SIAM J. ALG. DISC. METH.
Vol. 3, No. 4, December 1982

SPEED-UP IN DYNAMIC PROGRAMMING*

F. FRANCES YAO†

Abstract. Dynamic programming is a general problem-solving method that has been used widely in many disciplines, including computer science. In this paper we present some recent results in the design of efficient dynamic programming algorithms. These results illustrate two approaches for achieving efficiency: the first by developing general techniques that are applicable to a broad class of problems, and the second by inventing clever algorithms that take advantage of individual situations.

1. Introduction. Dynamic programming is a general problem-solving technique that has been used widely in operations research, economics, control theory and, more recently, computer science. The present paper will be oriented toward the use of dynamic programming as a paradigm for designing algorithms in computer science. As computational efficiency is a major goal in algorithm design, we will be interested in techniques which allow us to speed up algorithms produced by straightforward dynamic programming. There are two promising directions for such research, namely, the development of general techniques that are applicable to a large class of problems, and the invention of efficient algorithms for specific problems by taking advantage of their special properties. In this paper we give a review of some recent progress in these directions. In § 2, we discuss a general speed-up technique that can be applied to dynamic programming problems when the cost function satisfies certain restrictions known as the Quadrangle Inequalities. In § 3, we give an improved algorithm for finding the optimal order of multiplying a sequence of matrices.

We will not give proofs for the theorems cited in this paper. For proofs as well as further discussions, the reader is referred to [8] for the topic considered in § 2, and [4], [9] for the topic considered in § 3.

2. Quadrangle inequalities.

Example 1. Given a set of points X on the plane, how do we find five points that span a pentagon with maximum perimeter?

A natural solution based on dynamic programming would be to seek out maximum triangles, maximum quadrilaterals, and maximum pentagons in turn. It is not difficult to argue that we can restrict our consideration to the extreme points of X. Therefore let us assume the convex hull of X to be $P=\langle v_1, v_2, \cdots, v_n\rangle$, and the distance between v_i and v_j to be d_{ij}. Then maximum triangles can be found by computing the largest entry in the matrix $D+D\otimes D$, where $D=(d_{ij})$, and $\otimes$ denotes the (max, +)-multiplication of two matrices defined by

$$F\otimes G=(p_{ij}),\quad \text{where } p_{ij}=\max\{f_{ik}+g_{kj} \mid i\leqq k\leqq j\} \text{ for } F=(f_{ij}) \text{ and } G=(g_{ij}).$$

Since $\otimes$ is associative, we will write D^2 for $D\otimes D$, and D^t for $D^{t-1}\otimes D$. A maximum pentagon then corresponds to a maximum entry in $D+D^4$, where D^4 may be evaluated as $D^2\otimes D^2$. In general, a maximum t-gon can be found by first computing D^{t-1} and then finding a maximum entry in $D+D^{t-1}$. Since D^{t-1} can be obtained from D in $O(\log t)$(max, +)-multiplications (see [7, § 4.6.3], for example) at a cost of $O(n^3)$ steps per matrix multiplication, the answer can be obtained in $O(n^3 \log t)$ steps.

Now we pose the question: Can $D\otimes D$ be computed in time faster than $O(n^3)$? It turns out that, by properties of the Euclidean metric d_{ij}, if we let $K(i,j)$ denote

* Received by the editors February 25, 1982.

† Xerox Palo Alto Research Center, Palo Alto, California 94304.

$\max\{k \mid d_{ik}+d_{kj}=(D\otimes D_{ij}\}$, then $K(i,j)$ is a monotone function of i and j (see Theorem 1).

CLAIM 1. $K(i,j)\leqq K(i,j+1)\leqq K(i+1,j+1)$.

This property enables us to limit our search for the optimal k, while computing $(D\otimes D)_{i,j+1}=\max\{d_{i,k}+d_{k,j+1} \mid i\leqq k\leqq j+1\}$, to those k that lie between $K(i,j)$ and $K(i+1,j+1)$, provided that the latter two values are already known. This suggests computing $(D\otimes D)_{ij}$ by diagonals, in order of increasing values of $j-i$. The cost for computing all entries of one diagonal is $O(n)$ as a result of Claim 1 and the total cost for obtaining $D\otimes D$ is thus only $O(n^2)$.

More generally, when one forms the product $D^r\otimes D^s$ for any $r\geqq 1$ and $s\geqq 1$, monotonicity properties analogous to Claim 1 also hold (Theorems 1 and 2). This implies that our earlier dynamic programming algorithm for finding maximum t-gons can be speeded up from $O(n^3\log t)$ to $O(n^2\log t)$.

The critical property of the Euclidean distance function d_{ij} that makes Claim 1 true is what we call the "quadrangle inequalities". We say that a real-valued function $f(i,j)$, where $1\leqq i\leqq j\leqq n$, satisfies *convex quadrangle inequalities* (*convex* QI) if

$$f(i,k)+f(j,l)\geqq f(i,l)+f(j,k)\quad \text{for } i\leqq j\leqq k\leqq l.$$

The same inequalities with signs reversed are called *concave quadrangle inequalities* (*concave* QI):

$$f(i,k)+f(j,l)\leqq f(i,l)+f(j,k)\quad \text{for } i\leqq j\leqq k\leqq l.$$

Example 2. It is easy to see that the distance function d_{ij} for vertices of a convex polygon in Example 1 satisfies the convex QI. Some other examples of functions are given below.

$$\left.\begin{aligned} f(i,j)&=a_i+a_{i+1}+\cdots+a_j\\ f(i,j)&=a_i+a_{i+1}+\cdots+a_{j-1}\\ f(i,j)&=a_{i+1}+a_{i+2}+\cdots+a_{j-1}\end{aligned}\right\}\quad \text{all satisfy both concave QI and convex QI;}$$

$$f(i,j)=a_i\cdot a_{i+1}\cdot\cdots\cdot a_j\qquad \text{satisfies concave QI if all } a_k\text{'s are} \geqq 1.$$

Furthermore, convex QI are preserved by convex, nondecreasing mappings (for example, $\log d_{ij}$ satisfies convex QI); while concave QI are preserved by concave, nondecreasing mappings (for example, $f^2(i,j)$ satisfies concave QI for any of the four f's defined above). Additional QI-preserving mappings that are of particular importance to dynamic programming will be discussed in Theorem 2 and 3 below.

Our earlier Claim is derived from the following general theorem. Let $K_{f\otimes g}(i,j)$ denote $\max\{k \mid f(i,k)+g(k,j)=(f\otimes g)(i,j)\}$; that is, $K_{f\otimes g}(i,j)$ is the largest index k for which $f(i,k)+g(k,j)$ achieves the maximum. For simplicity, we will write $K(i,j)$ for $K_{f\otimes g}(i,j)$ whenever the context $f\otimes g$ is understood.

THEOREM 1. *If both f and g satisfy convex* QI, *then $K_{f\otimes g}(i,j)$ is a monotone function of i and j*:

$$K(i,j)\leqq K(i+1,j)\leqq K(i+1,j+1).$$

As we saw in Example 1, the above theorem allows us to compute $K_{f\otimes g}$ and $f\otimes g$ with a cost of only $O(n)$ per diagonal, thus $O(n^2)$ in total.

COROLLARY A. *If both f and g satisfy convex* QI, *then $f\otimes g$ and $K_{f\otimes g}$ can be computed in $O(n^2)$ time and space.*

The above results regarding convex QI and maximization problems have parallels in concave QI and minimization problems. Define

$$f \oslash g(i,j) = \min \{f(i,k) + g(k,j) | i \leqq j \leqq k\}.$$

COROLLARY B. *If both f and g satisfy concave* QI, *then $f \oslash g$ and $K_{f \oslash g}$ can be computed in $O(n^2)$ time and space.*

The following theorem allows us to apply these corollaries iteratively, in situations such as Example 1.

THEOREM 2. *If both f and g satisfy convex* QI, *then $f \otimes g$ also satisfies convex* QI. *If both f and g satisfy concave* QI, *then $f \oslash g$ also satisfies concave* QI.

We also find the concepts of QI useful in the evaluation of recurrence relations involving either minimization or maximization operations. We will mention one such result for concave QI.

A function $w(i,j)$ where $i \leqq j$ is said to be *monotone* if it is monotonically increasing on the lattice of intervals (ordered by inclusion), i.e.,

$$w(i,j) \leqq w(i',j') \quad \text{if } [i,j] \subseteq [i',j'].$$

THEOREM 3. *Let $c(i,j)$, where $i \leqq j$, be defined by*

$$(1) \qquad \begin{aligned} c(i,j) &= w(i,j) + \min_{i<j\leqq k} [c(i,k-1) + c(k,j)] \quad \text{if } i<j, \\ c(i,i) &= a(i). \end{aligned}$$

If w satisfies concave QI *and is monotone, then c satisfies concave* QI.

In consequence, we have the following speed-up result analogous to Theorem 1 and its corollaries.

COROLLARY. *For a function $c(i,j)$ satisfying the description of Theorem* 3, *we can compute $K_{c \oslash c}(i,j)$ and $c(i,j)$ for $0 \leqq i \leqq j \leqq n$ in $O(n^2)$ time and space.*

Example 3. A bookstore is interested in organizing its index files in a way to facilitate look-ups. Take the subject index for example. Suppose that the index, alphabetically ordered, consists of a number of key subjects such as {ART, COOKING, $\cdots$, TRAVEL}, plus other subjects that fall in the intervals in between, namely {A-ART, ART-COOKING, $\cdots$}. We will denote the key subjects by $\{K_1, K_2, \cdots, K_n\}$, and the intervals by $\{I_0, I_1, \cdots, I_n\}$. Assume that the access probability for key K_i is p_i, and that for interval I_j is q_j. We would like to build a binary tree, with the K_j's as internal nodes, and the I_j's as external nodes, such that the expected

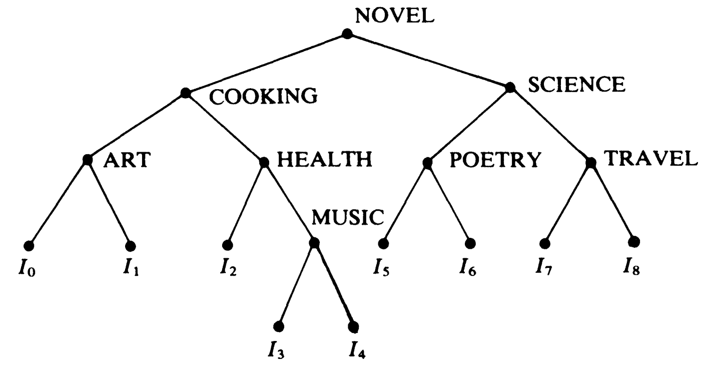

FIG. 1. *A binary search tree.*

number of comparisons in looking up a subject, namely

$$\sum_{1\leqq i\leqq n} p_i(1+\text{level of } K_i)+\sum_{0\leqq j\leqq n} q_j(\text{level of } I_j)$$

is minimized (Fig. 1).

Since all subtrees of an optimal tree must themselves be optimal, this problem can be solved by dynamic programming. One naturally arrives at recurrence relations of the form (1), with $c(i,j)$ being the minimum cost of a subtree for keys $\{K_{i+1},\cdots,K_j\}$ and intervals $\{I_i,\cdots,I_j\}$, and

$$(2)\qquad \begin{aligned} w(i,j)&=p_{i+1}+\cdots+p_j+q_i+\cdots+q_j,\\ a(i)&=0.\end{aligned}$$

The cost of the optimal tree that we are interested in is $c(0,n)$. As noted in Example 2, the function $w(i,j)$ in (2) satisfies concave QI. Therefore by the corollary to Theorem 3, we can compute the values of $c(i,j)$ in $O(n^2)$ time and space. Furthermore, once $c(0,n)$and $K_{c\oslash c}(0,n)$ are found, we can then trace the information in $K_{c\oslash c}(i,j)$ "from top down" to obtain the actual construction of an optimal binary tree in $O(n)$ steps.

Remarks. The problem of optimal binary search trees discussed above is a classical example of dynamic programming in the computer science literature. The original $O(n^3)$ solution by setting up the recurrence relations (1) was due to Gilbert and Moore [3]. Then Knuth [5] showed that the algorithm can be speeded up to $O(n^2)$ by proving that $K_{c\oslash c}(i,j)$ is monotone. However, his proof of monotonicity was given for the particular $w(i,j)$ as defined by (2), and thus not apparently generalizable. For the problem considered in Example 1, some recent results can be found in [2].

3. Multiplying a sequence of matrices. We now turn to another example of a classical dynamic programming algorithm [1] which saw much notable progress lately.

Example 4. Let $M_1, M_2,\cdots,M_n$ be n matrices of dimensions $d_1\times d_2$, $d_2\times d_3,\cdots,d_n\times d_{n+1}$, respectively. What is the optimal order, by multiplying two matrices at a time, for evaluating the product $M_1\times M_2\times\cdots\times M_n$?

To be more specific, let us assume that the cost for multiplying a $p\times q$ matrix with a $q\times r$ matrix is pqr. Consider, for example, four matrices $M_1,\cdots,M_4$ of dimensions 100×1, 1×50, 50×20 and 20×1. Evaluating their product in the left-to-right order $((M_1\times M_2)\times M_3)\times M_4$ would cost 125,000 operations, while the minimum cost, achieved by $M_1\times((M_2\times M_3)\times M_4)$, is only 2,200.

Using dynamic programming, a solution to this problem can be obtained by defining $c(i,j)$ to be the minimum cost for evaluating $M_i\times M_{i+1}\times\cdots\times M_j$, and setting up the recurrence relations

$$\begin{aligned} c(i,j)&=\min_{i<k\leqq j}[c(i,k-1)+c(k,j)+d_id_kd_j]\quad \text{if } i<j,\\ c(i,i)&=O.\end{aligned}$$

This gives an $O(n^3)$ algorithm for computing $c(1,n)$. Can we do any better? As our tools based on quadrangle inequalities (Theorem 3) do not apply to recurrence relations of the present form, we must come up with a different technique.

In the following, we will first develop a geometric representation for the problem. Then, by looking at the case $n=3$, we will extrapolate some simple properties of the optimal solution. We then show how these properties can be utilized to lead to an $O(n^2)$ dynamic programming algorithm.

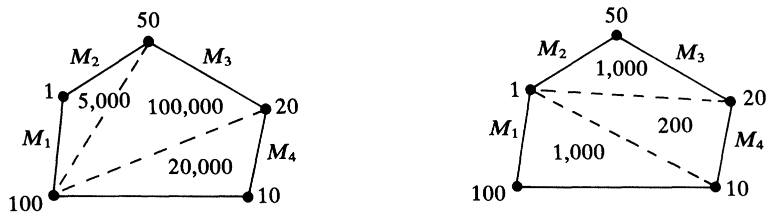

FIG. 2. *Geometric representation for the evaluation of a matrix chain.*

We will use the vertices of an $(n+1)$-sided convex polygon in the plane to represent the $n+1$ parameters $\langle d_1, d_2, \cdots, d_{n+1}\rangle$. A directed edge from d_i to d_j, where $i<j$, will be interpreted as a matrix of dimension $d_i \times d_j$, representing the product $M_i \times M_{i+1} \times \cdots \times M_{j-1}$. Thus, the $n+1$ sides of the convex polygon correspond to the n input matrices and the final product, while any chord represents a potential partial product. It is easy to see that there is a one-to-one correspondence between the different ways of parenthesizing $M_1 \times M_2 \times \cdots \times M_n$ and the possible ways of triangulating the polygon $\langle d_1, d_2, \cdots, d_{n+1}\rangle$. If we associate a cost of $d_i d_k d_j$ with a triangle whose vertices are labeled d_i, d_k and d_j, then our original problem becomes the problem of finding an optimal triangulation of the polygon $\langle d_1, d_2, \cdots, d_{n+1}\rangle$. Figure 2 illustrates the triangulations corresponding to the two different ways of evaluating $M_1 \times M_2 \times M_3 \times M_4$ mentioned earlier.

From now on, we will refer to the d_i's as *weights*. Let $w_1 \leqq w_2 \leqq \cdots \leqq w_n$ be the weights of an n-sided convex polygon P sorted into nondecreasing order. (The ordering may not be unique as some of the weights may be equal; we assume that a particular ordering is chosen and remains fixed.) We will use $w_i w_j$ to denote a directed edge from w_i to w_j, and $w_i w_j w_k$ to denote a triangle with vertices w_i, w_j and w_k, when there is no ambiguity to these notations. We will also use the term partition interchangeably with triangulation.

Consider the case of a quadrilateral. If w_1 and w_2 face each other, then the arc $w_1 w_2$ gives us an optimal partition. This is so because

$$w_1 \cdot w_2 \cdot w_4 + w_1 \cdot w_2 \cdot w_3 \leqq w_2 \cdot w_3 \cdot w_4 + w_1 \cdot w_3 \cdot w_4,$$

or

$$1/w_3 + 1/w_4 \leqq 1/w_1 + 1/w_2.$$

Similarly, if w_1 faces w_3, then $w_1 w_3$ is an optimal partition, because

$$1/w_2 + 1/w_4 \leqq 1/w_1 + 1/w_3.$$

On the other hand, if w_1 faces w_4, then either $w_1 w_4$ or $w_2 w_3$ could be optimal.

The above generalizes to an n-gon by an inductive argument.

LEMMA 1. *Let P be an n-gon with weights $w_1 \leqq w_2 \leqq \cdots \leqq w_n$. Then there exists an optimal partition π for which the following is true.*

(a) *w_1 and w_2 are adjacent (either by a side edge or by a chord); similarly for w_1 and w_3.*

(b) *if both $w_1 w_2$ and $w_1 w_3$ are side edges, then either $w_1 w_4$ or $w_2 w_3$ exists as a chord.*

Lemma 1 implies that we can set up the following recursive procedure for finding an optimal triangulation. We use P_{ij} to denote the subpolygon of P consisting of those vertices lying between w_i and w_j in a clockwise traversal.

```
    PROCEDURE Partition [$P$]
begin
    if $|P| = 1$ or 2 then return $\varnothing$
    else
        if $P$ is a triangle then return $P$
    else
        if $w_1$ and $w_2$ are not adjacent then return Partition [$P_{1,2}$] $\cup$ Partition [$P_{2,1}$]
    else
        if $w_1$ and $w_3$ are not adjacent then return Partition [$P_{1,3}$] $\cup$ Partition [$P_{3,1}$]
    else
        return better of {Partition [$P_{2,3}$] $\cup$ Partition [$P_{3,2}$],
                          Partition [$P_{1,4}$] $\cup$ Partition [$P_{4,1}$]};
    end.
```

As it is, this recursive algorithm requires exponential time, since in the worst case the last **else** clause could generate two problems of size $n-c$ for some constant c. We will show that, however, the total number of calls on *distinct* subpolygons $\{Q\}$ is bounded by $O(n^2)$. Furthermore, these $O(n^2)$ subpolygons can be ordered in such a way that in computing Partition[Q], solutions to its subproblems are already available. In other words, one can turn Partition into a dynamic programming algorithm with an $O(n^2)$ space and time bound. To this end, we need a characterization of those chords w_iw_j in the original polygon P that may arise as w_2w_3 in some recursive call spawned by Partition [P].

DEFINITION. A (directed) chord w_iw_j of P is called a *bridge*, if all weights w_k in P_{ij} satisfy $k \geqq \max\{i, j\}$.

Note that both w_1w_2 and w_2w_1 are bridges, and it is the only instance where two bridges correspond to the same (undirected) edge. The side edges of P may be viewed as degenerate bridges, henceforth we will include them in the definition for convenience.

It is easy to check that bridges have the following properties:

1. Two bridges never intersect (except possibly at the endpoints); therefore there are at most $O(n)$ bridges.

2. A partial order $<$ can be imposed on the set of bridges if we define $w_{i'}w_{j'} < w_iw_j$ to mean $P_{i'j'} \subseteq P_{ij}$.

3. The transitive reduction of $<$ (i.e., the subgraph of $<$ with all edges implied by transitivity removed) is a forest, for $a<b$ and $a<c$ imply that b and c are comparable in $<$. We shall denote this forest by $T[<]$. Note that w_1w_2 and w_2w_1 are the two roots of $T[<]$, and the leaves are the degenerate bridges (sides) of P.

4. Any nonleaf node w_iw_j of $T[<]$ has exactly two sons, namely w_iw_k and w_kw_j where k is the smallest index (aside from i and j) in P_{ij}; we will refer to them respectively, assuming $i<j$, as the *minson* and the *maxson* of w_iw_j. Thus $T[<]$ is actually the union of two binary trees. Figure 3 gives an example of a polygon P and the corresponding $T[<]$.

Procedure MarkBridges below identifies and outputs the bridges of P as it makes one clockwise scan of the weights. The bridges are actually generated in (slightly modified) postorder [6] of the tree $T[<]$; therefore, in particular, they are topologically sorted into a nondecreasing order consistent with $<$. The procedure employs a stack S, and we use the notations $S \Leftarrow x$ abd $x \Leftarrow S$ for pushing and popping as defined in [6].

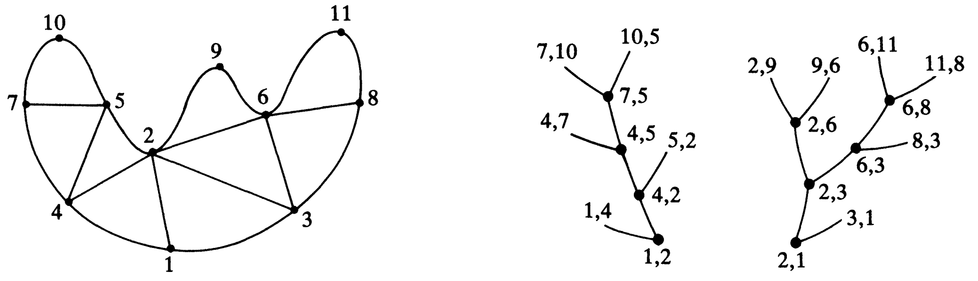

FIG. 3. *A polygon P and the corresponding forest* $T[<]$. (*The weights are represented by their indices only.*)

```
PROCEDURE MarkBridges [$P$];
begin
    find the minimum weight $w_1$;
    $w \leftarrow w_1$;
    repeat
        begin
            $S \Leftarrow w$;
            $w \leftarrow$ nextweight;                —Going clockwise from $w_1$.
            while top($S$) $> w$ do
                begin $t \Leftarrow S$;
                    output (top($S$), $t$) and ($t$, $w$) as bridges;
                end;
        end
    until $w = w_1$;                              —Halt after returning to $w_1$.
end.
```

DEFINITION. A subpolygon Q of P is called a *cone*, if $Q = P_{ij} \cup w_i w_j w_k$ where $b = w_i w_j$ is a bridge of P, and $k \leqq \min\{i, j\}$. We also denote a cone Q by (b, w_k) (Fig. 4).

In particular, P_{ij} for any bridge $w_i w_j$ is a cone, and P itself is the union of two cones $P_{1,2}$ and $P_{2,1}$. The existing partial orders on bridges and on weights induce a natural alphabetic order on cones.

DEFINITION. We say that a cone $Q' = (b', w_{k'})$ *precedes* a cone $Q = (b, w_k)$ if either (1) $b' < b$, or (2) $b' = b$ and $k' \geqq k$.

LEMMA 2. *Any subpolygon that may arise in the execution of* Partition $[Q]$, *for a cone* $Q = (b, w_k)$, *is either a triangle or a cone* Q' *which precedes* Q.

Thus we can use dynamic programming to compute and tabulate the solutions to all cones in accordance with their precedence order. The actual recurrence relations

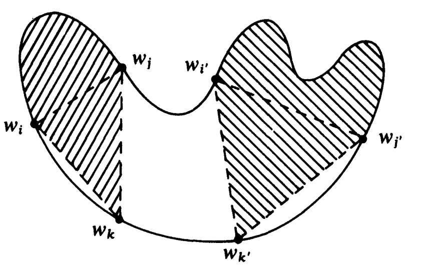

FIG. 4. *Example of two cones* (*the shaded regions*).

have been incorporated into the following program. We use Partition $[Q]$ to refer to the table entry containing the optimal solution to cone Q. The outer **for** loop iterates over all b in the order as they are generated by MarkBridges, while the inner **for** loops iterate over all w_k with $k \leqq i$ in decreasing order. The algorithm runs in $O(n^2)$ time, as there are at most $2n$ bridges, and at most n cones for a given bridge.

```
PROCEDURE DP-Partition [$P$]
begin
    for $b = w_i w_j \in B$ do                 —$B$ is the output of Markbridges [$P$].
    begin                                      —Assume that $i < j$.
        if $b$ is a leaf then
            for all cones $Q = (b, w_k\rangle$ with $k \leqq i$ do
                if $w_k = w_i$ then Partition [$Q$] ← $\emptyset$          —$Q = w_i w_j$
                               else Partition [$Q$] ← $Q$;                 —$Q = w_i w_j w_k$
        if $b$ is not a leaf then
            for all cones $Q = (b, w_k\rangle$ with $k \leqq i$ do
                if $w_k = w_i$ then Partition [$Q$] ← Partition[(minson ($b$), $w_i\rangle$] ∪
                                                       Partition[(maxson ($b$), $w_i\rangle$]
                               else Partition [$Q$]
                               ← better of {Partition [$(b, w_i\rangle$] ∪ $w_i w_j w_k$,
                                            Partition[(minson ($b$), $w_k\rangle$] ∪
                                            Partition [(maxson ($b$), $w_k\rangle$]};
    end;
    Partition [P] ← Partition [$P_{1,2}$] ∪ Partition [$P_{2,1}$];
end.
```

Remark: In 1980, Hu and Shing [4] gave an $O(n \log n)$ algorithm for solving this problem. However, their presentation is exceedingly long; a more concise exposition, including the preceding algorithm, can be found in Yao [9].

4. Conclusions. We surveyed some recent results in the design of dynamic programming algorithms. These results illustrate two approaches for obtaining speed-up in dynamic programming: one general and the other problem specific. In the first case, the quadrangle inequalities provide a type of sufficient conditions by which speed-up is guaranteed, and these conditions apply to a broad class of problems. In the second case, we present a nonobvious algorithm for solving the matrix chain product problem efficiently. Even though the techniques involved in the second case are problem specific, it serves as an excellent example for illustrating how speed-up comes about in dynamic programming: namely, by trying to solve individual subproblems fast, and by trying to keep small the total number of distinct subproblems that need solving.

REFERENCES

[1] A. AHO, J. HOPCROFT AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 1974.

[2] J. E. BOYCE, D. P. DOBKIN, R. L. DRYSDALE, III AND L. J. GUIBAS, *Finding extremal polygons*, Proc. 14th Annual ACM Symposium on Theory of Computing, (1982), pp. 282–289.

[3] E. N. GILBERT AND E. F. MOORE, *Variable length encodings*, Bell System Tech. J., 38 (1959), pp. 933–968.

[4] T. C. HU AND M. T. SHING, *Computation of matrix chain products, Part I and II*, manuscript (1981). (Extended abstract in Proc. 21st Annual Symposium on Foundations of Computer Science, 1980, pp. 28–35.)

[5] D. E. KNUTH, *Optimum binary search trees*, Acta Informatica, 1 (1971), pp. 14–25.

[6] ——, *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, second ed., Addison-Wesley, Reading, MA, 1975.

[7] ——, *The Art of Computer Programming, Vol 2: Seminumerical Algorithms*, second ed., Addison-Wesley, Reading, MA, 1981.

[8] F. F. YAO, *Efficient dynamic programming using quadrangle inequalities*, Proc. 12th Annual ACM Symposium on Theory of Computing, (1980), pp. 429–435.

[9] ———, *A note on optimally multiplying a sequence of matrices*, manuscript (1982).