Control and Cybernetics

vol. **35** (2006) No. 3

Canonical greedy algorithms and dynamic programming

by

Art Lew

Department of Information and Computer Sciences
University of Hawaii at Manoa, USA
e-mail: artlew@hawaii.edu

Abstract: There has been little work on how to construct greedy algorithms to solve new optimization problems efficiently. Instead, greedy algorithms have generally been designed on an ad hoc basis. On the other hand, dynamic programming has a long history of being a useful tool for solving optimization problems, but is often inefficient. We show how dynamic programming can be used to derive efficient greedy algorithms that are optimal for a wide variety of problems. This approach also provides a way to obtain less efficient but optimal solutions to problems where derived greedy algorithms are nonoptimal.

Keywords: greedy algorithm, dynamic programming, Dijkstra.

1. Introduction

It is common, in computer science textbooks (see e.g. Cormen et al., 2001; Horowitz, Sahni and Rajasekaran, 1996), to treat dynamic programming (DP) and greedy algorithms as separate and independent methodologies for solving optimization problems. In operations research textbooks (Hillier and Lieberman, 2001; Winston, 2004), greedy algorithms are typically mentioned only casually, by identifying the use of greedy strategies in passing as they are encountered in the description of specific algorithms. In this paper, we discuss greedy algorithms as a general methodology, but in the context of DP, providing first a constructive means of deriving greedy algorithms to more efficiently solve certain optimization problems, and then a framework in which greedy algorithms can be further analyzed.

Informally, a "greedy" optimization algorithm solves a global minimization or maximization problem by making a sequence of locally optimal decisions. A decision is "local" or "myopic" if it is based on partial information that does not include global knowledge about future consequences of current decisions (Heyman and Sobel, 1984); thus, the current decision, once made, might turn out to be nonoptimal. We say that a greedy optimization algorithm is *optimal* if it always results in finding the global optimum solution, in which case we say that each of the sequential decisions made by algorithm is also optimal. In this paper, a solution must be exact rather than approximate. While the intent of greediness is computational efficiency, to lower the time or space requirements of the solution process, efficiency is not part of our definition of greedy algorithms. However, efficiency is generally a by-product. This view of greedy algorithms has been formalized in various ways, generally by defining a restrictive context, such as the class of combinatorial optimization problems, which is the context that we also choose herein. We note that this restriction to combinatorial or discrete problems excludes gradient-based algorithms for solving continuous optimization problems. Gradient search algorithms, such as "steepest" ascent or descent, are of course examples where a greedy decision might lead to a local rather than global optimum. We will also restrict our attention to the context of DP problems, which excludes, for example, the basic simplex algorithm of linear programming where the choice of "pivot" is in some sense made greedily. Given a DP solution of an optimization problem, we address the question of how it can be solved using greedy principles, hopefully more efficiently.

In Section 2, we provide a brief introduction to DP as a method for solving optimization problems that can be formulated as sequential decision processes. In Section 3, we discuss relationships between greedy algorithms and DP. We show that a greedy policy, based perhaps upon heuristic principles, can used to speed up DP solutions. The problem of how to derive a greedy policy in the first place has seldom been addressed. In Section 4, we formally derive a greedy algorithm from a DP solution to a discrete optimization problem. We call greedy algorithms that can be so derived "canonical" ones. We will show that the class of canonical greedy algorithms is large, including, as prominent examples,

the minimum spanning tree algorithms of Kruskal or Prim, where the greedy policy is to choose the eligible arc with smallest weight;

the prefix coding tree algorithm of Huffman, where the greedy policy is to choose the pair of items having smallest weight;

the shortest path algorithm of Dijkstra, where the greedy policy is to choose the vertex closest to the source;

linear sequencing algorithms, where the greedy policy is, for example, to choose to place the item (to be linear searched) that has the maximum probability, or to choose to schedule the job that has shortest processing time;

the deadline scheduling algorithm, where the greedy policy is to choose to schedule the feasible job meeting its deadline that is most profitable.

In Section 5, we show that DP can also be used to derive some noncanonical greedy algorithms that are optimal as well as some canonical greedy algorithms that are not optimal. In addition to providing a *constructive* means of obtaining certain greedy algorithms, DP also provides a theoretical framework in which to prove their optimality.

2. Preliminaries

In this section, we introduce DP in terms that will be useful in our later discussion of greedy algorithms in which a sequence of decisions is made. We specifically distinguish between deciding firstly on the best order in which the decisions are to be made, and secondly on the best choice that is made for each decision. These problems are complicated by the fact that decisions are in general interrelated, imposing constraints upon each other. Moreover, a key to the use of DP is separability of values (costs or profits) that are associated with individual decisions.

For an optimization problem of the form $\mathbf{opt}_{d\in\Delta}\{\mathbf{H}(d)\}$, d is called the *decision*, which is chosen from a set of eligible decisions Δ, function $\mathbf{H}$ is called the *optimand*, and $\mathbf{H}^* = \mathbf{H}(d^*)$ is called the *optimum,* where d^* is that value of $d \in \Delta$ for which $\mathbf{H}(d)$ has the optimal (minimum or maximum) value. We also say that d^* *optimizes* $\mathbf{H}$, and write $d^* = \mathbf{argopt}_d\{\mathbf{H}(d)\}$. Many optimization problems consist of finding a set of decisions, $\{d_1, d_2, \ldots, d_n\}$, that taken together yield the optimum $\mathbf{H}^*$ of an objective function $h[d_1, d_2, \ldots, d_n]$. Solution of such problems by *enumeration*, i.e., by evaluating $h[d_1, d_2, \ldots, d_n]$ concurrently, for all possible combinations of values of its decision arguments, is called the "brute force" approach; this approach is manifestly inefficient. Rather than making decisions concurrently, we assume the decisions may be made in some specified sequence, say, $(d_1, d_2, \ldots, d_n)$, i.e., such that

$$\begin{aligned}\mathbf{H}^* &= \mathbf{opt}_{(d_1,d_2,\ldots,d_n)\in\Delta}\{h[d_1, d_2, \ldots, d_n]\}\\ &= \mathbf{opt}_{d_1\in D_1}\{\mathbf{opt}_{d_2\in D_2}\{\ldots\{\mathbf{opt}_{d_n\in D_n}\{h[d_1, d_2, \ldots, d_n]\}\}\}\}, \qquad (1)\end{aligned}$$

in what are known as *sequential decision processes*, where the ordered set $(d_1, d_2, \ldots, d_n)$ belongs to some *decision space* $\Delta = D_1 \times D_2 \times \ldots \times D_n$, for $d_i \in D_i$. Examples of decision spaces include: $\Delta = B^n$, the special case of Boolean decisions, where each *decision set* D_i equals $B = \{0, 1\}$; and $\Delta = \Pi(\mathbf{D})$, a permutation of a set of eligible decisions $\mathbf{D}$. The latter illustrates the common situation where decisions d_i are interrelated, e.g., where they satisfy constraints such as $d_i \neq d_j$ or $d_i + d_j \leqslant M$. In general, each decision set D_i depends on the decisions $\{d_1, d_2, \ldots, d_{i-1}\}$ that are earlier in the specified sequence, i.e., $d_i \in D_i(d_1, d_2, \ldots, d_{i-1})$. Thus, to show this dependence explicitly, we rewrite the foregoing equation in the form

$$\begin{aligned}\mathbf{H}^* &= \mathbf{opt}_{(d_1,d_2,\ldots,d_n)\in\Delta}\{h[d_1, d_2, \ldots, d_n]\}\\ &= \mathbf{opt}_{d_1\in D_1}\{\mathbf{opt}_{d_2\in D_2(d_1)}\{\ldots\{\mathbf{opt}_{d_n\in D_n(d_1,d_2,\ldots,d_{n-1})}\{h[d_1, d_2, \ldots, d_n]\}\}\}\}. \qquad (2)\end{aligned}$$

This nested set of optimization operations is to be performed from inside-out (right-to-left), the innermost optimization yielding the optimal choice for d_n as a function of the possible choices for $d_1, \ldots, d_{n-1}$, denoted $d_n^*(d_1, \ldots, d_{n-1})$, and the outermost optimization $\mathbf{opt}_{d_1 \in D_1}\{h[d_1, d_2^*, \ldots, d_n^*]\}$ yielding the optimal choice for d_1, denoted d_1^*. Note that while the initial or "first" decision d_1 in the specified sequence is the outermost, the optimizations are performed inside-out, each depending upon outer decisions. Furthermore, while the optimal solution may be the same for any sequencing of decisions, e.g.,

$$\begin{aligned} &\mathbf{opt}_{d_1 \in D_1}\{\mathbf{opt}_{d_2 \in D_2(d_1)}\{\ldots\{\mathbf{opt}_{d_n \in D_n(d_1,d_2,\ldots,d_n-1)}\{h[d_1, d_2, \ldots, d_n]\}\}\}\} \\ = &\mathbf{opt}_{d_n \in D_n}\{\mathbf{opt}_{d_{n-1} \in D_{n-1}(dn)}\{\ldots\{\mathbf{opt}_{d_1 \in D_1(d_2,\ldots,d_n)}\{h[d_1, d_2, \ldots, d_n]\}\}\}\}, \end{aligned} \tag{3}$$

the decision sets D_i may differ since they depend on different outer decisions. Thus, *efficiency may depend upon the order in which decisions are made.*

Referring to the foregoing equation, for a given sequencing of decisions, if the outermost decision is "tentatively" made initially, whether or not it is optimal depends upon the ultimate choices d_i^* that are made for subsequent decisions d_i; i.e.,

$$\begin{aligned} \mathbf{H}^* &= \mathbf{opt}_{d_1 \in D_1}\{\mathbf{opt}_{d_2 \in D_2(d_1)}\{\ldots\{\mathbf{opt}_{d_n \in D_n(d_1,d_2,\ldots,d_{n-1})}\{h[d_1, d_2, \ldots, d_n]\}\}\}\} \\ &= \mathbf{opt}_{d_1 \in D_1}\{h[d_1, d_2^*(d_1), \ldots, d_n^*(d_1)]\}, \end{aligned} \tag{4}$$

where each of the choices $d_i^*(d_1)$ for $i = 2, \ldots, n$ is constrained by – i.e., is a function of – the choice for d_1. Note that determining the optimal choice $d_1^* = \mathbf{arg\ opt}_{d_1 \in D_1}\{h[d_1, d_2^*(d_1), \ldots, d_n^*(d_1)]\}$ requires evaluating h for all possible choices of d_1 unless there is some reason that *certain choices can be excluded from consideration based upon a priori* (given or derivable) *knowledge that they cannot be optimal.* One such class of algorithms would choose $d_1 \in D_1$ independently of (but still constrain) the choices for $d_2, \ldots, d_n$, i.e., by finding the solution of a problem of the form $\mathbf{opt}_{d_1 \in D_1}\{\mathbf{H}'[d_1]\}$ for a function $\mathbf{H}'$ of d_1 that is myopic in the sense that it does not depend on other choices d_i. Such an algorithm is optimal if the locally optimal solution of $\mathbf{opt}_{d_1}\{\mathbf{H}'[d_1]\}$ yields the globally optimal solution $\mathbf{H}^*$.

Suppose that the objective function h is (*strongly*) *separable* in the sense that

$$h[d_1, d_2, \ldots, d_n] = C_1(d_1) \circ C_2(d_2) \circ \ldots \circ C_n(d_n), \tag{5}$$

where the decision-cost functions C_i represent the costs (or profits) associated with the individual decisions d_i, and where $\circ$ is a binary operation, usually addition or multiplication, with $\mathbf{opt}_d\{a \circ C(d)\} = a \circ \mathbf{opt}_d\{C(d)\}$ for any a that does not depend upon d. In the context of sequential decision processes, the cost C_n of making decision d_n may be a function not only of the decision itself, but also of the state $(d_1, d_2, \ldots, d_{n-1})$ in which the decision is made. To emphasize this, we will rewrite the above as

$$h[d_1, d_2, \ldots, d_n] = C_1(d_1|\emptyset) \circ C_2(d_2|d_1) \circ \ldots \circ C_n(d_n|d_1, \ldots, d_{n-1}). \tag{6}$$

We now define h as (*weakly*) *separable* if

$$h[d_1, d_2, \ldots, d_n] = C_1(d_1) \circ C_2(d_1, d_2) \circ \ldots \circ C_n(d_1, d_2, \ldots, d_n), \tag{7}$$

(strong separability is, of course, a special case). If h is (weakly) separable, we then have

$$\begin{aligned}
&\mathbf{opt}_{d_1 \in D_1}\{\mathbf{opt}_{d_2 \in D_2(d_1)}\{\ldots\{\mathbf{opt}_{d_n \in D_n(d_1,d_2,\ldots,d_{n-1})}\{h[d_1, d_2, \ldots, d_n]\}\}\}\} \\
&= \mathbf{opt}_{d_1 \in D_1}\{\mathbf{opt}_{d_2 \in D_2(d_1)}\{\ldots\{\mathbf{opt}_{d_n \in D_n(d_1,d_2,\ldots,d_{n-1})}\{C_1(d_1|\emptyset) \circ C_2(d_2|d_1) \\
&\qquad \circ \ldots \circ\ C_n(d_n|d_1, \ldots, d_{n-1})\}\}\}\} \\
&= \mathbf{opt}_{d_1 \in D_1}\{C_1(d_1|\emptyset) \circ \mathbf{opt}_{d_2 \in D_2(d_1)}\{C_2(d_2|d_1) \\
&\qquad \circ \ldots \mathbf{opt}_{d_n \in D_n(d_1,d_2,\ldots,d_{n-1})}\{C_n(d_n|d_1, \ldots, d_{n-1})\}\}\}.
\end{aligned} \tag{8}$$

Let the function $f[i|d_1, \ldots, d_{i-1}]$ be defined as the optimal solution of the sequential decision process where the decisions $d_1, \ldots, d_{i-1}$ have been made and the decisions $d_i, \ldots, d_n$ remain to be made; i.e.,

$$\begin{aligned}
&f[i|d_1, \ldots, d_{i-1}] \\
&= \mathbf{opt}_{d_i}\{\mathbf{opt}_{d_{i+1}}\{\ldots\{\mathbf{opt}_{d_n}\{C_i(d_i|d_1, \ldots, d_{i-1}) \circ C_{i+1}(d_{i+1}|d_1, \ldots, d_i) \\
&\qquad \circ \ldots \circ C_n(d_n|d_1, \ldots, d_{n-1})\}\}\}\}.
\end{aligned} \tag{9}$$

Explicit mentions of the decision sets D_i are omitted here for convenience. We have then

$$\begin{aligned}
&f[1|\emptyset] = \mathbf{opt}_{d_1}\{\mathbf{opt}_{d_2}\{\ldots\{\mathbf{opt}_{d_n}\{C_1(d_1|\emptyset) \circ C_2(d_2|d_1) \\
&\qquad \circ \ldots \circ C_n(d_n|d_1, \ldots, d_{n-1})\}\}\}\} \\
&= \mathbf{opt}_{d_1}\{C_1(d_1|\emptyset) \circ \mathbf{opt}_{d_2}\{C_2(d_2|d_1) \\
&\circ \ldots \mathbf{opt}_{d_{n-1}}\{C_{n-1}(d_{n-1}|d_1, \ldots, d_{n-2}) \circ\ \mathbf{opt}_{d_n}\{C_n(d_n|d_1, \ldots, d_{n-1})\}\}\}\} \\
&= \mathbf{opt}_{d_1}\{C_1(d_1|\emptyset) \circ f[2|d_1]\}.
\end{aligned} \tag{10}$$

Generalizing, we conclude that

$$f[i|d_1, \ldots, d_{i-1}] = \mathbf{opt}_{d_i \in D_i(d_1,d_2,\ldots,d_{i-1})}\{C_i(d_i|d_1, \ldots, d_{i-1}) \circ f[i+1|d_1, \ldots, d_i]\}. \tag{11}$$

This equation is a recursive functional equation; we call it a functional equation since the unknown in the equation is a function f, and it is recursive since f is defined in terms of f (but having different arguments). It is also called the *dynamic programming functional equation*(DPFE) for the given optimization problem. (In this paper, we assume that we are given DPFEs that are *properly* formulated, i.e., that their solutions exist; we address only issues of how to obtain these solutions.)

Dynamic programming (Bellman, 1957) is a method that in general solves optimization problems involving sequential decision processes by determining, for each decision, subproblems that can be solved in like fashion, such that an optimal solution of the original problem consists of optimal subsolutions of the subproblems. This defining characteristic of DP is known as the "principle of optimality" or "optimal substructure" property in operations research and computer science literature (Cormen at al., 2001; Hillier and Lieberman, 2001; Horowitz, Sahni, and Rajasekaran, 1996; Winston, 2004). In this paper, we are concerned with the computational solution of problems, for which the principle of optimality is given to hold. For DP to be computationally efficient (relative to enumeration), there should be common subproblems such that subproblems of one are subproblems of another, and in this event a solution need only be found once and reused as often as necessary. (However, we do not incorporate this requirement as part of our definition of DP.) Then, the problem of solving for $f[i|d_1, \ldots, d_{i-1}]$ depends upon the subproblem of solving for $f[i+1|d_1, ..., d_i]$. If we define the *state* $s = (d_1, ..., d_{i-1})$ as the sequence of the first i decisions, where $i = |s| + 1 = |\{d_1, \ldots, d_{i-1}\}| + 1$, we may rewrite the DPFE in the form

$$f[s] = \mathbf{opt}_{d_i \in D(s)}\{C(d_i|s) \circ f[s']\}, \quad \text{for} \quad s \in \Sigma, \tag{12}$$

where $s' = (d_1, \ldots, d_i)$, and Σ is a set of possible states, $\emptyset$ being the initial state. Since the DPFE is recursive, its solution requires *base cases* (or "termination" or "boundary" conditions), such as $f[s] = 0$ when s is a state in which no decision is eligible, i.e., when $D(s) = \emptyset$, or $f[s'] = \pm\infty$ when s' results from an ineligible decision or $s \notin \Sigma$. It should be noted that the sequence of decisions need not be limited to a fixed n, but may be of indefinite length, terminating when a base case is reached.

Application: Linear Search (Lew, 1985)

To illustrate the key concepts associated with DP that will prove useful in our later discussions, we examine a concrete example, the optimal "linear search" problem. This is the problem of permuting the elements of an array A of size N, whose element x has probability p_x, so as to optimize the linear search process by minimizing the "cost" of a permutation, defined as the expected number of comparisons required. For example, let $A = \{a, b, c\}$ and $p_a = .2$, $p_b = .5$, and $p_c = .3$. There are six permutations, namely, *abc*, *acb*, *bac*, *bca*, *cab*, *cba*; the cost of the fourth permutation *bca* is 1.7, which can be calculated in several ways, such as [Method S]: $1 \cdot p_b + 2 \cdot p_c + 3 \cdot p_a$ and [Method W]: $(p_a + p_b + p_c) + (p_a + p_c) + (p_a)$. The optimization problem can be regarded as a sequential decision process where three decisions must be made as to where the elements of A are to be placed in the final permuted array A'. The decisions are: which element is to be placed at the beginning of A', which element is to be placed in the middle of A', and which element is to be placed at the end of A'. The order in which these decisions are made does not necessarily matter, at least insofar as obtaining the correct answer is concerned; e.g., to obtain the permutation bca, our first decision may be to place element c in the middle of A'.

Of course, some orderings of decisions may lead to greater efficiency than others. Moreover, the order in which decisions are made affects later choices; if c is chosen in the middle, it cannot be chosen again. That is, the decision set for any choice depends upon (is constrained by) earlier choices. In addition, the cost of each decision should be separable from other decisions.

To obtain this separability, we must usually take into account the order, in which decisions are made. For Method S, the cost of placing element x in the i-th location of A' equals $i \cdot p_x$ regardless of when the decision is made. On the other hand, for Method W, the cost of a decision depends upon when the decision is made, more specifically upon its decision set. If the decisions are made in order from the beginning to the end of A', then the cost of deciding which member d_i of the respective decision set D_i to choose next equals $\Sigma_{x \in D_i} p_x$, the sum of the probabilities of the elements in $D_i = A - \{d_1, \ldots, d_{i-1}\}$. For example, let d_i denote the decision of which element of A to place in position i of A', and let D_i denote the corresponding decision set, where $d_i \in D_i$. If the decisions are made in the order $i = 1, 2, 3$, then $D_1 = A$, $D_2 = A - d_1$, $D_3 = A - d_1 - d_2$. For Method S, if the objective function is written in the form $h(d_1, d_2, d_3) = 1 \cdot p_{d_1} + 2 \cdot p_{d_2} + 3 \cdot p_{d_3}$, then

$$\begin{aligned} f[\emptyset] &= \mathbf{min}_{d_1 \in A}\{\mathbf{min}_{d_2 \in A-d_1}\{\mathbf{min}_{d_3 \in A-d_1-d_2}\{1 \cdot p_{d_1} + 2 \cdot p_{d_2} + 3 \cdot p_{d_3}\}\}\} \\ &= \mathbf{min}_{d_1 \in A}\{1 \cdot p_{d_1} + \mathbf{min}_{d_2 \in A-d_1}\{2 \cdot p_{d_2} + \mathbf{min}_{d_3 \in A-d_1-d_2}\{3 \cdot p_{d_3}\}\}\}. \end{aligned} \tag{13}$$

For Method W, if the objective function is written in the form $h(d_1, d_2, d_3) = \Sigma_{x \in A} p_x + \Sigma_{x \in A-d_1} p_x + \Sigma_{x \in A-d_1-d_2} p_x$, then

$$\begin{aligned} f[\emptyset] &= \mathbf{min}_{d_1 \in A}\{\mathbf{min}_{d_2 \in A-d_1}\{\mathbf{min}_{d_3 \in A-d_1-d_2}\{\Sigma_{x \in A} p_x + \Sigma_{x \in A-d_1} p_x \\ &\qquad + \Sigma_{x \in A-d_1-d_2} p_x\}\}\} \\ &= \mathbf{min}_{d_1 \in A}\{\Sigma_{x \in A} p_x + \mathbf{min}_{d_2 \in A-d_1}\{\Sigma_{x \in A-d_1} p_x \\ &\qquad + \mathbf{min}_{d_3 \in A-d_1-d_2}\{\Sigma_{x \in A-d_1-d_2} p_x\}\}\}. \end{aligned} \tag{14}$$

However, if the decisions are made in reverse order $i = 3, 2, 1$, then $D_3 = A$, $D_2 = A - d_3$, $D_1 = A - d_2 - d_3$, and the above must be revised accordingly. It should also be noted that if $h(d_1, d_2, d_3) = 0 + 0 + (1 \cdot p_{d_1} + 2 \cdot p_{d_2} + 3 \cdot p_{d_3})$, where all of the cost is associated with the final decision d_3, then

$$f[\emptyset] = \mathbf{min}_{d_1 \in A}\{0 + \mathbf{min}_{d_2 \in A-d_1}\{0 + \mathbf{min}_{d_3 \in A-d_1-d_2}\{1 \cdot p_{d_1} + 2 \cdot p_{d_2} + 3 \cdot p_{d_3}\}\}\}, \tag{15}$$

which is equivalent to enumeration. We conclude from this example that care must be taken in defining decisions and their interrelationships, and how to attribute separable costs to these decisions. Examples 3.1 and 4.1 below further develop these points.

3. Greedy policies

Consider a sequential decision process where each decision d is associated with an optimization problem of the form $\mathbf{opt}_{d\in D}\{\mathbf{H}[d]\}$.

As before, $d^* = \mathbf{arg\ opt}_{d\in D}\{\mathbf{H}(d)\}$ and $\mathbf{H}^* = \mathbf{H}(d^*)$. In some situations, it is possible to find the result $\mathbf{H}[d^*]$ of the optimization operation without evaluating $\mathbf{H}[d]$ for each d in D, provided there is some way to exclude certain values of d as a possible optimizing value d^*. For example, suppose there exists a linear ordering Π of D such that the optimal policy is the "leading" member of $\Pi(D)$, $\gamma[D] = \mathbf{arg\ opt}_{d\in D}(\Pi(D))$. If $\gamma[D] = d^*$, the problem $\mathbf{opt}_{d\in D}\{\mathbf{H}[d]\}$ can be replaced by the problem $\mathbf{opt}_{d\in D}(\Pi(D))$. This is only useful if the latter problem can be solved more efficiently than the former, which is generally the case if the ordering $\Pi(D)$ can be found using only partial or local information. The decisions themselves are considered *irrevocable*: once a decision d^* is made, it cannot be changed. An optimization algorithm based on such a "locally definable" ordering will be called a *greedy* one. The term greedy can be applied both to the ordering and to the decisions, even if the decisions themselves are Boolean. This is often the case for "priority" algorithms (Borodin, Nielsen, Rackoff, 2002), for which typically an optimal ordering must first be determined.

Consider now optimization problems that can be solved using DP. Assume we are given a (properly formulated) DPFE of the form

$$f[s] = \mathbf{opt}_{d\in D(s)}\{c(d|s) + f[\delta(s,d)]\}, \quad \text{for} \quad s \in \Sigma, \tag{16}$$

where $D(s)$ is the set of possible eligible decisions when in state $s \in \Sigma$, $c(d|s)$ is the cost of making a decision d when in state s, and $s' = \delta(s,d)$ is the next-state resulting from a decision d in state s. The optimal decision d^* is a function of s and $D(s)$, $d^*[s, D(S)] = \mathbf{arg\ opt}_{d\in D(s)}\{c(d|s) + f[\delta(s,d)]\}$. A greedy algorithm would allow us to choose to make a greedy decision $\gamma[s, D(s)]$ to determine $f[s]$ without the use of the DPFE. If $\gamma[s, D(s)] = d^*[s, D(S)]$, we say that the greedy policy is optimal. Note that the use of the DPFE requires the evaluation of $\{c(d|s) + f[\delta(s,d)]\}$ for all possible successor next-states $s' = \delta(s,d)$ of s. Using the greedy policy instead, $f[s]$ need only be evaluated for that successor s' resulting from making decision $\gamma[s, D(s)]$ when in state s, i.e.,

$$f[s] = \{c(\gamma[s, D(s)]|s) + f[\delta(s, \gamma[s, D(s)])]\}. \tag{17}$$

This reduction of the state space, or the number of states for which an optimization operation is required, is what makes such greedy algorithms more efficient. It is assumed of course that $\gamma[s, D(s)]$ can be evaluated relatively efficiently. In simple cases, $D(s) = s$, $\Pi(s)$ is a permutation ordering of s based upon some "priority" rule, and $\gamma(s, D(s))$ is the first member of $\Pi(s)$,

$$\gamma[s, D(s)] = \textbf{arg opt}_{d\in D(s)}(\Pi(D(s))). \tag{18}$$

Example 3.1 (Linear Search) *Consider the problem of permuting the elements of an array A of size N, whose element x has probability p_x, so as to optimize the linear search process. This problem of deciding the ordering of the elements from first to last can be solved by DP (compare Method W of Section 2) using the DPFE*

$$f[S] = \textbf{min}_{x\in S}\{c(x|S) + f[S - \{x\}]\}, \quad \textit{for} \quad s \in 2^A, \tag{19}$$

where $c(x|S) = \sum_{y\in S} p_y$, and 2^A denotes the power set of A. Our goal is to solve for $f[A]$ given the base case $f[\emptyset] = 0$. Note that the cost function $c(x|S)$ depends only on S, not on the decision x. If we order S by descending probability, it can be shown that the first element x^* in this ordering (that has maximum probability) minimizes the set $\{c(x|S)+f[S-\{x\}]\}$. Use of this greedy policy makes performing the minimization operation of the DPFE unnecessary; instead, we need only to find the maximum of a set of probabilities $\{p_x\}$. In the foregoing terms, $D = S$ and $x^* = g(s, D(s)) = \textbf{arg max}_y\{p_y|y \in S\}$ is that member of S having maximum probability.

In addition to linear search, other prominent examples of greedy algorithms that are optimal are discussed in Section 4. An example illustrating the nonoptimality of a greedy policy is that of constructing an optimal binary search tree (BST) by placing the element that has maximum probability at the root of the tree, in a fashion analogous to the one given above for linear search. This greedy policy clearly is not optimal since it may result in a highly unbalanced tree. However, the optimization problem can be solved by DP using a DPFE of the form

$$f[S] = \textbf{opt}_{d\in D(S)}\{c(d|S) + f[S'] + f[S'']\}, \tag{20}$$

where there are two next-states, S' and S'' (Cormen et al., 2001; Horowitz, Sahni, Rajasekaran, 1996). A DPFE is said to be *r-th order* (or "nonserial" if $r > 1$) if there may be r next states. A second-order DPFE is also used below to solve a scheduling (activity-selection) problem.

Regrettably, there are relatively few optimization problems for which a greedy algorithm is known to be an optimal one, i.e., for which a greedy policy $g(s, D(s))$ also optimizes $f[S]$. Ideally, this property should be proven to hold before using a prospective greedy algorithm. In the absence of such a proof (whether because of analytical difficulties or because nonoptimality actually holds), greedy algorithms, also called "heuristic" algorithms instead, are sometimes still used if they are significantly more efficient. Nonoptimal greedy algorithms for solving the traveling salesperson problem (TSP) are examples.

One class of optimization problems for which a greedy policy is provably optimal is associated with matroids (Papadimitriou, Steiglitz, 1982). However, not all greedy algorithms can be so modeled, such as Huffman's algorithm for finding an optimal code. Some other theoretical results appear in Bird, de Moor (1993), Borodin, Nielsen, Rackoff (2002), Curtis (2003), Davis, Impagliazzo (2004), Helman, Moret, Shapiro (1993). This paper views the subject from a very pragmatic point of view. Given a concrete DPFE that solves a specific optimization problem, we seek alternative ways to numerically solve the DPFE. In the next section, we formulate a large class of greedy algorithms as special cases of DP algorithms. This DP formulation is general enough to include all of the aforementioned prominent examples, including Huffman coding.

4. Canonical formulation

In this section, we show how to derive a greedy algorithm from a DP solution to an optimization problem. Given that $f[s] = \mathbf{opt}_{d\in D(s)}\{c(d|s) + f[\delta(s,d)]\}$, with $d^*[s] = \mathbf{arg\ opt}_{d\in D(s)}\{c(d|s) + f[\delta(s,d)]\}$, let

$$\gamma[s, D(s)] = \mathbf{argopt}_{d\in D(s)}\{c(d|s)\}. \tag{21}$$

For the given DPFE, we define the associated *canonical* greedy algorithm as that which adopts the greedy policy of choosing $\gamma[s, D(s)]$ to determine $f[s]$, i.e., so that $f[s] = \{c(\gamma[s, D(s)]|s) + f[\delta(s, \gamma[s, D(s)])]\}$. (A greedy algorithm not based on $\mathbf{opt}_{d\in D(s)}\{c(d|s)\}$ would be *noncanonical.*) If $\gamma[s, D(s)] = d^*[s]$, we say the greedy policy is optimal. Note that $\gamma[s, D(s)]$ optimizes the set of "local" costs $\{c(d|s)\}$ for $d \in D(s)$, where $D(s)$ is the set of eligible decisions when in state s. Borrowing Borodin's terminology (Borodin, Nielsen, Rackoff, 2002), we say that a canonical greedy policy is an *adaptive-priority* one if $c(d|s)$ depends upon both s and d; if $c(d|S)$ is independent of s, but not of d, we say the canonical greedy policy is a *fixed-priority* one. If $c(d|s)$ is independent of d, then there is no canonical greedy policy. There are of course other classification schemes. For example, following Curtis (2003), a canonical greedy algorithm corresponds to one in which the global optimality criterion has an equivalent local optimality criterion.

Example 4.1 (Linear Search) *For the foregoing linear search example, the canonical algorithm would choose (as the first item) x^* to minimize $\{\sum_{y\in S} p_y\}$; however, this sum is independent of x, i.e., the costs are identical for any state S, hence no useful greedy policy results. However, suppose we solve the same linear search problem by DP (see Method S of Section 2) using the alternative DPFE*

$$f[S] = \mathbf{min}_{x\in S}\{c(x|S) + f[S - \{x\}]\}, \quad \textit{for} \quad s \in 2^A, \tag{22}$$

where $c(x|S) = |S|\cdot p_x$. Our goal is to solve for $f[A]$ given the base case $f[\emptyset] = 0$. The canonical greedy algorithm would choose (as the last item!) x^* to minimize *the set $\{|S| \cdot p_x\}$ for $x \in S$, which (since $\mathbf{min}\{k \cdot p_x\} = k \cdot \mathbf{min}\{p_x\}$) requires finding the minimum of the set of probabilities $\{p_x\}$ as before. In the foregoing terms, $D = S$ and $x^* = g(s, D(s)) = \mathbf{arg\ min}_y\{p_y|y \in S\}$. For this DPFE, the corresponding canonical greedy algorithm is optimal, unlike the case for the DPFE of Example 3.1.*

It should be noted that if we choose the *first* item rather than the last, the same DPFE but with $c(x|S) = (N + 1 - |S|) \cdot p_x$ can be used to obtain the optimal solution, but the corresponding canonical greedy algorithm (that chooses the minimum probability item) is not optimal although a noncanonical greedy algorithm (that chooses the maximum probability item) is optimal. This linear search example suggests that canonical greedy algorithms that are optimal might not be very common. On the contrary, many prominent greedy algorithms can be formulated as canonical greedy algorithm, but the formulation process itself may require some effort.

Application: Minimum Spanning Tree (Cormen et al., 2001)

Consider the problem of finding the minimum-weight spanning tree (MST) of a graph with N vertices and having weighted arcs. This problem can be solved by DP using the DPFE

$$f[S] = \mathbf{min}_{x \notin S}\{w(x) + f[S + \{x\}]\}, \tag{23}$$

where S is a set of arcs in the "tree-so-far", and where adding arc x with weight $w(x)$ to the tree does not create a cycle. Our goal is to solve for $f[\emptyset]$ given the base cases $f[S] = 0$ for $|S| = N - 1$. The canonical greedy algorithm that chooses the eligible x^* with minimum weight is optimal.

Example 4.2 (Kruskal) *Kruskal's algorithm is based on the greedy choice of $x \notin S$ that minimizes $\{w(x)\}$, subject to the "eligibility" constraint that the addition of x to the tree does not create a cycle.*

Example 4.3 (Prim) *Prim's algorithm is based on the greedy choice of $x \notin S$ that minimizes $\{w(x)\}$, subject to the "eligibility" constraint that the addition of x to the tree does not create a cycle, but where x is further constrained to arcs incident with a vertex in the tree-so-far.*

Prim's algorithm is the one usually presented in operations research text, although not always by name; Prim is credited in most computer science texts. The resulting MST may not have certain other desirable properties, such as a limit on the degree of each vertex. If vertices represent servers in a computer network, for example, there may be a practical limit on how many vertices may be connected to any single vertex. Kruskal's and Prim's greedy algorithms cannot be used to find the MST satisfying this degree constraint. On the other hand, the (nongreedy) DP formulation still applies; however, it may no longer be efficient even though additional eligibility constraints on p may reduce the sizes of the state and decision spaces.

Application: Huffman Coding (Cormen et al., 2001)

Consider the problem of finding the optimal binary “prefix” code tree for a set D of $N \geqslant 2$ (say) alphabetical data items with given probability weights. This problem can be solved by DP using the DPFE

$$f[S] = \mathbf{min}_{a,b \in S}\{w(a) + w(b) + f[S - \{a, b\} + \{w(a) + w(b)\}]\}. \tag{24}$$

Our goal is to solve for $f[D]$ given the base cases $f[S] = 0$ for $|S| = 1$.

Example 4.4 (Huffman) *Huffman's algorithm, based on the greedy choice of the two members a and b of S that have the smallest weights, so as to minimize $\{w(a)+w(b)\}$, is the corresponding canonical greedy algorithm, which is optimal.*

The resulting Huffman code may not be ordered, in that for $a < b$ the binary code associated with item a may not be less than the binary code associated with item b; hence, we cannot alphabetically sort a Huffman-coded data set in a natural fashion, so as for example to efficiently search it. However, an optimal alphabetically ordered binary code can be found nongreedily using the above DPFE by regarding S as an ordered set and constraining the choice of a and b to be adjacent members of S; the greedy policy of choosing that pair of adjacent members with least sum is not optimal.

We note that the optimal binary prefix code tree has N leaves (corresponding to the N data items) and necessarily $N-1$ internal vertices, hence $2N-1$ vertices in all. Each of the $N-1$ decisions of a pair (a, b) specifies the two *successors* for one of the internal vertices; the final decision specifies the successors of the root. An alternative formulation makes $2N - 2$ decisions, each specifying the single *predecessor* (or *parent*) of one of the vertices (excluding the root, which has no predecessor). The canonical greedy algorithm corresponding to the DPFE for this latter formulation is not optimal, but the equivalent optimal tree results if the vertices are processed greedily in increasing order of their weights.

Application: Shortest Path (Cormen et al., 2001)

Consider the problem of finding the shortest path in a network (connected directed graph) having N vertices, where the graph may be cyclic but must have positive arc distances $d(p, q)$. Suppose we do so by finding the shortest paths from a designated “source” vertex s to each of the other vertices p. For a graph with positive arc distances, these shortest paths will all appear in a single spanning tree rooted at s. The problem of finding this shortest-path spanning tree can be solved by DP using the DPFE

$$f[S] = \mathbf{min}_{p \in S, q \notin S}\{(L(p) + d(p, q)) + f[S \cup \{(q, (L(p) + d(p, q)))\}]\}, \tag{25}$$

where $S = \{(p, L(p))\}$ is the tree-so-far, consisting of a set of vertices and their labels; and the label $L(p)$ is the length of the shortest path to p. (For simplicity, we write $p \in S$ if $(p, L(p)) \in S$.) Our goal is to solve for $f[\{(s,0)\}]$ given the base cases $f[S] = 0$ for $|S| = N$. When the algorithm terminates with all vertices in the tree, the label of each vertex p is the length of the shortest path from s to p, and $f[\{(s,0)\}]$ is the sum of these labels.

EXAMPLE 4.5 (Dijkstra) *Dijkstra's algorithm, based on the greedy choice of the pair $p \in S$ and $q \notin S$ that minimizes $\{(L(p) + d(p,q))\}$, where q is the vertex closest to s not yet in the tree-so-far, is the corresponding canonical greedy algorithm, which is optimal.*

Dijkstra's algorithm also has a "successive approximations" or "relaxation" step, so that whenever the sum $L(p) + d(p,q)$ is found to be less than $L(q)$, vertex q is relabeled such that $L(q) = L(p) + d(p,q)$. However, this successive approximations step is not incorporated in the foregoing DPFE; vertices not in the tree-so-far are not labeled at all until they are added to the tree. The foregoing DPFE does not explicitly incorporate either the greedy policy or the successive approximations part of Dijkstra's algorithm. They may be incorporated in the algorithmic solution of the DPFE. The greedy policy allows us to conclude that the arc (p,q) that minimizes $\{(L(p)+d(p,q))\}$ also minimizes $\{(L(p)+d(p,q))+f[S \cup \{(q,(L(p)+d(p,q)))\}]\}$. Further details are given in the Appendix.

Some variations, such as degree constraints (as for the minimum spanning tree example) or negative arc distances, make the optimization problem unsolvable by Dijkstra's greedy algorithm while it is still solvable by DP. We discuss this further in Section 5.

In addition, it should be noted that Dijkstra's algorithm has also been formulated in Dreyfuss, Law (1977), Sniedovich (2006), in a different fashion using a successive approximations approach characterized by a DPFE of the form

$$F[j] = \mathbf{min}\{F[j], F[k] + d(k,j)\}, j \notin T, \tag{26}$$

or equivalently, if successive states are staged, so that

$$F_i[j] = \mathbf{min}\{F_{i-1}[j], F_{i-1}[k] + d(k,j)\}, j \notin T. \tag{27}$$

Here, T denotes the vertices in the tree-so-far, initially empty. (Note that the latter minimization is over a set of two values, not over a set of values of k.) In this formulation, the DPFE is used to relabel vertices not in T, and greediness is used independently from the DPFE itself to choose $k = \mathbf{arg\ min}_{j \notin T}\{F[j]\}$ that is then added to T. This successive approximations approach is in contrast to our prior formulation, where once $f[S]$ is evaluated its value remains fixed.

Application: Scheduling (Cormen et al., 2001)

We now consider some scheduling problems. Let $\mathbf{P} = \{1, 2, ..., N\}$ be a set of processes to be executed at most one at a time on a single processor, where process i has associated with it a set of parameters $\{a_i, \Delta_i, s_i, t_i, d_i, w_i\}$, some given and others dependent on the schedule to be determined. A *schedule* is the ordered set of processes $S(\subseteq \mathbf{P})$ that are actually executed. For each process i, the *arrival* or *release* time a_i is the time it becomes available for execution, the *duration* or *processing time* Δ_i is the time required for it to actually execute, the *start* time s_i is the time it actually begins execution, the *termination* or *finish* time t_i is the time it ends execution, the *deadline* d_i is the time by which the process is required to end execution (in order to earn a profit or to avoid a penalty), and the *weight* w_i is the "profit" earned if the process is actually executed or equivalently the "penalty" cost incurred if the process is not executed or is late. The *lateness* of a process i is $\mathbf{max}(t_i - d_i, 0)$. If there is no deadline, d_i may be set to ∞. The *duration* of process i is given by $\Delta_i = t_i - s_i$. The *turnaround* time of process i is the time that elapses between when a process is available for execution and when it terminates execution, $t_i - a_i$, which also equals the time the process waits for execution plus its duration. The maximum turnaround time is the schedule *length.* A typical scheduling problem is to determine S so as to minimize or maximize some measure of its goodness based on parameters such as weights (of processes in S), lateness, turnaround times, etc.

EXAMPLE 4.6 (Deadline Scheduling) *Assume all processes in* $\mathbf{P}$ *arrive initially, and all have durations equal to one time unit. The unit-time "deadline-scheduling" problem is that of finding a schedule yielding a maximum total profit (or minimum total penalty); assuming each process can start initially, at time 0 (i.e.,* $a_i = 0$*). For unit-time processes,* $t_i = j$ *if process* i *is the* j*-th process to terminate. A schedule* S *with size* $|S|$ *is feasible if each process* k *in* S *can terminate before its deadline* $d_k \geqslant t_k$*; if* S *is ordered by increasing deadline, then* $d_j \geqslant j$ *for* $1 \leqslant j \leqslant |S|$*. The optimization problem can be solved using the DPFE*

$$f[S] = \mathbf{max}_{k \notin S}\{c(k|S) + f[S + k]\}, \tag{28}$$

where $c(k|S) = w_k$ *if* $S + k$ *is "feasible", else* $c(k|S) = 0$*. Our goal is to solve for* $f[\emptyset]$ *given the base case* $f[\mathbf{P}] = 0$*. The canonical greedy algorithm, which selects the process* k *with maximum weight or profit* w_k *such that* $S + k$ *is feasible, is optimal.*

EXAMPLE 4.7 (Shortest-Processing-Time) *Assume all processes in* $\mathbf{P}$ *arrive initially, and assume no deadlines. Consider the scheduling problem of finding a schedule with minimum total turnaround time; this problem can be solved using the DPFE*

$$f[S, j] = \mathbf{min}_{k \in S}\{c(k, j) + f[S - k, j + \Delta_k]\}, \tag{29}$$

where $c(k, j) = j + \Delta_k$*. Our goal is to solve for* $f[\mathbf{P}, 0]$ *given the base cases* $f[\emptyset, j] = 0$ *for any* j*. Note that* $\mathbf{min}\{j + \Delta_k\} = j + \mathbf{min}\{\Delta_k\}$*. Therefore, the canonical greedy algorithm, which selects that eligible process* k *whose processing* *time or duration Δ_k is minimal, is optimal. Remark: This greedy policy, called "shortest-processing-time" (SPT) or "shortest job-first" (SJF), is analogous to the linear search greedy policy. Thus, this scheduling problem can also be solved using a DPFE of the form given for Example 4.1.*

In addition, a class of scheduling problems, associated with CPM or PERT networks (Cormen et al., 2001; Hillier, Lieberman, 2001), may be formulated as (critical) longest path problems in an acyclic graph having no negative arcs, and hence may be solved using a variation of Dijkstra's algorithm, as described below.

5. Noncanonical algorithms

In the preceding section, we showed numerous examples where the canonical greedy algorithm associated with a DP formulation is optimal, including the "prominent" examples mentioned earlier. On the other hand, an example of a canonical greedy algorithm that is nonoptimal is obvious. The traditional DPFE for finding the shortest path in a directed acyclic graph (DAG) is

$$f[p] = \mathbf{min}_q\{d(p,q) + f[q]\}, \tag{30}$$

but the canonical greedy algorithm based on choosing q^* based on the smallest member of the set $\{d(p,q)\}$ is nonoptimal. However, if the graph is topologically ordered, this same DP formulation results in an efficient noncanonical greedy algorithm since the computations can be "staged". This algorithm, which is optimal, adopts a fixed-priority policy in which the topological ordering (instead of variable vertex labels, as in Dijkstra's algorithm) determines the order in which decisions about which vertex to process next are made. (This approach also applies to the aforementioned CPM/PERT networks.)

For a cyclic graph having a negative arc label, Dijkstra's greedy algorithm is no longer optimal. In fact, the associated DPFE is also nonoptimal because a shortest-path spanning tree no longer exists. However, DP can be used to find the shortest path in a cyclic graph with negative arcs (which may not exist if there are negative cycles) using "successive approximations", employing the Bellman/Ford algorithm (Dreyfuss, Law, 1977), for example. Moreover, DP can be used to find the shortest *simple* path in a cyclic graph with negative arcs (even negative cycles) using a variation of this "successive approximations" approach that uses explicit stages (called the "fixed-time" approach in Lew, 1985), or alternatively using a different DPFE similar to that used for the traveling salesperson problem (TSP) where previously visited in addition to the currently visited vertices must be incorporated in the definition of states. The solution for this alternative formulation, as well as for the TSP problem itself, is inefficient, and the associated canonical greedy algorithm is also nonoptimal.

One advantage of the canonical formulation is that it provides a general way to construct a greedy algorithm in terms that are easily accessible to DP practitioners based upon the DPFEs they develop. (This is in contrast to matroid or relational algebraic models, for example.) The greedy algorithms described in the prior section were all constructed and proven optimal by their original proponents without any explicit connection to DP. We have shown that these greedy algorithms can be viewed as special cases of DP whose DPFEs are not formulated from greedy principles. While we lack a general procedure to determine whether or not a canonical greedy algorithm is optimal, no general procedure exists for noncanonical greedy algorithms either.

Application: Knapsack Problem (Cormen et al., 2001)

The knapsack problem is that of deciding how to fill a knapsack of capacity M with objects from a set $\mathbf{P}$ having n_i objects of type i, each having size or weight W_i and value or cost C_i, so as to maximize the total cost of the objects included, subject to the constraint their total weight "fits", i.e., does not exceed the capacity M; n_i may be fractional or integral. The fractional knapsack problem, for $n_i = 1$ object of each of N types, can be solved by a DPFE of the form

$$f[S,m] = \mathbf{max}_{i\in S}\{c(i,m) + f[S-i, m-w(i,m)]\}, \tag{31}$$

where if $m \geqslant W_i$ then $c(i,m) = C_i$ and $w(i,m) = W_i$, but if $m < W_i$ then $c(i,m) = C_i \cdot m/W_i$ and $w(i,m) = W_i \cdot m/W_i$. Our goal is to solve for $f[\{1,2,\ldots,N\}]$ given the base cases $f[S,m] = 0$ for $S = \emptyset$ or $m < 0$. The canonical greedy algorithm that chooses the object with maximal cost is nonoptimal, but there is an alternate optimal greedy algorithm.

Example 5.1 (Fractional Knapsack Problem) *The optimal solution for the continuous or fractional knapsack problem can be found using a greedy algorithm. Specifically, a greedy policy that is optimal selects the object with the greatest cost-to-weight ratio; the knapsack is then filled with as much of this object as will fit, even fractionally, after which the object with the next greatest cost-to-weight ratio may be selected for inclusion provided it or any fraction of it also fits in the knapsack.*

The optimal solution for the *discrete* or *integer* knapsack problem cannot be found in this sequential fashion since all possible subsets of objects that fit must be considered in order to reduce any remaining capacity. In other words, the greatest cost-to-weight ratio greedy policy will not result in an optimal integer solution. However, the optimal solution can be found nongreedily using a DPFE of the form

$$f[i,m] = \mathbf{max}_{0\leqslant n_i \leqslant N_i}\{c(i,m) + f[i+1, m - n_i \cdot W_i]\}, \tag{32}$$

with $c(i,m) = n_i \cdot C_i$, where the nonnegative integer n_i may be bounded by N_i. Our goal is to solve for $f[1,M]$ given the base cases $f[i,m] = 0$ for $i > N$ or $m < 0$. The 0-1 knapsack problem, which is the special case where each object is either included or not, can be solved by this DPFE, where $N_i = 1$, which reduces to a DPFE of the form

$$f[i, m] = \mathbf{max}\{f[i + 1, m], C_i + f[i + 1, m - W_i]\}. \tag{33}$$

For this DPFE, decisions are Boolean, whether to include an object or not, so there is no associated canonical greedy algorithm.

Application: Scheduling

Assume processes $\mathbf{P} = \{1, 2, ..., N\}$ with parameters $\{a_i, \Delta_i, s_i, t_i, d_i, w_i\}$, as above.

Example 5.2 (Interval Scheduling/Activity Selection) *(Cormen et al., 2001). Assume processes must start upon arrival, or not at all, and assume no deadlines. The "interval scheduling" or "activity-selection" problem is that of finding a schedule $S(\subseteq \mathbf{P})$ of maximum total weight (or number of processes in the case of unit weights), where process i cannot execute before start time s_i nor after termination time t_i; this problem can be solved using the second-order DPFE*

$$f[p, q] = \mathbf{max}_k\{f[p, s_k] + c(k|p, q) + f[t_k, q]\}, \tag{34}$$

where $c(k|p, q) = w_k$ if $p \leqslant s_k$, $t_k \leqslant q$, else $c(k|p, q) = 0$. Our goal is to solve for $f[0, \mathbf{max}t_i]$ given the base cases $f[p, q] = 0$ for $p \geqslant q$. Each decision k (of a process to be included in S) is constrained by the eligibility requirement that the process must start after time p and terminate before time q. A (noncanonical) greedy policy that is optimal selects that eligible process k^* whose termination time t_{k*} is minimal.

It has also been shown that if processes are considered for inclusion or exclusion in S in increasing order of termination time, the optimal solution can be found using a very different DPFE of the form

$$f[k] = \mathbf{max}\{w_k + f[\pi(k)], f[k - 1]\}, \tag{35}$$

where $\pi(k) = \mathbf{max}(j|t_j \leqslant s_k)$. Our goal is to solve for $f[N]$ given the base case $f[0] = 0$. For this DPFE (as for the 0-1 knapsack problem), each decision is Boolean, whether or not to include process $k \in \mathbf{P}$ in S. While the decisions are Boolean, they are made in a greedy order.

Assume now that there may be precedence constraints that require one process i to finish before another process j is "eligible" to start. This can be modeled by a directed graph whose vertices represent processes and where an arc from i to j represents the foregoing precedence constraint. Alternatively, arcs may represent processes (including dummy ones of zero duration), and the foregoing precedence constraint may be represented by introducing vertices to represent the "event" that all predecessor processes have finished hence all successor processes are eligible to start. This is the assumption associated with

CPM/PERT networks that was mentioned above and can be solved as a longest path problem. We note that CPM/PERT assumes that there are no constraints on how many processes may be scheduled simultaneously. If there is such a constraint, a DP solution is still possible.

Example 5.3 (Mininum Schedule Length) *Suppose that at most m processes can execute simultaneously. Assuming all processes in* $\mathbf{P}$ *have unit-time durations for simplicity, a minimum length schedule (also known as the "makespan") can be found using a DPFE of the form*

$$f[S] = \mathbf{max}_{\{S' \subseteq S \mid |S'| \leqslant m \text{ and each } s \text{ in } S' \text{ is eligible}\}} \{1 + f[S - S']\}. \tag{36}$$

Our goal is to solve for $f[\mathbf{P}]$ *given the base case* $f[\emptyset] = 0$. *The cost of a decision does not depend explicitly on the decision, as in Example 3.1, therefore there is no associated canonical greedy algorithm.*

6. Conclusion

In summary, we have identified three classes of DP problems.

- *Class A:* Problems having a DP formulation whose canonical greedy algorithm is optimal. This class includes the linear search problem Examples 3.1 and 4.1 and the other examples in Section 4 (Examples 4.2-4.7).
- *Class B:* Problems having a DP formulation whose canonical greedy algorithm is nonoptimal, but for which an alternate noncanonical greedy algorithm is optimal. This class includes Examples 5.1 and 5.2.
- *Class C:* Problems having a DP formulation whose canonical greedy algorithm is nonoptimal, and which have no known optimal greedy algorithm. This class includes the binary search tree, shortest path in cyclic graph with negative arcs, traveling sales, integer knapsack, and Example 5.3 problems.

Example 5.2 illustrates a DP formulation in which a greedy policy is used to choose the order in which decisions are to be made, rather than what each decision should be. Example 5.1 illustrates a DP formulation in which one greedy policy is used to choose the order in which decisions are to be made, and a second greedy policy is used to choose what each decision should be.

Optimization problems having no DP formulation cannot have an optimal greedy formulation either, since we could always adopt the trivial one in which the set of all eligible decisions is constrained to those that are greedy, relying on the optimality of the greedy policy to justify the constraint. Hence, such problems are not relevant for our purposes. *Remark*: Lew (1985) contains the DPFEs for Example 5.3, as well as Examples 3.1, 4.1, and 4.4, but does not mention relationships between greedy algorithms and DP.

As shown in Section 4, a large class of greedy algorithms can be formulated as special cases of DP algorithms. Thus, DP provides a *constructive* way to derive greedy algorithms in practice. To be practical, an approach must be accessible to practitioners. An approach based on conventional DPFEs meets this requirement, unlike the algebraic approaches of, say, Bird, de Moor (1993), Curtis (2003), at least at the present time. After all, DPFEs are currently being developed in practice on a continuing basis. Furthermore, DP is broadly applicable and permits certain variations of the optimization problems to be solved as well. In Section 5, we showed (for Class B problems) that the canonical greedy algorithm associated with a DPFE need not be optimal, and also showed an optimal greedy algorithm that was not canonically formulated from a DPFE. As illustrated by the two linear search examples, one DPFE may lead to an optimal canonical greedy algorithm, while an alternative DPFE may not. Since deriving a DPFE for an optimization problem is in general still an "art", there being no mechanical process for doing so for arbitrarily given problems, deriving optimal greedy algorithms is likely to remain an art as well. Nevertheless, for a new application, developing a DP formulation initially is a reasonable strategy that may help achieve the ultimate objective of developing an efficient solution.

In conclusion, we have discussed greedy algorithms as a general methodology, but in the context of DP, providing first a constructive means of deriving greedy algorithms to more efficiently solve certain optimization problems, and then a framework in which greedy algorithms can be further analyzed. Much work in these directions remains to be done. We expect that extensions of the ideas reported here will lead to progress on several related theoretical questions, such as general characterizations of optimization problems in the foregoing three classes. One open problem is the development of a method for determining whether any given canonically derived greedy algorithm is optimal. For very special cases, it is possible to use DP to derive a provably optimal greedy algorithm (Sniedovich, 2006); we are investigating ways to generalize this approach to both canonical and noncanonical greedy algorithms, so that, at the least, the Class A algorithms of Section 4 can be proven optimal using DP without appealing to arguments of their originators. Class C problems include intractable (NP-hard) ones, for which there is no "efficient" solution. This does not mean, however, that an optimal greedy algorithm that reduces complexity, say from factorial time to exponential time, does not exist. This may be useful for relatively small problems, but for large problems the use of approximation techniques may be the only alternative.

Acknowledgements

The author thanks Moshe Sniedovich and Allan Borodin for constructive comments on an earlier draft of this paper.

References

BELLMAN, R. (1957) *Dynamic Programming.* Princeton University Press, Princeton.

BIRD, R. and DE MOOR, O. (1993) From dynamic programming to greedy algorithms. *Formal Program Development*, LNCS **755**, 43-61.

BORODIN, A., NIELSEN, M.N. and RACKOFF, C. (2002) (Incremental) priority algorithms. *13th Annual ACM-SIAM Symposium on Discrete Algorithms*, 752-761.

CORMEN, T.H., LEISERSON, C.E., RIVEST, R.L. and STEIN, C. (2001) *Introduction to Algorithms*, 2nd Ed. McGraw-Hill, New York.

CURTIS, S. (2003) The classification of greedy algorithms. *Science of Computer Programming* **49**, 125-157.

DAVIS S. and IMPAGLIAZZO, R. (2004) Models of greedy algorithms for graph problems. *15th Annual ACM-SIAM Symposium on Discrete Algorithms*, 381-390.

DREYFUSS S.E. and LAW, A.M. (1977) *The Art and Theory of Dynamic Programming.* Academic Press, New York.

HELMAN, P., MORET, B.M.E. and SHAPIRO, H. (1993) An exact characterization of greedy structures. *SIAM J. on Discrete Mathematics* **6**, 274-283.

HEYMAN, S.P. and SOBEL, M.J. (1984) *Stochastic Models in Operations Research, Volume II.* Academic Press, New York.

HILLIER F.S. and LIEBERMAN, G.J. (2001) *Introduction to Operations Research*, 7th Ed. McGraw-Hill, New York.

HOROWITZ, E., SAHNI, S. and RAJASEKARAN, S. (1996) *Computer Algorithms/C++.* Computer Science Press, New York.

LEW, A. (1985) *Computer Science: A Mathematical Introduction.* Prentice-Hall International, London.

PAPADIMITRIOU, C.H. and STEIGLITZ, K. (1982) *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs.

SNIEDOVICH, M. (2006) Dijkstra's algorithm revisited: the dynamic programming connexion. *J. Control and Cybernetics*, (this issue).

WINSTON, W.L. (2004) *Operations Research: Applications and Algorithms*, 4th Ed. Brooks/Cole, Belmont.

APPENDIX

To illustrate the concept of canonical greedy algorithms, we reconsider Dijkstra's algorithm (Example 4.5) for solving the problem of finding shortest paths in a network having no negative arc distances. This problem can be solved by constructing, choosing one arc at a time, the shortest-path spanning tree, which gives the shortest path from a designated source vertex s to every other vertex p. The optimal tree can be found by dynamic programming using the DPFE

$$f[S] = \mathbf{min}_{p\in S, q\notin S}\{(L(p)+d(p,q)) + f[S\cup\{(q,(L(p)+d(p,q)))\}]\},$$

where the state S is the "tree-so-far" consisting of the set of chosen arcs. We will denote $S = \{(p, L(p))\}$ by their incident vertices p with associated shortest-path-so-far labels $L(p)$, where $L(p)$ is the length of the shortest path (so far) to p; the arcs in S, although not explicitly shown, are implicitly part of the state. For simplicity, we write $p \in S$ if $(p, L(p)) \in S$. A decision pair or arc (p, q) is not eligible if its addition to the tree-so-far S would create a cycle or if arc distance $d(p, q)$ is infinite. Dijkstra's greedy policy permits us to replace the minimization of $\{(L(p) + d(p, q)) + f[S \cup \{(q, (L(p) + d(p, q)))\}]\}$ by the minimization of $\{(L(p) + d(p, q))\}$.

In addition, Dijkstra's algorithm incorporates a "successive approximations" or "relaxation" step so that whenever the sum $L(p) + d(p, q)$ is found to be less than $L(q)$, vertex q is relabeled, yielding $L(q) = L(p) + d(p, q)$. This requires saving "temporary" values for $L(q)$, rather than recalculating them. Since we only wish to illustrate how greediness is handled, this relaxation step is not included in our formulation.

As a numerical example, we use the network with vertices $\{s, x, y, t\}$ whose adjacency matrix of arc distances is

$$\begin{array}{cccc}
\infty & 3 & 5 & \infty \\
\infty & \infty & 1 & 8 \\
\infty & 2 & \infty & 5 \\
\infty & \infty & \infty & \infty
\end{array}$$

The initial vertex labels are $(s, 0)$, (x, ∞), (y, ∞), and (t, ∞). We interpret ∞ here to mean that the labels of the nonsource vertices x, y, and t are undefined, rather than set to some "infinite" value, since their values are never used until after the vertices are placed in the tree-so-far. Starting in the initial "goal" state $S_0 = \{(s, 0)\}$, the DPFE is

$$\begin{aligned}
f[\{(s,0)\}] &= \mathbf{min}_{p\in\{(s,0)\},q\notin\{(s,0)\}}\{(L(p) + d(p,q)) \\
&\qquad +f[\{s,0)\} \cup \{(q,(L(p) + d(p,q)))\}]\} \\
&= \mathbf{min}_{p\in\{(s,0)\},q\in\{(x,\infty),(y,\infty),(t,\infty\}}\{(L(p) + d(p,q)) \\
&\qquad +f[\{(s,0)\} \cup \{(q,(L(p) + d(p,q)))\}]\} \\
&= \mathbf{min}\{(L(s) + d(s,x)) + f[\{(s,0)\} \cup \{(x,(L(s) + d(s,x)))\}], \\
&\qquad (L(s) + d(s,y)) + f[\{(s,0)\} \cup \{(y,(L(s) + d(s,y)))\}]\} \\
&= \mathbf{min}\{(0+3) + f[\{(s,0)\} \cup \{(x,3)\}], \\
&\qquad (0+5) + f[\{(s,0)\} \cup \{(y,5)\}\}.
\end{aligned}$$

According to Dijkstra's greedy policy, since $\mathbf{min}(0 + 3, 0 + 5) = 3$, the choice $p = s$ & $q = x$ is the optimal decision, which leads to the next state $S_1 = \{(s, 0), (x, 3)\}$. At this point, vertex x is placed in the tree-so-far and labeled with the computed value 3. (The arc (s, x) with arc-distance 3 in the tree-so-far is not explicitly shown.) We continue by evaluating

$$\begin{aligned}
f[\{(s,0),(x,3)\}] &= \mathbf{min}_{p\in\{(s,0),(x,3)\},q\notin\{(s,0),(x,3)\}}\{(L(p)+d(p,q)) \\
&\qquad +f[\{(s,0),(x,3)\}\cup\{(q,(L(p)+d(p,q)))\}]\} \\
&= \mathbf{min}_{p\in\{(s,0),(x,3)\},q\in\{(y,\infty),(t,\infty)\}}\{(L(p)+d(p,q)) \\
&\qquad +f[\{(s,0),(x,3)\}\cup\{(q,(L(p)+d(p,q)))\}]\} \\
&= \mathbf{min}\{(L(s)+d(s,y))+f[\{(s,0),(x,3)\}\cup\{(y,(L(s)+d(s,y)))\}], \\
&\qquad (L(x)+d(x,y))+f[\{(s,0),(x,3)\}\cup\{(y,(L(x)+d(x,y)))\}], \\
&\qquad (L(x)+d(x,t))+f[\{(s,0),(x,3)\}\cup\{(t,(L(x)+d(x,t)))\}]\} \\
&= \mathbf{min}\{(0+5)+f[\{(s,0),(x,3)\}\cup\{(y,5)\}], \\
&\qquad (3+1)+f[\{(s,0),(x,3)\}\cup\{(y,4)\}], \\
&\qquad (3+8)+f[\{(s,0),(x,3)\}\cup\{(t,11)\}]\}.
\end{aligned}$$

According to Dijkstra's greedy policy, since $\mathbf{min}(0+5,3+1,3+8=4$, the choice $p=x$ & $q=y$ is the optimal decision, which leads to the next state $S_2=\{(s,0),(x,3),(y,4)\}$. At this point, vertex y is placed in the tree-so-far and labeled with the computed value 4. (Vertex y is never labeled with the "temporary" value 5, as would be the case if relaxation were used. The arcs (s,x) with arc-distance 3 and (x,y) with arc-distance 1 in the tree-so-far are not explicitly shown.) Hence, we continue by evaluating

$$\begin{aligned}
&f[\{(s,0),(x,3),(y,4)\}] \\
&= \mathbf{min}_{p\in\{(s,0),(x,3),(y,4)\},q\notin\{(s,0),(x,3),(y,4)\}} \\
&\{(L(p)+d(p,q))+f[\{(s,0),(x,3),(y,4)\}\cup\{(q,(L(p)+d(p,q)))\}]\} \\
&= \mathbf{min}_{p\in\{(s,0),(x,3),(y,4)\},q\in\{(t,\infty)\}} \\
&\{(L(p)+d(p,q))+f[\{(s,0),(x,3),(y,4)\}\cup\{(q,(L(p)+d(p,q)))\}]\} \\
&= \mathbf{min}\{(L(x)+d(x,t))+f[\{(s,0),(x,3),(y,4)\}\cup\{(t,(L(x)+d(x,t)))\}], \\
&(L(y)+d(y,t))+f[\{(s,0),(x,3),(y,4)\}\cup\{(t,(L(y)+d(y,t)))\}]\} \\
&= \mathbf{min}\{(3+8)+f[\{(s,0),(x,3),(y,4)\}\cup\{(t,11)\}], \\
&(4+5)+f[\{(s,0),(x,3),(y,4)\}\cup\{(t,9)\}]\}.
\end{aligned}$$

According to Dijkstra's greedy policy, since $\mathbf{min}(3+8,4+5)=9$, the choice $p=y$ & $q=t$ is the optimal decision, which leads to the next state $S_3=\{(s,0),(x,3),(y,4),(t,9)\}$. At this point, vertex t is placed in the tree-so-far and labeled with the computed value 9. (Vertex t is never labeled with the "temporary" value 11, as would be the case if relaxation were used.) The algorithm then uses the base case or termination condition $f[\{(s,0),(x,3),(y,4),(t,9)\}]=0$, where in the final state S_3 the label of each vertex q equals the length of the shortest path from s to q. (The arcs (s,x) with arc-distance 3, (x,y) with arc-distance 1, and (y,t) with arc-distance 5 in the final tree-so-far are not explicitly shown.) Note finally that $f[\{(s,0)\}]=0+3+4+9=16$.

In conclusion, we observe that the DPFE itself does not explicitly incorporate either the greedy policy or the relaxation part of Dijkstra's algorithm. They may be incorporated in the algorithmic solution of the DPFE. The greedy policy allows us to conclude that the arc (p, q) that minimizes $\{(L(p) + d(p, q))\}$ also minimizes $\{(L(p) + d(p, q)) + f[S \cup \{(q, (L(p) + d(p, q)))\}]\}$. The relaxation part would reduce the number of arcs (p, q) that must be considered in the minimization of $\{(L(p) + d(p, q))\}$.