NBER WORKING PAPER SERIES

THE DECISION-STATE METHOD: CONVERGENCE PROOF, SPECIAL APPLICATIONS, AND COMPUTATIONAL EXPERIENCE

V.K. Dharmadhikari*

Working Paper No. 94

COMPUTER RESEARCH CENTER FOR ECONOMICS AND MANAGEMENT SCIENCE
National Bureau of Economic Research, Inc.
575 Technology Square
Cambridge, Massachusetts 02139

July 1975

*NBER Computer Research Center. Research supported in part by National Science Foundation Grant DCR 70-03456-A04 to the National Bureau of Economic Research, Inc.

Abstract

This paper presents a new method for obtaining exact optimal solutions for a class of discrete-variable non-linear resource-allocation problems. The new method is called the decision-state method because, unlike the conventional dynamic programming method which works only in the state space, the new method works in the state space and the decision space. It generates and retains only a fraction of the points in the state space at which the state functions are discontinuous; and thus overcomes to some extent the curse of dimensionality. It carries the cumulative decision-strongs associated with these points, and thus avoids the back-tracking entailed by the conventional dynamic programming method for recovering the optimal decisions.

A concise and complete statement of the method is given in Algorithm 2 and it is proved that the algorithm finds all exact optimal solutions. In addition the method is adapted for solving some problems with special structures such as block-angular or split-block-angular constraints and the resultant substantial advantages are demonstrated. The performance of Algorithm 2 on many resource-allocations problems is reported, along with investigations on many tactical decisions which have substantial impact on the performance. The performance of the computer implementation of Algorithm 2 is compared with that of the MMDP algorithm and it showed that for the class of problems at which the two are aimed, the decision-state Algorithm 2 performed better than MMDP algorithm both in terms of storage requirement and solution time. In fact, it achieved an order of magnitude saving in storage requirement.

Contents

Tables

A large number of planning situations involve optimization of a sum of non-linear return functions of discrete-variables, such that the sums of non-linear, resource-consumption functions do not exceed the given resource-availabilities. Consider, for example, the following situations:

1. Design of multiple-reservoir, water-supply systems.
2. Deployment of self-contained manpower units for servicing a missile system subjected to constraints on personnel of different kinds, on equipment of different kinds, and constraints of logistics.
3. Selection of investment projects where, instead of mere acceptance/rejection, it is required to decide among various specified levels of projects in a given set.

1. Representation of the Problem.

Let I_+ denote the set of non-negative integers, and let R_+ denote the non-negative real line. Let functions $f_j(\cdot)$, $g_{ij}(\cdot)$ be defined as $f_j : I_+ \to R_+$, $g_{ij} : I_+ \to R_+$ for all i and j. Let the M-component vector $B=(b_1,b_2---b_M)$ represent the given resource availabilities, and M and N, the given positive integers. Then the problem can be represented as,

I. Maximize $\sum_1^N f_j(X_j)$

Subject to $\sum_1^N g_{ij}(X_j) \leq b_i, \quad 1 \leq i \leq M$

$X_j \in I_+, \quad 1 \leq j \leq N$

In this paper, we deal with the abov problem where $f_j(\cdot)$, $g_{ij}(\cdot)$ are assumed to be non-decreasing functions such that $\exists$ some real number $a \geq o \ni g_{ij}(a) > b_i$ for all i,j. Very often $f_j(\cdot)$ and $g_{ij}(\cdot)$ are polynomials or linear functions, and M is much smaller than N. Problem I is thus a special type of an integer programming problem.

2. Literature Survey

Two main categories of techniques that may be considered for solving the above problem are: 1) dynamic programming and 2) implicit enumeration. In this paper, we present a method that was derived from dynamic programming, but which also shares some characteristics of the implicit enumeration or directed tree search category. The standard dynamic programming procedure suffers from what Bellman [2] called, "The Curse of Dimensionality", arising from multiple constraints. For continuous variable problems, Larson's [3] State Increment Dynamic Programming method was able to handle at most 4 to 5 constraints by carrying out computations in what are called "blocks". In this method, as in other recent approaches such as those of Wong & Luenberger [4], Wong [5], and Yormark & Baker [6], the exhorbitant memory requirement is reduced at the cost of increased computation.

For the multiple-constraint, knapsack problems with zero-one variables, Weingartner and Ness [7], and Nemhauser and Ullman [8] developed specialized dynamic programming algorithms that perform much better than the conventional dynamic programming algorithm. In these algorithms, the dimensionality problem of conventional dynamic programming is mitigated by noting that the optimal return functions are monotone step functions and thus it is sufficient to record the optimal returns and decisions only at the points at which the step functions change value. Weingartner and Ness have also reported on various heuristics that can be used with their algorithm to achieve computational savings. They also consider elimination of solutions through bounding in addition to elimination through dominance. In a way, the theoretical work of Haymond [9] in the one-dimensional case, and some portions of the extensive paper on knapsack functions by Gilmore and Gomory [10] can be said to contain the germs of the ideas that are developed in [7], [8] cited above.

Although the algorithms in [7], [8] performed much better than conventional dynamic programming, they are quite inferior to other specialized algorithms such as Geoffrion's RIP-30 [11] designed for solving zero-one problems. Thus the main value of the ideas contained in these works is in how they can be applied to or extended for solving problems which are intractable to other known techniques.

Motivated by the above papers, Morin and Marsten [12] recently developed their Imbedded State Space Approach which can be considered as an extension of the ideas contained in those papers. It is aimed at solving general, non-linear, multi-dimensional knapsack problems. Later, an improved and somewhat more complete version of their algorithm was given in [13]. A different paper by Morin and Esogbue [14] gives a theoretical exposition of the method and its extension to a broader class of sequential decision problems with additive and multiplicative returns. Some computational results are given in [13], [14]. The Imbedded State Space Approach developed by Morin et al., it is seen, is conceptually similar to the decision-state approach developed in this dissertation. The implementations of the two basically similar approaches in the form of specific algorithms are, however, quite dissimilar in many important aspects.

3. The Decision-State Method

We now proceed to give an algorithmic statement of the decision-state method. By an algorithm we mean, in accordance with Knuth's [15] definition, a procedure consisting of a sequence of instructions which are unambiguous and executable such that theprocedure terminates in finite time.

In this section we define the symbols and operations used in the statement of the algorithm and its proof. The algorithm is stated in an easy to read FORTRAN-like language whose instructions are numbered statements in upper case letters. The instructions are preceded by explanatory comments distinguished by asterisks.

3.1 Definitions

Let the k-tuple $D = (x_1, x_2, \ldots, x_k)$ for $1 \leq k \leq N$, where all $x_j \epsilon I_+$, denote a particular set of values of the first k variables. For each decision vector D, the M-component vector-valued function ${}^kS(D)$ is defined as ${}^kS(D) = (s_1, s_2, \ldots, s_M) = S$ where each component is given by $s_i = \Sigma_1^k g_{ij}(x_j)$ for $1 \leq i \leq M$ and the scalar-valued function ${}^kv(D)$ is defined as ${}^kv(D) = \Sigma_1^k f_j(x_j) = v$. For a given D, the triplet (D,S,v) is said to form a list entry or "entry", where D is the decision vector, S is the state vector, and v is the objective value.

If there exist two list entries E' = (D', S', v') and E" = (D", S", v") such that $v' < v"$ and $S' \geq S"$ then (D', S', v') is said to be dominated by (D", S", v") and this is expressed as E" > E' or E' < E". It is established in Lemma 4 that a dominated entry (D', S', v') cannot be a part of an optimal solution because it can be replaced by (D", S", v") to increase the objective value. Hence, while searching for optimal solutions, only the undominated entries need be examined.

For an integer $k > 0$, given $D = (x_1, s_2, \ldots, x_k)$ and $x_{k+1} \epsilon I_+$, the notation (D, x_{k+1}) denotes the $(k + 1)$-tuple $(x_1, x_2, \ldots, x_k, x_{k+1})$. More generally, for $n > 0$, the notation $(D, x_{k+1}, \ldots, x_{k+n})$ denotes the $(k + n)$-tuple $(x_1, x_2, \ldots, x_{k+n})$. In the special case of $k = 0$ when D is vacuous the notation (D, x_{k+1}) denotes the one-tuple (x_1). Given an integer $x_{k+1} \epsilon I_+$ and a list L with k-component decision vectors, the notation (L, x_{k+1}) denotes a list which is identical with the list L except that in each entry (D, S, v) D is replaced by (D, x_{k+1}) and S and v are adjusted accordingly. Given a list L and an entry (D, S, v), the notation L.plus.(D, S, v) denotes the list containing all entries from L and the entry (D, S, v). Similarly, for a list L and an entry (D, S, v) which may or may not be in it, the notation L.minus. (D, S, v) denotes the list of all entries from L except the entry (D, S, v). Given a k-stage feasible entry (D, S, v), for $k < N$, we define its various order descendents as follows:

0-th order descendents

$$^{0}\mathrm{Desc}[(D,\ S,\ v)] = \{(D,\ S,\ v)\}$$

n-th order descendents for n = 1, 2, ..., N-k,

$$^{n}\mathrm{Desc}[(D,\ S,\ v)] = \left\{ (D',\ S',\ v') \;\middle|\; \begin{array}{l} D' = (D, x_{k+1}, \ldots, x_{k+n}),\ x_{k+j} \varepsilon I_{+},\ 1 \leq j \leq n, \\ S' = {}^{k+n}S(D') \leq B,\ v' = {}^{k+n}v(D') \end{array} \right\}.$$

In addition, the set of n-th order descendent entries of the entries in the k-stage list L is defined as

$$^{n}\mathrm{Desc}\ [L] = \bigcup_{E^{i} \varepsilon L} {}^{n}\mathrm{Desc}\ [E^{i}].$$

3.2 Decision-State Algorithm 2

In this section we develop and present Algorithm 2. It is divided in subsections, a subsection beginning with some motivation underlying the development. The steps of the procedure are preceded by asterisked comment statements.

```
* We begin with k = 0, and the list L containing one entry (D,S,v)
* where D is vacuous, and S and v are zero. L occupies one top
* location of the storage area, the rest of the locations are identi-
* fied as being empty.
```

Step 0: $k = 0$, $L = (-,0,0)$, KOUNTL = 1, KOUNTC = 1

3.2.1 Systematic Entry Generation

In Algorithm 1 we construct many entries (D,S,v) with $S \geq B$ which, upon testing for feasibility of state are then discarded. This is so because each value of X_{k+1} is combined with each entry in the k-th stage list of undominated entries, in order to insure that all feasible entries are constructed. By noting the non-decreasing property of the $g_{ij}(\cdot)$ functions, a new procedure is developed which tends to generate fewer infeasible entries. To achieve this, we first assume that the entries in L are in the non-increasing order of objective value. Then we combine each entry $E^{N1} \varepsilon L$ in a non-empty location [N1] starting with N1 = KOUNTC, with successively increasing values of $X_{k+1} \varepsilon I_+$. As soon as an infeasible entry is generated, the cycle is started all over for the next entry $E^{N1} \varepsilon L$ with N1 = N1-1; and so on. We describe below how the (k+1)- stage entry list C is generated from the k- stage list L. It is assumed that enough storage locations are available between the top and bottom of the storage area to carry out the procedure. Later in Chapter 4 we will deal with the question of precisely how many storage locations are needed in our specific computer implementation. For N1 an integer, [N1] denotes the N1-th location of the storage area and if [N1] is not empty then E^{N1} denotes the entry (D^{N1},S^{N1},v^{N1}) in location [N1].

* Steps 1 to 6 constitute the Systematic Entry Generation phase.

* We begin with list L occupying the upper KOUNTC locations of the

* storage area. It contains KOUNTL non-empty locations, containing

```
        * the k- stage entries interspersed with KOUNTC-KOUNTL empty
        * locations. The rest of the locations below [KOUNTC] are empty.
        * The entries are in monotonic order of the objective value so
        * that the top entry has the largest objective value. At the
        * end of the phase we will have generated list C of the (k+1)-
        * stage entries. C will be occupying the lower contiguous loca-
        * tions of the storage area.
        * We start with location KOUNTC at the bottom of the list L, with
        * the list C empty, the counter KOUNTC reset to zero and defining
        * $\ell$ to equal KOUNTL.
Step 1: N1 = KOUNTC, C = $\phi$, KOUNTC = 0, $\ell$ = KOUNTL
        * When the location [N1] is empty we go to the next location.
Step 2: IF [N1] IS EMPTY THEN GO TO 6
        * When N1 contains an entry, we refer to its three parts as
        * $E^{N1} = (D^{N1}, S^{N1}, v^{N1})$. We start the cycle by setting the new
        * stage-variable to zero.
Step 3: $X_{k+1} = 0$
        * Construct a (k+1)- stage triplet by combining $X_{k+1}$ with $E^{N1}$.
Step 4: $D = (D^{N1}, X_{k+1})$, $S = {}^{k+1}S(D)$, $v = {}^{k+1}v(D)$
        * The new triplet (D,S,v) is added at the lowermost empty location
        * in the storage area if and only if S is feasible. It is placed
        * above the previous entry, if there was one. The count KOUNTC
        * of the entries in C is now incremented and so is the value of
        * $X_{k+1}$.
Step 5: IF $S \le B$ THEN PLACE (D,S,v) IN LOWERMOST EMPTY LOCATION,
        KOUNTC = KOUNTC + 1, $X_{k+1} = X_{k+1} + 1$, GO TO 4
        * When S is infeasible we go to the next location from list L.
Step 6: N1 = N1 - 1,
        IF N1 > 0 THEN GO TO 2
        * When N1 equals zero, the Systematic Entry Generation phase is
        * complete. The entries in the lower contiguous locations con-
        * stitute list C. All the feasible entries in C which are de-
        * scendents of an entry in L are said to constitute a sublist.
        * The $\ell$ entries from L will give rise to $\ell$ sublists. Let the
        * sublists be numbered in the order in which these were generated.
        * $SL_1$ will be the sublist occupying the lowermost locations and
        * will consist of the feasible descendents of the lowermost entry
        * from L with the smallest objective value. The top sublist will
        * be $SL_\ell$. We define C(k+1) = C for the current value of k and
        * the current list C.
```

3.2.2 Merging

In this phase we merge the sublists comprising C into one merged list A. The task of this phase is similar to that in a situation where a given set of numbers is to be arranged in descending order. There are a number of ways (See Knuth [23]) this can be done, some being more suited for some conditions than others. We develop here a new merging procedure that exploits the particular situation at hand. Let the number of entries in the sublists $SL_1, SL_2, \ldots, SL_\ell$ be denoted, respectively, by $P(1), P(2), \ldots, P(\ell)$. Let $P = \text{MAX}\ \{P(i),\ 1 \leq i \leq \ell\}$. In Lemma 1 we prove that only P additional locations are sufficient for merging these sublists by the procedure given below. We assume that P empty locations are available above the uppermost sublist SL_ℓ. Again, how this affects the precise size required of the storage area in our computer implementation is dealt with in Chapter 4.

```
      * Steps 7 to 13 constitute the Merging phase. From the non-de-
      * creasing property of $f_k(\cdot)$, we know that the entries within each
      * sublist are in monotonic order of the objective value, with the
      * smallest entry at the bottom. Thus, in order to obtain one or-
      * dered list we need to merely merge these sublists. This we do
      * by first copying $SL_\ell$ in the upper locations of the storage area
      * and calling it list A. Then we merge A with $SL_{\ell-1}$, if it exists,
      * and again call the result, list A. Then we merge A with $SL_{\ell-2}$,
      * and so on. We begin by copying sublist $SL_\ell$.
Step 7: COPY SUBLIST $SL_\ell$ IN LOCATIONS [1] TO [P(1)] PRESERVING THE
        ORDER OF THE ENTRIES, KOUNTA = $SL_\ell$
      * When there is only one sublist we immediately go to the next
      * stage.
Step 8: IF $\ell = 1$ THEN GO TO 31
      * Initialize the counters for the first merge.
Step 9: $M1 = \ell-1$, $N1 = P(\ell)$, $N2 = P(M1)$, $N3 = 0$, $N4 = 0$
      * The list occupying the upper contiguous locations will be called
      * A. At this point it consists of only the entries from $SL_\ell$. At
      * the end of the first merge it will be the resultant ordered
      * list of the merge of $SL_\ell$ and $SL_{\ell-1}$, and so on.
      * In this step we identify the current lowest entries from the
```

* current list A called the initial list A and the portion re-
* maining currently of sublist SL_{M1} and also identify their com-
* ponents with specific labels.

Step 10: $E^A = (D^A, S^A, v^A)$ IS THE LOWEST REMAINING ENTRY IN THE INITIAL LIST A, $E^{M1} = (D^{M1}, S^{M1}, v^{M1})$ IS THE LOWEST REMAINING ENTRY IN SL_{M1}

* Now we compare v^A with v^{M1} and remove the entry with the smaller
* value and transfer it to a new location in the merged list A.
* Increment the counters and repeat until the sublist is exhausted.

Step 11: IF $v^A \leq v^{M1}$ THEN REMOVE E^A FROM THE INITIAL LIST A,
TRANSFER E^A TO [N1+N2-N3] IN THE MERGED LIST A, N3 = N3 + 1,
IF $v^A \geq v^{M1}$ THEN REMOVE E^{M1} FROM SL_{M1},
TRANSFER E^{M1} TO [N1+N2-N3] IN THE MERGED LIST A, N3 = N3 + 1,
N4 = N4 + 1

* As long as the counter N4 of the number of entries removed from
* SL_{M1} is smaller than P(M1), we repeat with the new lowest entries,
* E^A in the initial list A and E^{M1} in the remaining sublist SL_{M1}.

Step 12: IF N4< P(M1) THEN GO TO 10

* When N4 = P(M1) the merged list A contains the result of the
* merge of initial list A and SL_{M1}. If any more sublists remain
* to be merged, we reinitialize the counters and start again with
* the next sublist and current A as the initial list A.

Step 13: M1 = M1-1, N1 = N1 + P(M1+1), N2 = P(M1), N3 = 0, N4 = 0,
IF M1> 0 THEN GO TO 10

* When M1 = 0 all the sublists will have been merged into list A

* which will now contain all the KOUNTC entries arranged in
* monotonic order with the smallest objective value at the bot-
* tom in [KOUNTC]. We set counter KOUNTA to current value of
* KOUNTC. In order to avoid the substantial computations in
* the Identification and Elimination phases, when no greater
* than 10 new entries are generated in C as compared to those
* in L, we bypass these phases.

Step 14: KOUNTA = KOUNTC, IF (KOUNTC-KOUNTL)$\leq$ 10 THEN GO TO 31

3.2.3 Identification

In this phase we identify certain entries as distinguished entries, which are potentially likely to be found dominated, i.e., the non-distinguished entries cannot be dominated. These propositions are proved later in Section 3.3. As the distinguished entries are identified, certain M-component vectors called T vectors are stored as marker vectors in association with some entries. These are used in the Elimination phase. We assume that enough storage locations are available to store the marker vectors.

* Steps 15 to 21 constitute the Identification phase. For
* N1 = 1,2,..., KOUNTC we compute recursively an M-component
* T vector for each entry $E^{N1} \varepsilon A$. The i-th component T_i of the
* T vector is defined as the smallest of the i-th components of
* the state vectors S^{M1} of the entries in A where M1$\leq$ N1. If
* the N1-th T vector does not exceed S^{N1+1} in any component,
* then E^{N1+1} is identified as a distinguished entry. For a

```
        * prespecified positive integer t, every t-th vector is stored
        * as a marker vector and the entry is marked to show this.
        * We begin by setting the initial T vector equal to the right
        * hand side vector B.
Step 15: $T = B$, $N1 = 0$, $M1 = 0$
        * Increment the counters.  If all entries have been examined then
        * go to the Elimination procedure, otherwise proceed to the next
        * step.
Step 16: $N1 = N1+1$, $M1 = M1 + 1$,
         IF $N1 > KOUNTC$ THEN GO TO 22
        * Test if any component of $S^{N1}$ is smaller than the corresponding
        * component of the current T vector.  When it is, then $E^{N1}$ cannot
        * be dominated by entries above it.
Step 17: IF $\exists i$, $1 \leq i \leq M$  SUCH THAT $S_i^{N1} \leq T_i$ THEN GO TO 19
        * When $S^{N1} \geq T$  then it is possible for $E^{N1}$ to be dominated.
Step 18: IDENTIFY $E^{N1}$ AS A DISTINGUISHED ENTRY, GO TO 16
        * Update the T vector.
Step 19: $T = (T_1, T_2, \ldots, T_M)$ WHERE $T_i = \text{MIN}(S_i^{N1}, T_i)$
        * Store every t-th vector as a marker vector.
Step 20: IF $M1 = t$ THEN STORE T AS A MARKER VECTOR IN ASSOCIATION WITH
         ENTRY $E^{N1}$, $M1 = 0$
Step 21: GO TO 16
```

3.2.4 Elimination

In this phase we detect which of the distinguished entries are actually dominated, and then eliminate these by identifying the entry location as being empty. The marker vectors stored in association with some of the entries in the Identification phase are used to reduce the number of entry comparisons whenever possible, making use of Lemma 3 in Section 4.

```
        * Steps 22 to 30 constitute the Elimination phase. When there
        * are no distinguished entries we bypass the Elimination phase.
Step 22: IF THERE ARE NO DISTINGUISHED ENTRIES IN A THEN GO TO 31
        * If there exist any distinguished entries, we start with the
        * lowest one, with the smallest objective value.
Step 23: N1 = THE LOCATION OF THE LOWEST DISTINGUISHED ENTRY, M1 = N1-1
        * We go the next M1 if location [M1] is empty or if M1 = 0.
Step 24: IF (M1 = 0 OR [M1] IS EMPTY) THEN GO TO 29
        * When there is a marker vector in association with $E^{M1}$, we
        * identify it with a label.
Step 25: IF THERE IS NO MARKER VECTOR IN ASSOCIATION WITH $E^{M1}$ THEN
         GO TO 27,
         OTHERWISE LET TM BE THE MARKER VECTOR
        * If any component of the state of $E^{N1}$ is smaller than the re-
        * spective component of TM, then $E^{N1}$ cannot be dominated by any
        * further entries above $E^{M1}$; therefore, we proceed with the next
        * distinguished entry.
Step 26: IF $\exists i$, $1 \leq i \leq M$ SUCH THAT $S_i^{N1} < TM_i$ THEN GO TO 30
        * In order to ensure that all alternative optimal solutions are
        * obtained, entries with equal objective value are not compared
        * for dominance.
Step 27: IF $v^{N1} = v^{M1}$ THEN GO TO 29
        * When $S^{N1} \geq S^{M1}$ then $E^{N1}$ is dominated by $E^{M1}$ and we eliminate
        * $E^{N1}$ by identifying [N1] as being empty and decrease the counter
        * KOUNTA by 1.
Step 28: IF $\exists i$, $1 \leq i \leq M$, SUCH THAT $S_i^{N1} \leq S_i^{M1}$ THEN GO TO 29,
         OTHERWISE ELIMINATE $E^{N1}$ AND IDENTIFY [N1] AS BEING EMPTY,
         KOUNTA = KOUNTA-1
        * We prepare to examine the next upper location.
Step 29: M1 = M1-1,
         IF M1 >0 THEN GO TO 24
        * At this point we are finished with the distinguished entry in
        * [N1]. The elimination phase is complete if there is no distinguished
        * entry above [N1], Otherwise we reinitialize N1, M1 for the next
        * cycle.
Step 30: IF $E^{N1}$ WAS THE UPPERMOST DISTINGUISHED ENTRY THEN GO TO 31,
         OTHERWISE N1 = THE LOCATION OF THE NEXT DISTINGUISHED ENTRY
         ABOVE CURRENT [N1], M1 = N1-1, GO TO 24
        * At this point A occupies the upper KOUNTC locations and contains
        * KOUNTA non-empty locations, each in an undominated (k+1)- stage
        * entry. Now we rename this list as L, reinitialize KOUNTL with
        * the current value of KOUNTA, increment the stage number and
        * identify all locations below [KOUNTC] as being empty, and go back
        * to Systematic Entry Generation for the next stage if k is less
```

```
          * than  N, the number of stage-variables in the given problem.
          * Define L (k+1) = A for the current value of k and the current
          * list A.
Step 31: L = A, KOUNTL = KOUNTA, k = k+1, IF k< N THEN GO TO 1
          * When k equals N, the final list L of undominated entries con-
          * tains all the optimal entries.  Since the entries are in mono-
          * tonic order of the objective value, with the top entry having
          * the largest value, the optimal entries can be easily picked
          * up from the top.
          * We start with the top entry $E^1 = (D^1,S^1,v^1)$
Step 32: N1 = 1, OPTLIST = {$E^{N1}$}, OPTVAL = $v^{N1}$, N2 = 1
          * When the next entry below has equal objective value, we add
          * it to OPTLIST.  Otherwise we terminate the algorithm.
Step 33: N2 = N2 + 1
         IF N2> KOUNTL THEN GO TO 35
Step 34: IF $v^{N2} = v^{N1}$ THEN OPTLIST = OPTLIST.PLUS.$E^{N2}$, GO TO 33
          * When $v^{N2}$  $v^{N1}$ then OPTLIST will contain all the entries yield-
          * ing the optimal value OPTVAL.  The decision vectors D  of the
          * entries $(D,S,v) \varepsilon$ OPTLIST are the optimal solutions to the given
          * problem 1.
Step 35: END
```

4. Proof of Convergence for Algorithm 2

In this section we will establish that the 35-step procedure presented in Section 3.1 achieves what it sets out to do. First we will prove in Theorem 1 that the procedure can properly be called an algorithm in the sense that each step of the procedure is unambiguous and executable, and that the procedure terminates finitely. Then we will prove four lemmas that will help in proving Theorem 2 which states that Algorithm 2 finds all optimal solutions to the given Problem 1.

Theorem 1: The procedure of section 3 is an algorithm.

Proof: The procedure involves operations such as comparisons and additions of values, additions or deletions of list entries, storing entries in storage locations, identifying or marking storage locations, and so on, which are clearly executable. Thus we need only to establish finiteness. The Systematic Entry Generation phase starts with stage-number $k = 0$ and the list L containing only one entry. For each entry in the list L, we start with $X_{k+1} = 0$ and at each pass through the loop formed by Step 4 and Step 5 we increment X_{k+1} by 1. Since the functions $g_{ij}(\cdot)$ are such that there exists a non-negative real number a for which $g_{ij}(a) > b_i$ for all i and j, the termination from this loop occurs in a finite number of repetitions. Since we start with a finite number of entries in the list L, termination from the Systematic Entry Generation loop formed by Step 2 and Step 6 occurs finitely, and the list C contains a finite number of entries.

The number of sublists to be merged is finite, hence termination from the merging procedure occurs finitely. In the Identification procedure each entry is compared once and only once with the recursively computed T vector, hence termination from this procedure occurs finitely and the number of entries distinguished remains finite. In the Elimination procedure, each distinguished entry is compared with only a finite number of entries above it, hence termination from the Elimination procedure occurs finitely with the new list L containing only a finite number of entries.

In Step 31 the stage-number k is incremented by 1, and the next cycle through the procedure begins all over. Since we started with $k = 0$, and since k is incremented at each cycle, the termination through the largest loop, formed by Step 1 and Step 31, occurs finitely, after the N-th pass. Exit from the loop formed by Step 33 and Step 34 occurs finitely because the loop starts with $N2 = 1$, increments N2 by 1 at each pass, and since there are a finite number, KOUNTL entries in the list L. Thus termination from the entire procedure occurs finitely. ▽▽

Lemma 1 establishes that for the Merging procedure of Section 3.2.2 only P additional storage locations are sufficient to achieve a complete merge without losing any of the entries. This is done by showing that every entry is transferred to a location that has been made empty before the transfer. Lemma 2 helps us in determining some entries that cannot be dominated by any entries in the list. By the use of Lemma 2, in the Identification procedure we identify some entries from the list A as those that can possibly be dominated by some other entries. Lemma 3 helps us in reducing the number of entry comparisons in the process of determining if a distinguished entry is actually dominated.

Whenever we find a marker vector, at least one component of which exceeds the corresponding component of the state vector of the distinguished entry, we can stop the entry comparisons because Lemma 3 proves that no further comparisons can show that the distinguished entry is dominated. Lemma 4 establishes that any descendent of a dominated entry cannot be optimal.

Lemma 1: Suppose there are ℓ sublists $SL_1, SL_2, \ldots, SL_\ell$. Each sublist is in monotonic order of the objective value with the smallest entry at the bottom. Suppose $P(1)$, $P(2), \ldots$, $P(\ell)$ are the counts of the number of entries in the sublists, respectively; and that P is the largest of these numbers. Then the P additional storage locations attached above the topmost list SL_ℓ are sufficient to completely merge the sublists.

Proof: The procedure starts by copying $P(\ell)$ entries from SL_ℓ into the upper locations of the P additional locations attached and $P(\ell) \leq P$. This leaves $P - P(\ell)$ empty locations. The first merge of $SL_{\ell-1}$ with SL_ℓ entries will transfer entries to the uppermost $P(\ell) + P(\ell-1)$ locations, which number is no larger than the number $P + P(\ell)$ of locations available for the resultant list A since $P\ (\ell-1)\ \ P$. Thus $P(\ell)+P(\ell-1) \leq P+P(\ell)$. Moreover, no entry from $SL_{\ell-1}$ is transferred to a location that was occupied by an entry from SL_ℓ, until the latter was removed from it. The same holds for the next merge because $P(\ell) + P(\ell-1) + P(\ell-2) \leq P + P(\ell) + P(\ell-1)$. This continues to hold for all merges because $P(\ell)+P(\ell-1)+\ldots+P(\ell-n) \leq P +P(\ell)+\ldots+P(\ell-n+1)$ for $1 \leq n \leq \ell-1$. ▽▽

<u>Lemma 2</u>: In the Identification procedure, if there is an entry $E^{N1} = (D^{N1},S^{N1},v^{N1}) \varepsilon A$ such that a component of S^{N1} is smaller than the corresponding component of the (N1-1)-th T vector, then the entry E^{N1} cannot be dominated by any entry in the list A.

<u>Proof</u>: From the procedure, we see that the recursion for the i-th component, $1 \leq i \leq M$, of the (N1-1)-th T vector is given by $T_i = \text{MIN}(S_i^{N1-1},T_i)$. Thus $T_i = \text{MIN}\ \{S_i^1,S_i^2,\ldots,S_i^{N1-1}\}$. Now for $M1 < N1$, $v^{M1} > v^{N1}$ from the Merging procedure, and there exists an i such that $S_i^{N1} < T_i \leq S_i^{M1}$ for all $M1 < N1$. Therefore E^{N1} cannot be dominated by E^{M1} for $M1 < N1$. But for $M1 \geq N1$, $v^{M1} \leq v^{N1}$, therefore E^{N1} cannot be dominated by E^{M1} for $M1 \geq N1$.$\nabla\nabla$

<u>Lemma 3</u>: In the Identification procedure, suppose TM is a marker vector stored in association with the M1-th entry E^{M1} and that E^{N1} is a distinguished entry where $M1 < N1$. Suppose also that a component of S^{N1} is smaller than the corresponding component of TM. Then E^{N1} cannot be dominated by any entries above E^{M1}.

<u>Proof</u>: By its construction, we know that $TM_i = \text{MIN}\ \{S_i^1,S_i^2,\ldots,S_i^{M1}\}$. We are given that there exists an $i, 1 \leq i \leq M$ such that $S_i^{N1} < TM_i$. Therefore, $S_i^{N1} < TM_i \leq S_i^{M2}$ for all integers $M2 \leq M1$. From the Merging procedure we know that $v^{N1} \leq v^{M2}$ for $M2 \leq M1$. Hence E^{N1} cannot be dominated by any entry above E^{M1}. $\nabla\nabla$

<u>Lemma 4</u>: Let $E^1 \varepsilon C(N)$ and $E^2 \varepsilon C(k)$ for $k \leq N$. Suppose $E^1 \varepsilon\ {}^{N-k}\text{Desc}\ [E^2]$. If there exists an entry $E^3 \varepsilon C(k)$ such that $E^3 > E^2$, then E^1 cannot be optimal.

Proof: First, suppose that $k = N$. Then $E^2 \varepsilon C(N)$, $E^3 \varepsilon C(N)$, and $E^1 \varepsilon {}^0\text{Desc}\ [E^2]$. Since $E^3 > E^2$, $v^3 > v^2 = v^1$ and hence E^1 cannot be optimal. Now suppose $k < N$. Since $E^1 \varepsilon {}^{N-k}\text{Desc}\ [E^2]$, we have $X_j = X_j^2$ for $j = 1,2,\ldots k$. Since $E^2 < E^3$, $S_i^3 \leq S_i^2 = \Sigma_{j=1}^{j=k}\, g_{ij}(X_j^1)$ for $i = 1,2,\ldots,M$. Now construct a new entry $E^4 = (D^4, S^4, v^4)$, $D^4 = (X_1{}^4, X_2{}^4, \ldots, X_N{}^4)$ where $X_j{}^4 = X_j{}^3$ for $j = 1,2,\ldots,k$ and $X_j{}^4 = X_j{}^1$ for $j = k+1, k+2, \ldots, N$. For this entry, for $i = 1,2,\ldots,M$ we have

$$
\begin{aligned}
S_i{}^4 &= \Sigma_{j=1}^{j=N}\, g_{ij}(X_j{}^4) \\
&= \Sigma_{j=1}^{j=k}\, g_{ij}(X_j{}^4) + \Sigma_{j=k+1}^{j=N}\, g_{ij}(X_j{}^4) \\
&= \quad S_i{}^3 \qquad + \Sigma_{j=k+1}^{j=N}\, g_{ij}(X_j{}^1)
\end{aligned}
$$

Therefore, $S_i{}^4 \leq \Sigma_{j=1}^{j=k}\, g_{ij}(X_j{}^1) + \Sigma_{j=k+1}^{j=N}\, g_{ij}(X_j{}^1) = S_i{}^1 \leq B$, and entry E^4 is feasible. Now

$$
\begin{aligned}
v^4 &= \Sigma_{j=1}^{j=N}\, f_j(X_j{}^4) \\
&= \Sigma_{j=1}^{j=k}\, f_j(X_j{}^4) + \Sigma_{j=k+1}^{j=N}\, f_j(X_j{}^4) \\
&= \Sigma_{j=1}^{j=k}\, f_j(X_j{}^3) + \Sigma_{j=k+1}^{j=N}\, f_j(X_j{}^1)
\end{aligned}
$$

Since $E_2 < E_3$, $\Sigma_{j=1}^{j=k}\, f_j(X_j{}^3) = v^3 > v^2 = \Sigma_{j=1}^{j=k}\, f_j(X_j{}^2) = \Sigma_{j=1}^{j=k}\, f_j(X_j{}^1)$

Therefore, $v^4 > \Sigma_{j=1}^{j=k}\, f_j(X_j{}^1) + \Sigma_{j=k+1}^{j=N}\, f_j(X_j{}^1) = v^1$.

Thus (D^4, S^4, v^4) is a feasible N-stage entry with greater objective value than that of E^1. Therefore, E^1 cannot be optimal. $\nabla\nabla$

Theorem 2: Algorithm 2 finds all optimal solutions to the given Problem 1.

Proof: In Theorem 1 we proved that the procedure in Section 3.3 termed Algorithm 2 can properly be called an algorithm. In Lemma 1 we proved that the P additional storage locations attached above the top sublist SLℓ, where P is the largest of the sublist counts $P(1), P(2), \ldots, P(\ell)$, are sufficient for the Merging procedure in that all the entries in the ℓ sublists are preserved and arranged in one contiguous list A in monotonic order at the end of the merge. Lemma 2 and Lemma 3 prove that the Elimination procedure removes all the dominated entries from the list A and only the dominated entries from the list A. Lemma 4 proves that no descendent of a dominated entry can be optimal, hence no such descendents need be generated nor kept in the search for the optimal solutions. Using all the above, we prove below that the final list L(N) produced by the procedure of Section 3.2.4 contains all the optimal list entries having the optimal objective value. Let F and F_0 denote, respectively, the sets of feasible and optimal entries for the N-stage Problem 1. Symbolically,

$$F = \left\{ (D,S,v) \;\middle|\; \begin{array}{l} D = (X_1, X_2, \ldots, X_N),\ X_j \varepsilon J_+,\ 1 \leq j \leq N, \\ S = {}^N S(D),\quad v = {}^n v(D),\quad S \leq B \end{array} \right\},$$

$$F_0 = \left\{ (D,S,v) \;\middle|\; \begin{array}{l} (D,S,v) \varepsilon F,\ v \geq v^1 \\ \text{for any } (D^1, S^1, v^1) \varepsilon F \end{array} \right\}.$$

Clearly $F_0 C F$. From the definition of the descendents we know that for $k_1 = 1,2,\ldots,N-1$, $k_2 = 1,2,\ldots,N-k_1$, ${}^{k_2}\text{Desc}\,[C(k_1)] = {}^{k_2-1}\text{Desc}\,[{}^1\text{Desc}]$

$[C(k_1)]]$, ^{k_2}Desc $[C(k_1)]$ = ^{k_2}Desc $[L(k_1)]$ U ^{k_2}Desc $[C(k_1)$.minus.$L(k_1)]$ and ϕ = ^{k_2}Desc $[L(k_1)] \cap {}^{k_2}$Desc $C(k_1)$.minus.$L(k_1)$. At the end of the first pass through the Systematic Entry Generation phase, C(1) contains all feasible entries for the 1-stage problem, i.e., $C(1) = \{(D,S,v) / D = (X_1) \varepsilon I_+, S = {}^1S(D) \leq B, v = {}^1v(D)\}$. Therefore $F = {}^{N-1}$Desc $[C(1)]$. (Recall that $L(1) = C(1)$). At the end of the second pass through the Systematic Entry Generation phase, $C(2) = {}^1$Desc $[L(1)]$. Therefore, $F = {}^{N-1}$Desc $[L(1)] = {}^{N-2}$Desc$[{}^1$Desc $[L(1)]] = {}^{N-2}$Desc $[C(2)]$. Now $C(2) = L(2)$ U $(C(2)$.minus.$L(2))$. Thus, $F = {}^{N-2}$Desc $[L(2)]$ U ${}^{N-2}$Desc $[C(2)$.minus.$L(2)]$. From Lemma 4 we have, since $F_0 C F$, $F_0 C^{N-2}$Desc $[L(2)]$.

Proceeding in a similar manner, at the next pass we will obtain $F_0 C^{N-3}$Desc $[{}^1$Desc $[L(2)]] = {}^{N-3}$Desc $[C(3)] = {}^{N-3}$Desc $[L(3)]$ U ${}^{N-3}$Desc $[C(3)$.minus.$L(3)]$. This process can be continued till the (N-1)th pass to obtain $F_0 C^{N-(N-1)}$Desc $[L(N-1)]$. From this we get $F_0 C^1$Desc $[L(N-1)] = C(N)$. Thus all optimal entries are members of the final list C(N) and hence also of the final list L(N) of undominated entries. Since the optimal entries have the maximum objective value, they will be at the top of the list and thus OPTLIST will contain all of them. ▽▽

5. Computational and Storage Efficiency

As with any solution technique, the ultimate test of an algorithm is in its efficiency. It is not difficult to see how our algorithm is a substantial improvement over a standard dynamic programming algorithm (SDPA). SDPA requires as many locations for storing the state function table for each stage, as the total number of state values. For an M-dimensional resource allocation problem, this number is $\pi_{i=1}^{i=M}(b_i)$, and since it grows exponentially, it becomes too large even for a modern day computer when $M \geq 3$ for any realistic problem. Our algorithm avoids this excessive storage requirement by considering only those points in the state space at which the function changes value. Thus, whereas the standard dynamic programming algorithm examines all the lattice points of the state space, our method examines only a fraction of these points imbedded in this complete state space. Noting this, Morin and Marsten [13] have named the method, "The Imbedded State Space Approach".

More significant and, in practice, more important evaluation of our algorithm can be obtained by comparing it to the MMDP algorithm. Since both algorithms are aimed at solving the non-linear resource-allocation problems, and the performance of MMDP algorithm is reported on nine such problems, the identical nine problems were solved by using a computer implementation of our algorithm. These problems were constructed from Peterson's problems [16] and the data for these is given in Table 1. At the outset, it was evident that the algorithms are extremely sensitive to the sequence in which the stage variables of the problem are considered. For example, one 10-constraint, 28-variable problem took 39.9 seconds with one sequence and 8.5 seconds with another. The empirical performance reported below was obtained on the problems with the variables arranged in non-decreasing order of the peak resource consumption ratio defined below; the variable with the smallest consumption ratio was the first stage variable. The CPU time taken by the sequencing program is included in the solution time. The peak resource consumption ratio for the j-th variable of problem I is given by

$$\max_i \{g_{ij}(1)/b_i\}.$$

The data for the nine non-linear problems solved by MMDP algorithm is given in Table 1.

TABLE 1: INPUT DATA FOR NON-LINEAR PROBLEMS 1 TO 9

a_{ij}

$j \backslash i$	1	2	3	4	5	6	7	8	9	10	C_j
1	60	110	20	40	60	70	10	40	50	50	882
2	40	40	5	25	25	25	10	20	20	20	650
3	30	60	0	10	15	20	0	0	15	20	320
4	22	22	6	6	6	6	8	0	8	8	500
5	20	20	60	60	60	60	5	4	55	65	600
6	24	44	6	9	11	11	0	0	6	8	220
7	90	120	14	24	29	29	6	1	30	30	580
8	13	13	4	6	7	7	1	0	9	9	90
9	18	28	10	20	20	20	10	2	28	28	520
10	28	28	12	18	18	18	0	1	10	10	160
11	80	100	6	16	20	22	0	2	30	30	403
12	50	70	0	30	50	50	10	2	25	30	450
13	40	40	12	20	24	28	0	0	40	50	327
14	80	100	20	40	50	55	10	2	30	35	400
15	80	90	30	40	40	40	10	2	20	20	400
16	32	32	6	16	21	21	3	0	18	20	140
17	70	100	20	30	40	40	4	1	29	29	300
18	45	75	8	16	19	21	0	0	12	16	205
19	12	27	5	10	15	20	10	2	25	25	300
20	20	40	3	11	17	17	0	1	18	18	100
21	15	25	3	5	7	9	6	1	12	15	120
22	13	13	4	8	8	8	0	0	4	4	1200
23	12	12	6	10	13	13	0	0	2	2	600
24	64	75	18	32	42	48	0	0	0	8	2400
25	30	90	10	20	20	20	0	0	25	25	950
26	41	41	4	12	20	20	0	0	4	4	2000
27	50	80	40	50	55	55	10	3	50	55	1100
28	20	55	5	13	25	25	0	0	18	22	480
B	90	120	60	60	60	70	10	45	55	65	RHS

Table 2: General Non-linear Problems

Prob. No.	Storage in 60-bit words			Solution time in seconds of Cyber-72 CPU		
	MMDP Estimated	Algorithm 2	Difference	MMDP Equivalent	Algorithm 2	Difference
	(1)	(2)	$\frac{(1) - (2)}{(2)}$ %	(3)	(4)	$\frac{(3) - (4)}{(4)}$ %
1	3,600	320	1,025 %	3.8	2.6	46 %
2	5,140	385	1,185 %	5.2	3.6	45 %
3	10,280	960	970 %	11.3	6.2	82 %
4	6,180	320	1,831 %	3.8	2.5	52 %
5	5,620	400	1,305 %	5.2	3.3	57 %
6	8,940	770	1,061 %	10.9	6.3	73 %
7	3,830	320	1,096 %	4.1	2.6	57 %
8	4,130	360	1,047 %	5.2	3.4	53 %
9	6,220	640	871 %	9.5	5.3	79 %

Briefly, these problems are of the following form,

$$\text{Maximize} \quad \Sigma_{j=1}^{j=28} \; c_j \; f_j \; (x_j)$$

$$\Sigma_{j=1}^{j=28} \; a_{ij} \, g_j \; (x_j) \leq b_i, \quad i = 1,2,\ldots 10$$

$$x_j = 0, 1, 2, 3, 4, 5$$

where the functions f_j and g_j are chosen as x^2, x or $\sqrt{x}$. Problems 1 - 3 have $f_j \; (x_j) = \sqrt{x_j}$; 4 - 6 have $f_j \; (x_j) = x_j$; and 7 - 9 have $f_j \; (x_j) = x_j^2$. Problems 3, 6, 9 have $g_j \; (x_j) = \sqrt{x_j}$; 2, 5, 8 have $g_j \; (x_j) = x_j$; and 1, 4, 7 have $g_j = x_j^2$. Problems 1 to 9 correspond to MMDP problems 15 to 23, respectively, in [16].

The storage requirements and solution times (inclusive of the time taken by the sequencing program) for these problems are given in Table 7.1. The storage requirement for the MMDP algorithm depends on the maximum length UL of the list of undominated entries at any stage, and the total number LL of feasible list entries generated throughout. Since their paper [12] does not report these values, we estimate these by solving the problems by Algorithm 2 using the variables in the sequence in which they appear in the formulations provided by Morin and Marsten [12]. The solution times thus obtained are higher than those reported by Morin and Marsten which indicates that a sequence favorable to one algorithm is not necessarily favorable to the other. Our interest in this exercise was, however, to estimate the list lengths LL and UL that were obtained by the MMDP algorithm. When both algorithms use the same sequence of variables, the list lengths UL and LL yielded by Algorithm 2 provide good estimates for those yielded by the MMDP algorithm. Using these list length statistics, we estimate the storage requirement for the MMDP algorithm as follows. If M is the number of constraints, then each undominated list entry takes (2M + 4) computer words, in addition to the 2LL words needed to store the TRACE entries to enable retracing of the optimal solutions. Thus the MMDP storage requirement is given by (UL)(2M + 4) + 2LL. In Algorithm 2 however, the storage requirement depends on L, the maximum size of the list of feasible entries at any stage, and on $\lceil M/S \rceil$ where $\lceil A \rceil$ represents the smallest integer greater than or equal to A. In addition, we need memory space for storing the state of every tenth entry in the identification phase as mentioned in chapter 4. Thus the storage requirement for Algorithm 2 is given by $(L)(\lceil M/S \rceil + 1) + (\lceil L/10 \rceil)(\lceil M/S \rceil)$.

The computer programs for both algorithms were written in CDC's Extended FORTRAN and were compiled at optimization level 2, so that some code optimization was obtained. The solution times of the MMDP algorithm reported in [12] are based on a DCD-6400 computer., whereas those for Algorithm 2 are based on a Cyber-72. The Cyber-72 is similar to the CDC-6400 except that the CPU is slightly slower, being comparable to the CPU of a CDC-6200. According to a CDC manual [17], for some (Boolean) instructions the CDC-6200 takes 1.6 times the time taken by the CDC-6400, while for some others (Shift, Memory Access) the factor is 1.5, and finally for some arithmetic operations (Floating Multiply), the factor is 1.05. As a reasonable average factor we consider that 1 second of a CDC-6400 CPU is equivalent to 1.3 seconds of a CDC-6200 CPU. The running times reported in column (3) of Table 7.1 are then the equivalent Cyber-72 times for the times reported by Morin and Marsten.

6. Comparison of Algorithm 2 with the MMDP Algorithm

Both Algorithm 2 and the MMDP algorithm are aimed at solving general non-linear, resource-allocation problems involving integer-valued variables. Nine such problems were solved by Morin and Marsten [12]. The solution times reported in [12] are the smallest of the times obtained by using two variable sequencing heuristics and the time taken by the sequencing program is not included in reporting the solution times. For Algorithm 2 we solved these same problems using only the variable-sequencing heuristic given in Section 7.1, and the time taken by the sequencing program is included in the solution times reported. For these nine problems, Algorithm 2 performed better both in terms of high-speed storage requirement and in terms of solution time. Algorithm 2 saved an order of magnitude in storage and so appears significantly better in this respect. In terms of solution time, the MMDP algorithm took 45 to 82% more time than that taken by Algorithm2, but this time difference could be attributed to programming techniques, and is thus not as conclusive as the storage improvement.

7. Non-integrality.

The standard dynamic programming algorithm, as well as most other integer programming techniques demand that the resource availabilities b_i, and the values taken by the constraint functions $g_{ij}(\cdot)$ be integers. If, for example, b_i equals 203.443, or if one of the values taken by $g_{ij}(\cdot)$ is 19.4321, then the corresponding constraints will have to be multiplied with sufficiently large powers of 10 to make these values integers, accurate to desired number of significant digits. This would substantially increase the size of the right hand side values and thus the storage and computational requirements. In contrast to this, it is interesting to note that our algorithm is entirely insensitive to the non-integrality of these values. Instead of examining all lattice points of the state space as in SDPA, our algorithm examines only those points imbedded in the state space at which the state functions change value. Since the algorithm generates these points as it proceeds, instead of requiring these to be known a priori, the algorithm can handle the non-integral values without any difficulty.

8. Block-angular Constraint Set.

Suppose we have an M by N problem of a diagonal matrix structure depicted in Figure 1. This problem has a set of M_0 coupling constraints, and several blocks of constraints each of size M_i, i=1,2.... Let $M = M_0 + M_1 + \ldots$; $N = N_1 + N_2 + \ldots$; and assume that $M_0 + M_i \leq M^*$ for i = 1,2,.... Now consider applying our algorithm to this M by N problem. Each list entry, as usual, will have an M-component state vector. For the first block of N_1 stage variables x_j, $1 \leq j \leq N_1$, the state components S_i, $1 + M_0 + M_1 \leq i \leq M$, will all be identically equal to zero. Thus, the significant information in the state vectors can be stored by storing only the first $(M_0 + M_1)$ non-zero state components.

Now consider the second block of stage variables x_j, $1 + N_1 \leq j \leq N_1 + N_2$. For these stages, M_1 state components S_i, $1 + M_0 \leq i \leq M_0 + M_1$, will remain unchanged for all subsequent stages $j \geq 1 + N_1$. Thus, for stages $1 + N_1$ to $N_1 + N_2$, we do not need the state components S_i, $1 + M_0 \leq i \leq M_0 + M_1$. We need only the M_2 state components S_i, $1 + M_0 + M_1 \leq i \leq M_0 + M_1 + M_2$, in addition to the first M_0 state components of the coupling constraints, since the components S_i, $1 + M_0 + M_1 + M_2 \leq i \leq M$ are identically equal to zero.

Continuing in this manner, we see that during the stages in the k-th block, we need store and update only the M_k state components S_i, $1 + \Sigma_0^{k-1} M_i \leq i \leq \Sigma_0^k M_i$, in addition to the first M_0 state components corresponding to the coupling constraints. At the beginning of the next block, $j = 1 + N_1 + \ldots + N_k$, we can clear out the storage locations containing the M_k components S_i, $1 + \Sigma_0^{k-1} M_i \leq i \leq \Sigma_0^k M_i$ and use the same locations to store the newly active M_{k+1} components of S_i, $1 + \Sigma_0^k M_i \leq i \leq \Sigma_0^{k+1} M_i$. Hence, if $M_0 + M_1 \leq M^*$ for i = 1,2,..., then M* storage locations per list entry to store the active state components will be sufficient. Thus we will have effectively reduced the original M by N problem to an M* by N problem.

Resource-allocation problems of this special structure can occur, for example, in situations involving multiple time-period planning, or in situations involving variables that represent activities that use mutually exclusive classes of resources in addition to a few common resources that couple them together. In most such situations $\sum M_i = M$ is likely to be much larger than M*. Thus we can achieve substantial storage savings by recognizing and exploiting the block-angular constraint structure. Note also that in order to achieve this storage reduction, we did not have to incur any additional computation. In fact, by noting that many state components are inactive and hence need not be computed, we have actually reduced the amount of computation by a factor comparable to that for the reduction in storage. The above scheme of dealing with block-angular constraint structure was empirically tested on a few problems, and the substantial storage savings and some computational savings obtained are reported in 1 .

This can be easily generalized further to include the cases where the variables that use the same resources or have non-zero functions in the same constraints do not belong to adjacent blocks. Such a situation can occur in resource allocation and production scheduling problems where some resources are used by variables in more than one but not in all blocks. Usually such constraints will be lumped with the coupling constraints. We call such a constraint structure a split-block-angular structure and it is depicted in Figure 2. Our algorithm handles such a structure with only a slight additional bookkeeping work. For further details regarding the split-block-angular structure, see [1].

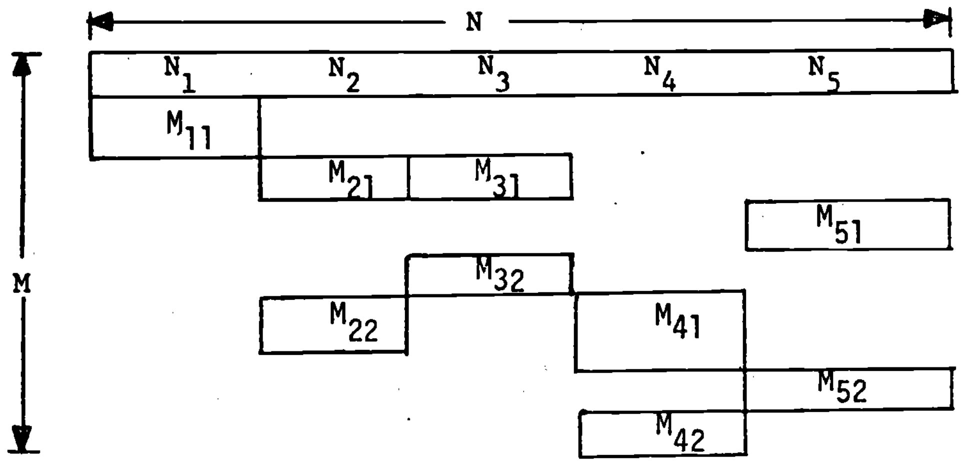

Figure 2

9. Mixed-Integer Problems

The decision-state method can very easily handle problems where some variables are discrete and some are continuous, as long as the variables are sum-separable. Consider the following mixed problem:

$$\text{Maximize} \quad \Sigma_1^N f_j(x_j) + \Sigma_1^{N'} f_j' (y_j')$$

$$\text{Subject to} \quad \Sigma_1^N g_{ij}(x_j) + \Sigma_1^{N'} g_{ij}'(y'_j) \leq b_i, \; 1 \leq i \leq M$$

$$x_j \in I_+, \qquad 1 \leq j \leq N$$

$$y_j' \in R_+, \qquad 1 \leq j \leq N'.$$

We will divide the mixed problem into two component problems.

Continuous-Component Problem: $\text{Max } \Sigma_1^{N'} f'_j(y'_j)$

$$\text{Subject to} \quad \Sigma_1^{N'} g'_{ij}(y'_j) \leq b_i, \; 1 \leq i \leq M$$

$$y'_j \in R_+, 1 \leq j \leq N'$$

Discrete-Component Problem: $\text{Max } \Sigma_1^N f_j(x_j)$

$$\text{Subject to } \Sigma_1^N g_{ij}(x_j) \leq b_i, \leq i \leq M$$

$$x_j \in I_+, 1 \leq j \leq N$$

The continuous-component problem can be solved parametrically on $b = (b_1, b_2 \ldots, b_M)$,the right-hand-side, i.e., the optimal solution can be obtained for the continuous-component problem for all feasible right-hand-side values ranging between zero and the given right-hand-side. This can be done by any conventional, non-linear programming techniques such as separable programming [18], [19]; a most advantageous technique which is inherently parametric on the right-hand-side is GLM [20]. In the simplest situation, which is perhaps the most prevalent in practice, the continuous-component problem is a linear programming problem. In this case, most production LP codes would be capable of providing the optimal solutions with some computational effort, for all feasible right-hand-sides.

The discrete-component problem, of course, can be solved by the decision-state method, which, being a dynamic programming method, automatically gives the optimal solution for all feasible right-hand side values. Let us denote the list of optimal solutions to the discrete-component problem as LD. Each entry in the list LD contains an optimal solution, the corresponding state vector and, the optimal value. Let LD be arranged in non-decreasing order of the optimal values. Then given any right-hand-side vector $b' \leq b$, the optimal solution to the discrete problem with right hand side b' is given by the entry with the largest objective value whose state vector does not exceed b' in any component. Similarly, let LC denote the list of optimal solutions to the continuous component-problem, parametrically on b, arranged in a manner similar to LD. Then we can obtain the optimal solution to the mixed problem by determining a combination of one entry from LD, $(D',S',v') \in$ LD, and one entry from LC, $(D'',S'',v'') \in$ LC, such that $v'+v'' \geq v^1+v^2$ for all entries $(D^1,S^1,v^1) \in$ LD and $(D^2,S^2,v^2) \in$ LC, and such that combination is feasible.

10. Bounding Elimination

Considering the given N-stage problem as a decision tree, each feasible decision vector $D = (x_1, x_2, \ldots, x_k)$ can be regarded as a node at the k-th echelon. The main thrust of the decision-state method is to generate only the feasible nodes on the undominated branches of the decision tree. When a list entry (D, S, v) corresponding to a node is found dominated, we know that the branches emanating from this node cannot be optimal for any member of the family of problems with identical $f_j(\cdot)$, $g_{ij}(\cdot)$ functions and with different right-hand sides. Typically, however, our lists will contain many entries that are undominated but are non-optimal for the particular problem being solved. It would be useful, therefore, if we find a way to detect and eliminate some of these 'dead-weight' entries in addition to eliminating those that are clearly dominated.

The directed-tree-search methods eliminate such nodes through the technique of bounding. That is, if the upper bound on the objective value for all nodes on the branches emanating from a given node is smaller than the value of the current best feasible solution, then the given node is said to be fathomed and can be eliminated. A similar approach can be used with our method. There are a number of ways in which an upper bound can be obtained. One simple way is to first identify one constraint, say the ℓ-th, as the tightest or the most binding constraint. Then the upperbound for a k-th stage node (D, S, v) is given by $v + \Sigma_{j=k+1}^{j=N} f_j\ (\overline{x}_j)$ where $\overline{x}_j$ is the greatest integer for which $g_{\ell j}(\overline{x}_j) \leq (b_\ell - s_\ell)$ for each j from k+1 to N. A better upper bound can be obtained through a little more computational effort as $v + \Sigma_{j=k+1}^{j=N} f_j(\overline{x}_j)$ where $\overline{x}_j$ for each j from k+1 to N is computed as the greatest integer for which $g_{ij}(x_j) \leq (b_i - s_i)$ for all i from 1 to M. The upper bounds obtained in these ways are likely to be loose. Thus, they will tend to be useful in eliminating a significant number of list entries when only the unproductive stage variables remain to be considered and most of the productive stage variables have already been considered. A workable strategy, then, may be to call upon bounding elimination only towards the tail end of the algorithm, the exact stage number dependent on the particular problem.

Of course, better and tighter bounds can be obtained by other sophisticated technqiues from the branch and bound category. A crucial question, however, is whether the computational effort expended in doing this would pay off in terms of reducing the entry lists and their computation. This area needs to be investigated empirically by solving problems of realistic sizes and structures. In the absence of such hard evidence, it seems that the option of bounding elimination should first be tried with the crude, easily obtainable upper bounds.

The above bounding elimination procedure assumes knowledge of a lower bound, that is, the value of the current best feasible solution. Of course, the entry in the current list with the largest objective value identified during the merging phase obviously gives such a lower bound. It may be possible, however, to improve this lower bound by using up the unused resource amounts. If the entry with the largest value is (D,S,v), then an improved lower bound can be obtained as $v + \sum_{j=k+1}^{j=N} f_j(\bar{x}_j)$ where $\bar{x}_{k+1}$ is the greatest integer such that $g_{i,k+1}(\bar{x}_{k+1}) \leq (b_i - s_i)$ for all i, and $\bar{x}_{k+2}$ is the greatest integer such that $g_{i,k+2}(\bar{x}_{k+2}) \leq b_i - s_i - g_{i,k+1}(\bar{x}_{k+1})$ for all i, etc. The larger the lower bound, and the smaller the upper bound associated with the nodes, the more list entries can be eliminated by this bounding elimination procedure. A good discussion of different bounding strategies, and their empirical evaluation can be found in [21].

11. Conclusions

In this paper we presented a new method for obtaining exact optimal solutions to certain types of discrete-variable, non-linear, resource allocation problems. The new method is called the decision-state method because, unlike the conventional dynamic programming method which works only in the state space, the new method works in the decision space as well as the state space. It generates and retains only a fraction of the points in the state space, thus overcoming much of the curse of dimensionality. It carries the cumulative decision strings associated with these points, and thus avoids the backtracking entailed by the conventional dynamic programming method for recovery of the optimal decisions at all the stages.

A concise yet complete and self-contained statement of the method was given in Chapter 3 in the form of an algorithm (Algorithm 2), and it was proven there that the algorithm indeed finds all exact optimal solutions to the given general, non-linear, resource allocation problem with discrete-value variables. In Chapter 5 we considered problems with special conditions such as block-angular or split-block-angular constraints, non-integrality and core limitations. We showed how the method could be specialized or adapted to accommodate effectively the above conditions. In Chapter 6 we considered such tactical options as sequencing the decision-variables, sequencing and inclusion of constraints, and bounding elimination.

Although an algorithmic statement of a problem-solving method may be precise and complete enough mathematically or theoretically, the algorithm can be implemented on a computer in different ways, some being more efficient than others. We gave, therefore, an advantageous computer implementation of the decision-state Algorithm 2 which combines the flexibility and adaptability of the basic algorithm with the characteristics of a particular digital computer and some good programming practices.

As with most large-scale, general-purpose, mathematical programming methods, the decision-state method offers certain options or opportunities for making the application of the method to a particular problem more advantageous. We discussed the availability and use of such options with regard to the method as well as in the formulation of the problem. We also gave some empirical evaluation of some of the different options that are intuitively appealing.

The computer implementation of Algorithm 2 developed in this dissertation was empirically tested and evaluated by using a number of resource allocation problems from the open literature and a number of problems that were artificially constructed to have certain desired structural characteristics simulating expected real conditions. The performance of Algorithm 2 was also compared with that of the MMDP algorithm of Morin and Marsten on identical problems. For all 9 problems run with both algorithms, Algorithm 2 performed consistently better than the MMDP algorithm, both in terms of high-speed memory requirement and in terms of solution time. In fact Algorithm 2 achieved an order of magnitude saving in memory requirement, and the MMDP algorithm took 45 to 82% more time than that taken by Algorithm 2. Although the storage saving is substantial, the time saving is not, since the latter might be attributed to the programming techniques used.

REFERENCES

1. Dharmadhikari, V.K., "Solving Discrete-Variable Multiple-Constraint Non-linear Programs: The Decision-State Method; doctoral dissertation, Department of Computer Science and Operations Research, Southern Methodist University, Dallas, Texas, (1975).

2. Bellman, R., Dynamic Programming, Princeton University Press, Princeton, N.J., (1957).

3. Larson, R.E., State Increment Dynamic Programming, American Elsevier Publishing Company, New York, (1968).

4. Wong, P.J., and Luenberger, D.G., "Reducing Memory Requirements of Dynamic Programming" Oper. Res., 16, 1115 - 1125, (1968)

5. Wong, P.J., "A New Decomposition Procedure for Dynamic Programming," Oper. Res., 18, 119-131, (1970).

6. Yormark, J.S., and Baker, N.R., "On a Two-Dimensional Resource Allocation Problem," presented at 39th ORSA Meeting, Dallas, Texas, (1971).

7. Weingartner, H.M., and Ness, D.N., "Methods for the Solution of the Multi-Dimensional 0-1 Knapsack Problem," Oper. Res., 15, 83-103, (1967).

8. Nemhauser, G.L., and Ullman, Z., "Discrete Dynamic Programming and Capital Allocation," Management Sci., 15, 494-505, (1969).

9. Haymond, R.E., "Discontinuities in the Optimal Return in Dynamic Programming," Jour. Math. Anal. Appl., 30, 159-169, (1970).

10. Gilmore, P.C., and Gomory, R.E., "The Theory and Computation of Knapsack Functions," Oper. Res., 14, 1045-1074, (1966).

11. Geoffrion, A.M., and Marsten, R.E., "Integer Programming Algorithms: A Framework and State-of-the-Art Survey," Management Sci., 18, (1972).

12. Morin, T.L., and Marsten, R.E., "An Algorithm for Non-linear Knapsack Problems," presented at ORSA/TIMS Meeting, Boston, Mass., (1974).

13. Morin, T.L., and Marsten, R.E., "An Efficient Dynamic Programming Algorithm for the General Non-linear Multidimensional Knapsack Problem," Discussion Paper (Rough Draft) I.E. and M.S. Dept., Northwestern University, Evanston, Ill., (1972).

14. Morin, T.L., and Esogbue, A.O., "Reduction of Dimensionality in Dynamic Programming of Higher Dimensions with the Imbedded State Space Approach," Discussion Paper, IE and MS Dept., Northwestern University, Evanston, Ill., (1973).

15. Knuth, D.E., The Art of Computer Programming, Vol. 1, 2, 3, Addison-Wesley Publishing Company, Reading, Mass., (1968).

16. Peterson, C.C., "Computational Experience with Variants of the Balas Algorithm Applied to the Selection of R and D Projects," Management Sci., 13, 736-750, (1967).

17. Grishman, Ralph, Assembly Language Programming for the Control Data 6000 Series and the Cyber 70 Series, Algorithmic Press, New York, (1974).

18. Hadley, G., Nonlinear and Dynamic Programming, Addison-Wesley Publishing Company, Reading, Mass., (1964).

19. Nemhauser, G.L., Introduction to Dynamic Programming, John Wiley, New York, (1966).

20. Greenberg, H.J., and Robbins, T.C., "The Theory and Computation of Everett's Lagrange Multipliers by Generalized Linear Programming," Technical Report CP-70008, Computer Science/Operations Research Center, Southern Methodist University, (1971).

21. Morin, T.L., and Marsten,R.E., "Branch and Bound Strategies for Dynamic Programming," Working Paper WP 750-754, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Mass., (1974).