SOLUTION OF THE MULTIPLE-CONSTRAINT ALLOCATION PROBLEM USING RECURSIVE SEARCH DYNAMIC PROGRAMMING

By

MARION LESTER WILLIAMS

Bachelor of Science
Texas A & M College
College Station, Texas
1956

Master of Science
University of New Mexico
Albuquerque, New Mexico
1967

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
DOCTOR OF PHILOSOPHY
May, 1971

SOLUTION OF THE MULTIPLE-CONSTRAINT ALLOCATION

PROBLEM USING RECURSIVE SEARCH

DYNAMIC PROGRAMMING

Thesis Approved:

James E. Shamblin
Thesis Adviser

M. Palmer Terrell

Earl J. Ferguson

Larry M. Perkins

D. Durham
Dean of the Graduate College

PREFACE

The solution of large scale allocation problems is an important factor in the current complex world economy. Decisions that were once made based solely upon subjective judgement must now be aided by powerful mathematical tools. Those factors which influence or control industrial management decisions are sometimes so numerous and complicated that intuition alone cannot be relied upon to render optimum decisions.

The objective of this investigation is to add to the tools available for solution of such problems. The technique developed in this thesis can be used to obtain the solution of many types of integer programming problems, such as the allocation problem, without being restricted by the "curse of dimensionality" which limits the size of problem that can be handled with conventional dynamic programming techniques.

I would like to take this opportunity to express my gratitude to those individuals without whose help and encouragement the attainment of this level of education would not have been possible. Primary among those are the members of my committee, Dr. James E. Shamblin, Dr. Earl Ferguson, Dr. Palmer Terrell, and Dr. Larry Perkins. Dr. Shamblin and Dr. Terrell provided the quantitative insight necessary for my major interest of operations research; Dr. Ferguson contributed wisdom in the art of leadership and management; and Dr. Perkins helped my understanding of real-world problems by tempering my enthusiasm on the quantitative aspects with reminders that humans seldom fit exactly the mold of mathematical symbols so readily fashioned by operations research analysts.

My special thanks to Dr. Shamblin who suggested this thesis topic and provided help and encouragement during its development.

My appreciation also to Margaret Estes for her excellent typing.

Above all, I would like to express my deep gratitude to my wife, Johnnie, and daughters, Tammy and Pamela, for their encouragement and patience during the attainment of this degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

i	index indicating project
j	index indicating time period
A	overall budget constraint
A_i	maximum allocation for the i^{th} project
B_j	maximum allocation for the j^{th} time period
$f_k^*(s_k)$	optimum total return for stages 1 through k for input s_k
$r_i(x_i)$	return function for i^{th} stage with allocation x_i
$r_{ij}(x_{ij})$	return for the i^{th} project in the j^{th} time period for allocation x_{ij}
s_k	input for the k^{th} stage
$\tilde{s}_k$	output for the k^{th} stage
S	state vector
x_i	allocation for the i^{th} stage
x_{ij}	allocation for the i^{th} project in the j^{th} time period
X	allocation vector
Δ	allocation variable incrementing value

CHAPTER I

INTRODUCTION

The allocation problem has received a considerable amount of attention in the literature, as might well be expected. The allocation of resources in order to maximize some kind of return is a fundamental problem in mathematical economics. As such, it is a fruitful area for study by the methods of operations research. Operations research is based in economics; it is the science of getting the most output for the least input -- i.e., optimization, and optimization is measured in terms of the economics of some objective function.

Types of Allocation Problems

Gue and Thomas (1) divide allocation problems into three broad areas. The first type occurs when there are tasks to be performed and there are exactly enough resources to perform the tasks. If each task requires only one resource, it is called an assignment problem. If there are tasks to which more than one resource is required, and if each resource may be used for more than one task, it then becomes a distribution problem. The transportation problem is a specific form of the distribution problem.

A second class of problem concerns the allocation or assignment of resources to activities when there are insufficient resources to satisfy all of the requirements, and one must decide which activities to include in the allocation. In this case, it is a zero-one problem in that activities are either included or excluded.

In the third type of problem, it is possible to control not only which activities are to be included, but also the level of resource that will be allocated to each of the activities.

This thesis is concerned with the third type of allocation problem, which may be described as follows:

Given a limited quantity of resource, such as money, time, materials, machines, etc., it is desired to distribute this resource in an optimum manner among competing activities, such as projects, products, etc. For each activity, the allocation of a quantity of resource provides a return of some kind. This return, or utility function, may be a linear or non-linear function of the amount of resource allocated to that activity.

Examples of Allocation Problems

Allocation problems of many forms arise in business and industry. The basic allocation problem considered in most texts is the "knapsack" problem. This general type of problem is aimed at determining the optimum loading of cargo, weapons, etc., in order to maximize return, whether the return is profit, damage potential, or some other measure of utility. These problems are usually referred to as one-dimensional, since only one resource is considered and there is a single constraint, such as volume or weight.

More complicated problems arise when there are multiple constraints because of several resources to be allocated, or because of several constraints on the allocation of a single resource.

The transportation and distribution problem are forms of the allocation problem with multiple constraints. In the transportation problem, it is desired to determine the least expensive routing system for shipping goods between shipping points and demand points. The distribution problem considers the optimum placement of goods or services at various facilities.

One of the important allocation problems with multiple constraints is that of budgeting and project selection. In this general type of problem, there are limited resources that must be divided among competing projects. There may be limitations on the amount of resource that can be given to a single project, as well as limitations on the amount of resources available in any given time period. Baker and Yormark (2) refer to this as the allocation problem with two-dimensional constraints. Two-dimensional refers to the fact that there are constraints on two entities, such as projects and time periods.

As an example, a manufacturer may produce automobiles and boats, each requiring a specific amount of a raw material such as steel. Since both products are to be produced, there is a limit as to the amount of steel that can be given to each production line. Also, since steel is provided to the manufacturer over a period of time, there may be limitations as to the amount of steel available to both production lines during any given time period. Because of seasonal variations, the return (profit) to the manufacturer may be a function of the time period; i.e., period of year, as well as the type of product. Additionally, the market can become saturated with either of these products, so that the return may not be a linear function of the amount produced, which complicates the problem even further. Thus, determining the optimum allocation for each production line and time period is not a simple problem.

A mathematically similar problem is that of portfolio selection, where a limited amount of money is available for investment in each of several time periods. In addition to the time period constraints, there may also be constraints on the type of investment, such as a limitation on the investment in a particular industry, or limitations on the general types of investments, etc.

There are innumerable other examples of allocation problems. In fact, many problems that at first appear to be totally unrelated can be shown to be a form of the allocation problem, or can be formulated and solved as such. For example, a linear or non-linear programming problem can be formulated as an allocation problem where a resource is to be "allocated" to each of the variables, and the amount of resource is governed by the problem constraints.

Mathematical Formulation

The allocation problem may be mathematically formulated as follows:

$$\text{Maximize } R(X) = \sum_{i=1}^{n} r_i(x_i)$$

subject to: (1-1)

$$\sum_{i=1}^{n} c_{ij} x_i \leq A_j \qquad j = 1, 2, \ldots, m$$

where $r_i(x_i)$ is the return obtained from the i^{th} of n activities when an amount of resource x_i is allocated to that activity. There are m constraints, each constraint controlled by an allocation amount A_j.

In those cases where the return (or utility) functions are linear, the solutions can usually be obtained through one of several mathematical programming techniques. The problem becomes more complex when the return functions are non-linear, although techniques are available which make them tractable, such as Beale's algorithm when the objective function is quadratic (1). In some instances, linear approximations to the objective function can be used and an approximate solution obtained using linear programming techniques. However, the linearized versions are usually inadequate.

The introduction of an additional requirement for integer solutions eliminates most available mathematical programming techniques. Exhaustive search is a possible, but very expensive, alternative. An approach often suggested is to assume a continuous problem, obtain a solution, then round or truncate to an integer solution. Unfortunately, the solution obtained in this manner is usually infeasible and/or non-optimal.

There have been various approaches to the solution of the different types of allocation problems. Some of the original techniques for the solution of linear versions of Equation (1-1) were developed by Koopmans (3). The capital budgeting version of the allocation problem was attacked through Lagrange multipliers by Lorie and Savage (4). Weingarten (5) applied integer programming. However, Nemhauser (6) concluded that dynamic programming provided the most efficient technique when there are not more than three constraints.

A survey of various approaches to the capital budgeting allocation problem is contained in Weingarten (7).

Solution by Dynamic Programming

Most of the work on allocation problems with integer solutions has been accomplished with dynamic programming. Examples are contained in Gue and Thomas (1) and Hillier and Lieberman (8). Unfortunately, this approach can be used only if there are few constraints. When there are several constraints, usually more than two or three, the number of calculations and size of computer memory required prohibit the use of this technique. This results from the fact that computer memory requirements increase exponentially with the number of problem constraints. This is referred to by Bellman as the "curse of dimensionality" (9).

The technique proposed by this thesis circumvents the limitations of conventional dynamic programming through the use of a recursive search technique. This technique eliminates the need for large computer memory which usually makes the solution of large scale problems impossible.

CHAPTER II

THE RESOURCE ALLOCATION PROBLEM

The general form of the resource allocation problem is given by Equation (1-1). When there is only one constraint, the problem may be written in the following form:

$$\text{Maximize } R(X) = \sum_{i=1}^{n} r_i(x_i)$$

subject to: (2-1)

$$\sum_{i=1}^{n} x_i \leq A \quad .$$

This particular form is referred to in the literature as the Lori-Savage model, since it was discussed originally by Lorie and Savage (4). Wagner (10) refers to this as the when-or-where model. This title comes from the fact that the Lorie-Savage model has several interpretations from an allocation standpoint. The usual definition is that there are n projects (products, etc.) and it is desired to maximize the return given by Equation (2-1) when an amount of resource A is distributed among these projects during a single time period, or single planning horizon. By a redefinition of terms, it can be considered as a problem of allocating an amount of resource A among the n time periods of a single project. Since only one constraint is present, this is a one-dimensional allocation problem.

Although the problem description has been in terms of projects and time periods, it could have easily been defined as availability and requirements in a transportation problem, or in many other terms. Throughout this thesis, the problem will be described as one of allocating resources over projects and time periods, recognizing the many other possible interpretations of this model.

Multiple-Constraint Problems

Generally, the allocation problems solved in textbooks are of the form given by Equation (2-1); i.e., single constraint or one-dimensional problems. This type of problem can be easily solved with dynamic programming, which is the most efficient approach when the solution is constrained to integer values. However, the problem takes on a different character when there are several constraints, such as the general allocation model given by Equation (1-1). Although dynamic programming is still the best approach for problems of this nature, the "curse of dimensionality" mentioned earlier limits the size of problem that can be handled.

As a specific example of a multiple-constraint problem, consider the project selection analysis studied by Baker and Yormark (2). As discussed earlier, in this situation, there are several projects and time periods, with varying budget constraints on both entities. This particular problem will be used as a model to demonstrate the recursive search technique.

Although the problem description has been in terms of projects and time periods, it could have easily been defined as availability and requirements in a transportation problem, or in many other terms. Throughout this thesis, the problem will be described as one of allocating resources over projects and time periods, recognizing the many other possible interpretations of this model.

Multiple-Constraint Problems

Generally, the allocation problems solved in textbooks are of the form given by Equation (2-1); i.e., single-constraint or one-dimensional problems. This type of problem can be easily solved with dynamic programming, which is the most efficient approach when the solution is constrained to integer values. However, the problem takes on a different character when there are several constraints, such as the general allocation model given by Equation (1-1). Although dynamic programming is still the best approach for problems of this nature, the "curse of dimensionality" mentioned earlier limits the size of problem that can be handled.

As a specific example of a multiple-constraint problem, consider the project selection analysis studied by Baker and Vormerk (2). As discussed earlier, in this situation, there are several projects and time periods, with varying budget constraints on both entities. This particular problem will be used as a model to demonstrate the recursive search technique.

Mathematical Model

The mathematical formulation of the allocation problem with constraints on two entities is given by:

$$\text{Maximize } R(X) = \sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij}(x_{ij})$$

subject to:

$$\begin{aligned}
&\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \leq A \\
&\sum_{j=1}^{m} x_{ij} \leq A_i \qquad i = 1, 2, \ldots, n \\
&\sum_{i=1}^{n} x_{ij} \leq B_j \qquad j = 1, 2, \ldots, m \\
&x_{ij} \geq 0 \text{ for all } i, j \\
&x_{ij} \text{ integers}
\end{aligned} \qquad (2\text{-}3)$$

where, for the project selection problem:

A = total budget constraint

A_i = budget constraint for the i^{th} project

B_j = budget constraint for the j^{th} time period

x_{ij} = amount of resource allocated to the i^{th} project in the j^{th} time period

$r_{ij}(x_{ij})$ = return from allocation x_{ij}

m = number of time periods

n = number of projects.

In this model, it is desired to maximize the return from allocation of a resource to specific projects and time periods. There are n projects and each project can be allocated no more than A_i of the resource. In addition, the projects will last a maximum of m time periods, and during any one time period the resource allocation to all projects must not exceed B_j. As an overall constraint, the total amount of resource available is A. For each project-time period there are discrete feasible funding levels, so that the x_{ij} must take on integer values corresponding to these levels. This is, therefore, an integer programming problem. This problem is shown in Figure 1.

Assumptions

As mentioned previously, this type of problem is difficult to solve by any method, but the most promising approach is dynamic programming. As with all methods for the solution of complex problems, certain assumptions are necessary. For this problem, the following assumptions are made:

(1) The return from different activities (where here an activity is a project-time period) can be measured in common units.

(2) The total return from any activity is independent of the allocations to the other activities.

(3) The total return can be obtained as the sum of the individual returns.

(4) The return functions are concave.

The first three assumptions are necessary to apply the dynamic programming technique. The last assumption is necessary to use the recursive search technique proposed by this thesis. This technique

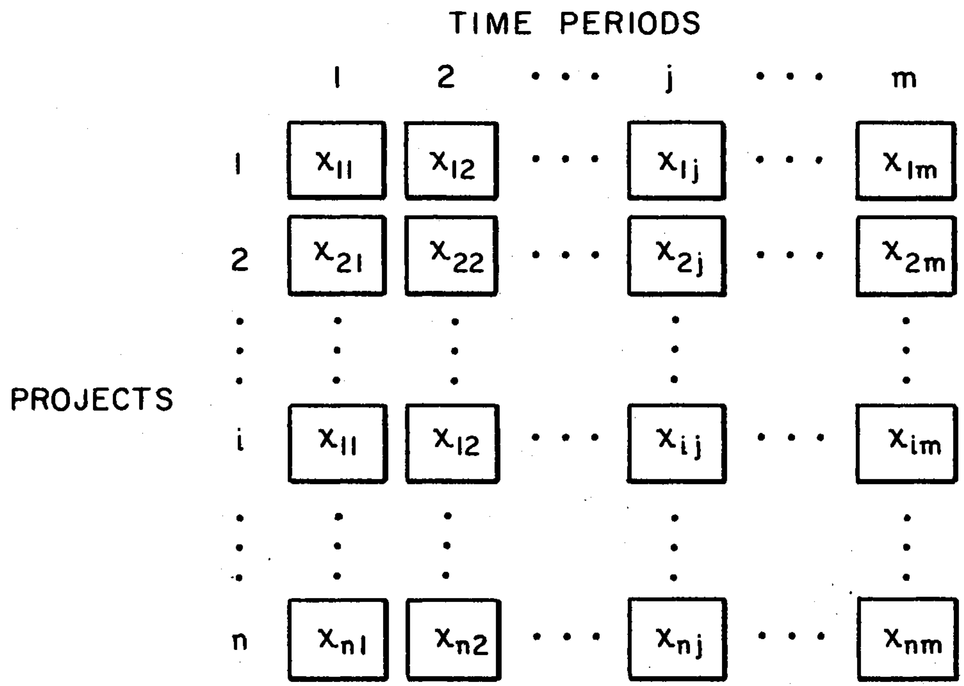


Figure 1. Allocation of Resources Over Projects and Time Periods

makes an exact solution of Equation (2-3) possible within the limits of present day computers.

Before discussing the details of the solution to Equation (2-3), it is necessary to briefly review dynamic programming as a basis for the solution developed in this thesis.

CHAPTER III

DYNAMIC PROGRAMMING

The theory and application of dynamic programming are discussed fully in several texts, such as Bellman (9), who developed the concept, Bellman and Dreyfus (11) and Nemhauser (12). There are also reports which discuss the specific problem of allocation of resources and solution using dynamic programming, such as Dreyfus (13) and Kalaba (14). These sources should be referred to for complete details; the following description is presented only as a basic review of dynamic programming and to establish the notation that will be used in the remainder of the thesis.

Dynamic programming is an approach to the solution of multistage decision problems which transforms these problems into a series of single stage problems. Dynamic programming can be applied to a wide variety of problems. It is more of a concept than a specific technique, and for this reason it is difficult to develop an algorithm which can be used to solve many types of problems; each problem must be specifically modeled for solution by this technique.

Principal of Optimality

Decomposition of a multistage decision problem is accomplished through mathematical formulation of Bellman's "principal of optimality" which states (9):

> An optimal policy has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

This says, in effect, that the optimum decision is one in which all subsequent decisions are optimum with respect to the state resulting from the previous decision.

Dynamic Programming Notation

The usual method of depicting a dynamic programming problem is shown in Figure 2, where the stages of the problem are numbered in reverse order in accordance with convention.

In Figure 2, the state variables and decision variables for the i^{th} stage are denoted by s_i and x_i, respectively. State variables represent the state or condition of the system at a particular point within the problem solution; i.e., at a particular stage. State variables are usually those conditions not under the control of the decision maker. The input state, s_i, is the value of the state variable entering the i^{th} stage. The output state, $\tilde{s}_i$, is the value after the decision x_i has been made. As can be seen in this figure, the output of the i^{th} stage is the input to the $(i-1)^{st}$ stage.

Decision variables, denoted by x_i, are those variables that are under the control of the decision maker.

The return function, $r_i(s_i, x_i)$, represents the return of the i^{th} stage where the input is s_i and the decision made at this stage is x_i. The state transformation function, $t_i(s_i, x_i)$, determines the value of the state variable at the $(i-1)^{st}$ stage as a function of the state and decision variable at the previous stage. That is, for a given input

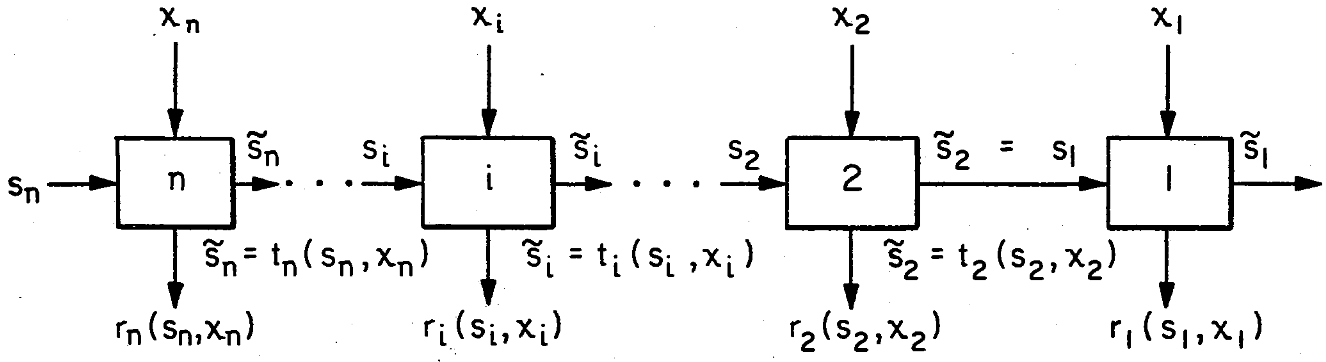


Figure 2. Dynamic Programming Notation

state and decision, the transformation function determines the output state for that stage.

Recursive Relationships

Now define the following:

$f_k(s_k, x_k)$ = the total return from stages 1 through k (k stages remaining) when the input state is given by s_k and decision x_k is made with optimum decisions made for the output state $\tilde{s}_k$ in stages 1 through $k-1$.

$f_k^*(s_k)$ = the optimum total return for stages 1 through k for the input state s_k.

Then for any stage k, Bellman's principal of optimality may be mathematically formulated as follows:

$$f_k^*(s_k) = \max_{x_k} f_k(s_k, x_k) \tag{3-1}$$

$$= \max_{x_k} [r_k(s_k, x_k) + f_{k-1}^*(s_{k-1})] \tag{3-2}$$

for

$$k = 1, 2, \ldots, n$$

where

$$f_o^*(s_o) \equiv 0,$$

where the input to the $(k-1)^{st}$ stage is determined from the transformation function:

$$s_{k-1} = \tilde{s}_k = t_k(s_k, x_k) \,. \tag{3-3}$$

Dynamic Programming Solution of the One-Dimensional Allocation Problem

With the above definitions, consider the dynamic programming approach to the one-dimensional allocation problem. As described previously, this is an allocation problem where there is one type of resource and one constraint, such as the following formulation of the Lorie-Savage model:

$$\text{Maximize } R(X) = \sum_{i=1}^{n} r_i(x_i)$$

subject to: (3-4)

$$\sum_{i=1}^{n} x_i \leq A$$

$$x_i \geq 0; \text{ integers .}$$

In problem solving with dynamic programming, the first step is the definition of stages, states and decisions. For the allocation problem, the stages correspond to the activities. The decisions then correspond to the amount of resource allocated at each stage (or activity), and the state variables represent the amount of resource remaining that could be allocated at each stage. If the problem is considered as allocating a portion of A at each stage, it can be seen that the constraint yields a transformation function:

$$s_{k-1} = s_k - x_k \quad . \tag{3-5}$$

The recursive equation, or functional relationship, of the principal or optimality for this problem is then given by:

$$f_k^*(s_k) = \max_{x_k \leq s_k} [r_k(x_k) + f_{k-1}^*(s_{k-1})] \tag{3-6}$$

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

where $f_o^*(s_o) \equiv 0$. (Note that since the return is a function only of the amount of resource allocated, it may be written as $r_k(x_k)$ instead of $r_k(s_k, x_k)$.)

Using the transformation function, Equation (3-5), Equation (3-6) becomes:

$$f_k^*(s_k) = \max_{x_k \leq s_k} [r_k(x_k) + f_{k-1}^*(s_k - x_k)] \tag{3-7}$$

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

where $f_o^*(s_o) \equiv 0$.

Notice that for a n stage problem, the optimum value for all stages is given by:

$$f_n^*(s_n) = f_n^*(A) \quad . \tag{3-8}$$

Computational Aspects of Dynamic Programming

For each stage of the dynamic programming process, it is necessary to calculate $f_k(s_k, x_k)$ for each feasible x_k and s_k, and then from these values, to determine the value of x_k which maximizes $f_k(s_k, x_k)$ to yield $f_k^*(s_k)$ for each s_k. Therefore, for state transformation functions given by Equation (3-5), if there are v feasible input states for each stage, then for n stages, there are approximately $\frac{1}{2}nv^2$ evaluations of Equation (3-7) required to determine the optimum allocation. Although this may seem to be a large number, compare this to the v^n calculations required for exhaustive search!

If this problem is to be solved on a digital computer (a necessity for large problems), an important factor is the required size of core memory. This can be determined as follows: At each stage in the dynamic programming solution, it is necessary to save the optimum value of Equation (3-7), and also the decision variable that yielded the optimum value, for each input state. However, $f_k^*(s_k)$ is needed only until $f_{k+1}^*(s_{k+1})$ is calculated. Again assuming n stages with v feasible values of s_k at each stage, the total memory requirement, not including memory for the program statements, is v(n+2) storage locations. Obviously quite large one-dimensional problems can be solved using large computers. However, it will be demonstrated later that the memory requirements mushroom when problems with several constraints are encountered.

Numerical Example

As an example of dynamic programming solution to a one-dimensional allocation problem, consider a single project, four time period optimization problem given by:

$$\text{Maximize } R(X) = \sum_{i=1}^{4} r_i(x_i)$$

subject to: (3-9)

$$\sum_{i=1}^{4} x_i \leq 10$$

$$x_i \geq 0, \quad \text{integers}$$

where the return functions, $r_i(x_i)$ are given in Table I. These return functions are of the form:

$$r_i(x_i) = \frac{ax_i}{bx_i + c} \quad . \tag{3-10}$$

The first two derivatives of Equation (3-10) are:

$$r_i'(x_i) = \frac{ac}{(bx_i + c)^2} \tag{3-11}$$

$$r_i''(x_i) = \frac{2b^2c + 2abc^2}{(bx_i + c)^4} \quad . \tag{3-12}$$

From these equations, the maximum occurs at $x = \infty$, and from Equation (3-12) the function is concave for all positive a, b and c. Thus, these return functions meet the assumptions of Chapter II.

The recursive equation for the first stage of the dynamic programming solution to this problem is given by:

$$f_1^*(s_1) = \max_{x_1 \leq s_1} r_1(x_1) \quad . \tag{3-13}$$

The first stage returns are given in Table II for each feasible input state. At the right side of the table are the optimum returns and decisions from this stage as a function of the input state. For a computer solution of this type problem, only the values in the last two columns need to be saved, and $f_1^*(s_1)$ is needed only until $f_2^*(s_2)$ is calculated.

Table III contains the returns from the first and second stages, obtained from the second stage recursive equation:

$$f_2^*(s_2) = \max_{x_2 \leq s_2} [r_2(x_2) + f_1^*(s_2 - x_2)] \quad . \tag{3-14}$$

TABLE I

RETURN FUNCTIONS FOR NUMERICAL EXAMPLE

x_i	Return			
	$r_1(x_1)$	$r_2(x_2)$	$r_3(x_3)$	$r_4(x_4)$
0	0	0	0	0
1	2619	3529	1244	1274
2	3437	3810	2074	2062
3	3837	3913	2667	2597
4	4074	3970	3111	2985
5	4231	4000	3457	3279
6	4342	4022	3733	3509
7	4425	4039	3960	3694
8	4490	4051	4148	3846
9	4541	4060	4308	3974
10	4583	4068	4444	4082

TABLE II

FIRST STAGE RECURSIVE ANALYSIS

s_1 \ x_1	0	1	2	3	4	5	6	7	8	9	10	$f_1^*(s_1)$	x_1^*
0	0											0	0
1	"	2619										2619	1
2	"	"	3438									3438	2
3	"	"	"	3837								3837	3
4	"	"	"	"	4074							4074	4
5	"	"	"	"	"	4231						4231	5
6	"	"	"	"	"	"	4342					4342	6
7	"	"	"	"	"	"	"	4425				4425	7
8	"	"	"	"	"	"	"	"	4490			4490	8
9	"	"	"	"	"	"	"	"	"	4541		4541	9
10	"	"	"	"	"	"	"	"	"	"	4853	4853	10

TABLE III

SECOND STAGE RECURSIVE ANALYSIS

s_2 \ x_2	0	1	2	3	4	5	6	7	8	9	10	$f_2^*(s_2)$	x_2^*
0	0											0	0
1	2619	3259										3259	1
2	2438	6148	3810									6148	1
3	3837	6967	6429	3913								6967	1
4	4074	4367	7247	6532	3967							7247	2
5	4231	7603	7647	7351	6586	4000						7647	2
6	4342	7660	7884	7750	7404	6619	4022					7884	2
7	4425	7872	8040	7987	7804	7438	6641	4039				8040	2
8	4490	7995	8152	8144	8041	7837	7460	6658	4051			8152	2
9	4541	8019	8235	8255	8198	8074	7860	7476	6670	4060		8255	3
10	4583	8071	8299	8338	8309	8231	8096	7876	7488	6679	4068	8338	3

Again for this table, the optimum return and decision for each input state are shown in the last two columns.

Similarly, Tables IV and V contain the return for the third and fourth stages, respectively. The fourth stage contains the total return from all four stages as a function of the input state. From this table, it can be seen that the maximum possible return is 12,675.

In order to determine the allocation which yielded this optimum return, it is necessary to trace back through the stages using the state transformation function, Equation (3-5). These calculations, as shown in Table V, given an optimum allocation $X^* = (2,1,4,3)$. Thus the optimum return for this project is 12,675 for an allocation of two units in time period one, one unit in time period two, four units in time period three, and three units in time period four. Any other allocation, where the allocation is restricted to integer values, would yield a lower return.

Dynamic Programming Solution of the Multiple Constraint Allocation Problem

As seen from the above example, the one-dimensional allocation problem is straightforward and can be readily solved with dynamic programming. As mentioned previously, this is the most efficient means of solution when the solution is restricted to integer values. However, now consider the same problem as before, but add constraints on time periods as well. The problem now becomes:

TABLE IV

THIRD STAGE RECURSIVE ANALYSIS

s_3 \ x_3	0	1	2	3	4	5	6	7	8	9	10	$f_3^*(s_3)$	x_3^*
0	0											0	0
1	3529	1244										3529	0
2	6148	4774	2044									6148	0
3	6967	7393	5604	2667								7393	1
4	7367	8211	8223	6196	3111							8223	2
5	7648	8611	9041	8815	6641	3457						9041	2
6	7884	8892	9441	9634	9260	6986	3733					9634	3
7	8040	9128	9722	10333	10780	9605	7263	3960				10780	4
8	8152	9285	9958	10314	10478	10424	9882	7489	4198			10478	4
9	8255	9396	10114	10550	10759	10823	10700	10108	7678	4308		10823	5
10	8338	9416	10226	10707	10995	11104	11100	10927	10287	7837	4444	11104	5

TABLE V

FOURTH STAGE RECURSIVE ANALYSIS

s_4 \ x_4	0	1	2	3	4	5	6	7	8	9	10	$f_4^*(s_4)$	x_4^*
10	11104	12097	12540	12675	12619	12320	11731	11087	10005	7503	4082	12675	3

Optimum allocation: $x_1 = 2$

$x_2 = 1$

$x_3 = 4$

$x_4 = 3$

Optimum return $= f_4^*(s_4) = 12675$

TABLE VI

OPTIMUM DECISIONS FOR NUMERICAL EXAMPLE

Stage	Input State s_i	Decision x_i	Output State $s_i - x_i$
4	10	3	7
3	7	4	3
2	3	1	2
1	2	2	0

$$\text{Maximize } R(X) = \sum_{i=1}^{n} r_i(x_i)$$

subject to:

$$\sum_{i=1}^{n} x_i \leq A$$

$$x_j \leq B_j \qquad j = 1, 2, \ldots, m \tag{3-15}$$

$$x_j \geq 0 \text{ , integer .}$$

In the dynamic programming formulation of the one-dimensional allocation model, the state variable represented the slack in the constraint -- the amount of unallocated resource -- at each stage in the solution. The state variable is also the slack in the constraints of Equation (3-15); however, since there are now $m+1$ constraints,[1] the state variable is now a vector with $m+1$ components. As in the previous problem, it is necessary to calculate the return for all feasible decisions and state variables. For the multiple constraint problem, however, the number of feasible states has increased significantly, since each combination of the $m+1$ components of the state vector represents a feasible state. If there are v feasible values of each of the $m+1$ components of the state vector, then the amount of storage space required to solve an n stage problem is approximately $v^{m+1}(n+2)$ storage locations. If there are n projects to be considered as well, then the storage requirements are approximately $v^{m+n+1}(n+2)$.

---

[1]The non-negativity constraints are not included in this number. Since the problem can be structured such that only positive allocations are considered, the non-negativity constraints do not increase the dimension of the problem.

As an example, consider the problem where there are four competing projects, and it is desired to obtain the optimum allocation for these projects for each of ten time periods. Assuming ten feasible values of each component of the state vector at each stage; i.e., ten feasible funding levels at each time period for each project, then the storage requirement is approximately 10^{15} locations. Obviously a problem of even this modest size could not be handled with present day computers, which have internal storage on the order of 10^6 locations. Of course, external memory could be used, but at a significant reduction in computational speed. This is a rather minor point, however, since the time required to perform the calculations necessary just to fill these storage spaces, assuming 10^6 calculations per second, is on the order of a century. There is little consolation in the fact that 10^{40} calculations are required to determine the optimum solution with exhaustive enumeration.

Obviously, conventional dynamic programming has severe limitations. Under certain conditions, however, these limitations can be overcome, as will be discussed in the next chapter.

CHAPTER IV

RECURSIVE SEARCH DYNAMIC PROGRAMMING

As discussed previously, the dynamic programming formulation of large allocation problems with several constraints requires more storage space than is available in even the largest computers. To reduce the storage requirements, various approaches have been investigated. Bellman (11) discusses the use of a polynominal approximation to the recursive equations. With this procedure, only the coefficients of the polynominal are stored, and interpolation is used to obtain values of the recursive equation at specific points.

Kalaba (14) uses Lagrange multipliers in conjunction with dynamic programming to reduce the number of constraints in the problem and, thus, reduce the dimension. However, neither polynominal approximation nor the Lagrange multipliers provides an efficient method of getting around the problem.

Various search techniques can be used when the return functions are unimodal. However, the search techniques discussed in the literature are not as efficient nor as easily programmed as desired; especially when vector state variables are involved.

One of the more recent and comprehensive investigations in the area of solution of the allocation problem with multiple constraints is reported in the previously-mentioned reference by Baker and Yormark (2). In this paper, a capital budgeting problem is investigated in which there are non-linear return functions, integer solutions, and several constraints. However, only an approximation to the optimum solution was obtained. Baker and Yormark also discuss related works by Hess (15) and Rosen and Souder (16) which formulate a research and development project selection problem, a form of the capital budgeting problem. In each case, the inherent limitations of conventional dynamic programming prevented obtaining exact solutions in an efficient manner.

This problem can be solved, however, with a modification of dynamic programming. This technique, referred to as recursive search dynamic programming, considerably reduces the computer storage requirements as well as the number of calculations necessary to obtain an optimum solution. Basically, the recursive search technique starts with a feasible solution, then searches over each of the recursive relationships until an optimum solution is reached. If the return functions are concave, then the solution is a global optimum.

Computational Advantages of Recursive Search

The recursive search method of dynamic programming provides an efficient means of solution of many forms of the allocation problem. With this technique, only a limited number of states and decision variables in each stage need to be investigated, so that computational time and computer memory requirements are significantly reduced. As will be seen later, the number of calculations required to reach the optimum solution by this technique is a function of the starting solution and only in a worse case condition approaches the number required by the conventional method. (For worse case conditions;

i.e., starting solution at one extreme boundry of the constraints and the optimum solution at the other extreme boundry, the recursive search calculates all values necessary for the conventional method.) In trial problems using this technique, the number of calculations was a small fraction of that required using the conventional method.

This technique utilizes a feasible starting solution which implicitly defines the state vector for each stage, so that it is not necessary to calculate the values of the state vector. A search procedure is then utilized which successively optimizes each recursive equation until a global optimum is reached.

The computer algorithm was originally developed to handle problems such as given by Equation (2-3); however, with modifications to the program, it can also handle various other types of problems, such as the manpower leveling problem.

Description of the Recursive Search Technique

First consider the allocation problem with constraints on two entities, such as projects and time periods in the case of the R & D budgeting problem. To obtain a form more compatible with the usual dynamic programming formulation, Equation (2-3) can be written with single subscripted variables with no loss of generality as follows:

$$\text{Maximize } R(X) = \sum_{i=1}^{N} r_i(x_i)$$

subject to: (4-1)

$$\sum_{i=1}^{N} \delta_{ij} x_i \leq A_j \qquad j = 1, 2, \ldots, M$$

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

where $N = m \times n$ and $M = m + n + 1$ so that there are the same number of variables and constraints as in Equation (3-2), and where each $\delta_{ij} = 0$ or 1 to account for the fact that all x_i's do not appear in every constraint.

To solve Equation (4-1) by dynamic programming, let the N variables x_i, x_2, ..., x_N correspond to the stages of the usual dynamic programming formulation. The decision variables are then the amount of resource to allocate at each stage. The states correspond to the amount of resource remaining to be allocated; i.e., the slack, and since there are M constraints, the state variable must be an M-dimensional vector. The k^{th} member of the state vector is the amount of slack in the k^{th} constraint.

Let S_i be the input state vector variable at stage i, and let s_{ij} represent the j^{th} component of that vector. Then s_{32}, for example, is a component of the vector S_3 and represents the amount of slack in the second constraint at the beginning of the third stage.

The state transformation resulting from the constraints of Equation (4-1) is given by:

$$S_{i-1} = T_i(S_i, x_i) \tag{4-2}$$

$$= (s_{i1} - \delta_{i1}x_i, s_{i2} - \delta_{i2}x_i, \ldots, s_{iM} - \delta_{iM}x_i) \tag{4-3}$$

or, letting

$$D_i = (\delta_{i1}, \delta_{i2}, \ldots, \delta_{iM}) \tag{4-4}$$

Equation (4-3) can be written:

$$S_{i-1} = (S_i - D_i x_i) \ . \tag{4-5}$$

With these definitions, the dynamic programming problem may be diagrammed as shown in Figure 3. In this figure, the input to the N^{th} stage is given by the amount of resource remaining (slack) in each constraint, and since nothing has been allocated at this point, S_N is given by:

$$S_N = (A_1, A_2, \ldots, A_M) \quad . \tag{4-6}$$

Thus, the slack at each stage is given by:

$$s_{ij} = A_j - \sum_{k=i+1}^{N} \delta_{jk} x_k \quad . \tag{4-7}$$

Now let

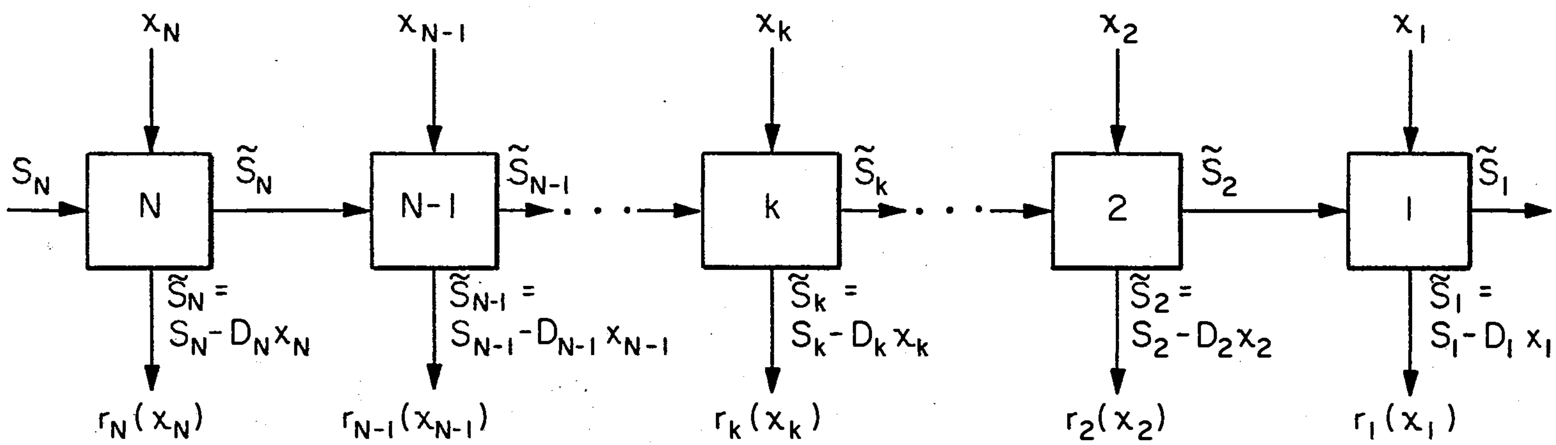
$$f_k^*(S_k) = \max_{x_k \leq \min S_k} f_k(S_k, x_k) \tag{4-8}$$

represent the return obtained by optimally allocating the resource represented by the state vector S_k over variables 1 through k, where min S_k indicates the minimum component of vector S_k. Then the dynamic programming principal of optimality is given by the recursive relationship:

$$f_k^*(S_k) = \max_{x_k \leq \min S_k} [r_k(x_k) + f_{k-1}^*(S_k - D_k x_k)] \tag{4-9}$$

where $f_o^*(S_1 - D_1 x_1) \equiv 0$.

With conventional dynamic programming it would be necessary to determine the optimum value of each decision variable, x_i $i = 1, 2, \ldots, N$, for each feasible input state. As discussed earlier, this would require storing approximately $(N+2)v^M$ values, so that a problem with a



$$\tilde{S}_k = S_k - D_k x_k = (s_{k1} - \delta_{k1} x_k,\ s_{k2} - \delta_{k2} x_k, \cdot \cdot \cdot, s_{kM} - \delta_{kM} x_k)$$

Figure 3. Dynamic Programming Formulation of the Allocation Problem with Multiple Constraints

modest number of constraints can easily exceed the memory capacity of the largest computer.

Now assume a starting solution $X = (x_1, x_2, \ldots, x_N)$ such that X satisfies the M constraints given in Equation (4-1). Resources are allocated by stage, beginning with stage N in the regular (backward recursive) dynamic programming manner. The input to the i^{th} stage (output of the $(i+1)^{st}$ stage) is given by the state transformation function, Equation (4-3), which, using Equation (4-7) may be written as:

$$S_i = (A_1 - \sum_{k=i+1}^{N} \delta_{ik} x_k, \ldots, A_M - \sum_{k=i+1}^{N} \delta_{ik} x_k) . \qquad (4\text{-}10)$$

Now the input vector to the N^{th} stage, S_N, is given by Equation (4-6). Since x_N is defined by the starting solution X, the output of the N^{th} stage (which is also the input to the $(N-1)^{st}$ stage, S_{N-1}) is defined by the state transformation function, Equation (4-3). Likewise, S_{N-1} and x_{N-1} specify the input state vector to the $(N-2)^{nd}$ stage, etc. Thus, with X defined, the input state vector to each of the N stages is specified.

Although X defines a feasible solution to Equation (4-1), it is not necessarily the optimum solution. The recursive search technique provides a method of improving the solution by successively incrementing the decision variables, and implicitly the state variables, until the optimum solution is reached. This technique begins by finding an optimum value for the first stage decision variable, x_1, for the stage 1 state vector, S_1, defined by the starting solution X. The first stage vector is given by:

$$S_1 = (A_1 - \sum_{k=2}^{N} \delta_{1k}x_k,\ A_2 - \sum_{k=2}^{N} \delta_{2k}x_k,\ \cdots,$$

$$A_M - \sum_{k=2}^{N} \delta_{Mk}x_k) \ . \tag{4-11}$$

With the first stage state vector fixed, a search over x_1 can be accomplished (while maintaining a feasible solution) to determine the value of x_1 which maximizes the recursive relationship for the first stage:

$$f_1^*(S_1) = \max_{x_1 \leq \min S_1} r_1(x_1) \ . \tag{4-12}$$

To determine the value of x_1 which optimizes Equation (4-12) for a given state S_1, increment x_1 by an amount delta (Δ) until a point x_1^* is reached where

$$f_1(S_1, x_1^* - \Delta) < f_1(S_1, x_1^*) > f_1(S_1, x_1^* + \Delta) \tag{4-13}$$

or until one of the constraints prevents incrementing x_1 further.

As a matter of notation, let:

$$f_1^*(S_1) = f_1(S_1, x_1^*) = \max_{x_1 \leq \min S_1} f_1(S_1, x_1) \tag{4-14}$$

so that $f_1^*(S_1)$ is the optimum return from the first stage for a fixed input vector S_1.

For the second stage, the dynamic programming recursive relationship is given by:

$$f_2(S_2, x_2) = r_2(x_2) + f_1^*(S_2 - D_2x_2) \tag{4-15}$$

where the first term is the return function for the second stage, and the second term is the optimum first stage return for the input state vector $(S_2 - D_2x_2)$. It is now necessary to find an optimum value of x_2 for the state vector S_2. (Recall that S_2 is specified by the starting solution vector x_3, x_4, ..., x_N which has not been changed thus far.)

To determine an optimum x_2, increment this decision variable by an amount delta (delta may be positive or negative, depending on the direction which causes Equation (4-15) to increase). Changing x_2, however, not only changes the second stage return, $r_2(x_2)$, but also the input to the first stage through the state transformation equation

$$S_1 = S_2 - D_2x_2 \; . \tag{4-16}$$

Therefore, for each change in x_2 and resulting change in S_1, it is necessary to calculate a new value of $f_1^*(S_2 - D_2x_2)$; i.e., reoptimize the first stage for the new input vector. This is accomplished in the same manner as before, incrementing x_1 until $f_1(S_1, x_1)$ is at a maximum within the constraints. It is necessary to reoptimize x_1 for each new S_1 before evaluating Equation (4-15) to determine if x_2 is at a maximum.

Continuing in this manner, x_2 is incremented (and x_1 reoptimized) until a point x_2^* is reached where:

$$f_2^*(S_2, x_2^* - \Delta) < f_2(S_2, x_2^*) > f_2(S_2, x_2 + \Delta) \tag{4-17}$$

where again:

$$f_2(S_2, x_2^*) = f_2^*(S_2) \; . \tag{4-18}$$

At this point, $f_2^*(S_2)$ is the optimum total return for the first and second stages for the state vector S_2.

Going next to the third stage recursive equation:

$$f_3(S_3, x_3) = r_3(x_3) + f_2^*(S_3 - D_3 x_3) \quad . \tag{4-19}$$

The optimum third stage return for a state vector S_3 is obtained by incrementing across x_3 in the same manner as before. In this case, it can be seen that changing x_3 changes the input to the second stage, and, therefore, to the first stage also, through the state transformation function. Thus, it is necessary to reoptimize the first stage, and then the second stage, in a manner identical to the previous steps.

This procedure is continued in a similar manner through stage N; incrementing across x_N and subsequent reoptimization of stages x_1 through x_{N-1} for the resulting state variables will result in an optimum return:

$$f_N^*(S_N) = f_N^*(A_1, A_2, \ldots, A_M) \tag{4-20}$$

at an optimum solution vector:

$$X^* = (x_1^*, x_2^*, \ldots, x_N^*) \quad .$$

This process is shown in Figure 4 for a three stage allocation problem; i.e., solving the problem:

$$\text{Maximize } R(X) = r_1(x_1) + r_2(x_2) + r_3(x_3) \tag{4-21}$$

subject to:

$$\delta_{i1}x_1 + \delta_{i2}x_2 + \delta_{i3}x_3 \leq A_i \qquad i = 1, 2, 3, 4 \quad . \tag{4-22}$$

This figure shows only the basics of the algorithm in order to describe the logic behind this technique. The details of the algorithm

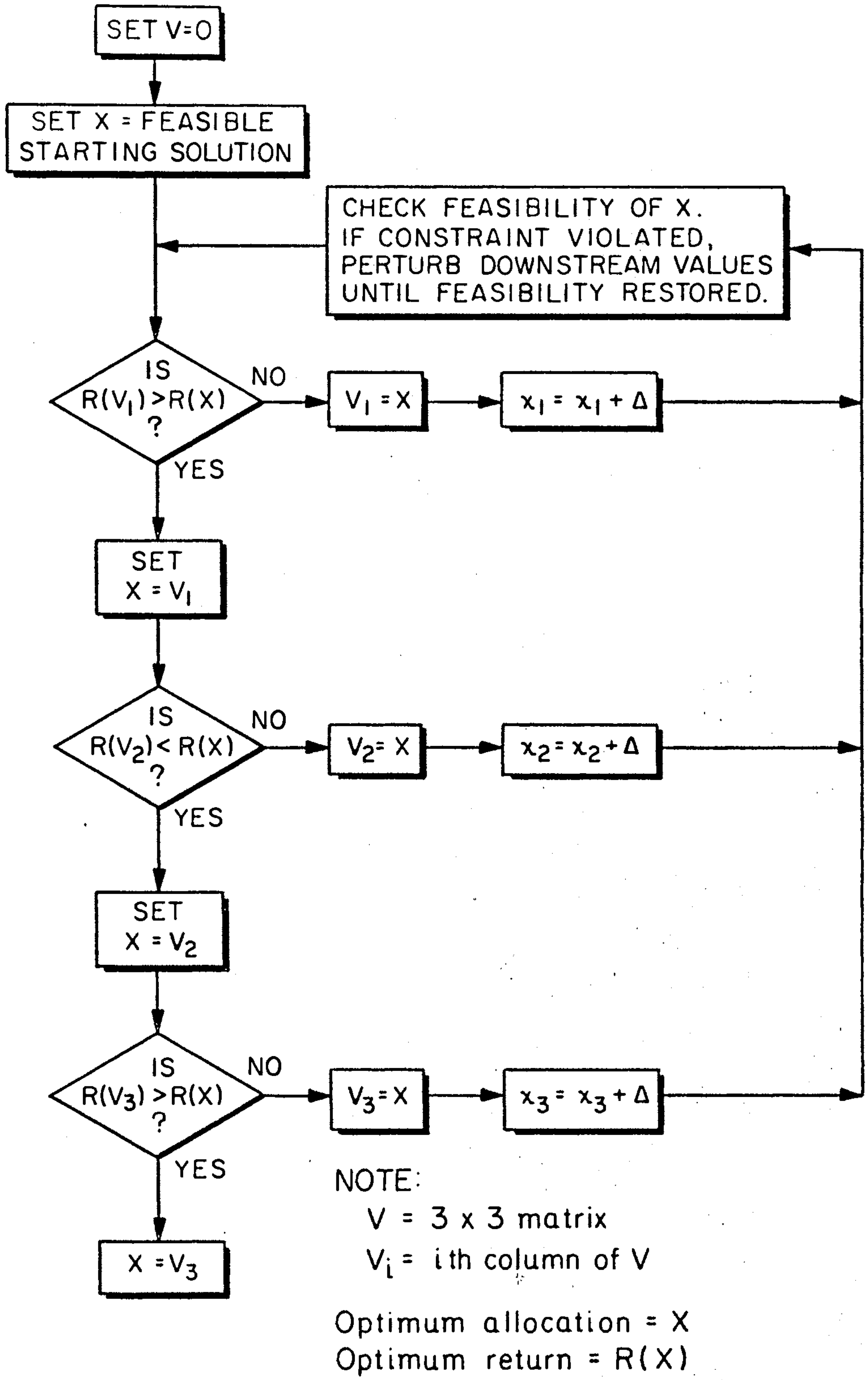


Figure 4. Recursive Search Algorithm for Three Stage Problem

vary depending on the particular problem being solved. The computer code which implements this algorithm for the allocation problem with constraints on two entities is given in Appendix A.

The algorithm starts by setting three vectors, V_1, V_2, and V_3 equal zero. Each vector contains the same number of components as stages, in this case three. Vector V_1, for example, will contain the current value of the vector with the optimum first stage decision for the input state specified by x_2 and x_3. Similarly, V_2 will contain the vector with the optimum value of x_2 for the input vector specified by x_3. Finally, V_3 will contain the optimum vector specified by the input state (A_1, A_2, A_3, A_4).

The starting solution $X = (x_1, x_2, x_3)$ is set equal to a feasible starting solution; a solution that satisfies the constraints of Equation (4-22).

Now letting

$$R(X) = \sum_{i=1}^{3} r_i(x_i) \tag{4-23}$$

a comparison is made between $R(X)$; i.e., the return obtained from the starting solution, and $R(V_1)$. Since $V_1 = (0, 0, 0)$ at this point, $R(X)$ is greater than $R(V_1)$ so that the "no" branch is taken. The vector V_1 will then be set equal to X and the first decision variable, x_1, incremented by delta. Next, a check is made to determine if the new solution vector $(x_1 + \Delta, x_2, x_3)$ still satisfies the constraints. If not, x_1 is at the optimum value for the input state specified by x_2 and x_3, and the algorithm proceeds to the next stage. (The portions of the algorithm that perform the feasibility check are omitted from this figure for simplicity.)

If the new trial solution is still feasible, $R(V_1)$ is compared to $R(X)$ to determine if incrementing x_1 increased the return function. If so, x_1 continues to be incremented until a constraint is reached, or until a further increase in x_1 causes the return function to decrease. At this point $X = (x_1^*, x_2, x_3)$ so that x_1 is at the optimum value for the input state S_1 specified by x_2 and x_3 as follows:

$$S_1 = S_2 - D_2 x_2 \tag{4-24}$$

$$S_1 = (A_1 - \delta_{12}x_2 - \delta_{13}x_3,\ A_2 - \delta_{22}x_2 - \delta_{23}x_3,\ A_3 - \delta_{32}x_2 - \delta_{33}x_3) \ . \tag{4-25}$$

At this point the working vector, X, is set equal to the optimum stage 1 vector, V_1, and $R(V_2)$ is compared to $R(X)$. Since $V_2 = (0, 0, 0)$ at this point, $R(V_2) < R(X)$ so the algorithm sets $V_2 = X$ and increments the second stage decision variable, x_2, by delta. However, incrementing x_2 changes S_1, so a new optimum value of x_1 for this new input state must be calculated. To accomplish this, the algorithm sets the elements of V_1 equal zero and reoptimizes x_1 until a point x_1^* is reached; $x_1^* \in (x_1^*, x_2 + \Delta, x_3)$. X is then set equal V_1 so that:

$$X = V_1 = (x_1^*, x_2, x_3) \tag{4-26}$$

$$V_2 = (x_1^*, x_2 + \Delta, x_3) \ . \tag{4-27}$$

$R(V_2)$ is now compared to $R(X)$ to determine if incrementing x_2 increased the return function. If so, x_2 is again increased and x_1 reoptimized for the new input state vector. This is continued until x_2 and x_1 are both at an optimum value for the input state S_2 specified by x_3.

It must now be determined if x_3 can be improved, so this decision variable is incremented in a search across the third stage recursive equation. The algorithm continues to increment x_3, and reoptimize x_1 and x_2 for each new input state, until a point is reached where:

$$R(x_1^*, x_2^*, x_3 - \Delta) < R(x_1^*, x_2^*, x_3) > R(x_1^*, x_2^*, x_3 + \Delta) \quad . \tag{4-28}$$

This is the optimum allocation $X^* = (x_1^*, x_2^*, x_3^*)$, and $R(X^*)$ is the optimum return.

Maintaining Feasibility

The recursive search technique requires that a feasible solution be maintained while searching across the recursive equations for the optimum value of the decision variable. This is accomplished as follows.

As each decision variable is incremented, the new trial solution is checked for feasibility. If the trial solution is infeasible, "downstream" decision variables are operated on until feasibility is restored. For example, if x_3 is increased and if this makes the trial solution infeasible, x_1 and/or x_2 are increased or decreased (depending on the type of constraint being violated) until feasibility is restored. This feature is not shown on the flow diagram due to dependence on the type of problem being solved.

Also included in the algorithm, shown in later figures, is a feature to allow the decision variable to be incremented in both positive and negative directions. It is not known beforehand whether increasing or decreasing a particular decision variable will cause the objective function to increase. Therefore, the algorithm provides for a search in both directions before proceeding to the next stage. If increasing the decision variable decreases the objective (return) function, the direction is reversed and that decision variable incremented in the negative direction. The algorithm continues to increment the decision variable in a direction that causes the return function to increase. After each increment is added, the trial solution is checked for feasibility. This process is repeated until further increasing the decision variable violates a constraint such that the solution cannot be made feasible by perturbing downstream variables, or until the return function starts to decrease. At this point the algorithm proceeds to the next stage.

Recursive Search Algorithm for n-Stage Problem

To make the algorithm more efficient, define an n x n matrix V, and let V_j represent the j^{th} column of that matrix. Each column of V contains n components, and V_j contains the optimum solution for the j^{th} stage for the input state defined by x_n, x_{n-1}, ..., x_{j-1}.

Also, let K represent an n-component vector, $K = (k_1, k_2, \ldots, k_n)$. The value of k_j determines the direction of search for the j^{th} variable; for k equal zero x_j is incremented in the positive direction. For k equal one x_j is incremented in the negative direction.

With these definitions, the algorithm for an n-stage recursive search solution is shown in Figure 5. To illustrate the use of this procedure, again consider the four stage dynamic programming problem given in Chapter III.

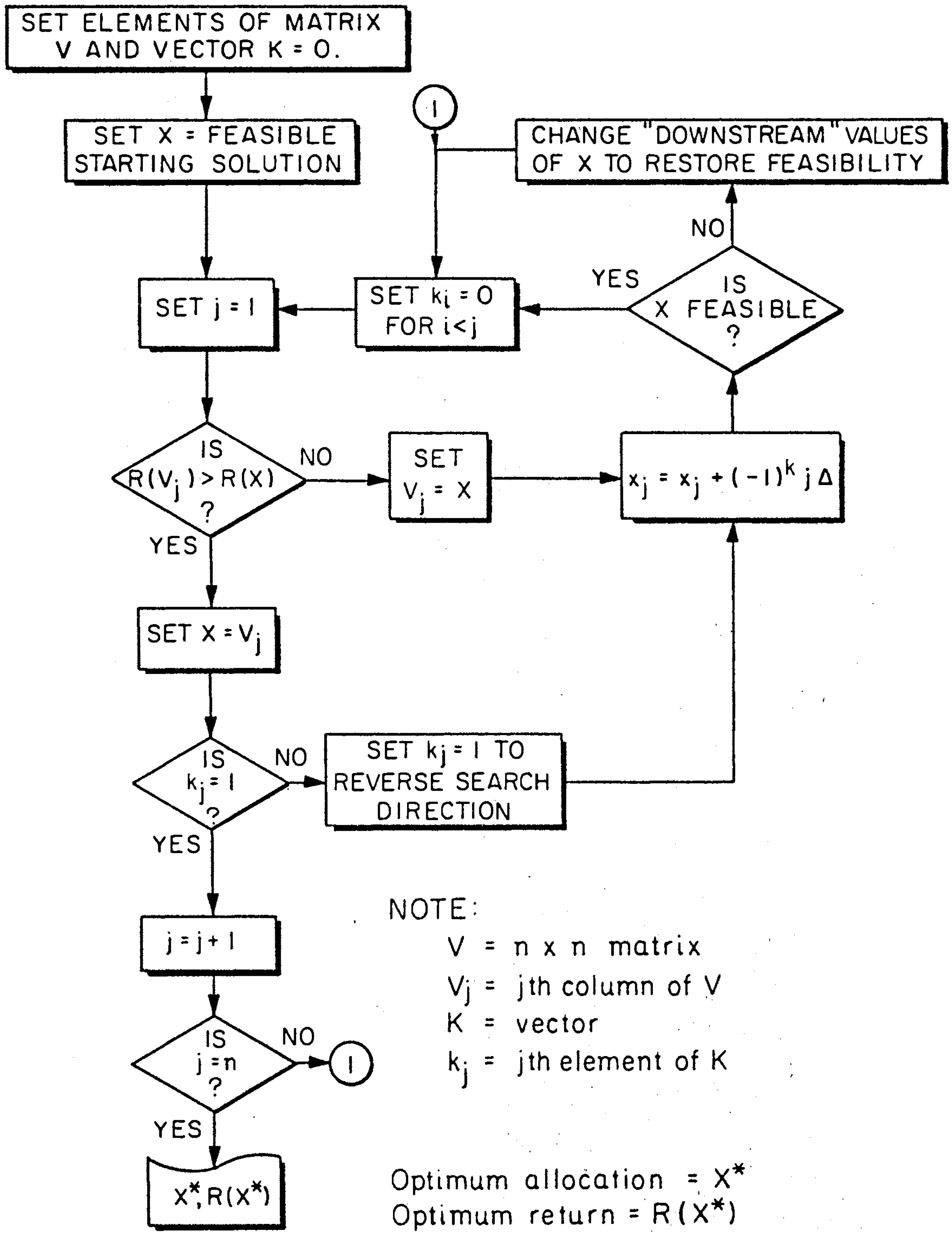


Figure 5. Recursive Search Algorithm for n-Stage Problem

$$\text{Maximize } R(X) = \sum_{i=1}^{4} r_i(x_i)$$

subject to:

$$\sum_{i=1}^{4} x_i \leq 10$$

where the return function for each stage is given in Table I.

In order to see the correlation between conventional dynamic programming and recursive search, calculate the input state specified by the starting solution and compare each step of the recursive search to the conventional dynamic programming solution given in Tables II through V. Notice, however, that with the recursive search, it is not necessary to calculate the state variables. Since a feasible decision is always defined, the state variables are implicitly in the solution, but never need to be determined.

Choose a starting solution X = (2, 2, 2, 2). With this starting solution, the input to each of the stages is determined as follows, using the transformation function, Equation (3-5).

$$s_4 = A = 10$$
$$s_3 = s_4 - x_4 = 8$$
$$s_2 = s_3 - x_3 = 6$$
$$s_1 = s_2 - x_2 = 4$$

and, using the returns of Table I,

$$R(X) = \sum_{i=1}^{4} r_i(x_i) = 11{,}383 \quad .$$

In accordance with the recursive search algorithm, increment x_1 by delta, which for this problem is chosen as a unit increment. The new vector is then $X' = (3, 2, 2, 2)$. Since

$$\sum_{i=1}^{4} x_i = 9 \quad ,$$

the constraint is not violated, and $R(X) = 11{,}783$.

Now since $R(X') > R(X)$, x_1 is again increased to give a new trial solution $X'' = (4, 2, 2, 2)$. Again, the constraint is not violated and $R(X'') = 12{,}020 > R(X')$. The first stage decision variable is again incremented to give $X''' = (5, 2, 2, 2)$. However, this solution violates the constraint, so $x_1 = 4$ is the optimum value for the input state $s_1 = 4$. It can be seen from Table II that an identical result is obtained in conventional dynamic programming.

Now set $V_1 = X'' = (4, 2, 2, 2)$, the optimum value of x_1 for $x_2 = x_3 = x_4 = 2$ (and, implicitly, $s_1 = 10 - 6 = 4$). Increment the second stage decision variable giving a new working vector $X = (4, 3, 2, 2)$. Since the constraint is violated for this solution, the downstream variable, x_1, is reduced until a feasible solution is obtained, giving $X' = (3, 3, 2, 2)$. Since x_1 cannot be increased without violating the constraint, $x_1 = 3$ is the optimum value for the input state s_1 specified by $x_2 = 3$, $x_3 = x_4 = 2$; i.e., for the input state

$$s_i = 10 - \sum_{i=2}^{4} x_i = 3 \quad .$$

At this point, $R(X') = 11{,}886$, and since $R(X') < R(V_1)$ the direction of search over the second stage is reversed to determine if decreasing x_2 will improve the solution. Thus the new trial solution vector is: $X'' = (1, 1, 2, 2)$. The input state to the first stage is now given by

$$s_1 = 10 - \sum_{i=2}^{4} x_i = 5 \quad .$$

Incrementing x_1 as before gives an optimum value of 5 for this input state. Then, for $X' = (5, 1, 2, 2)$, $R(X') = 11{,}896$. Since $R(X') < R(V_1)$, the optimum first and second stage decision variables for $s_2 = 6$ are $x_1 = 4$, $x_2 = 2$. Note that from Table III, for $s_2 = 6$, $x_2^* = 2$. Then $s_1 = 6 - 2 = 4$ and from Table II, $x_1^* = 4$. Thus, identical results are obtained with both conventional dynamic programming and the recursive search technique. The next step, in accordance with the algorithm, is to set $A_2 = X = (x_1^*, x_2^*, x_3, x_4) = (4, 2, 2, 2)$.

Next, x_3 is incremented, giving a new solution vector $X = (3, 2, 3, 2)$ where x_1, as a downstream variable, has been reduced until a feasible solution was obtained. Before the new trial solution for $x_3 = 3$ can be evaluated, however, it is necessary to reoptimize x_1 and x_2 for the input state $s_2 = 10 - 3 - 2 = 5$. This is accomplished in the same manner as before.

Succeeding steps of this algorithm continue to improve the solution by incrementing the decision variable at each stage until an optimum solution is found. In contrast to conventional dynamic programming, the recursive search calculates values of the return function only for those solutions on the path between the starting and optimum solution. Therefore, the number of calculations is usually reduced.

As shown in Appendix A, the optimum solution for this problem obtained by the recursive search technique is $X^* = (2, 1, 4, 3)$ giving an optimum return of 12,675; results that are identical with those obtained in Chapter III.

For a one-dimensional allocation problem, there is a small savings in computer memory, and also a reduction in the required number of calculations. However, now consider a problem where there is one project with a budget constraint, and in addition, constraints on each of the four time periods, such as:

$$\text{Maximize } R(X) = \sum_{i=1}^{4} r_i(x_i)$$

subject to:

$$\sum_{i=1}^{4} x_i \leq 10$$

$$x_i \leq 4 \qquad i = 1, 2, \ldots, 4$$

$$x_i \geq 0, \text{ integers} \quad .$$

With five constraints, the state variable is a vector with five components, and although there are only four feasible levels of the state variable at each stage, there are 4^5 feasible inputs to each stage, requiring approximately 6×4^5 storage spaces. However, with recursive search, the problem is not complicated in any way, since the optimum solution for every feasible input state need not be determined. The storage requirements remain $n \times n$, in this case 4×4. In fact, the problem requires fewer calculations since the feasible range of each decision variable has been reduced.

The solution for this problem is $X^* = (2, 1, 4, 3)$, which is identical to the previous problem since the time period constraints did not bind. If the time period constraints are reduced from four to three, however, the optimum allocation is $X^* = (3, 1, 3, 3)$ giving an optimum return of 12,631.

Mathematical Proof

The basis of this technique is that a search is conducted successively over the dynamic programming recursive relationships:

$$f_k(S_k, x_k) = r_k(x_k) + f^*_{k-1}(S_k - D_k x_k) \tag{4-37}$$

where $f^*_o(S_o, x_o) \equiv 0$, to determine the optimum return from stages 1 through k, k = 1, 2, ..., N, given by:

$$f^*_k(S_k) = \max_{x_k \leq \min S_k} f_k(S_k, x_k) \quad . \tag{4-38}$$

In order for this search technique to converge to a global maximum, a necessary and sufficient condition is that each $f_k(S_k, x_k)$ be concave (or conversely, to converge to a global minimum each $f_k(S_k, x_k)$ must be convex) over the decision variable x_k. This is proved in the following paragraphs.

A function g(z) is said to be concave if, for any point z^* between z_1 and z_2,

$$g(z^*) \geq \alpha g(z_1) + (1-\alpha)g(z_2) \tag{4-39}$$

for $0 \leq \alpha \leq 1$.

This says, in effect, that if g(z) is concave, then the function evaluated at any point between z_1 and z_2 is greater than or equal to any point on a linear interpolation between $g(z_1)$ and $g(z_2)$. If Equation (4-39) is a strict inequality, then $g(z)$ is said to be strictly concave.

To prove concavity in Equation (4-37), first consider stage one, where the recursive relationship is a function of the stage return only:

$$f_1^*(S_1) = \max_{x_1 \leq \min S_1} r_1(x_1) \quad . \tag{4-40}$$

As before, min S_1 indicates the minimum component of the vector S_1. Since $r_1(x_1)$ is assumed to be concave, $f_1^*(S_1)$ is also concave. It can be seen that the input vector simply limits the maximum value of x_1 to be less than or equal to the minimum slack in the state vector. The constrained maximum value can, therefore, be easily determined by incrementing x_1. Since integer values are desired, it is assumed that the decision variables are incremented by an integer amount in the search technique.

The second stage recursive relationship is given by:

$$f_2(S_2,\ x_2) = r_2(x_2) + f_1^*(S_2 - D_2x_2) \quad . \tag{4-41}$$

Now $r_2(x_2)$ is concave by assumption, and since the sum of concave functions is also concave, $f_2(S_2,\ x_2)$ is concave if $f_1^*(S_2 - D_2x_2)$ is concave. In searching for the optimum of Equation (4-41); i.e., $f_2^*(S_2)$, x_2 is incremented, holding S_2 constant, until a maximum value of $f_2(S_2,\ x_2)$ is obtained within the constraints. This increments the input to the first stage, from the transformation equation

$$S_1 = S_2 - D_2(x_2 + k\Delta) \quad k = 1,\ 2,\ \ldots \tag{4-42}$$

and for each new state vector S_1, a new optimum $f_1^*(S_1)$ must be determined. Thus, incrementing across x_2 causes a search across S_1 in the function $f_1^*(S_1)$. Therefore, it is necessary to prove that $f_1^*(S_1)$ is concave in S_1.

For the continuous case, $f_1^*(S_1)$ can be shown to be concave for concave stage return functions in a straightforward manner. However, the analysis becomes considerably more complex when the solution is restricted to integer values. Therefore, the continuous case will be proved, then a heuristic argument used to show where integer solutions can introduce non-concavity in constrained optimization problems which are more general than that given by Equation (4-1).

Let S_1^1 and S_1^2 be two state vectors in the first stage, and x_1^1 and x_1^2 be optimum values of x_1 for states S_1^1 and S_1^2, respectively. Then

$$f_1^*(S_1^1) = f_1(S_1^1,\ x_1^1) \tag{4-43}$$

$$f_1^*(S_1^2) = f_1(S_1^2,\ x_1^2)\ . \tag{4-44}$$

Multiplying Equation (4-43) by α and Equation (4-44) by $(1-\alpha)$ and adding:

$$\alpha f_1^*(S_1^1) + (1-\alpha)f_1^*(S_1^2) = \alpha f_1(S_1^1,\ x_1^1) + (1-\alpha)f_1^*(S_1^2,\ x_1^2)\ . \tag{4-45}$$

Now if S_1 is a state between S_1^1 and S_1^2, and x_1 is a decision between x_1^1 and x_1^2, and if min $S_1^1 <$ min S_1^2 and $x_1^1 < x_1^2$, then using the fact that the stage return is concave:

$$f_1(S_1,\ x_1) \geq \alpha f_1(S_1^1,\ x_1^1) + (1-\alpha)f_1(S_1^2,\ x_1^2)\ . \tag{4-46}$$

But since $f_1^*(S_1) = \max f_1(S_1,\ x_1)$, and using Equations (4-43) and (4-44),

$$f_1^*(S_1) \geq \alpha f_1^*(S_1^1) + (1-\alpha)f_1^*(S_1^2)\ . \tag{4-47}$$

Therefore, from the definition of concavity given in Equation (4-39), $f_1^*(S_1)$ is concave across S_1. Since both $r_2(x_2)$ and $f_1^*(S_1)$ are concave, then $f_2(S_2,x_2)$ is concave. Using an argument identical to the previous proof, if $f_2(S_2,x_2)$ is concave, then $f_2^*(S_2)$ is concave, and thus $f_3(S_3,x_3)$ is concave. Then, by induction $f_k(S_k,x_k)$ is concave for $k = 1, 2, \ldots, N$. Since each $f_k(S_k,x_k)$ is concave, it is possible to search across each of the functional relationships successively to arrive at a global maximum.

It was assumed in the above proof that there were no integer restrictions. Now consider the more complex case of integer solutions.

Recursive Search with Integer Restrictions

For the first stage, Equation (4-40) is a function of the stage return only. Since x_1 takes on only integer values in the problem formulation, Equation (4-40) is concave for integer solutions also. However, consider the second stage return, Equation (4-42), where the components of the vector D are not restricted to zero or one; i.e., the more general case where the constraints are of the form:

$$\sum_{i=1}^{n} c_{ij}x_i \leq A_j \qquad j = 1, 2, \ldots, m \tag{4-48}$$

with no restrictions on the c_{ij}.

Since the optimum first stage return is a function of $r_1(x_1)$, the second stage recursive relationship, Equation (4-42), may be written as:

$$f_2(S_2, x_2) = r_2(x_2) + \max_{x_1 \leq \min[S_1/C_1]} r_1(x_1) \tag{4-49}$$

where min $[S_1/C_1]$ is the minimum component of

$$[s_{11}/c_{11}, s_{12}/c_{12}, \ldots, s_{1M}/c_{1M}]$$

and where the brackets indicate that integer values are to be taken.

Assume that the k^{th} constraint of Equation (4-48) is binding, so that the maximum value of the second term occurs at this constraint. Then

$$\min[S_1/C_1] = [s_{ik}/c_{ik}] \quad . \tag{4-50}$$

Using the state transformation function, Equation (4-5), x_1 is limited by:

$$x_1 \leq \left[\frac{s_{2k} - c_{2k}x_2}{c_{1k}}\right] \tag{4-51}$$

and since the maximum occurs at this value, Equation (4-49) becomes:

$$f_2(S_2, x_2) = r_2(x_2) + r_1\left[\frac{s_{2k} - c_{2k}x_2}{c_{1k}}\right] . \tag{4-52}$$

It can be shown that the second term of Equation (4-52) is not concave for certain values of c_{1k} and c_{2k} when the solutions are restricted to integer values. To prove this, choose c_{1k} and c_{2k} such that:

$$\left[\frac{s_{2k} - c_{2k}(x_2 - \Delta)}{c_{1k}}\right] > \left[\frac{s_{2k} - c_{2k}x_2}{c_{1k}}\right] = \left[\frac{s_{2k} - c_{2k}(x_2 + \Delta)}{c_{1k}}\right] . \tag{4-53}$$

For example, let $c_{1k} = 3$ and $c_{2k} = 1$, and consider the case where $s_{2k} = 10$, $x_2 = 2$, $\Delta = 1$. Then the terms in Equation (4-53) become 3.0, 2.67, and 2.33, respectively. Taking integer values, these numbers become 3, 2, and 2 so that Equation (4-53) holds. Now consider the simplest case of a linear (and thus concave) return function of the form:

$$r_i(x_i) = x_i \qquad i = 1, 2, \ldots, N \ .$$

For this case, the test for concavity, Equation (4-39), does not hold; i.e., x_2 is between $x_2 - \Delta$ and $x_2 + \Delta$, but

$$r_2(x_2) \not> \alpha r_2(x_2 - \Delta) + (1 - \alpha) r_2(x_2 + \Delta) \tag{4-55}$$

since, using Equation (4-53)

$$\left[\frac{s_{2k} - c_{2k} x_2}{c_{1k}}\right] \not> \alpha \left[\frac{s_{2k} - c_{2k}(x_2 - \Delta)}{c_{1k}}\right] + (1 - \alpha)\left[\frac{s_{2k} - c_{2k}(x_2 + \Delta)}{c_{1k}}\right] . \tag{4-56}$$

For example, with $\alpha = .5$, using the values calculated previously, Equation (4-56) yields:

$$2 \not> (.5)(2) + (.5)(3) = 2.5$$

and, thus, the second term of Equation (4-42) is not necessarily concave. As a consequence, $f_2(S_2, x_2)$ is not necessarily concave for all functions. Notice, however, that under many conditions, this function is concave and a search technique can be used. For example, if the constraints do not bind, then Equation (4-41) is concave even for integer solutions.

If we now consider the problem given by Equation (4-1); i.e., coefficients on the constraint variables restricted to zero or one, then Equation (4-51) is of the form:

$$x_1 \leq s_{2k} - \delta_{2k} x_2 \ . \tag{4-57}$$

δ_{1k} must be equal one, since if it were zero that term could not have been the minimum and, thus, could not bind.

Since the x_i are restricted to integer values, each s_{ij} must also be integer-valued, from Equation (4-5). As a result, Equation (4-57) always produces integer values and, thus, there are no values for which Equation (4-53) holds. Therefore, the dynamic programming formulation of Equation (4-1) is concave for integer solutions, and a search technique can be used to determine the optimum solution. For the more general case, however, where the coefficients of the constraints are not restricted to zero or one, the constrained objective function is not necessarily concave for integer solutions, and a search technique may not converge to a global optimum.

CHAPTER V

RELATED PROBLEMS AND CONCLUSIONS

The technique for mathematical programming developed in this thesis provides an efficient method of solving certain classes of allocation problems with multiple constraints. The specific problem studied has been that of project selection; a form of the capital budgeting problem. As already mentioned, recursive search dynamic programming can also be applied to other types of problems amenable to solution by conventional dynamic programming. Any problem that can be formulated as a dynamic programming problem can be solved using this technique providing:

(1) The return functions are concave. (Or convex in the case of minimization problems.)

(2) The constraints are of the form given in Equation (2-3).

Although the discussions in this thesis have been centered around the economy of recursive search when applied to multiple-constraint problems, some unconstrained or partially constrained problems can be efficiently solved using this technique, especially when the solutions are restricted to integer values.

Manpower Leveling

Another optimization problem considered in the operations research literature is that of manpower leveling. In many businesses, the manpower requirements vary from year to year or from season to season. Although it would be possible to change the manning level to meet the requirements of each time period, there is a cost involved due to administrative expenses in hiring and firing and due to inefficiencies caused by the continual flux of personnel. On the other hand, however, if the same manpower level were to be maintained, during some of the time periods there would be an excess of personnel charged to overhead while in others a shortage would require increased costs for overtime. Thus, it is desired to determine employment levels which will minimize costs.

An example of manpower leveling is discussed in Hillier and Lieberman (8). In this case, continuous solutions are assumed to simplify the problem. However, recursive search can be readily applied to obtain integer solutions.

For this problem, the manpower requirements for each season of the year are as shown in Table VII. The manpower level for the preceeding season is 255, which is assumed to be fixed.

TABLE VII

MANPOWER REQUIREMENTS FOR MANPOWER LEVELING PROBLEM

Season	Summer	Autumn	Winter	Spring
Requirements	220	240	200	255

The decision variables for this problem, x_k, $(k = 1,2,3,4)$ are the employment levels at the k^{th} stage from the end, where stages correspond to seasons. The state variables, s_k, are the employment levels at the beginning of stage k. In this problem, the state variables are scalars instead of vectors as encountered in the multiple-constraint problem.

The cost of maintaining levels above the required manpower is assumed to be \$2000 per man per season. The total cost of changing the level of employment is assumed to be \$200 times the square of the difference in manpower levels. It is further assumed that the level cannot fall below the requirements (no overtime allowed), so that this is a partially constrained problem.

The recursive relationship for the k^{th} stage of this problem is given by:

$$f_k(s_k,\ x_k) = 200(x_k - s_k)^2 + 2000(x_k - w_k) + f^*_{k-1}(s_{k-1}) \tag{5-1}$$

where w_k is the required manpower level for the k^{th} season.

Since the state at the $(k-1)^{st}$ stage is the employment level at the k^{th} stage, the transformation function is given by:

$$s_{k-1} = x_k \tag{5-2}$$

so that Equation (5-1) can be written as:

$$f_k(s_k,\ x_k) = 200(x_k - s_k)^2 + 2000(x_k - w_k) + f^*_{k-1}(x_k)\ . \tag{5-3}$$

The basic recursive search algorithm given in Figure 2 is applied to this problem, using a starting solution vector $X = (255,200,240,220)$. In this case the starting solution is set equal to the requirements.

Since the stages are numbered in reverse order, x_i corresponds to the Spring employment, x_2 to the Winter level, etc.

Appendix B contains the computer code of the recursive search algorithm developed to solve the manpower leveling problem.

The solution obtained using recursive search is $X^* = (255, 247, 244, 247)$; i.e., Summer, Autumn, Winter, and Spring requirements of 247, 244, 247, and 255, respectively. The corresponding cost is $185,200. The solution obtained by Hillier and Lieberman, assuming continuous solutions, is 247.5, 245, 247.5, and 255 for a total cost of $185,000.

Another interesting aspect of this problem can be studied through a simple change to the return functions. Assume now that overtime can be used at time and one half regular time. In this case, the cost for a shortage of personnel is given by $1.5(2000)(x_k - w_k)$. The problem was solved again using recursive search, with the return function appropriately modified. The total cost in this case was $159,400, with the manning levels shown in Table VIII. Thus, a savings of over $25,000 can be obtained by using overtime.

TABLE VIII

OPTIMUM MANPOWER LEVELS WITH AND WITHOUT OVERTIME

Season	Summer	Autumn	Winter	Spring	Cost
No overtime	247	244	247	255	$185,200
With overtime	245	240	236	237	$159,400

The project selection and manpower leveling problems illustrate the variety of applications of the recursive search algorithm given in Figure 3. Although details of the computer code implementing the algorithm vary from one problem to another depending on the form of the recursive relationships and the number and type of constraints, the solution technique remains essentially the same.

Computational Considerations

Improved Search Technique

The recursive search technique can be made more efficient by modification of the method of search employed. In seeking to optimize the dynamic programming recursive relationships, the recursive search algorithm increments the decision variable, then reoptimizes previous stages until an optimum value of the decision variable is obtained for that particular stage and input state. In most problems, since integer solutions are desired, the decision variables are incremented by a unit amount in the search. However, for problems where the range of the decision variables are large, incrementing by a unit amount can use a lot of computer time, especially if the feasible starting solution is considerably different than the optimum solution.

In order to reduce computer time, the algorithm can be modified so that fewer calculations are required to converge to the optimum decision variable for each recursive equation. One method of doing this is to solve the problem several times; initially with a large delta (incrementing value) then reduce delta in subsequent passes until a unit delta is reached. This is analogous to the course-fine grid search technique proposed by Nemhauser (7).

For example, for the first pass through a problem, a delta of 100 can be used for the course grid search. This will result in a more rapid convergence to an approximate solution. If the solution obtained on this pass is given by $X = (x_1, x_2, \ldots, x_n)$, then it is known that the true optimum lies within the interval.

$$X^* = (x_1 - \Delta \leq x_1^* \leq x_1 + \Delta, \ldots, x_n - \Delta \leq x_n^* \leq x_n + \Delta) \ .$$

In the next pass through the problem, delta can be reduced to obtain an even better approximation until finally the exact integer solution is obtained when a unit delta is used.

Since it is known that each true optimum decision variable lies within delta of the approximate optimum, the algorithm must be changed to ensure that the recursive search for each decision variable is limited to the range $x_k - \Delta \leq x_k^* \leq x_k + \Delta$. This can be accomplished by adding additional constraints after each course grid solution. Since the number of constraints do not increase the number of state variables in the solution as with conventional dynamic programming, the additional constraints do not complicate the problem.

This feature has been incorporated into the manpower leveling code of Appendix B. The code initially sets limits within which the optimum solution vector must lie. For example, the lower limit is zero and the upper limit is arbitrarily set at 500 for this problem. The initial delta was set at 2, which yielded an approximate optimum solution of $X = (256, 246, 244, 246)$. The search width for the next pass was set at $x_k \pm \Delta$ so that the problem constraints for each decision variable were re-set to these values. The optimum allocation for the subsequent pass, for a unit delta, was $X^* = (255, 247, 244, 247)$, as before.

A reduction in the number of calculations can also be achieved through the use of the Fibonacci search (7), which, under some conditions, may be more efficient than the course-fine grid search.

Improved Starting Solution

Since the number of calculations necessary to converge to the optimum solution is a function of the starting solution, the efficiency of the algorithm can be improved by judicious selection of this starting solution.

Although the optimum solution is obviously not known in advance, the analyst usually has a fair idea of approximately where it lies. In this case, it is best to choose a feasible starting solution equal to this guess to reduce the number of feasible solutions on the path between the starting and optimum solutions.

The recursive search technique relies on maintaining a feasible solution, therefore, this initial guess must be feasible as well as being in the vicinity of the optimum. To simplify matters, the computer code given in Appendix A allows the analyst to choose the starting solution without worrying about feasibility. The code checks the starting solution and, if infeasible, restores feasibility before proceeding into the main part of the program. For the manpower leveling problem, the starting solution must be feasible, therefore, the algorithm sets the starting solution equal to the manpower requirement vector.

Infeasible Stages

In the project selection recursive search algorithm, it is assumed that there are $n \times m$ feasible stages. This means that there are n projects, and each project lasts m time periods. However, in many cases, the projects may last an unequal number of time periods. For example, project 1 may last ten time periods whereas project 2 may last only nine, or project 2 may not start until time period 2. In the first case, the stage corresponding to decision variable x_{29} is not feasible. Similarly, in the second case, the stage for variable x_{21} is not feasible. To ensure that no allocations are made to these infeasible stages, an artifical return is assigned to each such stage in the algorithm. For maximization problems, infeasible stages are assigned a large negative return. This is analogous to the "big M" technique of linear programming.

A similar problem can occur in a transportation problem where there is no route between a supply point and a demand point. Here the cost, or distance between these points, would be chosen as infinity.

Summary of Results

This research is directed to the solution of the allocation problem with multiple constraints and non-linear objective function using a technique referred to as recursive search dynamic programming. Integer solutions of resource allocation problems are usually obtained through application of dynamic programming developed by Richard Bellman. However, this technique becomes very inefficient when the resource allocation is restricted by several constraints, since the amount of computer memory required increases exponentially with the number of constraints. Thus, when the number of constraints is greater than two or three, the memory requirements usually exceed computer capacity.

Recursive search dynamic programming circumvents this "curse of dimensionality" by successively incrementing the decision variable in the recursive equation at each stage of the problem while maintaining a feasible solution. In this manner the number of constraints does not decrease the efficiency of the algorithm, but actually increases the efficiency by limiting the feasible range of the decision vector, and excluding some of the possible states.

This technique is proved to converge to a global optimum for problems of the form:

$$\text{Maximize (Minimize)} \sum_{i=1}^{n} r_i(x_i)$$

subject to:

$$\sum_{i=1}^{n} x_i (\leq, \geq) A_j \qquad j = 1, 2, \ldots, m$$

provided the return functions are concave for a maximization problem or convex for a minimization problem.

Recommendations for Further Research

Generalized Constraints

In the proof of the recursive search algorithm, it was demonstrated that integer solutions can introduce non-concavity when the constraints are not restricted to specific forms. For the cases discussed in this thesis, the constraints must be of the form given in Equation (2-1).

In solving integer programming problems of the more general form given by Equation (1-1), the recursive search algorithm yielded the optimum solution in most cases. In some cases, however, the non-concavity problem discussed earlier was encountered and the algorithm did not reach the global optimum.

It is believed that further research could result in a set of more general rules under which the recursive technique would provide the optimum solution. This would allow the use of this algorithm for a wider class of integer programming problems.

Non-Concave Objective Function

From the mathematical proof of the recursive search technique, convergence to a global maximum was shown only for the case of a concave objective function. There are several "real-world" problems, however, where the return functions are neither convex nor concave, but are monotonic. The proof for the recursive search technique should be extended to determine convergence properties of the algorithm when only a monotonic objective function can be assumed.

SELECTED BIBLIOGRAPHY

(1) Güe, Ronald L., and M. E. Thomas. Mathematical Methods in Operations Research. London: The Macmillan Company, 1968.

(2) Baker, N. R., and J. S. Yormark. "Resource Allocation, Two-Dimensional Constraints and Discrete Dynamic Programming." Purdue Laboratory for Applied Industrial Control, Report 17 (1968).

(3) Hood, W. E., and T. C. Koopmans (eds.). Studies in Economic Method. Cowles Commission Monograph No. 14. New York: John Wiley and Sons, 1953.

(4) Lorie, J. H., and L. J. Savage. "Three Problems in Rationing Capital." Journal of Business (October, 1955), pp. 229-239.

(5) Weingarten, H. M. Mathematical Programming and the Analysis of Capital Budgeting Problems. New York: Prentice-Hall, 1963.

(6) Nemhauser, G. L. "A Note on Capital Budgeting." Journal of Industrial Engineering, Vol. XVIII, No. 6 (1969).

(7) Weingarten, H. M. "Capital Budgeting of Interrelated Projects: Survey and Synthesis." Management Science, Vol. 12, No. 7 (1966).

(8) Hillier, F. S., and G. L. Lieberman. Introduction to Operations Research. San Francisco: Holder-Day, 1967.

(9) Bellman, R. A. Dynamic Programming. Princeton: Princeton University Press, 1953.

(10) Wagner, H. M. Principals of Operations Research. Englewood Cliffs: Prentice-Hall, 1969.

(11) Bellman, R. A., and S. E. Dreyfus. Applied Dynamic Programming. Princeton: Princeton University Press, 1962.

(12) Nemhauser, G. L. Introduction to Dynamic Programming. New York: John Wiley and Sons, 1967.

(13) Dreyfus, S. E. "Dynamic Programming Solution of Allocation Problems." Rand Corporation Report P-1083 (May, 1957).

(14) Kalaba, R. "Some Aspects of Nonlinear Allocation Processes." Rand Corporation Report P-2715 (March, 1963).

(15) Bellman, R. A. "Dynamic Programming and Mathematical Economics." Rand Corporation Memorandum RM-3539-PR (March, 1963).

(16) Hess, S. W. "A Dynamic Programming Approach to R & D Budgets and Project Selection." IEEE Transactions on Engineering Management, Vol. EM-9 (December, 1962), pp. 170-179.

(17) Rosen, E. M., and W. E. Souder. "A Method for Allocating R & D Expenditures." IEEE Transactions on Engineering Management, Vol. EM-12, No. 7 (Sept., 1965), pp. 87-93.

(18) Hadley, G. Non-Linear and Dynamic Programming. Reading: Addison-Wesley, 1964.

APPENDIX A

COMPUTER CODE FOR ALLOCATION PROBLEM

```
0001          DIMENSION V(20,20),X(20),K(20),B(20),IT(20),R(20),TA(10),PA(10),
             1C1(20),C2(20),C3(20),FKJ(11)
0002          COMMON  /Y/N,C1,C2,C3 /Z/DELTA,NP,NTP,BA,PA,TA,X
0003          INTEGER V,X,DELTA,B,TA,PA,BA
0004          DATA V/400*0/,K/20*0/,IT/20*0/
        C     DIMENSIONED FOR 20 STAGE PROBLEM
        C
        C
        C*****************************************************************************
        C
        C     NUMBER STAGES AS FOLLOWS
        C
        C                          TIME PERIODS
        C
        C                          1    2    3
        C
        C                1         1    2    3
        C  PROJECTS      2         4    5    6
        C                3         7    8    9
        C
        C*****************************************************************************
        C     DELTA = SEARCH INCREMENT
0005          READ (5,500) DELTA
0006      500 FORMAT (I10)
        C     BA = TOTAL BUDGET ALLOCATION
0007          READ (5,510) BA
0008      510 FORMAT (I10)
        C     NP = NUMBER OF PROJECTS
        C     PA(I) = MAXIMUM ALLOCATION FOR PROJECT I
0009          READ(5,520) NP,(PA(I),I=1,NP)
        C     NTP = MAXIMUM NUMBER OF TIME PERIODS
        C     TA(I) = MAXIMUM ALLOCATION FOR TIME PERIOD 1
0010          READ (5,520) NTP,(TA(I),I=1,NTP)
0011      520 FORMAT (5I10)
0012          N=NTP*NP
0013          NP1=N+1
        C     C1,C2,C3 = CONSTANTS FOR RETURN FUNCTION
        C     R(I) = RETURN FUNCTION FOR STAGE I
        C          = C1(I)*X(I)/(C2(I)*X(I)+C3(I))
0014          READ (5,530) (C1(I),C2(I),C3(I),I=1,N)
0015      530 FORMAT (3F10.5)
0016          READ (5,540) (X(I),I=1,N)
0017      540 FORMAT (10I5)
0018          JCOUNT=0
        C     LOOP TRAP STOPS CALCULATIONS IF SOLUTION HASN'T
        C     CONVERGED BY ICOUNT
0019          ICOUNT=5000
0020          CALL XCST(NP1,&558,&558)
0021       10 J=1
0022       20 CONTINUE
0023          DO 30 I=1,N
0024       30 B(I)=V(J,I)
0025          JCOUNT=JCOUNT+1
```

```
0026            IF(JCOUNT.GE.ICOUNT) GO TO 555
0027            IF(IT(J).EQ.0) GO TO 40
0028            CALL XFCN(B,V1)
0029            CALL XFCN(X,V2)
0030            IF(V1.GT.V2) GO TO 130
0031         40 CONTINUE
0032            DO 50 I=1,N
0033         50 V(J,I)=X(I)
0034            IT(J)=1
0035         60 CONTINUE
0036            X(J)=X(J)+(-1)**K(J)*DELTA
0037            IF(X(J).GE.0) GO TO 80
0038            DO 70 I=1,N
0039         70 X(I)=V(J,I)
0040            GO TO 150
0041         80 CONTINUE
0042            CALL XCST(J,&130,&558)
0043         90 CONTINUE
0044            J=1
0045        100 CONTINUE
0046            JM1=J-1
0047            IF(JM1.EQ.0) GO TO 120
0048            DO 110 I=1,JM1
0049            K(I)=0
0050        110 IT(I)=0
0051        120 CONTINUE
0052            GO TO 20
0053        130 CONTINUE
0054            DO 140 I=1,N
0055            X(I)=V(J,I)
0056        140 CONTINUE
0057            IF(K(J).EQ.1) GO TO 150
0058            K(J)=1
0059            GO TO 60
0060        150 CONTINUE
0061            J=J+1
0062            IF(J.LE.N) GO TO 100
0063        160 CONTINUE
0064            DO 170 I=1,N
0065        170 X(I)=V(N,I)
0066            CALL XFCN(X,ANS)
0067            WRITE (6,300)
0068        300 FORMAT (1H1,40X,'ALLOCATION PROBLEM')
0069            WRITE (6,310) NP
0070        310 FORMAT (1H0,25X,'NUMBER OF PROJECTS ',I6)
0071            WRITE (6,320) NTP
0072        320 FORMAT (1H0,25X,'NUMBER OF TIME PERIODS ',I6)
0073            WRITE (6,330) BA
0074        330 FORMAT (1H0,25X,'TOTAL RESOURCE CONSTRAINT ',I6)
0075            WRITE (6,340) (I,PA(I),I=1,NP)
0076        340 FORMAT (1H0,40X,'PROJECT RESOURCE CONSTRAINTS',//,40X,'PROJECT',9X
               1,'CONSTRAINT',//,(40X,I4,15X,I4))
0077            WRITE (6,350) (I,TA(I),I=1,NTP)
```

```
0078      350 FORMAT (1H0,40X,'TIME PERIOD CONSTRAINTS',//,36X,'TIME PERIOD',9X
              1,'CONSTRAINT',//,(40X,I4,15X,I4))
0079          WRITE (6,360)
0080      360 FORMAT (1H0,20X,'************************     RESULTS      **********
              1**************')
0081          WRITE (6,370)
0082      370 FORMAT (1H0,40X,'OPTIMUM RESOURCE ALLOCATION')
0083          WRITE (6,380)
0084      380 FORMAT (1H0,25X,'PROJECT',18X,'TIME PERIOD')
0085          WRITE (6,390) (KJ,KJ=1,NTP)
0086      390 FORMAT (1H0,34X,9I9)
0087          DO 180 I=1,NP
0088          II=(I-1)*NTP+1
0089          JJ=I*NTP
0090          WRITE (6,400) I,(X(IJ),IJ=II,JJ)
0091      400 FORMAT (1H0,20X,I9,5X,9I9)
0092      180 CONTINUE
0093          WRITE (6,410) ANS
0094      410 FORMAT (1H0,//,40X,'OPTIMUM RETURN',F10.3)
0095          WRITE (6,360)
0096          WRITE (6,420)
0097      420 FORMAT (1H1,30X,'RETURN FUNCTIONS')
0098          DO 200 I=1,NP
0099          II=(I-1)*NTP
0100          WRITE (6,430) I
0101      430 FORMAT (1H0,30X,'PROJECT',I6)
0102          WRITE (6,440)
0103      440 FORMAT (1H0,5X,'ALLOCATION',5X,'TIME PERIOD 1',5X'TIME PERIOD 2',
              15X,'TIME PERIOD 3',5X,'TIME PERIOD 4')
0104          DO 200 KK=1,11
0105          KM1=KK-1
0106          DO 190 J=1,NTP
0107          L=II+J
0108      190 FKJ(L)=C1(L)*KM1/(C2(L)*KM1+C3(L))
0109          WRITE (6,450) KM1,(FKJ(L),L=1,NTP)
0110      450 FORMAT (1H ,5X,I5,4F18.4)
0111      200 CONTINUE
0112          GO TO 777
0113      555 WRITE (6,999) JCOUNT
0114      999 FORMAT (1H0,'STOPPED AT JCOUNT='I6)
0115          GO TO 777
0116      558 WRITE (6,996)
0117      996 FORMAT (1H0,'NO FEASIBLE SOLUTION')
0118      777 CONTINUE
0119          STOP
0120          END
```

```
FORTRAN IV G LEVEL  18              XECN              DATE = 71020          10/10/15              PAGE 0001

                    SUBROUTINE XECN(X,VAL)
                    DIMENSION X(20),C1(20),C2(20),C3(20)
                    COMMON  /Y/N,C1,C2,C3
                    INTEGER X
                    VAL=0.
                    DO 10 I=1,N
                 10 VAL=VAL+C1(I)*X(I)/(C2(I)*X(I)+C3(I))
                    RETURN
                    END
```

```
      SUBROUTINE XCST(J,*,*)
      DIMENSION X(20),TA(10),PA(10)
      COMMON  /Z/DELTA,NP,NTP,BA,PA,TA,X
      INTEGER X,TA,PA,BA,TPP,A,DELTA
      TPP=NTP*NP
      A=0
      DO 10 I=1,TPP
   10 A=A+X(I)
      KI=1
   20 IF(A.LE.BA) GO TO 50
   30 INX=KI
      IF(INX.GE.J) GO TO 160
      IF(INX.GT.TPP) GO TO 170
      X(INX)=X(INX)-DELTA
      IF(X(INX).GE.0) GO TO 40
      X(INX)=0
      KI=KI+1
      GO TO 30
   40 A=A-DELTA
      GO TO 20
   50 CONTINUE
      DO 100 I=1,NP
      II=(I-1)*NTP+1
      JJ=I*NTP
      A=0
      DO 60 K=II,JJ
   60 A=A+X(K)
      KI=0
   70 IF(A.LE.PA(I)) GO TO 100
   80 INX=II+KI
      IF(INX.GE.J) GO TO 160
      IF(INX.GT.JJ) GO TO 170
      X(INX)=X(INX)-DELTA
      IF(X(INX).GE.0) GO TO 90
      X(INX)=0
      KI=KI+1
      GO TO 80
   90 A=A-DELTA
      GO TO 70
  100 CONTINUE
      DO 150 I=1,NTP
      II=I+NTP*(NP-1)
      A=0
      DO 110 K=I,II,NTP
  110 A=A+X(K)
      KI=0
  120 IF(A.LE.TA(I)) GO TO 150
  130 INX=I+KI
      IF(INX.GE.J) GO TO 160
      IF(INX.GE.II) GO TO 170
      X(INX)=X(INX)-DELTA
      IF(X(INX).GE.0) GO TO 140
      X(INX)=0
```

```
            KI=KI+NTP
            GO TO 130
        140 A=A-DELTA
            GO TO 120
        150 CONTINUE
            RETURN
        160 RETURN 1
        170 RETURN 2
            END
```

ALLOCATION PROBLEM

NUMBER OF PROJECTS 1

NUMBER OF TIME PERIODS 4

TOTAL RESOURCE CONSTRAINT 10

PROJECT RESOURCE CONSTRAINTS

PROJECT	CONSTRAINT
1	10

TIME PERIOD CONSTRAINTS

TIME PERIOD	CONSTRAINT
1	4
2	4
3	4
4	4

************************ RESULTS ************************

OPTIMUM RESOURCE ALLOCATION

PROJECT	TIME PERIOD			
	1	2	3	4
1	2	1	4	3

OPTIMUM RETURN 12.675

************************ RESULTS ************************

RETURN FUNCTIONS

PROJECT 1

ALLOCATION	TIME PERIOD 1	TIME PERIOD 2	TIME PERIOD 3	TIME PERIOD 4
0	0.0	0.0	0.0	0.0
1	2.6190	3.5294	1.2444	1.2739
2	3.4375	3.8095	2.0741	2.0619
3	3.8372	3.9130	2.6667	2.5974
4	4.0741	3.9669	3.1111	2.9851
5	4.2308	4.0000	3.4568	3.2787
6	4.3421	4.0223	3.7333	3.5088
7	4.4253	4.0385	3.9596	3.6939
8	4.4898	4.0506	4.1481	3.8462
9	4.5413	4.0602	4.3077	3.9735
10	4.5833	4.0678	4.4444	4.0816

APPENDIX B

COMPUTER CODE FOR MANPOWER LEVELING PROBLEM

```
0001            DIMENSION V(20,20),X(20),IT(20),R(20),XL(20),XU(20),B(20),DEL(20),
               1K(20)
0002            INTEGER V,X,DELTA,XL,XU,DEL,B,R,XXL,XXU
0003            DATA V/100*0/,K/10*0/,IT/10*0/
          C     N = NUMBER OF TIME PERIODS
          C     DIMENSIONED FOR 20 STAGE PROBLEM
0004            READ (5,500) N
0005        500 FORMAT (I5)
0006            READ (5,505) NDELTA,(DEL(I),I=1,NDELTA)
0007        505 FORMAT (10I5)
          C     X = FEASIBLE STARTING SOLUTION VECTOR
0008            NP1=N+1
0009            DO 10 I=1,NP1
          C
          C     XL(I) = LOWER MANPOWER LIMIT FOR TIME PERIOD I
          C     R(I) = MANPOWER REQUIREMENT FOR TIME PERIOD I
          C     XU(I) = UPPER MANPOWER LIMIT FOR TIME PERIOD I
          C
          C     NUMBER TIME PERIODS AS 0,1,2,..., WITH ZERO AS INITIAL TIME
          C     PERIOD.  INITIAL TIME PERIOD HAS FIXED MANPOWER LEVEL.
          C
          C     USE ONE INPUT CARD FOR EACH TIME PERIOD, EACH CARD CONTAINING
          C     LOWER LIMIT, REQUIREMENT (IDEAL LEVEL) AND UPPER LIMIT,
          C     RESPECTIVELY
          C
          C     FOR PROBLEM WITH NO OVERTIME ALLOWED, SET XL(I) = R(I)
          C
          C     FOR FIRST CARD (TIME PERIOD ZERO), SET XL(0) = XU(0) = R(0)
          C
0010         10 READ (5,510) XL(NP1-I+1),R(NP1-I+1),XU(NP1-I+1)
0011        510 FORMAT (3I10)
0012            DO 20 I=1,NP1
0013         20 X(I)=R(I)
          C     R(I) = REQUIREMENTS FOR ITH TIME PERIOD
0014            B(NP1)=R(NP1)
0015            JCOUNT=0
0016            ICOUNT=3000
          C     SET ICOUNT AT REASONABLE NUMBER OF ITERATIONS.  LOOP TRAP
          C     STOPS CALCULATIONS IF SOLUTION HASN'T CONVERGED BY ICOUNT
0017            IX=1
0018         30 J=1
0019            DELTA=DEL(IX)
0020         40 CONTINUE
0021            DO 50 I=1,N
0022         50 B(I)=V(J,I)
0023            JCOUNT=JCOUNT+1
0024            IF(JCOUNT.GE.ICOUNT) GO TO 555
0025            IF(IT(J).EQ.0) GO TO 60
0026            CALL XFCN(B,N,R,V1)
0027            CALL XFCN(X,N,R,V2)
0028            IF(V1.LT.V2) GO TO 150
0029         60 CONTINUE
0030            DO 70 I=1,N
```

```
   70 V(J,I)=X(I)
      IT(J)=1
   80 CONTINUE
      X(J)=X(J)+(-1)**K(J)*DELTA
      IF(X(J).GE.0) GO TO 100
      DO 90 I=1,N
   90 X(I)=V(J,I)
      GO TO 170
  100 CONTINUE
      IF(X(J).LT.XL(J).OR.X(J).GT.XU(J)) GO TO 150
  110 CONTINUE
      J=1
  120 CONTINUE
      JM1=J-1
      IF(JM1.EQ.0) GO TO 140
      DO 130 I=1,JM1
      K(I)=0
  130 IT(I)=0
  140 CONTINUE
      GO TO 40
  150 CONTINUE
      DO 160 I=1,N
      X(I)=V(J,I)
  160 CONTINUE
      IF(K(J).EQ.1) GO TO 170
      K(J)=1
      GO TO 80
  170 CONTINUE
      J=J+1
      IF(J.LE.N) GO TO 120
      IF(IX.GE.NDELTA) GO TO 190
      DO 180 I=1,N
C     RE-SET LOWER AND UPPER LIMITS OF EACH DECISION VARIABLE
      XXL=V(N,I)-2*DELTA
      XXU=V(N,I)+2*DELTA
      XL(I)=MAX0(XL(I),XXL)
      XU(I)=MIN0(XU(I),XXU)
  180 CONTINUE
      IX=IX+1
      GO TO 30
  190 CONTINUE
      DO 200 I=1,N
  200 X(I)=V(N,I)
      CALL XFCN(X,N,R,ANS)
      WRITE (6,600)
  600 FORMAT (1H1,35X,'MANPOWER LEVELING PROBLEM')
      WRITE (6,610)
  610 FORMAT (1H0,//,10X,'TIME PERIOD',4X,'LOWER LIMIT',4X,'REQUIREMENT'
     1,4X,'UPPER LIMIT',4X,'OPTIMUM MANPOWER LEVEL')
      DO 210 I=1,NP1
      IM1=I-1
  210 WRITE (6,630) IM1,XL(NP1-I+1),R(NP1-I+1),XU(NP1-I+1),X(NP1-I+1)
  630 FORMAT (1H ,12X,I3,10X,I5,11X,I5,10X,I5,15X,I5)
```

```
      WRITE (6,640) ANS
  640 FORMAT (1H0,//,10X,'MINIMUM LEVELING COST =',F20.2)
       GO TO 777
  555 WRITE (6,999) JCOUNT
  999 FORMAT (1H0,'STOPPED AT JCOUNT='I6)
  777 CONTINUE
      STOP
      END
```

```
FORTRAN IV G LEVEL  18              XFCN            DATE = 71020         10/15/49            PAGE 0001

0001                  SUBROUTINE XFCN(X,N,R,VAL)
0002                  DIMENSION X(10),R(10)
0003                  INTEGER X,R
              C       VAL = RETURN FOR TRIAL SOLUTION X
0004                  VAL=0.
0005                  DO 10 I=1,N
0006               10 VAL=VAL+200.*(X(I)-X(I+1))**2+2000.*(X(I)-R(I))
0007                  RETURN
0008                  END
```

MANPOWER LEVELING PROBLEM

TIME PERIOD	LOWER LIMIT	REQUIREMENT	UPPER LIMIT	OPTIMUM MANPOWER LEVEL
0	255	255	255	255
1	220	220	500	247
2	240	240	500	244
3	200	200	500	247
4	255	255	500	255

MINIMUM LEVELING COST = 185200.00

VITA

Marion Lester Williams

Candidate for the Degree of

Doctor of Philosophy

Thesis: SOLUTION OF THE MULTIPLE-CONSTRAINT ALLOCATION PROBLEM USING RECURSIVE SEARCH DYNAMIC PROGRAMMING

Major Field: Engineering

Biographical:

Personal Data: Born in Abilene, Texas, December 1, 1933, the son of Mr. and Mrs. Lester Williams.

Education: Attended elementary school in Abilene, Texas; graduated from Abilene High School in June, 1952; received the Bachelor of Science degree from Texas A & M College, College Station, Texas in May, 1956, majoring in Aeronautical Engineering; received the Master of Science degree from the University of New Mexico in May, 1967, with a major in Mechanical Engineering; completed the requirements for the Doctor of Philosophy degree in May, 1971, with a major in Industrial Engineering and Management.

Professional Experience: Lieutenant, USAF, 1956-1959; Aerodynamist, Sandia Corporation, 1959-1961; Senior Weapon System Engineer, Naval Weapons Evaluation Facility, 1961-1966; Senior Operations Research Analyst, Joint Chiefs of Staff Joint Task Force II, 1966-1968; currently Physical Scientist, Field Command, Defense Atomic Support Agency.