Certain sequencing and scheduling problems are formulated as shortest-route problems and treated in a uniform manner by dynamic programming. Computational considerations are discussed.

The construction of discrete dynamic programming algorithms

by M. Held and R. M. Karp

Many problems of systems engineering can be cast in the following terms: a system may be in any of a finite number of states, and there is a transition cost associated with taking the system from one state to another; the problem is to find a sequence of transitions that takes the system from a given initial state to a desired final state at minimum total cost. In some disciplines, such as control engineering, such formulations are well known; but the same viewpoint, applied to problems of sequencing and scheduling, can be a useful tool for the systems engineer. The purpose of this paper is to present several such problems, describe their formulations from a unified point of view, and tell how some of them were solved on a computer. The examples considered are drawn from the area of operations research, and include such well-known problems as the traveling-salesman problem and the assembly-line balancing problem.

The basic elements of such discrete "control" problems are exhibited by the following shortest-route problem: given a set of cities, with known distances between them, find the shortest path between two given cities. The main theme of the second section is that all of our examples can be reformulated as shortest-route problems; moreover, the dynamic programming algorithm for solving the shortest-route problem yields methods of solution for

the other examples. This unified viewpoint, once understood, may enable the reader to formulate and solve other related problems by dynamic programming.

In the third section, we summarize the techniques of programming some of these dynamic programming methods for the IBM 7094. Since the storage requirements of these programs grow rapidly with problem size, the dynamic programming procedure is not, by itself, adequate for large problems. However, optimum or near-optimum solutions can be found by a successive approximations technique employing dynamic programming at each iteration. A brief description of this technique is also given in the third section.

For certain problems of the type under consideration, alternative solution techniques exist. In the final section, we give examples in which dynamic programming is less efficient than some other approach; the reader, when confronted with his own applications, should keep these examples in mind.

Problem formulations and solutions

a shortest-route problem

Suppose we are given n cities, numbered 1, 2, $\cdots$, n, and a set of positive numbers (a_{ij}), where a_{ij} represents the cost of traveling from city i to city j. We do not require that $a_{ij} = a_{ji}$, and some of the a_{ij} may be infinite (if, for example, it is impossible to go from i to j without passing through some intermediate city). We shall consider the problem of finding a least-cost route from city 1 to city n. Such a route may be the direct one of cost a_{1n}, or may pass through some set of intermediate cities. Since the a_{ij} are positive, it follows that no least-cost route passes through any city more than once. Solutions to this problem are given in the literature;[1-3] we reproduce here a dynamic programming solution given by Bellman.[1]

We assume that there is a route of finite cost (and hence a least-cost route) from city 1 to any city j. Let $C(j)$ denote the cost of a shortest or least-cost route from city 1 to city j; then we assert that the following system of equations holds:

$$\left.\begin{aligned} C(1) &= 0 \\ C(j) &= \min_{i \neq j} [C(i) + a_{ij}]; \qquad j = 2, 3, \cdots, n. \end{aligned}\right\} \tag{1}$$

We argue as follows: any path from 1 to j passes through some city i just before reaching j (it is possible that $i = 1$); thus this path may be broken into two segments: a path from 1 to i having a cost of at least $C(i)$, and a direct link from i to j having cost a_{ij}. The cost of such a route is at least $C(i) + a_{ij}$; and therefore the cost of any route from 1 to j is at least $\min_{i \neq j} [C(i) + a_{ij}]$. On the other hand, if $C(i)+a_{ij}$ is minimized at $i=i^*$, this lower bound is met by the shortest route from 1 to i^*, followed by a link from i^* to j.

The first phase of an algorithm for finding a least-cost route from 1 to n, and its associated cost $C(n)$, is to solve (1) for all the quantities $C(j)$; the solution, which may be shown to be unique,

may be obtained iteratively in the following way: set $C^0(j) = a_{1j}$, and compute the quantities

$$C^k(1) = 0; \qquad k = 1, 2, \cdots$$

$$C^k(j) = \min_{i \neq j} [C^{k-1}(i) + a_{ij}]; \quad j = 2, 3, \cdots, n; \quad k = 1, 2, \cdots .$$

By induction on k, $C^k(j)$ can be shown to represent the minimum cost achievable by any route from 1 to j passing through at most k intermediate cities; thus $C(j) = C^{n-2}(j)$. Once the quantities $C(j)$ have been computed, the second phase of the algorithm determines a least-cost route from 1 to n in the following manner: since $C(n) = \min_{i \neq n}[C(i) + a_{in}]$, there is a value of i different from n, say i^*, such that $C(n) = C(i^*) + a_{i^*n}$. Then i^* is the stop just before n in a least-cost route from 1 to n. Similarly, if $i^* \neq 1$, then since $C(i^*) = \min_{i \neq i^*}[C(i) + a_{ii^*}]$, there is a value of i different from i^*, say i^{**}, such that $C(i^*) = C(i^{**}) + a_{i^{**}i^*}$; also, since $C(n) > C(i^*) > C(i^{**})$, i^{**} is not n, and we may take i^{**} as the stop just before i^*. Continuing in this fashion, an entire least-cost route is generated in reverse order of actual travel.

In cases where there is no path from any city back to itself (except along links of infinite cost), the cities may be renumbered so that a_{ij} is infinite if $j < i$, so that (1) becomes

$$\left.\begin{aligned} C(1) &= 0 \\ C(j) &= \min_{i<j} [C(i) + a_{ij}]. \end{aligned}\right\} \tag{1'}$$

The solution of (1′) can be obtained without iteration simply by computing, in order, $C(2)$, $C(3)$, $\cdots$, $C(n)$. This calculation is analogous to a well-known calculation associated with PERT networks.[4]

Although it is convenient to state the shortest-route problem in terms of routes between cities, the problem may be given a broader interpretation as follows: given any system, or process, which can exist in n distinct states, such that the cost of executing a transition from state i to state j is a_{ij}, find a sequence of transitions taking the system from state 1 to state n at least total cost. This problem is the same as the shortest-route problem as originally stated, except that we speak of states rather than cities. By putting the problem in this more general context, we shall find it easier to interpret a variety of sequencing problems as shortest-route problems.

the traveling-salesman problem

Suppose we are given a set of n cities, and an $n \times n$ matrix (a_{ij}), where a_{ij} represents the cost of traveling form city i to city j. The traveling-salesman problem is that of finding a shortest route or tour for a salesman who must start at city 1 (his home base), visit each of the remaining cities exactly once, and then return to city 1. A closely related variant of this problem is the "open-loop" traveling-salesman problem in which the salesman is not required to return to city 1 at the end of his tour.

Various types of problems may be formulated as traveling-

salesman problems. For example, consider the problem of scheduling the printing of several editions of a publication. If a_{ij} is taken to be the length of time the presses must be down if edition j is printed immediately after edition i, the problem of sequencing the editions so as to minimize total down time is equivalent to an open-loop traveling-salesman problem.

Let us reformulate the traveling-salesman problem as a shortest-route problem of the kind previously considered. In order to do so we must have a means of describing the state of a partially completed tour at any point. This description of state must satisfy the criterion that it be possible to determine the pairs of states between which transitions are directly possible, and to determine the cost of each such direct transition. Let us attempt to describe the state of a partially completed tour by specifying S, the set of cities that have been visited (excluding city 1), and f the city visited last, where f is an element of S. A state, then, will be denoted by the ordered pair (S, f); in addition, let A denote the initial state of being at city 1 before the tour begins, and let Ω denote the state reached at the end of the tour, when city 1 is revisited. If S consists of the single element f, then (S, f) is accessible from the state A at cost a_{1f} and is not accessible from any other state. Otherwise the states from which (S, f) is accessible by a direct transition (corresponding to a journey to city f from some other city in S) are those of the form $(S - f, m)$, where $S - f$ is the set obtained by deleting element f from S, and m is any element of $S - f$; the cost of such a transition is a_{mf}. For example, the possible transitions into the state $(\{3, 4, 6, 7\}, 6)$ are shown schematically as follows:

$$\left.\begin{array}{ll} (\{3,4,7\},3) & \overset{a_{36}}{\longrightarrow} \\ (\{3,4,7\},4) & \overset{a_{46}}{\longrightarrow} \\ (\{3,4,7\},7) & \overset{a_{76}}{\longrightarrow} \end{array}\right\} \quad (\{3,4,6,7\},6).$$

Finally, Ω is accessible from any state of the form $(\{2, 3, \cdots, n\}, f)$, with an associated cost a_{f1}. Thus our definition of state meets the requirements, and solving the traveling-salesman problem is equivalent to finding a shortest route from A to Ω.

Adapting the system of equations (1) to the present situation, we obtain:

$$\left.\begin{aligned} &C(A) = 0 \\ &C(\{f\}, f) = a_{1f} \quad \text{for} \quad f = 2, 3, \cdots, n \\ &C(S, f) = \min_{m \in S-f} [C(S - f, m) + a_{mf}], \\ &\qquad \text{for } S \neq \{f\}, \quad S \subseteq \{2, 3, \cdots, n\} \text{ and } f \in S \\ &C(\Omega) = \min_{f \in \{2,3,\cdots,n\}} [C(\{2, 3, \cdots, n\}, f) + a_{f1}]. \end{aligned}\right\} \tag{2}$$

These recurrence relations (which are also given in Reference 5) may perhaps be clarified by noting that $C(S, f)$ represents the

minimum cost of starting at city 1 and visiting all cities in the subset S, terminating at city f, where f is an element of S. By visiting city m just before f, a total cost of $C(S - f, m) + a_{mf}$ can be achieved. The recurrence relation expresses the fact that the best possible result is achieved by minimizing over all choices of m. It is easily seen that there is no path from any state back to itself, so that (2) can be solved without any iteration.

Two types of errors are possible in defining the states of a process. The first type of error yields a definition that is insufficient; for example, specifying the state of a partially completed tour by giving only f, the last city visited, is unacceptable because we do not know whether state m, for example, is accessible from f without knowing whether city m was previously visited in the partial tour leading to f. If the state associated with a partial tour is given simply by S, the set of cities visited, we know that S is directly accessible from every state of the form $S - f$, but we cannot determine the cost of this transition. The second type of error consists of giving superfluous information. Suppose the state associated with a partial tour is specified as the ordered set of cities visited during the partial tour. This specification certainly permits the transitions and their costs to be determined, but fails to combine into single states certain sets of partial tours that could be so combined. For example, the partial tours specified by the ordered sets (1, 2, 3, 4), (1, 3, 2, 4) both correspond to the pair (S, f) where $S = \{2, 3, 4\}$ and $f = 4$, and, therefore, our original definition subsumes them under a single state. If we fail to make full use of such equivalences, the number of states will become unnecessarily large, and the computation, unnecessarily cumbersome. It is possible to give a fully rigorous exposition of the concept of "state" for discrete decision processes, but we shall not do so here.[6]

The recurrence relations (2) may be used to solve traveling-salesman problems by first tabulating the quantities $C(S, f)$, and then constructing an optimum tour in a manner analogous to the second phase of the shortest-route algorithm.

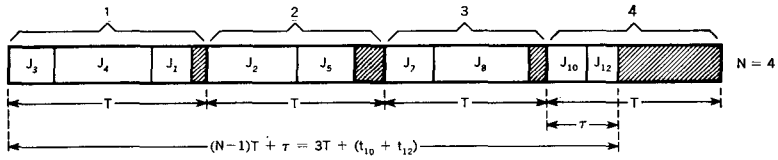
an assembly-line balancing problem

An assembly line is required to produce a unit of product every T units of time (T is called the *cycle time* of the line). For each unit produced, a set of elementary jobs $J_1, J_2, \cdots, J_n$ must be performed. A given job J_k may be executed in t_f units of time, and may be assigned to any of the work stations placed serially (and numbered 1, 2, 3, $\cdots$) along the assembly line. This assignment is to be made so that:

- each job must be assigned to exactly one work station;
- the sum of the execution times of the jobs assigned to any given work station does not exceed the cycle time T;
- if there is a technological requirement that J_i must precede J_j (denoted $J_i \prec J_j$), then J_i is assigned to the same work station as J_j, or to an earlier one;
- the number of work stations is minimized.

Let us view the process of assigning jobs to work stations as a

Figure 1 Geometric interpretation of a feasible assignment



dynamic one, in which one job is assigned at a time. The state of a partially completed assignment can be described by giving the set S of jobs that have been assigned. Not every set S may represent a valid state, however; for example, if $J_1 \prec J_2$, then any set S containing J_2, but not J_1, corresponds to an invalid state. In general, S is said to be a *feasible* set if, whenever it contains a job J_k, it also contains all the jobs required to precede J_k. Thus valid states correspond to feasible sets.

Now, to show that our definition of state is satisfactory, we must define the allowable transitions between states, and their costs. The state S is accessible from the states $S - f$, where f is an element of S, and the set $S - f$, obtained by deleting f from S, is feasible. The definition of transition costs is somewhat more complicated in this case than it was previously. There are usually many ways in which a state S can be reached, corresponding to the possible ways of assigning the jobs in the set S without violating precedence restrictions; any such assignment will be called *feasible*. The "cost" of a feasible assignment is defined to be $(N-1)T + \tau$, where N is the highest numbered work station to which jobs are assigned, and τ is the sum of the execution times of the jobs assigned to the Nth station. Figure 1 gives a geometric interpretation of a feasible assignment and its cost.

The minimum cost, $C(S)$, of reaching state S is defined as the cost of the "cheapest" feasible assignment of the jobs in S. The cost, $\Delta(S - f, S)$, of a transition from state $S - f$ to state S is then the incremental cost of adjoining job J_f to the best feasible assignment for $S-f$. If $C(S-f)$ is $(N-1)T+\tau$, where $0 < \tau \leq T$, then $\Delta(S - f, S)$ is defined as follows:

Case (a): if $T - \tau \geq t_f$, so that J_f can be assigned to work station N, then $\Delta(S - f, S) = t_f$;

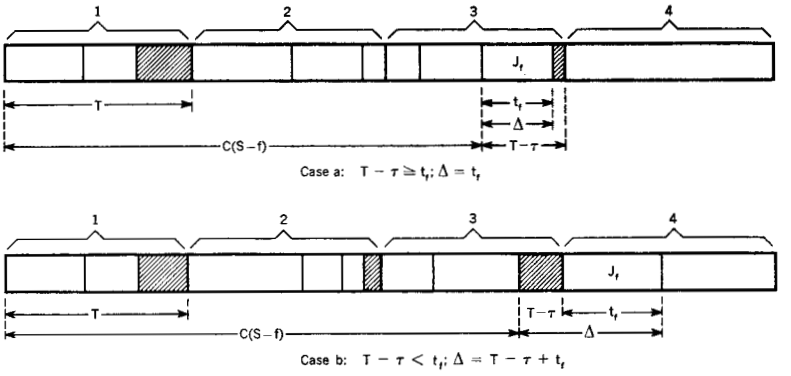
Case (b): if $T - \tau < t_f$, so that J_f must be assigned to work station $N + 1$, with the "idle time" at station N wasted, then $\Delta(S - f, S) = T - \tau + t_f$.

These cases are illustrated in Figure 2.

Thus we have the curious situation that $\Delta(S - f, S)$ cannot be determined until $C(S - f)$ is known. Nevertheless, the problem may be viewed as a shortest-route problem in which it is required to go, via feasible sets, from the empty set Φ to the set $\{1, 2, \cdots, n\}$. From(1) we obtain the following recurrence equations (which are also given in Reference 7):

$$\left.\begin{aligned} C(\Phi) &= 0 \\ C(S) &= \min_{\substack{f \in S \\ S-f \text{ feasible}}} [C(S - f) + \Delta(S - f, S)], \text{ for } S \text{ feasible} \end{aligned}\right\} \quad (3)$$

Figure 2 Two illustrative cases



As in the previous examples, a two-phase calculation is used: first $C(S)$ is tabulated, and then an optimal assignment is obtained by backtracking through the tabulated values.

a scheduling problem

As another example, we consider the problem of scheduling a set of jobs $J_1, J_2, \cdots, J_n$ which are to be executed successively on a single facility. Any given job J_k is assumed to require the services of the facility for τ_k units of time. For example, the facility might be a digital computer, and each job J_k a program with estimated running time τ_k. With J_k is also associated a function $c_k(t)$, giving the cost associated with completing J_k at time t. We assume that the facility is to be constantly in use, and that no job is to be interrupted before completion. There is no advantage in violating these assumptions when the functions $c_k(t)$ are monotone nondecreasing, representing penalties incurred for deferring the completion of the jobs. With these assumptions, any given sequence of execution of the jobs (a *schedule*) may be represented by an ordering $(i_1, i_2, \cdots, i_n)$ of the integers from 1 through n, indicating that the jobs are to be executed in the order $J_{i_1}, J_{i_2}, \cdots, J_{i_n}$. Given such a schedule, the termination time t_{i_k} of J_{i_k} is $\sum_{j=1}^{k} \tau_{i_j}$, and the total cost associated with the schedule is $\sum_{k=1}^{n} c_{i_k}(t_{i_k})$. We seek an ordering for which this total cost assumes its minimum value.

Let us view the process of executing jobs as a dynamic one. The state of a partially completed schedule is described by giving the set S of jobs that have been executed, and any state S is accessible from all states of the form $S - f$. The cost of a transition from state $S - f$ to state S is then the incremental cost incurred by executing job J_f last within the set S, and is given by $c_f(t_S)$, where $t_S = \sum_{i \in S} \tau_i$. Thus the problem may be viewed as a shortest-route problem in which it is required to go from the state Φ to the state $\{1, 2, \cdots, n\}$. Using (1) we obtain the recurrence relations (which are also given in Reference 5):

$$\left.\begin{aligned} C(\Phi) &= 0 \\ C(S) &= \min_{f \in S} [C(S - f) + c_f(t_S)]. \end{aligned}\right\} \tag{4}$$

As before, a two-phase calculation is employed to compute an optimum schedule.

Computational considerations and successive approximations

computational considerations

There are basic similarities among the implementations on a digital computer of the traveling-salesman, assembly-line balancing, and scheduling algorithms developed above. In each of these cases, all or part of the state description specifies a subset of a finite set of objects. A computer representation of such a subset can be obtained by setting up a correspondence between the elements of the set of objects and the bits of a field in a computer word. If an element is present in a subset, a 1 occupies the corresponding bit position; otherwise a 0 occupies that position. Thus the subsets are represented by integers in the binary representation.

The tabulation of the cost functions $C(S)$ in equations (3) and (4) and $C(S, f)$ in equations (2) must be ordered so that the evaluation of the left-hand side of the recurrence equations (which requires the previous evaluation of the quantities on the right-hand side) is possible. A simple procedure is to consider the subsets in the natural order of their binary integer representations, and compute the associated cost functions in this same order,

Finally, in each case, for the purpose of storage allocation, it is necessary to establish a simple correspondence between the quantities being tabulated and consecutive memory locations within the computer. In the case of the scheduling problem, this correspondence may be obtained by simply using the binary integer representation for a subset as the relative address of the cost function associated with it. For the traveling-salesman problem, $C(S, f)$ is assigned a location obtained by adding a base address associated with f to a relative address derived from the binary integer representation for S by "squeezing" out the 1 in bit position f and moving all bits to the left of it one position to the right.

The storage allocation problem for the line balancing algorithm is complicated by the fact that only the feasible subsets are considered. In the 7094 program, the problem is solved by storing a list of the feasible sets S in the natural order of their binary representations. The quantities $C(S)$ are stored in a second list, with S and $C(S)$ occupying the same positions relative to the initial addresses of the two lists. The location of $C(S)$ in the second list is determined by performing a binary search to locate S in the first list. The nature of the dynamic programming algorithm permits the calculation to be arranged so that sharp initial bounds are set on the range of each binary search performed. Although this simple approach is wasteful of storage, it yields an efficient program in terms of operating speed. It is possible to develop an alternative method which does not require the storage of two lists.[7] This procedure was not used in the 7094 program, however, because of its complex list processing structure.

successive approximations

The dynamic programming algorithms presented in the previous section are a vast improvement over complete enumeration, and permit the convenient solution of problems of moderate size. However, the complexity of these algorithms, as measured by numbers of arithmetic operations and storage requirements, grows quite rapidly. In the case of the traveling-salesman problem, for example, the number of storage locations required to tabulate the quantities $C(S, f)$ is

$$\sum_{k=1}^{n-1} k\binom{n-1}{k} = (n-1)2^{n-2}.$$

Thus, the memory capacity of a computer with $32K$ words of core storage (such as the 7094), is sufficient to permit the solution of traveling-salesman problems involving up to thirteen cities. Although it is possible to solve larger problems using auxiliary storage such as tapes or discs, only slight gains can be realized without introducing undue complexity into the program. In addition, the number of evaluations of $C(S - f, m) + a_{mf}$ in equations (2) is

$$\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k} = (n-1)(n-2)2^{n-3},$$

and, for large problems, the magnitude of this number of operations becomes quite considerable; for example, if $n = 20$, we have $19 \cdot 18 \cdot 2^{17} = 44, 826, 624$. Because of these considerations, the solution of large problems requires that the dynamic programming algorithms be supplemented by some other procedure. One such procedure is a method of successive approximations. This technique yields successively improved solutions; each solution is obtained from its predecessor by the solution of a derived subproblem of moderate size having the same structure as the given problem. The associated costs form a monotone nonincreasing sequence, and, although there is no guarantee that an optimum solution will eventually be reached, the method has been quite successful in practice.

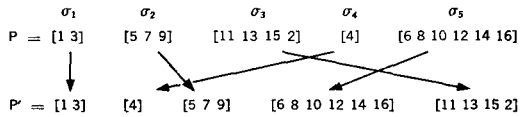
We shall use the traveling-salesman problem to illustrate this method of successive approximations. Any given traveling-salesman tour may be represented by a cyclic permutation $P = (1\, i_2\, i_3 \cdots i_{n-1}\, i_n)$ indicating that the salesman starts at city 1, proceeds to city i_2, then to city i_3, $\cdots$, then to city i_{n-1}, then to city i_n, finally returning to city 1 from city i_n. Suppose we decompose this tour in the following manner:

$$\begin{aligned} P &= (1 i_2 i_3 \cdots i_{n-1} i_n) \\ &= (1 \cdots i_{n_1})(i_{n_1+1} \cdots i_{n_2}) \cdots (i_{n_{u-1}+1} \cdots i_n) = \sigma_1 \sigma_2 \cdots \sigma_u. \end{aligned}$$

We may now define a u-city traveling-salesman problem in which each σ_k is treated as a city, and the cost of going from the "city" $(i_m\, i_{m+1} \cdots i_f)$ to the "city" $(i_p\, i_{p+1} \cdots i_q)$ is $a_{i_f i_p}$. If u is not too large, this derived problem may be solved by the dynamic pro-

gramming algorithm given above for the traveling-salesman problem. The solution implies a reordering P' of P, with traveling-salesman tour represented by P' having cost less than or equal to that represented by P. An example of a decomposition and possible reordering is shown in Figure 3.

Figure 3 An example of decomposition and possible reordering



The method of successive approximations proceeds by solving a sequence of derived problems, each of which is of the type just introduced. In order to completely specify the procedure, it is necessary to determine which derived problems are to be solved and when to stop the computation. A systematic strategy, which was suggested by computational experience, has been worked out. The details are given in References 5 and 9.

The authors have developed similar successive approximation methods for the scheduling and assembly-line balancing problems; these are discussed in References 7, 8, and 10. In all cases, the procedures have led to highly satisfactory results; traveling-salesman problems with up to 50 cities and assembly-line balancing problems with up to 612 jobs have been successfully treated. Summaries of the computational results obtained may be found in References 5, 7, 8, 9, and 10.

Other approaches

As we have seen, dynamic programming is a powerful tool for the solution of sequencing and scheduling problems arising in operations research. However, in many cases, alternative, and perhaps more effective, solution techniques exist. For example, in the case of the single-facility scheduling problem, if each cost function $c_k(t)$ is linear, i.e., $c_k(t) = a_k t$ for some $a_k \geq 0$, then as McNaughton[11] has shown, the jobs should be ordered according to the ratios a_k/τ_k, with the job having the highest ratio being performed first. This simple decision rule is obviously more appropriate than dynamic programming for this special case. As another example, the problem of cutting specified lengths $t_1, t_2, \cdots, t_n$ of stock from a minimum number of standard reels of length T has exactly the same mathematical structure as the assembly-line balancing problem without precedent restrictions. Thus this cutting-stock problem may be treated by the recurrence relations (3), with all sets S taken as feasible. In this case, the dynamic programming approach is probably useful for small problems; for the large problems which usually occur in practice, the column-generating technique of Gilmore and Gomory[12,13] makes linear programming a more efficient means of solution.

As another example, consider a simple scheduling problem in which n jobs $J_1, J_2, \cdots, J_n$ are to be processed on two machines A and B. Any given job J_k is assumed to require the services of machine A for a_k units of time and of machine B for b_k units of time. It is also assumed that each job must first be processed by machine A and then by machine B. It is required to find a sequence of processing the jobs which will minimize the total time, T, needed for completion of all jobs on both machines. It can be shown that to obtain an optimal sequence it is sufficient to consider that the jobs are processed in the same order through both machines, and consequently it is relatively simple to give a dynamic programming algorithm for the solution of this problem. Viewing the processing of jobs as a dynamic one, the state of a partially completed sequence is described by giving the set S of jobs that have been processed; a state S is accessible from all states of the form $S - f$. To obtain the transition costs in this case, we first define $I(S)$ as the total "idle" time on machine B after processing the jobs in S, and note that

$$T = I(\{J_1, J_2, \cdots, J_n\}) + \sum_{j=1}^{n} b_j;$$

it follows that we may take the idle time on machine B as our measure of the efficiency of a sequence of jobs, and that it is this quantity that we wish to minimize. The cost $\Delta(S - f, S)$, of a transition from state $S - f$ to state S may then be taken as the incremental idle time incurred by adjoining job J_f to the best ordering for $S - f$. This incremental idle time is given by $\max [0, \sum_{i \in S} a_i - \sum_{i \in S-f} b_i - I(S-f)]$. Thus the problem may be viewed as a shortest-route problem in which it is required to go from the state corresponding to the empty set Φ to the state $\{1, 2, \cdots, n\}$. Using (1), we obtain the recurrence relations

$$\left.\begin{aligned} I(\Phi) &= 0 \\ I(S) &= \min_{f \in S} [I(S - f) + \Delta(S - f, f)] \end{aligned}\right\} \tag{5}$$

and, as in the previous examples, a two-phase calculation may be employed to compute an optimum sequence.

While the dynamic programming solution to the two-machine problem is correct in principle, it is much more efficient to solve the problem by applying a simple decision rule due to S. Johnson[14] which states that an optimal ordering is determined as follows: J_r precedes J_s if $\min [a_r, b_s] < \min [a_s, b_r]$.

Summary

Thus, while dynamic programming, particularly when combined with the method of successive approximations, is a powerful and flexible weapon for attacking operations research type problems and belongs in the arsenal of every systems engineer, the other available techniques of linear programming, simulation, and simple decision rules, should be considered in each case.

CITED REFERENCES AND FOOTNOTE

1. R. Bellman, "On a routing problem," *Quart. Appl. Math.* **16**, 87–90 (1958).
2. G. B. Dantzig, "On the shortest route through a network," *Management Science* **6**, 187–190 (1960).
3. M. Pollack and W. Wiebenson, "Solutions of the shortest route problem—a review," *Operations Research* **8**, 224–230 (1960).
4. J. E. Kelley, Jr., "Critical-path planning and scheduling: mathematical basis," *Operations Research* **9**, 296–320 (1961).
5. M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," *J. Soc. Indust. Appl. Math.* **10**, 196–210 (1962).
6. In a forthcoming paper, the authors will discuss this problem.
7. M. Held, R. M. Karp, and R. Shareshian, "Assembly-line balancing—dynamic programming with precedence constraints," *Operations Research* **11**, 442–459 (1963).
8. *IBM 7090/7094 Assembly-Line Balancing Program, Reference Manual* (7090-MF-02X), International Business Machines Corporation.
9. *The Traveling Salesman Problem: An Application of Dynamic Programming, Reference Manual* (7090-CO-05X) (1963), International Business Machines Corporation.
10. M. Held, R. M. Karp, and R. Shareshian, "Scheduling with arbitrary profit functions," SHARE Distribution SD3223-IBAPF, Program Information Department, International Business Machines Corporation.
11. R. McNaughton, "Scheduling with deadline and loss functions," *Management Science* **6**, 1–12 (1959).
12. P. C. Gilmore, and R. E. Gomory, "A linear programming approach to the cutting stock problem," *Operations Research* **9**, 849–859 (1961).
13. ———, "A linear programming approach to the cutting stock problem—Part II," *Operations Research* **11**, 863–888 (1963).
14. S. M. Johnson, "Optimal two-and-three stage production schedules with setup times included," *Nav. Res. Log. Quart.* **1**, 61–68 (1954).