n |c_i|^2. \quad (1.12)$$

From (1.10), (1.11), (1.12) we have

$$|\det(a_1, \dots, a_n)| \leq |a_1| \dots |a_n|. \quad (1.13)$$

This inequality represents the fact that the volume of a parallelepiped is at most the product of the length of the sides.

Convex hull

The convex hull is the natural boundary of a point set. The convex hull of a set of points in the plane is defined to be the smallest convex polygon containing them all. A convex polygon is a polygon with the property that the line segment connecting any two points inside the polygon must itself lie inside the polygon. A fundamental property of the convex hull is that any line outside the hull, when moved in any direction towards the hull, either hits the hull at one of its vertex points, or hits the hull at one of its edges.

Hyperplane

A hyperplane in $\mathbb{R}^n$ is the set of all points $x = (x_1, \dots, x_n) \in \mathbb{R}^n$, satisfying a single linear equation $a_1 x_1 + \dots + a_n x_n = b$, where $a_1, \dots, a_n$ are given numbers and at least one of the a_i is non-zero, that is $(a_1, \dots, a_n) \neq 0$.

N-Simplex

In N dimensions, a simplex is the set of all points $x = x_0 v^0 + \dots + x_n v^n$ such that $x_0, \dots, x_n \geq 0$ and $x_0 + \dots + x_n = 1$; the $n+1$ fixed vectors $v^0, v^1, \dots, v^n$ are the vertices of the simplex. The numbers $x_0, \dots, x_n$ are by definition the barycentric coordinates of the vector x . The simplex is nondegenerate if the n vertices $v^1 - v^0, v^2 - v^0, \dots, v^n - v^0$ are linearly independent and the barycenter b of an n -dimensional simplex is the center of gravity and is such that $b = \frac{1}{n+1}(v^0 + v^1 + \dots + v^n)$.

The class P

The class P is the set of all problems which can be solved by deterministic algorithms in polynomial time. By a deterministic algorithm, we mean that the algorithm has only one thing it can do next.

The class NP

The class NP is the set of all problems that can be solved by non-deterministic algorithms in polynomial time. By a non-deterministic algorithm, we mean that the algorithm when faced with a choice of several options, has the power to guess the right one.

NP -Complete problem is the term used to describe problems that are the hardest ones in NP in the sense that no NP -Complete problem can be solved by any known polynomial time algorithm, and if there is a polynomial time algorithm for any NP -Complete problem, then there are polynomial time algorithms for all NP -Complete problems.

1.3 Some concepts

The following concepts are used throughout this thesis. We shall prove them in this section, and later in the thesis, we will refer to them. The first concept concerns the nature of the convex set of the system $Ax \leq b$. We will show that we may assume that this convex set is bounded and has a positive volume; by positive volume, we mean that a convex hull with vertices $v_0, v_1, \dots, v_n$ has n dimensions or full dimension. The next concept concerns a regular simplex (S), an equilateral simplex. There exist two closed balls with center p (barycenter of S) such that $B(p, r) \subset S \subset B(p, R)$, where r, R are the radii, and R/r is bounded by some constant value. As the last concept, we shall show that, we can find an upper bound of the distance between any vector $x \in \mathbb{R}^n$ and a vector $y \in \text{lattice}$.

1.3.1 An assumption for Convex Hulls

Let K be the closed convex set, $K = \{x \in \mathbb{R}^n : Ax \leq b\}$, where A is an $(m \times n)$ -matrix, $b \in \mathbb{Z}^m$. We would like to ensure

- a) K is bounded
- b) K has a positive volume (full dimension).

To show a). We shall invoke a result of Von zur Gathen and Sieveking, [4], stating that $K \cap \mathbb{Z}^n \neq \emptyset$ if and only if $K \cap \mathbb{Z}^n$ contains a vector whose coefficients are bounded by $(n+1)n^{\frac{n(n+1)}{2}} a^n$ in absolute value, where a denotes the maximum of the absolute values of the coefficients of A and b . Adding these inequalities to the system makes K bounded.

If K is bounded, K is also a convex hull of the set of its vertices. A vertex is obtained by replacing n linearly independent inequalities in the system $Ax \leq b$ by equalities, and testing whether the unique

vector satisfying these equalities also satisfies the other inequalities; if it does, it is a vertex of K .

To justify b), let $v_0, v_1, \dots, v_M$ be the vertices for the system $Ax \leq b$. Hence, K contains $v_0, v_1, \dots, v_M$ and $M > 0$ if K is not empty. Let d be the dimension of the linear subspace V of $\mathbb{R}^n$ generated by the vectors $v_j - v_0$, $1 \leq j \leq M$. If $d = n$, where n is the number of variables in the system $Ax \leq b$, then there are $n + 1$ vertices among $v_0, v_1, \dots, v_M$ whose convex hull is an n -simplex of positive volume. For example

The system of $x_1 + 2x_2 + x_3 \leq 4$

$$x_1 + 2x_2 + 2x_3 \leq 6$$

$$x_1, x_2, x_3 \geq 0.$$

A convex hull: $v_0 = 0, 0, 0$

$$v_1 = 0, 0, 3$$

$$v_2 = 4, 0, 0$$

$$v_3 = 0, 2, 0.$$

Since $n=3$ and $d=3$, a convex hull is a 3-simplex of positive volume.

We will now prove that even if $d < n$ we may assume that the convex hull, K , has a positive volume.

We need to show that the problem can be transformed into an integer programming problem with only d variables. Choose, for each j , an integer multiple w_j of $v_j - v_0$ such that $w_j \in Z^n$, and let W be the $(n \times M)$ -matrix whose columns are w_j . There exists an integral $(n \times n)$ -matrix U with $\det(U) = \pm 1$, such that

$$UW = [k_{ij}]_{1 \leq i \leq n, 1 \leq j \leq M}. \quad (1.13)$$

$$\begin{cases} k_{ij} = 0 & \text{if } i > j \text{ or } i > d \\ k_{ii} \neq 0 & \text{if } 1 \leq i \leq d. \end{cases}$$

Such a matrix U can be found in polynomial time by repeated application of the Euclidean algorithm.

An example of finding a matrix U

$$\text{Let } W = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 7 & 3 \\ 1 & 2 & 2 \end{bmatrix}$$

Step 1 Find a matrix U_{21} with integral coefficient and $\det U_{21} = \pm 1$ such that

$$U_{21}W = [k_{ij}]_{1 \leq i, j \leq 3} \quad \text{with}$$

$$k_{11} = \text{g.c.d.}(w_{11}, w_{21})$$

$$k_{21} = 0.$$

Then take

$$U_{21} = \begin{bmatrix} s_1 & t_1 & 0 \\ -\frac{w_{21}}{k_{11}} & \frac{w_{11}}{k_{11}} & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

such that

$$\begin{bmatrix} s_1 & t_1 & 0 \\ -\frac{w_{21}}{k_{11}} & \frac{w_{11}}{k_{11}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ 0 & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix},$$

where s_1, t_1 are chosen such that these three equations are satisfied.

$$\left(\frac{w_{11}}{k_{11}}\right)s_1 - \left(\frac{w_{21}}{k_{11}}\right)t_1 = 1 \quad \text{:(This is possible by the Euclidean algorithm),}$$

$$\left. \begin{aligned} w_{12}s_1 + w_{22}t_1 &= k_{12} \\ w_{13}s_1 + w_{23}t_1 &= k_{13} \end{aligned} \right\} \quad k_{12}, k_{13} \text{ are integers.}$$

In our case,

$$U_{21} = \begin{bmatrix} s_1 & t_1 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ and}$$

$$\begin{bmatrix} s_1 & t_1 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 2 & 1 \\ 6 & 7 & 3 \\ 1 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & k_{12} & k_{13} \\ 0 & 3 & 1 \\ 1 & 2 & 2 \end{bmatrix}.$$

Let $s_1 = -1$, $t_1 = 1$, hence

$$U_{31} = \begin{bmatrix} -1 & 1 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ and } K_1 = U_{31}W = \begin{bmatrix} 3 & 5 & 2 \\ 0 & 3 & 1 \\ 1 & 2 & 2 \end{bmatrix}.$$

Step 2 Repeat Step 1, find a matrix U_{31} with the integral coefficients and $\det U_{31} = \pm 1$ such that

$$U_{31}K_1 = [k'_{ij}]_{1 \leq i, j \leq 3} \text{ with}$$

$$k'_{11} = g.c.d.(k_{11}, k_{31}),$$

$$k'_{31} = 0.$$

Then take

$$U_{31} = \begin{bmatrix} s_2 & 0 & t_2 \\ 0 & 1 & 0 \\ \frac{-k'_{21}}{k'_{11}} & 0 & \frac{k'_{31}}{k'_{11}} \end{bmatrix}, \text{ with } \left(\frac{k_{11}}{k'_{11}}\right)s_2 - \left(\frac{-k_{31}}{k'_{11}}\right)t_2 = 1.$$

Select s_2, t_2 (the same way as s_1, t_1 were selected), let $s_2 = -1, t_2 = 4$, then

$$U_{31} = \begin{bmatrix} -1 & 0 & 4 \\ 0 & 1 & 0 \\ -1 & 0 & 3 \end{bmatrix}, \text{ and } K_2 = U_{31}K_1 = \begin{bmatrix} 1 & 3 & 6 \\ 0 & 3 & 1 \\ 0 & 1 & 4 \end{bmatrix}$$

Step 3 Find a matrix U_{32} with integral coefficients and $\det U_{32} = \pm 1$ such that

$$U_{32}K_2 = [k''_{ij}]_{1 \leq i, j \leq 3} \text{ with}$$

$$k''_{32} = 0$$

$$k''_{22} = g.c.d.(k'_{22}, k'_{32})$$

Using the same process as in Step 1 and Step 2, we can obtain

$$U_{32} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -2 \\ 0 & -1 & 3 \end{bmatrix}, \text{ and } K_3 = U_{32}K_2 = \begin{bmatrix} 1 & 3 & 6 \\ 0 & 1 & 7 \\ 0 & 0 & 11 \end{bmatrix}$$

We stop at this step, since $K_3 = [k''_{ij}]_{1 \leq i, j \leq 3}$ with

$$\begin{cases} k''_{ij} = 0 & \text{if } i > j \\ k''_{ii} \neq 0 & \text{if } 1 \leq i \leq 3 \end{cases}$$

Step 4 We can obtain U as $U_3 U_{31} U_{21}$; and $\det U = \det U_{32} \det U_{31} \det U_{21} = \pm 1$. In this example

$$U = \begin{bmatrix} 1 & -1 & 4 \\ -4 & 3 & -6 \\ 5 & -4 & 9 \end{bmatrix}$$

In general, the formula for such a matrix U_{ij} such that $U_{ij}W = K_i$, where K_i is a matrix with element $k_{ij} = 0$ for $i > j$, is

$$u_{jj} = 0, \quad u_{ji} = t$$

$$u_{ij} = \frac{-w_{ij}}{g}, \quad u_{ii} = \frac{w_{jj}}{g}; \quad \text{where } g = g.c.d.(w_{jj}, w_{ij})$$

$$u_{ef} = 0, \quad u_{ee} = 1; \quad \text{for all } (e, f) \neq (i, j),$$

and

$$\left(\frac{w_{jj}}{g}\right)t + \left(\frac{w_{ij}}{g}\right)t = 1$$

Hence, U is a unimodular matrix. Denote by $u_1, \dots, u_n$ the columns of the integral matrix U^{-1} , then U^{-1} is also a unimodular matrix with $\det U^{-1} = \pm 1$. These $u_1, u_2, \dots, u_n$ form a basis of $\mathbb{R}^n$ and $Z = \sum_{j=1}^n \mathbb{Z} u_j$. The linear subspace V of $\mathbb{R}^n$ generated by the vectors $v_j - v_0$, $1 \leq j \leq M$, or in other word $K \subset V + v_0$; and let W be the $(n \times M)$ -matrix whose j -column is an integer multiple of $v_j - v_0$, the column of $W = U^{-1}[k_j]$, so (1.13) implies that

$$V = \sum_{j=1}^d \mathbb{R} u_j. \quad (1.14)$$

Let $v_0 = \sum_{j=1}^n r_j u_j$, ($r_j \in \mathbb{R}$, $1 \leq j \leq n$). Now suppose that $x \in K \cap Z^n$, then $x = \sum_{j=1}^n y_j u_j$ with $y_j \in \mathbb{Z}$, $1 \leq j \leq n$,

$$x - v_0 = (y_1 - r_1)u_1 + \dots + (y_d - r_d)u_d + (y_{d+1} - r_{d+1})u_{d+1} + \dots + (y_n - r_n)u_n. \quad (1.15)$$

Suppose $x - v_0 \in V$, then by (1.14), (1.15) with $y_j = r_j$, $d < j \leq n$. If at least one of the $r_{d+1}, \dots, r_n$ is not an integer, then this implies that $x - v_0 \notin V$, since $y_j \in \mathbb{Z}$ ($1 \leq j \leq n$), then $K \cap Z^n = \emptyset$ (no

such x). On the other hand, $r_{d+1}, \dots, r_n$ are all integer, so that the problem can be transformed to an integer programming with only d variables $y_1, \dots, y_d$, as required. Hence, we can therefore assume that we have a case of full dimension.

1.3.2 To find a radius of the Euclidean ball for a regular simplex

Lenstra derives the way of finding a radius for a Euclidean ball of a regular simplex such that :

Lemma 2 By $C(u_0, \dots, u_n)$, we denote the convex hull of $u_0, \dots, u_n$, and by vol we denote the volume of the convex hull. Let $z_0, z_1, \dots, z_n \in \mathbb{R}^n$ be the vertices of a regular n -simplex S , with barycenter $p = \frac{1}{n+1} \sum_{i=0}^n z_i$. Let T be the set of all $x \in \mathbb{R}^n$ for which

$$\text{vol}(C(x, z_0, z_1, \dots, z_{i-1}, z_{i+1}, \dots, z_n)) \leq \text{vol} S,$$

for $i = 0, 1, \dots, n$. Then there are positive real numbers, r and R such that

$$B(p, r) \subset S \subset T \subset B(p, R), \text{ and}$$

$$\left(\frac{R}{r}\right)^3 = \begin{cases} n^3 + 2n^2 & \text{if } n \text{ is even} \\ n^3 + n^2 - n & \text{if } n \text{ is odd.} \end{cases}$$

where $B(p, r)$ is a Euclidean closed ball in $\mathbb{R}^n$ with center p and radius r ; that is

$$B(p, r) = \{x \in \mathbb{R}^n : |x - p| \leq r\},$$

where $p \in \mathbb{R}^n$, $r \in \mathbb{R} \geq 0$, $||$ denotes the Euclidean length in $\mathbb{R}^n$.

Proof We can identify $\mathbb{R}^n$ with the hyperplane $\{(r_j)_{j=0}^n \in \mathbb{R}^{n+1} : \sum_{j=0}^n r_j = 1\}$ in $\mathbb{R}^{n+1}$ such that $z_0, z_1, \dots, z_n$ is the standard basis of $\mathbb{R}^{n+1}$. Then we have $p = \left(\frac{1}{n+1}, \dots, \frac{1}{n+1}\right)$, and

$$T = \left\{ (r_j)_{j=0}^n \in \mathbb{R}^{n+1} : |r_j| \leq 1 \text{ for } 0 \leq j \leq n \text{ and } \sum_{j=0}^n r_j = 1 \right\}$$

It is easily seen that T is the convex hull of the set of points obtained by permuting the coordinates of the point $\sum_{j=0}^k z_j - \sum_{j=k+1}^{2k} z_j$, where $n = 2k$ if n is even, or $n = 2k + 1$ if n is odd.

It follows that $T \subset B(p, R)$, where R is the distance of p to the above point.

$$R^2 = \begin{cases} \frac{n^2 + 2n}{n+1} & \text{if } n \text{ is even} \\ \frac{n^2 + n - 1}{n+1} & \text{if } n \text{ is odd.} \end{cases}$$

Further, $B(p, r) \subset S$, where r is the distance of p to $(0, \frac{1}{n}, \dots, \frac{1}{n})$ the center of one of the faces of S :

$$r^2 = \frac{1}{n(n+1)}.$$

Since, trivially $S \subset T$ and

$$B(p, r) \subset S \subset T \subset B(p, R);$$

$$\left(\frac{R}{r}\right)^2 = \begin{cases} n^2 + 2n & \text{if } n \text{ is even} \\ n^2 + n^2 - n & \text{if } n \text{ is odd.} \end{cases}$$

1.3.3 An upper bound of a nearest lattice point from any vector

Let $\mathcal{L} = \sum_{i=1}^n \mathbb{Z} b_i = \{\sum_{i=1}^n m_i b_i : m_i \in \mathbb{Z}, (1 \leq i \leq n)\}$. Then $\mathcal{L}$ is a lattice in $\mathbb{R}^n$ and $b_1, b_2, \dots, b_n$ is a basis for $\mathbb{R}^n$.

Lemma 3 Let $b_1, b_2, \dots, b_n$ be any basis for $\mathcal{L}$. For any vector $x \in \mathbb{R}^n$, there exists a vector $y \in \mathcal{L}$ such that

$$|x - y|^2 \leq \frac{1}{4}(|b_1|^2 + \dots + |b_n|^2), \quad (1.18)$$

and furthermore if we number b_i such that

$$|b_n| = \max\{|b_i| : 1 \leq i \leq n\},$$

then (1.18) implies that

$$\forall x \in \mathbb{R}^n : \exists y \in \mathcal{L} : |x - y| \leq \frac{1}{2} \sqrt{n} |b_n|. \quad (1.19)$$

Proof The proof is by induction on n , the case $n = 1$, (or $n = 0$) being obvious since vector x lies between y_1, y_2 where $x \in \mathbb{R}^1$ and $y_1, y_2 \in \mathcal{L}$. For example, let $b_1 = 1$, and $x = a(1)$; $a \in \mathbb{R}$,

$$y_1 = b(1) \quad , \quad b \in \mathbb{Z} \text{ and } b < a,$$

$$y_2 = c(1) \quad ; \quad c \in \mathbb{Z} \text{ and } a < c.$$

Hence, the distance from x to either y_1 or y_2 is at most $\frac{1}{2}|b_1|$, $|x - y_1|^2$ or $|x - y_2|^2 \leq \frac{1}{4}|b_1|^2$.

Assume that it is true for a lattice of dimension $n - 1$. Then

$$\forall x \in \mathbb{R}^{n-1} : \exists y \in \mathcal{L}' : |x - y|^2 \leq \frac{1}{4}(|b_1|^2 + \dots + |b_{n-1}|^2),$$

where $\mathcal{L}' = \sum_{i=1}^{n-1} \mathbb{Z} b_i$, this is a lattice in the $(n - 1)$ dimensional hyperplane $\mathcal{N} = \sum_{i=1}^{n-1} \mathbb{R} b_i$. Denote by h the distance of b_n to $\mathcal{N}$. Clearly we have

$$h \leq |b_n|. \quad (1.20)$$

Now, to prove (1.18), let $x \in \mathbb{R}^n$. We wish to change x by an element of $\mathcal{L}$ such that its length becomes small. First subtract an integral multiple of b_n from x such that its distance to $\mathcal{N}$ becomes $\leq \frac{1}{2}h$.

Write $x = x_1 + x_2$, with $x_1 \in \mathcal{N}$ and x_2 perpendicular to $\mathcal{N}$, then $|x_2| \leq \frac{1}{2}h \leq \frac{1}{2}|b_n|$. By the induction

hypothesis, we can change x_1 by an element of $\mathcal{L}'$ and achieve that $|x_1|^2 \leq \frac{1}{4}(|b_1|^2 + \dots + |b_{n-1}|^2)$,

since x_1, x_2 are perpendicular to each other, $|x_1|^2 + |x_2|^2 \leq \frac{1}{4}(|b_1|^2 + \dots + |b_{n-1}|^2) + \frac{1}{4}(|b_n|^2)$ or $|x|^2 \leq$

$\frac{1}{4}(|b_1|^2 + \dots + |b_n|^2)$. This proves (1.18). If we number the b_i such that $|b_n| = \max\{|b_i| : 1 \leq i \leq n\}$,

then $(|b_1|^2 + \dots + |b_n|^2) \leq n|b_n|^2$, from (1.18) we implies that

$$|x - y|^2 \leq \frac{1}{4}n|b_n|^2 \quad \text{or} \quad |x - y| \leq \frac{1}{2}\sqrt{n}|b_n|,$$

This proves (1.19).

Chapter 2

An algorithm for solving the ILP problem within a polynomial time

This chapter describes the algorithm to transform an ILP problem into an equivalent one with an additional property as describe in the *Introduction* : either the existence of a vector $x \in \mathbb{Z}^n$ satisfying $Ax \leq b$ is obvious ; or it is known that the last coordinate of any such x belongs to an interval whose length is bounded by a constant only depending on n . This algorithm is only concerned with the constraints of an ILP problem, and the algorithm will terminate if there exists an integer point in the feasible region. We shall describe the algorithm in *Section 2.1*. In order to understand the algorithm clearly, we shall present an example of this algorithm in *Section 2.2*.

2.1 An algorithm to transform an ILP problems

Let an ILP problem have constraints $Ax \leq b$, where A is a matrix with integer coefficients of order $m \times n$, and b is a column vector of order m . Let K be the convex hull of the system $Ax \leq b$. Throughout this algorithm, we solve any LP problem by *the Simplex method* . In the worst case, the complexity of this method is not a polynomial running time. In practice, it is a very practical method and its worst case is seldom occurs. The algorithm is as follows :

Step 1 Find a Simplex in K

Maximize an arbitrary function on K ; by using *the simplex method* , maximize an arbitrary function

subject to $Ax \leq b$, $x \geq 0$. Note that x is not necessarily an integer vector. One finds a vertex v_0 of K , unless K is empty. Now to construct, by induction, vertices $v_1, \dots, v_n$ of K such that the vectors $v_j - v_0$, ($1 \leq j \leq n$), are linearly independent. Suppose that vertices $v_0, \dots, v_d$ for $d < n$ have been constructed for which $v_1 - v_0, \dots, v_d - v_0$ are linearly independent. By induction, we mean to maximize or minimize several linear functions on K that are constant on $\{v_0, \dots, v_d\}$. For example, we can find the equation of the plane passing through $\{v_0, \dots, v_d\}$, then maximize this function on K . we might find v_{d+1} (if it exists), check whether $v_j - v_0$ for $1 \leq j \leq d+1$ are linearly independent. We consider 2 cases

Case 1 We can find a vertex v_{d+1} such that $v_1 - v_0, \dots, v_d - v_0$ are linearly independent, and if $(d+1) = n$, then we stop. This is the case of full dimension.

Case 2 We discover that $K \subset v_0 + V$, where $V = \sum_{j=1}^d \mathbb{R}(v_j - v_0)$. In this case, as described in Subsection 1.3.1 of Chapter 1, we can change the original problem with n variables into one with only d variables. This means that there doesn't exist v_{d+1} such that $v_1 - v_0, \dots, v_d - v_0, v_{d+1} - v_0$ are linearly independent. Now, W which is a matrix of an integral multiple of $v_j - v_0$ where $1 \leq j \leq M$, as in Subsection 1.3.1, is an $(n \times d)$ -matrix rather than $(n \times M)$. This is a problem with d dimension.

Note: If the LP problem is infeasible in this step, terminate from the algorithm, and conclude that the ILP problem also infeasible.

Step 2 Find a large Simplex inside K

Select a positive real number ϵ . For some i , find a vertex v of K such that

$$\text{vol}(C(v, v_0, \dots, v_{i-1}, v_{i+1}, \dots, v_n)) \geq (1 + \epsilon) \cdot \text{vol}(C(v_0, v_1, \dots, v_n)),$$

where $C(v_0, \dots, v_n)$ denotes the convex hull, and vol the volume. If such a v exists, it can be found

by solving two linear programming problems for fixed i . The two linear programming problems can be constructed by maximizing and minimizing the linear function $\text{vol}(C(v, v_0, \dots, v_{i-1}, v_{i+1}, v_n))$, subject to the original constraints. Now, replace v_i by v (if such exists) and repeat the process for each i until $i = n$, keeping v fixed. If there doesn't exist such v then the convex hull is unchanged. With every step $\text{vol}(C(v_0, \dots, v_n))$ increases by a factor of at least $(1 + \epsilon)$. On the other hand, it is bounded by $\text{vol}(K)$. After this step we have a simplex in K spanned by $v_0, \dots, v_n$ such that

$$\text{vol}(C(x, v_0, \dots, v_{i-1}, v_{i+1}, \dots, v_n)) < (1 + \epsilon) \text{vol}(C(v_0, \dots, v_n));$$

$$\text{for all } x \in K \text{ and for all } i \in \{0, 1, \dots, n\}. \quad (2.1)$$

Step 3 Mapping the Simplex to a regular Simplex

The convex hull $C(v_0, \dots, v_n)$ in Step 2 forms an n -dimensional simplex D of K . Choose a linear transformation (r) of $\mathbb{R}^n$ such that the vectors $z_j = r(v_j)$, for $0 \leq j \leq n$, span a regular n -dimensional simplex (S) . Hence, D is transformed by r into S ; this is an affine transformation. From Step 2, $\text{vol}(D)$ is bounded by $\text{vol}(K)$, $\text{vol}(D) \leq \text{vol}(K)$; and

$$\text{vol}(S) = |\det r| \cdot \text{vol}(D) \leq |\det r| \cdot \text{vol}(K) = \text{vol}(rK),$$

then $S \subset rK$. If we apply r to (2.1):

$$\text{vol}(C(y, z_0, \dots, z_{i-1}, z_{i+1}, \dots, z_n)) < (1 + \epsilon) \text{vol}(rD) = (1 + \epsilon) \text{vol}(S);$$

$$\text{where } y = r(x), \text{ for all } x \in K \text{ and for all } i \in \{0, 1, \dots, n\}. \quad (2.2)$$

Let

$$T_i = \{z \in \mathbb{R}^n : \forall i \in \{0, 1, \dots, n\} :$$

$$\text{vol}(C(z, z_0, \dots, z_{i-1}, z_{i+1}, \dots, z_n)) < (1 + \epsilon) \text{vol}(S). \quad (2.3)$$

Since $K \in \mathfrak{R}^n$, and r is a linear transformation; then from (2.2), (2.3): $rK \subset T_\epsilon$. Recall the concept from Subsection (1.3.2) of Chapter 1: $S \subset T$ where

$$T = \{z \in \mathfrak{R}^n : \forall i \in \{0, 1, \dots, n\} :$$

$$\text{vol}(C(z, z_0, \dots, z_{i-1}, z_{i+1}, \dots, z_n)) \leq \text{vol}(S)\}, \quad (2.4)$$

then there exists two Euclidean balls of $B(p, r)$, $B(p, R)$ in $\mathfrak{R}^n$ such that

$$B(p, r) \subset S \subset T \subset B(p, R), \quad (2.5)$$

where $p \in \mathfrak{R}^n$ is a center with radius r and R , and p is also a barycenter of S . It is obvious from (2.3), (2.4) that $T \subset T_\epsilon$ for a positive value of ϵ . Let ϵ be small, then T_ϵ is slightly bigger than T . There also exists a Euclidean ball $B(p, R')$ with radius R' and center p such that $T_\epsilon \subset B(p, R')$ and R' is slightly different from R for some small ϵ . Finally, we conclude that

$$B(p, r) \subset S \subset rK \subset T_\epsilon \subset B(p, R).$$

Remark The value of ϵ should be small enough to get the vertices of K that spanned as large a simplex of K as possible.

If we use a similarity transformation (as describe in Subsection 1.3.2), we can identify $\mathfrak{R}^n$ with the hyperplane $\{(r_j)_{j=0}^n \in \mathfrak{R}^n : \sum_{j=0}^n r_j = 1\}$ in $\mathfrak{R}^{n+1}$ such that $z_0, \dots, z_n$ (which are vertices of S) are a standard basis of $\mathfrak{R}^{n+1}$. Then p is $(1/n+1, \dots, 1/n+1)$, and

$$\left(\frac{R}{r}\right)^2 = \begin{cases} n^2 + 2n, & \text{if } n \text{ is even;} \\ n^2 + n^2 - n, & \text{if } n \text{ is odd.} \end{cases}$$

Step 4 Examine the Image of Z^n

Fix r ; we want to find an integral vector x with $r(x) \in rK$. We are therefore interested in the lattice $\mathcal{L} = r(Z^n)$. Let $b_i = r(e_i)$ with e_i standard basis vectors. Then $\mathcal{L}$ is a lattice in $\mathbb{R}^n$ and $b_1, \dots, b_n$ is a basis for $\mathcal{L}$. Let $d(\mathcal{L})$ be the determinant of $\mathcal{L}$ and from the inequality of Hadamard (in Chapter 1)

$$d(\mathcal{L}) \leq \prod_{i=1}^n |b_i|.$$

We can change the basis of $\mathcal{L}$ by the reduction process (the details for this process are described in Chapter 3) into one such that the opposite inequality holds:

$$\prod_{i=1}^n |b_i| \leq c_2 d(\mathcal{L}), \quad (2.6)$$

where c_2 is a constant only depending on n , e.g. $c_2 = c^{n(n-1)/4}$ with c any constant greater than $4/3$. We shall call a basis $b_1, \dots, b_n$ satisfying (2.6) a "Reduced Basis".

Step 5 Concluding Step

We number the reduced basis $b_1, \dots, b_n$ such that $|b_n| = \max\{|b_i| : 1 \leq i \leq n\}$. Then from (1.19) of Lemma 3 in Subsection 1.3.3:

$$\forall x \in \mathbb{R}^n, \exists y \in \mathcal{L} : |x - y| \leq \frac{1}{2} \sqrt{n} |b_n|.$$

We will consider the value of $\frac{1}{2} \sqrt{n} |b_n|$, and divide into 2 cases.

Case 1 If $\frac{1}{2} \sqrt{n} |b_n| \leq r$, then

$$\forall x \in \mathbb{R}^n : \exists y \in \mathcal{L} : |x - y| \leq \frac{1}{2} \sqrt{n} |b_n| \leq r.$$

If we let x as p (the barycenter of S). This mean that $B(p, r)$ is big enough to contain a point of $\mathcal{L}$, and

$$y \in B(p, \frac{1}{2}\sqrt{n}|b_n|) \cap \mathcal{L} \subset B(p, r) \cap \mathcal{L} \subset rK \cap \mathcal{L},$$

so $rK \cap \mathcal{L} \neq \emptyset$. To find y , we can represent x as a linear combination of the reduced basis such that $x = x_1 b_1 + \dots + x_n b_n$, where $x_i \in \mathbb{R}$, and $y = y_1 b_1 + \dots + y_n b_n$, where $y_i \in \mathbb{Z}$. The problem is to find integral coordinates of y , such that $|x - y| \leq r$. Let z be the point at the boundary of $B(p, r)$, and $z = z_1 b_1 + \dots + z_n b_n$, where $z_i \in \mathbb{R}$. Hence, $|\sum_{i=1}^n (z_i - x_i) b_i|^2 = r^2$; then to find $y_i \in \mathbb{Z}$ such that $|\sum_{i=1}^n (y_i - x_i) b_i|^2 \leq r^2$ is enough to find $y_i \in \mathbb{Z}$ such that $y_i \leq z_i$. For example, let y_i be the nearest integer for x_i , each y_i is distance from x_i at most $|\frac{1}{2}|$, then

$$\begin{aligned} |\sum_{i=1}^n (y_i - x_i) b_i|^2 &\leq |\sum_{i=1}^n (\frac{1}{2}) b_i|^2 \\ &\leq \frac{1}{4} |\sum_{i=1}^n b_i|^2 \\ &\leq \frac{1}{4} n |b_n|^2 \end{aligned}$$

Hence, $|\sum_{i=1}^n (y_i - x_i) b_i| \leq \frac{1}{2} \sqrt{n} |b_n|$, which is less than or equal r . Hence, $y \in \mathcal{L}$ and

$$y \in B(p, \frac{1}{2}\sqrt{n}|b_n|) \cap \mathcal{L} \subset B(p, r) \cap \mathcal{L} \subset rK \cap \mathcal{L}.$$

Since, we have found a point y in $rK \cap \mathcal{L}$, we have found an integer point $r^{-1}(y)$ in K ; we are done.

Case 2 If $\frac{1}{2}\sqrt{n}|b_n| > r$, this implies that $B(p, r)$ is too small to be sure of containing a point of $\mathcal{L}$.

If we write $\mathcal{L}$ as

$$\mathcal{L} = \mathcal{L}' + \mathbb{Z}b_n \subset N + \mathbb{Z}b_n = \cup_{k \in \mathbb{Z}} (N + kb_n),$$

where $\mathcal{L}' = \sum_{i=1}^{n-1} \mathbb{Z}b_i$, this is a lattice in the hyperplane $N = \sum_{i=1}^{n-1} \mathbb{R}b_i$. Hence, $\mathcal{L}$ is contained in

the union of countably many parallel hyperplanes which have successive distances h from each other

where we define h as the distance of b_n to $\mathcal{H}$. We are only interested in those hyperplanes that have a non-empty intersection with rK , and since $B(p, r) \subset rK \subset B(p, R)$, this implies that these are some of the hyperplanes which intersect $B(p, R)$. Suppose l is the upper bound of the number of the hyperplanes $\mathcal{H} + kb_n$ which intersect with $B(p, R)$. Then we have clearly

$$l - 1 \leq \frac{2R}{h}.$$

and

$$d(\mathcal{L}) = h \cdot d(\mathcal{L}') \quad (2.7)$$

From (2.6), (2.7) and *The inequality of Hadamard*, applied to $\mathcal{L}'$, we get

$$\prod_{i=1}^n |b_i| \leq c_2 \cdot d(\mathcal{L}) = c_2 \cdot h \cdot d(\mathcal{L}') \leq c_2 \cdot h \cdot \prod_{i=1}^{n-1} |b_i|,$$

and from $h \leq |b_n|$, (1.20), we have

$$c_2^{-1} |b_n| \leq h \leq |b_n|. \quad (2.8)$$

By $R/r \leq c_1 \cdot \frac{1}{2} \sqrt{n} |b_n| > r$, and (2.8), we get

$$2R \leq 2rc_1 < c_1 \sqrt{n} |b_n|,$$

$$l \geq c_2^{-1} |b_n|.$$

so

$$l - 1 \leq \frac{2R}{h} < \frac{c_1 \sqrt{n} |b_n|}{c_2^{-1} |b_n|} < c_1 c_2 \sqrt{n}.$$

Hence, the number of values for k that have to be considered is bounded by a constant only depending on n . Which values of k need to be considered can easily be deduced from a representation of p (p is

the barycenter of S) as a linear combination of $b_1, \dots, b_n$ (the reduced basis). Suppose we found the value of k in the interval $[k_1, k_2]$, then k runs through every integer of this interval. For example, if k is in the interval of $[-2, 4]$; then, k can be $-2, -1, 0, 1, 2, 3, 4$ and there are 7 hyperplanes of K intersecting rK

Note It will be clear from the example in Section 2.2, how to obtain the values of k

(**) For each fixed value of k , we restrict attention to those $y = \sum_{i=1}^n y'_i b_i$ where $y = (y_1, \dots, y_n)$ and $y \in \mathcal{L}$. We can replace $x = (x_1, \dots, x_n)$ of the original constraint problem by y , where $y = (y_1, \dots, y_n) = \sum_{i=1}^{n-1} y'_i b_i + kb_n$. This leads to an integer constraint problem with $(n-1)$ variables $y'_1, \dots, y'_{n-1}$. We can apply this algorithm for the new problem. Each of the lower dimensional constraint problems treated recursively. The case of dimension is 1 or 0 may serve as a basis for the recursion. We can produce an vector y with $y \in B(p, R) \cap \mathcal{L}$. From $rK \subset B(p, R)$, and solve y subject to the constraint problem, hence

$$y \in rK \cap \mathcal{L} \subset B(p, R) \cap \mathcal{L}$$

Since, we have found a point y in $rK \cap \mathcal{L}$, we have found an integer point $r^{-1}(y)$ in K ; we are done. The problem of solving for y in the constraint problem is an equivalent problem to the original constraint problem with a property that the last coordinate of y belongs to an interval whose length is bounded by the value of k , where $k \leq l-1 \leq c_1 c_2 \sqrt{n}$, so k is a constant only depending on n .

If there doesn't exist such y for that particular value of k , such that $y \in rK \cap \mathcal{L}$, then select a new fixed integer value of k in the interval $[k_1, k_2]$. Repeat (**) for each value of k , until discover that there exists or there doesn't exist such $y \in rK \cap \mathcal{L}$. If there exists at least one vector of y , then terminate. If there doesn't exist such y for every possible value of k then we conclude that our constraint problem

has no integer solution.

Note. It might happen that, for some fixed k the new problem is infeasible. Terminate from that k and select a new value of k .

2.2 An Example of the algorithm in Section 2.1

The following is an example of the algorithm in Section 2.1. The purpose of this algorithm is to find whether there is an integer point in the feasible region, that is to find x as a integer vector such that $Ax \leq b$. As mentioned in the Introduction, this is a feasibility problem. Step 1-Step 4 are preparation steps. The results from those steps are used to transform the problem into a new problem. To solve the new problem is to find a vector y such that $A'y \leq b'$, where

$$A' = \begin{bmatrix} A \\ I \end{bmatrix}, b' = \begin{bmatrix} b \\ 0 \end{bmatrix},$$

and y is a point in a lattice. Step 5 is a concluding step, either there exists a vector y in a lattice, or we need to construct another problem with one dimension less than the original problem, and return to Step 1 with the new problem. Hence we can decide recursively the solution to the original problem. In the example, we will illustrate step by step the algorithm of Section 2.1. Even if in Step 1 we find an integer solution satisfy the constraints problem, we will continue through the calculation, so that it is clear how each step works. In this example, a complete calculation is given for a d -dimension problem only, where d is less than or equal to the number of variables. The problems of the lower dimension can be solved in the same way.

PROBLEM

Inequality constraints are as follows :

$$x_1 + 2x_2 + x_3 \leq 4$$

$$x_1 + 2x_2 + 2x_3 \leq 6$$

$$x_1, x_2, x_3 \geq 0.$$

Figure 3-1: The feasible region of this problem

Step 1 Find a Simplex in K

Use the Simplex method to solve LP problems.

1.1) Maximize any arbitrary linear function, let us say

$$\max z_1$$

$$\text{subject to } z_1 + 2z_2 + z_3 \leq 4$$

$$z_1 + 2z_2 + 2z_3 \leq 6$$

$$z_1, z_2, z_3 \geq 0.$$

An optimal solution is $z_1 = 4, z_2 = 0, z_3 = 0$, then $v_0 = (4, 0, 0)$.

1.2) Maximize another arbitrary linear function,

$$\max z_2$$

$$\text{subject to } x_1 + 2x_2 + x_3 \leq 4$$

$$x_1 + 2x_2 + 2x_3 \leq 6$$

$$x_1, x_2, x_3 \geq 0.$$

An optimal solution is $x_1 = 0, x_2 = 0, x_3 = 3$; so that $v_1 = (0, 0, 3)$.

1.3) Find the equation of a plane passing through two vertices $v_0 = (4, 0, 0)$ and $v_1 = (0, 0, 3)$.

The equation is $3x_1 + 4x_3 - 12 = 0$, then our problem is

$$\max 3x_1 + 4x_3$$

$$\text{subject to } x_1 + 2x_2 + x_3 \leq 4$$

$$x_1 + 2x_2 + 2x_3 \leq 6$$

$$x_1, x_2, x_3 \geq 0.$$

An optimal solution is $x_1 = 2, x_2 = 0, x_3 = 2$; then $v_2 = (2, 0, 2)$, and $(v_2 - v_0), (v_1 - v_0)$ are linearly independent.

1.4) The equation of the plane spanned by 3 points v_0, v_1, v_2 is $-2x_2$. Then we solve the problem of

$$\min -2x_2 \text{ or } \max 2x_2$$

$$\text{subject to } x_1 + 2x_2 + x_3 \leq 4$$

$$x_1 + 2x_2 + 2x_3 \leq 6$$

$$x_1, x_2, x_3 \geq 0.$$

An optimal solution is $x_1 = 0, x_2 = 2, x_3 = 0$; then $v_3 = (0, 2, 0)$; and $(v_3 - v_0), (v_2 - v_0), (v_1 - v_0)$ are linearly independent.

1.5) In the original problem, the number of variables (n) is 3; from above, there are 4 vertices such that $(v_3 - v_0), (v_2 - v_0), (v_1 - v_0)$ are linearly independent and the dimension is 3. We conclude that the convex hull K contains v_0, v_1, v_2, v_3 and K has a positive volume.

Step 2 Find a large Simplex inside K

The convex hull (K) contains

$$v_0 = (4, 0, 0)$$

$$v_1 = (0, 0, 3)$$

$$v_2 = (2, 0, 2)$$

$$v_3 = (0, 2, 0).$$

These vertices form a 3-dimensional simplex in K , let us call it D ; volume of D or $\text{vol } D$ is $2/3$.

Remark:

$$\text{volume of } n\text{-dimensional simplex } (v_0, \dots, v_n) = \frac{1}{n!} \left| \det \begin{pmatrix} 1 & 1 & \dots & 1 \\ v_0 & v_1 & \dots & v_n \end{pmatrix} \right|.$$

2.1) For $i = 0$, let $v_0 = (x_1, x_2, x_3)$; then the volume of a new D composed of $v_0 = (x_1, x_2, x_3), v_1 = (0, 0, 3), v_2 = (2, 0, 2), v_3 = (0, 2, 0)$ is $1/6 (-2x_1, -6x_2 - 4x_3 + 12)$. We will solve two LP problems: maximizing and minimizing the volume of the new D on K . Check for each new v_0 , compare the new volume D with the old one. Replace v_0 with x_0, x_1, x_3 , if the volume of a new D is greater than or equal to $(1 + \epsilon) \text{vol } D$, where ϵ is some positive real number, otherwise v_0 is unchanged. Let $\epsilon = 0.1$.

2.1.1) Solve the problem of

$$\begin{aligned} \max \quad & -\frac{1}{3}x_1 - x_2 - \frac{2}{3}x_3 \\ \text{subject to} \quad & x_1 + 2x_2 + x_3 \leq 4 \\ & x_1 + 2x_2 + 2x_3 \leq 6 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

An optimal solution is (0,0,0).

2.1.2) Solve the problem of

$$\begin{aligned} \min \quad & -\frac{1}{3}x_1 - x_2 - \frac{2}{3}x_3 \\ \text{subject to} \quad & x_1 + 2x_2 + x_3 \leq 4 \\ & x_1 + 2x_2 + 2x_3 \leq 6 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

An optimal solution is (0,1,2).

2.1.3) Check the volume of the new D .

a) Volume of D spanned by $v_0 = (0, 0, 0)$, $v_1 = (0, 0, 3)$, $v_2 = (2, 0, 2)$, $v_3 = (0, 2, 0)$ is 2.

b) Volume of D spanned by $v_0 = (0, 1, 2)$, $v_1 = (0, 0, 3)$, $v_2 = (2, 0, 2)$, $v_3 = (0, 2, 0)$ is $1/3$.

2.1.4) Volume of D with $v_0 = (0, 0, 0) \geq (1 + 0.1)$ volume of D with $v_0 = (4, 0, 0)$;

and volume of D with $v_0 = (0, 1, 2) < (1 + 0.1)$ volume of D with $v_0 = (4, 0, 0)$.

Thus, we now replace v_0 with $(0, 0, 0)$; then the 3-simplex D spanned by

$$v_0 = (0, 0, 0)$$

$$v_1 = (0, 0, 3)$$

$$v_2 = (2, 0, 2)$$

$$v_3 = (0, 2, 0).$$

2.2) Repeat *Step 2.1* for $i = 1, 2, 3$. Finally, a 3-dimensional simplex D is spanned by

$$v_0 = (0, 0, 0)$$

$$v_1 = (4, 0, 0)$$

$$v_2 = (0, 0, 3)$$

$$v_3 = (0, 2, 0).$$

and the volume of D is 4.

step 3 Mapping the Simplex to a regular Simplex

3.1) Select a regular 3-dimensional simplex (S) with vertices

$$z_0 = (0, 0, 0)$$

$$z_1 = (12, 12, 0)$$

$$z_2 = (0, 12, 12)$$

$$z_3 = (12, 0, 12),$$

and the length of this regular simplex (S) is 16.97.

3.2) Find a mapping r such that $(z_i - z_0) = r(v_i - v_0)$ for $1 \leq i \leq 3$. Then

$$r = \begin{pmatrix} 3 & 0 & 4 \\ 3 & 6 & 0 \\ 0 & 6 & 4 \end{pmatrix}$$

3.3) There exist two Euclidean balls such that

$$B(p, r) \subset S \subset T \subset B(p, R),$$

where r is the radius of a ball contained inside S , R is the radius of a ball surrounding T . We can find the values of r and R by transforming S into a standard regular simplex S' in $\mathbb{R}^n$, such that

$$z'_0 = (1, 0, 0, 0)$$

$$z'_1 = (0, 1, 0, 0)$$

$$z'_2 = (0, 0, 1, 0)$$

$$z'_3 = (0, 0, 0, 1),$$

are the vertices of S , and its edge length is 1.414.

By Lemma 2 of Subsection 1.3.2 :

$$B(p, r') \subset S' \subset B(p, R'),$$

with

$$r' = \sqrt{\frac{1}{n(n+1)}}$$

$$R' = \begin{cases} \sqrt{\frac{n^2+2n}{n+1}}, & \text{if } n \text{ is even} \\ \sqrt{\frac{n^2+n-1}{n+1}}, & \text{if } n \text{ is odd.} \end{cases}$$

Hence, for $n = 3$, $r' = 0.289$, $R' = 1.66$, and

$$\begin{aligned} r &= r' \times \frac{\text{edge length of simplex } S}{\text{edge length of simplex } S'} \\ &= 0.289 \times \frac{16.97}{1.414} \\ &= 3.46, \end{aligned}$$

$$\begin{aligned}
 R &= R' \times \frac{\text{edge length of simplex } S}{\text{edge length of simplex } S'} \\
 &= 1.66 \times \frac{16.97}{1.414} \\
 &= 19.92.
 \end{aligned}$$

Step 4 Examine the image of Z^n

4.1) Find the basis $b_i (1 \leq i \leq 3)$ for a lattice $(\mathcal{L})$ such that $\mathcal{L} = rZ^n$. Let $e_i, 1 \leq i \leq 3$ be the standard basis for Z^n . Then $b_i = r(e_i)$, and

$$b_1 = (3, 3, 0)$$

$$b_2 = (0, 6, 6)$$

$$b_3 = (4, 0, 4).$$

4.2) Find the reduced basis for a lattice. We can find a reduced basis by using an computer (the implementation for both algorithms of the reduction process are obtained in Chapter 3). Thus, the reduced basis for this problem is

$$b_1 = (3, 3, 0)$$

$$b_2 = (-4, 6, 2)$$

$$b_3 = (1, -3, 4).$$

Step 5 Concluding Step

Renumber the reduced basis such that $|b_3| = \max \{|b_i| : 1 \leq i \leq 3\}$. Thus, the reduced basis is

$$b_1 = (3, 3, 0) \quad \text{norm} = 4.24,$$

$$b_2 = (1, -3, 4) \quad \text{norm} = 5.66,$$

$$b_3 = (-4, 6, 2) \quad \text{norm} = 7.28.$$

We transform the problem $Ax \leq b$ into $A'y \leq b$, where $x \in \mathbb{R}^n$, ($y = rx$) and $y \in \mathcal{L}$. Then

$$A' = Ar^{-1}.$$

In this example ;

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 2 & 2 \end{bmatrix}, \quad r^{-1} = \begin{bmatrix} 1/6 & 1/6 & -1/6 \\ -1/12 & 1/12 & 1/12 \\ 1/8 & -1/8 & 1/8 \end{bmatrix}.$$

Then

$$A' = \begin{bmatrix} 1/8 & 5/24 & 1/8 \\ 1/4 & 1/12 & 1/4 \end{bmatrix}.$$

Problem 1 (original problem)

Problem 2 (transformed problem)

$$x_1 + 2x_2 + 2x_3 \leq 4$$

$$1/8y_1 + 5/24y_2 + 1/8y_3 \leq 4$$

$$x_1 + 2x_2 + 2x_3 \leq 6$$

$$1/4y_1 + 1/12y_2 + 1/4y_3 \leq 6$$

$$x_1, x_2, x_3 \geq 0, \text{ integers}$$

$$y_1, y_2, y_3 \geq 0, \in \mathcal{L}.$$

Consider the value of $\frac{1}{2}(\sqrt{n}|b_n|)$

For $n = 3, |b_3| = 7.48$, then $\frac{1}{2}\sqrt{3}(7.48) = 6.48$. From Step 3, $r = 3.46$. Hence, $\frac{1}{2}\sqrt{3}(7.48) > r$.

Then we are in the situation of Case 5.2.

a) Let p be a barycenter of S , then

$$p = \frac{1}{4} \{ (0, 0, 0) + (12, 12, 0) + (12, 0, 12) + (0, 12, 12) \} = (6, 6, 6).$$

Represent p as a linear combination of the reduced basis, which is

$$\frac{9}{4}b_1 + \frac{5}{4}b_2 + \frac{1}{2}b_3.$$

Figure 2-2:

b) Find the actual number of hyperplanes ($\mathcal{H} = \sum_{i=1}^{n-1} \mathbb{R}b_i$) intersecting $B(p, R)$ with radius $R = 19.92$.

From Figure 2.2, if M is a point at the boundary of B , then we can represent M as a linear combination of the reduced basis vectors such that

$$M = (m_1, m_2, m_3) = \frac{9}{4}b_1 + \frac{5}{4}b_2 + k'b_3$$

where k' is any real number. Then

$$(m_1 - 6)^2 + (m_2 - 6)^2 + (m_3 - 6)^2 \leq (R)^2 = (19.92)^2$$

$$(8 - 4k' - 6)^2 + (3 + 6k' - 6)^2 + (5 + 2k' - 6)^2 \leq 396.81$$

$$-2.16 \leq k' \leq 3.10.$$

Since we represent $\mathcal{L}$ such that $\mathcal{L} = \cup_{k \in \mathbb{Z}} (\mathcal{H} + kb_n)$, there are 6 hyperplanes of $\mathcal{L}$ intersecting to $B(p, R)$, and $k = (3, 2, 1, 0, -1, -2)$.

c) Represent y as a linear combination of the reduced basis vectors,

$$y = (y_1, y_2, y_3) = y'_1b_1 + y'_2b_2 + y'_3b_3.$$

Fix the value of y_3 , select from one of k . Hence, $y = y'_1 b_1 + y'_2 b_2 + 3b_3$, and

$$y_1, y_2, y_3 = (3y'_1 + y'_2 - 12), (3y'_1 - 3y'_2 + 18), (4y'_2 + 6) \quad (2.9)$$

Substituting y from (2.9) into *Problem 2*, we now obtain *Problem 3* such that

Problem 3

$$y'_1 \leq 1$$

$$y'_1 + y'_2 \leq 6$$

$$3y'_1 + y'_2 \geq 12$$

$$3y'_1 - 3y'_2 \geq -18$$

$$y'_2 \geq -\frac{3}{2}$$

$$y'_1, y'_2 \in \mathbb{Z}.$$

We now return to *Step 1* with the new integer programming problem with only two variables, apply the algorithm to this new problem. We treat the lower dimension problems recursively, then we can decide the problem. If this problem has no solution (no such $y \in \mathcal{L}$), then we fix a new value of k , obtain a new problem with two variables and repeat the process.

Finally, we can decide that the transformation problem has a solution or has no solution. If the transformation problem has a solution (at least one), this implies that the original integer linear program has at least one integer point in its feasible region, and we can obtain x which is a solution for the original problem from $x = r^{-1}y$. If the transformed problem has no solution then the original problem has no integer point (not even one) in its feasible region. From the next section, Lenstra [12], has shown that the running time of this algorithm is bounded by a polynomial time in the input length with a fixed value of number of variables. This finishes the example of the algorithm.

2.3 Analysis of the algorithm

We measure the complexity (the amount of work done) of an algorithm as a function of the size of the input. The input in combinatorial optimization problems is a combinatorial object — a graph, a set of integers, a family of finite sets, and so on. We must somehow encode it or represent it as a sequence of symbols over some fixed alphabet such as bits or ASCII characters. We define the size of the input to be the length of this sequence, that is, the number of symbols in it. We define the complexity of the algorithm for the input size to be the worst-case behavior of the algorithm.

If $f(n)$, $g(n)$ are functions from the positive integers to the positive reals, we define $f(n) = O(g(n))$ if there exists a constant $c > 0$ such that for large enough n , $f(n) \leq cg(n)$. We speak of $f(n) = O(g(n))$ as the growth rates of a function $f(n)$ and O as "big-oh".

An algorithm is a practically useful solution to a computational problem only if its complexity grows polynomially with respect to the size of the input. The complexity may not be a polynomial itself but if bounded by a polynomial, this also qualifies. Examples of such polynomially bounded functions are $n^{2.5}$ and $n \log n$.

Some general rules for analysis of the algorithm

- 1) The running time of each assignment, read and write statement can usually be taken to be $O(1)$.
- 2) The running time of a sequence of statements is determined by the sum rule. That is, the running time of the sequence is the largest running time of any statement in the sequence. For example, suppose we have three steps whose running times are, respectively, $O(n^3)$, $O(n^3)$, $O(n \log n)$. Then the running time of the first two steps executed sequentially is $O(\max(n^3, n^3))$ which is $O(n^3)$. The running time of all three together is $O(n^3, n \log n)$ which is $O(n^3)$.

3) The running time of an *if*-statement is the cost of the conditionally executed statements plus the time for evaluating the conditions. The time to evaluate the condition is normally $O(1)$. The time for an *if-then-else* construct is the time to evaluate the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.

4) The time to execute a loop is the sum, over all times around the loop, of the time to execute the body and the time to execute the condition for termination (usually the latter is $O(1)$). Often this time is, up to a constant factor, the product of the number of times around the loop and the largest possible time for the one execution of the body, but we must consider each loop separately to make sure. The number of iterations around a loop is usually clear, but there are times when the number of iterations can not be computed precisely. It could even be that the program is not a finite algorithm, and there is no limit to the number of times we go around certain loops.

The rule for products is the following: if $T_1(n)$ and $T_2(n)$ are $O(f(n))$ and $O(g(n))$, respectively, then $T_1(n)T_2(n)$ is $O(f(n)g(n))$, and it follows from the product rule that $O(cf(n))$ means the same things as $O(f(n))$, if c is any positive constant.

5) If we have a program with procedures, none of which is recursive, then we can compute the running time of the various procedures one at a time, and use the rule of sum and the rule of product to evaluate the running time.

6) If there are recursive procedures, then we can not find an ordering of all procedures so that each calls only previously evaluated procedures. What we must do is associate with each recursive procedure an unknown time function $T(n)$, where n measures the size of the arguments to the procedure. We

can get a recurrence for $T(n)$, that is, an equation of $T(n)$ in terms of $T(k)$ for various values of k

For example, we might get

$$T(n) = \begin{cases} c + T(n-1), & \text{if } n > 1 \\ d, & \text{if } n \leq 1, \end{cases}$$

where c, d are some constants

Complexity of the algorithm

We first define the length of the input for a constraint problem with a fixed n . Since A is an $(m \times n)$ -matrix and b is an m -vector, there are $(n+1)m$ coordinates. Let a denote the maximum of the absolute values of the coefficients of A and b . Then each coordinate has at most a constant times $\log(a+2)$ binary digits, so we can take the length of the input l as $(n+1)m \cdot \log(a+2)$ or $nm \log(a+2)$

From Step 1 and Step 2, we solve several linear programming problems to get $v_0, v_1, \dots, v_M$.

By [10], Khachiyan shows that each linear programming can be solved within a polynomial time in l , in fact we need $O(n^3(n^2 + m)l)$. Hence, in these 2 steps, the running time is a polynomial bounded by a function of l . From Step 3, Lovass, [7] has shown that a transformation r can be founded in polynomial time bounded by a function of l , even for vary n .

In Step 4, as will see later in Chapter 3, there are two algorithms for finding a reduced basis. Even though the complexity of the reduction process I is bounded by polynomial time of its input length, but it is difficult to compute its complexity precisely. Therefore we will not include the complexity for the reduction process I in computing the complexity of this algorithm. We consider the reduction process II instead, which is $O(n^4 \log B)$ where B is the upper bound of the norm of input basis. Since the input basis is computed from the convex set K , which are the optimum solutions of several LP problems,

each optimal solution is bounded by the length of input (l) , [3], hence B is also bounded by the input length (l) . Thus, the complexity of *Step 4* is bounded by a polynomial time in the length of input (l) .

For *Step 5*, if the algorithm terminates in *Case 1*, then it is clear that this step can be done in a polynomial time in l . In *Case 2*, we construct a new constraint problem with only $(n - 1)$ variables, and its input size is bounded by the input size of the previous one. Therefore, in *Case 2*, its complexity depends on *Step 1* to *Step 4*, which is a polynomial time bounded by l . Since each step is a polynomial time bounded by l , thus, in *Case 2*, we can finish in a time bounded by a polynomial in l .

From the above information, each step can be done within a polynomial time of l . Therefore, from the rule of summation, the complexity of this algorithm is bounded by a polynomial function of its input length.

It is very difficult to compute the actual complexity, especially in the worst case performance. From *Step 5*, it is not clear how many new integer programming problems we need to construct in order to get an integer point. Its complexity might be a very large polynomial of function. In fact, this algorithm is not practical for a large value of n ; the cutting plane method or branch and bound method also work well. Consider *Steps 1, 2, 3*; we computed their complexity based on an ellipsoid method which is not a practical method also. In general, this algorithm is useful only from a theoretical point of view.

Chapter 3

A REDUCTION PROCESS FOR LATTICES

In Chapter 2, we referred to a process that finds a reduced basis for a lattice. This process changes a given basis for a lattice $(\mathcal{L})$ in $\mathbb{R}^n$ into a reduced basis (b_i) of $\mathcal{L}$. We shall define any basis as a reduced basis if

$$\prod_{i=1}^n |b_i| \leq c_2 \cdot d(\mathcal{L}), \quad (3.1)$$

where $\{b_i\}$ is a basis for $\mathcal{L}$, $(1 \leq i \leq n)$, c_2 is a constant only depending on n , $d(\mathcal{L})$ is a determinant of $\mathcal{L}$.

We will describe two algorithms for changing a given basis for $\mathcal{L}$ to a reduced one. The algorithms are given in Section 3.1 and Section 3.2. The first algorithm, we shall call the *Reduction process I*, is a recursive algorithm. This is an algorithm by *Lenstra* -see [12], its complexity is bounded by a polynomial function of the input length only for a fixed value of n . Whereas the second one (the reduction process II) is a non-recursive one, this is an algorithm by *Lovász* -see [13], its complexity is bounded by a polynomial function of its input length even for a vary n . The second algorithm is an improvement of the first one.

At the end of this Chapter, we include the implementation of both algorithms, some results and also some limitations.

3.1 The reduction process I

Let $b_1, \dots, b_n \in \mathbb{R}^n$ be n linearly independent vectors. Put $\mathcal{L} = \sum_{i=1}^n \mathbb{Z} b_i$; this is a lattice in the linear subspace $\sum_{i=1}^n \mathbb{R} b_i$ and $d(\mathcal{L}) = |\det(b_1, \dots, b_n)|$. If we regard the $b_1, \dots, b_n$ in $\det(b_1, \dots, b_n)$ first as rows and then as columns and multiply the two determinants together, we have

$$d(\mathcal{L})^2 = \det\{(b_i, b_j)\}_{1 \leq i, j \leq n}$$

The reduction process is as follows

Step 1

If $n > 1$ then renumber the b_i such that $|b_1| = \min\{|b_i| : 1 \leq i \leq n\}$, and let V be the hyperplane in $\mathbb{R}^n$ orthogonal to b_1 so that $V = \{x \in \mathbb{R}^n : (x, b_1) = 0\}$. We can take the projection $\bar{b}_i$ of b_i on V namely

$$\bar{b}_i = b_i - \frac{(b_i, b_1)}{(b_1, b_1)} b_1 \quad \text{for } (2 \leq i \leq n),$$

and put $\bar{\mathcal{L}} = \sum_{i=2}^n \mathbb{Z} \bar{b}_i$. Then $\bar{\mathcal{L}}$ is a lattice in the $(n-1)$ -dimensional vector space $V = \sum_{i=2}^n \mathbb{R} \bar{b}_i$ and $d(\bar{\mathcal{L}}) = \frac{d(\mathcal{L})}{|b_1|}$. Repeat this step recursively until $n = 1$; for $n = 1$, we have $d(\mathcal{L}) = |b_1|$

Step 2

We can change the basis $\bar{b}_2, \dots, \bar{b}_n$ for $\bar{\mathcal{L}}$ into a basis $\bar{b}'_2, \dots, \bar{b}'_n$ for $\bar{\mathcal{L}}$ satisfying (3.1). The new basis can be written in terms of the old one by $\bar{b}'_i = \sum_{j=2}^n m_{ij} \bar{b}_j$, with $m_{ij} \in \mathbb{Z}$, and its determinant, $\det(m_{ij})_{2 \leq i, j \leq n} = \pm 1$. Now we can write $b'_i = \sum_{j=2}^n m_{ij} b_j$ (for $2 \leq i, j \leq n$). Then $b_1, b'_2, \dots, b'_n$ is a basis for $\mathcal{L}$ and $\bar{b}'_i$ is the projection of b'_i on V :

$$\bar{b}'_i = b'_i - \frac{b'_i, b_1}{b_1, b_1} b_1, \quad (2 \leq i \leq n).$$

Let n_i be the integer nearest to $(b'_i, b_1)/(b_1, b_1)$, and put $b''_i = b'_i - n_i b_1$, $2 \leq i \leq n$. Then $b_1, b''_2, \dots, b''_n$ is a basis for $\mathcal{L}$.

We show how to get a matrix M with $m_{ij} \in \mathbb{Z}$ and its determinant is ± 1 . In the case of $n = 1$, we start M with the identity matrix; for $n > 1$, we can generate a new matrix M from the previous M at the 1 dimension lower and n_i such that

$$\text{a matrix } M \text{ at } k\text{-dimension } \bar{L} = \begin{bmatrix} 1 & 0 \\ n_i & \text{a matrix } M \text{ at } k-1 \text{ dimension } \bar{L} \end{bmatrix}$$

for $2 \leq i \leq k$

Step 3

At each k -dimensional $\bar{L}$, we get $b_1, b''_2, \dots, b''_k$ from Step 2. We need to check whether $b_1, b''_2, \dots, b''_k$ for $1 \leq k \leq n$ is a reduced basis (i.e. satisfies the equation (3.1)). Let $c' = \frac{1}{4}(c-1)^{-1}$, where $c > \frac{4}{3}$, therefore $0 < c' < \frac{3}{4}$. We distinguish two cases.

case 1 If $c'|b_1|^2 \leq |b_i|^2$ for $i = 2, \dots, k$. Hence, the basis $b_1, b''_2, \dots, b''_k$ for $\bar{L}$ satisfies (3.1), and $b_1, b''_2, \dots, b''_k$ is a reduced basis for the k -dimensional lattice $\bar{L}$. If $k = n$ the algorithm terminates.

case 2 If $c'|b_1|^2 > |b_i|^2$ for $2 \leq i \leq k$. Hence, the shortest vector in the basis $b_1, b''_2, \dots, b''_k$ for the k -dimensional $\bar{L}$ is substantially shorter than the shortest vector b_1 in the basis $b_1, \dots, b_k$. Now we return to the beginning of the algorithm Step 1 with $b_1, \dots, b_k$ replaced by $b_1, b''_2, \dots, b''_k$.

This finishes the algorithm. The following is the proof of how the algorithm can terminate in Case 1.

Proof that the Basis in Case 1 is Reduced

To prove that $b_1, b''_2, \dots, b''_n$ for $\mathcal{L}$ satisfies (3.1). Since $\bar{L}$ is a lattice in the $(n-1)$ -dimensional vector space $\sum_{i=2}^n \mathbb{Z}b_i$, and

$$d(\bar{L}) = \frac{d(\mathcal{L})}{|b_1|}. \quad (3.2)$$

From Step 2

$$b_i'' = b_i' - n_i b_1$$

where $n_i =$ integer nearest to $\frac{(b_i', b_1)}{(b_1', b_1)}$.

$$\text{and } b_i' = b_i'' + \frac{(b_i', b_1)}{(b_1', b_1)} b_1$$

From the above equations, we get

$$b_i'' = b_i' + r_i b_1 \text{ for some } r_i \in \mathbb{R}, |r_i| \leq 1/2, \text{ and } 2 \leq i \leq n$$

Since, $(b_i', b_1) = 0$, this implies that

$$|b_i''|^2 \leq |b_i'|^2 + \frac{1}{4} |b_1|^2 \quad (3.3)$$

By the condition in Case 1, $c' |b_1|^2 \leq |b_i|^2$ for $i = 2, \dots, n$ and (3.3), we get

$$|b_i''|^2 \leq |b_i'|^2 \left(1 + \frac{1}{4} c'^{-1}\right) = c |b_i'|^2, \quad (3.4)$$

since $c' = \frac{1}{4}(c-1)^{-1}$ or $c = (1 + \frac{1}{4}c'^{-1})$. We want to prove that:

$$\prod_{i=1}^n |b_i| \leq c_2 d(\mathcal{L}) \leq c^{n(n-1)/4} d(\mathcal{L}), \text{ where } c > \frac{4}{3}$$

We prove this by induction. For $n = 1$ it is obvious. Assume that it true for $\bar{\mathcal{L}}$ ($n-1$ dimension)

From Step 2, we changed the basis $b_2, \dots, b_n$ for $\bar{\mathcal{L}}$ into a basis $b_2', \dots, b_n'$ for $\bar{\mathcal{L}}$ satisfying

$$\prod_{i=1}^n |b_i'| \leq c_2' d(\mathcal{L}) \quad (3.5)$$

where $c_2' = c^{\frac{(n-1)(n-2)}{4}}$. From (3.4).

$$\prod_{i=2}^n |b_i''|^2 \leq c^{(n-1)} \prod_{i=2}^n |b_i'|^2.$$

$$\prod_{i=2}^n |b_i''| \leq c^{(n-1)/2} \prod_{i=2}^n |b_i'|.$$

then (3.5) implies that

$$|b_1| \prod_{i=2}^n |b_i''| \leq c^{(n-1)/2} |b_1| \prod_{i=2}^n |b_i'|.$$

$$|b_1| \prod_{i=2}^n |b_i''| \leq c^{(n-1)/2} c^{(n-1)(n-2)/4} d(\mathcal{L})$$

$$\leq c^{(n-1)/2} d(\mathcal{L})$$

Hence the basis $b_1, b_2'', \dots, b_n''$ for $\mathcal{L}$ satisfies the required inequality.

For *Case 2* of Step 5, since $c' |b_1'|^2 > |b_1''|^2$ for some $i \in \{2, \dots, n\}$. Then for this i we have by

(3.3)

$$|b_i''| \leq (c' + \frac{1}{4}) |b_1'|^2. \quad (3.6)$$

where $(c' + \frac{1}{4}) < 1$ since $0 < c' < \frac{3}{4}$. This contradicts the fact that b_1 is a vector with the minimum norm. Then we return to Step 1 with a new basis. This finishes the proof.

Complexity of the algorithm

Let n and c be fixed, and the initial basis vectors for a lattice $\mathcal{L}$ have rational coefficients. We denote these initial vectors by b_i^0 , $1 \leq i \leq n$, in order to avoid confusion with the changing meaning of b_i during the several passes of the algorithm.

The basic operation for this algorithm is the number of times that we loop through the algorithm. This implies that we only consider the number of times passing *Case 2*.

Let a_1 denotes the maximum absolute value of the numerators and denominators of the coefficients of $b_1^0, \dots, b_n^0$, and let $m(\mathcal{L}) = \min\{|x|^2 : x \in \mathcal{L}, x \neq 0\}$. From (3.6), the worst case that can happen is that we always get $|b_i''|^2$ is less than but almost equal to $(c' + 1/4)|b_1|^2$. This is a situation of *Case 2*, so we keep on reducing the basis in that dimension until $|b_i''|^2 = m(\mathcal{L})$. This situation can happen in every

lower dimension. If we assume that it is an n dimensions, then we replace b_1^0 by b_1'' , and reduce b_1^0 . Let $u = c' + 1/4$ and $|b_1^0|^2 = K$. Each times $|b_1^0|^2$ is reduced by a factor of u (since $|b_1''|^2$ is almost equal to $|b_1^0|^2$). The sequence of $|b_1''|^2$ generated from the reduction process will be $K, Ku, Ku^2, \dots, Ku^r$. If $Ku^r = m(\mathcal{L})$ then we stop the process, and r is the number of times passing through Case 2 for the current basis n . Hence,

$$= \frac{\log(K/m(\mathcal{L}))}{|\log u|}, \text{ or } r = \frac{\log |b_1^0|^2 / m(\mathcal{L})}{|\log(c' + 1/4)|}$$

Since this situation can occur in every lower dimension, we can conclude that the complexity of this algorithm is a function bounded by $\log\{|b_1^0|^2 / m(\mathcal{L})\} / |\log(c' + 1/4)|$

Consider the value $m(\mathcal{L})$. We can use the lower bound for $m(\mathcal{L})$ such that $m(\mathcal{L}) \geq \text{den}(\mathcal{L})^{-2}$, where $\text{den}(\mathcal{L})$, the denominator of $\mathcal{L}$, denotes the least positive integer k with $\mathcal{L} \in \frac{1}{k} \mathbb{Z}^n$, it is the least common multiple of the denominators of the coefficients of the b_i^0 . Thus,

$$\frac{\log(|b_1^0|^2 / m(\mathcal{L}))}{|\log(c' + 1/4)|} \leq \frac{\log(|b_1^0|^2 / \text{den}(\mathcal{L})^{-2})}{|\log(c' + 1/4)|}$$

Let $a_1 = p/q$ where a_1 is a maximum of absolute values of the coefficients $b_1^0, \dots, b_n^0$, and $na_1^2 = np^2/q^2$. Clearly, $np^2/q^2 > |b_1^0|^2$, then

$$c'' \log |b_1^0|^2 \text{den}(\mathcal{L})^2 \leq c'' \log \frac{np^2}{q^2} \text{den}(\mathcal{L})^2,$$

where $c'' = 1/|\log(c' + 1/4)|$. The value of a_1 is a factor of $\text{den}(\mathcal{L})$ and we can write $\text{den}(\mathcal{L})$ as a function of a_1 . Thus, $c'' \log(np^2/q^2) \text{den}(\mathcal{L})^2$ is a polynomial function of $\log a_1$. We conclude that the complexity of this algorithm is bounded by a polynomial in the length of its input.

It is difficult to calculate the actual work for this algorithm, especially in the worst case. The complexity could take a very large polynomial function of the number of constraints if we fixed the number of

variables, but the constant term might be an exponential function of the number of variables. There still is room for improvement. Lovass, [13], introduced *The Reduction Process II*, which has running time bounded by a polynomial function in the length of the input even for the varying number of constraints and variables. We will present in the next section

3.2 The reduction process II

Let $\mathcal{L}$ be an n -dimensional lattice in $\mathbb{R}^n$, n a positive integer. Then $\mathcal{L}$ has the structure

$$\mathcal{L} = \sum_{i=1}^n \mathbb{Z} b_i = \left\{ \sum_{i=1}^n m_i b_i : m_i \in \mathbb{Z} \right\} \quad (1 \leq i \leq n),$$

and $b_1, \dots, b_n$ is a basis for $\mathcal{L}$. In this algorithm, a reduced basis for $\mathcal{L}$ is defined in such a way that

$$|\mu_{ij}| \leq \frac{1}{2} \quad (1 \leq j < i \leq n), \quad (3.7)$$

and

$$|b_i^* + \mu_{i,i-1} b_{i-1}^*|^2 \geq c_3 |b_{i-1}^*|^2 \quad (1 < i \leq n) \quad (3.8)$$

where

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{ij} b_j^*, \quad (1 \leq i \leq n), \quad (3.9)$$

$$\mu_{ij} = \frac{(b_i, b_j^*)}{(b_j^*, b_j^*)}, \quad (1 \leq j < i \leq n), \quad (3.10)$$

and c_3 is a constant strictly between $\frac{1}{4}$ and 1, and $(\cdot, \cdot)$ denotes the ordinary inner product in $\mathbb{R}^n$.

This is the *Gram-Schmidt orthogonalization process*. Notice that b_i^* is the projection of b_i on the orthogonal complement of $\sum_{j=1}^{i-1} \mathbb{R} b_j$, and that $\sum_{j=1}^{i-1} \mathbb{R} b_j = \sum_{j=1}^{i-1} \mathbb{R} b_j^*$, for $1 \leq i \leq n$; and μ_{ij} , for $1 \leq j < i \leq n$ are real numbers. Hence $b_1^*, \dots, b_n^*$ is an orthogonal basis of $\mathbb{R}^n$, and $b_1, \dots, b_n$ is a reduced basis for a lattice $(\mathcal{L})$, if (3.7), (3.8) are satisfied.

Before, we discuss the algorithm, we would like to relate the reduction processes I and II by showing that they will reduce a given basis in the same sense. We describe it in *proposition 1*.

Proposition 1. Let $b_1, \dots, b_n$ be a reduced basis for a lattice $(\mathcal{L})$ in $\mathbb{R}^n$, let $b_1^*, \dots, b_n^*$ be defined as in (3.9). Then we have:

$$|b_j|^2 \leq 2^{i-1} |b_i^*|^2 \quad \text{for } 1 \leq j < i \leq n, \quad (3.11)$$

$$d(\mathcal{L}) \leq \prod_{i=1}^n |b_i| \leq 2^{\frac{n(n-1)}{4}} d(\mathcal{L}). \quad (3.12)$$

Remark If c_3 in (3.8) is $3/4$, then (3.11), (3.12) are in the form above. If we replace c_3 by y , with $1/4 < y < 1$, then the power of 2 appearing in (3.11), (3.12) must be replaced by the same power of $4/(4y-1)$.

Proof: From (3.7), (3.6), then

$$|b_i^*|^2 \geq \left(\frac{3}{4} - \mu_{i-1}^2\right) |b_{i-1}^*|^2 \geq \frac{1}{2} |b_{i-1}^*|^2,$$

or we also can write it as

$$|b_i^*|^2 \geq 2^{i-1} |b_{i-1}^*|^2,$$

$$|b_{i-1}^*|^2 \leq 2^{i-1} |b_i^*|^2 = 2^{-(i-1)} |b_i^*|^2. \quad (3.13)$$

Now, we can prove by induction that

$$|b_j^*|^2 \leq 2^{i-j} |b_i^*|^2 \quad \text{for } 1 \leq j < i \leq n; \quad (3.14)$$

for $i = 2$, it is obvious from (3.13); then we assume that

$$|b_j^*|^2 \leq 2^{i-j} |b_i^*|^2 \quad \text{for } 1 \leq j < i \leq n-1, \quad (3.15)$$

and from (3.13), letting $i = n$, we have

$$|b_{n-1}^*|^2 \leq 2 |b_n^*|^2.$$

Substituting this into (3.15), for $i = n-1$, we get

$$|b_j^*|^2 \leq 2^{(n-1)-j} 2 |b_n^*|^2 = 2^{n-j} |b_n^*|^2, \quad \text{for } 1 \leq j \leq n.$$

If we substitute each $i = 2, \dots, n-2$ in (3.13) and (3.15), we can prove (3.14)

From (3.9):

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{ij} b_j^*,$$

and

$$|b_i|^2 = |b_i^*|^2 + \sum_{j=1}^{i-1} \mu_{ij}^2 |b_j^*|^2.$$

then, from (3.14) and $|\mu_{ij}| \leq \frac{1}{2}$, we get

$$\begin{aligned} |b_i|^2 &\leq |b_i^*|^2 + \sum_{j=1}^{i-1} \left\{ \frac{1}{4} (2^{i-j}) |b_j^*|^2 \right\} \\ &\leq \left(1 + \frac{1}{4} (2^i - 2) \right) |b_i^*|^2 \\ &\leq (2^{i-1}) |b_i^*|^2. \end{aligned}$$

It follows that $|b_j|^2 \leq 2^{j-1} |b_j^*|^2$. From (3.14)

$$|b_j|^2 \leq 2^{j-1} \cdot 2^{i-j} |b_i^*|^2 = 2^{i-1} |b_i^*|^2,$$

for $1 \leq j \leq n$. This finishes the proof of (3.11).

From the determinant of $\mathcal{L}$, $d(\mathcal{L}) = | \det(b_1, \dots, b_n) |$, and from (3.9), $b_i = b_i^* + \sum_{j=1}^{i-1} \mu_{ij} b_j^*$; then each column of b_i is a linear combination of the previous column of b_i . Thus $\det(\mathcal{L}) = | \det(b_1, \dots, b_n) | = | \det(b_1^*, \dots, b_n^*) |$. As before, we can write $d(\mathcal{L})^2 = | \det(b_i, b_j) |_{1 \leq i, j \leq n}$, and since the b_i^* are pairwise orthogonal, then

$$d(\mathcal{L}) = \prod_{i=1}^n |b_i^*|.$$

From $|b_i^*| \leq |b_i|$ and $|b_i| \leq 2^{\frac{(i-1)}{2}} |b_i^*|$, then $\prod_{i=1}^n |b_i| \leq 2^{\frac{n(n-1)}{4}} \prod_{i=1}^n |b_i^*|$. We now obtain (3.12)

Note: From (3.12), we can see that a reduced basis for *The Reduction Process II* is also reduced in the same sense as a reduced basis in *The Reduction Process I*.

The Algorithm for Reduction Process II

Step 1 Compute b_i^* ($1 \leq i \leq n$), and μ_{ij} ($1 \leq j < i \leq n$) from (3.9), (3.10), use c_3 as $3/4$

Step 2 Check whether b_i^* ($1 \leq i \leq n$), μ_{ij} ($1 \leq j < i \leq n$) satisfy the condition (3.7), (3.8) with

$c_3 = 3/4$. Let k be the current subscript and $k \in \{1, \dots, n+1\}$. Thus (3.7), (3.8) are

$$|\mu_{ij}| \leq \frac{1}{2} \quad \text{for } 1 \leq j < i < k. \quad (3.16)$$

$$|b_i^* + \mu_{i+1} b_{i+1}^*|^2 \geq \frac{3}{4} |b_{i+1}^*|^2 \quad \text{for } 1 < i < k. \quad (3.17)$$

If (3.16), (3.17) are satisfied by $b_1, \dots, b_k$, then $b_1, \dots, b_k$ is a reduced basis of $\mathcal{L}'$, where $\mathcal{L}'$ is a lattice in $\mathbb{R}^k$, and if $k = n+1$, then the basis $b_1, \dots, b_n$ is a reduced basis for $\mathcal{L} \in \mathbb{R}^n$. The algorithm terminates

We begin k with 2, and for $k \leq n$, we first achieve that

$$|\mu_{kk-1}| \leq \frac{1}{2} \quad \text{if } k > 1. \quad (3.18)$$

If this doesn't hold, let r be the nearest integer to μ_{kk-1} and replace b_k by $b_k - rb_{k-1}$. The numbers μ_{kj} with $j < k-1$ are replaced by $\mu_{kj} - r\mu_{k-1j}$, since

$$\mu_{kj} = \frac{(b_k, b_j^*)}{(b_j^*, b_j^*)} \quad \text{is replaced by} \quad \frac{((b_k - rb_{k-1}), b_j^*)}{(b_j^*, b_j^*)},$$

which is

$$\frac{(b_k, b_j^*)}{(b_j^*, b_j^*)} - \frac{r(b_{k-1}, b_j^*)}{(b_j^*, b_j^*)} = \mu_{kj} - r\mu_{k-1j}.$$

Moreover, μ_{kk-1} becomes $\mu_{kk-1} - r$, since $\mu_{kk-1} = \frac{(b_k, b_{k-1}^*)}{(b_{k-1}^*, b_{k-1}^*)}$ is replaced by $\frac{((b_k - rb_{k-1}), b_{k-1}^*)}{(b_{k-1}^*, b_{k-1}^*)}$, which

is $\mu_{kk-1} - r$. The other μ_{ij} ($i \neq k$) and b_i^* ($i < k$) are unchanged.

Case of $i = k$

$$b_k^* = b_k - \sum_{j=1}^{k-1} \mu_{kj} b_j^*.$$

$$\begin{aligned} b_k^* \text{ is replaced by } (b_k - r b_{k-1}) - \sum_{j=1}^{k-2} (\mu_{kj} - r \mu_{k-1,j}) b_j^* - (\mu_{kk-1} - r) b_{k-1}^* \\ = (b_k - \sum_{j=1}^{k-1} \mu_{kj} b_j^*) - r (b_{k-1} - \sum_{j=1}^{k-2} \mu_{k-1,j} b_j^*) + r b_{k-1}^* \\ = b_k^* - r b_{k-1}^* + r b_{k-1}^* \\ = b_k^* \end{aligned}$$

Thus, it is clear that $b_i^* (i > k)$ remain unchanged

After all these changes, we have a new basis $b_1, \dots, b_n$ such that b_k is replaced by $b_k - r b_{k-1}$ and all other $b_i (i \neq k)$ are unchanged, and (3.16) holds. Then we distinguish 2 cases.

Case 1 Suppose $k \geq 2$ and $|b_k^* + \mu_{kk-1} b_{k-1}^*|^2 < \frac{3}{4} |b_{k-1}^*|^2$. In this case

a) interchange b_{k-1} and b_k , leave the other b_i the same.

b) the vector b_{k-1}^* and b_k^* and the numbers $\mu_{kk-1}, \mu_{k-1,j}, \mu_{kj}, \hat{\mu}_{kk-1}, \mu_{ik}$ for $j < k-1$ and $i > k$ have now to be changed.

The following are the formulas for changing all elements in a), b). The proof for these formulas will be given after finishing the description of this algorithm.

Let c_i, c_i^* and μ_{ij} denote the vectors and the numbers that will replace b_i, b_i^* and μ_{ij} respectively

The new basis $c_1, \dots, c_n$ is given by

$$c_{k-1} = b_k, \quad c_k = b_{k-1}, \quad c_i = b_i, \quad \text{for } i \neq k-1, k$$

Let c_i^* be the projection of c_i such that $c_i^* = c_i - \sum_{j=1}^{k-1} \mu_{ij} c_j^*$.

Formula 1: $c_{k-1}^* = b_k^* + \mu_{kk-1} b_{k-1}^*$

Formula 2 $v_{kk-1} = \mu_{kk-1} |b_{k-1}^*|^2 / |c_{k-1}^*|^2$

Formula 3 $c_k^* = b_{k-1}^* - v_{kk-1} c_{k-1}^*$

Formula 4 for $i > k$

$$v_{ik-1} = \mu_{ik-1} v_{kk-1} + \mu_{ik} |b_k^*|^2 / |c_{k-1}^*|^2$$

and

$$v_{ik} = \mu_{ik-1} - \mu_{ik} \mu_{kk-1}$$

Formula 5 for $1 \leq j < k-1$

$$v_{k-1j} = \mu_{kj}$$

and

$$v_{kj} = \mu_{k-1j}$$

Formulas 6 if $1 \leq j < i \leq n$ and $\{i, j\} \cap \{k-1, k\} = \emptyset$, then

$$v_{ij} = \mu_{ij}$$

Remarks For $j > k-1$ and $i < k$ there doesn't exist values of $\mu_{k-1j}, \mu_{kj}, \mu_{ik-1}, \mu_{ik}$, there is a value for μ_{rs} only if $1 \leq s < r \leq n$

After these changes have been made, we replace k by $k-1$. Then we are in the situation described by (3.11) (3.12) we proceed with the algorithm from there

Case 2 Suppose that $k=1$ or $|b_k^* + \mu_{kk-1} b_{k-1}^*|^2 \geq \frac{3}{4} |b_{k-1}^*|^2$. In this case, we first achieve that

$$|\mu_{kj}| \leq \frac{1}{2} \quad \text{for } 1 \leq j \leq k-1 \quad (3.19)$$

If (3.19) doesn't hold, let l be the largest index $< k$ with $|\mu_{kl}| > \frac{1}{2}$, let r be the integer nearest to μ_{kl} , and replace b_k by $b_k - rb_l$. The numbers μ_{kj} with $j < l$ are then replaced by $\mu_{kj} - r\mu_{lj}$; and μ_{kl} by $\mu_{kl} - r$; the other μ_{ij} and all b_i^* are unchanged. This is repeated until (3.19) holds.

Next, we replace k by $k + 1$, then we are in the situation described by (3.11), (3.12); and we proceed with the algorithm from there.

Notice that in the case $k = 1$, we have done no more than replace k by 2. This finishes the description of the algorithm.

Proof of the formulas used in Case 1

Let $b_1, \dots, b_n$ be the current basis and b_i^*, μ_{ij} be as in (3.9), (3.10); k be the current subscript as defined in the algorithm. By c_i, c_i^* and v_{ij} , we denote the vectors and numbers that will replace b_i, b_i^* and μ_{ij} respectively. Then the new basis is given by

$$c_{k-1} = b_k, \quad c_k = b_{k-1}, \quad c_i = b_i, \quad \text{for } i \neq k-1, k.$$

Formula 1: to prove $c_{k-1}^* = b_k^* + \mu_{kk-1}b_{k-1}^*$.

Proof

Since c_{k-1}^* is the projection of b_k on the orthogonal complement of $\sum_{j=1}^{k-2} \mathbb{R}b_j$, and $b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{ij}b_j^*$, then

$$c_{k-1}^* = b_k - \sum_{j=1}^{k-2} \mu_{kj}b_j^*,$$

and

$$\begin{aligned} b_k^* &= b_k - \sum_{j=1}^{k-2} \mu_{kj}b_j^* - \mu_{kk-1}b_{k-1}^*, \\ &= c_{k-1}^* - \mu_{kk-1}b_{k-1}^*. \end{aligned}$$

Hence, $c_{k-1}^* = b_k^* + \mu_{kk-1}b_{k-1}^*$.

Formulas 2.3.5, 6 : to prove

$$v_{kk-1} = \mu_{kk-1} |b_{k-1}^{\circ}|^2 / |c_{k-1}^{\circ}|^2,$$

$$c_k^{\circ} = b_{k-1}^{\circ} - v_{kk-1} c_{k-1}^{\circ},$$

$$v_{k-1,j} = \mu_{kj}, \quad v_{kj} = \mu_{k-1,j}, \quad \text{if } (1 \leq j < k-1),$$

and

$$v_{ij} = \mu_{ij}, \quad \text{if } \{i, j\} \cap \{k-1, k\} = \emptyset, \quad 1 \leq j < i \leq n.$$

Proof.

From

$$c_k^{\circ} = c_k - \sum_{j=1}^{k-1} v_{kj} c_j^{\circ}, \quad (3.20)$$

and $c_i^{\circ} = b_i^{\circ}$ for $i \neq k-1, k$,

$$v_{kj} = \frac{(c_k, c_j^{\circ})}{(c_j^{\circ}, c_j^{\circ})} = \frac{(b_{k-1}, b_j^{\circ})}{(b_j^{\circ}, b_j^{\circ})}; \quad j \neq k-1, k,$$

and

$$v_{kj} = \mu_{k-1,j}.$$

Similarly,

$$v_{k-1,j} = \mu_{kj},$$

$$v_{kj} = \mu_{k-1,j},$$

for $1 \leq j < k-1$, and

$$v_{ij} = \mu_{ij}, \quad 1 \leq j < i \leq n, \quad \{i, j\} \cap \{k-1, k\} = \emptyset.$$

These prove formula 5 and formula 6. From (3.20), we have

$$\begin{aligned} c_k^* &= c_k - \sum_{j=1}^{k-2} v_{kj} c_j^* - v_{kk-1} c_{k-1}^*, \\ &= b_{k-1} - \sum_{j=1}^{k-2} \mu_{k-1,j} c_j^* - v_{kk-1} c_{k-1}^*, \end{aligned}$$

then

$$c_k^* = b_{k-1}^* - v_{kk-1} c_{k-1}^*. \quad (3.21)$$

From the above equation, we can obtain v_{kk-1} by taking the projection of b_{k-1}^* on the orthogonal complement of $\text{Re } c_{k-1}^*$. This leads to

$$\begin{aligned} v_{kk-1} &= (b_{k-1}^*, c_{k-1}^*) / (c_{k-1}^*, c_{k-1}^*), \\ &= (b_{k-1}^*, (b_k^* + \mu_{kk-1} b_{k-1}^*)) / |c_{k-1}^*|^2. \end{aligned}$$

By using the formula 1, and b_{k-1}^* is perpendicular to b_k^* then

$$v_{kk-1} = \mu_{kk-1} |b_{k-1}^*|^2 / |c_{k-1}^*|^2.$$

This finishes the proofs for formula 2, formula 3.

Formula 4: for $i > k$,

$$v_{ik-1} = \mu_{ik-1} v_{kk-1} + \mu_{ik} |b_k^*|^2 / |c_{k-1}^*|^2,$$

and

$$v_{ik} = \mu_{ik-1} - \mu_{ik-1} \mu_{kk-1}.$$

Proof

Since, $v_{ik-1} = \frac{(c_i, c_{k-1}^*)}{(c_{k-1}^*, c_{k-1}^*)}$, and $b_i = c_i$ for $i \neq k-1, k$; then

$$v_{ik-1} = \frac{(b_i, c_{k-1}^*)}{|c_{k-1}^*|^2}.$$

From formula 1, formula 2 :

$$\begin{aligned}
 v_{ik-1} &= \frac{(b_i, (b_k^* + \mu_{kk-1} b_{k-1}^*))}{|c_{k-1}^*|^2}, \\
 &= \frac{(b_i, b_k^*)}{|c_{k-1}^*|^2} + \mu_{kk-1} \frac{(b_i, b_{k-1}^*)}{|c_{k-1}^*|^2}, \\
 &= \frac{\mu_{ik} |b_k^*|^2}{|c_{k-1}^*|^2} + \frac{\mu_{kk-1} \mu_{ik-1} |b_{k-1}^*|^2}{|c_{k-1}^*|^2}.
 \end{aligned}$$

Hence,

$$v_{ik-1} = \frac{\mu_{ik} |b_k^*|^2}{|c_{k-1}^*|^2} + \mu_{ik-1} v_{kk-1}; \quad \text{for } i > k.$$

In the same way, we can obtain the second equation such that $v_{ik} = \mu_{ik-1} - \mu_{ik} \mu_{kk-1}$; for $i > k$.

This finishes the proof.

We shall include the entire algorithm of the reduction process II in Figure 1.3. Let $E_i = |b_i^*|^2$,

4

and k be a current subscript.

The entire Algorithm for The Reduction Process II

(**)

For $i = 1$ to n do

begin

 $b_i^* = b_i$ For $j = 1$ to $i-1$ do

begin

 $\mu_{ij} = (b_i, b_j^*)/E_i$ $b_j^* = b_j^* - \mu_{ij} b_i^*$

end ;

 $E_i := (b_i^*, b_i^*)$

end ;

 $k = 2$;

(1)

Perform (*) for $l = k-1$;If $E_k < (3/4 - \mu_{kk-1}^2) E_{k-1}$, goto (2) .Perform (*) For $l = k-2, k-3, \dots, 1$.If $k = n$, Terminate . $k := k+1$.

goto (1);

(2)

$$\mu = \mu_{kk-1}$$

$$E = E_k + \mu^2 E_{k-1}$$

$$\mu_{kk1} = (\mu E_{k-1})/E$$

$$E_k = (E_{k-1} E_k)/E$$

$$E_{k-1} = E$$

$$\begin{pmatrix} b_{k-1} \\ b_k \end{pmatrix} = \begin{pmatrix} b_k \\ b_{k-1} \end{pmatrix}$$

For $i = 1, 2, \dots, k-2$ do

$$\begin{pmatrix} \mu_{k-1,j} \\ \mu_{k,j} \end{pmatrix} := \begin{pmatrix} \mu_{k,j} \\ \mu_{k-1,j} \end{pmatrix}$$

For $i = k+1, \dots, n$ do

$$\begin{pmatrix} \mu_{k-1} \\ \mu_k \end{pmatrix} := \begin{pmatrix} 1 & \mu_{kk-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -\mu \end{pmatrix} \begin{pmatrix} \mu_{k-1} \\ \mu_k \end{pmatrix}$$

If $k > 2$ Then

$$k := k - 1$$

goto (1).

(*)

If $|\mu_{kl}| > 1/2$ Then $r =$ integer nearest to μ_{kl} $b_k = b_k - rb_l$ For $j = 1, 2, \dots, l-1$ do $\mu_{kj} = \mu_{kj} - r\mu_{lj}$ $\mu_{kl} = \mu_{kl} - r$

Figure 3-1:

Remark. After (**), it is not necessary to keep track of the vectors b_i^* . It suffices to keep track of the numbers $|b_i^*|^2$, in addition to μ_{ij} and the vector b_i . Notice that $|c_k^*|^2 = (|b_{k-1}^*|^2 |b_k^*|^2) / |c_{k-1}^*|^2$ (from formula 2, 3), and $|b_k^* + \mu_{kk-1} b_{k-1}^*|^2 = |b_k^*|^2 + \mu_{kk-1}^2 |b_{k-1}^*|^2$, since b_k^* is perpendicular to b_{k-1}^*

Complexity of the algorithm

Let $B \in \mathbb{R}$ and $B \geq 2$, with the basis $b_1, \dots, b_n$ for $\mathcal{L}$ such that $|b_i|^2 \leq B$ for $1 \leq i \leq n$. Using Figure 3.1 to analyse the running time, the basic operation is the number of times passing through (1) and (2). In (2) the value of k is decreased by 1; and in (1), it is increased by 1. Initially, we have $k = 2$. $k \leq n + 1$ throughout the algorithm, therefore the number of times that we pass through (1) is at most $(n - 1)$ more than the number of times that we pass through (2).

If we are in the situation of (2), implies that $E_k < (3/4 - \mu_{kk-1}^2)E_{k-1}$, where $E_k = |b_k^*|^2$. The question is how many times we have to go through (2) to get into the situation that $E_k > (3/4 - \mu_{kk-1}^2)E_{k-1}$ for all $1 \leq k \leq n$.

In order to get an upper bound for the number of times passing through (2), we introduce the quantities

$$d_i = \det((b_j, b_l))_{1 \leq j, l \leq i} \quad (3.22)$$

Since $b_i = b_i^* + \sum_{j=1}^{i-1} \mu_{ij} b_j^*$, and $b_1^*, \dots, b_n^*$ is an orthogonal basis of $\mathbb{R}^n$, then $d_i = \det((b_j^*, b_l^*))$, for $1 \leq j, l \leq i$, and

$$d_i = \prod_{j=1}^i |b_j^*|^2 \quad \text{for } 1 \leq i \leq n. \quad (3.23)$$

Hence the d_i are positive real number; since $d(\mathcal{L}) = \prod_{j=1}^n |b_j^*|^2$ it follows that $d_n = d(\mathcal{L})^2$.

In (2), if $|b_k^* + \mu_{kk-1} b_{k-1}^*|^2 \leq 3/4 |b_{k-1}^*|^2$, then we replace b_{k-1}^* by $b_k^* + \mu_{kk-1} b_{k-1}^*$ (from Formula 1). Then the new $|b_{k-1}^*|^2$ is at most $3/4$ times the old one. From (3.23), the number d_{k-1} is reduced by a factor of at most $3/4$, whereas the other d_i , $1 \leq i \leq k-1$ are unchanged. From Formulas 2, 3, it follows that the product of norms for the new b_k^*, b_{k-1}^* is unchanged from the product

of the old ones, hence it implies that $d_i, i \geq k-1$ are unchanged. If we let

$$D = \prod_{i=1}^{n-1} d_i, \quad (3.24)$$

then the number D is changed if some d_i is changed. Hence, D is reduced by a factor at most $3/4$.

If we are in (2) with the current dimension k , after replacing $|b_{k-1}^*|^2$ with a value less than $3/4$ times $|b_{k-1}^*|^2$; and if $k > 2$, then we are in the same situation as described above, which is checked for $|b_k^* + \mu_{kk-1} b_{k-1}^*|^2 \leq 3/4 |b_{k-1}^*|^2$ for k replaced by $k-1$, $2 \leq k \leq n$. In the worst case, if we let r be the number of times passing through (2): If we start with the sequence

$$|b_1^*|^2, \dots, |b_k^*|^2,$$

where $|b_1^*|^2$ is a maximum norm of this sequence, and $|b_k^*|^2$ is a minimum norm of this sequence. We will generate the sequence of

$$(3/4)^r |b_1^*|^2, (3/4)^{r-1} |b_1^*|^2, \dots, |b_1^*|^2,$$

and $|b_1^*|^2$ is bounded by a value of d_k . The upper bound of $(3/4)^r |b_1^*|^2$ is the value of the minimum norm of the started sequence which is $|b_k^*|^2$. This sequence satisfies the condition $|b_i^* + \mu_{i,i-1} b_{i-1}^*|^2 > 3/4 |b_{i-1}^*|^2, i < k$. If let $p = (3/4)^r |b_1^*|^2$, then $p \leq |b_k^*|^2$. From (3.25),

$$p, \left(\frac{4}{3}\right)p, \left(\frac{4}{3}\right)^2 p, \dots, \left(\frac{4}{3}\right)^r p.$$

Hence, $\left(\frac{4}{3}\right)^r p \leq d_k$, then

$$r_k \leq \frac{\log \frac{d_k}{p}}{\log(4/3)}$$

$$\leq \frac{\log(d_k)}{p}$$

$$\leq \log(d_{k-1}).$$

The situation described above can happen for every $2 \leq k \leq n$. Hence, the number of times that the algorithm goes through (2) is $(\log d_1 + \dots + \log d_{n-1}) = \log \prod_{i=1}^{n-1} d_i = \log D$.

The bound $B \geq |b_i|^2$ implies that $B' \geq \prod_{j=1}^n |b_j|^2$. By (3.12), $\prod_{j=1}^n |b_j|^2 \geq d(\mathcal{L})^2 \geq d_n$; this implies that $\prod_{j=1}^n |b_j|^2 \geq d$. Hence, $B' \geq d$, and $\prod_{i=1}^{n-1} B' \geq \prod_{i=1}^{n-1} d$; then by (3.25), $D \leq B^{n(n-1)/2}$. Thus, an upper bound for the number of times passing through (2) is $\log B^{n(n-1)/2}$, which is $O(n^2 \log B)$.

In Case 1, the number of times passing through is $(n-1)$ more than Case 2. Then an upper bound for the number of times passing through (1) is $O\{(n^2 \log B) + (n-1)\}$ which is $O(n^2 \log B)$. In (1), to perform (*) we need $O(n)$ arithmetic operations (we mean an addition, subtraction, multiplication, division). We perform (*) for the values of l from $k-2$ to 1, then we need $O(n^2)$ of the arithmetic operations for (1). Hence, the overall performance for (1) is the product of $O(n^2)$ and $O(n^2 \log B)$ which is $O(n^4 \log B)$.

In (2), we need $O(n)$ to perform the μ -value then the overall performance for (2) is the product of $O(n)$ and $O(n^2 \log B)$ which is $O(n^3 \log B)$. In (**), we need $O(n^2)$ arithmetic operations.

From the above information, we can get the overall performance for the whole algorithm by summation of the time required in (**), (1), and (2); which is $O(n^2) + O(n^3 \log B) + O(n^4 \log B)$. Thus, the complexity of this algorithm is $O(n^4 \log B)$, which is bounded by a polynomial function in the length of input, (B is an upper bound of the input basis).

3.3 Implementation of The Reduction Process

The following are the implementations of *The Reduction Process I II*. Both implementations are used on the Digital Professional 850 microcomputer, and they are implemented in the *USCD p-system*. The structure of data is the following:

Figure 3-2:

The procedure *GetBasis* is the procedure to read-in an input, and also construct those input as a linked list as Figure 3.2. This procedure is using in both implementations, we only present this procedure in *Reduction Process I*. This procedure also checks for some possible errors, for e.g. input as a character instead of a number.

Some limitations for both algorithms

- input are vectors of m elements, and n vectors where m, n , can not be greater than 10.

This is because of the limitation of memory in the microcomputer. Those n vectors are linearly independent.

- b) If there occurs an error such that Stack overflow, it means that the computer has used all the memory. You should turn the computer off and turn it on again and re-input (the computer will proceed the garbage collection and also re-initialize the system). If you get an error message again this times, it means that your problem take too many iterations, and the computer's memory is too small for your problem.

- c) There is no restriction on m, n so that m can be greater than or less than n .

This page has been intentionally left blank

Implementation of The Reduction Process I

Program

Global Variable

Const ItemMax = 6;

Type ArrayBoolean = Array [1..ItemMax] of boolean;

ItemArray = array [1..ItemMax] of real;

Matrix = array [1..ItemMax, 1..ItemMax] of integer;

Item = array [1..ItemMax] of integer;

PtrToMatrix = ~~Matrix~~

ArrayOfMatrix = Array [1..ItemMax] PtrToMatrix;

ElementPointer = ListElement;

ListElement =

record

Data : real;

Next : ElementPointer;

end;

HeadPointer = HeadElement;

HeadElement =

record

Position : integer;

ThisVector : ElementPointer;

NextVector : HeadPointer;

end;

Var

OutData : text;

IsSave : ArrayBoolean;

NumElement, Dimension, Dim, IndexOfMatrix, Count, Index, AccuCount : integer;

HeadOfBasis, AnotherHead, OriginalBasis : HeadPointer;

RightBasis, IsRecursive, FirstRound, IsOkay : boolean;

Aconstant, Piconstant, Determinant : real;

StackOfMatrix : ArrayOfMatrix;

Coeffmatrix *PtrToMatrix* .

Norm *ItemArray* .

NValue *Item* .

Command *char* .

Segment Procedure *GetBasis* (Var *HeadOfBasis* *HeadPointer*; Var *IsOkay* *Boolean*

Var *Dimension*: *Integer*);

Var *I*, *Index* *Integer*;

Okay, *Negative* *boolean*;

Member *Array*; *Norm*;

Count *Integer*;

Basis, *Temp1* *HeadPointer*;

Vector, *Temp2* *ElementPointer*;

Function *ParseGroup* *boolean*.

Const *Eos* = '';

Var *S*, *Prompt* *string*;

P, *Index* *Integer*;

Procedure *ParseError* (*Msg* *Integer*);

begin

writeln(' ' P + length(*Prompt*), ' ', *Msg*);

exit(*ParseGroup*);

end; {*ParseError*}

Procedure *SkipSpaces*;

begin

while *S*[*P*] = ' ' do

P := *P* + 1;

end; {*SkipSpaces*}

Function *ParseNum* : *real*;

Var *I* : *Integer*;

N, *D* : *real*;

begin

SkipSpaces;

If not (*S*[*P*] in ['0'..'9', '.', '-']) then

ParseError ('Expecting number');

```

I := 0;
N := 0;
D := 0;
If S[P] = '-' then
begin
    Negative := true;
    P := P + 1;
end
else
    Negative = false;
    If (S[P] <> '.') then
    repeat
        N := N * 10 + ord(S[P]) - ord('0');
        P := P + 1;
    until (not (S[P] in ['0'..'9', '.'])) or (S[P] = '.');
    If (S[P] = '.') then
    begin
        P := P + 1;
        while S[P] in ['0'..'9'] do
        begin
            I := I + 1;
            D := D * 10 + ord(S[P]) - ord('0');
            P := P + 1;
        end;
    end;
    If not (S[P] in ['0'..'9', '.', '-', '+']) then
        ParseError ('Expecting number');
    N := N + (D / (powof10(I)));
    If Negative then
        ParseNum := -(N)
    else

```

ParseNum = *N*.

end. (*ParseNum*)

begin (*ParseGroup*)

Index = 1.

Prompt = 'Enter basis please

write(*Prompt*);

readln(*S*);

S = Concat(*S*, *Eos*);

ParseGroup = false;

P = 1.

SkipSpaces.

If *S*[*P*] <> *Eos* then

begin

Member[*Index*] = *ParseNum*;

SkipSpaces;

while *S*[*P*] <> *Eos* do

 If *S*[*P*] <> ' ' then

 ParseError ('Expecting " "')

 else

 begin

P := *P* + 1;

Index := *Index* + 1;

Member[*Index*] := *ParseNum*;

 SkipSpaces;

 end;

 end;

while *Index* < *NumElement* do

begin

Index := *Index* + 1;

Member[*Index*] := 0;

```

end.

ParseGroup := true.

end {ParseGroup}

begin {GetBasis}

  IsOkay := true.

  writeln('Enter your dimension, please ')
  readln(Dimension).

  writeln('Your Dimension  Dimension')
  writeln('OutData  Your dimension is  Dimension').

  writeln('Enter the maximum number of elements of the basis please ')
  readln(NumElement)

  If NumElement < Dimension then
    NumElement := Dimension.

  If (Dimension >= 1) and (Dimension <= Item.Max) then
    begin
      new(Basis).
      with Basis do
        begin
          Positive := 0;
          ThisVector := nil;
          NextVector := nil;

        end;
      HeadOfBasis := Basis;
      Temp1 := HeadOfBasis;
      Count := 0;
      repeat
        Count := Count + 1;
        Okay := ParseGroup;
        If Okay then
          begin

```

```

new(Basis);
with Basis | do
begin
    Position = Count;
    ThisVector = nil;
    NextVector = nil;
end;
Temp1 | NextVector = Basis;
Temp1 | = Temp1 | NextVector;
new(Vector);
with Vector | do
begin
    Data = Member[1];
    Next = nil;
end;
Temp1 | ThisVector := Vector;
Temp2 | = Temp1 | ThisVector;
for I := 2 to NumElement do
begin
    new(Vector);
    with Vector | do
    begin
        Data := Member[I];
        Next := nil;
    end;
end
end
else
begin
    writeln('Re_enter your data');
    IsOkay := false;
    exit(GetBasis);

```

```
        end;  
        until (Count = Dimension).  
    end  
    else  
    begin  
        writeln(' Valid dimension is in between 1 and ',ItemMax).  
        IsOkay = false.  
    end.  
end. {ietBasis}
```


Function ZeroMatrix (AMatrix : PtrToMatrix, Size : Integer) : PtrToMatrix ;

Var I, J : Integer ;

begin

For I = 1 to Size do

For J = 1 to Size do

AMatrix[I, J] := 0 ;

ZeroMatrix := AMatrix ;

end; {ZeroMatrix}

Function IsEmpty (IndexOfStack : Integer) : boolean ;

begin

IsEmpty := true

else

IsEmpty := false ;

end; {IsEmpty}

Procedure InitArray (var WhichArray : ArrayBoolean) ;

Var

I : Integer ;

begin

For I = 1 to ItemMax do

WhichArray[I] := false ;

end; {InitArray}

Procedure WriteMyResult (HeadOfBasis : HeadPointer ; Norm : ItemArray) ;

Var

Temp1 : HeadPointer ;

Temp2 : ElementPointer ;

I : Integer ;

begin

Temp1 := HeadOfBasis ;

I := 0 ;

Repeat

I := I + 1 ;

Temp1 := Temp1 ↑ .NextVector ;

Write (OutData , '(') ;

Temp2 := Temp1 ↑ .ThisVector ;

While (Temp2 ↑ .Next <> nil) do

begin

write (OutData , Temp2 ↑ .Data : 9 : 2 , ' ') ;

Temp2 : Temp2 ↑ .Next ;

end;

write (OutData , Temp2 ↑ .Data : 9 : 2 , ') ' , Norm[I] : 7 : 2) ;

writeln (OutData) ;

Until (Temp1 ↑ .NextVector = nil) ;

writeln (OutData , '*****') ;

writeln (OutData) ;

end ; {WriteMyResult}

Function NewHead (Size , NumElement **Integer**) HeadPointer ,

Var

Basis , Temp1 HeadPointer ,

Vector , Temp2 ElementPointer ,

Count , I **Integer** ,

begin

New (Basis) ,

With Basis **do**

begin

Position = 0 ,

ThisVector = nil ,

NextVector = nil ,

end ,

NewHead = Basis ,

Temp1 = Basis ;

Count = 0 ,

Repeat

Count = Count + 1 ,

New (Basis) ,

With Basis **do**

begin

Position := Count ,

ThisVector := nil ,

NextVector := nil ,

end ;

Temp1 | NextVector = Basis ;

Temp1 = Temp1 | NextVector ;

New (Vector) ,

With Vector **do**

begin

Data := 0 ;

Program FindNorm (Dim : Integer ; Var Norm : ItemArray , HeadOfBasis : HeadPointer) ,

Var

Temp1 : HeadPointer ;

Temp2 : ElementPointer ,

Dum : ItemArray ;

I , J : Integer ;

Procedure Initialize ,

Var

I : Integer ,

begin

For I = 1 to Dim do

Dum[I] = 0 ,

end ,

begin

Initialise ,

Temp1 := HeadOfBasis ,

I := 1 ,

While (Temp1.NextVector <> nil)

begin

Temp1 := Temp1.NextVector ,

Temp2 := Temp2.ThisVector ,

Dum[I] = Dum[I] + Sqr (Temp2.Data) ,

Repeat

Temp2 = Temp2.Next ,

Dum[I] = Dum[I] + Sqr(Temp2.Data) ,

Until (Temp2.Next = nil) ,

Norm[I] = Sqrt[I] ,

I = I + 1 ,

end ,

end , {FindNorm}

```

Procedure GetMinNormAndNumber (Dim Integer, Norm ItemArray,
                               HeadOfBasis HeadPointer, MinNorm real) ,
Var
    Temp1, Temp2, Temp3 HeadOfPointer ,
    Location Integer ,
begin
    Minnorm = Norm[I] ,
    Location = HeadOfBasis + NextVector + Position
    For I = 2 to Dim do
        If (Norm[I] < MinNorm) then
            begin
                Location = I ,
                MinNorm = Norm[I] ,
            end.
        Temp2 = HeadOfBasis ,
        For I = 1 to Location do ,
            begin
                Temp3 = Temp2 ,
                Temp2 = Temp2 + NextVector
                Temp1 = Temp2 + NextVector
            end.
        If (Temp2 + Position <> HeadOfBasis + NextVector + Position) then
            If (Temp3 + Position = HeadOfBasis + NextVector + Position) then
                begin
                    HeadOfBasis + NextVector = Temp2
                    Temp3 + NextVector = Temp1
                    Temp2 + NextVector = Temp3 ,
                end
            else
                begin
                    Temp3 + NextVector = HeadOfBasis + NextVector ,

```

Temp2 | *NextVector* := *Temp3* | *NextVector* | *NextVector* ;

Temp3 | *Nextvector* | *NextVector* := *Temp1* ;

HeadOfBasis | *NextVector* := *Temp2* ;

end

end. {*GetMin.Norm.AndRenumber*}

```

Procedure PushMatrix (Var AStack ArrayOfMatrix , var Top integer , Amatrix PtrToMatrix)
begin

```

```

    Top := Top + 1 ;

```

```

    AStack[Top] := Amatrix ,

```

```

end. {PushMatrix}

```

```

Procedure PopMatrix (Var AStack ArrayOfMatrix , Var Top integer Var Amatrix PtrToMatrix )
begin

```

```

    If Not IsEmpty (Top) then

```

```

        begin

```

```

            Amatrix = AStack[Top] ,

```

```

            Top = Top - 1 .

```

```

        end

```

```

        else

```

```

            begin

```

```

                writeln (OutData , '.....Stack is empty.....') ,

```

```

                Exit(Reduction) ;

```

```

            end;

```

```

end. {PopMatrix}

```

Function *NewHead* (*Size* , *NumElement* **Integer**) *HeadPointer* ,

Var

Basis , *Templ* *HeadPointer* ,

Vector , *Temp2* *ElementPointer* ,

Count , *I* **Integer** ,

begin

New (*Basis*) ,

With *Basis* **do**

begin

Position = 0 ,

ThisVector = nil ,

NextVector = nil ,

end;

NewHead = *Basis* ,

Templ = *Basis* ,

Count := 0 ,

Repeat

Count = *Count* + 1 ,

New (*Basis*) ,

With *Basis* **do**

begin

Position := *Count* ,

ThisVector := nil ,

NextVector = nil ,

end;

Templ | *NextVector* = *Basis* ,

Templ := *Templ* | *NextVector* ;

New (*Vector*) ,

With *Vector* **do**

begin

Data := 0 ;


```

        Next nil ;

    end;

    Temp1 | ThisVector := Vector ;
    Temp2 := Temp1 | ThisVector ;
    For I := 2 to NumElement do
    begin
        New (Vector) ;
        With Vector | do
        begin
            Data := 0 ;
            Next := nil ;

        end;
        Temp2 | Next := Vector ;
        Temp2 := Temp2 | Next ;
    end;
    Until (Count = Size) ;
end; {NewHead}

```

```

Function Copy (Head : HeadPointer ; Size : Integer) : HeadPointer ;
Var
    Ahead , Temp1 , Temp2 : HeadPointer ;
    Temp3, Temp4 : ElementPointer ;
begin
    Ahead : NewHead (Size , NumElement) ;
    Temp1 := Head ;
    Temp2 := Ahead ;
    While Temp1 | .NextVector <> nil do
        begin
            Temp1 := Temp1 | .NextVector ;
            Temp2 := Temp2 | .NextVector ;
            Temp3 := Temp1 | .ThisVector ;
            Temp4 := Temp2 | .ThisVector ;
            While Temp3 <> nil do
                begin
                    Temp4 | .Data := Temp3 | .Data ;
                    Temp3 := Temp3 | .Next ;
                    Temp4 := Temp4 | .Next ;
                end;
            end;
            Temp2 := Ahead ;
        end; {Copy}
    end;

```

Procedure *GetProjection* (**Var** *AnotherHead* : *HeadPointer* ; *Dim* : **Integer** ;

HeadOfBasis : *HeadPointer* , **Var** *NValue* : *Item*) ;

Var

Temp1 , *Temp2* , *Temp6* : *HeadPointer* ;

Temp3 , *Temp4* , *Temp5* : *ElementPointer* ,

Division , *InnerProduct* , *Multiplier* : **real** ;

I : **Integer** ;

begin

AnotherHead := *NewHead* (*Dim* - 1 , *NumElement*) ,

Temp6 := *AnotherHead* ;

I := 0 ;

Temp3 := *HeadOfBasis* ↑ *NextVector* ↑ *ThisVector* ;

Division := *Sqr* (*Temp3* ↑ *Data*) ;

Repeat

Temp3 := *Temp* ↑ *Next* ;

Divisor := *Divisor* + *sqr* (*Temp3* ↑ *Data*) ;

Until (*Temp3* ↑ *Next* = *nil*) ;

Temp2 := *HeadOfBasis* ↑ *NextVector* ;

Repeat

I := *I* + 1 ;

Nvalue[*I*] := 0 ;

Temp6 := *Temp* ↑ *NextVector* ;

Temp2 := *Temp2* ↑ *NextVector* ;

Temp3 := *HeadOfBasis* ↑ *NextVector* ↑ *ThisVector* ;

Temp4 := *Temp2* ↑ *ThisVector* ;

InnerProduct := *Temp4* ↑ *Data* * *Temp3* ↑ *Data* ;

Repeat

Temp3 := *Temp3* ↑ *Next* ;

Temp4 := *Temp4* ↑ *Next* ;

InnerProduct := *InnerProduct* + (*Temp4* ↑ *Data* * *Temp3* ↑ *Data*)

Until (*Temp4* ↑ *Next* := *nil*) ;

```

Multiplier := InnerProduct/Divisor ;

If IsRecursive then
    Nvalue[I] := -Round (InnerProduct/Divisor) ;
    Temp5 := Temp6 ↑ .ThisVector ;
    Temp3 := HeadOfBasis ↑ .NextVector ↑ .ThisVector ;
    Temp4 := Temp2 ↑ .ThisVector ;
    Temp5 ↑ .Data := Temp4 ↑ .Data - (Multiplier * Temp3 ↑ .Data) ;
    Temp4 ↑ .Data := Temp4 ↑ .Data + NValue[I] * Temp3 ↑ .Data ;
Repeat
    Temp3 := Temp3 ↑ .Next ;
    Temp4 := Temp4 ↑ .Next ;
    Temp5 := Temp5 ↑ .Next ;
    Temp5 := Temp4 ↑ .Data - (Temp3 ↑ .Data * Multiplier) ;
    Temp4 := Temp4 ↑ .Data + (Nvalue[I] * Temp ↑ .Data) ;
Until (Temp4 ↑ .Next = nil) ;
Until (Temp2 ↑ .NextVector = Nil) ;
end; {GetProjection}

```

```
Function NewBasis (PoppedElement : HeadPointer ; TheMatrix : PtrToMatrix , Size : Integer )
    HeadPointer ;
```

```
Var
```

```
    Temp1 , Temp4 , AnotherHead , Temp6 : HeadPointer ;
```

```
    Temp2 , Temp3 , Temp5 : ElementPointer ;
```

```
    I , J : Integer ;
```

```
begin
```

```
    AnotherHead = NewHead (Size , NumElement) ;
```

```
    Temp1 := PoppedElement ;
```

```
    Temp6 := AnotherHead ;
```

```
    For I = 1 to Size do
```

```
        begin
```

```
            Temp6 := Temp6 ↑ .NextVector ;
```

```
            Temp1 := Temp1 ↑ .NextVector ;
```

```
            Temp6 ↑ .Position := Temp1 ↑ .Position ;
```

```
            Temp4 := PoppedElement ;
```

```
            For J = 1 to Size do
```

```
                begin
```

```
                    Temp4 := Temp4 ↑ .NextVector ;
```

```
                    Temp2 := Temp4 ↑ ThisVector ;
```

```
                    Temp6 := Temp6 ↑ ThisVector ;
```

```
                    While Temp2 <> nil do
```

```
                        begin
```

```
                            Temp5 ↑ .Data := Temp5 ↑ .Data + TheMatrix ↑ [I, J] *
```

```
                            Temp2 ↑ .Data ;
```

```
                            Temp2 := Temp2 ↑ .Next ;
```

```
                            Temp5 := Temp5 ↑ .Next ;
```

```
                        end;
```

```
                    end;
```

```
                end;
```

```
            NewBasis := AnotherHead ;
```

end, {NewBasis}

```

Procedure MultiplyMatrix (Var CoeffMatrix : PtrToMatrix ,
                           TempMatrix , Amatrix , PtrToMatrix , Size : Integer) ,
Var
    I , J , K : Integer ;
begin
    CoeffMatrix := ZeroMatrix (CoeffMatrix , Size) ;
    For I := 1 to Size do
        For J = 1 to Size do
            For K := 1 to Size do
                CoeffMatrix ↑ [I, J] := CoeffMatrix ↑ [I, J] +
                    TempMatrix ↑ [I, K] * Amatrix ↑ [K, J] ;
            end ; {MultiplyMatrix}
    Procedure GetNewBasis (HeadOfBasis : HeadPointer ;
                           Var AnotherHead : HeadPointer ; Var NValue : Item ;
                           CoeffMatrix : PtrToMatrix ; Dim : Integer ; Var Index : Integer)
    Var
        TempHead : HeadPointer ;
    begin
        Index := HeadOfBasis ↑ NextVector ↑ Position ;
        TempHead := NewBasis (HeadOfBasis ↑ NextVector , CoeffMatrix , Dim - 1) ;
        HeadOfBasis ↑ NextVector ↑ NextVector := TempHead ↑ NextVector ;
        GetProjection (AnotherHead , Dim , HeadOfBasis , NValue) ;
    end ; {GetNewBasis}

```

Procedure *Checking* (**Var** *RightBasis* : **boolean** ; *PiConstant* : **real** ; *Dim* : **integer** ;
 HeadOfBasis : *HeadPointer* ; *AnotherHead* : *HeadPointer*) ;

Var

Finish : **boolean**

Temp2 : *ElementPointer* ;

Temp1 : *HeadPointer* ;

Norm1 , *Dum* : **real** ;

begin

Finish := *false* ;

Norm1 := 0 ;

Temp1 := *HeadOfBasis* ↑ *.NextVector* ;

Temp2 := *Temp1* ↑ *.ThisVector* ;

While (*Temp2* <> *nil*) **do**

begin

Norm1 := *Norm1* + (*sqr*(*Temp2* ↑ *.Data*)) ;

Temp2 := *Temp1* ↑ *.Next* ;

end;

Temp1 := *AnotherHead* ;

While (*Temp1* ↑ *.NextVector* <> *nil*) **and** (**Not** *Finish*) **do**

begin

Temp1 ↑ *.NextVector* ;

Dum := 0 ;

Temp2 := *Temp1* ↑ *.ThisVector* ;

While *Temp2* <> *nil*) **do**

begin

Dum := *Dum* + (*sqr*(*Temp2* ↑ *.Data*)) ;

Temp2 := *Temp2* ↑ *.Next* ;

end;

If (*Dum* < (*PiConstant* * *Norm1*)) **then**

begin

RightBasis := *false* ;


```

        Finish = true ;
        DisPose (AnotherHead) ,
    end
    else
        RightBasis := true ,
    end;
    If RightBasis then ;
    begin
        DisPose (AnotherHead) ,
        DisPose (HeadOfBasis) ;
    end;
end; {Checking}

```

Procedure *GetMatrix* (*CofMatrix* *PtrToMatrix*, **Var** *TempMatrix* *PtrToMatrix*
NValue *Item*, *Index*, *Size* **Integer**) ;

Var

I, *J*, *Temp* **Integer** ;

begin

New (*TempMatrix*) ;

For *I* = 1 **to** *Size* **do**

For *I* = 1 **to** *Size* **do**

If (*I* = 1) **and** (*J* = 1) **then**

TempMatrix ↑ [*I*, *J*] = 1

else

If *I* = 1 **then**

TempMatrix ↑ [*I*, *J*] = 0

else

If *J* = 1 **then**

TempMatrix ↑ [*I*, *J*] :=

NValue[*I* - 1]

else

TempMatrix ↑ [*I*, *J*] :=

CofMatrix ↑ [*I* - 1, *J* - 1] ;

If *Index* <> 1 **then**

begin

For *J* := 1 **to** *Size* **do**

begin

Temp := *TempMatrix* ↑ [*Index*, *J*] ;

TempMatrix ↑ [*Index*, *J*] := *TempMatrix* ↑ [*I*, *J*] ;

TempMatrix ↑ [*I*, *J*] := *Temp* ;

end;

For *I* := 1 **to** *Size* **do**

begin

Temp := *TempMatrix* ↑ [*I*, *Index*] ;

$TempMatrix \upharpoonright [I, Index] = TempMatrix \upharpoonright [I, 1]$.

$TempMatrix \upharpoonright [I, 1] = Temp$.

end;

end.

end. {GetMatrix}

```

Procedure SetUpMatrix (Var CoefMatrix : PtrToMatrix ; NValue : Integer ,
                      Var IsSave : ArrayBoolean , Var StackOfMatrix : ArrayOfMatrix ,
                      Var IndexOfMatrix : Integer ; RightMatrix : Boolean ,
Var
    TheMatrix , TempMatrix : PtrToMatrix ,
    I : Integer ,
begin
    GetMatrix (CoefMatrix , TempMatrix , NValue , Index , Size) ,
    If IsSave[Index] then
        begin
            PopMatrix (StackOfMatrix , IndexOfMatrix , TheMatrix) ,
            MultiplyMatrix (CoefMatrix , TempMatrix , TheMatrix , Size) ;
            IsSave[Index] := false ;
        end
    else
        CoefMatrix := TempMatrix ;
        If Not RightMatrix then
            begin
                PushMatrix (StackOfMatrix , IndexOfMatrix , CoefMatrix) ;
                IsSave[Index] := True ;
            end;
    end;
end; {SetUpMatrix}

```

```

Procedure Reposition(Var ABasis  HeadPointer , Total  Integer) ,
Var
    Temp1 , Temp2 , Temp3 , TempHead  HeadPointer ,
    I  Integer ,
begin
    I = 0 ,
    New (Temp1) ,
    With Temp1 do
    begin
        Position = 0 ,
        ThisVector := nil ,
        NextVector = nil ;
    end;
    TempHead := Temp1 ;
    Repeat
        I := I + 1 ;
        Temp3 := ABasis ;
        Temp2 := ABasis | NextVector ;
        While Temp2 | Position <> I do
        begin
            Temp3 := Temp2 ,
            Temp2 := Temp2 | NextVector ;
        end;
        Temp3 | NextVector := Temp2 | NextVector ,
        Temp1 | NextVector := Temp2 ;
        Temp1 := Temp | NextVector ;
        Temp1 | NextVector := nil ;
    Until (I = Total) ;
    ABasis := TempHead ;
end; (Reposition)

```

```

Procedure Reduce (HeadOfBasis , HeadPointer , Var CoeffMatrix , PtrToMatrix , Dim , Integer) .
Var
    MinNorm , real ,
begin
    Repeat
        If Dim = 1 then
            begin
                HeadOfBasis := AnotherHead ,
                If FirstRound then
                    begin
                        FindNorm (Dim , Norm , HeadOfBasis) ,
                        Determinant := Determinant * Norm[1] ;
                        FirstRound := false ;
                    end;
                New (CoeffMatrix) ;
                CoeffMatrix ↑ [1, 1] := 1 ;
                RightBasis := true ;
            end
        else
            begin
                If Not IsRecursive then
                    HeadOfBasis := AnotherHead
                else
                    Reposition (HeadOfBasis , Dim) ;
                IsRecursive := false ;
                FindNorm (Dim , Norm , HeadOfBasis) ;
                GetMinNormAndReNumber (Dim , Norm , HeadOfBasis , MinNorm) ;
                GetProjection (AnotherHead , Dim , HeadOfBasis , NValue) ;
                If FirstRound then
                    Determinant := Determinant * MinNorm ;
                AccuCount := AccuCount + 1 ;
            end
        end
    until FirstRound = false ;
end

```

```

Reduce (HeadOfBasis , CoefMatrix , Dim - 1) ,
If Count <= 72 then
begin
    If Determinant <> 0 then
    begin
        IsRecursive := true ;
        GetNewBasis (HeadOfBasis , AnotherHead , NValue ,
        CoefMatrix , Dim , Index) ;
        Checking (RightBasis , PiConstant , Dim , HeadOfBasis ,
        AnotherHead) ,
        SetUpMatrix (CoefMatrix , NValue , Index , Dim,
        IsSave , StackOfMatrix , IndexOfMatrix , RightBasis) ;
    end
    else
        Exit (Reduce) ;
    end
else
begin
    writeln ('...Stack overflow...') ;
    writeln ('Re_enter your constant , must be greater than 4/3') ;
    Exit (Reduction) ;
end;
end;
Until (RightBasis) ;
end; {Reduce}

```



```

OriginalBasis := Copy (AnotherHead , Dim) ,
Reduce (AnotherHead , CoefMatrix , Dim) ,
IsOkay := true ;
If (Determinant = 0) then
begin
    writeln ('your input basis is linearly dependent vectors') ,
    IsOkay := false ,
end,
If (IsOkay) then
begin
    OriginalBasis := NewBasis (OriginalBasis ,
    Coefmatrix , Dimension) ;
    writeln (OutData) ;
    writeln (OutData , '..NumElement * 2 , 'Reduce Basis ,
    '..Norm..' : NumElement * 8) ;
    writeln (OutData) ;
    FindNorm (Dimension , Norm , OriginalNorm) ;
    WriteMyResult (OriginalBasis , Norm) ;
    writeln (OutData) ;
    writeln (OutData , '..Determinant of Lattice := ,
    , Determinant : 12 : 4) ;
    writeln (OutData) ;
    writeln (OutData , ..Total number of recursive : ,
    Count) ;
    writeln (OutData) ;
    writeln (OutData , =====) ;
end;
end
else
    writeln ('..Error in your data') ;
end;
end;

```

```
writeln ('Type 'E' or 'e' to Stop or Type any character to continue' ) ;  
readln (command) ;  
Until ((Command = 'E' ) or (Command = 'e')) ,  
Page (OutData) ;  
Page (OutData) ;  
end. {Reduction}
```

Implementation of The Reduction Process II

Program

Global Variable

```

Const   ItemMax = 10,
Type    ArrayCoef = Array[1..ItemMax] of real,
        ArrayCoef = Array[1..ItemMax, 1..ItemMax] of real;
        ElementPointer = ↑ ListElement;
        ListElement =
            record
                Data real;
                Next ElementPointer;
            end;
        HeadPointer = ↑ HeadElement;
        HeadElement =
            record
                Position : integer;
                ThisVector : ElementPointer;
                NextVector : HeadPointer;
            end;
↑ Var   OutData : text;
        IsOkay : boolean;
        NumElement, Dimension, I, Count1, Count2 : integer;
        HedOfBasis, Ahead : HeadPointer;
        Command : char;
        Norm : ArrayNorm;

```

Note : Procedure GetBasis present here, see from the reduction process I

```

Procedure WriteMyResult (HeadOfBasis HeadPointer Norm ArrayNorm)
Var    Temp1 HeadPointer,
        Temp2 ElementPointer,
        I Integer
begin
    Temp1 = HeadOfBasis ^ NextVector
    I = 0;
    while Temp1 <> nil do
    begin
        I = I + 1;
        write(OutData, '(' );
        Temp2 = Temp1 ^ ThisVector
        while Temp2 <> nil do
        begin
            write(OutData, Temp2 ^ Data:9:2, ', ');
            Temp2 = Temp2 ^ Next;
        end;
        writeln(OutData, ')', ' ', Sqrt(Norm[I]):7:2);
        Temp1 = Temp1 ^ NextVector;
    end;
    writeln(OutData, '*****');
    writeln(OutData);
end; {WriteMyResult}

```

```

Function NewHead (Dimension, NumElement Integer) HeadPointer
Var      Basis Temp1 HeadPointer
        Vector Temp2 ElementPointer;
        I J Integer
begin
    new(Basis);
    with Basis ' do
    begin
        Position = 0;
        ThisVector = nil;
        NextVector = nil;
    end;
    NewHead = Basis;
    Temp1 = Basis;
    for I = 1 to Dimension do
    begin
        new(Basis);
        with Basis ' do
        begin
            Position = I;
            ThisVector = nil;
            NextVector = nil;
        end;
        Temp1 | NextVector := Basis;
        Temp1 = Temp1 | NextVector;
        new(Vector);
        with Vector ' do
        begin
            Data := 0;
            Next = nil;
        end;
    end;

```

```

Temp1 ↑ .ThisVector := Vector;
Temp2 := Temp1 ↑ .ThisVector;
for J := 2 to NumElement do
begin
    new(Vector);
    with Vector ↑ do
    begin
        Data := 0;
        Next := nil;
    end;
    Temp2 ↑ NextVector := Vector;
    Temp2 := Temp2 ↑ .NextVector;
end;
end;
end; {NewHead}

```

```

Function Copy (Head : HeadPointer, Dimension : Integer) : HeadPointer;
Var    AHead, Temp1, Temp2 : HeadPointer,
        Temp3, Temp4 : ElementPointer,
begin
    AHead := NewHead(Dimension, NumElement);
    Temp1 := Head;
    Temp2 := AHead;
    while Temp1 ≠ Nil do
        begin
            Temp1 := Temp1 → NextVector;
            Temp2 := Temp2 → NextVector;
            Temp3 := Temp3 → NextVector;
            Temp4 := Temp4 → NextVector;
            while Temp3 ≠ Nil do
                begin
                    Temp4 → Data := Temp3 → Data;
                    Temp3 := Temp3 → NextVector;
                    Temp4 := Temp4 → NextVector;
                end;
            end;
        end;
    end; {Copy}
end;

```

```

Function FindNorm (FirstVector, SecondVector, HeadPointer) real,
Var    Temp1, Temp2 : ElementPointer,
        Dum real,
begin
    Temp1 := FirstVector | ThisVector;
    Temp2 := SecondVector | ThisVector;
    Dum := 0;
    while Temp2 <> nil do
        begin
            Dum := Dum + (Temp1 | Data * Temp2 | Data);
            Temp2 := Temp2 | Next;
            Temp1 := Temp1 | Next;
        end;
    FindNorm := Dum;
end;

```



```

Procedure Subtract (FirstVector, SecondVector HeadPointer, A_Value real)
Var    Temp1, Temp2 ElementPointer;
begin
    Temp1 = FirstVector | ThisVector;
    Temp2 = SecondVector | ThisVector;
    while Temp1 <> nil do
        begin
            Temp1 | Data = Temp1 | Data - (A_Value * Temp2 | Data);
            Temp2 = Temp2 | Next;
            Temp1 = Temp1 | Next;
        end;
    end; {Subtract}

Function Search (AnyBasis HeadPointer; WhichPosition Integer) HeadPointer;
Var    Temp1 HeadPointer;
begin
    Temp1 := AnyBasis | NextVector;
    If WhichPosition > 0 then
        begin
            while (Temp1 <> nil) and (Temp1 | Position <> WhichPosition) do
                Temp1 := Temp1 | NextVector;
            Search = Temp1;
        end
    else
        Search = AnyBasis;
    end; {Search}

```

```

Procedure GetProjection (HeadOfBasis : HeadPointer, Var Norm : Array, Norm,
                        Var Coefficient : Array, Coef);
Var    Temp1, Temp2, AnotherHead, Temp3 : HeadPointer;
        I, J : Integer;
        Dummy, Dummy1 : real;

begin
    AnotherHead = Copy (HeadOfBasis, Dimension);
    Temp1 = AnotherHead ↑ NextVector;
    Temp3 = HeadOfBasis ↑ NextVector;
    for I := 1 to Dimension do
        begin
            Temp2 = AnotherHead ↑ NextVector;
            for J = 1 to I - 1 do
                begin
                    Dummy := FindNorm (Temp3, Temp2);
                    Coefficient[I, J] := Dummy / Norm[J];
                    Dummy1 := Coefficient [I, J] ;
                    Subtract (Temp1, Temp2, Dummy1);
                    Temp2 := Temp2 ↑ NextVector;
                end;
            Norm[I] := FindNorm (Temp1, Temp1);
            Temp1 := Temp1 ↑ NextVector;
            Temp3 := Temp3 ↑ NextVector;
        end;
    end; {GetProjection}

```

```

Procedure Reduce_A_Basis (HeadOfBasis HeadPointer; K_Value, L_Value Integer;
                          Var Coefficient ArrayCoe),
Var    Temp1, Temp2 HeadPointer,
        Round_Value, J Integer,
begin
    Temp1 := Search (HeadOfBasis, K_Value);
    Temp2 := Search (HeadOfBasis, L_Value);
    Round_Value = Round (Coefficient[K_Value, L_Value]);
    Subtract (Temp1, Temp2, Round_Value);
    for J = 1 to L_Value - 1 do
        Coefficient[K_Value, J] := "Coefficient[K_Value, J] -
                                   (Round_Value * Coefficient[L_Value, J]);
    Coefficient[K_Value, L_Value] = Coefficient[K_Value, L_Value]
                                   - Round_Value;
end; {Reduce_A_Basis}

```

```

Procedure Modify_A_Basis (HeadOfBasis : HeadPointer; Var Norm : ArrayNorm;
                          Var Coefficient : ArrayCoef, K_Value : Integer);

Var    Temp, Temp1, Temp2 : HeadPointer;
        Dummy, Dummy1, Dummy2 : real;
        J : Integer;
        Save : ArrayCoef;

begin
    Dummy := Coefficient[K_Value, K_Value - 1];
    Dummy1 := Norm[K_Value] + ((sqr(Dummy)) * Norm[K_Value - 1]);
    Coefficient[K_Value, K_Value - 1] := (Dummy * Norm[K_Value - 1]) / Dummy1;
    Norm[K_Value] := (Norm[K_Value - 1] * Norm[K_Value]) / Dummy1;
    Norm[K_Value - 1] := Dummy1;
    Temp := Search (HeadOfBasis, K_Value - 2);
    Temp1 := Search (HeadOfBasis, K_Value - 1);
    Temp2 := Search (HeadOfBasis, K_Value);
    Temp1 ↑ Position := K_Value;
    Temp2 ↑ Position := K_Value - 1;
    Temp ↑ NextVector := Temp2;
    Temp1 ↑ NextVector := Temp2 ↑ NextVector;
    Temp2 ↑ NextVector := Temp1;
    for J := 1 to K_Value - 2 do
        begin
            Dummy2 := Coefficient[K_Value, J];
            Coefficient[K_Value, J] := Coefficient[K_Value - 1, J];
            Coefficient[K_Value - 1, J] := Dummy2;
        end;
    for J := K_Value + 1 to Dimension do
        begin
            Sqr[J, K_Value - 1] := (Coefficient[K_Value, K_Value - 1] *
            Coefficient[J, K_Value - 1]) + (Coefficient[J, K_Value] *
            (1 - Dummy * Coefficient[K_Value, K_Value - 1]));
            Save[J, K_Value] := Coefficient[J, K_Value - 1] - (Dummy *
            Coefficient[J, K_Value])
        end;
    for J := K_Value + 1 to Dimension do

```

```
begin
    Coefficient[J, K_Value - 1] := Save[J, K_Value - 1];
    Coefficient[J, K_Value] := Save[J, K_Value],
end;
end, {Modify_A_Basis}
```

```

Procedure Reduce (HeadOfBasis : HeadPointer; Var Norm : ArrayNorm);
Var    Finished, Okay : boolean;
        K_Value, L_Value, L : integer;
        Coefficient : ArrayCoef;
        Check_Value : real;

begin

    GetProjection (HeadOfBasis, Norm, Coefficient);
    K_Value := 2;
    Count1 := 0;
    Count2 := 0;
    Finished := false;
    while not Finished do
        begin
            L_Value := K_Value - 1;
            if Abs (Coefficient[K_Value, L_Value]) > 0.5 then
                Reduce_A_Basis (HeadOfBasis, K_Value, L_Value, Coefficient);
                Check_Value := (0.75 - Sqr(Coefficient[K_Value, K_Value - 1]))
                    * Norm[K_Value - 1];
                if Norm[K_Value] < Check_Value then
                    begin
                        Modify_A_Basis (HeadOfBasis, Norm, Coefficient, K_Value);
                        if K_Value > 2 then
                            begin
                                Count1 := Count1 + 1;
                                K_Value := K_Value - 1;
                            end;
                        Okay := false;
                    end
                end
            else
                Okay := true;
            if Okay then
                begin
                    if K_Value > 2 then
                        for L := K_Value - 2 downto 1 do
                            if Abs (Coefficient[K_Value, L]) > 0.5 then

```

```

                                Reduce_A_Basis (HeadOfBasis,
                                                K_Value, L, Coefficient);
If K_Value = Dimension then
    Finished := true
else
begin
    K_Value := K_Value + 1;
    Count2 := Count2 + 1;
    Finished := false;
end;
end
else
    Finished := false;
end;
end; {Reduce}

```

{Main Program}

begin

 writeln('Valid dimension is in between 1 and ', *ItemMax*);

 rewrite(*OutData*, 'Printer: '), ' '

repeat

 page(*OutData*);

GetBasis (*HeadOfBasis*, *IsOkay*, *Dimension*);

if *IsOkay* **then**

begin

 writeln(*OutData*, ' ', *Dimension* * 2, 'Input basis ', ' ',

 '..Norm..': *Dimension* * 8);

 writeln(*OutData*);

AHead := *HeadOfBasis* ↑ *NextVector*;

for *I* := 1 **to** *Dimension* **do**

begin

Norm[*I*] := *FindNorm* (*AHead*, *AHead*);

AHead := *AHead* ↑ *NextVector*;

end;

WriteMyResult (*HeadOfBasis*, *Norm*);

Reduce (*HeadOfBasis*, *Norm*);

 writeln(*OutData*, ' ', *Dimension* * 2, 'Reduced Basis ', ' ',

 '..Norm..': *Dimension* * 8);

 writeln(*OutData*);

AHead := *HeadOfBasis* ↑ *NextVector*;

for *I* := 1 **to** *Dimension* **do**

begin

Norm[*I*] := *FindNorm* (*AHead*, *AHead*);

AHead := *AHead* ↑ *NextVector*;

end;

WriteMyResult (*HeadOfBasis*, *Norm*);

 writeln(*OutData*);


```

        writeln(OutData,'Count1 = ',Count1,' Count2 = ',Count2);
    end
    else
        writeln('Error in input basis');
        writeln('Type "E" to stop or type any character to continue');
        readln(Command);
    until ((Command = 'E') or (Command = 'e')),
        page (OutData),
        page (OutData),
end. {Reduction}

```

Chapter 4

Some Interesting Algorithms

Besides the algorithm in Chapter 2, there are 2 more algorithms which can, in polynomial time, solve the ILP problems in special cases. We will present these two algorithms in Section 4.2 and Section 4.3. In fact, the algorithm by KANNAN [9] is similar to the one by HIRSCHBERG and WONG [8], since KANNAN's algorithm breaks the ILP problem into at most n -knapsack problems. Both algorithms solve the ILP problems with only 2 variables. Unlike the HIRSCHBERG and WONG's algorithm, KANNAN's algorithm is a recursive one. Both algorithms can solve the ILP problems in their special cases within time bounded by a polynomial function of the length of input.

In Section 4.1, we will show that the algorithm in Chapter 2 can also solve the ILP problems with a fixed value of m in a polynomial time, where m is the number of constraint equations.

4.1 A polynomial time algorithm for the integer linear programming problem

with a fixed number of constraints.

This algorithm was noted by P. Van Emde Boas as an immediate consequence of the main result from the algorithm in Chapter 2. The algorithm is as follows.

Let the ILP problem be as the one in Chapter 2, we have to decide whether there exists a vector $x \in Z^n$ for which $Ax \leq b$, where A is $(m \times n)$ -matrix, b is a m -vector. By repeated application

of the Euclidean algorithm (as shown in Chapter 2), we can find an $(n \times n)$ - matrix U with integral coefficients and determinant ± 1 such that the matrix

$$AU = \{a'_{ij}\}_{1 \leq i \leq m, 1 \leq j \leq n}$$

is a lower triangle matrix with

$$a'_{ij} = 0 \quad \text{for } j > i. \quad (4.1)$$

Let $y = U^{-1}x$; we can see that the existence of $x \in Z^n$ with $Ax \leq b$ is equivalent to the existence of $y \in Z^n$ with

$$(AU)y \leq b \quad (\text{from } x = Uy).$$

If $n > m$, then the coordinates $y_{m+1}, \dots, y_n$ of y do not occur in these inequalities, since (4.1) implies that $a'_{ij} = 0$ for $j > m$. Hence, the original problem can be reduced to a problem with only $\min\{n, m\}$ variables. In both cases, we can use the algorithm in Chapter 2 to solve them. It follows from the main result of Chapter 2, we can conclude that any ILP problem with a fixed value of m is a polynomially solvable problem.

4.2 A polynomial time algorithm for the knapsack problem with two variables

The knapsack problem considered here is the following,

$$\begin{aligned} & \text{maximize} \quad \sum_{i=1}^n c_i x_i \\ & \text{subject to} \quad \sum_{i=1}^n a_i x_i \leq b \end{aligned}$$

and x_i non-negative integers,

where a_1, c_1, b are positive numbers. In general, we consider the knapsack problems as the ILP problems with any 1 constraint.

In [8], HIRSCHBERG and WONG give an algorithm to solve the knapsack problems with only 2 variables. This algorithm runs in polynomial time of length of the input. Thus, our problem is

Problem 1

$$\text{maximize } c_1 x_1 + c_2 x_2$$

$$\text{subject to } a_1 x_1 + a_2 x_2 \leq b$$

and x_1, x_2 non-negative integers.

Note that the naive approach (test $x_1 = i$, $x_2 = \lfloor (b - a_1 i) / a_2 \rfloor$ as i take on integer value from 0 to $\lfloor b/a_1 \rfloor$) take time proportional to b/a_1 which is exponential in the length of the input.

This algorithm originates from a well-known continued fraction method. We first transform *Problem 1* into *Problem 2* such that

Problem 2

$$\text{maximize } x_1 + k_1 x_2$$

$$\text{subject to } x_1 + k_2 x_2 \leq k_3$$

x_1, x_2 non-negative integers

$$\text{and } k_1 = c_2/c_1, k_2 = a_2/a_1, k_3 = b/a_1$$

(without loss of generality, we can assume that $k_2 > k_1$).

The algorithm generates an integer sequence b_i ($0 \leq i \leq k$), such that the last b value doesn't exceed the value of k_3/k_2 . We consider the value of b_i such that

a) If $k_2 > k_1$ then one of b_0 or b_k is a solution for the x_2 -value.

b) If $k_2 = k_1$ then the last b -value (b_k) is a solution for the x_2 -value.

From this stage, we can generate x_1 from x_2 such that $x_1 = \lfloor k_3 - k_2 x_2 \rfloor$.

HERSCHBERG and WONG also show that this algorithm runs in polynomial time of the input length

4.3 A polynomial algorithm for the two-variable integer programming problem

We consider the following integer programming problem :

Problem 1

$$\text{maximize } c_1 x_1 + c_2 x_2$$

$$\text{subject to } a_{i1} x_1 + a_{i2} x_2 \leq b_i \quad i = 1, 2, \dots, n$$

and x_1, x_2 non-negative integers

where a_{ij}, c_j and b_i are assumed to be non-negative integers. We can obtain the solution of *Problem 1* from the solutions to at most n problems, each of which is of the form:

Problem 2

$$\text{maximize } c_1 x_1 + c_2 x_2$$

$$\text{subject to } a_1 x_1 + a_2 x_2 \leq b$$

$$L \leq x_1 \leq U$$

and x_1, x_2 non-negative integers

where all the constants involved are positive integers, $U \leq \lfloor b/a_1 \rfloor$. If any one of the following three conditions is met, then *Problem 2* is obviously trivially solved.

a) c_1 (or c_2) ≤ 0 : x_1 (or x_2) can be set to its lowest permissible value and the resulting one-variable problem solved.

b) a_1 or $a_2 = 0$, or $b \leq 0$

c) $L = U$

If none of these three conditions is satisfied, the algorithm by KANAN [9] successively reduces *Problem 2* to other problems of the same form until one of the conditions a), b) or c) is met. We can assume, without loss of generality, that $a_1 U \leq b$. If not replace U by $\lfloor b/a_1 \rfloor$. This algorithm can be performed recursively until the new problem with one of those three conditions can be constructed.

Thus, to solve *Problem 1*, we only need to find the maximum of $c_1 x_1 + c_2 x_2$ over the integer points in each set of the at most n new problems, and then take the maximum among all these maxima. KANNAN shows that the problem can be solved within a time bounded by a polynomial function of the length of the input.

REFERENCES

- [1] J.W.S. Cassels. *An introduction to the geometry of numbers*. Springer. Berlin 1959
- [2] Katta G. Murty. *Linear and Combinatorial programming*. John Wiley & Sons, INC Toronto 1976
- [3] Christos H. Papadimitriou. *Combinatorial optimization Algorithms and Complexity*. Prentice-Hall INC, 1982
- [4] J. von zur Gathen, M. Sieveking. *Linear integer inequalities are NP-Complete*. unpublished manuscript
- [5] I. Borosh, L. B. Treybig. *Bounds on positive integral solutions of linear diophantine equations*. Proc Amer Math Soc 55 (1976), pp 299-344.
- [6] M. R. Garey, D. S. Johnson. *Computers and intractability a guide of the theory of NP-Completeness*. Freeman & Co., San Francisco 1979.
- [7] M. Grotschel, L. Lovass, A. Schrijver. *The ellipsoid method and its consequences in combinatorial optimization*. Combinatoria 1, (1981)
- [8] D. S. Hirschberg, C.K. Wong. *A polynomial-time algorithm for the knapsack problem with two variables*. J Assoc. Comput March 23 (1976), pp 147-154.
- [9] R. Kannan. *A polynomial algorithm for the two-variable integer programming problem*. J. Assoc. Comput march 27 (1980), pp 118-122.
- [10] L.G. Khachiyan. *A polynomial algorithm in linear programming*. Dokl. Akad. Nauk SSSR 244 (1979), pp 1093-1096

- [11] Vasek Chavtal. *Linear programming* , W. H. Freeman and Comany 1983.
- [12] H. W. Lenstra, Jr. *Integer linear programming with a fixed number of variables* , Math oper.

Res

- [13] A.K.Lenstra, H.W.Lenstra, L.Lavass . *Factoring polynomials with rational coefficients* ,
Mathematische Annalen 261 (1982),pp 513-534.

END

1.5104.86

FIN