NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-79-106

Dynamic Programming in Computer Science

Kevin Q. Brown

February 1979

University Libraries
Carnegie Mellon University
Pittsburgh PA 15213-3890

DEPARTMENT
of
COMPUTER SCIENCE

510.7808
C28R
79-106

Carnegie-Mellon University

Table of Contents

1. Introduction

2. An Example

3. Brief Summary of the Theory

4. Single Source Shortest Path Problems

5. Optimal Parenthesization Problems

   5.1 Optimizing the Multiplication of N Matrices
   5.2 Context Free Language Recognition
   5.3 Optimal Alphabetic Encoding and Optimal Search Trees
   5.4 Optimizing the Search for N Order Statistics

6. Optimal Partition Problems

7. Optimal Matching Problems

8. 'Hard' Combinatorial Problems

9. Conclusion

References

1. Introduction

> "Dynamic programming is an important technique for the solution of problems involving the optimization of a sequence of decisions. The simple idea underlying this technique is to represent the problem by a process which evolves from state to state in response to decisions. A typical optimization problem then becomes one of guiding the system to a terminal state at minimum cost. When the cost structure is appropriate, the determination of an optimal policy (sequence of decisions) may be reduced to the solution of a functional equation in which the state appears as an independent variable." Karp and Held, 1967.

Dynamic programming (DP) is one of several *problem solving methods* used in computer science (and operations research). One of the distinguishing features of a DP algorithm is the way it decomposes a problem into subproblems. A problem of size N (that is, N *stages*) typically decomposes to several problems of size N - 1, each of which decomposes to several problems of size N - 2, etc. To avoid recomputation of common subproblems, a DP algorithm typically records the solutions to all subproblems it encounters. Thus, at the expense of extra storage, a DP algorithm can reduce the time required to solve the original problem. DP is most appropriately applied to problems for which this table of subproblem solutions helps eliminate a great number of redundant computations. As the quote at the beginning of this chapter suggests, DP is applicable primarily to the types of problems that can be expressed as a sequence of decisions to be made at each of several stages.

For other types of problems there are corresponding problem solving methods. These methods include divide-and-conquer, linear and integer programming, and branch-and-bound. Divide-and-conquer is applicable to problems that can be divided into subproblems of (approximately) equal size, the results of which can be efficiently used to solve the original problem [2]. For example, a divide-and-conquer decomposition of a problem of size N may produce two subproblems of size N/2. Linear and integer programming are applicable to problems with linear optimization functions and linear constraints [11]. (Gilmore [23] describes some interesting interrelationships between linear and integer programming and dynamic programming.) Branch-and-bound is commonly used for pruning of large tree searches, and is also closely related to DP [39, 49].

Dynamic programming originated with the work of Bellman [5] and has been applied to problems in operations research, economics, control theory, computer science, and several other areas. The theory has evolved in several directions: discrete vs. continuous states, finite vs. infinite (countable and uncountable) number of states, and deterministic vs. stochastic systems. Not surprisingly, the literature on dynamic programming is enormous. The survey paper by Thomas [69] is a good overview of the several branches of DP. The orientation of this paper is toward the uses of DP in computer science, in particular combinatorial problems. The DP algorithms for these problems are typically discrete, finite,

and deterministic. Tarjan's [68] recent survey article on combinatorial algorithms includes a brief description of the applications of DP to computer science.

This paper is intended to be a reference to the uses of dynamic programming in computer science. Many problems are covered, along with the DP algorithms and references to related problems and more advanced results. We introduce dynamic programming in the form of an example in Chapter 2 below, and in Chapter 3 present some of the theory of DP and subsequently apply it to the problem of Chapter 2. In the following four chapters we present and analyze examples from several subclasses of DP problems. These subclasses are (for the most part) defined by the form of the recursion for breaking a problem into subproblems. Chapter 4 covers Single Source Shortest Path Problems, Chapter 5 describes Optimal Parenthesization Problems, Optimal Partition Problems are analyzed in Chapter 6, in Chapter 7 we present Optimal Matching Problems, and 'Hard' Combinatorial Problems are the topic of Chapter 8. Each of these chapters are independent and can be read separately from the others. (It is suggested, however, that Chapters 2 and 3 be read before attacking any of the material in the later chapters.)

2. An Example

Before diving into the theory, it will be helpful to give some of the flavor of DP algorithms by giving an example. The diagram in Figure 2-1 is a graph with (non-negative) weighted edges. These weights are interpreted as the "distances" along the edges connecting the nodes. Thus, the distance from node (1,1) to node (2,1) is 3 and the distance from node (2,1) to node (3,2) is 2. (The distance from node (2,1) to node (1,1), on the other hand, is not defined in the graph and can thus be assumed to be infinite.) The distance along path (1,1) - (2,1) - (3,2) is 3 + 2 = 5. The problem is to find the shortest path from node (0,1) to node (4, 1).

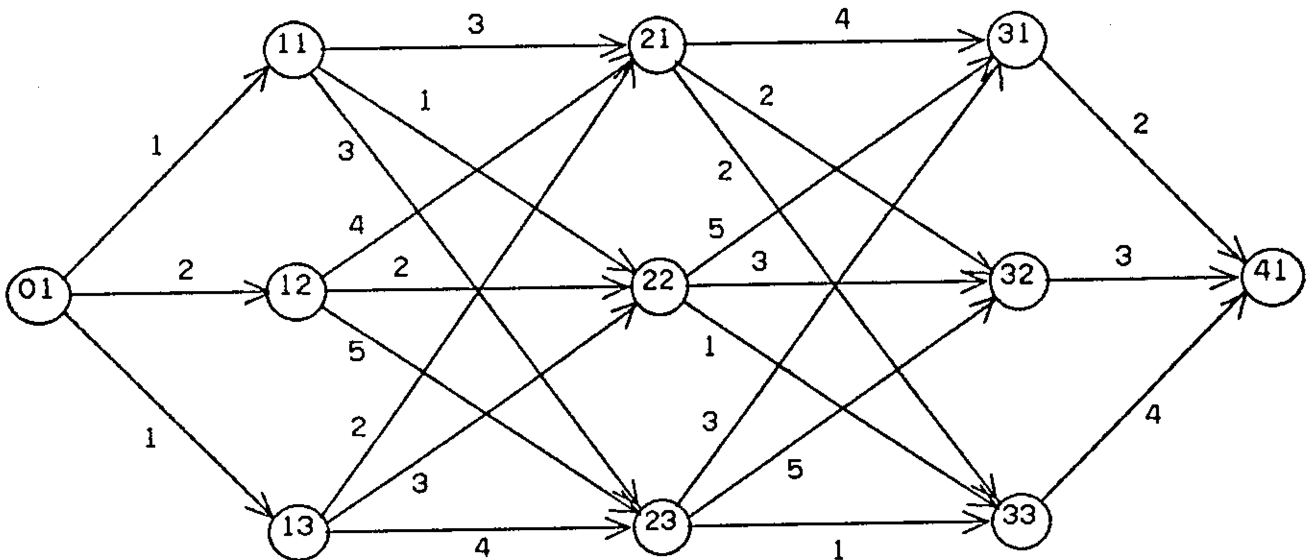


Figure 2-1: Find the shortest path from Node (0,1) to Node (4,1)

There are several approaches that we can take to solve this problem. One approach is to enumerate *all* of the paths from node (0,1) to node (4,1) and choose the one with minimum distance. For the graph of Figure 2-1 there are $3^3 = 27$ different paths from node (0,1) to node (4,1). Examing all these paths is somewhat tedious, but definitely manageable. If, however, instead of 3 columns of 3 nodes there were K columns of L nodes, then there would be L^K paths to examine! Thus, the number of paths grows very rapidly with L and K and for large problems of this type another method will have to be used.

A dynamic programming approach turns out to be just what we need. Let

$$C_{ijk} = \text{the (nonnegative) "distance" from node (i,j) to node (i+1,k)}$$

and let

$$M_{ij} = \text{length of the shortest path from node } (0,1) \text{ to node } (i,j).$$

The optimization for the problem in Figure 2-1 proceeds in four *stages*. In each stage i we find the lengths of the shortest paths from node (0,1) to the nodes (i,j), for all j, which we denote by M_{ij}. Stage 1 is easy since there is only one path from node (0,1) to each of the nodes (1,1), (1,2), and (1,3). Thus, we have

$$M_{11} = C_{011}, M_{12} = C_{012}, M_{13} = C_{013}.$$

Stage 2, however, requires some computation since there are three paths from node (0,1) to each of the nodes (2,1), (2,2), and (2,3). Thus,

$$M_{21} = \min(C_{011} + C_{111}, C_{012} + C_{121}, C_{013} + C_{131})$$
$$M_{22} = \min(C_{011} + C_{112}, C_{012} + C_{122}, C_{013} + C_{132})$$
$$M_{23} = \min(C_{011} + C_{113}, C_{012} + C_{123}, C_{013} + C_{133})$$

In stage 3 we find the lengths of the shortest paths from node (0,1) to nodes (3,1), (3,2), and (3,3). Here is where the great advantage of dynamic programming becomes apparent. It is not necessary to examine all paths from node (0,1) to nodes (3,1), (3,2), and (3,3) because we can use the results from stage 2 as follows:

$$M_{31} = \min(M_{21} + C_{211}, M_{22} + C_{221}, M_{23} + C_{231})$$
$$M_{32} = \min(M_{21} + C_{212}, M_{22} + C_{222}, M_{23} + C_{232})$$
$$M_{33} = \min(M_{21} + C_{213}, M_{22} + C_{223}, M_{23} + C_{233})$$

Similarly, in stage 4 we can use the results from stage 3.

$$M_{41} = \min(M_{31} + C_{311}, M_{32} + C_{321}, M_{33} + C_{331})$$

The solution for Figure 2-1 is path (0,1) - (1,1) - (2,2) - (3,3) - (4,1) of length 7. Note that it is straightforward to determine that the length $M_{41} = 7$, but recovering the corresponding path (or paths) is a slightly different problem. It is, however, simply solved by simply recording a pointer P_{ij} with each distance M_{ij} which points back to the node (i-1,k) from which the shortest path to node (i,j) came.

Why can the results of stage 2 be used to solve stage 3, and the results of stage 3 be used to solve stage 4? This is because of the *Markov property* of this type of problem. The optimum path from a node (i,j) to node (4,1) does not depend on the path taken from node (0, 1) to node (i,j). This property can be expressed as Bellman's Principle of Optimality:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." Bellman[57], p.83.

We can use this principle for the general case of K columns of L nodes. The dynamic programming approach will take only $O(K*L^2)$ time, a significant improvement over the exhaustive approach of examining all L^K paths. Both of these methods require $O(K*L^2)$ storage to store the distances (C_{ijk}). The general formulas describing the optimization for this type of problem, known as the *functional equations* for this problem, are

$$M_{ij} = \min_{k} (M_{(i-1),k} + C_{(i-1),k,j})$$

where $M_{0,k} = 0$, for all k. In the next chapter we show how to derive this equation.

3. Brief Summary of the Theory

Bellman's Principle of Optimality, as shown in the example of Chapter 2, is an important and powerful principle for dynamic programming. It is, however, lacking in at least two ways: (1) It is not stated very rigorously, and (2) the problem of obtaining the functional equations is not even addressed. To obtain the functional equations, we must rely on methods beyond the scope of the Principle of Optimality.

By the mid 1960's several people had solved the first problem by showing that the Principle of Optimality derives from a monotonicity property of the cost functions. (See Karp and Held [44] for references.) Karp and Held subsequently made considerable progress on the second problem. In their formulation, a combinatorial decision problem is assumed to be expressed in the form of a *discrete decision process* (DDP). This form is usually a fairly natural way to express the problem. The dynamic programming algorithm to solve the problem, on the other hand, corresponds to a *sequential decision process* (SDP). Karp and Held describe the transform (if it exists) from a discrete decision process to a sequential decision process.

The result of Karp and Held can be used to derive the functional equations for the problem of Figure 2-1. Their result, however, will not be followed exactly because it is long and complicated to explain (and thus discouraging for readers). Thus, to derive the functional equations as painlessly as possible, a modified approach will be described which shouldn't go too far wrong. The first step is to define the discrete decision process. The DDP is a four-tuple $D = (A, S, P, f)$ where

> $A = \{$ primitive decisions $\}$,
>
> $S = \{$ feasible policies $\} = \{$ tuples of primitive decisions which transform the start state to a final state $\}$,
>
> $P = \{$ sets of data used by the cost function f $\}$, and
>
> f = cost function, defined over all $x \in A^*$,

where A^* is the set of all n-tuples of elements of A for $n \geq 0$. The case $n = 0$ is the empty string and is denoted "e". For the problem of Figure 2-1 the DDP $D = (A, S, P, f)$ is defined by

> $A = \{ a_1, a_2\ a_3 \}$ where a_j transforms the state from node (i-1,k) to node (i,j),
>
> $S = \{ a_i a_j a_k a_1 \mid i,j,k \in \{1,2,3\} \}$,
>
> $P = \{$ matrices $C \mid C_{ijk} \geq 0, \forall i \in \{0,1,2,3\}, j,k \in \{1,2,3\} \}$, and
>
> f is defined recursively by: $f(e;p) = 0$, $f(a_i,p) = C_{01i}$, $f(xa_ia_j,p) = f(xa_i,p) + C_{nij}$ where $n = \text{length}(x) + 1$, and $p \in P$.

The conversion from DDP to SDP (sequential decision process) is not entirely automatic. We must first define the states for the SDP (and thus the DP algorithm) or do something else that is equivalent.[1] Once the states are defined, the other components of the SDP convert fairly automatically from the DDP. The discrete decision process D is transformed to a sequential decision process W = (Z, P, h, k) where

Z is a finite state automaton $Z = (A, Q, q_0, F, \lambda)$ where

A = alphabet for Z (same A as for the DDP above),

Q = { states },

q_0 = initial state,

F = { final states }, *! correspond to S in the DDP*

λ = transition function,

P = data for the cost functions h and k (same as P for the DDP),

h = cost function defined by: $h(r,q,a,p)$ = cost of reaching state $\lambda(q,a)$ by first reaching state q at a cost r and then applying the sequence of decisions a. The function h relates to f by: $h(r,q,a,p) = f(xa,p)$ where $f(x,p) = r$ and $\lambda(q_0,x) = q$.

k(p) is the cost of the null sequence (of primitive decisions) e.

The SDP for the problem of Figure 2-1 is a four - tuple W = (Z, P, h, k) where

Z is an fsa $Z = (A, Q, q_0, F, \lambda)$ where

$A = \{ a_1, a_2, a_3 \}$,

$Q = \{ s_{01}, s_{11}, s_{12}, s_{13}, s_{21}, s_{22}, s_{23}, s_{31}, s_{32}, s_{33}, s_{41} \}$,

$q_0 = \{ s_{01} \}$,

$F = \{ s_{41} \}$,

λ is defined by: $\lambda(s_{ij},a_k) = s_{(i+1),k}$,

$P = \{ \text{matrices } C \mid C_{ijk} \geq 0, \forall\, i \in \{0,1,2,3\},\ j,k \in \{1,2,3\} \}$,

h is defined by: $h(r,s_{ij},a_k,p) = r + C_{ijk}$, and

k is defined by: $k(p) = 0$.

Now that we have defined the SDP, if the monotone property (described below) is satisfied then the SDP is a ***monotone sequential decision process*** (MSDP) from which the functional equations for the DP algorithm can be derived.

---

[1] Karp and Held use a complicated argument that involves a (finite rank) equiresponse relation of an automaton from which the states are derived.

Monotone property: If $r_1 \le r_2$ then $h(r_1,q_0,x,p) \le h(r_2,q_0,x,p)$, $\forall x \in A^*$.

(For the problem of Figure 2-1 this is easily satisfied because of the additivity of h(r,q,a,p).) The functional equations define a function G such that

$$G(q,p) = \min_{\{x|\lambda(q_0,x)=q\}} h(k(p),q_0,x,p).$$

That is, G(q,p) is the minimum possible cost to get to state q, under the cost function h. Now it is possible to define the functional equations.

Functional Equations:

$$G(q_0,p) = \min\left(k(p), \min_{\{(s,a)|\lambda(s,a)=q_0\}} h(G(s,p),q,a,p) \right)$$

$$G(q,p) = \min_{\{(s,a)|\lambda(s,a)=q\}} h(G(s,p),q,a,p) \text{ for } q \ne q_0.$$

For the problem of Figure 2-1 this is

$$G(s_{01},p) = 0$$

$$G(s_{ij},p) = \min_{(s_{(i-1),k},a_j)} (G(s_{(i-1),k},p) + C_{(i-1),k,j})$$

Letting $M_{ij} = G(s_{ij},p)$, this is

$$M_{01} = 0$$

$$M_{ij} = \min_k (M_{(i-1),k} + C_{(i-1),k,j}), \text{ for } (i,j) \ne (01).$$

This equation is the same as the functional equation given for this problem in Chapter 2.

The results of Karp and Held provide a powerful approach for the construction of DP algorithms but several details are left unsolved. For example, Karp and Held note that a good heuristic for reducing the time required by a DP algorithm is to minimize the number of states in the SDP. The problem of minimizing the number of states, however, is not in general solvable in a finite number of steps [38]. Another detail left unsolved by Karp and Held is the problem of transforming the problem statement to a discrete decision process. Given a problem statement, there may be several ways to represent the corresponding DDP.

In the following chapters we will apply the techniques presented in this chapter to several subclasses of DP problems. Since the notation is somewhat cumbersome a small compromise has been made. The cost functions f(x,p) and h(r,q,a,p) shall be (more conveniently, although less correctly) denoted by simply f(x) and h(r,q,a), respectively.

4. Single Source Shortest Path Problems

The problem of Chapter 2 is a special case of the single source shortest path problem. But before describing the general problem, we should first note some other interesting problems that fall into this special case. One of the better known problems of this type is the stagecoach problem ([72]]), which is commonly used as an introduction to DP in operations research textbooks. A rather curious example of this type of algorithm is the ordinary, grade school algorithm for long division [8]. But of more interest to computer scientists is the Viterbi algorithm. The Viterbi algorithm [71, 19] is formally a stagecoach type of problem, so it is not necessary to describe it in detail here. Its applications, however, include not only estimation problems in digital communications [9] but also text recognition [19] and syntactic pattern recognition [61]. The stagecoach problem has also been encountered in speech recognition [3] and in image understanding systems [16].

We will now describe the general single source shortest path problem. Let G be a (directed) graph of N nodes and E edges. The edge (if any) from node N_i to node N_j is edge E_{ij}. The cost structure is defined by the weights $C_{ij} \geq 0$ corresponding to the edges E_{ij}. The problem is to find the shortest path from node N_1 to node N_N. For now we will assume that the graph G is a complete graph. (For each pair of nodes N_i and N_j there is an edge E_{ij}.) If a particular graph is not already complete, we can easily make it so by augmenting it with edges E_{ij} with weights $C_{ij} = \infty$.

Our goal now is to obtain the functional equations for a DP algorithm that solves this problem. To do this, however, the problem will have to reworded slightly. Instead of directly finding the shortest path from node N_1 to node N_N, the algorithm will find the *length* of the shortest path. For this problem only a trivial modification of the algorithm is then necessary to obtain the actual shortest path. For some problems in other chapters, however, the distinction is not so trivial.

To obtain the functional equations the first step is to define the discrete decision process (DDP) D = (A, S, P, f). For the single source shortest path problem this is:

$A = \{ a_1, a_2, \ldots a_N \}$, *where "$a_i$" means "go to node i"*

$S = \{ A^* a_N \}$, = *set of all feasible paths from node 1 to node N*

$P = \{$ matrices $C \mid C_{ij} \geq 0, C_{ii} = 0, i,j=1,2,\ldots N \}$, and

f, the cost function, is defined recursively by: $f(e) = 0$, $f(a_i) = C_{1i}$, $f(xa_ia_j) = f(xa_i) + C_{ij}$, $\forall x \in A^*$.

The corresponding SDP W = (Z, P, h, k) is

$Z = fsa\ (A, Q, q_0, F, \lambda)$ where

A = same as for the DDP,

$Q = \{ s_1, s_2, \ldots s_N \}$, *where "$s_i$" means "at node i"*

$q_0 = \{ s_1 \}$,

$F = \{ s_N \}$,

$\lambda(s_i, a_j) = s_j$,

P = same as for the DDP,

$h(r, s_i, a_j) = r + C_{ij}$, and

$k(p) = 0$.

The monotonicity property is satisfied easily by the additivity of the cost function. Thus, the functional equations are:

$$G(s_1) = 0$$
$$G(s_i) = \min_{(s_j, a_i)} h(G(s_j), s_j, a_i) = \min_j (G(s_j) + C_{ji})$$

Letting $M(i) = G(s_i)$ this is

$$M(1) = 0$$
$$M(i) = \min_j (M(j) + C_{ji})$$

We will now show how to solve these functional equations.

It is not as intuitively obvious how to solve these equations as it was for the stagecoach problems. The solution is known as Dijkstra's algorithm [12, 6, 60] and requires $O(N^2)$ time and storage. (For a general graph this is the best that one can do because there are $O(N^2)$ edges and any shortest path algorithm will, in the worst case, have to look at all of the edges.) The algorithm runs in $N - 1$ stages and in each stage i finds the ith closest node to node N_1. But each stage requires a comparison between $O(N)$ nodes, so the total time is $O(N^2)$.

Algorithm for Single Source Shortest Path

Input: Complete graph G with internode distances C(I,J).
Output: G(I) = distance of shortest path from Node 1 to Node I.
H(I) = last node before Node I in an optimal path from Node 1 to Node I.
Time: $O(N^2)$, Storage: $O(N^2)$

```
! Initialization;
For I ← 1 thru N do
     begin G(I) ← C(1,I); H(I) ← 1; end;
NODES ← { 1, 2, 3, . . . N };

! S = set of nodes for which an optimal path has been found;
     S ← { 1 };   NEW ← 1;

! Optimization;
For I ← 2 thru N do
begin
     MINCOST ← ∞;
     For J ∈ NODES - S do
     begin
          NEWCOST ← G(1,NEW) + C(NEW,J);
          If NEWCOST < G(J) then
          begin
               G(J) ← NEWCOST;   H(J) ← NEW;
          end;
          If G(J) < MINCOST then
          begin
               MINCOST ← G(J);   MINJ ← J;
          end
     end;
     S ← S ∪ { MINJ };   NEW ← MINJ;
end;
```

This algorithm can be modified to run in O(E logN) time and O(E) storage, where E is the number of edges in the graph. (Aho, Hopcroft, and Ullman [2] give credit to Johnson [42].) For graphs where $E < N^2/\log N$ this is better than $O(N^2)$ time and space. For example, any planar graph of N nodes (N > 2) has at most 3*N - 6 (undirected) edges (or 6*N - 12 directed edges) [26]. Thus, the single source shortest path problem for a planar graph can be solved in O(N logN) time and O(N) storage. These and related graph algorithms are described in Aho, Hopcroft, and Ullman [2]. An interesting application of Dijkstra's algorithm since 1974 is described in Duncan [14]. A recent survey of fast *expected time* algorithms for the shortest path problem (and other simple path problems) is provided in Perl [57]. Fredman [20] (and references) describe the *all points* shortest path problem.

5. Optimal Parenthesization Problems

In this chapter we present several problems that can be described as optimal parenthesization problems. That is, they can all be put in the form:

> Given a string of N elements, find an optimal (in some sense) parenthesization of the string.

These problems include optimization of the multiplication of N matrices, context free language recognition, various kinds of optimal search trees, and optimizing the search for N order statistics.

The obvious brute force method for solving an optimal parenthesization problem is to simply examine all of the possible complete parenthesizations of N elements and choose the best one. Unfortunately, this rapidly becomes unmanageable. The number of ways to completely parenthesize a string of N elements is the Nth Catalan number

$$\mathbf{X}(N) = \frac{\binom{2N}{N}}{(N+1)}.$$

It can be shown that $\mathbf{X}(N) \geq 2^{N-2}$ ([2], p.73).

The DP algorithms for these problems offer substantial improvement over the brute force method. This is because an optimally parenthesized string must be composed of optimally parenthesized substrings. (If a substring can be better parenthesized, then the entire string can be better parenthesized.) DP uses this property of optimal parenthesization problems to achieve functional equations of the form

$$M_{ik} = \min_{i<j<k} f_{ijk}(M_{ij}, M_{(j+1),k}).$$

The optimal parenthesization for the substring (i,k) is determined by examining the k - i pairs of M_{ij} and $M_{(j+1),k}$ (which have already been computed). Since there are $O(N^2)$ substrings (i, k), the total time is $O(N^3)$, much less than $O(\mathbf{X}(N))$.

For two of the problems, optimal alphabetic encoding and optimal search for N order statistics, the $O(N^3)$ time can be improved to $O(N \log N)$ time. Unfortunately, this speedup relies on properties that are peculiar to these individual problems and cannot be applied to all of the parenthesization problems. A nice decomposition into *parallel* computations, however, is available for any optimal parenthesization problem. Guibas, Kung, and Thompson [25] have shown how $(N + 1)(N + 2) / 2 = O(N^2)$ mesh-connected parallel processors can solve an optimal parenthesization problem in O(N) time. (In general, a K by K triangular mesh-connected processor can improve the time to $O(N^3/K^2)$.)

5.1 Optimizing the Multiplication of N Matrices

The first problem that we will describe in this chapter is the problem of optimizing the multiplication of N matrices. Since matrix multiplication is associative, the multiplications can be grouped as desired and (mathematically) the result will still be the same. Computationally, however, it may make a tremendous difference. For example, consider the following multiplication problem:

$$M = \underset{[5\times 10]}{M_1} \times \underset{[10\times 100]}{M_2} \times \underset{[100\times 2]}{M_3} \times \underset{[2\times 20]}{M_4} \times \underset{[20\times 50]}{M_5}.$$

We will assume in this discussion that the multiplication of a p x q matrix and a q x r matrix costs pqr operations. (Probert [62] generalizes this problem to allow Strassen matrix multiplication as well as the ordinary block multiplication.) If the multiplications are grouped

$$M = M_1 \times (M_2 \times ((M_3 \times M_4) \times M_5))$$

then the multiplication will cost 156,500 operations. If, however, the grouping is

$$M = (M_1 \times (M_2 \times M_3)) \times (M_4 \times M_5)$$

then the multiplication will cost only 4600 operations.

We will now describe the general problem and the corresponding $O(N^3)$ time DP algorithm.[1] Let

$$M = M_1 \times M_2 \times \ . \ . \ . \times M_N$$

where M_1 is a c_0 by c_1 matrix, M_2 is a c_1 by c_2 matrix, etc. The discrete decision process D = (A, S, P, f) is defined by

> $A = \{ a_{ijk} \mid 1 \le i \le j < k \le N \}$, *where* a_{ijk} *means "multiply the result of* $M_i \times M_{i+1} \times \ldots \times M_j$ *by the result of* $M_{j+1} \times M_{j+2} \times \ldots \times M_k$."
>
> S = { sequences of a_{ijk} corresponding to complete parenthesization of the N matrices },
>
> $P = \{ c \in (I^+)^{N+1} \}$, and
>
> $f(e) = 0$, $f(xa_{ijk}) = f(x) + c_{i-1} * c_j * c_k$.

We must first define the states Q before constructing the SDP. The most natural set of states for this problem is the set of all partial parenthesizations of N elements. The SDP W = (Z, P, h, k) is then defined by

---

[1] Chin [10] describes a non-DP O(N) time approximation algorithm that is never worse than a factor of 5/4 from optimal.

$Z = (A, Q, q_0, F, \lambda)$, (The components are obvious.)

P = same as for the DDP,

$h(r,q,a_{ijk}) = r + c_{i-1} * c_j * c_k$, and

$k(p) = 0$.

Monotonicity of the SDP is assured by the additivity of h. Thus the functional equations are

$$G(q_0) = 0, \text{ and}$$

$$G(q) = \min_{\{(s,a)|\lambda(s,a)=q\}} h(G(s),s,a)$$

The optimization procedure suggested by these equations will certainly produce an optimal parenthesization. The time spent, however, will be prohibitive. This is because the number of states that will have to be examined is the number of partial parenthesizations of N elements, which is at least as great as the Nth Catalan number $X(N)$. This does not compare well with the $O(N^3)$ time that was promised above. Obviously, the choice of states will have to be made more carefully. The natural choice is not always the best.

Let's re-examine the situation. To produce a faster DP algorithm the number of states must be reduced. There is no general procedure for this. In fact, Ibaraki [38] has shown that the problem is in general unsolvable. Thus, to reduce the number of states we must exploit properties peculiar to this particular problem. In this case a clue is provided by a closer examination of the primitive operators a_{ijk} in the set A. If the substrings $(M_i,M_{i+1},...M_j)$ and $(M_{j+1},M_{j+2},...M_k)$ are already fully parenthesized, then application of the operator a_{ijk} will make $(M_i,M_{i+1},...M_j,...M_k)$ a fully parenthesized substring. This implies that the states should be of the form s_{ij}, $1 \le i \le j \le N$ where s_{ij} has the interpretation "substring $(M_i,M_{i+1},...M_j)$ is fully parenthesized."

Unfortunately, there is no way that a parenthesization such as ((M x M) x (M x M)) can be produced by the primitive operators with only these states S_{ij}. Thus, another property peculiar to parenthesization problems will have to be pulled out of a hat. This one is a kind of *decomposition* property and it looks more like *divide-and-conquer* [2] than dynamic programming. The property is that the parenthesization of a substring (i,j) is independent of the parenthesization of a substring (k,l) iff $i < j < k < l$ or $k < l < i < j$. Thus, the problem of producing a state, call it $(S_{ij} \cup S_{(j+1),k})$, for which a_{ijk} fully parenthesizes substring (i,k) (and produces state S_{ik}) can be decomposed into the problem of solving S_{ij} and solving $S_{(j+1),k}$. Since the cost function is additive,

$$G((S_{ij} \cup S_{(j+1),k})) = G(S_{ij}) + G(S_{(j+1),k}).$$

Thus, the functional equations become

$$G(q_0) = 0, \text{ and}$$

$$G(S_{ik}) = \min_{(S_{ij} \cup S_{(j+1),k})} G((S_{ij} \cup S_{(j+1),k})) + c_{i-1} * c_j * c_k$$

$$= \min_{j} G(S_{ij}) + G(S_{(j+1),k}) + c_{i-1} * c_j * c_k.$$

Letting $G_{ij} = G(S_{ij})$, this becomes

$$G_{ii} = 0, \text{ and}$$

$$G_{ik} = \min_{j} G_{ij} + G_{(j+1),k} + c_{i-1} * c_j * c_k.$$

which is the type of recurrence desired. Here is a pseudo-Algol algorithm that solves the functional equations:

Algorithm For Computing Optimal Multiplication of N Matrices

Input: integer N > 0, array C[1:N+1] of positive integers
Output: Array G where G(i,j) = minimum cost of parenthesizing substring (i,j).
Array H where H(i,j) = subscript k where optimal parenthesis for substring (i,j) is located.
Time: $O(N^3)$, Storage: $O(N^2)$

```
For L ← 1 thru N do
      G(L,L) ← 0;
For L ← 2 thru N do           ! L = length of substrings optimized
      For I ← 1 thru N+1-L do
      begin
            MINCOST ← ∞;
            J ← I + L -1;
            For K ← I thru J-1 do   ! Find optimal paren for substring (I,J)
            begin
                  COST ← G(I,K) + G(K+1,J) + C(I-1) * C(K) * C(J);
                  If COST < MINCOST then
                  begin
                        MINCOST ← COST;
                        G(I,J) ← COST;
                        H(I,J) ← K
                  end
            end
      end;
```

5.2 Context Free Language Recognition

Context free language recognition is a well studied problem. Younger [75] showed that a Turing machine can recognize a general context free language in $O(N^3)$ time, where N is the length of the string being recognized. Kasami and Torii [45] constructed an $O(N^2)$ time (RAM) algorithm for *unambiguous* context free language recognition and Early [15] produced an algorithm that not only recognizes general context free languages in $O(N^3)$ time, but also unambiguous languages in $O(N^2)$ time, and some further restricted classes of languages (such as LR(k)) in O(N) time. Since then Valiant [70] constructed an $O(N^{2.81})$[1] time general context free recognition algorithm (by using Strassen matrix multiplication), and Graham, Harrison, and Ruzzo [24] produced an $O(N^3/\log N)$ time on-line algorithm for general context free recognition. In this paper we describe the simplest $O(N^3)$ time algorithm for general context free language recognition.

The DP algorithm for general context free recognition that is presented in this paper requires the context free grammar to be put in Chomsky normal form [32]. (Early [15] does not require Chomsky normal form, but his algorithm is more complicated, too.) The grammar is a 4 - tuple

G = (Nonterminals, Terminals, Productions, Start symbol).

Nonterminal symbols are indicated by upper case letters (A, B, C, . . .) and terminal symbols are indicated by lower case letters (a, b, c, . . .). The productions are all of the form $A \to BC$ or $A \to a$ (Chomsky normal form). If m, n, and q are strings of terminals and nonterminals, then $mAn \to mqn$ iff $A \to q$ is a production. The language generated by the grammar is the set of strings of terminals w such that $S \to* w$ where $\to*$ is the transitive closure of $\to$.

To construct the DP algorithm we must first define the DDP. The DDP makes use of a function called "Prefix" defined as follows:

> Let the input string be $Z = z_1z_2...z_N$, and let x be a string of primitive operators a_{ijk}. Prefix(x,a_{ijk}) = the string x minus all operators to the right of a_{ijk}. If a_{ijk} does not occur in x then Prefix(x,a_{ijk}) = e.

For example, if $x = a_{233}a_{123}a_{455}a_{135}$, then Prefix$(x,a_{455}) = a_{233}a_{123}a_{455}$. The DDP is D = (A, S, P, f):

> $A = \{ a_{ijk} \mid 1 \le i < j \le k \le N \}$ where a_{ijk} produces the set $\{ B \mid B \to CD$ and $C \to* z_iz_{i+1}...z_j$ and $D \to* z_{j+1}z_{j+2}...z_k \}$,
>
> $S = \{ A^* a_{1jN} \}$,

---

[1]Now improved to $O(N^{2.79})$ with the new matrix multiplication algorithm of Pan [54].

$P = \{$ productions $\}$, and

$f(e) = e$, $f(xa_{ijk}) = \{ B \mid B \to CD$ where $\exists p\ C \in f(\text{Prefix}(x,a_{ipj}))$ and $\exists q\ D \in f(\text{Prefix}(x,a_{(j+1),q,k})) \}$.

This is a most curious way to define a DDP. The primitive operators and, especially, the cost function return *sets* rather than scalars. The reason for returning sets is that in this problem we are interested in finding *all* of the nonterminals that produce the substrings $z_i z_{i+1} \ldots z_j$. If there are two or more nonterminals that can generate a substring $z_i z_{i+1} \ldots z_j$ it is not sufficient to keep only one nonterminal and throw the others away because the productions do not treat the nonterminals equivalently. Thus, if $f(xa_{ipk}) = \{ B, D, E \}$ and $f(xa_{iqk}) = \{ D, E \}$, then a_{ipk} will certainly be preferred over a_{iqk}. If, however, $f(xa_{ipk}) = \{ B, D, E \}$ and $f(xa_{irk}) = \{ C \}$ then we cannot say which is better because the two sets are not comparable. This situation is described as a *partial order* relation. It is defined by

$$f(xa_{ipk}) \le f(xa_{iqk}) \text{ iff } f(xa_{ipk}) \subset f(xa_{iqk}).$$

Furthermore, the object is to *maximize* rather than *minimize* f under the partial order in this problem. (Accordingly, we will replace "min" with "union" in the functional equations.)

Now it is time to produce the SDP. Let the state S_{ij} be interpreted as "$z_i z_{i+1} \ldots z_j$ has been parsed". To produce an efficient algorithm, we use pseudo-states $(S_{ij} \cup S_{kl})$ as in the previous parenthesization problem. The SDP is $W = (Z, P, h, k)$ where

$Z = \text{fsa} (A, Q, q_0, F, \lambda)$ where

A = same as for the DDP,

$Q = \{ S_{ij}, (S_{ij} \cup S_{kl}) \}$,

$q_0 = \{ e \}$,

$F = \{ S_{1N} \}$,

$\lambda((S_{ij} \cup S_{(j+1),k}), a_{ijk}) = S_{ik}$,

P = same as for the DDP,

$h(r,(S_{ij} \cup S_{(j+1),k}),a_{ijk}) = f(xa_{ijk})$ where $\lambda(q_0,x) = (S_{ij} \cup S_{(j+1),k})$, and

$k(p) = e$.

Before we derive the functional equations we must first show that the monotonicity property is satisfied. Note that the cost function is not simply additive as in the previous problems so monotonicity does not come quite as automatically. Thus, we must show that if $x,y \in A^*$, then

$$\text{If } \lambda(q_0,x) = \lambda(q_0,y) \text{ and } f(x) \ge f(y) \text{ then } f(xw) \ge f(yw), \forall w \in A^*.$$

This is true because if $f(x) \supset f(y)$ then certainly $\forall w \in A^*\ f(xw) \supset f(yw)$. We conclude that the functional equations are

$G(q_0) = e$, and

$$G(S_{ik}) = \bigcup_{(S_{ij} \cup S_{(j+1),k}, a_{ijk})} h(\, G(S_{ij} \cup S_{(j+1),k}), (S_{ij} \cup S_{(j+1),k}), a_{ijk}, p)$$

$$= \bigcup_{j} f(x_1 x_2 a_{ijk}) \text{ where } \lambda(q_0,x_1) = S_{ij} \text{ and } \lambda(q_0,x_2) = S_{(j+1),k}$$

Here is an algorithm for solving the functional equations:

Algorithm For General Context Free Language Recognition

Input: Nonterminals = $\{A_1, A_2, \ldots A_{NT}\}$ *! A_1 = start symbol*
Terminals = $\{t_1, t_2, \ldots t_T\}$
Finite set of productions of the form: $A_i \rightarrow A_j A_k$ or $A_i \rightarrow t_j$
String $Z = z_1 z_2 \ldots z_N$ where $z_i \in$ Terminals
Output: Array M where $M(I,K) = \{ A \mid A \rightarrow^* z_I z_{I+1} \ldots z_K \}$
Thus, Z is recognized iff start symbol = $A_1 \in M(1,N)$.
Time: $O(N^3)$, Storage: $O(N^2)$.

```
For I ← 1 thru N do
      M(I,I) ← { A | "A→a_I" ∈ Productions }
For L ← 2 thru N do              ! L = length of string;
      For I ← 1 thru N+1-L do
      begin
            K ← I + L - 1;
            M(I,K) ← e;              ! e = empty set
            For J ← I thru K-1 do
                  M(I,K) ← M(I,K) ∪ { A | ∃α∈M(I,J),β∈M(J+1,K) (A→αβ) }
      end;
```

Note that the inner loop is not implemented the same here as for the previous two algorithms. This is because the partial order relation forces us to use the union operation rather than a min. Since the set of nonterminals is finite, the size of each of the sets M(I,K) is bounded by a constant. Thus, the entire algorithm still requires only $O(N^3)$ time.

5.3 Optimal Alphabetic Encoding and Optimal Search Trees

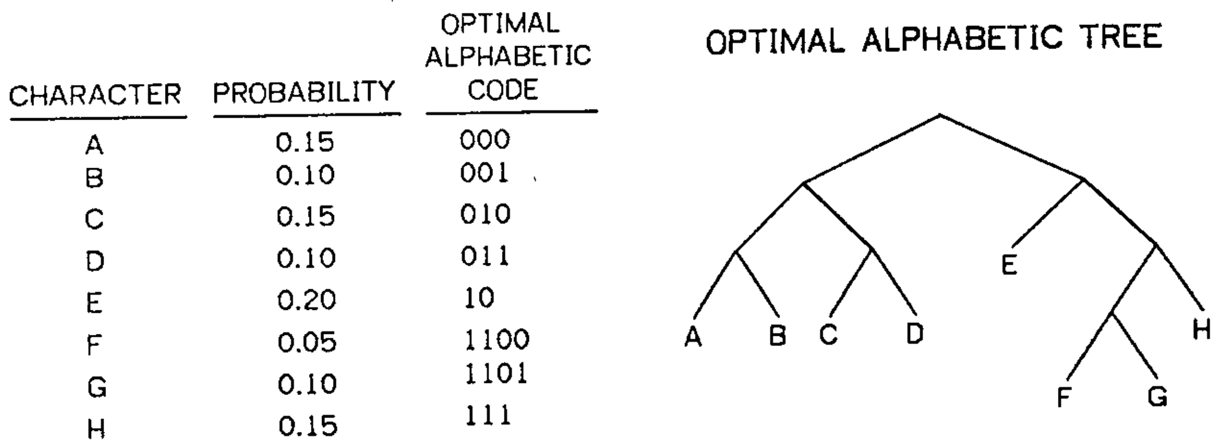
There are many kinds of optimal search trees and optimal alphabetic encoding corresponds naturally to one of the simpler kinds. An optimal alphabetic encoding bears several similarities to a Huffman code [34]. We are given an alphabet ALPH of N characters and the probabilities p_i that a randomly chosen character is character i. Each of the characters must

be represented by a *prefix code* chosen from a set of r symbols.[1] Furthermore, for an alphabetic encoding, the codes for the letters of ALPH must be in increasing numerical order. (For a Huffman encoding the codes are not restricted to be ordered.) Let l_i = length of the encoding of character i. The objective is to minimize the expected length of a message, that is

$$\min_{\{\text{alphabetic encodings}\}} \sum_{i=1}^{N} p_i l_i.$$

Gilbert and Moore [22] were the first to use DP to solve the problem of constructing an optimal alphabetic encoding. Their algorithm is very similar to the algorithms for the other optimal parenthesization problems, and it requires $O(N^3)$ time and $O(N^2)$ storage.

The tree corresponding to an optimal alphabetic encoding is illustrated in Figure 5-1. A "0" in the encoding corresponds to a left branch of the tree and a "1" corresponds to a right branch. Each leaf of the tree corresponds to one of the letters of the alphabet. The internal nodes do not represent letters of the alphabet. A general binary search tree, on the other hand, may have elements at internal nodes as well as the leaves.



CHARACTER	PROBABILITY	OPTIMAL ALPHABETIC CODE
A	0.15	000
B	0.10	001
C	0.15	010
D	0.10	011
E	0.20	10
F	0.05	1100
G	0.10	1101
H	0.15	111

Figure 5-1: Example of Optimal Alphabetic Encoding and Related Decision Tree

Knuth describes the construction of a general optimal search tree in $O(N^2)$ time (and storage) ([46], [47],p.433-439, [2], p.119-123). Not only does this search tree include elements at internal nodes but it also is extended to allow (nonzero) probabilities that

[1] For this presentation we will assume that r = 2, thus making the codes binary codes.

unsuccessful searches are made in the tree. That is, the search key may be less than A, between A and B, between B and C, etc. Hu and Tucker [36] produced a much different (and complicated) algorithm to construct an optimal alphabetic tree in $O(N^2)$ time and $O(N)$ storage (see also Hu [33]), which Knuth [47] improved to $O(N\log N)$ time and $O(N)$ storage. Since then, Itai [40] has produced DP solutions for several constrained variations of the problem. Payne and Meisel [56] describe an interesting DP algorithm for an optimal multi-dimensional binary tree. We will present the optimal alphabetic encoding algorithm of Gilbert and Moore. (For treatments of the general optimal search tree the reader is referred to the texts of Knuth and Aho, Hopcroft, and Ullman cited above.)

The first step toward solving the optimal alphabetic encoding problem is to define the DDP. Let the alphabet ALPH = $(z_1, z_2, \ldots z_N)$. The DDP D = (A, S, P, f) is

> $A = \{ a_{ijk} \mid 1 \le i \le j < k \le N \}$ where a_{ijk} appends a "0" to *the front of* the code for $z_i, z_{i+1}, \ldots z_j$ and appends a "1" to the front of the code for $z_{j+1}, z_{j+2}, \ldots z_k$,
>
> S = { sequences of a_{ijk} corresponding to total parenthesizations of N elements },
>
> $P = \{ (p_1, p_2, \ldots p_N) \in (\text{Reals}^+)^N \}$, and
>
> $f(e) = 0,\ f(xa_{ijk}) = f(x) + \sum_{m=i,k} p_m.$

Note that the cost function f does not seem to depend on j at all. The dependence on j comes from the constraint on the legal parenthesizations in S, as will be shown in the functional equations.

To construct the SDP we must first define the states. Let the state S_{ij} be interpreted as "an optimal alphabetic code has been constructed for $z_i z_{i+1} \ldots z_j$". To produce an efficient algorithm, we will use pseudo-states $(S_{ij} \cup S_{kl})$ as in the previous optimal parenthesization problems. The SDP is W = (Z, P, h, k) where

$Z = (A, Q, q_0, F, \lambda)$ where

A = same as for the DDP,

$Q = \{ S_{ij}, (S_{ij} \cup S_{kl}) \mid 1 \le i \le j < k \le l \le N \} \cup \{ e \}$,

$q_0 = e$,

$F = \{ S_{1N} \}$,

$\lambda((S_{ij} \cup S_{(j+1),k}), a_{ijk}) = S_{ik}$,

P = same as for the DDP,

$h(r, (S_{ij} \cup S_{(j+1),k}), a_{ijk}) = r + \sum_{m=i,k} p_m$, and

$k(p) = 0$.

The monotonicity property is satisfied by the additivity of the cost function and thus the functional equations are

$$G(e) = 0, \text{ and}$$

$$G(S_{ijk}) = \min_{(S_{ij} \cup S_{(j+1),k}), a_{ijk}} h(G(S_{ij} \cup S_{(j+1),k}), (S_{ij} \cup S_{(j+1),k}), a_{ijk})$$

$$= \min_j G_{ij} + G_{(j+1),k} + \sum_{m=i}^{k} p_m.$$

Letting $G_{ij} = G(S_{ij})$ this becomes

$$G_{ii} = 0, \text{ and}$$

$$G_{ik} = \min_j G_{ij} + G_{(j+1),k} + \sum_{m=i}^{k} p_m.$$

These functional equations can be solved in $O(N^3)$ time because the summation from m=i to m=k of p_m can be computed in O(1) time after some preprocessing. The following algorithm shows how that is done.

Algorithm for Optimal Alphabetic Encoding

Input: integer N > 0, array P[1:N] of positive reals
Output: G(I,K) = cost of optimal encoding of $z_I z_{I+1} \ldots z_K$.
H(I,K) = subscript J for optimal partition of $z_I z_{I+1} \ldots z_K$.
Time: $O(N^3)$, Storage: $O(N^2)$

```
! Initialization of G and C.;
C(0) ← 0;
For I ← 1 thru N do
begin
     G(I,I) ← 0;   C(I) ← C(I-1) + P(I)
end;
! Optimization over all pairs (I,K).;
For L ← 2 thru N do    ! L = length of string;
     For I ← L thru N+1-L do
     begin
          K ← I + L - 1;
          MINCOST ← ∞;
          For J ← I thru K-1 do
          begin
               COST ← G(I,J) + G(J+1,K) + C(K) - C(I-1);
               If COST < MINCOST then
               begin
                    MINCOST ← COST
                    G(I,K) ← COST;
                    H(I,K) ← J;
               end
          end
     end;
```

5.4 Optimizing the Search for N Order Statistics

Optimizing the search for N order statistics is very similar to optimal alphabetic encoding. The problem is described as follows:

> Let X be a set of M real numbers x_i. We want to find N ($N \le M$) order statistics s_i where s_i equals the a_i'th smallest element of X. For example, if $X = \{1,5,2,9,-2\}$, $M = 5$, $N = 1$, and $a_1 = 4$, then $s_1 = 5$. The problem is to find the N order statistics s_i as quickly as possible.

One way to find the order statistics is to sort the M elements of X in O(MlogM) time and then obtain the N order statistics in O(N) time (by using the a_i as indices into the sorted set X) for a total time of O(MlogM). For large M we can do better. For the case N = 1, the order statistics s_i can be found in O(M) time[7, 63]. Pohl [59] describes a special case of N = 2, the

min and the max, and shows how they can be found in $\lceil 3M/2 - 2 \rceil$ comparisons, which is better than the obvious $2M - 2$ comparison algorithm. The DP algorithm that we describe in this paper makes optimal use of a linear time selection algorithm. It does not attempt to do any optimization of the kind described by Pohl.

To construct the DP algorithm we start, as always, with the discrete decision process $D = (A, S, P, f)$:

> $A = \{ a_0, a_1, a_2, \ldots a_N, a_{N+1} \}$, // *where a_i, $1 \le i \le N$ finds the order statistic s_i, $a_0 = 0$, and $a_{N+1} = M+1$.*
>
> $S = \{ x \in A^* \mid \text{all } a_i \text{ are in } x \}$,
>
> $P = \{ a \in M^N \}$, and
>
> $f(e) = 0$, $f(a_i) = M$, $f(xa_i) = f(x) + a_k - a_j$ where j = max subscript of a in x that is less than i, and k = min subscript of a in x that is greater than i.

When constructing the SDP we encounter exactly the same problems with the number of states as with the problem of optimal multiplication of N matrices. Fortunately, the decomposition property still applies so the trick of constructing "states" $(S_{ij} \cup S_{kl})$ will work. Since the details are basically the same here as for the previous parenthesization problems, we will present only the recurrence relation and the algorithm.

Functional Equations for Optimizing the Search for N Order Statistics

$$G(i,i+1) = 0$$

$$G(i,k) = \min_{i<j<k} G(i,j) + G(j,k) + a_k - a_i - 1$$

Algorithm for Optimizing the use of a Linear Selection Algorithm to Find N Order Statistics

Input: Vector A such that 0 = A(0) < A(1) < A(2) < . . . A(N) < A(N+1) = M+1
Output: Array G where G(I,K) = minimum cost of finding all order statistics A(J), I < J < K, given that A(I) and A(K) have already been determined.
Array H where H(I,K) = subscript J of optimal A(J) for subdividing the interval [A(I),A(K)].
Time: $O(N^3)$, Storage: $O(N^2)$

```
For L ← 0 thru N do
     G(L,L+1) ← 0;
For L ← 2 thru N+1 do          ! L = length of substrings optimized
     For I ← 0 thru N+1-L do
     begin
          MINCOST ← ∞;
          K ← I    + L;
          ! Find optimal paren for substring (I,K;)
          For J ← I+1 thru K-1 do
          begin
               COST ← G(I,J) + G(J,K) + A(K) - A(I) - 1;
               If COST < MINCOST then
               begin
                    G(I,K) ← MINCOST ← COST;
                    H(I,K) ← J
               end
          end
     end;
```

We can improve this algorithm to O(NlogN) time and O(N) storage. This is because it can be put it in the form of an optimal alphabetic encoding problem. In the functional equations above, the quantity G(i,k) is interpreted to equal the minimum cost of finding all order statistics a_j, $i < j < k$, given that a_i and a_k have already been determined. Change this to

> G(i,k) = minimum cost of finding all order statistics a_j, $i < j \le k$, given that a_i and a_{k+1} have already been determined.

Also rewrite the quantity $a_{k+1} - a_i$ as the sum (from m=i to m=k) of $(a_{m+1} - a_m)$. The new functional equations are

N = 4
M = 100

ORDER STATISTICS
20
35
70
90

G	0	1	2	3	4	5
0		0	34	103	177	229
1			0	49	118	159
2				0	54	95
3					0	30
4						0
5						

H	0	1	2	3	4	5
0			1	2	2	2
1				2	3	3
2					3	3
3						4
4						
5						

Figure 5-2: Example of finding optimal use of a linear selection algorithm.

$G(i,i) = 0$, and

$$G(i,k) = \min_{i<j<k} G(i,j) + G(j+1,k) + \sum_{m=i}^{k} (a_{m+1} - a_m) - 1.$$

The functional equations are now in the same form as the functional equations for an optimal alphabetic encoding. Thus, the O(NlogN) time and O(N) storage algorithm follows.

6. Optimal Partition Problems

Optimal partition problems are not entirely unlike optimal parenthesization problems. They both involve a parenthesization of a string of N elements. The partition problem, however, does not produce a *complete* parenthesization of the N elements. Instead, it partitions the string into K sets of consecutive elements (with K sets of unnested parentheses) for some value of K. The value of K may be a given constant or it may be left unspecified.

Peterson, Bitner, and Howard [58] describe a problem for which the number of partitions K is unspecified. The problem is to select optimal tab settings to minimize the number of blanks in a document. N is the number of columns and K is the number of tabs set. The functional equations, however, are of the form

$$G_0 = 0, \text{ and}$$
$$G_j = \max_i G_i + C_{ij}$$

which is very similar to the solution for the single source shortest path problem described in Chapter 4. Because of this similarity this problem will not be discussed further. The remainder of this chapter will instead describe an example where K is a given constant.

The author knows of two nontrivial problems which are of the optimal partition type (with a given value for the required number of partitions K). One is the resource allocation problem ([13], ch.3), but the best example of this type of problem is the one solved by Fisher's algorithm [17, 27].[1] To describe the problem we will first have to introduce some notation. Let the input string of N elements be denoted $x_1, x_2, x_3, \ldots x_N$, where x_i is a real. A *cluster* (i,j) is simply the substring $(x_i, x_{i+1}, \ldots x_j)$. Corresponding to each cluster (i,j) is a *diameter* $D(i,j) \geq 0$ that is a function of $(x_i, x_{i+1}, \ldots x_j)$. The diameter function that will be used in this description is

$$D(i,j) = \sum_{k=i}^{j} (x_k - \underline{x})^2, \text{ where } \underline{x} = \sum_{k=i}^{j} \frac{x_k}{(j-i+1)}$$

Let a *K - partition* of the N elements be represented by a K+1 - tuple $(I_0, I_1, I_2, \ldots I_K)$ where $0 = I_0 < I_1 < I_2 < \ldots < I_{K-1} < I_K = N$. The interpretation is that the first cluster is (I_0+1, I_1), the second cluster is (I_1+1, I_2), etc. The optimal partition problem (which Fisher's algorithm solves) is

[1] Fast algorithms for several special cases of Fisher's clustering problem will be presented by Shamos, Brown, Saxe, and Weide [66].

$$\min_{\{K\text{-partitions}\}} \sum_{j=0}^{k} D(I_j+1, I_{j+1}).$$

One method of solution is to simply generate all possible K - partitions and choose the one with minimum cost. But there are

$$\binom{N-1}{K-1} = \frac{(N-1)!}{(N-K)! * (K-1)!}$$

possible partitions, which for large K and N will be prohibitive. Dynamic programming provides considerable improvement. To produce the DP algorithm we first construct the DDP $D = (A, S, P, f)$:

$A = \{ a_{ij} \mid 1 \le i < j \le N \}$ where a_{ij} defines the substring $(x_i x_{i+1} \ldots x_j)$ as one of the K sets of the partition,

$S = \{$ sequences of K a_{ij}'s that produce legal partitions $\}$,

$P = \{$ N - tuples of reals $\}$, and

$f(e) = 0$, $f(xa_{ij}) = f(x) + D(i,j)$ for all legal sequences xa_{ij}.

To define the SDP we must first define the states. One way to do this is to let S_{hij} be interpreted "substring (i,j) is partitioned into h sets of consecutive elements." This leads to a DP algorithm that costs $O(KN^3)$ time and $O(KN^2)$ storage. Another choice of states is available, however, that naturally leads to a DP algorithm that costs only $O(KN^2)$ time and $O(KN)$ storage. The improvement is to consider only states $S_{hj} = S_{h1j}$, which are interpreted as "substring (1,j) is partitioned into h sets of consecutive elements." Thus, the SDP $W = (Z, P, h, k)$ is

$Z = (A, Q, q_0, F, \lambda)$ where

A = same as for the DDP,

$Q = \{ S_{hj} \mid 1 \le h \le k, 1 \le i \le N \} \cup \{ e \}$,

$q_0 = e$,

$F = \{ S_{KN} \}$,

$\lambda(S_{hi}, a_{(i+1),j}) = S_{(h+1),j}$,

P = same as for the DDP,

$h(r, S_{hi}, a_{(i+1),j}) = r + D(i,j)$, and

$k(p) = 0$.

Monotonicity of the SDP follows from additivity of the cost function D and the fact that $D(i,j) \ge 0$. Thus, the functional equations are

$$G(e) = 0, \text{ and}$$

$$G(S_{hj}) = \min_{(S_{(h-1),i}, a_{(i+1),j})} h(\, G(S_{(h-1),i}),\, S_{(h-1),i},\, a_{(i+1),j})$$

$$= \min_i G(S_{(h-1),i}) + D(i+1,j)$$

Letting $G_{hj} = G(S_{hj})$, this becomes

$$G_{hj} = \min_i G_{(h-1),i} + D(i+1,j).$$

Fisher's version of the algorithm for solving these equations first computes all of the diameters $D(i,j)$ and stores them in an array, costing $O(N^2)$ storage. The algorithm below creates vectors SUM and SUMSQ, which enable the diameter to be computed in $O(N)$ storage at no extra expense in time. (This trick does not work for all diameter functions, though.)

Algorithm for Constructing an Optimal K-Partition with the Sum of Squares About the Mean Diameter Function

Input: X = N - tuple of reals, K = no. of sets in the partition.
Output: Array G such that G(h,j) = minimum cost for partitioning $x_1 x_2 \ldots x_j$ into h sets of consecutive elements. Array H such that H(h,j) = subscript k for the h'th set of the optimal partition of $x_1 x_2 \ldots x_j$.
Time: $O(KN^2)$, Storage: O(KN)

```
! Define statement function;
     D(I,J) = (SUMSQ(J) - SUMSQ(I-1)) - (SUM(J) - SUM(I-1))^2 / (J - I + 1)
! Compute partial sums and sums of squares of X(I;)
     SUM(0) ← SUMSQ(0) ← 0;
     For J ← 1 thru N do
     begin
          SUM(J) ← SUM(J-1) + X(J);
          SUMSQ(J) ← SUMSQ(J-1) + X(J)*X(J);
          G(1,J) ← D(1,J);   H(1,J) ← J;
     end;
! Optimization;
     For L ← 2 thru K do   ! L = no. of sets in partition;
          For J ← L thru N do
               MINCOST ← ∞;
               For I ← L-1 thru J-1 do
               begin
                    COST ← G(L-1,I) + D(I+1,J);
                    If COST < MINCOST then
                    begin
                         MINCOST ← COST;
                         G(L,J) ← COST;
                         H(L,J) ← I;
                    end
               end;
```

We can further reduce the storage required by this algorithm to O(N). To find the *cost* of the optimal partition in O(N) storage and $O(KN^2)$ time is fairly simple, but the recovery of the optimal partition itself (within the same time and space bounds) is more difficult. First, we consider finding the *cost* of the optimal partition in O(N) storage and $O(KN^2)$ time.

The only trick is to eliminate unnecessary storage that is being used. The diameter function is already computed in O(N) storage. Only the array G costs O(KN) storage. (Forget about H for now since only the optimal cost is being computed.) But in the inner two loops of the above algorithm only the vectors G(L-1,*) and G(L,*) are referenced. That is, at stage L only the vectors G(L-1,*) and G(L,*) are needed. Thus, the K x N array G can be replaced by two vectors of length N and the cost of the optimal partition can be computed in O(N) storage

and $O(KN^2)$ time.

The recovery of the optimal partition itself, rather than just the cost, in $O(N)$ storage and $O(KN^2)$ time can be accomplished by combining divide - and - conquer with dynamic programming. The method is similar to the algorithm of Hirschberg [29] that solves the longest common subsequence problem (to be described later in this paper) in $O(N)$ storage and $O(N^2)$ time.

Recall that the state S_{hi} is interpreted to mean "$x_1 x_2 ... x_i$ are partitioned into h sets of consecutive elements." Define a new set of states S'_{hi} that are interpreted "$x_i x_{i+1} ... x_N$ are partitioned into h sets of consecutive elements." Let $G'(h,i)$ = the minimum cost to reach state S'_{hi}. Also, assume for convenience that K is a power of 2. Here is how the divide-and-conquer algorithm works:

1. Compute $G(K/2,i)$ for all i in $O(K/2 * N^2)$ time and compute $G'(K/2,j)$ for all j in $O(K/2 * N^2)$ time for $O(KN^2)$ total time.

2. Find the subscript i that minimizes $G(K/2,i) + G'(K/2,i+1)$. This subscript i equals the $I_{K/2}$ in the K+1 - tuple $(I_0,I_1,I_2,...I_K)$ that represents the optimal partition of the N elements.

3. Now find $I_{K/4}$ and $I_{3K/4}$ by applying steps 1. and 2. to the respective subproblems. (This is the divide step - solving two subproblems of half the size.)

4. Repeat the divide step of step 3. That is, find $I_{K/8}$, $I_{3K/8}$, $I_{5K/8}$, and $I_{7K/8}$, etc. After logK iterations we will have determined the entire optimal partition.

The total time to recover the optimal partition may at first appear to be $O(KN^2 * \log K)$. With careful analysis, however, we can show it to be only $O(KN^2)$. Let $U(K,N)$ be the time to construct the optimal K-partition of N elements and let $T(K,N) = O(KN^2)$ be the time to find the *cost* of the optimal K-partitions for all the clusters (1,N), (1,N-1), (1,N-2), etc. The above divide-and-conquer algorithm leads to the following equations:

$$T(I,J) = IJ^2$$

$$U(K,N) = \sum_{i=1}^{\log K} \sum_{j=1}^{2^{i-1}} \left[2\, T\left(\frac{K}{2^i}, I_{jk/2^{i-1}} - I_{(j-1)k/2^{i-1}}\right) + O\left(I_{jk/2^{i-1}} - I_{(j-1)k/2^{i-1}}\right)\right]$$

$$= \sum_{i=1}^{\log K} \left[O(N) + 2 \sum_{j=1}^{2^{i-1}} \frac{K}{2^i} \left(I_{jk/2^{i-1}} - I_{(j-1)k/2^{i-1}}\right)^2 \right]$$

$$\leq \sum_{i=1}^{\log K} \left[O(N) + \frac{K}{2^{i-1}} N^2 \right] = O(KN^2)$$

The time U(K,N) depends somewhat on the optimal partition $(I_0, I_1, \ldots I_K)$ but in the worst case the total time is still $O(KN^2)$.

7. Optimal Matching Problems

One of the major differences between optimal matching problems and optimal parenthesization or partition problems is that instead of just one string of elements, two (or more) are involved. The author knows of two problems in this class. The first, and most important, is the longest common subsequence (LCS) problem (and its variations), for which DP produces a good (if not optimal) algorithm. The second problem is the game of NIM, for which a much faster algorithm is known and thus will not be described further [21]. We will now describe the LCS problem.

Let $Y = y_1 y_2 \ldots y_N$ and $Z = z_1 z_2 \ldots z_N$ be two strings of length N whose elements y_i and z_i are chosen from a finite alphabet ALPH. (The strings Y and Z are chosen to be the same length only for convenience in presentation.) A string B is a *subsequence* of Y if B can be obtained by deleting characters from Y. For example, if Y = "dyanamic programming", then B = "main" is a subsequence of Y. The longest common subsequence problem (for the two strings Y and Z) is to find the (not necessarily unique) longest string B that is a subsequence of both Y and Z.

The LCS problem can be solved in $O(N^2)$ time and space [29]. The string editing problem, a generalization of the LCS problem, is to determine the minimum number of changes, insertions, or deletions to transform a string Y to a string Z. This problem can also be solved in $O(N^2)$ time and space [73]. There have been several efforts to improve these algorithms and/or show lower bounds for the problems. Hirschberg [29] reduced the space for the LCS problem from $O(N^2)$ to $O(N)$ while maintaining the bound of $O(N^2)$ time. Paterson [55] produced an $O(N^2 \log\log N/\log N)$ time algorithm for the LCS problem [30]), which Masek and Paterson [52] have recently improved to $O(N^2/\log N)$ time with their solution to the string editing problem. Hunt and Szymanski [35] have constructed an algorithm that spends $O((R+N) \log N)$ time where R is the total number of pairs of characters at which the two sequences match. Thus, R may be as large as N^2 but if it is small, then the algorithm takes only $O(N \log N)$ time. Wong and Chandra [74] showed an $\Omega(N^2)$ lower (time) bound of the string editing problem for a model of computation allowing only equal / unequal comparisons between elements of strings. Aho, Hirschberg, and Ullman [1] showed the $\Omega(N^2)$ lower bound for the LCS problem (with only equal / unequal comparisons). Hirschberg [31] has produced an $\Omega(N \log N)$ lower bound for the LCS problem where less-than-or-equal comparisons are allowed. Hirschberg [30] provides an overview of the results for the LCS and related problems and Selkow [65] describes an extension of these string problems to trees. Itakura [41] gives an interesting application of the LCS problem to speech recognition. Fitch and Margoliash [18] apply the string editing problem to determination of the mutation distance of the DNA of several species.

We will now present the $O(N^2)$ time and $O(N^2)$ space algorithm for the LCS problem. The two strings are, as before, Y and Z, each consisting of N characters from the alphabet ALPH. The DDP D = (A, S, P, f) is defined by

$A = \{ a_{ij} \}$ where a_{ij} compares y_i and z_j for equality,

$S = \{ e \} \cup \{ a_{ij} \mid i,j \le N \} \cup \{ x a_{ij} a_{k\ell} \mid x \in S$ and $i<k$ and $j<\ell \}$, *! The subscripts of the a_{ij} must be strictly increasing.;*

$P = \{$ 2*N - tuples of elements in the alphabet ALPH $\}$, and

$f(e) = 0$, $f(a_{ij}) = C(y_i,z_j)$, $f(x a_{ij} a_{k\ell}) = f(x a_{ij}) + C(y_i,z_j)$ for $x \in ALPH^*$ and $i<k$ and $j<\ell$, where $C(y_i,z_j) = 1$ if $y_i = z_j$ and $C(y_i,z_j) = 0$ otherwise.

Note that the cost function f is written so that the object is to *maximize* f rather than minimize it. Let the states of the SDP be written S_{ij} and be interpreted "$(y_1,y_2,...y_i)$ has been compared with $(z_1 z_2 ... z_j)$ by a sequence ending in a_{ij}." Thus, the SDP is W = (V, P, h, k) where

$V = (A, Q, q_0, F, \lambda)$ where

A = same as for the DDP,

$Q = \{ S_{ij} \mid i,j = 1,2,...N \} \cup \{ S_{00} \}$,

$q_0 = S_{00}$,

$F = \{ S_{NN} \}$,

$\lambda(S_{ij},a_{k\ell}) = S_{k\ell}$ for $k>i$, $\ell>j$,

P = same as for the DDP,

$h(r,S_{ij},a_{k\ell}) = r + C(y_k,z_\ell)$ for $i<k$, $j<\ell$, and

$k(p) = 0$.

The SDP is monotone by additivity of the cost function. Thus, the functional equations are

$$G(S_{00}) = 0, \text{ and}$$

$$G(S_{ij}) = \max_{\{(S_{kl},a_{ij}) \mid k<i \text{ and } l<j\}} G(S_{kl}) + C(y_i,z_j).$$

This gives an $O(N^4)$ time algorithm. However, a monotone property of the function G can minimize the search for max in the equation above. Since the subscripts of the a_{ij} are required to be strictly increasing in each legal sequence of comparisons, it follows that

If $i < k$ and $j < \ell$, then $G(S_{ij}) \le G(S_{k\ell})$.

Thus, to find $G(S_{ij})$ it is sufficient to compute

$$G(S_{ij}) = \max \left(\left[\max_{k<i} G(S_{k,(j-1)}) \right], \left[\max_{k<j} G(S_{(i-1),k}) \right] \right) + C(y_i,z_j).$$

The time is now reduced to $O(N^3)$ but further improvement is possible. The max(k<i) and the max(k<j) can be propagated throughout the computation of all $G(S_{ij})$ so that only constant time per state S_{ij} is required. One way to do this is to create an array MAXG(i,j) to propagate these max's. Let $G(i,j) = G(S_{ij})$. When G(i,j) is defined, propagate the max's through the steps

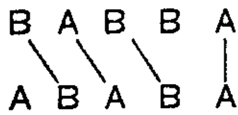
$$\begin{aligned}
&G(i,j) \leftarrow MAXG(i-1,j-1) + C(y_i,z_j)\\
&MAXG(i,j) \leftarrow G(i,j)\\
&MAXG(i+1,j) \leftarrow \max(\ MAXG(i+1,j), G(i,j)\ )\\
&MAXG(i,j+1) \leftarrow \max(\ MAXG(i,j+1), G(i,j)\ ).
\end{aligned}$$

Another way to accomplish the propagation is to make a small change in the definition of G so that it accomplishes the same task as MAXG, thus saving one array [29]. This is what is done in the following algorithm.

Algorithm for the Longest Common Subsequence of Two Strings

Input: integer N > 0, arrays Y[1:N], Z[1:N]
Output: Array G such that G(I,J) = length of longest common subsequence of $(y_1,y_2,...y_i)$ and $(z_1,z_2,...z_j)$. Array H such that H(I,J) = pointer to either (I,J-1) or (I-1,J), whichever has the largest value of G.
Time: $O(N^2)$, Storage: $O(N^2)$

```
! initialization;
      For I ← 0 thru N do
            For J ← 0 thru N do
                  G(I,J) ← H(I,J) ← 0;
! optimization;
      For I ← 1 thru N do
            For J ← 1 thru N do
            begin
                  If Y(I) = Z(J) then
                        G(I,J) ← G(I-1,J-1) + 1
                    else
                        G(I,J) ← max( G(I,J-1), G(I-1,J) );
                  If G(I,J-1) ≥ G(I-1,J) then
                        H(I,J) ← (I,J-1)
                    else
                        H(I,J) ← (I-1,J)
            end;
```



G	B	A	B	B	A
A	0	1	1	1	1
B	1	1	2	2	2
A	1	2	2	2	3
B	1	2	3	3	3
A	1	2	3	3	4

H	B	A	B	B	A
A	0,0	0,0	1,2	1,3	1,4
B	0,0	2,1	2,2	2,3	2,4
A	2,1	3,1	3,2	3,3	3,4
B	3,1	3,2	4,2	4,3	4,4
A	4,1	4,2	4,3	5,3	5,4

Figure 7-1: The matrices G and H produced for the strings ABABA and BABBA.

8. 'Hard' Combinatorial Problems

The problems described in the previous sections can all be solved in $O(N^2)$, $O(N^3)$, or more generally time that is bounded above by a polynomial in N. The problems in this section have never been solved in less than $O(2^N)$ time. Many of these problems are NP-Complete [43, 2]. (The other problems appear to be at least this hard.) For problems where a naive algorithm costs O(N!) time, DP can typically bring an improvement to $O(P(N)*2^N)$ time where P(N) is a polynomial in N. For example, the traveling salesperson problem (TSP) runs in time $O(N^2 2^N)$ [28, 6]. An improved DP algorithm for the TSP and knapsack problems that utilizes branch-and-bound is described by Morin [53]. Constructing optimal decision trees from a set of yes/no tests is NP-complete [37] and the DP algorithm runs in $O(3^N)$ time and space [64]. (Lew [50] describes a solution to a generalization of this problem.) Bayes [4] describes a disk file placement problem - a variation of the module placement problem in Karp and Held [44] - for which a DP algorithm spends $O(2^N)$ time and O(C(N,N/2)) storage. Kohler and Steiglitz [49] and Kohler [48] describe DP (and branch-and-bound) solutions of NP-Complete scheduling problems. Gilmore [23] shows interconnections between cutting stock, linear programming, knapsacking, dynamic programming, and integer programming. Shapiro [67] shows how the knapsack problem can be reduced to a (huge) shortest path problem, and Lloyd [51] describes a DP algorithm for the minimum weight triangulation problem.

There are far too many problems to describe them all in full detail. But to give some of the flavor of what a DP algorithm for this type of problem can be like, we will now describe the traveling salesperson (TSP) problem.

The salesperson must travel to each of N cities, starting and ending at city number one. The distances D_{ij} between the cities are given. The problem is to find the route with minimum total distance traveled. The naive solution is to simply compare all (N-1)! possible routes and choose the minimum. Use of a DP algorithm, however, can reduce this to $O(N^2 2^N)$ time.

The first step in construction of the DP algorithm is to define the DDP D = (A, S, P, f) where

$A = \{ a_i \}$ where a_i means "go to city i",

$S = \{$ permutations of the a_i ending in $a_1 \}$,

$P = \{$ matrices $D_{ij} \mid D_{ij} \geq 0, \forall i,j \}$, and

$f(e) = 0$, $f(a_i) = D_{1i}$, $f(xa_ia_j) = f(xa_i) + D_{ij}$.

To construct the SDP we must first define the states. It is tempting to let the states be S_i where S_i is interpreted "at city i". But since we must travel to all of the N cities, not just the

closest one, this won't work. Instead, we must consider the cities already visited by the time the salesperson reaches city i. Let C = { cities } = { 1, 2, 3, . . . N }. If M is a subset of C representing the cities that have been visited by the salesperson and $i \in M$ is the city where the salesperson is currently located, then the state is $S_{M,i}$. With this definition the SDP is W = (Z, P, h, k) where

Z = (A, Q, q_0, F, λ) where

A = same as for the DDP,

$Q = \{ S_{M,i} \mid M \subset C \text{ and } i \in M \}$,

$q_0 = S_{\{1\},1}$,

$F = \{ S_{C,1} \}$,

$\lambda(S_{M,i}, a_j) = S_{M \cup \{j\},j}$,

P = same as for the DDP,

$h(r, S_{M,i}, a_j) = r + D_{ij}$, and

$k(p) = 0$.

Monotonicity of the SDP follows from the additivity of the cost function. Thus, the functional equations are

$$G(S_{\{1\}}) = 0, \text{ and}$$

$$G(S_{M,i}) = \min_j G(S_{M-\{i\},j}) + D_{ij}.$$

Here is an algorithm to solve the functional equations:

Algorithm for Traveling Salesperson Problem

Input: "Distance" matrix D(I,J), ∀I,J=1,...N
Output: (Let M be a subset of {1,2,...N}.) Array G such that G(M,I) = minimum cost of traveling to the cities of M and ending at city I. Array H such that H(M,I) = last city reached before city I in the optimal subtour corresponding to G(M,I).
Time: $O(N^2 2^N)$, Storage: $O(N 2^N)$ (The analysis is in [28].)

```
! MINIMIZE finds the best subtour for (M,i;)
        procedure MINIMIZE( G, H, D, M, I )
        begin
              MINCOST ← ∞;
              For all J ∈ M-{I} do
              begin
                    COST ← G(M-{I},J) + D(J,I);
                    If COST < MINCOST then
                    begin
                          G(M,I) ← MINCOST ← COST;
                          H(M,I) ← J;
                    end
              end
        end;

! main routine;
        ! Initialization;
        C ← { 2, 3, 4, . . . N };
        For I ← 1 thru N do
              G({I},I) ← D(1,I);
        ! Optimization;
        For K ← 2 thru N-1 do ! K = size of subsets;
              For all M ⊂ C such that |M| = K do
                    For all I ∈ M do
                          MINIMIZE(G,H,D,M,I);
        MINIMIZE(G,H,D,C∪{1},1);
```

9. Conclusion

Dynamic Programming is a useful problem solving method for many problems in computer science. In this paper we have described problems in several categories of DP: Shortest Path Problems, Optimal Parenthesization Problems, Optimal Partition Problems, Optimal Matching Problems, and 'Hard' Combinatorial Problems. The approach taken in this paper toward solution of these problems is a modification of the treatment in Karp and Held [44]. Each analysis of a problem proceeds in the following steps:

1. Produce the discrete decision process for the problem,
2. Decide what are the relevant *states* and construct the corresponding sequential decision process,
3. Check the monotonicity conditions and (if satisfied) produce the functional equations, and
4. From the functional equations construct a dynamic programming algorithm that solves the problem.

In this paper we have also presented new problems and results including optimizing the search for N order statistics and a more space efficient version of Fisher's algorithm for an optimal partition.

Acknowledgments

Jon Bentley suggested dynamic programming as the topic for my area qualifier and has given numerous helpful comments that have improved the quality of this paper.

References

[1] A.V. Aho, D.S. Hirschberg, and J.D. Ullman, *Bounds on the Complexity of the Longest Common Subsequence Problem*, J. Assoc. Comp. Mach., 23 (1976), pp. 1-12.

[2] A.V. Aho, J. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974. Constructing optimal binary search tree, pp. 119-123..

[3] J.K. Baker, *The DRAGON System - An Overview*, IEEE Trans. Acoust. Speech Signal Process., ASSP-23 (1975), pp. 24-29.

[4] A.J. Bayes, *The Optimised Placing of Files on Disk Using Dynamic Programming*, Optimization, R.S. Anderssen, L.S. Jennings, and D.M. Ryan, ed., University of Queensland Press, St. Lucia, Queensland, 1972, pp. 182-186.

[5] R. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, New Jersey, 1957.

[6] R. Bellman, *On a routing problem*, Quart. Appl. Math., 16 (1958), pp. 87-90.

[7] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan, *Time bounds for selection*, J. Comput. System Sci., 7 (1972), pp. 448-461.

[8] D.W. Boyd, *Long Division: An Example of Dynamic Programming*, Amer. Inst. Indust. Engin. Trans., volume (1974), pp. 365-366.

[9] C. Cafforio and F. Rocca, *Methods for Measuring Small Displacements of Television Images*, IEEE Trans. Information Theory, IT-22 (1976), pp. 573-579.

[10] F.Y. Chin, *An O(N) Algorithm for Determining a Near-Optimal Computation Order of Matrix Chain Products*, Comm. ACM, 21 (1978), pp. 544-549.

[11] G.B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey, 1963.

[12] E.W. Dijkstra, *A Note on Two Problems in Connection With Graphs*, Numer. Math., 1 (1959), pp. 269-271.

[13] S.E. Dreyfus and A.M. Law, *The Art and Theory of Dynamic Programming*, Academic Press, New York, 1977.

[14] G.T. Duncan, *Optimal Diagnostic Questionnaires*, Operations Res., 23 (1975), pp. 22-32.

[15] J. Early, *An efficient context-free parsing algorithm*, Comm. ACM, 13 (1970), pp. 94-102.

[16] M.A. Fischler and R.A. Elschlager, *The Representation and Matching of Pictorial Structures*, IEEE Trans. Computers, C-22 (1973), pp. 67-92.

[17] W.D. Fisher, *On grouping for maximum homogeneity*, J. Amer. Statist. Assoc., 53 (1958), pp. 789-798.

[18] W.M. Fitch and E. Margoliash, *Construction of Phylogenetic Trees*, Science, 155 (1967), pp. 279-284.

[19] G.D. Forney Jr., *The Viterbi Algorithm*, Proc. IEEE, 61 (1973), pp. 268-278.

[20] M.L. Fredman, *On the Decision Tree Complexity of the Shortest Path Problems*, 16th Annual Symposium on Foundations of Computer Science, IEEE, Long Beach, CA., Oct., 1975, pp. 98-99.

[21] M. Gardner, *Concerning the game of nim and its mathematical analysis*, Mathematical Games section of Scientific American, 198 (1958), pp. 104-111.

[22] E.N. Gilbert and E.F. Moore, *Variable length encodings*, Bell System Tech. J., 38 (1959), pp. 933-968.

[23] P. Gilmore, *Cutting stock, Linear Programming, knapsacking, Dynamic Programming, Integer Programming: some interconnections*, Tech. Rep. RC 6528 (28188), IBM T.J. Watson Research Center, Yorktown Heights, New York, Sept 1977. 33 pages.

[24] S.L. Graham, M.A. Harrison, and W.L. Ruzzo, *On Line Context Free Language Recognition in Less Than Cubic Time*, 8th Annual ACM Symposium on Theory of Computing, Assoc. Comput. Mach. Special Interest Group on Automata and Computing Theory, New York, May, 1976, pp. 112-120.

[25] L.J. Guibas, H.T. Kung, and C.D. Thompson, *Direct VLSI Implementation of Combinatorial Algorithms*, Conference on Very Large Scale Integration, Industrial Associates, California Institute of Technology, Jan., 1979,

[26] F. Harary, *Graph Theory*, Addison-Wesley Publishing Company, Reading, Mass., 1969.

[27] J.A. Hartigan, *Clustering Algorithms*, John Wiley & Sons, New York, 1975. Fisher Algorithm on pages 130-140.

[28] M. Held and R.M. Karp, *A Dynamic Programming Approach to Sequencing Problems*, J. SIAM, 10 (1962), pp. 196-210.

[29] D.S. Hirschberg, *A Linear Space Algorithm for Computing Maximal Common Subsequences*, Comm. ACM, 18 (1975), pp. 341-343.

[30] D.S. Hirschberg, *Complexity of Common Subsequence Problems*, Springer-Verlag, New York, 1977. Lecture Notes in Computer Science,No. 56, Fundamentals of Computation Theory.

[31] D.S. Hirschberg, *An information - theoretic lower bound for the longest common subsequence problem*, Information Processing Lett., 7 (1978), pp. 40-41.

[32] J.E. Hopcroft and J.D. Ullman, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

[33] T.C. Hu, *A new proof of the T - C algorithm*, SIAM J. Appl. Math., 25 (1973), pp. 83-94.

[34] D.A. Huffman, *A Method for the Construction of Minimum Redundancy Codes*, Proc. Inst. Radio Engin., 40 (1952), pp. 1098-1101.

[35] J.W. Hunt and T.G. Szymanski, *A Fast Algorithm for Computing Longest Common Subsequences*, Comm. ACM, 20 (1977), pp. 350-353.

[36] T.C. Hu and A.C. Tucker, *Optimum Binary Search Trees*, SIAM J. Appl. Math., 21 (1971), pp. 514-532.

[37] L. Hyafil and R.L. Rivest, *Constructing Optimal Binary Decision Trees is NP-Complete*, Information Processing Lett., 5 (1976), pp. 15-17.

[38] T. Ibaraki, *Minimal Representations of Some Classes of Dynamic Programming*, Information and Control, 27 (1975), pp. 289-328.

[39] T. Ibaraki, *Branch-and-bound procedure and state-space representation of combinatorial optimization problems*, Information and Control, 36 (1978), pp. 1-27.

[40] A. Itai, *Optimal alphabetic trees*, SIAM J. Comput., 5 (1976), pp. 9-18.

[41] F. Itakura, *Minimum Prediction Residual Principle Applied to Speech Recognition*, IEEE Trans.Acoust. Speech Signal Process., ASSP-23 (1975), pp. 67-72.

[42] D.B. Johnson, *Algorithms for shortest paths*, Ph.D. Thesis, Dept. of Computer Science, Cornell University, Ithaca, New York, 1973. Cited in Aho, Hopcroft, and Ullman.

[43] R.M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, Miller and Thatcher, ed., Plenum Press, New York, 1972, pp. 85-104.

[44] R.M. Karp and M. Held, *Finite-State Processes and Dynamic Programming*, SIAM J. Applied Math, 15 (1967), pp. 693-718.

[45] T. Kasami and K. Torii, *A syntax-analysis procedure for unambiguous context-free grammars*, J. Assoc. Comp. Mach., 16 (1969), pp. 423-431.

[46] D.E. Knuth, *Optimal Binary Search Trees*, Acta Informat., 1 (1971), pp. 14-25.

[47] D.E. Knuth, *Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973. Algorithm for binary search tree in Section 6.2.2.

[48] W.H. Kohler, *Computational Experience with Efficient Exact and Approximate Algorithms for an NP-Complete Scheduling Problem*, Proceedings of the Ninth Hawaii International Conference on System Sciences, University of Hawaii, Manoa, 1976, pp. 116-118.

[49] W.H. Kohler and K. Steiglitz, *Enumerative and Iterative Computational Approaches*, Computer and Job-Shop Scheduling Theory, E.G. Coffman, Jr., ed., John Wiley & Sons, New York, 1976, pp. 229-287.

[50] A. Lew, *Optimal Conversion of Extended-Entry Decision Tables with General Cost Criteria*, Comm. ACM, 21 (1978), pp. 269-279.

[51] E.L. Lloyd, *On triangulations of a set of points in the plane*, 18th Annual Symposium on Foundations of Computer Science, IEEE, Long Beach, CA, Oct.31-Nov.2, 1977, pp. 228-240. Also available as MIT Technical Report MIT/LCS/TM-88.

[52] W.J. Masek and M.S. Paterson, *A Faster Algorithm Computing String Edit Distances*, Tech. Rep. MIT/LCS/TM-105, MIT Laboratory for Computer Science, Cambridge, Mass., May 1978.

[53] T.L. Morin, *Branch and Bound Strategies for Dynamic Programming*, Operations Res., 24 (1976), pp. 611-627.

[54] V.Y. Pan, *An introduction to the trilinear technique of aggregating, uniting and cancelling and applications of the technique for constructing fast algorithms for matrix operations,* Proceedings of the Nineteenth Annual Symposium on the Foundations of Computer Science, IEEE, Long Beach, CA, 1978,

[55] M.S. Paterson, University of Warwick, England.

[56] H.J. Payne and W.S. Meisel, *An Algorithm for Constructing Optimal Binary Decision Trees,* IEEE Trans. Computers, C-26 (1977), pp. 905-916.

[57] Y. Perl, *Average Analysis of Simple Path Algorithms,* Tech. Rep. UIUCDCS-R-77-905 UILU-ENG 77 1759, Dept. of Computer Science, University of Illinois, Urbana-Champaign, Nov. 1977.

[58] J.L. Peterson, J.R. Bitner, and J.H. Howard, *The Selection of Optimal Tab Settings,* Comm. ACM, 21 (1978), pp. 1004-1006.

[59] I. Pohl, *A sorting problem and its complexity,* Comm. ACM, 15 (1972), pp. 462-464.

[60] M. Pollack and W. Wiebenson, *Solutions of the shortest route problem - A review,* Operations Res., 8 (1960), pp. 224-230.

[61] F.P. Preparata and S.R. Ray, *An approach to artificial symbolic cognition,* Information Sci., 4 (1972), pp. 65-86.

[62] R.L. Probert, *An Extension of Computational Duality To Sequences of Bilinear Computations,* SIAM J. Comput., 7 (1978), pp. 91-98.

[63] A. Schonhage, M. Paterson, and N. Pippenger, *Finding the median,* J. Comput. System Sci., 13 (1976), pp. 184-189.

[64] H. Schumacher and K.C. Sevcik, *The Synthetic Approach to Decision Table Conversion,* Comm. ACM, 19 (1976), pp. 343-351. See also the Technical Correspondence section of the C.ACM 21(1978), pp. 179-180.

[65] S.M. Selkow, *The tree-to-tree editing problem,* Information Processing Lett., 6 (1977), pp. 184-186.

[66] M.I. Shamos, K.Q. Brown, J. Saxe, and B.W. Weide, Clustering in One Dimension, In preparation.

[67] J.F. Shapiro, *Shortest Route Methods for Finite State Space Deterministic Dynamic Programming Problems,* SIAM J., 16 (1968), pp. 1232-1250.

[68] R.E. Tarjan, *Complexity of Combinatorial Algorithms,* SIAM Review, 20 (1978), pp. 457-491.

[69] M.E. Thomas, *A Survey of the State of the Art in Dynamic Programming,* Amer. Inst. of Indust. Engin., 8 (1976), pp. 59-69.

[70] L.G. Valiant, *General Context-Free Recognition in Less than Cubic Time,* J. Comput. System Sci., 10 (1975), pp. 308-315.

[71] A.J. Viterbi, *Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,* IEEE Trans. Information Theory, IT-13 (1967), pp. 260-269.

[72] H.M. Wagner, *Principles of Operations Rescarch*, Prentice-Hall, Englewood Cliffs, N.J., 1969.

[73] R.A. Wagner and M.J. Fischer, *The string-to-string correction problem*, J. Assoc. Comp. Mach., 21 (1974), pp. 168-173.

[74] C.K. Wong and A.K. Chandra, *Bounds for the String Editing Problem*, J. Assoc. Comp. Mach., 23 (1976), pp. 13-16.

[75] D.H. Younger, *Recognition and parsing of context-free languages*, Infomation and Control, 10 (1967), pp. 189-208.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Carnegie Mellon University Libraries
3 8482 01022 4034

JUN 30 1999

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Value
1. REPORT NUMBER	CMU-CS-79-106
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	DYNAMIC PROGRAMMING IN COMPUTER SCIENCE
5. TYPE OF REPORT & PERIOD COVERED	Interim
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Kevin Q. Brown
8. CONTRACT OR GRANT NUMBER(s)	N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	Office of Naval Research Arlington, VA 22217
12. REPORT DATE	February 1979
13. NUMBER OF PAGES	49
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	Same as above
15. SECURITY CLASS. (of this report)	UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report)	Approved for public release; distribution unlimited
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	
18. SUPPLEMENTARY NOTES	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)	
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)	

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)