AN ALGORITHM FOR THE SOLUTION OF ZERO-ONE RESOURCE ALLOCATION PROBLEMS

by

GOLAM MOWLA, M. Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

September, 1975

© GOLAM MOWLA 1976

AN ALGORITHM FOR RESOURCE ALLOCATION PROBLEMS

MASTER OF SCIENCE (1975)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: An Algorithm for the Solution of Zero-One Resource Allocation Problems

AUTHOR: Golam Mowla, M.Sc. (Karachi University)

SUPERVISOR: Professor P. C. Chakravarti

NUMBER OF PAGES: v, 85

ABSTRACT

An algorithm is developed for discrete optimization of zero-one resource allocation problems. A single constraint problem is first formulated in dynamic programming. This formulation then undergoes a number of modifications to develop the algorithm. This algorithm leads to a significant reduction in computational requirements as compared to the dynamic programming method. Three theorems and several lemmas are proved which are central in making the algorithm efficient. Different relevant features are included in the study to extend the algorithm to solve problems with more than one constraint.

ACKNOWLEDGEMENT

I would like to thank my supervisor, Dr. P. C. Chakravarti, for his proper guidance and superivsion during the course of this work and in the preparation of this thesis.

I would also like to thank Ken Chapman for his excellent typing, and Hasan Murshed for his invaluable aid in proofreading this thesis.

TABLE OF CONTENTS

1. INTRODUCTION 1
The Cutting Plane Algorithm of Gomory 2
The Land-Doig Algorithm 5
Dakin's Algorithm 9
The Zero-One Algorithm of Balas 12
Generalized Lagrange Multiplier (GLM) Method of Everett 18
Kaplan's extension of Everett's GLM method 19

2. A REVIEW OF DYNAMIC PROGRAMMING THEORY 22
2.1 Multistage analysis 23
2.2 Multistage decision system 26
2.3 Development of the Recursive Equations 29
2.4 Characteristics of the Dynamic Recursive Process and the Principle of Optimality 32

3. THE SOLUTION OF THE ZERO-ONE SINGLE CONSTRAINT INTEGER PROGRAMMING PROBLEM BY USING DYNAMIC PROGRAMMING 34
3.1 Basic Dynamic Programming Formulation 35
3.2 Computational Scheme in Dynamic Programming 36
3.3 Step by Step Procedure for Computation Using Dynamic Programming 38

4. A REFINED ALGORITHM FOR THE SOLUTION OF THE ZERO-ONE SINGLE CONSTRAINT INTEGER PROGRAMMING PROBLEM 41
4.1 Lemma 1 43
4.2 Lemma 2 44
4.3 Lemma 3 45
4.4 Lemma 4 46
4.5 Theorem 1 47
4.6 Definition of U_i^m 49
4.7 Theorem 2 50
4.8 Theorem 3 52
4.9 Discussion of Lemmas and Theorems 53
4.10 The Refined Algorithm 55
4.11 Step by Step Procedure for the Solution of Problem (4.1) by using the Refined Algorithm 57
4.12 Numerical example

5. AN APPLICATION OF THE REFINED ALGORITHM TO SOLVE LORIE-SAVAGE TYPE PROBLEMS WITH EQUALITY CONSTRAINTS 66
5.1 Reduction in number of variables 69
5.2 Solution of the Lorie-Savage Problem 69

6. THE REFINED ALGORITHM AND THE LAGRANGE MULTIPLIER TECHNIQUE FOR REDUCING DIMENSIONALITY 73
6.1 The solution scheme for a two constraint problem 74
6.2 A numerical example 76

7. CONCLUSIONS 80

BIBLIOGRAPHY 82

CHAPTER I

INTRODUCTION

There are many real life situations in which a class of linear programming problems are restricted to have integer solutions for the variables as well as the objective functions. These are called all-integer problems. An important type of this class of problems is the one in which the variables are restricted to the values zero or one only. Mathematically, we write

$$\left.\begin{array}{lll} \text{maximize } Z_0 = \sum_{i=1}^{n} c_i x_i & & \\ \text{subject to } \sum_{i=1}^{n} l_{ij} x_i \leq b_j & & j = 1, \ldots, m \\ \qquad x_i = 0, 1 & & i = 1, \ldots, n \end{array}\right\} \quad (1)$$

These problems are known as zero-one integer programming problems. They arise in real life situations in which several activities are competing for limited resources. Typically they are capital budgeting problem, knapsack problem, travelling salesman problem, etc.

In view of the importance of the problem defined by (1), several methods for its solution have been put forward in [2], [7], [10], [13], [17], [19], [20], [22], [30], and [34] by the researches in the field of optimization. These methods can be divided into two classes: (i) those which are independent of the Lagrange multiplier technique; and (ii) those which are based on the Lagrange multiplier technique. Each class of methods can be divided into two subclasses:

(a) those applicable to the solution of integer programming problems (including zero-one integer problems) in general; and (b) those applicable to the solution of zero-one integer programs only. The better known methods of class (i) (Taha, 1971) are given in Gomory [19], [20], Land and Doig [34], Dakin [7], and Balas [2]. Of these, [19], [20], [34], [7] belong to subclass (a), and [2] belongs to subclass (b). The methods in class (ii) are given in [10] and [30] of which [10] belongs to subclass (a) and [30] to subclass (b).

The Cutting Plane Algorithm of Gomory

Dantzig [8] suggested the cutting plane approach for solving integer programs. Gomory ([19] and [20]) developed Dantzig's approach into a systematic algorithm for the solution of both integer and mixed problems. In order to apply this algorithm to zero-one integer problems, one has to add a constraint $x_i \leq 1$ for each x_i $(i = 1, \ldots, n)$.

The algorithm makes use of the dual simplex method. The important aspect of the algorithm is that it constructs secondary constraints called the Gomory Constraints. These constraints, when added to the optimal non-integer solution, will effectively cut the solution space toward the required result. A basic requirement for this algorithm is that all the coefficients and the right hand side constant of each constraint must be in integer form.

The algorithm is carried out in the following way. First the problem is solved as a regular linear programming problem disregarding

the integrality conditions. If the optimum solution happens to be all integers, the goal is achieved. Otherwise Gomory constraints which will force the solution toward the integer point are developed as follows.

Let x_i ($i = 1, \ldots, n$) be the basic variables, and s_j be the nonbasic variables in the optimal simplex tableau. Let the value of x_i as obtained from the optimal simplex tableau for the noninteger solution be given by

$$x_i = d_i - \sum_{j=1}^{m} e_i^j s_j \tag{2}$$

where d_i are non-integer, and e_i^j are the coefficients of the nonbasic variables for the jth constraint.

Let $d_i = [d_i] + f_i$

$$e_i^j = [e_i^j] + f_{ij}$$

where $[d_i]$ and $[e_i^j]$ are the largest integers contained in d_i and e_i^j respectively. It follows that $0 < f_i < 1$ and $0 \leq f_{ij} < 1$. Substituting for d_i and e_i^j in (2), we have

$$x_i = [d_i] + f_i - \sum_{j=1}^{m} ([e_i^j] + f_{ij}) s_j$$

or

$$f_i - \sum_{j=1}^{m} f_{ij} s_j = x_i - [d_i] + \sum_{j=1}^{m} [e_i^j] s_j \tag{3}$$

Now, for all the variables x_i and s_j to be integer valued, the right hand side of (3) must be an integer. This suggests that the left hand side of (3) must also be an integer. Since $0 < f_i < 1$, and

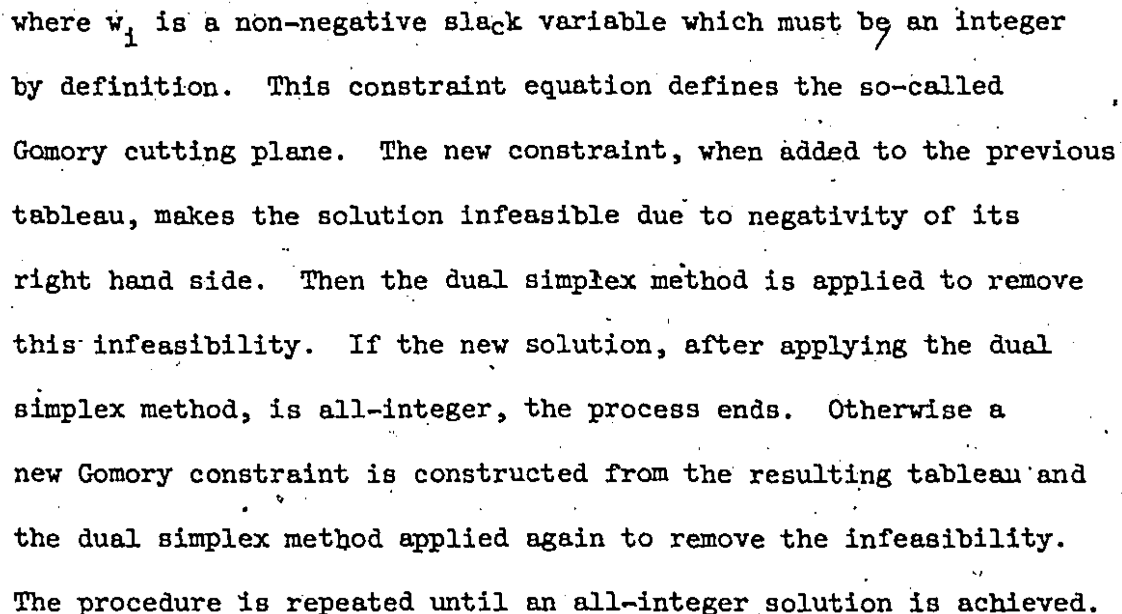
$\sum_{j=1}^{m} f_{ij}s_j \geqslant 0$, it follows that a necessary condition is

$$f_i - \sum_{j=1}^{m} f_{ij}s_j \leqslant 0 \qquad (4)$$

This is true since $f_i - \sum_{j=1}^{m} f_{ij}s_j < f_i < 1$. But since $f_i - \sum_{j=1}^{m} f_{ij}s_j$ is an integer, it can either be zero or a negative integer. The relation (4) represents the so-called Gomory Constraint.

The new constraint, (4), is put at the bottom of the tableau in the form

$$w_i = \sum_{j=1}^{m} f_{ij}s_j - f_i \qquad (5)$$



where w_i is a non-negative slack variable which must be an integer by definition. This constraint equation defines the so-called Gomory cutting plane. The new constraint, when added to the previous tableau, makes the solution infeasible due to negativity of its right hand side. Then the dual simplex method is applied to remove this infeasibility. If the new solution, after applying the dual simplex method, is all-integer, the process ends. Otherwise a new Gomory constraint is constructed from the resulting tableau and the dual simplex method applied again to remove the infeasibility. The procedure is repeated until an all-integer solution is achieved.

The Land-Doig Algorithm

Land and Doig [34] developed an algorithm for solving integer programming problems. The Land-Doig algorithm is carried out by successively making parallel shifts in the objective hyperplane toward the interior of the solution space such that each new shift will generate an integer value of at least one variable. These shifts are made in an orderly manner by successively applying an explore-label-and-augment procedure so that a superior integer point in the solution space is never by-passed. Thus, let Z^1, Z^2, ... Z^k represent the values of Z_0 corresponding to the first, second, ... and k^{th} shifts in the objective hyperplane. The optimum solution is reached at the kth shift if, for the first time, all the variables assume integer values.

The procedure advanced by Land and Doig is essentially enumerative and starts by finding the solution to the problem neglecting the integrality condition. If an all-integer solution is achieved, the process ends. Otherwise, let Z^0 be the corresponding value of the objective function. A variable, x_p, is selected for integrality consideration and let x_p^* be its optimal non-integer value corresponding to Z^0

Let Z^1 specify the first shift in the objective hyperplane, and $[x_p^*]$ be the largest integer value included in x_p^*. Let Z_p and $Z_{p'}$ denote the optimal values of Z_0 corresponding to the linear programming problem subject to the additional constraints $x_p = [x_p^*]$ and

$x_p = [x_p^*] + 1$ respectively.

The determination of Z^1 can be achieved by using the concept of a decision tree. The first node in the tree is represented by Z^0. Two branches corresponding to $x_p = [x_p^*]$ and $x_p = [x_p^*] + 1$ emanate from this node (see figure 1).

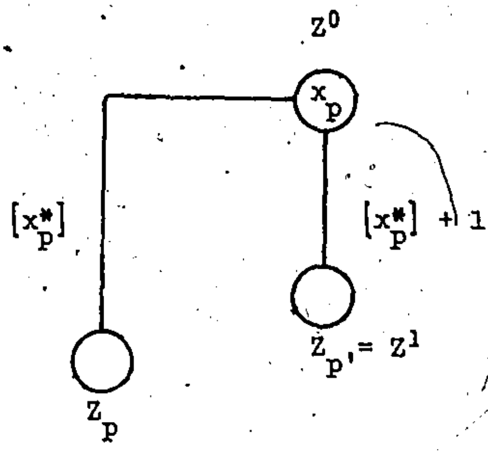


Figure 1

The generation of these two branches from a node is termed exploration. The end nodes of these two branches will be identified with Z_p and $Z_{p'}$ respectively. This yields $Z^1 = \max \{Z_p, Z_{p'}\}$. It is now said that the nodes associated with Z^0 and Z^1 are labelled. Figure 1 illustrates the case where $Z^1 = Z_{p'}$. In general, a node is labelled if it defines the next shift in the objective hyperplane.

It is to be noted that the highest node in the tree represents the largest value of the objective function Z_0. Every node will be associated with a variable. The node Z^0 is reserved for the variable x_p. It is possible, however, that more than one node may be

associated with the same variable. If, at the node Z^1, the solution is all-integer, the process ends. Otherwise, an augmentation procedure is applied, after Z^1 has been labelled, to generate a new branch from the node Z^0 from which Z^1 originated. If $x_p = [x_p^*] + 1 = v$ gives Z^1, then augmentation is done with $x_p = v + 1$ (see figure 2). Otherwise, if $x_p = [x_p^*] = v$ gives Z^1, then augmentation is done with $x_p = v - 1$. Figure 2 illustrates the case where Z^1 is given by

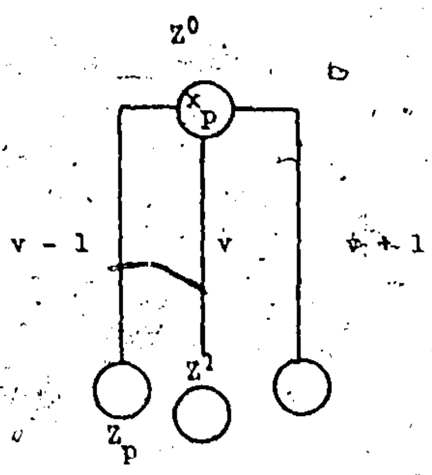


Figure 2

$x_p = [x_p^*] + 1 = v$. In the augmentation procedure, the value of Z_0 due to the augmentation branch is noted as a node.

Then, for the next shift another variable, say x_q, is taken for integrality consideration. The node Z^1 represents x_q. Let x_q^* by the optimal non-integer value of x_q corresponding to Z^1. Two branches are then drawn from Z^1 with $x_q = [x_q^*]$ and $x_q = [x_q^*] + 1$, and

the corresponding values of Z_0, namely Z_q and $Z_{q'}$, are represented as nodes. Then labelling is done by selecting Z^2 from amongst all the unlabelled nodes as the one having the largest value of Z_0 (see figure 3). If at Z^2 the solution is all-integer, the process ends. Otherwise, the augmentation procedure is performed to generate a new branch from the node from which Z^2 originated.

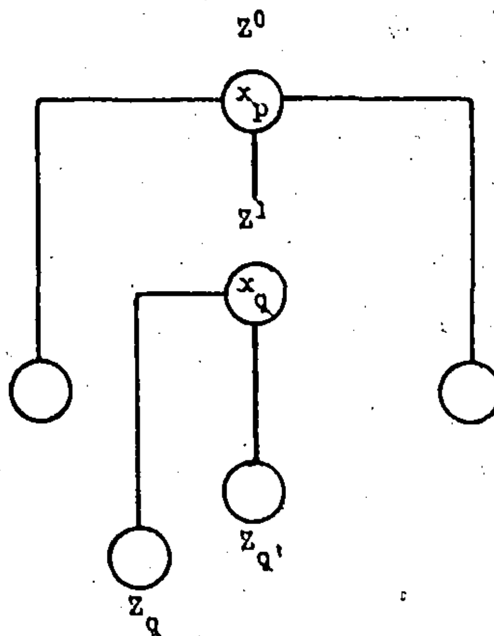


Figure 3

This is followed by an exploration for a third variable for integrality consideration, provided Z^2 originated from the node associated with x_q. Otherwise, if Z^2 originated from a node associated with x_p, then, after augmentation, x_q is reconsidered for the integrality condition.

The procedures of exploration, augmentation, and labelling

are continued until an all-integer solution is obtained.

It must be noted that, in applying either the augmentation or the exploration procedure, the new branches may give rise to infeasible solutions. In those cases, such branches must be discarded. Furthermore, for future considerations, any branch originating from the same node which corresponds to any integer value beyond the ones proved infeasible will also yield an infeasible solution.

The main drawback of the algorithm is that, as one goes down the tree, the number of unlabelled nodes increases enormously. If the problem has a large number of variables, even with a single constraint, the solution by this method becomes cumbersome, sometimes even impossible.

Dakin's Algorithm

Dakin [7] introduced an algorithm to solve integer programming problems which is a modified version of the Land-Doig algorithm. The algorithm guarantees that at each node there will be exactly two branches. In the Land-Doig algorithm, the variables are forced to take exact integral values. Dakin, on the other hand, suggested that suitably chosen bounds can be used to cover the entire range for each of the variables.

The algorithm may, in short, be described as follows: at any iteration t, there is an available lower bound, Z_0^t, of the objective

function for an all-integer solution. In addition to the lower bound, there is also a list of linear programming problems. At iteration 1, the list contains only the original problem disregarding integrality restrictions. Let us call it problem P_0.

If the solution to P_0 is all-integer, the process ends. Otherwise, let $Z_0{}^1 = 0$. A variable x_p, having a non-integer solution $x_p = x_p^*$, is arbitrarily chosen for integrality consideration. The problem P_0 is then replaced by two problems, each having one of the constraints

$$x_p \leq [x_p^*]$$

$$\text{and } x_p \geq [x_p^*] + 1$$

added to problem P_0. Let these two problems be called problem P_1 and problem P_2. By considering these two new problems, one is actually taking into account all the possible integer values that may be generated for the variable x_p. If both of the problems give all-integer solutions, the process ends with the best solution as optimum. Otherwise, further problems are generated by considering a non-integer variable for integrality. These problems are added to the list and are considered one at a time.

In general, at any iteration t, we take out a problem from the current list of problems, and this is then solved. If it has an infeasible solution, then it is discarded. If the problem yields a solution such that $Z_0 \leq Z_0{}^t$, then it is also discarded. In both these cases, we set $Z_0{}^{t+1} = Z_0{}^t$, then we do the following.

(i) If it is an all-integer solution, then we keep a record of the solution and set $Z_0^{t+1} = Z_0$.

(ii) Otherwise, we arbitrarily choose a variable which has a non-integer value for integrality consideration. This adds two new problems to the list of problems, each containing an additional constraint, as described above, added to the present problem.

The process continues until the list of problems is empty. At termination, if a feasible solution yielding Z_0^t is recorded, it is optimal; otherwise no feasible solution exists.

The method is applicable to zero-one integer programming problems. In that case, the new problems are generated with the constraints $x_i = 0$ and $x_i = 1$ for all i. The equality restriction for the variables resembles that in the Land-Doig algorithm.

One essential difference between this method and the Land-Doig algorithm is that it does not need the augmentation procedure at every node. Also, the Land-Doig algorithm does not discard any unlabelled node, whereas the special features of this algorithm often make it possible to avoid further branch generation.

The method works well in problems with a few variables, but if the number of variables is large, or if the solution to problem P_0 is far from the optimal integer solution, then the number of problems generated may be too large for a practical application of the algorithm.

The Zero-One Algorithm of Balas

Balas [2] has developed an additive algorithm specifically for solving zero-one integer programming problems. His algorithm is enumerative in nature and starts by setting all n variables equal to zero. It then successively assigns the value one to certain variables in such a way that, after trying a part of all the 2^n possible combinations, one obtains either an optimal solution or evidence of the fact that there exists no feasible solution. The scheme remained cumbersome until Glover [17] introduced the idea of backtracking, which was later implemented by Geoffrion [13].

To carry out the Balas method, the problem is converted into the minimization type with all the coefficients in the return function as positive. The conversion is made by simply substituting $(1 - x_I)$ for all those x_I's which have negative coefficients in the return function. The starting solution for Balas' algorithm is the same as that for the dual simplex method, yielding

$$S_j{}^0 = b_j \text{ and } x_i = 0 \text{ for all i, j}$$

where $S_j{}^0$ are the initial values of the slack variables.

The algorithm is carried out in three steps. Before describing the steps, we introduce the following notations:

at any iteration t, let

N = set of subscripts for all x_i variables

I_t = set of subscripts of all the x_i variables assigned

a binary value. Elements in I_t constitute a partial solution at the tth iteration

S_j^t = value of the slack variable S_j of the jth constraint at iteration t.

$I_0 = \phi$ by definition

$N - I_t$ = set of subscripts of all free variables not included in the partial solution

N_t = set of subscripts of the x_i variables selected from $N - I_t$ which are candidates for improving the solution

Z_{min} = minimum value of the objective function out of all feasible solutions obtained so far

$Z_{min} = \infty$, initially.

Step 1: determination of the entering variable

Given N and I_t, the entering variable is determined from $N - I_t$ in two phases:

(i) determining the set N_t of the subscripts of the x_i variables supposed to improve the solutinn, and

(ii) possibility of obtaining a feasible solution ($S_j \geq 0$) if all the variables whose subscripts are in N_t were assigned the value 1.

At phase (i) we determine N_t. This is carried out by performing two tests, (ia) and (ib).

(ia) For any free variable x_p such that $p \in (N - I_t)$, if

$l_{jp} \geq 0$ for all j for which $S_j < 0$, the new values of S_j will be given by

$$S_j^{t+1} = S_j^{t} - l_{jp}, \quad l_{jp} \geq 0, \; S_j^{t} < 0$$

after x_p has been set equal to one. This does not force any negative S_j toward the feasible space. Then, denoting the corresponding set of indices for such variables as G_t, any variable x_p such that $p \in G_t$ should be excluded as a non-promising variable.

(ib) Given Z_{min} as defined above, let

$$Z_t = \sum_{i \in I_t} c_i x_i$$

be the current value (infeasible) of the return function. A free variable $x_{i'}$ such that $i' \in (N - I_t)$ cannot improve the solution if, by adding it (i.e., setting it equal to 1), the resulting new value of the return function is greater than or equal to Z_{min}. This means that free variables $x_{i'}$, for which the inequality

$$c_{i'} + Z_t \geq Z_{min}$$

is satisfied, should not be considered for entering the solution. Denoting by H_t the set of subscripts of such free variables, then the set N_t of the subscripts of the variables improving the solution is given by

$$N_t = N - I_t - (G_t \cup H_t)$$

If $N_t = \phi$, then partial solution I_t has no better feasible completion. I_t is said to be fathomed in this case. Then backtracking

is needed. Otherwise we proceed to phase (ii).

Phase (ii) is also carried out by performing two tests.

(iia) Let us consider any constraint j:

$$\sum_{i=1}^{n} l_{ji} x_i + S_j = b_j$$

If for any $S_j^t < 0$, the condition

$$\sum_{i \in N_t} l_{ji} > S_j^t$$

$$l_{ji} < 0$$

is satisfied, then N_t should be abandoned because all such variables whose subscripts are in N_t cannot bring feasibility to the solution. This is again equivalent to having $N_t = \phi$ so that I_t is fathomed and backtracking is needed. Otherwise, (iib) is checked.

(iib) Let us define

$$v_i^t = \sum_{\text{all } j} \min (0, S_j^t - l_{ji})$$

We compute v_i^t for all $i \in N_t$. The quantity v_i^t gives a measure of the total infeasibility in S_j^{t+1} after x_i is set to 1. The entering variable is then selected as x_k such that

$$v_k^t = \max_{i \in N_t} \{v_i^t\}$$

The new partial solution I_{t+1} is now obtained by augmenting I_t by {k} so that

$$I_{t+1} = I_t \cup \{+k\}$$

$$= \{I_t, +k\}$$

The next step at this point is to branch to step 2 for the determination of the new values for S_j, i.e., S_j^{t+1} and Z_{min}.

Step 2: determination of new solution

Given I_{t+1} as determined from step 1, then

$$S_j^{t+1} = S_j^{t} - 1_{jk}$$

and $Z_{t+1} = Z_t + c_k$

Now

(2a) if $S_j^{t+1} \geq 0$, for all j, then we set

$$Z_{min} = Z_{t+1}$$

This means I_{t+1} is fathomed and backtracking is needed.

(2b) if any $S_j^{t+1} < 0$, return is made to step 1 for a new augmentation of I_{t+1}.

Step 3: Backtracking and determination of optimal solution

Whenever a partial solution I_t is fathomed, backtracking is required. The procedure of backtracking will terminate only after all 2^n solutions have implicitly been enumerated. At any iteration t, let

$$I_t = \{+1, +4, +5, +6\}$$

When I_t is fathomed, we write, after backtracking

$$I_{t+1} = \{+1, +4, +5, -6\}$$

which means that all elements of I_{t+1} are 1 except the one with the negative sign. The conversion of a positive element into a negative element is always right justified.

In the general case, the right most positive element of I_t is made negative and all the negative elements to the right side of it are deleted. If I_{t+1} is fathomed further, then we write

$$I_{t+2} = \{+1, +4, -5\}$$

If I_{t+1} cannot be fathomed, then a variable, say x_9, is included with a value of 1 to augment it and we get

$$I_{t+2} = \{+1, +4, +5, -6, +9\}$$

Then attempts are made to fathom I_{t+2}. This process is repeated until, at some later trial t', $I_{t'}$ is fathomed.

The procedure of backtracking is repeated as necessary. Backtracking is complete when all the elements of a fathomed partial solution are negative. At this point, all 2^n solutions of the problem have been effectively enumerated.

The algorithm is different from the previous algorithms in the sense that it requires only addition and subtraction to compute the result. However, at every iteration, one has to perform a number of tests. As the number of variables in a problem increases, the algorithm needs a larger number of iterations for the solution to the problem.

Generalized Lagrange Multiplier (GLM) Method of Everett

Everett [10] made an interesting study in the field of integer linear optimization. He established the fact that, in favorable situations, the Lagrange multiplier technique can be applied to solve integer linear programming problems.

He observed that if the solution set $\{x_i^*\} \subset Y$, where Y is the set containing all possible solution sets $\{x_i\}$, maximizes the unconstrained Lagrangian function

$$\sum_{i=1}^{n} C_i x_i - \sum_{j=1}^{m} \sum_{i=1}^{n} \lambda_j l_{ji} x_i \tag{6}$$

where the m constants λ_j are non-negative multipliers and C_i and l_{ji} are integers, then $\{x_i^*\}$ is a solution to the constrained problem

$$\left.\begin{aligned} &\text{maximize } \sum_{i=1}^{n} C_i x_i \\ &\text{such that} \\ &\sum_{i=1}^{n} l_{ji} x_i \leq \sum_{i=1}^{n} l_{ji} x_i^* \qquad j = 1, \ldots, m \end{aligned}\right\} \tag{7}$$

In general different combinations of λ_j's in (6) will lead to different solutions and it is necessary to adjust them by trial and error to determine if a given set of constraints is adequately satisfied.

The idea put forward by Everett does not say anything about the manner in which one can obtain the maxima of the unconstrained Lagrangian functions. All it says is that if one can find

the maximum of the modified function (6), then it is in fact a solution to the modified constrained problem given by (7). The method does not guarantee that a solutinn to the original problem can always be found. Because of the simplicity of the method, it is widely used in practice. It has also been asserted that the method succeeds in obtaining a satisfactory solution to a given problem in a surprising fraction of cases.

Kaplan's extension of Everett's GLM method

Kaplan [30] has extended Everett's method for solving the zero-one integer programming problem defined by (1). He makes use of the procedure originally outlined by Lorie and Savage [35] which calls for the independently maximizing each of the n functions

$$\gamma_i = C_i x_i - \sum_{j=1}^{m} \lambda_j l_{ji} x_i \qquad i = 1, \ldots, n$$

where each $\lambda_j > 0$ represents a multiplier.

Since the only possible solutions are $x_i = 0$ or $x_i = 1$, each of the γ_i is maximized by choosing x_i as follows:

$$x_i = 1 \text{ if } \beta_i = C_i - \sum_{j=1}^{m} \lambda_j l_{ji} > 0$$

$$x_i = 0 \text{ if } \beta_i = C_i - \sum_{j=1}^{m} \lambda_j l_{ji} \leq 0$$

The solution obtained by using this procedure is indeed

identical to the generalized Lagrange multiplier (GLM) solution which maximizes the Lagrangian function given by (6).

For actual calculations, he uses the following fact which is an extension of a theorem due to Everett.

If for any $x_i = 0$ (or $x_i = 1$) in a GLM optimal solution, the corresponding β_i is less than unity in absolute value, a new optimal solution will be created by letting $x_i = 1$ (or $x_i = 0$) and keeping all other variables as they were. More generally, variables from an original GLM solution can be dropped or added to form other optimal solutions, so long as $\sum_{i \in I} |\beta_i| < 1$ where I is the set of the subscripts of all those variables x_i which are dropped or added. All other variables not in the set I are kept at the GLM solution levels.

By using this procedure, many optimal solutions to problem (1) having different constraint values can be generated. One can then choose the solution whose constraints lie closest to the originally specified constraints. It may be noted that the above procedure is not guaranteed to produce an optimal solution to the original problem.

In this thesis, we develop an algorithm to solve single constraint, zero-one integer programming problems. The specific problem is formulated in dynamic programming. Several lemmas and theorems are presented in this thesis which modify the dynamic programming formulation of the problem. These modifications enable us to develop the new algorithm.

The special features of the algorithm make it possible to achieve a substantial reduction in computational and storage requirements as compared to the dynamic programming method. The most important feature of the algorithm is that it achieves a substantial reduction in the number of entries for the state values.

We then consider the solution of multi constraint zero-one integer programming problems. It is well known that, in dynamic programming, as the number of constraints increases, the computational difficulty increases exponentially. This problem can be avoided by applying available methods to either (i) reduce a multi constraint problem to a single constraint one with a large range for the state values, or (ii) replace the multi constraint problem by a large number of single constraint problems.

In both cases, the refined algorithm is found to be very effective in reducing the total amount of computation and storage space required to achieve the optimal solution.

In chapter 2 we add a review of basic dynamic programming theory. In chapter 3 we formulate the problem in dynamic programming and discuss the computational scheme in dynamic programming. In chapter 4 we first state and prove three theorems and four lemmas which are employed then to develop the refined algorithm. Chapters 5 and 6 deal with the application of the refined algorithm to solve problems having more than one constraint. Chapter 7 ends the thesis with concluding remarks.

18

CHAPTER 2

A REVIEW OF DYNAMIC PROGRAMMING THEORY

The most important factor in scientific decision making is to build up a mathematical model. Once the model is built up, an appropriate optimization technique is applied to solve the model. Dynamic programming is such an optimization technique used to solve complex optimization problems. Basically, dynamic programming is a multistage decision-making tool. It converts a multistage decision problem into a series of single stage decision problems.

> *"Dynamic programming starts with a small portion of the problem and finds the optimal solution for this smaller problem. It then gradually enlarges the problem, finding the correct optimal solution from the previous one, until the original problem is solved in its entirety."* ([25] p.241)

The basic principle of dynamic programming lies in two processes: *decomposition* and *composition*. The process of converting a problem into a number of subproblems is called *decomposition*. Then each subproblem is solved, and later their results are combined to compute the result of the original problem. This is called *composition*.

For a complex problem, if the decomposition-composition principle is followed, the resulting computational scheme may turn out to be more efficient than by solving the problem in a single stage. The dynamic programming theory is based on developing recursive equations which are in turn based on a number of vari-

ables. Before developing the recursive equations, we discuss the concept of *multistage analysis, multistage decision system*, and a number of definitions useful in the development of the recursive equations. For a more detailed analysis of these topics, reference can be made to [3], [4], [6], [11], [12], [21], [24], [26], [38], and [41].

2.1 Multistage analysis

The solution of a complex problem with the aid of the multistage approach lies in finding out a suitable decomposition into subproblems. The decomposition can be carried out through an appropriate transformation. The transformation may be made in forward or backward directions .

2.1.1 Forward transformation

Let a problem consist of n stages and let the system be initially defined by the state U_0; then there exists some transformation which can change the system characteristic so that the resulting system can be described by the state U_n. Let there be a transformation t'_n which transforms U_0 to U_n. We write

$$U_n = t'_n(U_0) \tag{2.1.1}$$

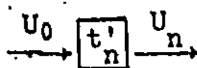


Figure 1

Let us assume that there is a certain transformation t_n which, when applied to a system in the state U_{n-1}, would change the system to Un, i.e.,

$$U_n = t_n(U_{n-1})$$

This is shown in figure 2.

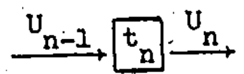


Figure 2

To solve the original problem, we now only need to find a transformation that will change the system from U_0 to U_{n-1}. Let t'_{n-1} be such a transformation. Figure 3 illustrates the transformations that change U_0 to U_n.

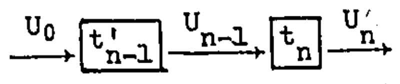


Figure 3

Proceeding in this way, we can show that there are transformations t_1, t_2, ..., t_n such that they transform the state of the system from U_0 to U_n as illustrated by figure 4.

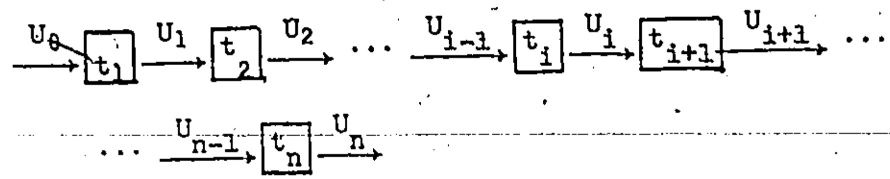


Figure 4

Thus, to arrive at a solution, it may be necessary to break up the problem (2.1.1) into n subproblems, which are as follows:

1. $U_1 = t_1(U_0)$

2. $U_2 = t_2(U_1)$

$\vdots$

i $U_i = t_i(U_{i-1})$

$\vdots$

n $U_n = t_n(U_{n-1})$

2.1.2 Backward transformation

If we have reached a state U_n of the system from the state U_0, then it is possible to construct an inverse transformation to arrive at U_0 from U_n. Furthermore, given a system at state U_n, it is possible to arrive at the system defined at the state U_0 through a series of transformations. This is called backward multistage problem solving. Thus, we may obtain the following scheme:

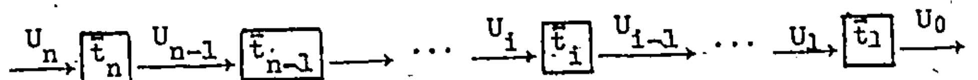


Figure 5

The corresponding subproblems are:

1 $U_{n-1} = \bar{t}_n(U_n)$

$\vdots$

n-i $U_i = \bar{t}_{i+1}(U_{i+1})$

n-i+1 $\quad U_{i-1} = \bar{t}_i(U_i)$

⋮

n $\quad U_0 = \bar{t}_1(U_1)$

Definition 2.1: Each subproblem i (i = 1, ..., n), into which the problem is decomposed is called a stage.

Definition 2.2: The decision making at each stage involves the selection of one of the alternatives of the stage. This is referred to as *stage decision*.

Definition 2.3: The stages in a problem are dependent. But it is necessary to treat them separately. That is, we need to separate the stages. This is done by the concept of *state* which summarizes the status of the system at every stage (with regard to the limitations that bind all the stages) which will permit making a feasible decision for the current stage.

2.2 Multistage decision system

In the previous section we have discussed the multistage problem solving approach. In this section, we further develop the approach to fit into our study. We achieve this by introducing the concept of decision making into the multistage problem solving

technique.

For an n stage decision system, the system at any stage i is characterized by the following factors:

(1) *A State*, U_i, that gives the status of the system at any stage i.

(2) *A decision variable*, x_i, that controls the operation of the system at any stage i.

(3) *A stage return*, r_i, that measures the utility of the system at stage i, i.e., r_i is a single-valued function of the decision variable x_i and the state U_i.

$$r_i = r_i(U_i, x_i)$$

(4) *A state transformation*, $\bar{t}_i$, which is a single-valued transformation at stage i. The transformation $\bar{t}_i$ is sometimes called the stage-coupling function (or stage inversion) such that, given the state U_i at stage i and its optimal stage decision x_i, one can determine U_{i-1}. We write

$$U_{i-1} = \bar{t}_i(U_i, x_i) \tag{2.2.1}$$

Thus a serial multistage system consists of a set of stages which are joined together by equation (2.2.1) such that, in general, the state U_i summarizes the status of the system at stages i, i-1, ..., 1.

From (2.2.1) it also follows that the state U_i at stage i depends on all the decisions made prior to stage i. Typically the decision at the first stage in a dynamic programming formulation

refers to the last decision which must be made in a series of sequential decisions. In general, the decision at the current stage i refers to the $(n-i+1)^{th}$ decision made in an n-stage problem. Thus, for state U_i we have at stage i+1

$$U_i = \bar{t}_{i+1}(U_{i+1}, x_{i+1})$$

where $\bar{t}_{i+1}$ is the state transformation

$$= \bar{t}_{i+1}(\bar{t}_{i+2}(U_{i+2}, x_{i+2}), x_{i+1})$$

$$= \bar{t}_{i+1}(U_{i+2}, x_{i+2}, x_{i+1})$$

$$= \bar{t}_{i+1}(\bar{t}_{i+3}(U_{i+3}, x_{i+3}), x_{i+2}, x_{i+1})$$

$$= \bar{t}_{i+1}(U_{i+3}, x_{i+3}, x_{i+2}, x_{i+1})$$

$$\vdots$$

$$U_i = \bar{t}_{i+1}(U_n, x_n, x_{n-1}, \ldots, x_{i+1}) \qquad (2.2.2)$$

The return from stage i is given by

$$r_i = r_i(U_i, x_i) \qquad (2.2.3)$$

By substituting for U_i from (2.2.2) in (2.2.3), we have

$$r_i = r_i(\bar{t}_{i+1}(U_n, x_n, \ldots, x_{i+1}), x_i)$$

$$= r_i(U_n, x_n, \ldots, x_i) \qquad (2.2.4)$$

From (2.2.4) it is evident that the return from stage i depends only on the decisions $(x_i, x_{i+1}, \ldots, x_n)$.

The total return R_n from stages 1 through n is some function g of the individual stage returns. We write

$$R_n(U_n, x_n, x_{n-1}, \ldots, x_1) = g[r_n(U_n, x_n), r_{n-1}(U_{n-1}, x_{n-1}), \ldots, r_1(U_1, x_1)] \quad (2.2.5)$$

Denoting the maximum n-stage return by $f_n^*(U_n)$, we write, using (2.2.5)

$$f_n^*(U_n) = \max_{x_n, \ldots, x_1} \{g[r_n(U_n, x_n), r_{n-1}(U_{n-1}, x_{n-1}), \ldots, r_1(U_1, x_1)]\}$$

$$\text{where } U_{i-1} = \bar{t}_i(U_i, x_i) \quad i=2, \ldots, n.$$

This equation for $f_n^*(U_n)$ will be used in the next section to develop the recursive equations of dynamic programming theory.

2.3 Development of the Recursive Equations

The formulation of the problem in dynamic programming theory is based on a class of equations called recursive equations. In this section, we shall develop the recursive equations by decomposing the problem

$$f_n^*(U_n) = \max_{x_n, \ldots, x_1} g[r_n(U_n, x_n), r_{n-1}(U_{n-1}, x_{n-1}), \ldots, r_1(U_1, x_1)] \quad (2.3.1)$$

$$\text{where } U_{i-1} = \bar{t}_i(U_i, x_i) \quad i = 2, \ldots, n$$

into n equivalent subproblems, each characterized by only one state and containing a decision variable. Each of the subproblems will be equivalent to a one-stage optimization problem.

We shall apply the multistage problem solving technique to decompose (2.3.1) into n subproblems. To achieve the decomposition, we make a highly restrictive assumption about the function g. Here we suppose that g is additive. Let

$$g[r_n(U_n, x_n), r_{n-1}(U_{n-1}, x_{n-1}), \ldots, r_1(U_1, x_1)]$$
$$= r_n(U_n, x_n) + r_{n-1}(U_{n-1}, x_{n-1}) + \ldots + r_1(U_1, x_1)$$

Thus we have

$$f^*_n(U_n) = \max_{x_n, \ldots, x_1} [r_n(U_n, x_n) + r_{n-1}(U_{n-1}, x_{n-1}) + \ldots \\ \ldots + r_1(U_1, x_1)] \qquad (2.3.2)$$

$$\text{where } U_{i-1} = \bar{t}_i(U_i, x_i) \quad i = 2, \ldots, n$$

Since the nth stage return $r_n(U_n, x_n)$ does not depend on $x_{n-1}, \ldots, x_1$, (see 2.2.4), therefore we can write (2.3.2) as

$$f^*_n(U_n) = \max_{x_n} [r_n(U_n, x_n) + \max_{x_{n-1}, \ldots, x_1} \{r_{n-1}(U_{n-1}, x_{n-1}) + \\ \ldots + r_1(U_1, x_1)\}]$$

$$\text{where } U_{i-1} = \bar{t}_i(U_i, x_i) \quad i = 2, \ldots, n$$

Hence, writing

$$f^*_{n-1}(U_{n-1}) = \max_{x_{n-1}, \ldots, x_1} [r_{n-1}(U_{n-1}, x_{n-1}) + \ldots \\ + r_1(U_1, x_1)]$$

we have

$$f^*_n(U_n) = \max_{x_n} [r_n(U_n, x_n) + f^*_{n-1}(U_{n-1})] \qquad (2.3.3)$$

$$\text{where } U_{n-1} = \bar{t}_n(U_n, x_n)$$

The determination of $f_n^*(U_n)$ and x_n, given $f_{n-1}^*(U_{n-1})$, is then simply a one-stage optimization problem with state U_n and decision variable x_n.

We can proceed further by treating $f_{n-1}^*(U_{n-1})$ and then $f_{n-2}^*(U_{n-2})$ ..., $f_2^*(U_2)$ in the same way, and decompose the original problem into n one-stage optimization problems as follows:

$$1 \quad f_1^*(U_1) = \max_{x_1}[r_1(U_1, x_1)]$$

$$2 \quad f_2^*(U_2) = \max_{x_2}[r_2(U_2, x_2) + f_1^*(U_1)]$$

$$\vdots$$

$$i \quad f_i^*(U_i) = \max_{x_i}[r_i(U_i, x_i) + f_{i-1}^*(U_{i-1})]$$

$$\vdots$$

$$n \quad f_n^*(U_n) = \max_{x_n}[r_n(U_n, x_n) + f_{n-1}^*(U_{n-1})]$$

$$\text{where } U_{i-1} = \bar{t}_i(U_i, x_i) \quad i = 2, \ldots, n$$

The solution of the above problems is equivalent to solving the following equations recursively:

$$\left.\begin{aligned} & f_1^*(U_1) = \max_{x_1}[r_1(U_1, x_1)] \\ \text{and } & f_i^*(U_i) = \max_{x_i}[r_i(U_i, x_i) + f_{i-1}^*(U_{i-1})] \\ & \qquad \text{where } U_{i-1} = \bar{t}_i(U_i, x_i) \quad i = 2, \ldots, n \end{aligned}\right\} \quad (2.3.4)$$

The equations in (2.3.4) represent the usual recursive equations in dynamic programming. These recursive equations are then

solved for all the stages. The optimal returns obtained in the present stage are used additively to compute the optimal returns at the next stage. Thus, in the last stage, the optimal value of return function is obtained by determining the maximum of $f_n^*(U_n)$.

Definition 2.4: A strategy, or a sequence of allowable decisions $(x_1, \ldots, x_n)$, will be called a policy, specifically an n-stage policy. Very typically the decision x_i will be the choice of a non-negative integer value of a single real variable.

Definition 2.5: An n-stage policy which yields the maximum value of some return function will be called an optimal policy. It will be denoted by $x_1^{opt}, x_2^{opt}, \ldots, x_n^{opt}$.

2.4 Characteristics of the Dynamic Recursive Process and the Principle of Optimality

The process of determining the optimal solution to an optimization problem with the help of dynamic programming is essentially recursive in nature. The recursive class of processes arising in dynamic programming has the property that after any number, say k, of the stage decisions have been made, the effect upon the total return due to the remaining n-k stages of the process depends only upon (a) the state of the system at the end of the kth

decision, and (b) the decisions at the subsequent stages. To achieve the total optimal return, we need to consider *only* the optimal return at every stage and the associated decisions. In other words, it is unnecessary to consider returns that are not optimal at each stage. After all, if we are to obtain an optimal solution for a system, any portion of the system must yield a solution which is optimal for that portion of the system. This is known as Bellman's Principle of Optimality. To quote Bellman:

> *"An optimal policy has the property that whatever the initial stage and the decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."* ([3] p. 83).

CHAPTER 3

THE SOLUTION OF THE ZERO-ONE SINGLE CONSTRAINT INTEGER PROGRAMMING PROBLEM BY USING DYNAMIC PROGRAMMING

From this chapter onward, we shall consider a special case of the problem defined in chapter 1. The problem to be considered henceforth for numerical solution will have only one constraint. This however, does not rule out the possibility of solving a multi-constraint problem with the algorithm developed in this thesis. In some later chapters, different available techniques will be applied to transform a multiconstraint problem into a single constraint one so that the solution to the original problem is the same as that obtained by solving the transformed problem. It appears that the solution of the transformed problem with the help of the refined algorithm developed in this thesis has some advantages over the direct solution of the multiconstraint problem.

The single-constraint problem may be written as

$$\left.\begin{aligned} &\text{maximize } Z_0 = \sum_{i=1}^{n} C_i x_i \\ &\text{subject to } \sum_{i=1}^{n} l_i x_i \leq b_1 \\ &\qquad x_i = 0, 1 \qquad i = 1, \ldots, n \end{aligned}\right\} \quad (3.1)$$

where it is assumed, without any loss of generality that C_i and l_i are positive integers. [17]

Throughout the remaining study of the thesis this problem will be

discussed from different points of view.

In this chapter, we formulate the problem in basic dynamic programming. The resulting computational scheme is then discussed and transformed into a step by step algorithm for the solution of problem (3.1) through dynamic programming.

3.1 Basic Dynamic Programming Formulation

In dynamic programming terminology, the problem defined by (3.1) may be viewed as an n stage problem. In our formulation, the state of the system at any stage i ($i = 1, 2, \ldots, n$) is defined by the limited resource U_i allocated so far such that $0 \leq U_i \leq b_1$, integer U_i. The decision variable x_i is restricted to the values $x_i = 0$ or 1, and satisfies the constraint $U_i - l_i x_i \geq 0$.

Let us denote the cumulative return at stage i from allocated resource U_i by $f_i(U_i)$. We denote the optimal values of $f_i(U_i)$ by $f_i^*(U_i)$. It is convenient to assume that $f_0^*(U_0) = 0$, for $0 \leq U_0 \leq b_1$. Applying the principle of optimality and using the forward induction, we may define the recursive equations for our problem by (3.1.1) and (3.1.2).

$$\left.\begin{aligned} & f_i(U_i) = \{C_i x_i + f_{i-1}^*(U_{i-1})\} \\ & \text{where } U_{i-1} = U_i - l_i x_i, \quad i = 1, \ldots, n, \\ & 0 \leq U_{i-1}, U_i \leq b_1 \end{aligned}\right\} \quad (3.1.1)$$

$$\left.\begin{array}{l} \text{and} \quad f_i^*(U_i) = \max_{x_i} f_i(U_i), \\ 0 \leq U_i \leq b_1, \quad i = 1, \ldots, n \end{array}\right\} \quad (3.1.2.)$$

Successively solving (3.1.1) and (3.1.2) for $i = 1, 2, \ldots, n$, we shall arrive at the optimal solution.

It is useful to note that each $f_i^*(U_i)$ generated by the recursive equations given by (3.1.1) and (3.1.2) represents an optimal solution to a subproblem of the problem (3.1). Thus for $i = j$, and $U_i = k$, $f_i^*(U_i)$ gives an optimal solution to the problem

$$\left.\begin{array}{l} \text{maximize} \sum_{i=1}^{j} C_i x_i \\ \text{subject to} \sum_{i=1}^{j} l_i x_i \leq k \\ \qquad x_i = 0, 1 \end{array}\right\} \quad (3.1.3)$$

The problem (3.1.3) is obtained by putting $n = j$, and $b_1 = k$ in (3.1).

3.2 Computational Scheme in Dynamic Programming

In order to carry out the computation of a discrete optimization problem through dynamic programming, we have to build up tables of $f_i(U_i)$, $U_i = 0, 1, 2, \ldots, b_1$ for each stage i with $i = 1, 2, \ldots, n$ in succession. First initialization is made by setting $f_i^*(U_i) = 0$ for $i = 0$ and $0 \leq U_i \leq b_1$. Then table forumlation

for $i = 1$ through n starts. Each table contains values of U_i in the first column. Values of $f_i(U_i)$ for $x_i = 0$ and 1 are calculated for each row and then a comparison is made row-wise to choose $f_i^*(U_i)$. At every stage i, the $f_i^*(U_i)$ column is stored. At the same time, we also store the value of x_i corresponding to each $f_i^*(U_i)$. Nemhauser [38] suggests that it is convenient to store x_i's as $x_i = x_i(U_i)$. The advantage of recording x_i's in this way lies in the fact that we do not have to keep a separate column for U_i and also that the corresponding value of x_i is readily located.

It is evident from the recursive equation that $f_{i-1}^*(U_{i-1})$ is needed in calculating $f_i^*(U_i)$, $i = 1, \ldots, n$. The scheme thus needs to reserve spaces for two optimal return tables $f_{i-1}^*(U_{i-1})$ and $f_i^*(U_i)$ at a time so that as soon as $f_i^*(U_i)$ has been calculated for all values of U_i, the $f_{i-1}^*(U_{i-1})$ column can be replaced by the column containing values of $f_i^*(U_i)$. However, for the storage of optimum decision variables $x_i(U_i)$, we must also reserve space for additional n columns. We continue proceeding in this way until we compute $f_n^*(U_n = b_1)$. We then choose

$$f_n^*(U_n^{opt}) = \max_{U_n} f_n^*(U_n) \qquad (3.2.1)$$

and find out $U_n^{opt} = U_n$ giving (3.2.1). We also determine the corresponding $x_n(U_n^{opt}) = x_n^{opt}$ as the optimal policy for the nth stage.

We then follow a traceback procedure to calculate U_{i-1}^{opt} from relations (3.2.2) and (3.2.3):

$$U_{i-1}^{opt} = U_i^{opt} - l_i x_i^{opt} \tag{3.2.2}$$

$$x_{i-1}^{opt} = x_{i-1}(U_{i-1}^{opt}) \tag{3.2.3}$$

for $i = n, n-1, \ldots, 2$

and successively compute $x_{n-1}^{opt}, x_{n-2}^{opt}, \ldots, x_1^{opt}$. Thus we get the optimum decision policy $x_1^{opt}, \ldots, x_n^{opt}$.

3.3 Step by Step Procedure for Computation Using Dynamic Programming

Here we transform the scheme discussed in section 3.2 into a step by step algorithm for computation. We call it algorithm 1.

Step 1: Initialization

Set $i \leftarrow 0$

Set $f_i^*(U_i) \leftarrow 0$ for $U_i = 0, 1, \ldots, b_1$

Step 2: Increment stage index i and initialize U_i

Set $i \leftarrow i + 1$, $U_i \leftarrow 0$

Step 3: Calculate $f_i^*(U_i) = \max_{x_i} f_i(U_i)$ and store the corresponding value of $x_i(U_i)$

Calculate $f_i(U_i)\Big|_{x_i=0} = \{C_i x_i + f_{i-1}^*(U_i - l_i x_i)\}\Big|_{x_i=0}$

If $U_{i-1} = U_i - l_i < 0$, then set $f_i^*(U_i) \leftarrow f_i(U_i)\Big|_{x_i=0}$

and $x_i(U_i) \leftarrow 0$.

Otherwise calculate $f_i(U_i)\Big|_{x_i=1} = \{C_i x_i + f_{i-1}^*(U_i - l_i x_i)\}\Big|_{x_i=1}$

and set $f_i^*(U_i) \leftarrow \max \left\{ f_i(U_i)\Big|_{x_i=0}, f_i(U_i)\Big|_{x_i=1} \right\}$

and if $f_i^*(U_i) = f_i(U_i)\Big|_{x_i=0}$ then set $x_i(U_i) \leftarrow 0$, otherwise

set $x_i(U_i) \leftarrow 1$.

Step 4: Loop on U_i

If $U_i = b_1$, then go to step 5;

otherwise set $U_i \leftarrow U_i + 1$ and go to step 3.

Step 5: Loop on i

If $i < n$, go to step 2;

otherwise go to step 6.

Step 6: Find $f_n^*(U_n^{opt})$ and U_n^{opt}

Calculate $f_n^*(U_n^{opt}) = \max_{U_n} f_n^*(U_n)$, save U_n^{opt},

Initialize $i \leftarrow n$ for backtracking.

Step 7: Find x_i^{opt} by backtracking

Set $x_i^{opt} \leftarrow x_i(U_i^{opt})$

and calculate $U_{i-1}^{opt} = U_i^{opt} - l_i x_i^{opt}$

Step 8: Loop on i for backtracking

If $i > 1$, set $i \leftarrow i-1$ and go to step 7;

otherwise go to exit.

CHAPTER 4

A REFINED ALGORITHM FOR THE SOLUTION OF THE ZERO-ONE SINGLE CONSTRAINT INTEGER PROGRAMMING PROBLEM

In chapter 3 we observed that, for the solution of the zero-one single constraint problem by using dynamic programming, at any stage i, we need to calculate $f_i(U_i)\Big|_{x_i=0}$ and $f_i(U_i)\Big|_{x_i=1}$ and compare them to determine $f_i^*(U_i)$ and to store $x_i(U_i)$ for all integer values of U_i in the range $0 \leq U_i \leq b_1$. This requires a large amount of computation as well as a large amount of storage space. To achieve a substantial amount of reduction in these requirements, in this chapter we prove some theorems and lemmas which are then utilized to develop a refined algorithm. The refined algorithm does the following:

(i) It helps to avoid the calculation of values for $f_i(U_i)\Big|_{x_i=0}$ and thus the number of columns required for storage is reduced by one.

(ii) It reduces the number of entries for U_i's at stage i. This enables us to achieve a reduction in the number of computations and also in the amount of overall storage requirement for $x_i(U_i)$. It can be noted that the smaller the number of entries for U_i, the less will be the amount of the associated requirements to determine $f_i^*(U_i)$ and to store $x_i(U_i)$.

Lemma 2 is used in achieving (i) above. Theorem 1 and Theorem 2 together establish (ii). Theorem 3, on the other hand, overcomes the disadvantage due to the nonavailability of certain $f_i^*(U_i)$, in the immediately preceding stage, which are required for the calculation of

$$f_{i+1}^*(U_{i+1}), \quad i = 1, \ldots, n-1.$$

For the convenience of developing the refined algorithm, we reformulate problem (3.1) so that the variables x_i appear in an appropriate order such that

$$\frac{C_i}{l_i} \geq \frac{C_{i+1}}{l_{i+1}}, \quad i = 1, \ldots, n-1.$$

We write the problem as

$$\left.\begin{aligned} &\text{maximize } Z_0 = \sum_{i=1}^{n} C_i x_i \\ &\text{subject to } \sum_{i=1}^{n} l_i x_i \leq b_1 \\ &\qquad x_i = 0, 1 \quad i = 1, \ldots, n \\ &\text{where } C_i \text{ and } l_i \text{ are positive integers satisfying the} \\ &\text{condition } \frac{C_i}{l_i} \geq \frac{C_{i+1}}{l_{i+1}} \quad i = 1, \ldots, n-1 \\ &\text{and } l_i \leq b_1 \text{ for all } i. \end{aligned}\right\} \quad (4.1)$$

It will soon be seen that the symbol U_i^* plays an important role throughout the remaining discussions in this chapter. We introduce the following definition for U_i^*:

Definition 4.1: Given i and $f_i^*(U_i)$, we define the ith optimum (resource) allocation U_i^* as the least value of U_i such that

$$f_i^*(U_i^*) = \max_{U_i} f_i^*(U_i).$$

4.1 Lemma 1

For a multistage single constraint problem of zero-one integer programming, the optimal results at the first stage are given by

(i) $f_1^*(U_1) = 0, \quad x_1(U_1) = 0$ if $U_1 < l_1$

(ii) $f_1^*(U_1) = C_1, \quad x_1(U_1) = 1$ if $U_1 \geq l_1$

and (iii) $U_1^* = l_1, \quad f_1^*(U_1^*) = C_1$

Proof: The optimal returns at the first stage are given by the recursive equation

$$f_1^*(U_1) = \max_{\substack{x_1 \\ 0 \leq U_1 \leq b_1}} \{C_1 x_1 + f_0^*(U_1 - l_1 x_1)\}$$

If $U_1 < l_1$, then $U_1 - l_1 < 0$ and therefore $x_1(U_1) = 0$ is the only admissible value. In that case

$$f_1^*(U_1) = f_0^*(U_0) = 0, \text{ by assumption.}$$

This prives (i).

If, on the other hand, $U_1 \geq l_1$, then for $x_1(U_1) = 1$ we get $U_1 - l_1 = U_0 \geq 0$ yielding the optimal value

$$f_1^*(U_1) = C_1$$

which proves (ii).

From (ii) and definition (4.1) we obtain $U_1^* = l_1$ and $f_1^*(U_1^*) = C_1$

which proves (iii).

4.2 Lemma 2

Given stage i and $f_i^*(U_i)$, the value of $f_{i+1}(U_{i+1})\Big|_{x_{i+1}=0}$

at the subsequent stage is given by

$$f_{i+1}(U_{i+1})\Big|_{x_{i+1}=0} = f_i^*(U_i) \quad \text{for } U_i = U_{i+1}$$

Proof: We have, from (3.1.1)

$$f_{i+1}(U_{i+1}) = \{C_{i+1}x_{i+1} + f_i^*(U_{i+1} - l_{i+1}x_{i+1})\}$$

where $U_i = U_{i+1} - l_{i+1}x_{i+1}$

For $x_{i+1} = 0$, we get

$$f_{i+1}(U_{i+1})\Big|_{x_{i+1}=0} = f_i^*(U_i)$$

where $U_i = U_{i+1}$

This proves lemma 2.

4.3 Lemma 3.

$f_i^*(U_i)$ is a non-decreasing function of both i and U_i.

Proof: In order to prove that $f_i^*(U_i)$ is non-decreasing in i, we observe that by definition

$$f_i^*(U_i) = \max \left\{ f_i(U_i)\Big|_{x_i=0},\ f_i(U_i)\Big|_{x_i=1} \right\}$$

But

$$f_i(U_i)\Big|_{x_i=0} = f_{i-1}^*(U_i)$$

Hence

$$f_i^*(U_i) \geq f_{i-1}^*(U_i)$$

To prove that $f_i^*(U_i)$ is non-decreasing in U_i, we note that for i = 1, lemma 1 implies that $f_1^*(U_1)$ is indeed a non-decreasing function of U_1.

Let us assume that for some value k of i, $f_k^*(U_k)$ is a non-decreasing function of U_k. Consider $f_{k+1}^*(U_{k+1})$. Let j_1 and j_2 be two values such that $j_2 > j_1$.

From definition we have

$$f_{k+1}^*(j_1) = \max \{f_k^*(J_1),\ c_{k+1} + f_k^*(J_1 - l_{k+1})\} \quad (4.3.1),$$

and

$$f_{k+1}^*(j_2) = \max \{f_k^*(j_2),\ c_{k+1} + f_k^*(j_2 - l_{k+1})\} \quad (4.3.2)$$

Since $j_1 < j_2$ we also have

$$f_k^*(j_1) \leq f_k^*(j_2) \tag{4.3.3}$$

and

$$C_{k+1} + f_k^*(j_1 - l_{k+1}) \leq C_{k+1} + f_k^*(j_2 - l_{k+1}) \tag{4.3.4}$$

Combining (4.3.1), (4.3.2), (4.3.3), and (4.3.4), we have

$$f_{k+1}^*(j_2) \geq f_{k+1}^*(j_1).$$

This proves the lemma.

4.4 Lemma 4

If $f_j^*(U_j - 1) \neq f_j^*(U_j)$, then

$f_j^*(U_j)$ is given by

$$\left.\begin{aligned} & f_j^*(U_j) = \max_{x_i} \sum_{i=1}^{j} C_i x_i \\ & \text{subject to the equality constraint} \\ & \sum_{i=1}^{j} l_i x_i = U_j \end{aligned}\right\} \quad (4.4.1)$$

Proof: From (3.1.3) we have

$$\left.\begin{aligned} & f_j^*(U_j - 1) = \max_{x_i} \sum_{i=1}^{j} C_i x_i \\ & \text{such that } \sum_{i=1}^{j} l_i x_i \leq U_j - 1 \end{aligned}\right\} \quad (4.4.2)$$

If, in (4.4.1) $\sum_{i=1}^{j} l_i x_i \neq U_j$, then $\sum_{i=1}^{j} l_i x_i \leq U_j - 1$ since U_j is an integer. Then (4.4.1) becomes identical to (4.4.2). Therefore, they must yield the same solution, which contradicts our assumption.

Hence the lemma.

It may be noted that because of lemma 4 and the definition 4.1 of U_j^*, it follows that $f_j^*(U_j^*)$ is given by

$$f_j^*(U_j^*) = \max_{x_i} \sum_{i=1}^{j} C_i x_i.$$

such that the equality $\sum_{i=1}^{j} l_i x_i = U_j^*$ holds.

4.5 Theorem 1

Given i, if U_i^* is the ith optimum allocation as defined by definition 4.1, then for all i

$$\text{(i)}\quad f_i^*(U_i) < f_i^*(U_i^*) \quad \text{for } U_i < U_i^*$$

$$f_i^*(U_i) = f_i^*(U_i^*) \quad \text{for } U_i \geq U_i^*$$

for $i = 1, 2, \ldots, n$

and (ii) $U_{i+1}^* \geq U_i^*$

for $i = 1, 2, \ldots, n-1$.

Proof: Since $U_i \in \{0, 1, \ldots, b_1\}$, U_i^* exists. This fact combined with lemma 3 establishes (i).

To prove (ii) we first observe that

$$f_i^*(U_i^*) = \max \left\{ f_i(U_i^*)\Big|_{x_i=0},\ f_i(U_i^*)\Big|_{x_i=1} \right\}$$

Let

$$f_i^*(U_i^*) = f_i(U_i^*)\Big|_{x_i=0} = f_{i-1}^*(U_i^*) \tag{4.5.1}$$

But we have

$$f_{i-1}^*(U_{i-1}^*) \geq f_{i-1}^*(U_i^*) \tag{4.5.2}$$

$$\text{and } f_i^*(U_i^*) \geq f_{i-1}^*(U_{i-1}^*) \tag{4.5.3}$$

Inequalities (4.5.2) and (4.5.3) give

$$f_i^*(U_i^*) \geq f_{i-1}^*(U_{i-1}^*) \geq f_{i-1}^*(U_i^*)$$

and hence using (4.5.1), we obtain

$$f_i^*(U_i^*) = f_{i-1}^*(U_i^*) = f_{i-1}^*(U_{i-1}^*).$$

Therefore

$$U_i^* = U_{i-1}^*.$$

If on the other hand

$$f_i^*(U_i^*) = f_i(U_i^*)\Big|_{x_i=1}$$

$$= C_i + f_{i-1}^*(U_i^* - l_i) \tag{4.5.4}$$

then either

$$\text{(a) } U_i^* - l_i \geq U_{i-1}^*$$

or

$$\text{(b) } U_i^* - l_i < U_{i-1}^*$$

If $U_i^* - l_i \geq U_{i-1}^*$, then, since $l_i > 0$

$$U_i^* \geq U_{i-1}^* + l_i > U_{i-1}^*.$$

If $U_i^* - l_i < U_{i-1}^*$, then let $\delta = U_{i-1}^* - (U_i^* - l_i)$, and we obtain

$$f_{i-1}^*(U_i^* - l_i) = f_{i-1}^*(U_{i-1}^* - \delta)$$
$$\leq f_{i-1}^*(U_{i-1}^*) - (U_{i-1}^* - U_i^* + l_i)\frac{C_{i-1}}{l_{i-1}}$$

by using lemma 4.

Combining this with (4.5.4) we obtain

$$f_i^*(U_i^*) \leq C_i + f_{i-1}^*(U_{i-1}^*) - (U_{i-1}^* - U_i^* + l_i)\frac{C_{i-1}}{l_{i-1}}$$

which yields, on using lemma 3

$$C_i - (U_{i-1}^* - U_i^* + l_i)\frac{C_{i-1}}{l_{i-1}} \geq f_i^*(U_i^*) - f_{i-1}^*(U_{i-1}^*) \geq 0$$

Consequently

$$(U_{i-1}^* - U_i^* + l_i)\frac{C_{i-1}}{l_{i-1}} \leq C_i$$

or

$$(U_{i-1}^* - U_i^*)\frac{C_{i-1}}{l_{i-1}} \leq \left(\frac{C_i}{l_i} - \frac{C_{i-1}}{l_{i-1}}\right) l_i$$

where $\left(\frac{C_i}{l_i} - \frac{C_{i-1}}{l_{i-1}}\right) \leq 0$

Hence $U_i^* \geq U_{i-1}^*$.

This proves (ii).

4.6 Definition of U_i^m

It will be seen that the quantity U_i^m plays an important role in the remaining discussion in this chapter. We introduce the following definition for U_i^m.

Definition 4.2: Given stage i, we define U_i^m as the minimum value of U_i for which the inequality

$$f_i^*(U_i) + (b_1 - U_i)\frac{c_{i+1}}{l_{i+1}} < f_i^*(U_i^*) \qquad (4.6.1)$$

does not hold.

It is useful to note that the left hand side of the inequality (4.6.1) is an upper bound on the return Z_0 which includes $f_i^*(U_i)$ as an optimal partial return at the ith stage for a given U_i.

4.7 Theorem 2

If U_i^m is as defined above for stage i, then U_i^m satisfies the inequalities

(i) $U_i^m \leq U_i^*$

(ii) $U_i^m \leq U_k^m$ for $i < k \leq n-1$

Proof: To prove (i), we observe that

$$f_i^*(U_i) = f_i^*(U_i^*) \text{ for } U_i \geq U_i^*$$

and that

$$(b_1 - U_i)\frac{c_{i+1}}{l_{i+1}} \geq 0 \quad \text{for all } U_i.$$

Consequently the relationship (4.6.1) which defines U_i^m is violated for $U_i \geq U_i^*$. This proves (i).

To prove (ii), we need to show that for all $U_k \geq U_i^m$ where

$i < k \leq n-1$, the inequality

$$f_k^*(U_k) + (b_1 - U_k)\frac{C_{k+1}}{l_{k+1}} < f_k^*(U_k^*) \quad (4.7.1)$$

holds.

Let us consider $k = i + 1$. Let

$$f_{i+1}^*(U_{i+1}) = f_{i+1}(U_{i+1})\Big|_{x_{i+1}=0} = f_i^*(U_{i+1})$$

Then, since

$$\frac{C_{i+2}}{l_{i+2}} \leq \frac{C_{i+1}}{l_{i+1}},$$

we obtain

$$\left.\begin{aligned} f_{i+1}^*(U_{i+1}) + (b_1 - U_{i+1})\frac{C_{i+2}}{l_{i+2}} &\leq f_i^*(U_{i+1}) + (b_1 - U_{i+1})\frac{C_{i+1}}{l_{i+1}} \\ &< f_i^*(U_i^*) \end{aligned}\right\} (4.7.2)$$

for $U_{i+1} < U_i^m$

If, on the other hand

$$f_{i+1}^*(U_{i+1}) = f_{i+1}(U_{i+1})\Big|_{x_{i+1}=1} = C_{i+1} + f_i^*(U_{i+1} - l_{i+1}) \quad (4.7.3)$$

then, for $U_{i+1} - l_{i+1} < U_i^m$, we have

$$f_i^*(U_{i+1} - l_{i+1}) + (b_1 - U_{i+1} + l_{i+1})\frac{C_{i+1}}{l_{i+1}} < f_i^*(U_i^*).$$

Combining the above with (4.7.3), we obtain

$$f_{i+1}^*(U_{i+1}) + (b_1 - U_{i+1})\frac{C_{i+1}}{l_{i+1}} < f_i^*(U_i^*).$$

Since $\frac{C_{i+2}}{l_{i+2}} \leq \frac{C_{i+1}}{l_{i+1}}$, the above inequality reduces to

$$f^*_{i+1}(U_{i+1}) + (b_1 - U_{i+1})\frac{C_{i+2}}{l_{i+2}} < f^*_i(U^*_i) \qquad (4.7.4)$$

Combining (4.7.2) and (4.7.4) and using the fact that $f^*_i(U^*_i) \leq f^*_{i+1}(U^*_{i+1})$, we obtain

$$f^*_{i+1}(U_{i+1}) + (b_1 - U_{i+1})\frac{C_{i+2}}{l_{i+2}} < f^*_{i+1}(U^*_{i+1})$$

for $U_{i+1} < U_i{}^m$

This implies that $U_{i+1}{}^m \geq U_i{}^m$.

Recursion on i proves (ii).

4.8 Theorem 3

If at any stage i+1, the inequality

$$U_{i+1} - l_{i+1} < U_{i-1}{}^m$$

holds, then

(i) $f_{i+1}(U_{i+1})\Big|_{x_{i+1}=1} + (b_1 - U_{i+1})\cdot\frac{C_{i+1}}{l_{i+1}} < f^*_i(U^*_i)$

hence also

(ii) $f_{i+1}(U_{i+1})\Big|_{x_{i+1}=1} + (b_1 - U_{i+1})\cdot\frac{C_{i+2}}{l_{i+2}} < f^*_i(U^*_i)$.

Proof: Since at any stage i for $U_i{}^m \leq U_{i-1}{}^m$, we have for $U_i < U_{i-1}{}^m$

$$f^*_i(U_i) + (b_1 - U_i)\frac{C_{i+1}}{l_{i+1}} < f^*_i(U^*_i)$$

If in the above we put $U_i = U_{i+1} - l_{i+1}$, we obtain for $U_{i+1} - l_{i+1} < U_{i-1}{}^m$

$$f_i^*(U_{i+1} - l_{i+1}) + (b_1 - U_{i+1} + l_{i+1})\frac{C_{i+1}}{l_{i+1}} < f_i^*(U_i^*)$$

The above combined with the fact that

$$f_{i+1}(U_{i+1})\Big|_{x_{i+1}=1} = C_{i+1} + f_i^*(U_{i+1} - l_{i+1}),$$

yields (i).

The inequality (ii) follows from (i) since

$$\frac{C_{i+2}}{l_{i+2}} \leq \frac{C_{i+1}}{l_{i+1}}.$$

4.9 Discussion of Lemmas and Theorems

In this section we shall discuss the lemmas and theorems in the light of their possible contribution in the development of the refined algorithm.

From lemma 1 it is evident that we can avoid the initialization step in the dynamic programming algorithm discussed in section 3.3 of chapter 3. The results for stage 1 can directly be computed from lemma 1.

From lemma 2 it follows that at any stage i+1, the calculation of $f_{i+1}(U_{i+1})\Big|_{x_{i+1}=0}$ can be avoided since these values are already in the ith stage as $f_i^*(U_i)$.

The properties established in lemma 3 help us in proving

theorem 1 which established the fact that the optimum allocation and return at stage i+1 cannot be less than those obtained in the ith stage.

Lemma 4 on the other hand establishes the fact that for the optimum resource allocation U_i^* at the ith stage, the constraint is satisfied in the equality sense with b_1 replaced by U_i^*.

The quantity U_i^m defined by definition 4.2 and determined by the inequalities in theorem 2 is a powerful factor in achieving a reduction in the state values U_{i+1}. The test for determining U_i^m ensures that the minimum value of U_{i+1}, which is needed for arriving at the optimal solution is U_i^m. The facts $U_i^m \leq U^*$, and $U_i^m \leq U_k^m$, for $i < k \leq n - 1$, enable us to keep all the possible values of U_{i+1} which will definitely contribute to the optimal values at the subsequent stages.

In theorem 3 the inequalities (i) and (ii) are satisfied for $U_{i+1} - l_{i+1} = U_i < U_{i-1}^m$, where U_{i-1}^m is the first recorded entry for U_i at the ith stage. This means that in such cases the values $f_{i+1}(U_{i+1})\Big|_{x_{i+1}=1}$ cannot contribute to the optimal results at the subsequent stages. Hence we do not need to consider the value $x_{i+1} = 1$ for those values of U_{i+1} satisfying (i) and (ii). Therefore, it is sufficient to consider the only other value $x_{i+1} = 0$ for those values of U_{i+1} and hence using lemma 2 we set $f_{i+1}^*(U_{i+1}) =$

$$f_{i+1}(U_{i+1})\Big|_{x_{i+1}=0} = f_i^*(U_{i+1}).$$

4.10 The Refined Algorithm

In this section we describe the refined algorithm in the light of the theorems and lemmas established and discussed in the preceding sections. This is followed by a description of the step by step procedure for the refined algorithm.

One important difference between the refined algorithm and the dynamic programming algorithm described in chapter 3 is that, at every stage i, it needs the calculation of $f_i(U_i)$ for $x_i = 1$ only. This is a contribution of lemma 2 in this algorithm. Furthermore, unlike the previous algorithm here we do not have to make the initialization for i = 0. We start computation by applying lemma 1 to calculate the values of $f_1^*(U_1)$. We then determine U_1^* and $f_1^*(U_1^*)$. We test the results in the $f_1^*(U_1)$ column against the inequality (4.6.1) to determine U_1^m for the second stage.

Given the return $f_i^*(U_i)$ corresponding to the partial resource U_i allocated so far at stage i, we determine the maximum possible return to be achieved from all the subsequent stages by allocating the remaining resource $(b_i - U_i)$ to these stages. The determination of an upper bound for this later return uses the ordered property $\frac{c_i}{l_i} \geq \frac{c_{i+1}}{l_{i+1}}$.

The test (4.6.1) thus tests the sum of $f_i^*(U_i)$ and the maximum possible return from the subsequent stages against the maximum return $f_i^*(U_i^*)$ which we have already achieved at this stage. The test when satisfied for certain values of U_i does not therefore lead to an improved optimal solution at the subsequent stages.

We may therefore conclude that those values of U_i, for which the test (4.6.1) is satisfied, cannot lead to an improved optimal solution. Thus we go on applying the test (4.6.1) until we get a value U_i^m of U_i which violates the test. At the next stage i+1, we then have to determine $f_{i+1}^*(U_{i+1})$ and $x_{i+1}(U_{i+1}$ for $U_i^m \leq U_{i+1} \leq b_1$.

After the calculation for $f_{i+1}(U_{i+1})\Big|_{x_{i+1}=1}$ has been performed, we determine $f_{i+1}^*(U_{i+1})$ from

$$f_{i+1}^*(U_{i+1}) = \max\left\{f_i^*(U_{i+1}),\ f_{i+1}(U_{i+1})\Big|_{x_{i+1}=1}\right\}$$

and store the corresponding value of $x_{i+1}(U_{i+1})$.

The principle for storage of $f_i^*(U_i)$ and $x_i = x_i(U_i)$ at every stage i used in this algorithm is the same as in the previous algorithm. However, the storage procedure requires some attention in this algorithm.

In this algorithm the determination of $f_{i+1}^*(U_{i+1})$ for those values of U_{i+1} for which $U_i = U_{i+1} - l_{i+1} < U_{i-1}^m$ requires the

application of theorem 3 by virtue of which we set $f^*_{i+1}(U_{i+1}) \leftarrow f^*_i(U_{i+1})$ and $x_{i+1}(U_{i+1}) \leftarrow 0$.

The determination of $f_n^{opt}(U_n) = \max_{U_n} f^*_n(U_n)$ is the same as in the previous algorithm. The backtracking scheme for determining the optimum values of the decision variables $x_n^{opt}, \ldots, x_1^{opt}$ is also the same as in the previous algorithm.

4.11 Step by Step Procedure for the Solution of Problem (4.1) by using the Refined Algorithm

In this section, we present a step by step procedure for the refined algorithm discussed in the previous section.

Step 1: Calculate results for stage 1 from lemma 1.

Set $i \leftarrow 1$.

Set $f^*_i(U_i) \leftarrow 0$, $x_i(U_i) \leftarrow 0$, for $U_i < l_1$

and $f^*_i(U_i) \leftarrow C_1$, $x_i(U_i) \leftarrow 1$ for $U_i \geq l_1$

Set $U^*_i \leftarrow l_1$ and $f^*_i(U^*_i) \leftarrow C_1$.

Step 2: Apply test (4.6.1) to determine U_i^m for the next stage. Use the inequality

$$f^*_i(U_i) + (b_1 - U_i) \cdot \frac{C_{i+1}}{l_{i+1}} < f^*_i(U^*_i)$$

and determine U_i^m which is the lowest value of U_i for which the

inequality does not hold.

Step 3: Increment stage index i and set starting value of U_i.

Set $i \leftarrow i+1$,

then set $U_i \leftarrow U_{i-1}^{m}$

Step 4: Calculate $f_i^*(U_i)$ and store the corresponding $x_i(U_i)$.

Calculate $f_i(U_i)\Big|_{x_i=1} = C_i + f_{i-1}^*(U_i - l_i)$

If $U_{i-1} = U_i - l_i < U_{i-2}^{m}$, then set $f_{i-1}^*(U_i) \leftarrow f_i^*(U_i)$ and $x_i(U_i) \leftarrow 0$. Otherwise set $f_i^*(U_i) \leftarrow \max\left\{f_{i-1}^*(U_i),\ f_i(U_i)\Big|_{x_i=1}\right\}$ and if $f_i^*(U_i) = f_{i-1}^*(U_i)$, then set $x_i(U_i) \leftarrow 0$; otherwise set $x_i(U_i) \leftarrow 1$.

Step 5: Loop on U_i.

If $U_i = b_i$, then go to step 6; otherwise set $U_i \leftarrow U_i+1$ and go to step 4.

Step 6: Determine U_i^* and $f_i^*(U_i^*)$.

Use definition (4.1) to determine U_i^* and $f_i^*(U_i^*)$.

Step 7: Loop on i.

If $i < n$, go to step 2; otherwise go to step 8.

Step 8: Find $f_n^*(U_n^{opt})$ and U_n^{opt}.

Calculate $f_n^*(U_n^{opt}) = \max_{U_n} f_n^*(U_n)$, and save U_n^{opt}.

Initialize $i \leftarrow n$ for backtracking.

Step 9: Find x_i^{opt} by backtracking.

Set $x_i^{opt} \leftarrow x_i(U_i^{opt})$ and calculate $U_{i-1}^{opt} = U_i^{opt} - l_i x_i^{opt}$.

Step 10: Loop on i for backtracking.

If $i > 1$, set $i \leftarrow i-1$ and go to step 9; otherwise go to exit.

4.12 Numerical example

In this section we solve a numerical example to show the effectiveness of the refined algorithm and also to demonstrate how the reduction in the number of entries for the state values is achieved. The example is taken from [43].

It will be seen that the refined algorithm achieves a substantial reduction in the total number of the entries for the state values U_i as compared to those needed in the dynamic programming algorithm given in chapter 3.

Example 4.1:

$$\text{maximize } Z_0 = 60x_1 + 60x_2 + 40x_3 + 10x_4 + 20x_5 + 10x_6 + 3x_7$$

$$\text{subject to } 3x_1 + 5x_2 + 4x_3 + x_4 + 4x_5 + 3x_6 + x_7 \leq 10$$

$x_i = 0, 1 \quad i = 1, \ldots, 7.$

Solution: We have the recursive equations

$$f_i(U_i) = \{ C_i x_i + f^*_{i-1}(U_{i-1})\}$$

where $U_{i-1} = U_i - l_i x_i$

and $f^*_i(U_i) = \max_{x_i} f_i(U_i)$

Stage 1.

We apply lemma 1 to achieve the results of stage 1. Thus, we have the following table:

U_1	$f^*_1(U_1)$	$x_1(U_1)$
0	0	0
1	0	0
2	0	0
3	60	1
4	60	1
5	60	1
6	60	1
7	60	1
8	60	1
9	60	1
10	60	1

We also have $U^*_1 = 3$ and $f^*_1(U_1) = 60$. Now the values in $f^*_1(U_1)$ column are tested against the inequality (4.6.1) and we get $U_1^m = 0$. Therefore, in stage 2 we have $0 \leq U_2 \leq 10$.

Stage 2.

U_2	$f_2(U_2)\big\vert_{x_2=1}$	$f_2^*(U_2)$	$x_2(U_2)$
0	--	0	0
1	--	0	0
2	--	0	0
3	--	60	0
4	--	60	0
5	60 + 0	60	0
6	60 + 0	60	0
7	60 + 0	60	0
8	60 + 60	120	1
9	60 + 60	120	1
10	60 + 60	120	1

Here $U_2^* = 8$ and $f_2^*(U_2^*) = 120$. Test (4.6.1) give $U_2^m = 3$.

Thus in stage 3, we have $3 \leq U_3 \leq 10$

Stage 3.

U_3	$f_3(U_3)\big\vert_{x_3=1}$	$f_3^*(U_3)$	$x_3(U_3)$
3	--	60	0
4	40 + 0	60	0
5	40 + 0	60	0
6	40 + 0	60	0
7	40 + 60	100	1
8	40 + 60	120	0
9	40 + 60	120	0
10	40 + 60	120	0

Here $U_3^* = 8$ and $f_3^*(U_3^*) = 120$. Test (4.6.1) gives $U_3^m = 3$. Thus, in stage 4, we have $3 \leq U_4 \leq 10$.

Stage 4.

U_4	$f_4(U_4)\big\vert_{x_4=1}$	$f_4^*(U_4)$	$x_4(U_4)$
3	10 + ?	60	0
4	10 + 60	70	1
5	10 + 60	70	1
6	10 + 60	70	1
7	10 + 60	100	0
8	10 + 100	120	0
9	10 + 120	130	1
10	10 + 120	130	1

Here, for $U_4 = 3$, we get $U_4 - 1_4 < U_3^m$. Therefore we are unable to compute $f_4(U_4 = 3)\big\vert_{x_4=1}$ since the associated value for $f_3^*(U_3 = 2)$ is not available in stage 3. This fact is indicated in table by using the sumbol "?". Hence we apply theorem 3 and just set $f_4^*(U_4 = 3) \leftarrow f_3^*(U_3 = 3)$ and $x_4(U_4 = 3) \leftarrow 0$.

Here $U_4^* = 9$ and $f_4^*(U_4^*) = 130$. Test (4.6.1) gives $U_4^m = 8$. Thus in stage 5 we have $8 \leq U_5 \leq 10$.

Stage 5.

U_5	$f_5(U_5)\big\vert_{x_5=1}$	$f_5^*(U_5)$	$x_5(U_5)$
8	20 + 70	120	0
9	20 + 70	130	0
10	20 + 70	130	0

Here $U_5^* = 9$ and $f_5^*(U_5^*) = 130$. Test (4.6.1) gives $U_5^m = 9$.

Thus, in stage 6, we have $9 \leq U_6 \leq 10$.

Stage 6.

U_6	$f_6(U_6)\big\vert_{x_6=1}$	$f_6^*(U_6)$	$x_6(U_6)$
9	10 + ?	130	0
10	10 + ?	130	0

Here, for both $U_6 = 9$ and 10, we get $U_6 - l_6 < U_5^m$. Hence, as in stage 4, we apply theorem 3 and set $f_6^*(U_6 = 9) \leftarrow f_5^*(U_5 = 9)$, $x_6(U_6 = 9) \leftarrow 0$ and $f_6^*(U_6 = 10) \leftarrow f_5^*(U_5 = 10)$, $x_6(U_6 = 10) \leftarrow 0$.

We have here $U_6^* = 9$ and $f_6^*(U_6) = 130$. Test (4.6.1) gives $U_6^m = 9$. Thus in stage 7 we have $9 \leq U_7 \leq 10$.

Stage 7.

U_7	$f_7(U_7)\big\vert_{x_7=1}$	$f_7^*(U_7)$	$x_7(U_7)$
9	3 + ?	130	0
10	3 + 130	133	1

Here for $U_7 = 9$, we get $U_7 - l_7 < U_6^m$. Hence we apply theorem 3 and set $f_7^*(U_7 = 9) \leftarrow f_6^*(U_6 = 9)$, and $x_7(U_7 = 9) \leftarrow 0$.

Since this is the last stage we determine $f_7^*(U_7^{opt} = 10) = 133$ and $x_7^{opt} = x_7(U_7^{opt} = 10) = 1$ respectively. We note that $Z_0 = f_7^*(U_7^{opt}) = 133$. Then we backtrack by using

$$x_i^{opt} \leftarrow x_i(U_i^{opt}), \text{ and } U_{i-1}^{opt} = U_i^{opt} - l_i x_i^{opt}$$

starting with $U_7^{opt} = 10$ and $x_7^{opt} = 1$. The resulting values for U_i^{opt} and x_i^{opt} for $i = 6, 5, \ldots, 1$ are given in the following table:

i	U_i^{opt}	x_i^{opt}
6	9	0
5	9	0
4	9	1
3	8	0
2	8	1
1	3	1

It is noted that no reduction in the number of state values could be achieved for stage 2. For the third and the fourth stages we get some reduction which yields $3 \leq U_3, U_4 \leq 10$. For the subsequent stages, the amount of reduction obtained forms a nondecreasing sequence.

The importance of theorem 3 in achieving the reduction in the refined algorithm is exemplified by the situations encountered in

stages 4, 6, and 7 for $U_4 = 3$, $U_6 = 9$, and $U_7 = 9$ respectively.

The following table gives a comparison between the refined algorithm and the dynamic programming algorithm in terms of the U_i entries required for their application.

Stage i	Number of entries for U_i	
	The Dynamic Programming Algorithm	The Refined Algorithm
1	11	11
2	11	11
3	11	8
4	11	8
5	11	3
6	11	2
7	11	2
Total number of entries for all U_i's	77	45

CHAPTER 5

AN APPLICATION OF THE REFINED ALGORITHM TO SOLVE LORIE-SAVAGE TYPE PROBLEMS WITH EQUALITY CONSTRAINTS

In this chapter, we shall show how the algorithm, developed in chapter 4, can be used to solve the zero-one integer programming problems with more than one constraint. In 1971, Bradley [5] has shown that it is possible to transform any bounded integer programming problem to an equivalent integer problem with a single constraint and the same number of variables. Thus, the single constraint problem can be solved instead of the original problem. Here we briefly review Bradley's work.

In general, integer programming problems have $m(\geq 1)$ constraints. Bradley combines two equality constraints at a time to get a single constraint. Then this new constraint is combined with another constraint and the process continues until all the constraints are combined into one. The idea of solving an equivalent problem instead of the original problem is one of the most powerful notions in integer programming theory. Most of the algebraic algorithms for solving integer programs may be viewed as a process of transforming an integer program to an equivalent integer program that is, in some well defined sense, easier to solve. The other methods given in, for example, [7], [19], [34] and discussed in chapter 1 also transform the original problem into an equivalent problem. But they differ from Bradley's technique in the fact that,

while they add more constraints to the original problem, Bradley's method reduces the number of constraints into one.

To demonstrate Bradley's method of combining constraints, let us consider the following problem with two constraints:

$$\left.\begin{aligned} &\text{maximize } Z_0 = \sum_{i=1}^{n} C_i x_i \\ &\text{subject to } \sum_{i=1}^{n} l_{1i} x_i = b_1 \\ &\qquad\qquad \sum_{i=1}^{n} l_{2i} x_i = b_2 \\ &\qquad\qquad x_i = 0, 1 \qquad i = 1, \ldots, n \end{aligned}\right\} \quad (5.1.1)$$

where l_{1i}, l_{2i}, b_1 and b_2 are integers.

$$\text{Let } SP1 = \max \sum_{i=1}^{n} l_{1i} x_i - b_1$$

$$IF1 = \min \sum_{i=1}^{n} l_{1i} x_i - b_1$$

where $x_i = 0, 1$.

Defining $l_{1i}^{+} = \max \{0, l_{1i}\}$ and $l_{1i}^{-} = \min \{0, l_{1i}\}$, we obtain

$$SP1 = \sum_{i=1}^{n} l_{1i}^{+} x_i^{+} - b_1$$

$$\text{and } \quad IF1 = \sum_{i=1}^{n} l_{1i}^{-} x_i^{+} - b_1$$

where x_i^{+} is the upper bound of x_i, i.e., 1 in our case.

Hence, for zero-one integer programming problems, we have

$$SP1 = \sum_{i=1}^{n} l_{1i}^{+} - b_1$$

and $$IF1 = \sum_{i=1}^{n} l_{1i}^{-} - b_1$$

Finally, we define,

$$SF = \max \{SP1, |IF1|\}$$

It can be noted that in our case

$$SF = \max_{x_i=\{0,1\}} \left| \sum_{i=1}^{n} l_{1i} x_i - b_1 \right|$$

Let α be any integer such that $\alpha > SF$. Then the two constraints in (5.1.1) can be combined in the form

$$\sum_{i=1}^{n} (l_{1i} + \alpha l_{2i}) x_i = b_1 + \alpha b_2 \qquad (5.1.2)$$

to yield the required single constraint of the the equivalent problem

The results of Bradley's work go a long way towards removing Bellman's "curse of dimensionality" [3]*. It should be noted that, since the combination of two constraints is multiplicative in nature, the right hand side of the single constraint (5.1.2), obtained from Bradley's method, will be a large integer.

The solution of the modified problem by using dynamic programming will therefore need a proportionately large amount of computations and storage space. In the case of such large problems, the savings acheived, both in the amount of computations and the storage

* - In dynamic programming, for discrete optimization problems, as the number of constraints increases, the difficulty in computation increases exponentially. Bellman, inventor of dynamic programming called this the curse of dimensionality.

required by the refined algorithm, is very substantial.

This is demonstrated by solving the well-known Lorie-Savage problem with the help of the refined algorithm.

5.1 Reduction in number of variables

We state the following lemma which, when applicable, gives a reduction in the number of variables.

Lemma 5: If in any of the constraints, say the jth constraint, we have $l_{ji} > b_j$ for some i, then the associated variable x_i can be eliminated from the problem.

Proof: The lemma follows immediately from the fact that in such cases the value $x_i=1$ violates the constraint.

5.2 Solution of the Lorie-Savage Problem

In 1955, Lorie and Savage |35| formulated a problem dealing with the optimal investment allocation to projects such that the return from the investment is maximized. Kaplan |30| solved the Lorie-Savage problem by using the generalized Lagrange multiplier technique.

The problem, with the equality constraints, can be stated as:

$$\left.\begin{aligned}
&\text{maximize } Z_0 = 14x_1 + 17x_2 + 17x_3 + 15x_4 + 40x_5 + \\
&\qquad 12x_6 + 14x_7 + 10x_8 + 12x_9 + 15x_{10} \\
&\text{subject to } 12x_1 + 54x_2 + 6x_3 + 6x_4 + 30x_5 + 6x_6 + \\
&\qquad 48x_7 + 36x_8 + 18x_9 + 6x_{10} = 48 \\
&\text{and } 3x_1 + 7x_2 + 6x_3 + 2x_4 + 35x_5 + 6x_6 + \\
&\qquad 4x_7 + 3x_8 + 3x_9 + 7x_{10} = 20 \\
&x_i = 0, 1, \quad i = 1, \ldots, 10
\end{aligned}\right\} \quad (5.2.1)$$

Applying lemma 5 to the above problem, we observe that the variables x_2 and x_5 can be eliminated. Removing these two variables and renumbering the remaining variables, we have the following problem:

$$\left.\begin{aligned}
&\text{maximize } Z_0 = 14x_1 + 17x_2 + 15x_3 + 12x_4 + 14x_5 + \\
&\qquad 19x_6 + 12x_7 + 15x_8 \\
&\text{subject to } 12x_1 + 6x_2 + 6x_3 + 6x_4 + 48x_5 + 36x_6 \\
&\qquad + 18x_7 + 6x_8 = 48 \\
&\text{and } 3x_1 + 6x_2 + 2x_3 + 6x_4 + 4x_5 + 3x_6 + \\
&\qquad 3x_7 + 7x_8 = 20 \\
&x_i = 0, 1 \quad i = 1, \ldots, 8
\end{aligned}\right\} \quad (5.2.2)$$

Let us consider the second constraint. We have

$SP1 = 34 - 20 = 14$

$IF1 = 0 - 20 = -20$

$SF = \max (SP1, |IF1|) = 20$

Thus taking $\alpha = 21$ and using (5.1.2), we may reduce (5.2.2) to the following equivalent single constraint problem:

$$\left.\begin{array}{l} \text{Maximize } Z_0 = 14x_1 + 17x_2 + 15x_3 + 12x_4 + 14x_5 + \\ \qquad 10x_6 + 12x_7 + 15x_8 \\ \text{subject to } 255x_1 + 132x_2 + 128x_3 + 132x_4 + 1012x_5 + \\ \qquad 759x_6 + 381x_7 + 133x_8 = 1028 \\ x_i = 0, 1 \quad i = 1, \ldots, 8 \end{array}\right\} \quad (5.2.3)$$

Rearranging the variables so that $\frac{c_i}{l_i} \geq \frac{c_{i+1}}{l_{i+1}}$, $i = 1, \ldots, 7$, we obtain:

$$\left.\begin{array}{l} \text{Maximize } Z_0 = 17x_1 + 15x_2 + 12x_3 + 14x_4 + 5x_5 + \\ \qquad 12x_6 + 10x_7 + 14x_8 \\ \text{subject to } 132x_1 + 128x_2 + 132x_3 + 255x_4 + 133x_5 + \\ \qquad 381x_6 + 749x_7 + 1012x_8 = 1028 \\ x_i = 0, 1 \quad i = 1, \ldots, 8 \end{array}\right\} \quad (5.2.4)$$

The problem (5.2.4) can now be solved by using the refined algorithm. The optimal solution is given by $Z_0 = 70$ and

$$x_1^{opt} = x_2^{opt} = x_3^{opt} = x_4^{opt} = x_6^{opt} = 1$$

and $x_5^{opt} = x_7^{opt} = x_8^{opt} = 0$

This solution is the same as that obtained by Kaplan.

A comparison between the refined algorithm and the dynamic programming algorithm in terms of the U_i entries required for their

application is given in the following table:

Stage i	Number of entries for U_i	
	The Dynamic Programming Algorithm	The Refined Algorithm
1	1029	1029
2	1029	1029
3	1029	1029
4	1029	1029
5	1029	900
6	1029	637
7	1029	382
8	1029	382
Total number of entries required	8232	6417

CHAPTER 6

THE REFINED ALGORITHM AND THE LAGRANGE MULTIPLIER TECHNIQUE FOR REDUCING DIMENSIONALITY

The Lagrange multiplier and dynamic programming techniques have an important common feature. In both, the original problem with n variables and m constraints is embedded in a space with m + n dimensions. In the Lagrange multiplier method, there is a multiplier for each constraint, while in dynamic programming, each constraint gives rise to a state variable. When it is possible to achieve a dynamic programming formulation, there is the advantage that the m + n dimensional problem can be split up into n subproblems, each having one decision variable and m state variables. On the other hand, in Lagrange multiplier technique, m multipliers are fixed and a series of n-dimensional subproblems are solved each with a different set of values of the multiplier until the desired solution is obtained. When the number of state variables is large, the dynamic programming approach may not be computationally feasible. To get rid of the difficulty caused by high state variable dimensionality, and to preserve the advantage of dynamic programming, Nemhauser [38] introduced the idea that the Lagrange and the dynamic programming approaches can be synthesised by treating some of the constraints with Lagrange multipliers and the remainder with state variables.

Nemhauser uses Everett's GLM generalized Lagrange multiplier technique to reduce the state variable dimension in dynamic

programming. He combines some constraints with the objective functions by using a non-negative multiplier for each of these constraints. He then applies dynamic programming to optimize the new objective function subject to the remaining constraints. Different combinations of λ's are taken and the resulting new problem is solved for every set of λ's by dynamic programming. The process continues and the optimal policy $x_1^{opt}, \ldots x_n^{opt}$ for each problem is tested against the absorbed constraints to determine if the solution is feasible. From amongst the set of all such feasible solutions, the one which yields the maximum return is accepted as the best solution. The method is said to be less reliable but computationally more feasible because of reduced state dimensionality.

In practice, this method is often put to use. It may be emphasized that the reduction of state variable dimensionality creates a large number of dynamic programming problems. This is again a situation in which the refined algorithm will lead to substantial savings in computation and storage requirements.

6.1 The solution scheme for a two constraint problem

In this section we shall transform a two-constraint problem into a single constraint one by absorbing one of the constraints with the objective function by using the Lagrange multiplier technique. Let us consider the problem with two constraints:

$$\left.\begin{aligned} &\text{maximize } Z_0 = \sum_{i=1}^{n} C_i x_i \\ &\text{subject to } \sum_{i=1}^{n} l_{1i} x_i \leq b_1 \\ &\qquad\qquad \sum_{i=1}^{n} l_{2i} x_i \leq b_2 \\ &\qquad\qquad x_i = 0, 1 \end{aligned}\right\} \quad (6.1.1)$$

We absorb the second constraint of (6.1.1) with the help of a non-negative Lagrange multiplier λ. Then we have

$$\left.\begin{aligned} &\text{maximize } Z_0 = \sum_{i=1}^{n} (C_i - \lambda l_{2i}) x_i \\ &\text{subject to } \sum_{i=1}^{n} l_{1i} x_i \leq b_1 \\ &\qquad\qquad x_i = 0, 1 \end{aligned}\right\} \quad (6.1.2)$$

In (6.1.2) each value of λ generates a new problem. We can apply the refined algorithm to solve each new problem thus generated. To apply the refined algorithm, we arrange the variables such that

$$\frac{C_i - \lambda l_{2i}}{l_{1i}} \geq \frac{C_{i+1} - \lambda l_{2,i+1}}{l_{1,i+1}}, \quad i = 1, \ldots, n-1.$$

It is to be noted that the recursive equation is now

$$f_i(U_i)\Big|_{x_i=1} = C_i - \lambda_2 l_i + f^*_{i-1}(U_i - l_i)$$

$$\text{and} \quad f^*_i(U_i) = \text{Max}\left\{ f_i(U_i)\Big|_{x_i=1},\ f^*_{i-1}(U_i) \right\}$$

After achieving the optimal solution for each problem, we put the associated values of the decision variables in the objective

function and the absorbed constraint. Thus we compute

$$Z_0 = \sum_{i=1}^{n} C_i x_i^{opt} \tag{6.1.3}$$

and

$$\sum_{i=1}^{n} l_{2i} x_i^{opt} \tag{6.1.4}$$

If the value given by (6.1.4) satisfies the second constraint of (6.1.1), we note the problem as giving a feasible solution. The value of Z_0 obtained from (6.1.3) gives the corresponding value of the optimum return Z_0.

6.2 A numerical example

To demonstrate the application of the refined algorithm, we consider the following problem which is an extension of problem 4.1, with a second constraint added to it.

Example 6.1:

$$\text{maximize } 60x_1 + 60x_2 + 40x_3 + 10x_4 + 20x_5 + 10x_6 + 3x_7$$

$$\text{subject to } 3x_1 + 5x_2 + 4x_3 + x_4 + 3x_5 + 3x_6 + x_7 \leq 10$$

$$3x_1 + 4x_2 + 4x_3 + 2x_4 + 5x_5 + 4x_6 + 2x_7 \leq 9$$

$$x_i = 0, 1, \quad i = 1, \ldots, 7.$$

The second constraint is absorbed with the objective function with the help of the non-negative Lagrange multiplier λ. The following table gives a list of the solutions for $\lambda = 0.0(.05)1.55$ with

$\lambda = 1.55$, giving rise to the optimal solution.

λ	Values of the decision variables	Value of the right hand side of the second constraint	Value of the objective function
0	1, 1, 0, 1, 0, 0, 1	11	133
.05	1, 1, 0, 1, 0, 0, 1	11	133
.10	1, 1, 0, 1, 0, 0, 1	11	133
.15	1, 1, 0, 1, 0, 0, 1	11	133
.20	1, 1, 0, 1, 0, 0, 1	11	133
.25	1, 1, 0, 1, 0, 0, 1	11	133
.30	1, 1, 0, 1, 0, 0, 1	11	133
.35	1, 1, 0, 1, 0, 0, 1	11	133
.40	1, 1, 0, 1, 0, 0, 1	11	133
.45	1, 1, 0, 1, 0, 0, 1	11	133
.50	1, 1, 0, 1, 0, 0, 1	11	133
.55	1, 1, 0, 1, 0, 0, 1	11	133
.60	1, 1, 0, 1, 0, 0, 1	11	133
.65	1, 1, 0, 1, 0, 0, 1	11	133
.70	1, 1, 0, 1, 0, 0, 1	11	133
.75	1, 1, 0, 1, 0, 0, 1	11	133
.80	1, 1, 0, 1, 0, 0, 1	11	133
.85	1, 1, 0, 1, 0, 0, 1	11	133
.90	1, 1, 0, 1, 0, 0, 1	11	133
.95	1, 1, 0, 1, 0, 0, 1	11	133
1.00	1, 1, 0, 1, 0, 0, 1	11	133
1.05	1, 1, 0, 1, 0, 0, 1	11	133
1.10	1, 1, 0, 1, 0, 0, 1	11	133
1.15	1, 1, 0, 1, 0, 0, 1	11	133
1.20	1, 1, 0, 1, 0, 0, 1	11	133
1.25	1, 1, 0, 1, 0, 0, 1	11	133
1.30	1, 1, 0, 1, 0, 0, 1	11	133

1.35	1, 1, 0, 1, 0, 0, 1	11	133
1.40	1, 1, 0, 1, 0, 0, 1	11	133
1.45	1, 1, 0, 1, 0, 0, 1	11	133
1.50	1, 1, 0, 1, 0, 0, 1	11	133
1.55	1, 1, 0, 1, 0, 0, 0	9	130

From the table we observe that we need to solve 32 problems to arrive at the optimal solution to the problem. For $\lambda = 1.55$, we obtain the optimal solution. The following table gives a comparison of the number of entries for U_i's required in the case of the refined algorithm and for the dynamic programming algorithm.

Stage i	Number of entries of U_i	
	The dynamic programming algorithm	The refined algorithm
1	11	11
2	11	11
3	11	8
4	11	8
5	11	3
6	11	3
7	11	3
Total	77	47

The 32 problems, when solved by using dynamic programming, need a total of 2464 entries for the U_i's, while in the case of the refined algorithm we need only 1504 entries.

CHAPTER 7

CONCLUSIONS

In this study, we have been mainly concerned with the problem of achieving the solution to zero-one integer programming problems with a reduced amount of computational requirements. The algorithm we have developed for this purpose is essentially based on dynamic programming.

From the study it is clear that the reduction in the number of entries for the state values U_i is very effective in achieving a reduction in computational requirements. The quantities U_i^* and U_i^m, as we have pointed out, played very important roles in developing the algorithm.

Of the three theorems developed in chapter 4, theorem 3 enables us to overcome the difficulty in calculating the recursive equation for entries of the state values for which the associated entries in the preceeding stage were out of the range. Thus, the main difficulty in constructing the algorithm is overcome by theorem 3.

Although the algorithm developed is suitable for solving single constraint problems, we have shown that it is possible to apply the algorithm to economically solve problems with more than one constraint.

It should be noted that it is possible to convert a general integer programming problem into a zero-one programming problem by applying an appropriate binary transformation. Thus it is worth

mentioning that the refined algorithm is not limited to zero-one integer programming problems only; it can also be applied to solve general integer programming problems.

BIBLIOGRAPHY

1. ACKOFF, R. L., GUPTA, S.K., and MINAS, J.S., *Scientific Method: Optimizing Applied Research Decisions*, (New York: John Wiley and Sons, 1962).

2. BALAS, E., "An Additive Algorithm for Solving Linear Programs with Zero-One Variables," *Operations Research*, Vol. 13, (1965), pp. 517-546.

3. BELLMAN, R., *Dynamic Programming*, (Princeton, N.J.: Princeton University Press, 1957).

4. BELLMAN, R. and DREYFUS, S., *Applied Dynamic Programming*, (Princeton, N.J.: Princeton University Press, 1962).

5. BRADLEY, G.H., "Transformation of Integer Programs to Knapsack Problems," *Discrete Mathematics*, Vol. 1 (1971), pp. 29-45.

6. CARR, C. R. and HOWE, W. C., *Quantitative Decision Procedures in Management and Economics*, (New York: McGraw-Hill Book Company, 1964).

7. DAKIN, R. J., "A Tree-Search Algorithm for Mixed Integer Programming Problems," *The Computer Journal, (1966)*, pp. 250-255.

8. DANTZIG, G. B., "Discrete Variable Extremum Problems," *Operations Research*, Vol. 5 (1957), pp. 266-277

9. DANTZIG, G. B., *Linear Programming and Extensions*, (Princeton, N.J.: Princeton University Press, 1963).

10. EVERETT, H., "Generalized Lagrange Multiplier Method for Solving Problems of Optimum Allocation of Resources," *Operations Research*, Vol. 11 (1963), pp. 399-417.

11. FAN, L. T. and WANG, C. S., *The Discrete Maximum Principle: A Study of Multistage Systems Optimization*, (New York: John Wiley and Sons, 1964).

12. GARFINKEL, R. S. and NEMHAUSER, G. L., *Integer Programming*, (New York: John Wiley and Sons, 1972).

13. GEOFFRION, A. "Integer Programming by Implicit Enumeration and Balas Method," The Rand Corporation, RM-4783-PR, (February, 1966).

14. GILMORE, P. C. and GOMORY, R. E., "A Linear Programming Approach to the Cutting Stock Problem," *Operations Research*, Vol. 9 (1961), pp. 849-859.

15. GILMORE, P. C. and GOMORY, R. E., "A Linear Programming Approach to the Cutting Stock Problem -- Part II", *Operations Research*, Vol. 11 (1963), pp. 863-888.

16. GLOVER, F. and ZIONTS, S., "A Note on the Additive Algorithm of Balas," *Operations Research*, Vol. 13, (1965), pp. 546-549.

17. GLOVER, F., "A Multiphase-Dual Algorithm for the Zero-One Integer Programming Problem," *Operations Research*, Vol. 13 (1965), pp. 879-919.

18. GOMORY, R. E., "Outline of an Algorithm for Integer solutions to Linear Programs," *Bulletin of the American Mathematics Society*, Vol. 4 (1958), pp. 275-278.

19. GOMORY, R. E., "An All-Integer Programming Algorithm," in J. R. Muth and G. L. Thompson (eds.), *Industrial Scheduling*, (Englewood Cliffs, N.J.: Printice-Hall, Inc., 1963), ch. 13.

20. GOMORY, R. E., "An Algorithm for Integer Solutions to Linear Programs," in R. L. Graves and P. L. Wolfe (eds.), *Recent Advances in Mathematical Programming*, (New York: McGraw-Hill, 1963), pp. 269-302.

21. GLUSS, B., *An Elementary Introduction to Dynamic Programming*, (Boston: Allyn and Bacon, Inc., 1972).

22. GREENBERG, H. and HEGERICH, R. L., "A Branch Search Algorithm for the Knapsack Problem," *Management Science*, Vol. 16, (1970), pp. 327-332.

23. GUE, R. L. and THOMAS, E. M., *Mathematical Methods in Operations Research*, (London: Collier-McMillan Limited, 1968).

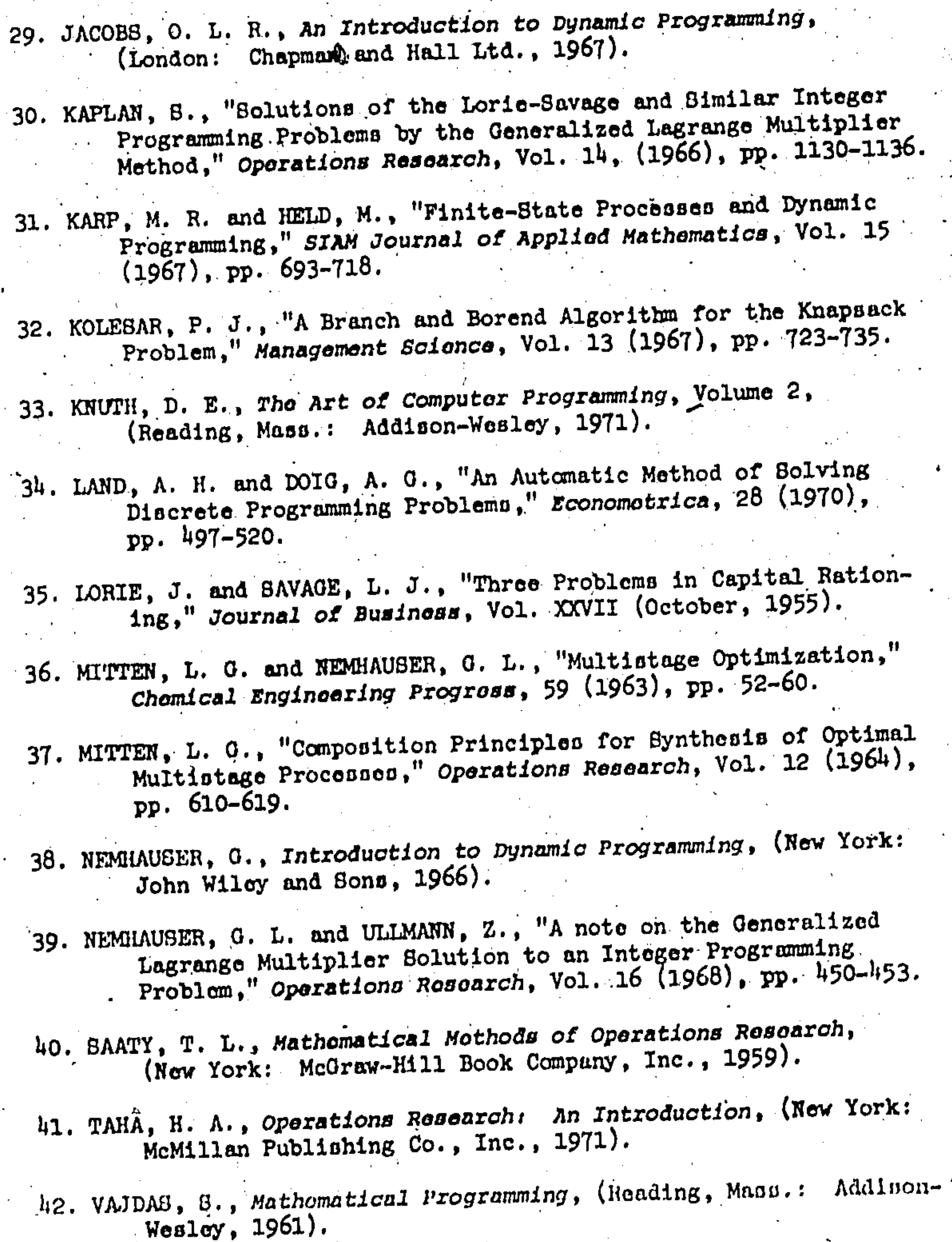
24. HADLEY, G., *Nonlinear and Dynamic Programming*, (Reading, Mass.: Addison-Wesley, 1964).

25. HILLIER, F. and LIBERMAN, G., *Introduction to Operations Research*, (San Francisco: Holden-Day, Inc., 1967).

26. HOWARD, R. A., "Dynamic Programming," *Management Science*, Vol. 12 (1966), pp. 317-348.

27. HU, T. C., "Some Problems in Discrete Optimization," *Mathematical Programming*, Vol. 1 (1971), pp. 102-113.

28. INGARGIOLA, P. G. and KORSH, J. F., "Reduction Algorithm for the Zero-One Single Knapsack Problems," *Management Science*, Vol. 20 (1973), pp. 460-463.



29. JACOBS, O. L. R., *An Introduction to Dynamic Programming*, (London: Chapman and Hall Ltd., 1967).

30. KAPLAN, S., "Solutions of the Lorie-Savage and Similar Integer Programming Problems by the Generalized Lagrange Multiplier Method," *Operations Research*, Vol. 14, (1966), pp. 1130-1136.

31. KARP, M. R. and HELD, M., "Finite-State Processes and Dynamic Programming," *SIAM Journal of Applied Mathematics*, Vol. 15 (1967), pp. 693-718.

32. KOLESAR, P. J., "A Branch and Borend Algorithm for the Knapsack Problem," *Management Science*, Vol. 13 (1967), pp. 723-735.

33. KNUTH, D. E., *The Art of Computer Programming*, Volume 2, (Reading, Mass.: Addison-Wesley, 1971).

34. LAND, A. H. and DOIG, A. G., "An Automatic Method of Solving Discrete Programming Problems," *Econometrica*, 28 (1970), pp. 497-520.

35. LORIE, J. and SAVAGE, L. J., "Three Problems in Capital Rationing," *Journal of Business*, Vol. XXVII (October, 1955).

36. MITTEN, L. G. and NEMHAUSER, G. L., "Multistage Optimization," *Chemical Engineering Progress*, 59 (1963), pp. 52-60.

37. MITTEN, L. G., "Composition Principles for Synthesis of Optimal Multistage Processes," *Operations Research*, Vol. 12 (1964), pp. 610-619.

38. NEMHAUSER, G., *Introduction to Dynamic Programming*, (New York: John Wiley and Sons, 1966).

39. NEMHAUSER, G. L. and ULLMANN, Z., "A note on the Generalized Lagrange Multiplier Solution to an Integer Programming Problem," *Operations Research*, Vol. 16 (1968), pp. 450-453.

40. SAATY, T. L., *Mathematical Methods of Operations Research*, (New York: McGraw-Hill Book Company, Inc., 1959).

41. TAHA, H. A., *Operations Research: An Introduction*, (New York: McMillan Publishing Co., Inc., 1971).

42. VAJDAS, S., *Mathematical Programming*, (Reading, Mass.: Addison-Wesley, 1961).

43. WAGNER, H., *Principles of Operations Research*, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1963).

44. WEINGARTNER, H. M., *Mathematical Programming and the Analysis of Capital Budgeting Problems*, (Englewood Cliffs, N.J., Prentice-Hall, Inc., 1963).

45. WILDE, D. J. and BEIGHTLER, C. S., *Foundations of Optimization*, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1967).