A FORMALISM FOR DYNAMIC PROGRAMMING

Stephen M. Pollock

Robert L. Smith

Technical Report 85-8

Department of Industrial and Operations Engineering
The University of Michigan
Ann Arbor, Michigan 48109

A FORMALISM FOR DYNAMIC PROGRAMMING

by

Stephen M. Pollock
Department of Industrial
and Operations Engineering
The University of Michigan
Ann Arbor, Michigan 48109

Robert L. Smith
Department of Industrial
and Operations Engineering
The University of Michigan
Ann Arbor, Michigan 48109

Abstract

We introduce a formal structure for Dynamic Programming that associates a unique dynamic programming functional equation to every decision tree. Since in general, the computational complexity of the resulting functional equation is dependent on the decision tree chosen, the art of dynamic programming is shown to lie in the choice of decision tree to represent the problem.

A Formalism for Dynamic Programming

Dynamic Programming (DP) is a methodology developed by Richard Bellman in the early 1950's for efficiently solving problems involving sequential decision making. There is extensive literature dealing with its theoretical foundations as well as with its applications to a wide variety of problems (see, for example Denardo [1982]). Typical examples include equipment replacement, capacity expansion, resource allocation, and inventory planning. In general, these problems involve making a sequence of decisions (or a single decision that can be viewed as a sequence) that eventually optimizes some criterion, usually cost or profit.

What distinguishes dynamic programming from other optimization techniques, and in particular linear programming, is that although it is restricted to fewer decision variables, it allows far greater complexity in the problem's constraints and objective function. The formulation of a problem, however, in terms of a DP formalism, is usually presented as an arcane matter, and as a process that most people find, after the fact, to be an "aha" experience (Dreyfus & Law [1977]). In this paper we introduce a novel approach to formulation of problems in the dynamic programming framework; specifically we develop a formalism that we argue takes much of the mystery out of the so-called art of dynamic programming.

1. Deterministic Decision Trees

We begin with the fundamental concept of a deterministic decision tree (DDT). A DDT is a graph representation of the underlying sequential decision problem, where arcs correspond to possible decisions and nodes correspond to opportunities to select from these decisions. The graph is a tree rooted at the original decision node, and every path in a DDT corresponds to a (feasible) sequence of decisions. Moreover, associated with every decision arc, there is a

corresponding cost of making that decision, and the overall objective is to choose a feasible decision sequence that minimizes the sum† of the associated decision costs. This is clearly the equivalent to finding the minimum length path in the DDT whose arc lengths are decision costs.

For example, suppose we have a piece of equipment that is one year old, and we wish to decide on a replacement strategy over the next three years. At the beginning of each year we can keep the existing equipment or buy a new piece of equipment. The purchase price, salvage values and operating costs are given in Table 1. The objective is to minimize the total cost over three years.

	Age of Equipment in Years at Beginning of Year			
	0	1	2	3
Purchase Price	10	--	--	--
Operating Cost (per year)	0	.5	.75	1
Salvage Value at end of year	8	7	5	4.25

Table 1 - Cost Data For Equipment Replacement Example

The DDT for this problem is given in Figure 1, where the possible decisions at each node, representing yearly decision points, are to replace (R) by a new piece of equipment or keep (K) the current equipment. The minimum cost sequence of decisions is KKK leading to a total cost of -2. An obvious solution method

† We will not address here the more general problem class which includes criteria such as the product and the maximum of the minimum of associated decision costs.

for this small problem is simple inspection: complete enumeration of all feasible paths.

Insert Figure 1 Here

Explicit enumeration of all sequences clearly becomes infeasible for large problems. For a general problem of T periods, and D decisions per period, there are D^T distinct paths or feasible decision sequences. Each path requires T additions to evaluate its cost, for a total of TD^T additions. Finally, we need to perform D^T-1 comparisons. The total computation thus involves $(T+1)D^T-1$ elementary operations: an exponential amount of effort. Here is where dynamic programming enters. It prunes most of the tree's branches, typically reducing the computational effort to a polynomial function of problem size (in this case, the numbers D and T).

3. Aggregation of Nodes

Our notion of the key idea of DP is to aggregate nodes in the decision tree from which the remaining decision sequences are indistinguishable as we look forward in time. That is, nodes can be aggregated if they are the roots of (smaller) decision trees that are identical in structure and costs. This defines an equivalence relation which partitions the nodes of a DDT into aggregate classes. Once this aggregation is accomplished we can form a network, called the Dynamic Programming Network (DPN), whose nodes (called states) correspond to the classes of this partition. The construction of the rest of the DPN is straightforward: each arc in the DPN corresponds to a set of arcs in the DDT; the cost assigned to each DPN arc is the smallest of the costs associated with the set of corresponding arcs of the decision tree.

Figure 2 illustrates the aggregation process for the equipment replacement example. The shaded sets correspond to the aggregate classes. Note, for

example, that d and f in the DDT are in the same class D in the DPN, since they are both roots of an identical tree.

Insert Figure 2 Here

The corresponding Dynamic Programming Network is given in Figure 3.

Insert Figure 3 Here

What has been gained by this new representation? First, there are in general fewer nodes and fewer arcs in the DPN than in the original DDT, since redundant subtrees have been aggregated.

Second, every shortest route in the decision tree corresponds to a shortest route in the dynamic programming network, and vice versa. If the decision tree is finite (as in our example), it can be readily proven that all nodes of the DDT associated with a given aggregate node of the DPN must lie along distinct paths of the DDT. This means that there cannot be any directed cycles in the dynamic programming network: we have thus transformed the problem to finding the shortest route in a general acyclic finite network.

Finally, the nodes of the dynamic programming network typically have a natural interpretation, providing a reason for the use of the term "state." In the example, the DPN nodes can be labelled by a two component state variable: the age of the current equipment, and the number of years left in the study. Indeed, some reflection on the cost structure makes it clear that this is the only information on past decisions needed to determine the effect possible future decisions can have on total costs. Note, however, that this insight was not in principle a necessary prerequisite to being able to formulate the dynamic programming representation of Figure 3. This is what makes our approach different than the usual one. We note, however, that a computer implementation of the aggregation procedure would in general take an exponential amount of effort since every arc of the decision tree must be checked.

Construction of the Dynamic Programming Network from the original DDT completes the formulation phase of Dynamic Programming. Efficiently finding the shortest path in the dynamic programming network constitutes the solution phase of Dynamic Programming. (Moreover, for finite decision trees, this solution phase reduces to the problem of finding the shortest path in a general acyclic finite network.)

4. Formal Discussion

We now develop and summarize the preceding in a more formal manner.

Definition: A (Directed) Graph (N,A) is a set of nodes N together with a set of directed node pairs $(u,v) \in A \subseteq N \times N$ called arcs where $N \times N = \{(u,v) \mid u, v \in N\}$. A tree (N,A,r_0) is a directed graph with a distinguished node r_0 (called the root) from which there is a unique directed path to all other nodes.

Definition: A (directed) cycle is a directed path in a graph that begins and ends at the same node.

Definition: A decision tree (N,A,C,r_0) is a tree (N,A,r_0) rooted at r_0 together with a cost function C that associates a cost $C(u,v)$ with every arc $(u,v) \in A$.

Definition:

Two nodes u and v in a decision tree (N,A,C,r_0) are equivalent if the tree rooted from node u is identical in structure and costs with the tree rooted at v. More formally, u and v are equivalent if and only if there is an isomorphism between the trees rooted at u and v that preserves arc costs.

We can then partition the nodes of the decision tree into mutually exclusive and exhaustive aggregate classes B_1, B_2, B_3, ... of nodes, such that every node in a class is equivalent to all other nodes of that class and to no other nodes outside that class.

Definition:

The Dynamic Programming Network $(\underline{N}, \underline{A}, \underline{C})$ associated with the decision tree (N,A,C,r_0) consists of the nodes s_1, s_2, s_3, $\ldots \varepsilon \underline{N}$ (called DP states) corresponding to the classes B_1, B_2, B_3, ... If $u \varepsilon B_i$, we say u corresponds to s_i.

The arcs of the DPN are constructed as follows: An arc $(s_i, s_j) \varepsilon \underline{A}$ if and only if there is an arc $(k, \ell) \varepsilon A$ for corresponding $k \varepsilon B_i$ and $\ell \varepsilon B_j$. The arc cost (or length) is defined to be $\underline{C}(s_i, s_j) = \min_{\substack{k \varepsilon B_i \\ \ell \varepsilon B_j}} C(k,\ell)$.

The construction of a DPN ends the formulation phase of DP.

Using these definitions, it is possible to prove the following results (the proofs, being straightforward, are omitted).

Theorem: Every optimal (minimum cost) path of the decision tree corresponds to a shortest (minimum length) path in the dynamic programming network.

Moreover, we are now able to give the key result that allows for efficient calculation of a shortest path in the DP network, the well-known Principle of Optimality. Note that it is now sufficient without loss of generality to state this abstract principle only in terms of a shortest route problem.

Lemma 1 (Principle of Optimality): If a shortest route in a DP network from node s to node t passes through a node w, then this route must contain a shortest route from w to t.

This Principle of Optimality can now be used, in the usual way, to obtain a functional equation that may be solved for the length of a shortest path out of the root node. In particular, let f(s) be the length of a shortest route out of any node $s \varepsilon \underline{N}$ in the dynamic programming network. f, called the optimal value function, is the unique solution to the functional equation

$$f(s) = \min_{(s,t)\in \underline{A}} \{\underline{C}(s,t) + f(t)\} \tag{1}$$

where

$$f(s) = 0 \text{ if } \{(s,t)|(s,t)\in \underline{A}\} = \emptyset.$$

It is sometimes convenient (for gaining insight or neatness in coding) to group the DP states into other classes called stages.

Definition: A stage variable corresponds to the indices of a partition $\underline{N}_1$, $\underline{N}_2$, $\underline{N}_3$, ... of the nodes of the DP network, with the property that for $s \in \underline{N}_i$, $(s,t) \in \underline{A}$ only if $t \in \underline{N}_{i+1}$.

Stages are often a surrogate for the time dimension over which decisions can be thought to be sequenced, and thus can help in defining the states of a problem.

Application of the various methods to solve the functional equation (1) represent the solution phase of dynamic programming. We restrict consideration here to solution methods for finite decision trees.

Lemma 2: Suppose the decision tree (N,A,C,r_0) has a finite number of nodes and arcs. If both u and v correspond to s, then u and v cannot lie along the same directed path out of r_0.

Corollary: The DPN corresponding to a finite decision tree cannot have directed cycles.

It is the Corollary that allows a simple recursive procedure to find the shortest route in the DPN in the finite network case. Label the n nodes of the DP network $(\underline{N},\underline{A},\underline{C})$ with node numbers i = 0, 1, 2, ..., n with the property that: arc $(i, j) \in \underline{A}$ only if $i < j$ for all i and j; 0 is the root node, and n is the (unique) terminal node. This is always possible since $(\underline{N},\underline{A},\underline{C})$ is an acyclic finite network.

We can now describe a technique, known as recursive fixing (Denardo [1982]), for solution of (1).

Recursive Fixing

1. $i \leftarrow n$ and $f(n) \leftarrow 0$.
2. If $i = 0$, stop; otherwise set $i \leftarrow i-1$ and continue.
3. $f(i) \leftarrow \min_{j>i} \{\underline{C}(i,j) + f(j)\}$. Go to step 2.

Recursive fixing requires at most n-1 additions and comparisons for each of n nodes. This produces a computational complexity that is order n^2, and hence polynomial in the number of states.

5. Non-Unique Formulations

We have shown how the functional equation for a given problem can be uniquely derived from the DDT formulation of the problem via the DPN. Hence if we want to generate a different functional equation (i.e. a different DP formulation), we must first represent the problem with a different decision tree. The resulting number of states (and therefore the worst case computational complexity of recursive fixing for solving the functional equation) can vary significantly depending upon the choice of the original DDT. Thus the "art" of dynamic programming can be completely encapsulated by the choice of the decision tree. This can be illustrated by considering several different formulations of the classic knapsack problem.

The knapsack problem is to pack a maximal value knapsack of weight not exceeding W from N item types. Item type i ($i = 1,2,\ldots,N$) has weight w_i and value v_i. There are an infinite number of each item type available. If x_i is

the number of items of type i packed in the knapsack, then the problem can be expressed as the following mathematical program:

$$\max\ v_1x_1 + v_2x_2 + \dots + v_Nx_N$$

subject to

$$w_1x_1 + w_2x_2 + \dots + w_Nx_N \le W$$

$$x_i \ge 0 \text{ integer for } i = 1, 2, \dots, N$$

Perhaps the most natural decision tree is that generated by the sequence of decisions $x_1, x_2, \dots, x_N$, i.e. how many type 1 items, how many type 2 items, etc.... to place in the knapsack. For example, consider the problem with the data as given in Table 2.

Item Type	i	1	2	3
Weight	w_i	2	3	4
Value	v_i	2	5	8

W = weight available = 5

Table 2 - Data for Knapsack Problem

The DDT is shown in Figure 4, followed by the aggregation of nodes that leads to the DPN (shown in Figure 5).

Insert Figure 4 Here

Insert Figure 5 Here

As formulated in Figure 5, an appropriate state description for the nodes of the DPN might be the pair (number of item types for which allocations have been made, useable weight remaining for unallocated item types). Note that, strictly speaking, nodes f, g, h and i of the DDT are equivalent, given the particular data elements of the problem. Indeed, if the weight of item 3 were 3 rather than 4, then nodes f and g would not be equivalent, since the decision $x_3=1$ would then be possible from node g, but not from node f. The range of possible state values is thus complexly dependent on the item weights, making it difficult to write down the functional equation in advance of its solution.

It is possible to make all weights feasible from 0, 1, 2,..., W by re-formulating the problem to include slack items with weight 1 and value 0. Specifically, we allow at every value of i, the option to include any feasible number of slack items with items of type i. Letting x_o^i be the number of slack items included with the x_i items of type i, we obtain the DDT shown partly in Figure 6. The resulting DPN is given in Figure 7.

Insert Figure 6 Here

Insert Figure 7 Here

The state variable that results from the DPN (Figure 7) now becomes s = (n,w): at that node there has been allocated a weight of <u>at most</u> w to the first n items, where n is seen to be a stage variable. The DPN thus gives us the following optimal value function for the general case (where we use the conventional notation of subscripting with the stage variable)

$f_n(w)$ = maximum value knapsack of weight at most w
using item types i = 1, 2,...n.

The functional equation becomes:

$$f_n(w) = \begin{cases} \max\limits_{\substack{x \le \left\lfloor \frac{w}{w_n} \right\rfloor, \\ x \text{ integer}}} v_n x + f_{n-1}(w - w_n x) & \text{for } n = 1,2,\ldots,N \\ & \quad w = 0,1,\ldots,W \\ 0 & \text{for } n = 0,\ w = 0,1,\ldots,W. \end{cases}$$

Recursive Fixing, therefore, solves a series of knapsack problems, increasing the number of items considered by one each time until all N items are considered. The computational complexity for solution of this formulation of the knapsack problem is seen to be of order NW^2, since there are N stages, each with order W^2 arcs connecting them. Although we have not done so, one can establish rigorously that $f_n(w)$ satisfies the functional equation above by mathematically demonstrating that all decision trees out of nodes with the same values of n and w must be identical.

There is, however, still a third way to construct the DDT: each decision can be an item type number for the next item to be added to the knapsack. Figure 8 gives this decision tree where t_i is the ith item type chosen to include in the knapsack for i=0, 1, 2, 3 where again item type 0 is a slack item of unit weight and zero value.

Insert Figure 8 Here

The aggregation step then identifies a natural state variable: w, the weight of a knapsack sufficient to hold items added thus far. These are shown in Table 3.

Value of State Variable w	Corresponding Nodes of DDT
0	0
1	1
2	2, 5
3	3, 6, 9, 15
4	4, 7, 10, 12, 16, 18, 21, 25
5	8, 11, 13, 14, 17, 19, 20, 22, 23, 24, 26, 27, 28, 29, 30

Table 3 - Aggregation for DDT of Figure 8

The resulting DPN is shown in Figure 9; one that is quite different from that in Figure 5.

Insert Figure 9 Here

The associated functional equation becomes (with $f(w)$ interpreted as the maximal value knapsack of weight at most w)

$$f(w) = \begin{cases} \max \{f(w-1), \max\limits_{\substack{i=1,2,\ldots,N \\ \text{with } w_i \leq w}} (v_i + f(w-w_i))\} & \text{for } w = 1,2,\ldots,W \\ 0 & \text{for } w = 0. \end{cases}$$

The computational complexity of recursive fixing applied to this last functional equation is only of order NW. This is strictly better than the previous formulation for all values of W and N.

There are more efficient procedures than recursive fixing to solve these functional equations, for example, reaching with acceleration devices (see

Denardo [1982]). However, in general, the third formulation yields uniformly better computational complexity. Thus we have seen how the art in dynamic programming can be subsumed in the choice of best decision tree representation of the problem. The corresponding functional equation then follows uniquely from that decision tree choice.

6. Conclusion

We have demonstrated, by a simple formalism with examples, a method for formulating dynamic programming representations of sequential decision problems. The procedure requires an initial deterministic decision tree, from which a node aggregation operation produces a network. This network, in turn, allows a computationally attractive shortest (or longest) path solution via a DP functional equation. We have made no claims to offering new insights into appropriate solution methods for these functional equations, nor advice on how to abstract and conceive of the original DDT. In fact we argue the latter task represents the art of Dynamic Programming. We do show, however, that an aggregation process not only gives a direct way to write appropriate recursive equations, but also automatically identifies that elusive feature of Dynamic Programming: the state variable.

Acknowledgement

The work of Robert L. Smith was supported by the National Science Foundation under Grant No. ECS-8409682.

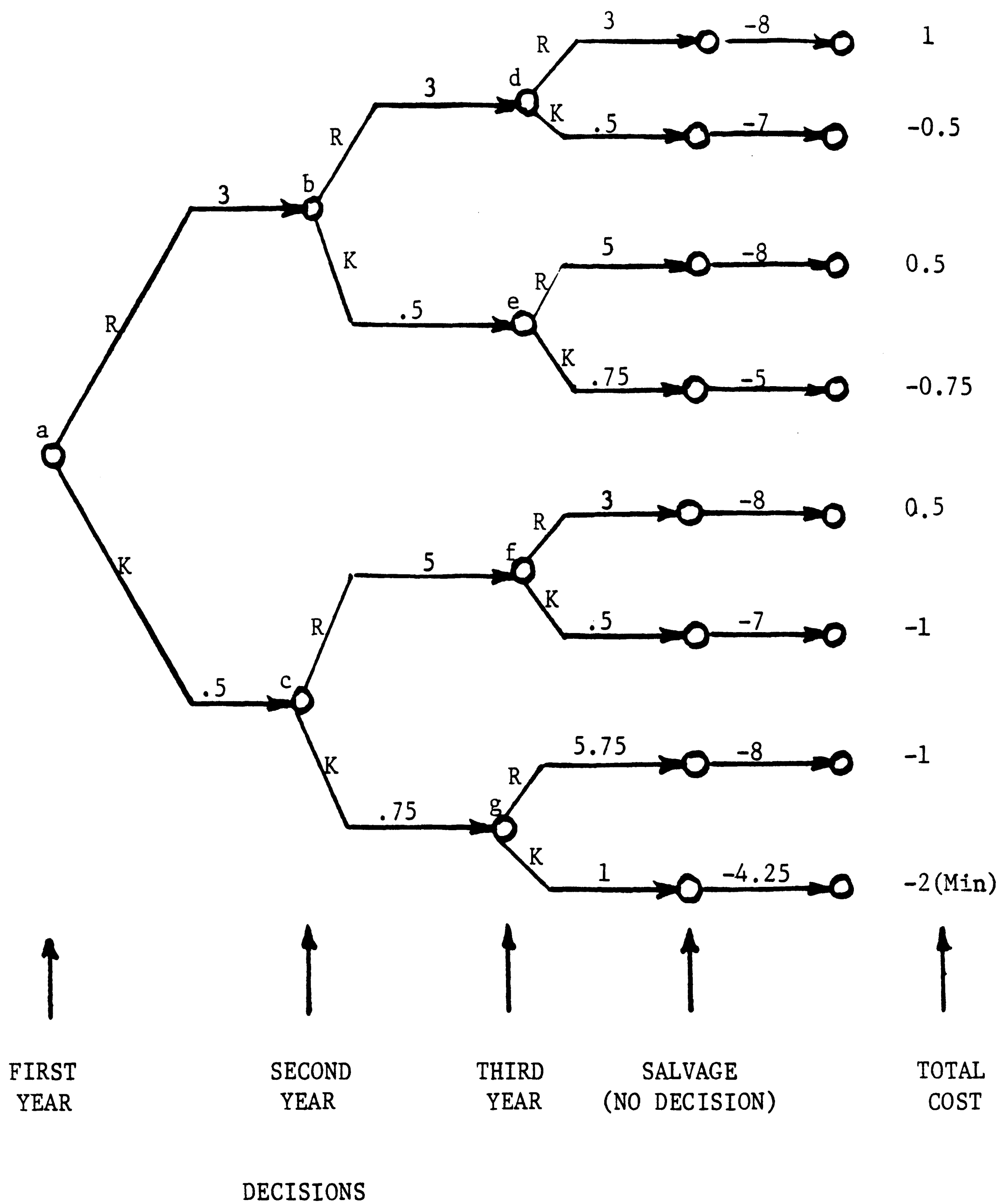

FIGURE 1

DECISION TREE FOR EQUIPMENT REPLACEMENT EXAMPLE

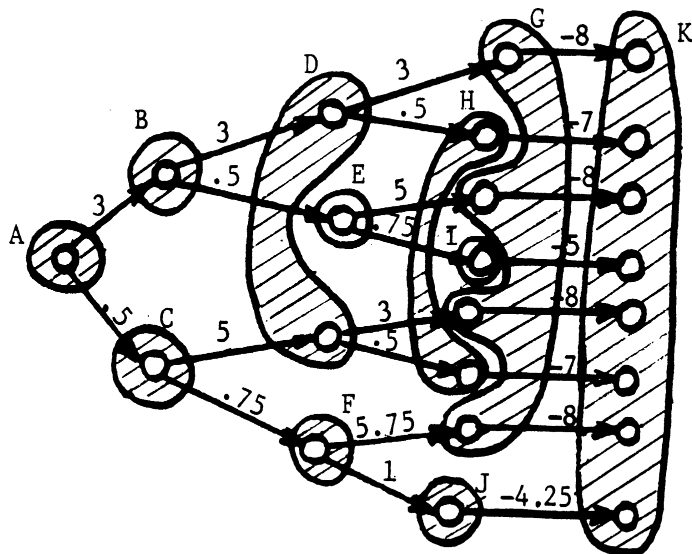

FIGURE 2

AGGREGATE CLASSES FOR EQUIPMENT REPLACEMENT EXAMPLE

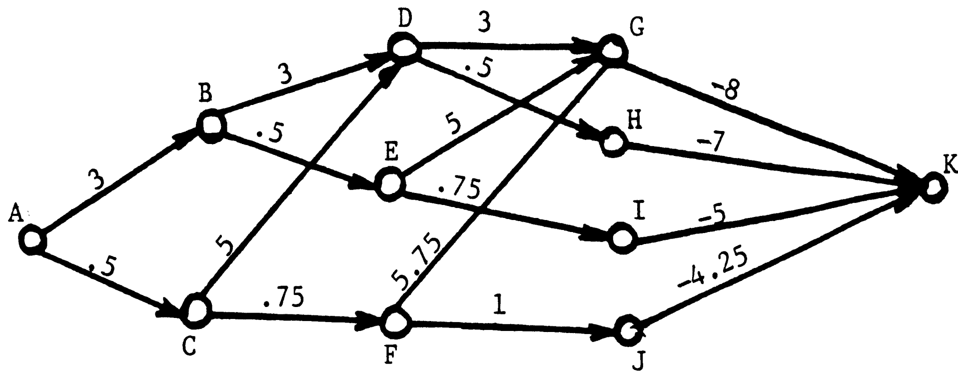

FIGURE 3

THE DYNAMIC PROGRAMMING NETWORK
FOR THE EQUIPMENT REPLACEMENT EXAMPLE

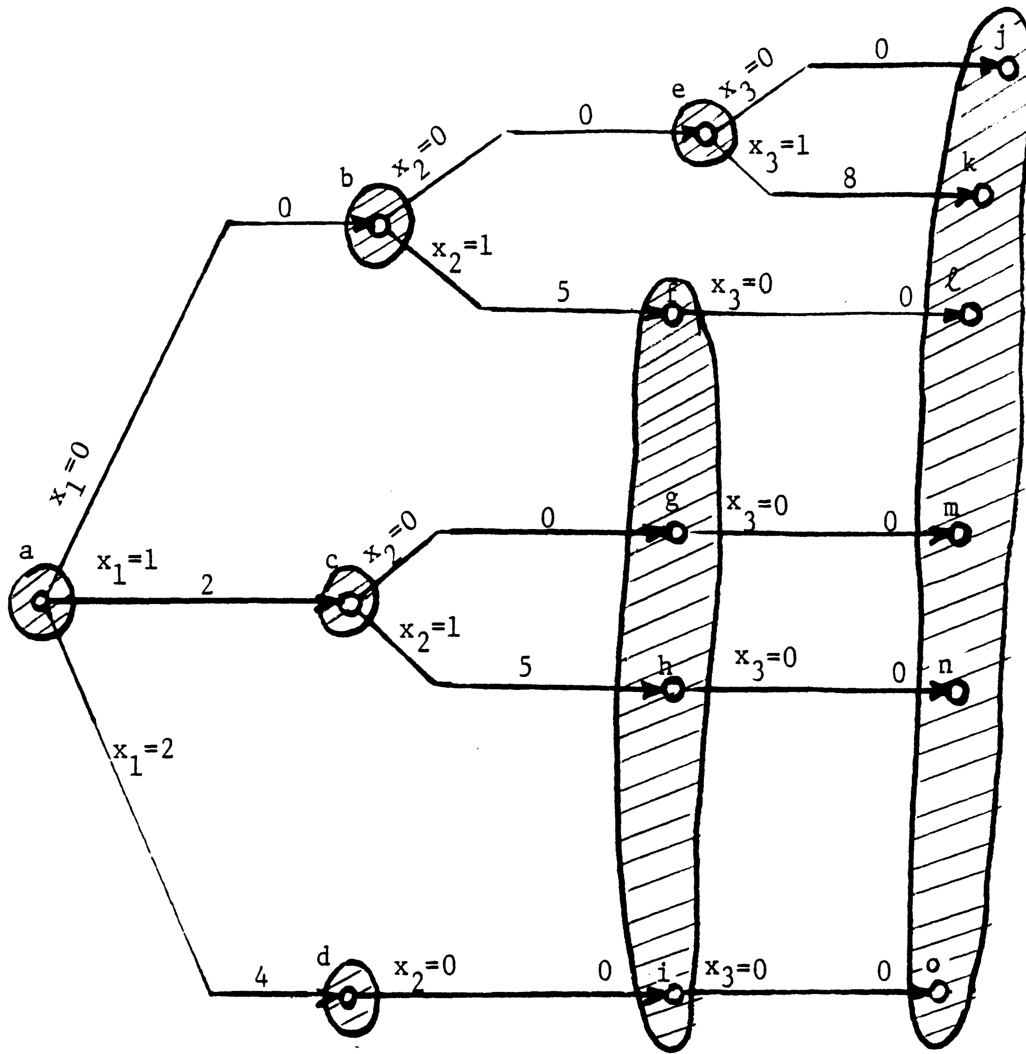

FIGURE 4

DDT 1 FOR KNAPSACK PROBLEM

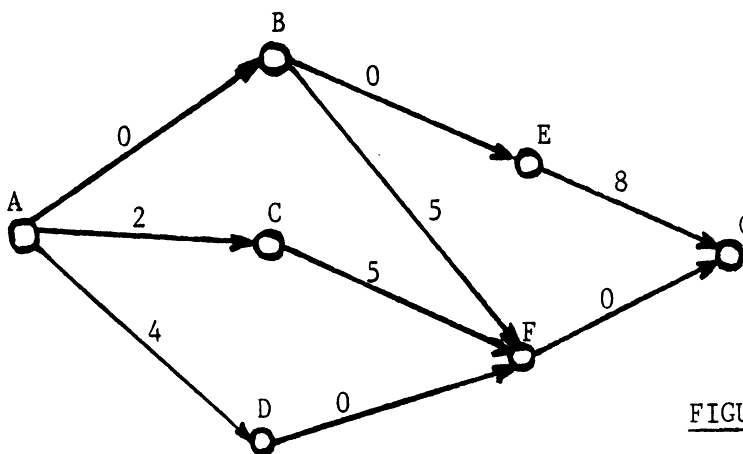

FIGURE 5

DPN 1 FOR KNAPSACK PROBLEM

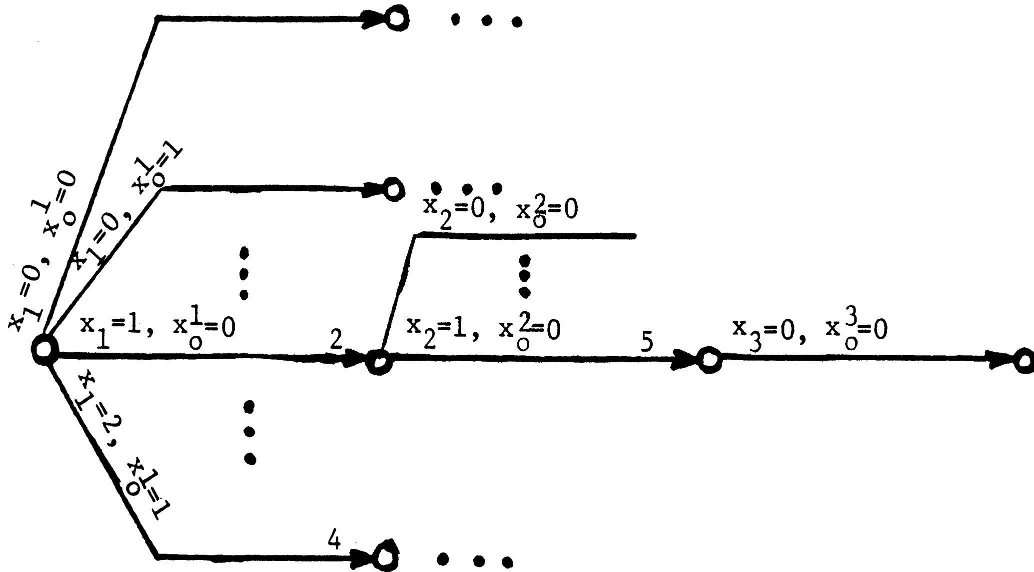

FIGURE 6

DDT 2 FOR THE KNAPSACK PROBLEM

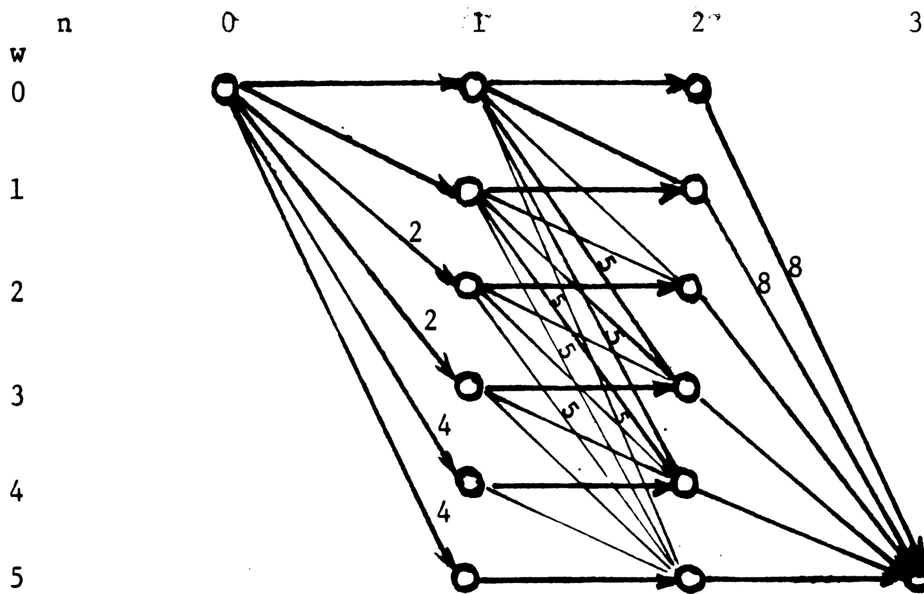

FIGURE 7

DPN 2 FOR KNAPSACK PROBLEM

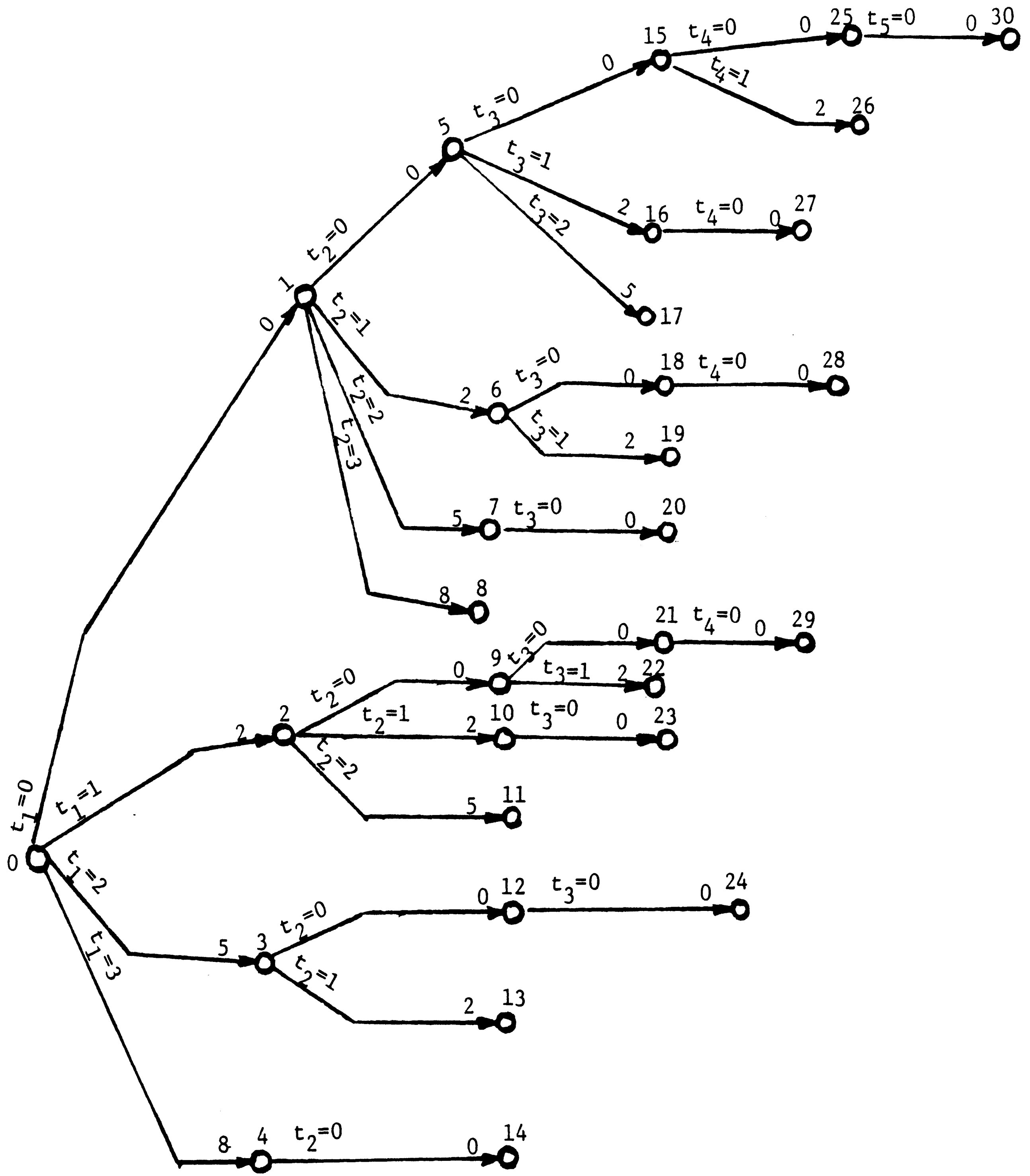

FIGURE 8

DDT 3 FOR KNAPSACK PROBLEM

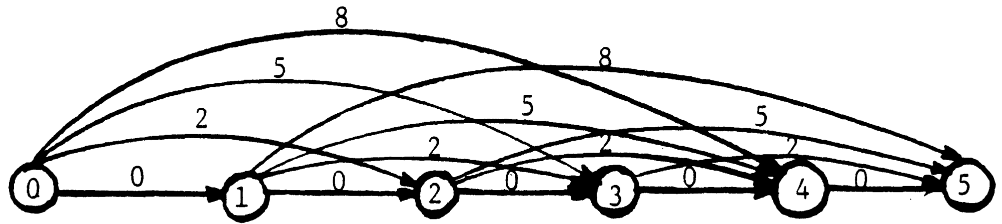

FIGURE 9

DPN 3 FOR KNAPSACK PROBLEM

References

1. Denardo, Eric V. (1982), Dynamic Programming: Models and Applications, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

2. Dreyfus, S. E. and A. Law (1978), The Art and Theory of Dynamic Programming, Academic Press, New York.