A Formalism for Dynamic Programming

Yu-Li Chou* Stephen M. Pollock† H. Edwin Romeijn‡
Robert L. Smith§

October 1, 2001

Abstract

We introduce a formal structure for dynamic programming that associates a unique dynamic programming functional equation to every deterministic, separable decision tree representation of the underlying problem. Since, in general, the computational complexity of the resulting functional equation depends on the decision tree chosen, the art of dynamic programming is shown to lie in the choice of decision tree to model the problem. The development is illustrated with alternative formulations of the classic knapsack problem.

1 Introduction

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 [3], Dreyfus and Law[5], Bertsekas[1]). Typical examples include the problems of 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 particularly linear programming, is that although it cannot handle large scale problems, it allows

*Saltare, San Mateo, California; e-mail: yulichou@hotmail.com.

†Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, Michigan 48109-2117; e-mail: pollock@umich.edu. The work of this author was supported in part by NSF Grants SBR 9712997 and DMI 9713654

‡Department of Industrial and Systems Engineering, University of Florida, 303 Weil Hall, P.O. Box 116595, Gainesville, Florida 32611-6595; e-mail: romeijn@ise.ufl.edu.

§Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, Michigan 48109-2117; e-mail: rlsmith@umich.edu. The work of this author was supported in part by NSF Grants DMI-9900267 and DMI-9713723

for far greater complexity in the problem's constraints and objective function. The formulation of a problem, however, is usually presented as an arcane matter, and as a process that most people find, after the fact, to be an "aha" experience (Dreyfus and Law [5]). In this paper we introduce what we believe to be a novel approach to formulating problems as dynamic programs. In particular, we develop a formalism that we argue takes much of the mystery out of the so-called art of dynamic programming. We believe this rigorous formulation, which extracts a DP functional equation from more fundamental elements, offers both pedagogic and research opportunities. Alternative DP formulations of the same underlying problem become alternative manifestations of that problem arising from alternative decision tree representations. Subtly different decision tree formulations can give rise to surprisingly different DP formulations as we illustrate in section 5 with the classic knapsack problem. This serves the pedagogic function of uncovering the machinery underlying alternative DP formulations of the same problem. It also provides a rigorous framework to potentially pose (and answer) questions about finding an "optimal" DP formulation of a problem with respect to minimizing computational effort to solve the corresponding DP functional equation (see e.g., Chou [2] for an attempt at this.)

In the literature, White [11] and others have long noted that solving a (deterministic) DP functional equation is equivalent to finding a greatest return path in a directed network. What we show is that this network, which we refer to as the dynamic programming network, is rigorously derivable from a more elementary object, namely the deterministic decision tree. A key to this approach is to *construct* the DP states (via a tree-node aggregation process) in such a way that it represents the *minimal* information about past decisions necessary to define the problem confronting us along choice points in the decision process. If one were to adopt the more general notion of state that relaxes this minimality condition, then the nodes of the original decision tree would also be "states". However, this is inconsistent with conventional notions of DP states; it also fails to result in an efficient formulation. Elmagraghby [6] comes very close to our point of view in his notion of a "minimal" state space, but fails to pursue a rigorous construction of the concept.

We also believe our framework, in addition to being rigorous, is more transparent and elementary than other approaches. For example, Karp and Held [7] model the original problem as a finite automaton which incorporates potential states in its definition. They then posit a certain monotonicity property that, when true, lends the automaton states the property of being DP states, i.e. with the property that the principle of optimality holds. However, the states are not defined in more elementary terms, nor are they necessarily minimal in information content, i.e. they may not correspond to states as we define them or as in conventional DP. (Mitten [9] and Denardo and Mitten [4] introduce a similar monotonicity property that lends "states" the principle of optimality property. However, these states are merely posited and in general may fail to be minimal.) Nemhauser [10] takes an algebraic approach to state identification via a procedure known as constructive derivation. He begins with a mathematical programming formulation of the problem and proceeds to algebraically transform relationships that emulate the DP functional equation. However the procedure, although rigorous, presumes *a priori* knowledge of where and how the problem separates or (equivalently) what the underlying algebraic states in fact are.

How decision trees are aggregated to construct a dynamic programming network is informally described in sections 2 and 3. Section 4 provides a formal presentation of this process and its properties. In section 5, we illustrate the method via the classic knapsack problem. We end the paper in section 6 with some concluding remarks.

2 Deterministic Decision Trees

We begin with the elementary concept of a *deterministic decision tree* (DDT). A DDT is a digraph representation of an underlying sequential decision problem whose arcs and associated profits correspond to possible decisions and their profits and nodes correspond to opportunities to select from decisions emanating from that node. The digraph is a directed tree rooted at the original decision node, and every path out of this root corresponds to a sequence of decisions. A complete path, ending in a node which has no successors, corresponds to a (feasible) solution to the decision problem. In this paper we restrict ourselves to the case of finite decision trees, i.e. trees with a finite number of nodes (thus all feasible paths consist of a finite number of arcs, or decisions). Since our objective is to choose a feasible decision sequence that maximizes the sum of the associated decision profits, the optimization problem is equivalent to finding a longest path in the DDT whose arc lengths are the corresponding decision profits. Clearly, an analogous statement holds when there is a cost associated with each decision, where the cost may be seen as a negative profit, and the objective becomes that of minimizing total costs or equivalently finding a shortest path.

For example, consider the classic equipment replacement problem. Suppose we have a piece of equipment that is one year old, and we wish to decide on a least cost investment 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 cost over the three years.

Age of equipment in years at beginning of year	0	1	2	3	4
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 Problem.*

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) the current equipment 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 for this small problem is simple inspection via a complete enumeration of all paths.

Explicit enumeration of all sequences clearly becomes infeasible for large problems. For a general equipment replacement problem with 2 possible decisions in each of T periods, there are 2^T distinct paths or feasible decision sequences. Each path requires T additions

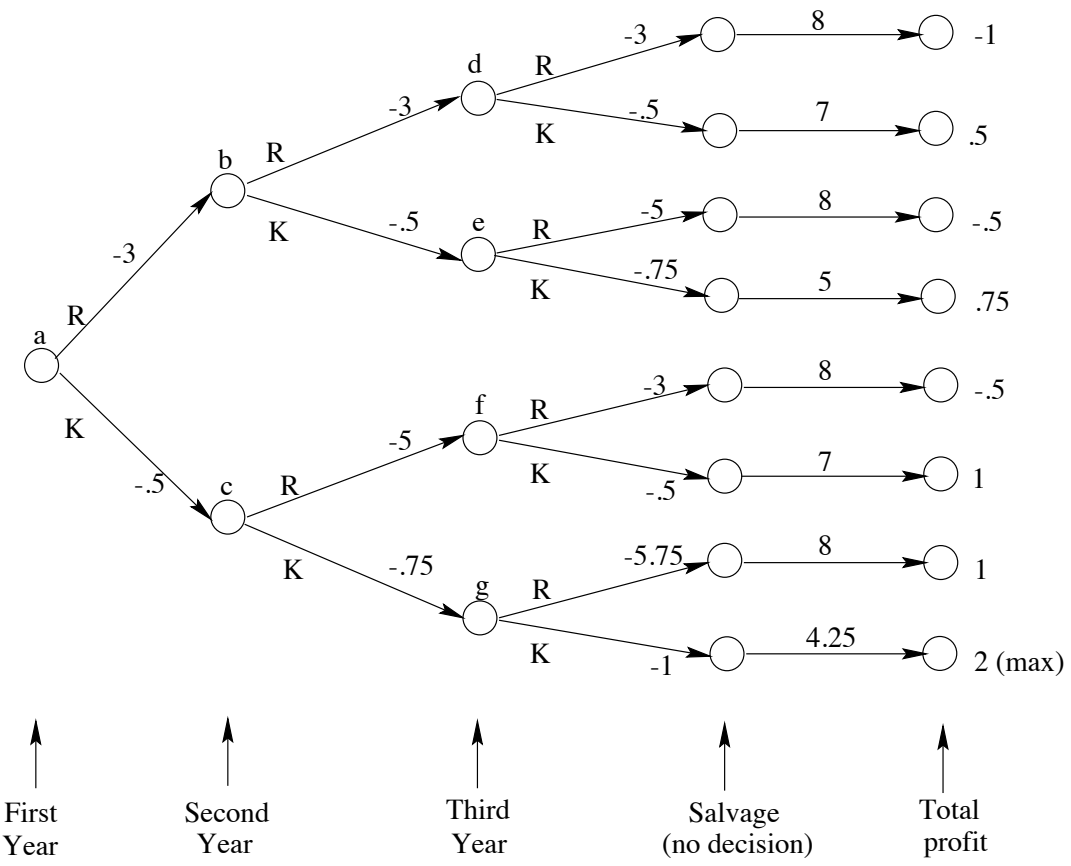

Figure 1: *The DDT for the equipment replacement problem.*

to evaluate its cost, for a total of $T2^T$ additions. Finally, $2^T - 1$ comparisons must be performed. The total computation thus involves $(T+1)2^T - 1$ elementary operations: an exponential amount of effort. Dynamic programming, by contrast, aims at pruning most of the tree's branches, typically reducing the computational effort to a polynomial function of the problem size (in this case, the number of periods T).

3 Aggregation of Nodes

In this section we informally describe our notion of the key idea of Dynamic Programming: *aggregation* of nodes in the decision tree so that the remaining decision sequences are indistinguishable as we look forward in time. More formally, we seek to identify nodes that are the roots of decision subtrees that are identical in structure and profits. This defines an equivalence relation which partitions the nodes of a DDT into equivalence classes we call *aggregate classes*. Once this aggregation process is accomplished, we form a network, called the *Dynamic Programming Network* (DPN), whose nodes (called *states*) correspond to the classes of this partition. The construction of the remainder of the DPN is straightforward:

each arc in the DPN corresponds to the set of corresponding arcs in the DDT; the profit assigned to each DPN arc is the maximum profit associated with the corresponding arcs of the decision tree.

For example, Figure 2 shows the aggregation process for the DDT of Figure 1.

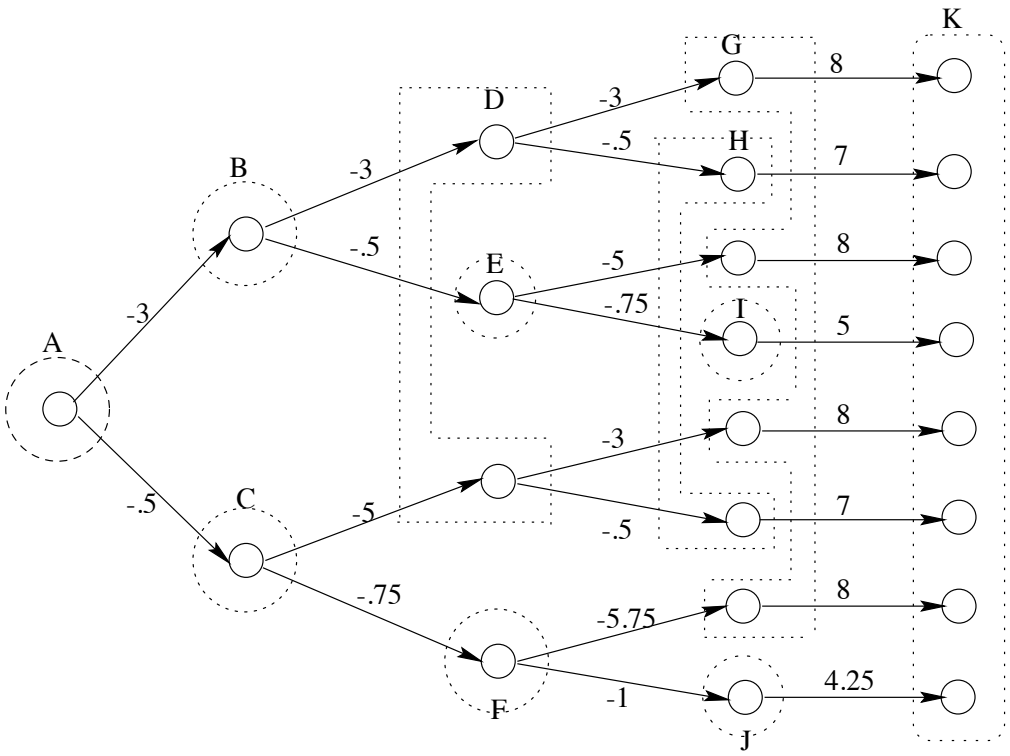

Figure 2: *The aggregation process for the equipment replacement problem of Figure 1.*

The dotted lines denote the aggregate classes. Note that, for example, nodes d and f in the DDT are in the same class D in the DPN, since they are roots of identical trees. The Dynamic Programming Network corresponding to Figure 2 is given in Figure 3.

One may ask the question: what has been gained by representing a decision tree as a dynamic programming network? First, there are, in general, fewer nodes and fewer arcs in the DPN than in the original DDT, since equivalent subtrees have been aggregated. The DPN is in this sense a more economical representation of the problem then the original DDT. However all paths of the DDT remain in the DPN and vice versa. Hence, every longest path in the decision tree corresponds to a longest path in the dynamic programming network, and vice versa. Since the decision tree is finite, 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 one of finding the longest 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 labeled by using the age of the current equipment and the number of

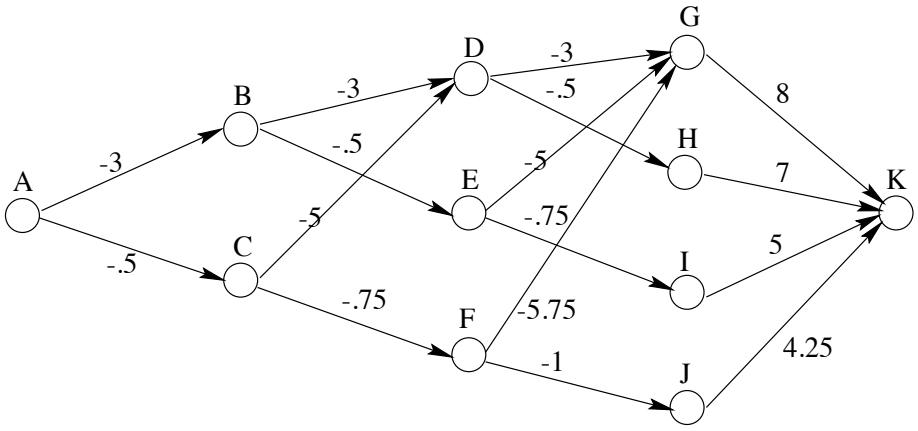

Figure 3: *The DPN for the Equipment Replacement problem of Figure 1.*

years left in the study. Indeed, some reflection on the profit structure of this simple example makes it clear that this is the only information about past decisions needed to determine the effect that possible future decisions can have on total profits. 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 "artless" approach different than the usual one.

The construction of the Dynamic Programming Network from the original DDT completes the *formulation phase* of Dynamic Programming. Efficiently finding the longest 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 a longest path in a general acyclic finite network, a topic with a long and fruitful literature (see e.g., Denardo [3]).

4 Formal Discussion

In this section we formalize the preceding discussion.

4.1 Definitions

Definition 4.1 *A* directed graph (or digraph) $G = (N, A)$ *is a set of* nodes N*, together with a set of directed node pairs* $(u, v) \in A \subseteq N \times N$*, called* arcs.

Definition 4.2 *A* (directed) tree $T = (N, A, r)$ *is an acyclic directed graph* (N, A) *with a distinguished node* $r \in N$ *called the* root*, from which there is a unique directed path to all other nodes. Furthermore, let* $N_T \subseteq N$ *denote the set of* terminal nodes *of* T*, i.e. the set of nodes from which no arcs emanate.*

Definition 4.3 *A* deterministic decision tree (or DDT) $\mathcal{T} = (T, P, \circ)$ *is a tree* $T = (N, A, r)$*, together with a profit function* $P : A \to \mathbb{R}$*, and a symmetric composition operator* $\circ$ *on* R^2*,*

for combining arc profits. The corresponding path profit function is denoted by $\varphi : N \times N \to \mathbb{R} \cup \{-\infty\}$. *If there exists a directed path from* $i \in N$ *to* $j \in N$, *then it is unique, say* $(i, \ell_1, \ell_2, \ldots, \ell_n, j)$, *and*

$$\varphi(i,j) \equiv P(i,\ell_1) \circ P(\ell_1,\ell_2) \circ \cdots \circ P(\ell_n,j) \qquad \forall \ell_i \in N$$

Otherwise, $\varphi(i,j) = -\infty$. *Let* $\mathbf{0}$ *be the zero element of* $\circ$, *i.e.* $a \circ \mathbf{0} = a$ *for all* $a \in Range(\varphi)$, *and let* $\varphi(i,i) = \mathbf{0}$ *for all* $i \in N$.

Each of the nodes N in a DDT represents an opportunity to make a decision. Furthermore, the arcs A represent possible decisions, and the root r represents the start of the decision process. The composition operator "$\circ$" defining the path profit function is often "addition" ($\circ = +$, with $\mathbf{0} = 0$), but could also be, for example, multiplication ($\circ = \cdot$, with $\mathbf{0} = 1$), or taking the minimum ($\circ = \wedge$, $\mathbf{0} = \infty$) or maximum ($\circ = \vee$, $\mathbf{0} = -\infty$).

Note that every directed path connecting the root node to a terminal node, called a *complete path*, consists of a sequence of decisions which corresponds to a feasible solution to the original sequential decision problem. Thus, the decision problem is reduced to the problem of finding a maximum profit path in $\mathcal{T}$.

4.2 Forming the Dynamic Programming Network

The following definition establishes an equivalence relation among nodes in N which provides the basis for forming the corresponding dynamic programming network.

Definition 4.4 *The decision subtree rooted at* $u \in N$ *is said to be* **equivalent** *to the decision subtree rooted at* $v \in N$ *if these trees are identical in graphical structure and profits. 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 profits.*

If E denotes the equivalence relation described in Definition 4.4, then uEv represents the condition {any directed forward path from u is in one-to-one correspondence with a forward path from v, both corresponding to the same decision sequence }. The nodes of the decision tree can then be partitioned into mutually exclusive and exhaustive (aggregate) classes such that every node in a class is equivalent to all other nodes in that class and to no other nodes. That is, for all $u \in N$, we let

$$[u] \equiv \{v \in N \mid uEv\}.$$

The *Dynamic Programming Network (or DPN)* $\mathcal{G} = (\mathcal{N}, \mathcal{A}, \mathcal{P}, \rho)$ corresponding to a DDT $\mathcal{T}$ consists of a set of nodes $\mathcal{N}$, containing the equivalence classes of E, and a set of arcs $\mathcal{A}$, where an arc is present in $\mathcal{A}$ if and only if there is at least one arc in A connecting nodes from the two corresponding equivalence classes. The profit of a DPN arc, given by the function $\mathcal{P}$, is the maximum profit P of all arcs connecting nodes from the two corresponding equivalence classes. Finally, $\rho = [r] = \{r\}$ is the equivalence class corresponding to the root

r of the DDT. More formally,

$$\begin{aligned}
\mathcal{N} &= \{[u] \mid u \in N\} \\
\mathcal{A} &= \{([u],[v]) \mid (u,v) \in A\} \\
\mathcal{P}([u],t) &= \max_{\{v \in N \mid [v]=t,\,(u,v)\in A\}} P(u,v) \qquad \text{for all } u \in N,\ t \in \mathcal{N}, \text{ such that} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{v \in N \mid [v]=t,\,(u,v)\in A\} \neq \emptyset \\
\rho &= [r].
\end{aligned}$$

Each complete path in a DDT thus corresponds to a path in the corresponding DPN which consists of the same sequence of decisions. The construction of a DPN ends the *formulation phase* of DP.

From the above discussion, the following lemma can easily be proven.

Lemma 4.5 *Every maximum profit path in the decision tree (DDT) corresponds to a maximum profit path in the dynamic programming network (DPN).*

Proof: Omitted. □

4.3 Recursive Functional Equations

In this section we present a mild regularity condition on the profit structure of the decision tree, under which a recursive equation, called the DP functional equation, can be uniquely derived from the DPN. This functional equation provides a computationally attractive way of finding the optimal solution to the original sequential decision problem. The regularity condition (which we will assume is satisfied in the remainder of the paper) is:

Assumption 4.6 *For all $a \in Range(\varphi)$, $a \circ b$ is nondecreasing in $b \in Range(\varphi)$.*

This assumption is obviously satisfied if $\circ = +$. Other examples satisfying the assumption are the cases where $\circ = \wedge$ or $\circ = \vee$. If $\circ = \cdot$, we need to impose the additional constraint that Range(φ) is a subset of the nonnegative reals. This can be accomplished by assuming that all the arc profits are nonnegative.

Now let $f(s)$ (for all $s \in \mathcal{N}$) be the maximum profit obtainable for reaching node s from the initial state ρ in the dynamic programming network, i.e. from Lemma 4.5,

$$f(s) = \max_{\{u \mid [u]=s\}} \varphi(r,u).$$

The following theorem shows that the values of $f(s)$ can be computed recursively.

Theorem 4.7 *Given a decision tree $\mathcal{T} = (T,P,r)$, and corresponding dynamic programming network $\mathcal{G} = (\mathcal{N},\mathcal{A},\mathcal{P},\rho)$. Then, for all $s \in \mathcal{N}$, the following relationship holds:*

$$f(s) = \begin{cases} \mathbf{0} & \text{if } s = \rho \\ \max_{\{t \mid (t,s)\in\mathcal{A}\}} f(t) \circ \mathcal{P}(t,s) & \text{for all } s \in \mathcal{N}\backslash\{\rho\}. \end{cases} \tag{1}$$

Proof: Clearly, we have

$$\begin{aligned} f(\rho) &= \max_{\{u|[u]=\rho\}} \varphi(r,u) \\ &= \varphi(r,r) \\ &= \mathbf{0}. \end{aligned}$$

If $s \neq \rho$, then we have

$$\begin{aligned} f(s) &= \max_{\{u|[u]=s\}} \varphi(r,u) \\ &= \max_{\{u|[u]=s\}} \max_{v:(v,u)\in A} \varphi(r,v) \circ P(v,u) \\ &= \max_{\{v|\exists u:(v,u)\in A,[u]=s\}} \max_{\{u:[u]=s,\,(v,u)\in A\}} \varphi(r,v) \circ P(v,u) \\ &\leq \max_{\{v|\exists u:(v,u)\in A,[u]=s\}} \varphi(r,v) \circ \left(\max_{\{u|[u]=s,\,(v,u)\in A\}} P(v,u) \right) \qquad (2) \\ &= \max_{\{v|([v],s)\in\mathcal{A}\}} \varphi(r,v) \circ \mathcal{P}([v],s) \qquad (3) \\ &= \max_{\{t|(t,s)\in\mathcal{A}\}} \max_{\{v|[v]=t\}} \varphi(r,v) \circ \mathcal{P}(t,s) \qquad (4) \\ &\leq \max_{\{t|(t,s)\in\mathcal{A}\}} \left(\max_{\{v|[v]=t\}} \varphi(r,v) \right) \circ \mathcal{P}(t,s) \qquad (5) \\ &= \max_{\{t|(t,s)\in\mathcal{A}\}} f(t) \circ \mathcal{P}(t,s). \end{aligned}$$

Note that (3) implies (4) by the definition of class equivalence, symmetry is used in (5), and assumption 4.6 is used in equations (2) and (5). The conclusion now follows since the left hand side of (5) is attained by a path and hence $f(s) \geq \max_{\{t|(t,s)\in\mathcal{A}\}} f(t) \circ \mathcal{P}(t,s)$. □

Indeed, Theorem 4.7 is an extension of the well-known "Principle of Optimality" that accounts for the existence of equivalence classes. The optimal solution in the DPN can then be obtained by computing $f(t)$, where $t \in \{[v] \mid v \in N_T\}$.

The formulation phase of DP can now be summarized as follows:

a) generate a decision tree by some decision rule;

b) form the corresponding dynamic programming network;

c) if the profit structure of the resulting decision tree satisfies Assumption 4.6, formulate a recursive functional equation according to Theorem 4.7.

5 The knapsack problem

In this section we rigorously apply the DP formalism introduced in section 4 to the classic knapsack problem: select a number of items with maximal total value, from K item types, under the constraint that the total weight of the items does not exceed W. Associated with

each item type i ($i = 1, \ldots, K$) there is a weight w_i and value v_i. We assume that there is an unlimited supply of each item type available. Moreover, for convenience we assume that the knapsack capacity W, as well as the item values and weights, are positive integers. Without loss of optimality, it can be assumed that none of the item types have the same weight (see Martello and Toth [8]). As a numerical example, consider the knapsack problem with the data given in Table 2.

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

Table 2: *Data for the knapsack problem;* $W = 5$.

One way to construct a decision tree is to have the k^{th} decision be the item *type number* t_k of the k^{th} item to be added to the knapsack. It is convenient to allow the weight corresponding to a selection of items to take on any integer value $w = 0, \ldots, W$. To achieve this we add to the problem "slack" items (of type 0) having weight $w_0 = 1$, and value $v_0 = 0$. The DDT for this problem, called DDT1, is given in Figure 4, where the possible decisions at each node are to add an item of type 0, 1, 2 or 3 to the knapsack. The letters A, B, ... F on the nodes indicate the aggregate classes. The associated DPN, called DPN1, is shown in Figure 5.

5.1 The formulation phase

We now formally apply the formulation phase developed in the previous section to our general knapsack problem by formally constructing the DPN from the DDT given above, i.e. where the decision corresponding to this DDT is: "the item type number to be next added to the knapsack." Denote the DDT by $\mathcal{T} \equiv (T = (N, A, r), P, +)$. Furthermore, for all $u \in N$, let $\mathcal{T}_u$ be the decision subtree rooted at u, that is: $\mathcal{T}_u \equiv (T_u = (N_u, A_u, u), P_u, +)$, where

$$\begin{aligned} N_u &= \{v \in N \mid \text{there is a directed path from } u \text{ to } v \text{ in } T\} \\ A_u &= (N_u \times N_u) \cap A \\ P_u &= P \text{ restricted to } A_u. \end{aligned}$$

In the next lemma we show that for this DDT the equivalence relation E of Definition 4.4 becomes: uEv if and only if $\omega(r, u) = \omega(r, v)$ where $\omega(i, j)$ is $\varphi(i, j)$ in Definition 4.3, with profit v_i replaced by weight w_i. That is, two nodes in this DDT are equivalent if and only if the weight of the knapsack at those nodes are equal.

Lemma 5.1 *Two subtrees* $\mathcal{T}_u$ *and* $\mathcal{T}_v$ *(*$u, v \in N$*) are equivalent if and only if* $\omega(r, u) = \omega(r, v)$.

Proof: First, let $u, v \in N$ satisfy $\omega(r, u) = \omega(r, v)$. Now for each $i \in N_u$, let $\mathbf{d}^i$ denote the decision sequence leading from u to i. That is, for some n, $\mathbf{d}^i = (d_1, \ldots, d_n)$, where

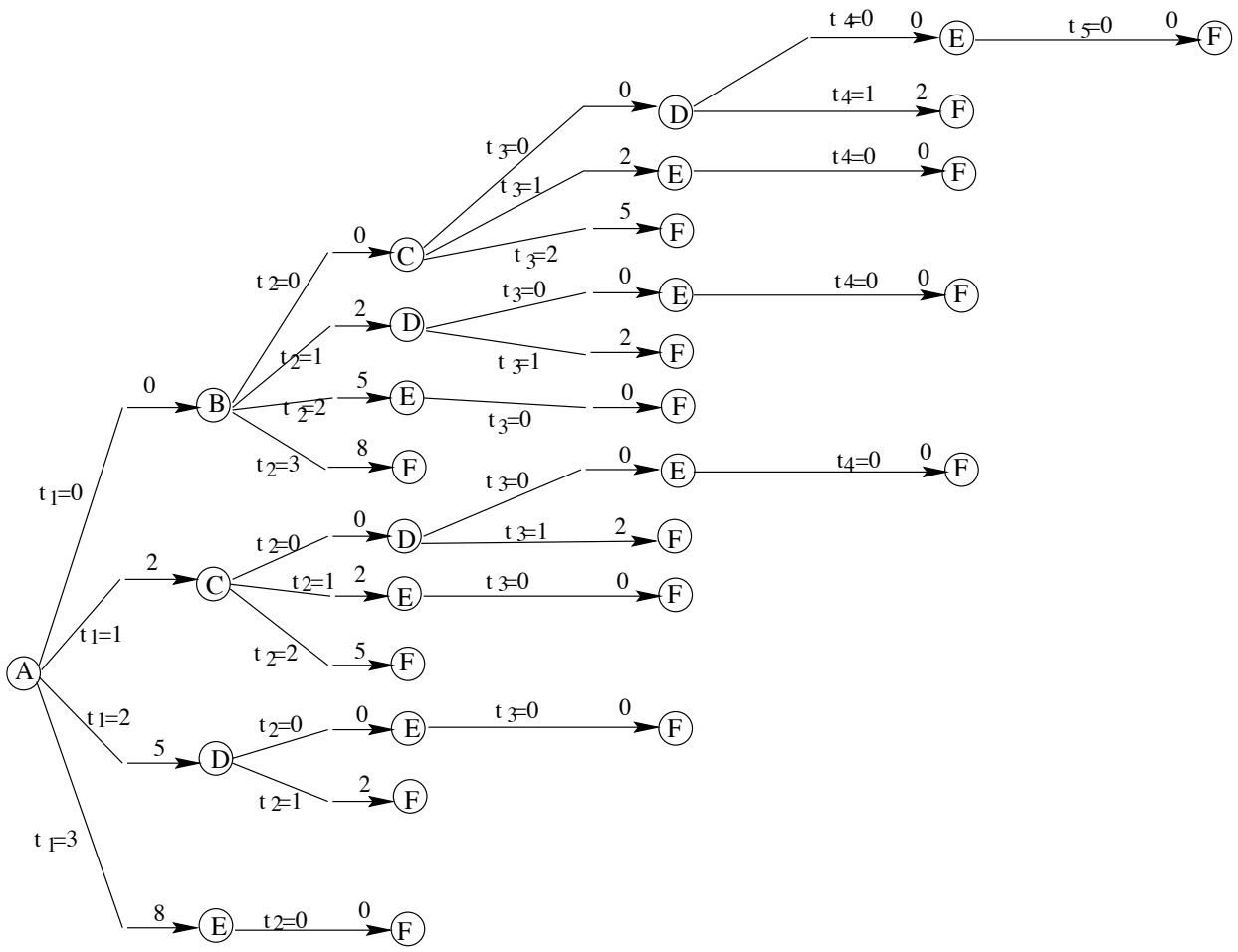

Figure 4: *DDT1 for the knapsack problem, with data from Table 2.*

$d_j \in \{0, \ldots, K\}$ is the type of item to be added to the knapsack next. Applying this decision sequence to v yields a feasible solution since the weight of all items in the knapsack after applying this sequence is $\omega(r, v) + \sum_{j=1}^{n} w_{d_j} = \omega(r, u) + \sum_{j=1}^{n} w_{d_j}$, which is a feasible weight. This identifies a unique node in N_v, which we will call $g(i)$. We have thus defined a function $g : N_u \to N_v$. This function is clearly one-to-one. It is also onto, since each feasible decision sequence from v is feasible from u. So, $N_v = \{g(i) \mid i \in N_u\}$ and $A_v = (N_v \times N_v) \cap A = \{(g(i), g(j)) \mid (i, j) \in A_u\}$, which implies that g is an isomorphism. Moreover, it preserves arc profits, since the profit $P(i, j)$ only depends on the decision associated with moving from node i to node j. If $(i, j) \in A_u$, then the same decision moves from $g(i)$ to $g(j)$, so that $P(i, j) = P(g(i), g(j))$. Therefore, $\mathcal{T}_u$ and $\mathcal{T}_v$ are equivalent.

Now suppose $\mathcal{T}_u$ and $\mathcal{T}_v$ are equivalent for some $u, v \in N$. One feasible decision sequence from u is to add $W - \omega(r, u)$ slack items. Since the subtrees are equivalent, this must be a feasible decision sequence from v as well, so $W - \omega(r, v)$ (the remaining weight at v) should at least be equal to $W - \omega(r, u)$ (the weight added by the decision sequence). Thus, $\omega(r, v) \leq \omega(r, u)$. Similarly, a feasible decision sequence from v (and thus from u) is to add

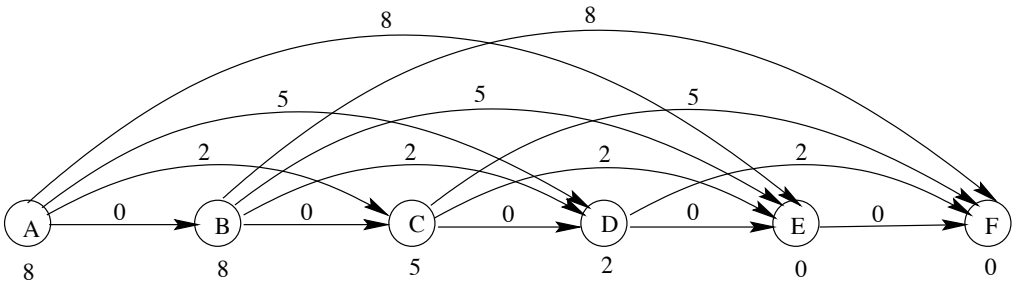

Figure 5: *DPN1 associated with DDT1 of Figure 4.*

$W - \omega(r, v)$ slack items, which implies $\omega(r, u) \leq \omega(r, v)$. Thus $\omega(r, u) = \omega(r, v)$. □

Recall that Assumption 4.6 is satisfied when the path profit function is the sum of the arc profits, so that Theorem 4.7 can be used to derive a functional equation through which the optimal solution to the decision problem can be computed recursively. The following lemma gives the functional equation for the knapsack problem corresponding to the DDT described above.

Lemma 5.2 *The functional equation corresponding to DDT1 for the knapsack problem is:*

$$f(w) = \begin{cases} 0 & w = 0 \\ \max\left(f(w-1), \max_{\{k=1,\ldots,K|w_k \leq w\}}[v_k + f(w - w_k)]\right) & w = 1, \ldots, W. \end{cases}$$

The optimal solution to the knapsack problem is then $f(W)$.

Proof: By Lemma 5.1 the set of nodes for the DPN is given by the set of attainable weights, i.e.

$$\mathcal{N} = \{0, \ldots, W\}.$$

Furthermore, it is easy to see that $(w, w') \in \mathcal{A}$ if and only if there is some $k \in \{0, \ldots, K\}$ such that $w' - w = w_k$, and that the corresponding profit is $\mathcal{P}(w, w') = v_k$. Thus, by Theorem 4.7 (where $f(w)$ denotes the maximal value of a knapsack filled to weight w) we have

$$f(w) = \begin{cases} 0 & w = 0 \\ \max_{\{k=0,\ldots,K|w_k \leq w\}}[v_k + f(w - w_k)] & w = 1, \ldots, W. \end{cases}$$

Substituting $w_0 = 1$ and $v_0 = 0$ we obtain

$$f(w) = \begin{cases} 0 & w = 0 \\ \max\left(f(w-1), \max_{\{k=1,\ldots,K|w_k \leq w\}}[v_k + f(w - w_k)]\right) & w = 1, \ldots, W. \end{cases}$$

The remainder of the lemma follows by definition of f. □

Note that the optimal solution to the knapsack problem can be found using the above functional equation in $\mathcal{O}(KW)$ time, since the DPN contains $W + 1$ nodes, each (but one) having $\mathcal{O}(K)$ arcs emanating from it.

5.2 Alternative formulations

We have shown how a functional equation for a given problem can be uniquely derived from the DDT formulation of the problem via the DPN. If we wish to generate a *different* functional equation (i.e. a different DP formulation), we must first represent the problem as a different decision tree. The resulting number of states in DPN depends heavily on the original DDT. Thus the "art" of dynamic programming can be completely encapsulated in the choice of the underlying decision tree.

5.2.1 DDT2 and DPN2 for the knapsack problem

An alternative to decision rule 1 is to decide, at the i^{th} step, the number of items x_i of type i to add to the knapsack, $i = 1, 2, \ldots, K$. The corresponding decision tree (DDT2) is shown in Figure 6, for the data given in Table 2. Aggregation of nodes leads to DPN2 as shown in Figure 7.

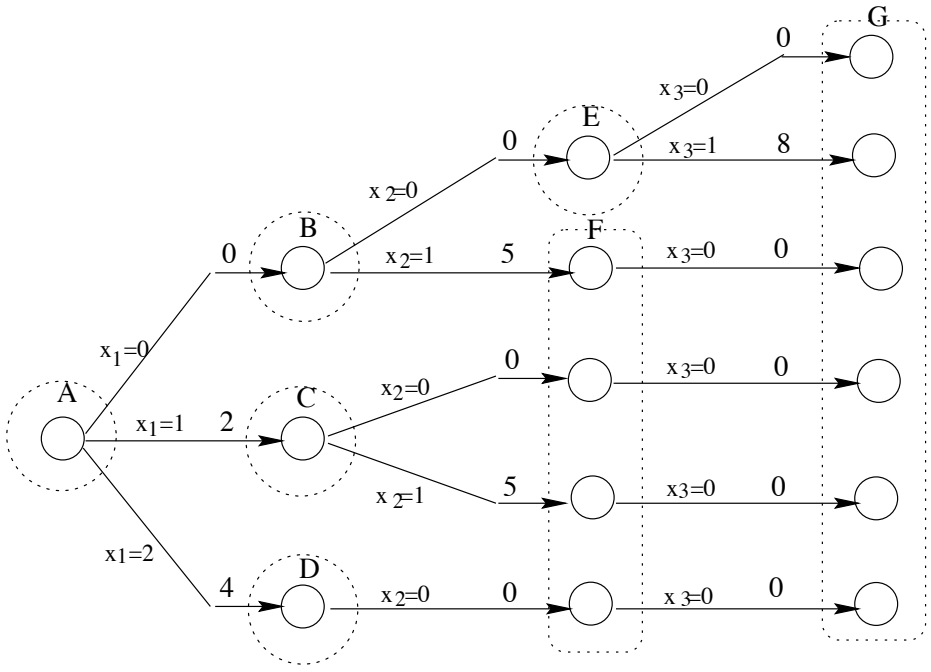

Figure 6: *DDT2 for the knapsack problem, using the data from Table 2.*

Figure 7 suggests that one state interpretation for the nodes of DPN2 is: $s = (n, w')$, where n is the last item type considered, and w' is the usable weight remaining in the knapsack. Note that in this case the range of possible state values depends in a complex way on the item weights. This makes it difficult to write down a general functional equation and to specify the computational complexity.

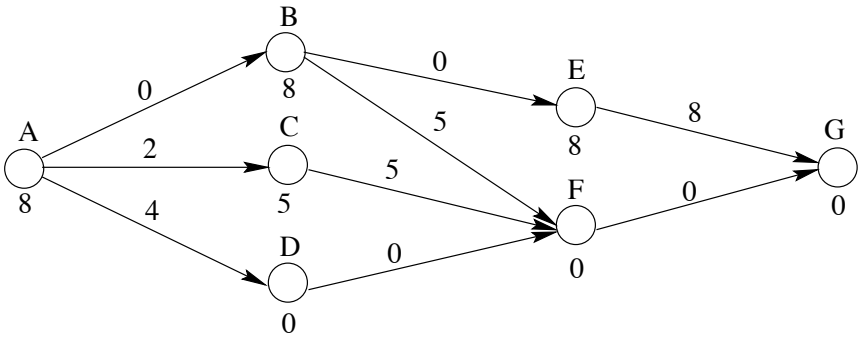

Figure 7: *DPN2 corresponding to DDT2.*

5.2.2 DDT3 and DPN3 for the knapsack problem

A third DDT for the knapsack problem is similar to DDT2. The difference is that, in step i, a number x_0^i of slack items is also added at the same time. Figure 8 shows (part of) the resulting decision tree DDT3 for the data given in Table 2 . The corresponding DPN3 is shown in Figure 9.

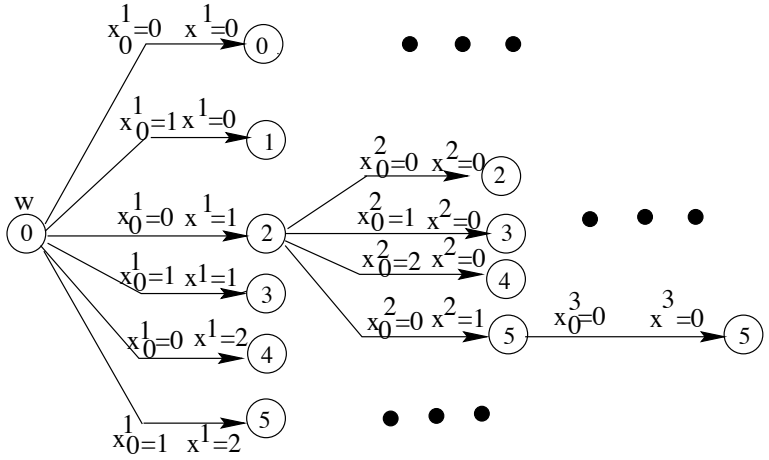

Figure 8: *DDT3 for the knapsack problem, using the data from Table 2.*

As can be seen in Figure 9, the state variable that characterizes DPN3 is $s = (n, w)$, where n is the last item type considered, and w is the weight of the first n item types (including the slack items) allocated so far. If $f_n(w)$ is defined to be the maximum value of the knapsack with weight at most w, using items of types $i = 1, \ldots, n$ only, then the corresponding functional equation is:

$$f_n(w) = \begin{cases} 0 & n = 0, \quad w = 1, \ldots, W \\ \max\limits_{x=0,1,\ldots,\lfloor \frac{w}{w_n} \rfloor} [v_n x + f_{n-1}(w - w_n x)] & n = 1, \ldots, N, \quad w = 1, \ldots, W. \end{cases}$$

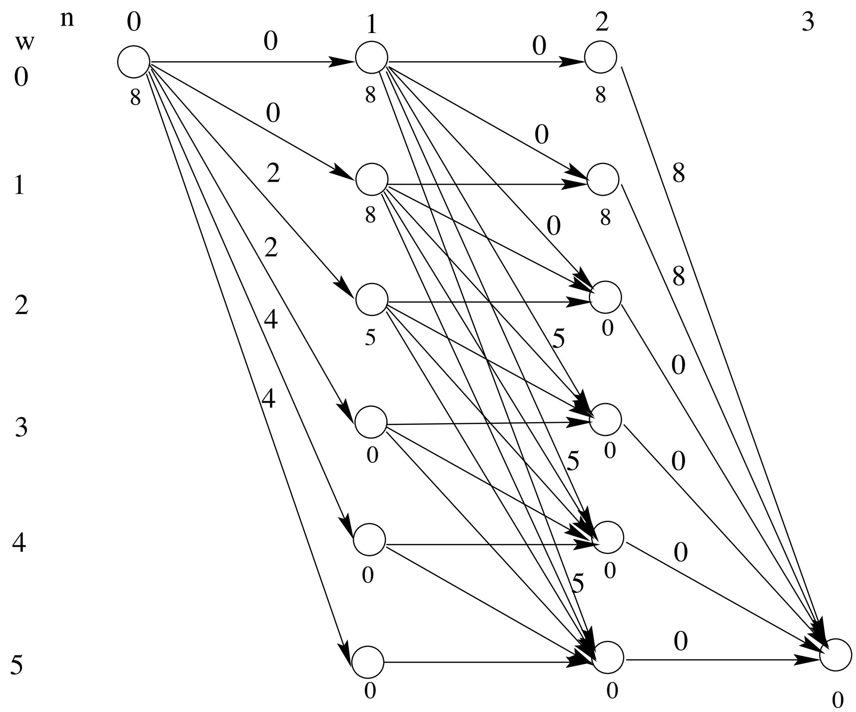

Figure 9: *DPN3 corresponding to DDT3.*

The use of recursive fixing to evaluate $f_n(w)$ solves, in fact, a series of knapsack problems, increasing the number of items considered by one each iteration until all N items are considered. The computational complexity for solving the functional equation is $\mathcal{O}(NW^2)$. Note that this complexity is strictly worse than the complexity for DDT1.

6 Concluding remarks

We have presented a method for formulating all possible dynamic programming representations of sequential decision problems. The procedure requires an initial deterministic decision tree, from which a unique dynamic programming network is constructed using node aggregation. These networks typically allow a computationally attractive maximal return path solution via recursive solution of the associated dynamic programming functional equations. We argue that the aggregation process that generates the DPN from a DDT not only offers a direct way to write appropriate recursive equations, but also automatically identifies that elusive feature of dynamic programming: the state variable. Alternative DP formulations of a given underlying sequential decision problem can therefore be seen to be manifestations of alternative decision tree formulations of that same problem. The question as to which decision tree representation gives rise to the most efficient DP formulation can then perhaps be rigorously posed and potentially answered. We have restricted consideration in this paper to the finite deterministic case although similar aggregation ideas seem generalizable to both

infinite and stochastic problems.

References

[1] D.M. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[2] J. Chou. "Accelerating the Solution of Dynamic Programs through State Aggregation," Unpublished Ph.D. dissertation, Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, Michigan 48109 1995.

[3] E.V. Denardo. *Dynamic Programming: Models and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[4] E.V. Denardo and L.G. Mitten. Elements of Sequential Decision Processes. *Journal of Industrial Engineering*, 18:106–112, 1967.

[5] S.E. Dreyfus and A. Law. *The Art and Theory of Dynamic Programming*. Academic Press, New York, 1978.

[6] S.E. Elmaghraby. Teh Concept of State in Discrete Dynamic Programming. *Journal of Mathematical Analysis and Applications*, 29:523–557, 1970.

[7] R.M. Karp and M. Held. Finite-State Processes and Dynamic Programming. *SIAM Journal of Appliedf Mathematics*, 15:693–717,1967.

[8] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, New York, 1990.

[9] L.G. Mitten. Composition Principles for Synthesis of Optimal Multistage Processes. *Operations Research*, 12:610–619, 1964.

[10] G.L. Nemhauser. *Introduction to Dynamic Programming.* John Wiley and Sons, New York, 1966.

[11] D.J. White. *Dynamic Programming.* Holden-Day, San Francisco, 1969.