Theoretical Computer Science 92 (1992) 49–76
Elsevier

Dynamic programming with convexity, concavity and sparsity*

Zvi Galil

Department of Computer Science, Columbia University, New York, NY 10027, USA and Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel

Kunsoo Park

Department of Computer Science, Columbia University, New York, NY 10027, USA

Abstract

Galil, Z. and K. Park, Dynamic programming with convexity, concavity and sparsity, Theoretical Computer Science 92 (1992) 49–76.

Dynamic programming is a general problem-solving technique that has been widely used in various fields such as control theory, operations research, biology and computer science. In many applications dynamic programming problems satisfy additional conditions of convexity, concavity and sparsity. This paper presents a classification of dynamic programming problems and surveys efficient algorithms based on the three conditions.

1. Introduction

Dynamic programming is a general technique for solving discrete optimization problems. The idea underlying this technique is to represent a problem by a decision process (minimization or maximization) which proceeds in a series of stages; i.e., the problem is formulated as a recurrence relation involving a decision process. For formal models of dynamic programming, see [32, 21]. Dynamic programming decomposes a problem into a number of smaller subproblems each of which is further decomposed until subproblems have trivial solutions. For example, a problem of size n may decompose into several problems of size $n-1$, each of which decomposes into several problems of size $n-2$, etc. This decomposition seems to lead to an exponential-time algorithm, which is indeed true in the traveling salesman problem [20, 9]. In many problems, however, there are only a polynomial number of distinct

* Work supported in part by NSF Grant CCR-88-14977.

subproblems. Dynamic programming gains its efficiency by avoiding solving common subproblems many times. It keeps track of the solutions of subproblems in a table, and looks up the table whenever needed. (It was called the tabulation technique in [10].)

Dynamic programming algorithms have three features in common:

(1) a table,

(2) the entry dependency of the table, and

(3) the order to fill in the table.

Each entry of the table corresponds to a subproblem. Thus the size of the table is the total number of subproblems including the problem itself. The entry dependency is defined by the decomposition: if a problem P decomposes into several subproblems $P_1, P_2, \ldots, P_k$, the table entry of the problem P depends on the table entries of $P_1, P_2, \ldots, P_k$. The order to fill in the table may be chosen under the restriction of the table and the entry dependency. Features (1) and (3) were observed in [6], and feature (2) can be found in the literature [10, 44] as dependency graphs. The dynamic programming formulation of a problem always provides an obvious algorithm which fills in the table according to the entry dependency.

Dynamic programming has been widely used in various fields such as control theory, operations research, biology, and computer science. In many applications dynamic programming problems have some conditions other than the three features above. Those conditions are convexity, concavity and sparsity among which convexity and concavity were observed in an earlier work [8]. Recently, a number of algorithms have been designed that run faster than the obvious algorithms by taking advantage of these conditions. In the next section we introduce four dynamic programming problems which have wide applications. In Section 3 we describe algorithms exploiting convexity and concavity. In Section 4 we show how sparsity leads to efficient algorithms. Finally, we discuss open problems and some dynamic programming problems which do not belong to the class of problems we consider here.

2. Dynamic programming problems

We use the term *matrices* for tables especially when the table is rectangular. Let A be an $n \times m$ matrix. $A[i, j]$ denotes the element in the ith row and the jth column. $A[i:i', j:j']$ denotes the submatrix of A which is the intersection of rows $i, i+1, \ldots, i'$ and columns $j, j+1, \ldots, j'$. $A[i, j:j']$ is a shorthand notation of $A[i:i, j:j']$. Throughout the paper we assume that the elements of a matrix are distinct, since ties can be broken consistently.

Let n be the size of problems. We classify dynamic programming problems by the table size and entry dependency: a dynamic programming problem is called a *tD/eD* problem if its table size is $O(n^t)$ and a table entry depends on $O(n^e)$ other entries. In the following we define four dynamic programming problems. Specific problems such as the least weight subsequence problem [24] are not defined in this paper since they are abstracted in the four problems; their definitions are left to references.

Problem 2.1 ($1D/1D$). Given a real-valued function $w(i,j)$ for integers $0 \leqslant i < j \leqslant n$ and $D[0]$, compute

$$E[j] = \min_{0 \leqslant i < j} \{D[i] + w(i,j)\} \quad \text{for } 1 \leqslant j \leqslant n, \tag{1}$$

where $D[i]$ is computed from $E[i]$ in constant time.

The least weight subsequence problem [24] is a special case of Problem 2.1 where $D[i] = E[i]$. Its applications include an optimum paragraph formation problem and the problem of finding a minimum height B-tree. The modified edit distance problem (or sequence alignment with gaps) [15], which arises in molecular biology, geology and speech recognition, is a $2D/1D$ problem, but it is divided into $2n$ copies of Problem 2.1. A number of problems in computational geometry are also reduced to a variation of Problem 2.1 [1].

Problem 2.2 ($2D/0D$). Given $D[i,0]$ and $D[0,j]$ for $0 \leqslant i,j \leqslant n$, compute

$$E[i,j] = \min\{D[i-1,j] + x_i,\, D[i,j-1] + y_j,\, D[i-1,j-1] + z_{i,j}\}$$
$$\text{for } 1 \leqslant i,j \leqslant n, \tag{2}$$

where x_i, y_j, and $z_{i,j}$ are computed in constant time, and $D[i,j]$ is computed from $E[i,j]$ in constant time.

The string edit distance problem [49], the longest common subsequence problem [22], and the sequence alignment problem [42] are examples of Problem 2.2. The sequence alignment with linear gap costs [18] is also a variation of Problem 2.2.

Problem 2.3 ($2D/1D$). Given $w(i,j)$ for $1 \leqslant i < j \leqslant n$ and $C[i,i] = 0$ for $1 \leqslant i \leqslant n$, compute

$$C[i,j] = w(i,j) + \min_{i < k \leqslant j} \{C[i,k-1] + C[k,j]\} \quad \text{for } 1 \leqslant i < j \leqslant n. \tag{3}$$

Some examples of Problem 2.3 are the construction of optimal binary search trees [36, 55], the maximum perimeter inscribed polygon problem [55, 56], and the construction of optimal t-ary tree [55]. Wachs [48] extended the optimal binary search tree problem to the case in which the cost of a comparison is not necessarily one.

Problem 2.4 ($2D/2D$). Given $w(i,j)$ for $0 \leqslant i < j \leqslant 2n$, and $D[i,0]$ and $D[0,j]$ for $0 \leqslant i,j \leqslant n$, compute

$$E[i,j] = \min_{\substack{0 \leqslant i' < i \\ 0 \leqslant j' < j}} \{D[i',j'] + w(i'+j', i+j)\} \quad \text{for } 1 \leqslant i,j \leqslant n, \tag{4}$$

where $D[i,j]$ is computed from $E[i,j]$ in constant time.

Problem 2.4 has been used to compute the secondary structure of RNA without multiple loops. The formulation of Problem 2.4 was provided by Waterman and Smith [50].

The dynamic programming formulation of a problem always yields an obvious algorithm whose efficiency is determined by the table size and entry dependency. If a dynamic programming problem is a tD/eD problem an obvious algorithm takes time $O(n^{t+e})$ provided that the computation of each term (e.g. $D[i]+w(i,j)$ in Problem 2.1) takes constant time. The space required is usually $O(n^t)$. The space may be sometimes reduced. For example, the space for the $2D/0D$ problem is $O(n)$. (If we proceed row by row, it suffices to keep only one previous row.) Even when we need to recover the minimum path as in the longest common subsequence problem, $O(n)$ space is still sufficient [22].

In the applications of Problems 2.1, 2.3 and 2.4 the cost function w is either convex or concave. In the applications of Problems 2.2 and 2.4 the problems may be sparse; i.e., we need to compute $E[i,j]$ only for a sparse set of points. With these conditions we can design more efficient algorithms than the obvious algorithms. For nonsparse problems we would like to have algorithms whose time complexity is $O(n^t)$ or close to it. For the $2D/0D$ problem $O(n^2)$ time is actually the lower bound in a comparison tree model with equality tests [5, 54]. For sparse problems we would like algorithms whose time complexity is close to the number of points for which we need to compute E.

3. Convexity and concavity

The convexity and concavity of the cost function w is defined by the *Monge* condition. It was introduced by Monge [41] in 1781, and revived by Hoffman [25] in connection with a transportation problem. Later, Yao [55] used it to solve the $2D/1D$ problem (Problem 2.3). Recently, the Monge condition has been used in a number of dynamic programming problems.

We say that the cost function w is *convex*[1] if it satisfies the convex Monge condition:

$$w(a,c)+w(b,d)\leqslant w(b,c)+w(a,d)\quad \text{for all } a<b \text{ and } c<d.$$

We say that the cost function w is *concave* if it satisfies the concave Monge condition:

$$w(a,c)+w(b,d)\geqslant w(b,c)+w(a,d)\quad \text{for all } a<b \text{ and } c<d.$$

Note that $w(i,j)$ for $i\geqslant j$ is not defined in Problems 2.1, 2.3, and 2.4. Thus w satisfies the Monge condition for $a<b<c<d$.

A condition closely related to the Monge condition was introduced by Aggarwal et al. [2]. Let A be an $n\times m$ matrix. A is *convex totally monotone* if for $a<b$ and $c<d$,

$$A[a,c]\geqslant A[b,c]\ \Rightarrow\ A[a,d]\geqslant A[b,d].$$

[1] The definitions of convexity and concavity were interchanged in some references.

Similarly, A is *concave totally monotone* if for all $a<b$ and $c<d$,

$$A[a,c] \leqslant A[b,c] \Rightarrow A[a,d] \geqslant A[b,d].$$

Let r_j be the row index such that $A[r_j,j]$ is the minimum value in column j. Convex total monotonicity implies that $r_1 \leqslant r_2 \leqslant \cdots \leqslant r_m$ (i.e. the minimum row indices are nondecreasing). Concave total monotonicity implies that $r_1 \geqslant r_2 \geqslant \cdots \geqslant r_m$ (i.e. the minimum row indices are nonincreasing). We say that an element $A[i,j]$ is *dead* if $i \neq r_j$ (i.e. $A[i,j]$ is not the minimum of column j). A submatrix of A is dead if all of its elements are dead. The convex (concave) Monge condition implies convex (concave) total monotonicity, respectively, but the converse is not true. Though the condition we have for the problems is the Monge condition, all the algorithms in this paper use only total monotonicity.

3.1. The 1D/1D problem

Hirschberg and Larmore [24] solved the convex $1D/1D$ problem in $O(n \log n)$ time using a queue. Galil and Giancarlo [15] proposed $O(n \log n)$ algorithms for both convex and concave cases. Miller and Myers [40] independently discovered an algorithm for the concave case, which is similar to Galil and Giancarlo's. We describe Galil and Giancarlo's algorithms. We should mention that if $D[i]=E[i]$ and $w(i,i)=0$ (i.e. w satisfies the triangle inequality or inverse triangle inequality), then the problem is trivial: $E[j]=D[0]+w(0,j)$ in the concave case and $E[j]=D[0]+w(0,1)+w(1,2)+\cdots+(j-1,j)$ in the convex case [8, 15].

Let $B[i,j]=D[i]+w(i,j)$ for $0 \leqslant i<j \leqslant n$. We say that $B[i,j]$ is *available* if $D[i]$ is known and therefore $B[i,j]$ can be computed in constant time. That is, $B[i,j]$ is available only after the column minima for columns $1,2,\ldots,i$ have been found. We call such matrix B *on-line* since its entries become available as we proceed. If all entries of a matrix are given at the beginning, that matrix is called *off-line*. The $1D/1D$ problem is to find the column minima in the on-line upper triangular matrix B. It is easy to see that when w satisfies the convex (concave) Monge condition, so does B. Hence B is totally monotone.

Galil and Giancarlo's algorithms find column minima one at a time and process available entries so that we keep only possible candidates for future column minima. In the concave case we use a stack to maintain the candidates. Figure 1 shows the outline of the algorithm. For each j, $2 \leqslant j \leqslant n$, we find the minimum at column j as

```
procedure concave $1D/1D$
  initialize stack with row 0;
  for $j \leftarrow 2$ to $n$ do
    find minimum at column $j$;
    update stack with row $j-1$;
  end do
end
```

Fig. 1. Galil and Giancarlo's algorithm for the concave $1D/1D$ problem.

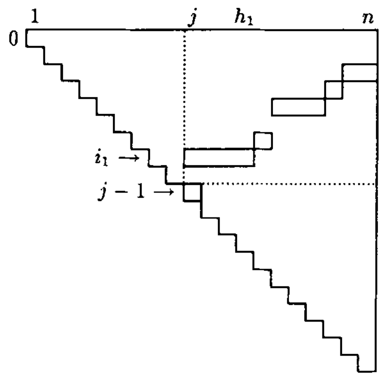

Fig. 2. Matrix B and stack elements at column j.

follows. Suppose that $(i_1, h_1), \ldots, (i_k, h_k)$ are on the stack ((i_1, h_1) is at the top of the stack). Initially, $(0, n)$ is on the stack. The invariant on the stack elements is that in submatrix $B[0:j-2, j:n]$ row i_r for $1 \leqslant r \leqslant k$ is the best (gives the minimum) in the column interval $[h_{r-1}+1, h_r]$ (assuming $h_0+1=j$). By the concave total monotonicity of B, $i_1, \ldots, i_k$ are nonincreasing (see Fig. 2). Thus the minimum at column j is the minimum of $B[i_1, j]$ and $B[j-1, j]$.

Now we update the stack with row $j-1$ as follows.

(1) If $B[i_1, j] \leqslant B[j-1, j]$, row $j-1$ is dead by concave total monotonicity. If $h_1 = j$, we pop the top element because it is of no use in the future.

(2) If $B[i_1, j] > B[j-1, j]$, we compare row $j-1$ with row i_r at h_r (i.e. $B[i_r, h_r]$ vs. $B[j-1, h_r]$) for $r = 1, 2, \ldots$ until row i_r is better than row $j-1$ at h_r. If row $j-1$ is better than row i_r at h_r, row i_r cannot give the minimum for any column because row $j-1$ is better than row i_r for column $l \leqslant h_r$ and row i_{r+1} is better than row i_r for column $l > h_r$; we pop the element (i_r, h_r) from the stack and continue to compare row $j-1$ with row i_{r+1}. If row i_r is better than row $j-1$ at h_r, we need to find the border of the two rows $j-1$ and i_r, which is the largest $h < h_r$ such that row $j-1$ is better than row i_r for column $l \leqslant h$; i.e., finding the zero z of $f(x) = B[j-1, x] - B[i_r, x] = w(j-1, x) - w(i_r, x) + (D[j-1] - D[i_r])$, then $h = \lfloor z \rfloor$. If $h \geqslant j+1$, we push $(j-1, h)$ into the stack.

Popping stack elements takes amortized constant time because the total number of pop operations cannot be greater than n. If it is possible to compute the border h in constant time, we say that w satisfies the *closest zero property*. Concave functions such as $w(i, j) = \log(j-i)$ and $\sqrt{j-i}$ satisfy the closest zero property. For those simple functions the total time of the algorithm is $O(n)$. For more general w we can compute h in $O(\log n)$ time using a binary search. Hence the total time is $O(n \log n)$ in general.

The convex case is similar. We use a queue instead of a stack to maintain the candidates. Suppose that $(i_1, h_1), \ldots, (i_k, h_k)$ are in the queue at column j ((i_1, h_1) is at the front and (i_k, h_k) is at the rear of the queue). The invariant on the queue elements is that in $B[0{:}j-2, j{:}n]$ row i_r for $1 \leqslant r \leqslant k$ is the best in the interval $[h_r, h_{r+1}-1]$ (assuming $h_{k+1}-1=n$). By the convex total monotonicity of B, $i_1, \ldots, i_k$ are nondecreasing. Thus the minimum at column j is the minimum of $B[i_1, j]$ and $B[j-1, j]$. One property satisfied by the convex case only is that if $B[j-1, j] \leqslant B[i_1, j]$, the whole queue is emptied by convex total monotonicity. This property is used to get a linear-time algorithm for the convex case as we will see shortly.

3.2. The matrix searching technique

Aggarwal et al. [2] show that the *row maxima* of a totally monotone matrix A can be found in linear time if $A[i,j]$ for any i,j can be computed in constant time (i.e. A is off-line). Since in this subsection we are concerned with row maxima rather than column minima, we define total monotonicity with respect to rows: an $n \times m$ matrix A is totally monotone if for all $a<b$ and $c<d$,

$$A[a,d] \geqslant A[a,c] \;\Rightarrow\; A[b,d] \geqslant A[b,c].$$

Also, an element $A[i,j]$ is dead if $A[i,j]$ is not the maximum of row i.

We find the row maxima in $O(m)$ time on $n \times m$ matrices when $n \leqslant m$. The key component of the algorithm is the subroutine *REDUCE* (Fig. 3). It takes as input an $n \times m$ totally monotone matrix A with $n \leqslant m$. *REDUCE* returns an $n \times n$ matrix Q which is a submatrix of A with the property that Q contains the columns of A which have the row maxima of A.

Let k be a column index of Q whose initial value is 1. *REDUCE* maintains an invariant on k that for all $1 \leqslant j \leqslant k$, $Q[1{:}j-1]$ is dead (see Fig. 4). Also, only dead columns are deleted. The invariant holds trivially when $k=1$. If $Q[k,k] > Q[k,k+1]$ then $Q[1{:}k, k+1]$ is dead by total monotonicity. Thus if $k<n$, we increase k by 1. If $k=n$, column $k+1$ is dead and k remains the same. If $Q[k,k] \leqslant Q[k,k+1]$ then $Q[k{:}n, k]$ is dead by total monotonicity. Since $Q[1{:}k-1, k]$ was already dead by the invariant, column k is dead; we decrease k by 1 if k was greater than 1.

```
procedure REDUCE(A)
  $Q \leftarrow A$;
  $k \leftarrow 1$;
  while $Q$ has more than $n$ columns do
    case
    $Q[k,k] > Q[k,k+1]$ and $k < n$: $k \leftarrow k+1$;
    $Q[k,k] > Q[k,k+1]$ and $k = n$: delete column $k+1$;
    $Q[k,k] \leq Q[k,k+1]$:
      delete column $k$;
      if $k > 1$ then $k \leftarrow k-1$;
    end case
  end do
  return($Q$);
end
```

Fig. 3. The *REDUCE* procedure.

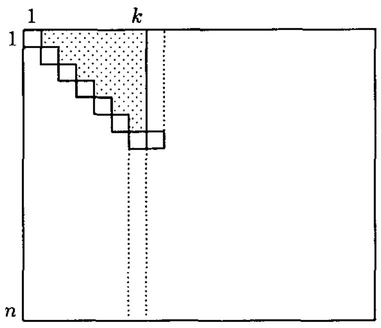

Fig. 4. Matrix Q (the shaded region is dead).

```
procedure $ROWMAX(A)$
  $Q \leftarrow REDUCE(A)$;
  if $n = 1$ then output the maximum and return;
  $P \leftarrow \{Q_2, Q_4, \ldots, Q_{2\lfloor n/2 \rfloor}\}$;
  $ROWMAX(P)$;
  from the known positions of the maxima in the even rows of $Q$,
    find the maxima in its odd rows;
end
```

Fig. 5. The SMAWK algorithm.

For the time analysis of *REDUCE*, let a, b, and c denote, respectively, the number of times the first, second, and third branches of the **case** statement are executed. Since a total of $m-n$ columns are deleted and a column is deleted only in the last two branches, we have $b+c=m-n$. Let c' be the number of times k decreases in the third branch. Then $c' \leqslant c$. Since k starts at 1 and ends no larger than $n, a-c \leqslant a-c' \leqslant n-1$. We have time $t=a+b+c \leqslant a+2b+c \leqslant 2m-n-1$.

In order to find the row maxima of an $n \times m$ totally monotone matrix A with $n \leqslant m$, we first use *REDUCE* to get an $n \times n$ matrix Q, and then recursively find the row maxima of the submatrix of Q which is composed of even rows of Q. After having found the row maxima of even rows, we compute the row maxima in odd rows. The procedure *ROWMAX* in Fig. 5 shows the algorithm.

Let $T(n, m)$ be the time taken by *ROWMAX* for an $n \times m$ matrix. The call to *REDUCE* takes time $O(m)$. The assignment of the even rows of Q to P is just the manipulation of a list of rows, and can be done in $O(n)$ time. P is an $(n/2) \times n$ totally monotone matrix, so the recursive call takes time $T(n/2, n)$. Once the positions of the maxima in the even rows of Q have been found, the maximum in each odd row is restricted to the interval of maximum positions of the neighboring even rows. Thus, finding all maxima in the odd rows can be done in $O(n)$ time. For some constants c_1 and c_2, the time satisfies the following inequality

$$T(n, m) \leqslant c_1 n + c_2 m + T(n/2, n),$$

which gives the solution $T(n,m) \leqslant 2(c_1+c_2)n+c_2m$. Since $n \leqslant m$, this is $O(m)$. The bookkeeping for maintaining row and column indices also takes linear time.

When $n>m$, the row maxima can be found in $O(m(1+\log(n/m)))$ time [2]. But it suffices to have $O(n)$ time by extending the matrix to an $n \times n$ matrix with $-\infty$ and applying *ROWMAX*. Therefore, we can find the row maxima of an $n \times m$ matrix in $O(n+m)$ time. By simple modifications we can find the column minima of a matrix in $O(n+m)$ time whenever the matrix is convex or concave totally monotone. We will refer to the algorithm for finding the column minima as the SMAWK algorithm.

There are two difficulties in applying the SMAWK algorithm to the $1D/1D$ problem. First of all, B is not rectangular. In the convex case we may put $+\infty$ into the lower half of B. Then B is still totally monotone. In the concave case, however, total monotonicity no longer holds with $+\infty$ entries. A more serious difficulty is that the SMAWK algorithm requires that matrix A be off-line. In the $1D/1D$ problem $B[i,j]$ is available only after the column minima for columns $1, \ldots, i$ have been found. Though the SMAWK algorithm cannot be directly applied to the $1D/1D$ problem, many algorithms for this problem [52, 11, 17, 34, 1, 33, 35] use it as a subroutine.

3.3. The 1D/1D problem revisited

Wilber [52] proposed an $O(n)$ time algorithm for the convex $1D/1D$ problem when $D[i]=E[i]$. However, his algorithm does not work if the matrix B is on-line, which happens when many copies of the $1D/1D$ problem proceed simultaneously (i.e. the computation is interleaved among many copies) as in the mixed convex and concave cost problem [11] and the modified edit distance problem [15]. Eppstein [11] extended Wilber's algorithm for interleaved computation. Galil and Park [17] gave a linear time algorithm which is more general than Eppstein's and simpler than both Wilber's and Eppstein's. Larmore and Schieber [38] developed another linear-time algorithm, which is quite different from Galil and Park's. Klawe [34] reported yet another linear-time algorithm.

For the concave $1D/1D$ problem a series of algorithms were developed, and the best algorithm is almost linear: an $O(n\log\log n)$ time algorithm due to Aggarwal and Klawe [1], an $O(n\log^* n)$ algorithm due to Klawe [33], and finally an $O(n\alpha(n))$ algorithm due to Klawe and Kleitman [35]. The function $\log^* n$ is defined to be $\min\{i \mid \log^{(i)} n \leqslant 1\}$, where $\log^{(i)} n$ means applying the log function i times to n, and $\alpha(n)$ is a functional inverse of Ackermann's function as defined below. We define the functions $L_i(n)$ for $i=-1,0,1,2,\ldots$ recursively: $L_{-1}(n)=n/2$, and for $i \geqslant 0$, $L_i(n)=\min\{s \mid L_{i-1}^{(s)}(n) \leqslant 1\}$. Thus, $L_0(n)=\lceil \log n \rceil$, $L_1(n)$ is essentially $\log^* n$, $L_2(n)$ is essentially $\log^{**} n$, etc. We now define $\alpha(n)=\min\{s \mid L_s(n) \leqslant s\}$.

Aggarwal and Klawe [1] introduce *staircase matrices* and show that a number of problems in computational geometry can be reduced to the problem of finding the row minima of totally monotone staircase matrices. We define staircase matrices in terms of columns: an $n \times m$ matrix S is a staircase matrix if

(1) there are integers f_j for $j=1,2,\ldots,m$ associated with S such that $1\leqslant f_1\leqslant f_2\leqslant\cdots\leqslant f_m\leqslant n$, and

(2) $S[i,j]$ is a real number if and only if $1\leqslant i\leqslant f_j$ and $1\leqslant j\leqslant m$. Otherwise, $S[i,j]$ is undefined.

We call column j a *step-column* if $j=1$ or $f_j>f_{j-1}$ for $j>2$, and we say that S has t steps if it has t step-columns. Fig. 6 shows a staircase matrix with four steps.

Instead of the $1D/1D$ problem we consider its generalization:

$$E[j]=\min_{0\leqslant i\leqslant f_j}\{D[i]+w(i,j)\}\quad\text{for } 1\leqslant j\leqslant n, \tag{5}$$

where $0\leqslant f_1\leqslant\cdots\leqslant f_n<n$. $D[i]$ for $i=0,\ldots,f_1$ are given, and $D[i]$ for $i=f_{j-1}+1,\ldots,f_j$ $(j>1)$ are easily computed from $E[j-1]$. This problem occurs as a subproblem in a solution of the $2D/2D$ problem. It becomes the $1D/1D$ problem if $f_j=j-1$.

Let $B[i,j]=D[i]+w(i,j)$ again. Now B is an on-line staircase matrix. When w is convex, we present a linear-time algorithm that is an extension of Galil and Park's algorithm [17] to on-line staircase matrices.

As we compute E from $E[1]$ to $E[n]$, we reduce the staircase matrix B to successively smaller staircase matrices B'. For each staircase matrix B' we maintain two indices r and c: r is the first row of B', c its first column. That is, $B'[i,j]=B[i,j]$ if $i\geqslant r$ and $j\geqslant c$, $B'[i,j]$ is undefined otherwise. We use an array $N[1:n]$ to store interim column minima before row r; i.e., $N[j]=B[i,j]$ for some $i<r$ (its usage will be clear shortly). At the beginning of each stage the following invariants hold:

(a) $E[j]$ for all $1\leqslant j<c$ have been found.

(b) $E[j]$ for $j\geqslant c$ is min $(N[j], \min_{i\geqslant r}B'[i,j])$.

Invariant (a) means that $D[i]$ for all $0\leqslant i\leqslant f_c$ are known, and therefore $B'[r:f_c, c:n]$ is available. Initially, $r=0$, $c=1$, and $N[j]=+\infty$ for all $1\leqslant j\leqslant n$.

Let $p=\min(f_c+c-r+1,n)$. We construct a matrix G which consists of $N[c:p]$ as its first row and $B'[r:f_c,c:p]$ as the other rows. G is a square matrix unless

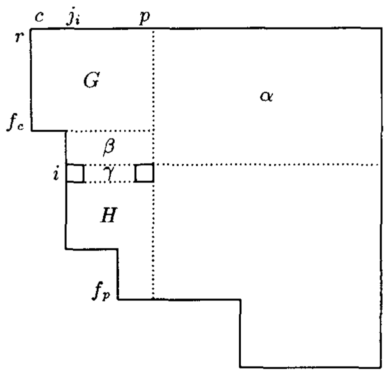

Fig. 6. Staircase matrix B'.

$n<f_c+c-r+1$. Since matrix G is convex totally monotone, its column minima $F[c:p]$ can be found using the SMAWK algorithm. Let c' be the first step-column after c, and H be the part of B' below G.

(1) If $c'>p$ (i.e., H is empty), we have found column minima $E[c:p]$ which are $F[c:p]$ by invariant (b).

If $c'\leqslant p$, column minima $E[c:c'-1]$ have been found. For the other column minima we need to process H. For each row in H we make two comparisons: one with its first entry, the other with its last entry until case (2) or (3) happens. Let i be the current row of H, and j_i the first column of row i. Initially, $i=f_c+1$. Assume inductively that the part of H above row i(β in Fig. 6) is dead.

(2) If $B'[i,j_i]\leqslant F[j_i]$, then rows $r,\ldots,i-1$ of B' are dead by the convex total monotonicity of B' (i.e. emptying the queue in Galil and Giancarlo's algorithm).

(3) If $B'[i,j_i]>F[j_i]$ and $B'[i,p]\leqslant F[p]$, then $B'[r:i-1,p+1:n]$ (α is in Fig. 6) is dead. Though G is not dead, we can store $F[j_i:p]$ into $N[j_i:p]$ and remove rows $r,\ldots,i-1$.

(4) Otherwise, $B'[i,j_i:p]$ (γ in Fig. 6) is dead by convex total monotonicity. We go on to the next row of H.

If case (4) is repeated until the last row of H, we have found column minima $E[c:p]$. This case will be called (4′). Note that whenever a new step-column starts, column minima for previous columns have been found, so all entries involved in the computation above become available. If case (2) or (3) happens, we start a new stage with $r=i$ and $c=j_i$. Otherwise, we start a new stage with $c=p+1$ (r remains the same). The two invariants hold at the beginning of new stages.

Let i' be the last row of B' which was involved in the computation (e.g. $i'=f_c=f_p$ in case (1) and $i'=f_p$ in case (4′)). Each stage takes time $O(i'-r)$. If case (2) or (3) happens, we charge the time to rows $r,\ldots,i'-1$ because these rows are removed from B'. If case (1) or (4′) happens, as long as $p<n$, $O(f_c-r)$ time is charged to columns $c,\ldots,p$ because these columns are removed, but the remaining $O(f_p-f_c)$ in case (4′) is counted separately. Note that f_p-f_c is the number of rows in H, and these rows can never be a part of a future H. Overall, the remaining $O(f_p-f_c)$'s are summed up to $O(n)$. If $p=n$ in case (1) or (4′), we have finished the whole computation, and rows $r,\ldots,n-1$ have not been charged yet; $O(f_p-r)$ time is charged to the rows. Thus the total time of the algorithm is linear in n.

When w is concave, Klawe and Kleitman's algorithm [35] solves Recurrence 5 in $O(n\alpha(n))$ time. They gave an algorithm for finding the row minima of off-line staircase matrices, and then modified it to work for on-line staircase matrices. If the matrix B is transposed, finding column minima in B becomes finding row minima in the transposed matrix. We say that a staircase matrix has *shape* (t, n, m) if it has at most t steps, at most n rows, and at most m columns. We describe their off-line algorithm without going into the details.

The idea of Klawe and Kleitman's algorithm is to reduce the number of steps of a staircase matrix to two, for the row minima of a staircase matrix with two steps can be found in linear time using the SMAWK algorithm. We first reduce a staircase matrix of shape (n, n, m) to a staircase matrix of shape $(n/(\alpha(n))^3, n/(\alpha(n))^2, m)$ in $O(m\alpha(n)+n)$ time. Then we successively reduce the staircase matrix to staircase matrices with fewer steps by the following reduction scheme: a staircase matrix of shape $(n/L_s(n), n, m)$ is reduced to a set of staircase matrices $G_1, \ldots, G_k$ in $O(m+n)$ time, where G_i is of shape $(n_i/L_{s-1}(n_i), n_i, m_i)$, $\sum_{i=1}^k n_i \leqslant n$, and $\sum_{i=1}^k m_i \leqslant m+n$. It is shown by induction on s that by the reduction scheme the processing (finding row minima) of a staircase matrix of shape $(n/L_s(n), n, m)$ can be done in $O(sm+s^2n)$ time. Thus the staircase matrix of shape $(n/(\alpha(n))^3, n/(\alpha(n))^2, m)$ can be processed in $O(m\alpha(n)+n)$.

3.4. *Mixed convex and concave costs*

Eppstein [11] suggested an approach for general $w(i,j)$ based upon convexity and concavity. In many applications the cost function $w(i,j)$ may be represented by a function of the difference $j-i$; i.e. $w(i,j)=g(j-i)$. Then g is convex or concave in the usual sense exactly when w is convex or concave by the Monge condition. For general w, it is not possible to solve the $1D/1D$ problem more efficiently than the obvious $O(n^2)$ algorithm because we must look at each of the possible values of $w(i,j)$. However, this argument does not apply to g because $g(j-i)$ has only n values. A general g can be divided into piecewise convex and concave functions. Let $c_0, c_1, \ldots, c_s$ be the indices in the interval $[1, n]$ ($c_0=1$ and $c_s=n$) such that g is convex in $[1, c_1]$, concave in $[c_1, c_2]$, and so on. By examining adjacent triples of the values of g, we can easily divide $[1, n]$ into s such intervals in linear time. We form s functions $g_1, g_2, \ldots, g_s$ as follows: $g_p(x)=g(x)$ if $c_{p-1} \leqslant x \leqslant c_p$; $g_p(x)=+\infty$ otherwise. Then Recurrence 1 becomes

$$E[j]=\min_{1\leqslant p\leqslant s} E_p[j],$$

where

$$E_p[j]=\min_{0\leqslant i<j} \{D[i]+g_p(j-i)\}. \tag{6}$$

We solve Recurrence 6 independently for each g_p, and find $E[j]$ as the minimum of the s results obtained.

It turns out that convex g_p remains convex in the entire interval $[1, n]$, and therefore we could apply the algorithm for the convex $1D/1D$ problem directly. The solution for concave g_p is more complicated because the added infinity values interfere with concavity. Eppstein solves both convex and concave g_p in such a way that the average time per segment is bounded by $O(n\alpha(n/s))$. Consider a fixed g_p, either convex or concave. Let $B_p[i,j]=D[i]+g_p(j-i)$. Then $E_p[j]$ is the minimum of those values $B_p[i,j]$ such that $c_{p-1} \leqslant j-i \leqslant c_p$; the valid B_p entries for E_p form a diagonal strip of width $a_p=c_p-c_{p-1}$. We divide this strip into triangles as shown in Fig. 7. We solve the recurrence independently on each triangle, and piece together the solutions to obtain the solution for the entire strip. $E_p[j]$ for each j is determined by an upper

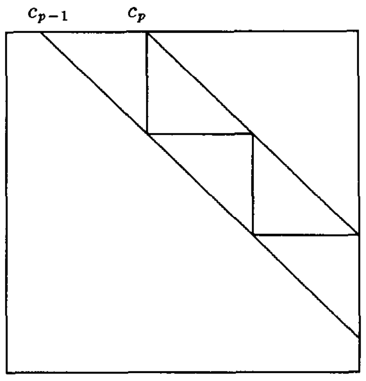

Fig. 7. B_p entries and triangles.

triangle and a lower triangle. The computation within an upper triangle is a $1D/1D$ problem of size a_p. The computation within a lower triangle does not look like a $1D/1D$ problem. However, observe that all B_p entries for a lower triangle are available at the beginning of the computation; we can reverse the order of j. Each triangle is solved in $O(a_p\alpha(a_p))$ time. (If the triangle is convex, it is actually $O(a_p)$.) Since a strip is composed of $O(n/a_p)$ upper and lower triangles, the total time to compute E_p is $O(n\alpha(a_p))$. The time for all E_p's is $O(\sum_{p=1}^{s} n\alpha(a_p))$. Since $\sum_{p=1}^{s} a_p = n$, the sum is maximized when each $a_p = n/s$ by the concavity of α; the time is $O(ns\alpha(n/s))$. Since the time for combining values to get E is $O(ns)$, the overall time is $O(ns\alpha(n/s))$. Note that the time bound is never worse than $O(n^2)$.

3.5. The $2D/1D$ problem

For the convex $2D/1D$ problem Frances Yao [55] gave an $O(n^2)$ algorithm. She imposed one more condition on w which comes from the applications:

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

That is, w is monotone on the lattice of intervals ordered by inclusion.

Lemma 3.1. (Yao [55]). *If w satisfies the convex Monge condition and the condition above, C defined by Recurrence 3 also satisfies the convex Monge condition.*

Let $C_k(i,j)$ denote $w(i,j) + C[i,k-1] + C[k,j]$. Let $K_{i,j}$ be the index k where the minimum is achieved for $C[i,j]$; i.e. $C[i,j] = C_k(i,j)$.

Lemma 3.2 (Yao [55]). *If C satisfies the convex Monge condition, we have*

$$K_{i,j-1} \leqslant K_{i,j} \leqslant K_{i+1,j} \quad \text{for } i<j.$$

Proof. It is true when $i+2=j$; therefore assume that $i+2<j$. We prove the first inequality. Let $k=K_{i,j-1}$. Thus $C_k(i,j-1)<C_{k'}(i,j-1)$ for $i<k'<k$. We need to show that $C_k(i,j)<C_{k'}(i,j)$ for $i<k'<k$, which means $K_{i,j}\geqslant k$. Take the convex Monge condition of C at $k'<k\leqslant j-1<j$

$$C[k',j-1]+C[k,j]\leqslant C[k,j-1]+C[k',j].$$

Adding $w(i,j-1)+w(i,j)+C[i,k'-1]+C[i,k-1]$ to both sides, we get

$$C_{k'}(i,j-1)+C_k(i,j)\leqslant C_k(i,j-1)+C_{k'}(i,j).$$

Since $C_k(i,j-1)<C_{k'}(i,j-1)$, we have $C_k(i,j)<C_{k'}(i,j)$. Similarly, the second inequality follows from the convex Monge condition of C at $i<i+1\leqslant k<k'$ with $k=K_{i+1,j}$. □

Once $K_{i,j-1}$ and $K_{i+1,j}$ are given, $C[i,j]$ and $K_{i,j}$ can be computed in $O(K_{i+1,j}-K_{i,j-1}+1)$ time by Lemma 3.2. Consequently, when we compute $C(i,j)$ for diagonal $d=j-i=1,2,\ldots,n-1$, the total amount of work for a fixed d is $O(n)$; the overall time is $O(n^2)$.

The $O(n^2)$ time construction of optimal binary search trees due to Knuth [36] is immediate from Yao's result because the problem is formulated as the convex $2D/1D$ problem. Yao [55, 56] gave an $O(n^2 \log d)$ algorithm for the maximum perimeter inscribed d-gon problem using the $2D/1D$ problem, which was improved to $O(dn \log^{d-2} n)$ by Aggarwal and Park [3]. Yao's technique is general in the sense that it applies to any dynamic programming problem with convexity or concavity. It gives a tighter bound on the range of indices as in the convex $2D/1D$ problem. In some cases, however, the tighter bound by itself may not be sufficient to reduce the asymptotic time complexity. The $1D/1D$ problem and the concave $2D/1D$ problem are such cases. Aggarwal and Park [3] solved the concave $2D/1D$ problem in time $O(n^2\alpha(n))$ by dividing it into n concave $1D/1D$ problems. If we fix $\hat{i}$ in Recurrence 3, it becomes a $1D/1D$ problem:

$$D'[q]=\min_{0\leqslant p<q}\{D'[p]+w'(p,q)\} \quad \text{for } 1\leqslant q\leqslant m,$$

where $D'[q]=C[\hat{i},q+\hat{i}]$, $w'(p,q)=C[p+\hat{i}+1,q+\hat{i}]+w(\hat{i},q+\hat{i})$, and $m=n-\hat{i}$.

3.6. The 2D/2D problem

We define a partial order on the points $\{(i,j)|0\leqslant i,j\leqslant n\}$: $(i',j')\prec(i,j)$ if $i'<i$ and $j'<j$. We define the *domain* $d(i,j)$ of a point (i,j) to be the set of points (i',j') such that $(i',j')\prec(i,j)$, and the *diagonal index* of a point (i,j) to be $i+j$. Let $d_k(i,j)$ be the set of points (i',j') in $d(i,j)$ whose diagonal index is $k=i'+j'$. The $2D/2D$ problem has a property that the function w depends only on the diagonal index of its two argument points. Thus when we compute $E[i,j]$, we have only to consider the minimum among the $D[i',j']$ entries in $d_k(i,j)$, where $k=i'+j'$.

Waterman and Smith [51] used this property to obtain an $O(n^3)$ algorithm without convexity and concavity assumptions. Their algorithm proceeds row by row to compute $E[i,j]$. When we compute row i, for each column j we keep only the minimum D entry in $d_k(i,j)$ for each k. That is, we define

$$H^i[k,j] = \min_{(i',j')\in d_k(i,j)} D[i',j'].$$

Then, Recurrence 4 can be written as

$$E[i,j] = \min_{k<i+j-1} \{H^i[k,j] + w(k,i+j)\}.$$

For each i,j we compute $H^i[k,j]$ from $H^{i-1}[k,j]$ for all k in $O(n)$ time, since $H^i[k,j] = \min\{H^{i-1}[k,j], D[i-1,k-i+1]\}$. Then $E[i,j]$ is computed in $O(n)$ time. The overall time is $O(n^3)$.

Using the convexity and concavity of w. Eppstein et al. [12] gave an $O(n^2\log^2 n)$ algorithm for the $2D/2D$ problem. As in Galil and Giancarlo's algorithm for the $1D/1D$ problem, it maintains possible candidates for future points in a data structure. Subsequently, asymptotically better algorithms were discovered.

Aggarwal and Park [3] suggested a divide-and-conquer approach combined with the matrix searching technique for the $2D/2D$ problem. We partition E into four $n/2 \times n/2$ submatrices E_1, E_2, E_3, and E_4 as shown in Fig. 8. Let $T(n)$ be the time complexity of the algorithm for the problem of size n. We recursively compute E_1 in $T(n/2)$ time. We compute the *influence* of E_1 on E_2 in $O(n^2)$ time, and recursively compute E_2 in $T(n/2)$ time, taking into account the influence of E_1 on E_2; similarly for E_3. Finally, we compute the influence of E_1 on E_4, the influence of E_2 on E_4, and the influence of E_3 on E_4, all in $O(n^2)$ time, and then recursively compute E_4 in $T(n/2)$ time. This yields $T(n) = 4T(n/2) + O(n^2)$. Since $T(1) = O(1)$, we get $T(n) = O(n^2 \log n)$.

The influence of E_1 on E_2 is computed as follows (other influences are computed similarly). Each point $(i, j+n/2)$ in row i of E_2 has the same domain in E_1, and thus

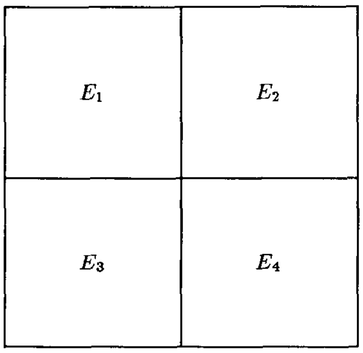

Fig. 8. The partition of matrix E.

depends on the same diagonal minima in E_1. Consequently, consider the matrix $X_i[k,j]$ for $1 \leqslant j \leqslant n/2$ and $0 \leqslant k < i+n/2$ where

$$X_i[k,j] = H^i[k, n/2+1] + w(k, i+j+n/2). \tag{7}$$

That is, $X_i[k,j]$ contains the influence of E_1 on point $(i, j+n/2)$. Then $E[i, j+n/2]$ is the minimum of the two: $\min_k X_i[k,j]$, and the recursive solution of E_2. Computing the influence of E_1 on E_2 reduces to computing the column minima of X_i for $1 \leqslant i \leqslant n/2$. The matrix X_i is totally monotone when w is either convex or concave. As mentioned before, $H^i[k, n/2+1]$ for all k can be computed from $H^{i-1}[k, n/2+1]$ in $O(n)$ time. Once this has been done, the column minima of X_i can be computed in $O(n)$ time using the SMAWK algorithm. The total time for computing the influence is $O(n^2)$.

Larmore and Schieber [38] gave an algorithm for the $2D/2D$ problem, which runs in $O(n^2)$ time for the convex case and in $O(n^2\alpha(n))$ time for the concave case. Their algorithm uses an algorithm for the $1D/1D$ problem as a subroutine, and this is the only place where the convex and concave cases differ.

We assume that $n+1$ is a power of two. The algorithm can be easily modified for $n+1$ that is not a power of two. For $0 \leqslant l \leqslant \log_2(n+1)$, we define a *square of level* l to be a $2^l \times 2^l$ square of points whose upper left corner is at the point (i,j) such that both i and j are multiples of 2^l. Let $S^l_{u,v}$ be the square of level l whose upper left corner is at the point $(u2^l, v2^l)$. Let $S^l_{u,*}$ be the set of squares of level l whose upper left corner is in row $u2^l$. Similarly $S^l_{*,v}$ is the set of squares of level l whose upper left corner is in column $v2^l$. Note that each square $S^l_{u,v}$ consists of four squares of level $l-1$. Let $S^l(i,j)$ be the square of level l that contains the point (i,j). We extend the partial order $\prec$ over the squares: $S' \prec S$ if every point in S' precedes every point in S.

For $0 \leqslant l \leqslant \log_2(n+1)$, we define

$$E_l[i,j] = \min\{D[i',j'] + w(i'+j', i+j) \mid S^l(i',j') \prec S^l(i,j)\}.$$

Note that $E_0[i,j] = E[i,j]$, and $E_l[i,j]$ for $l>0$ is an approximation of $E[i,j]$.

Suppose that the matrix D is off-line. Let $E_{\log_2(n+1)}[i,j] = +\infty$ for all (i,j). We compute the matrix E_l given the matrix E_{l+1} from $l = \log_2(n+1)-1$ to 0. Consider a point (p,q), and *let* $S^{l+1}_{u,v} = S^{l+1}(p,q)$. There are four cases depending on the position of the square $S^l(p,q)$ in the square $S^{l+1}(p,q)$.

(1) $S^l(p,q)$ is the upper left subsquare of $S^{l+1}(p,q)$. That is, $S^l(p,q) = S^l_{2u,2v}$. It is easy to see that $E_l[p,q] = E_{l+1}[p,q]$.

(2) $S^l(p,q)$ is the upper right subsquare of $S^{l+1}(p,q)$. That is, $S^l(p,q) = S^l_{2u,2v+1}$. The points in the squares of $S^l_{*,2v}$ that precede the square $S^l_{2u,2v+1}$ have to be considered when computing $E_l[p,q]$ (see Fig. 9). For the points (i,j) in the squares of $S^l_{*,2v+1}$, we define the *column recurrence*

$$CR^l_v[i,j] = \min\{D[i',j'] + w(i'+j', i+j) \mid (i',j') \in S^l_{*,2v} \text{ and } S^l(i',j') \prec S^l(i,j)\}.$$

Then, $E_l[p,q] = \min\{E_{l+1}[p,q], CR^l_v[p,q]\}$.

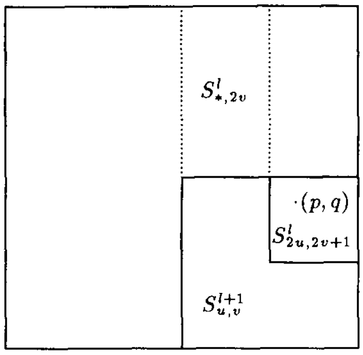

Fig. 9. $S^l(p,q)$ is the upper right subsquare of $S^{l+1}(p,q)$.

(3) $S^l(p,q)$ is the lower left subsquare of $S^{l+1}(p,q)$. That is, $S^l(p,q)=S^l_{2u+1,2v}$. The points in the squares of $S^l_{2u,*}$ that precede the square $S^l_{2u+1,2v}$ have to be considered when computing $E_l[p,q]$. For the points (i,j) in the squares of $S^l_{2u+1,*}$, we define the *row recurrence*

$$RR^l_u[i,j]=\min\{D[i',j']+w(i'+j',i+j)\,|\,(i',j')\in S^l_{2u,*} \text{ and } S^l(i',j')\prec S^l(i,j)\}.$$

Then, $E_l[p,q]=\min\{E_{l+1}[p,q], RR^l_u[p,q]\}$.

(4) $S^l(p,q)$ is the lower right subsquare of $S^{l+1}(p,q)$. That is, $S^l(p,q)=S^l_{2u+1,2v+1}$. Similarly to the previous cases, $E_l[p,q]=\min\{E_{l+1}[p,q], CR^l_v[p,q], RR^l_u[p,q]\}$.

From the fact that the function w depends only on the diagonal index of the points we have the following observations:

(a) The value of $E_l[i,j]$ (also $CR^l_v[i,j]$ and $RR^l_u[i,j]$) is the same for all points in a square of level l which have the same diagonal index $i+j$.

(b) To compute $CR^l_v[i,j]$ (or $RR^l_u[i,j]$) it is sufficient to consider only the minimum $D[i',j']$ among all points in a square of level l which have the same diagonal index $i'+j'$.

By observation (a) we keep $2^{l+1}-1$ values for each square of level l (corresponding to the diagonals). Since there are $(n+1)^2/2^{2l}$ squares at level l, E_l has $O(n^2/2^l)$ values. Thus the overall computation above takes $O(n^2)$ time except for the row and column recurrences. To compute E_l from E_{l+1}, we have to solve $(n+1)/2^{l+1}$ row recurrences and $(n+1)/2^{l+1}$ column recurrences. We will show that each recurrence is solved by four instances of Recurrence 5. Overall, there are $O(n)$ instances of Recurrence 5, which implies $O(n^2)$ time for convex w and $O(n^2\alpha(n))$ for concave w.

Now we show how to compute CR^l_v. The algorithm for the row recurrences is analogous. Each square of level l will be assigned a color, either red or black. The upper left square is red, and all squares of level l are assigned colors in the checkerboard fashion. Consider the diagonals that intersect the squares in $S^l_{*,2v+1}$. Each diagonal intersects at most two squares in $S^l_{*,2v+1}$, one red square and one black square. By observation (a) the value of $CR^l_v[i,j]$ is the same for a diagonal in a square, but each diagonal intersects two squares. Thus we divide the row recurrence into two recurrences: the *red recurrence* for points in red squares and the *black recurrence* for points in black squares. By observation (b) we need to consider only the minimum $D[i',j']$ for a diagonal in a square, but again each diagonal in $S^l_{*,2v}$ intersects two squares, one red square and one black square. Thus we divide the red recurrence into two recurrences: The *red-red recurrence* and the *red-black recurrence*. Similarly, the black recurrence is divided into two recurrences. Therefore, each row or column recurrence is solved by four instances of the $1D/1D$ problem.

In the $2D/2D$ problem, however, the matrix D is on-line. Larmore and Schieber modified the algorithm above so that it computes the entries of $E=E_0$ row by row and uses only available entries of D without increasing the time complexity. Therefore, it leads to an $O(n^2)$ time algorithm for the convex case and an $O(n^2\alpha(n))$ time algorithm for the concave case.

3.7. Variations of the 2D/2D problem

When $w(i,j)$ is a general function of the difference $j-i$ (i.e. $w(i,j)=g(j-i)$, and g is piecewise convex and concave with s segments), Eppstein [11] suggests an $O(n^2 s\alpha(n/s)\log n)$ algorithm for the $2D/2D$ problem. We use the partition of Aggarwal and Park in Fig. 8. Since Equation 7 involves w, finding the column minima of X_i differs from the case where w is either convex or concave. As in Subsection 3.4, we divide the matrix X_i into diagonal strips and the strips into triangles. Thus the column minima of X_i can be found in $O(ns\alpha(n/s))$ time. Computing the influence for all $1\leqslant i\leqslant n/2$ takes time $O(n^2 s\alpha(n/s))$. This gives the time

$$T(n)=4T(n/2)+O(n^2 s\alpha(n/s))=O(n^2 s\alpha(n/s)\log n).$$

If $w(i,j)$ is a linear function of the difference $j-i$ (i.e. $w(i,j)=a(j-i)+b$ for some constants a and b), the $2D/2D$ problem reduces to a $2D/0D$ problem as follows [31, 51]. We divide the domain $d(i,j)$ of point (i,j) into three regions: $d(i-1,j)$, $d(i,j-1)$, and $(i-1,j-1)$. (Of course, $d(i-1,j)$ and $d(i,j-1)$ overlap, but this does not affect the minimum.) The minimization among points in $d(i-1,j)$ to compute $E[i,j]$ may be simplified:

$$\begin{aligned}&\min_{i'<i-1,j'<j}\{D[i',j']+w(i'+j',i+j)\}\\&\quad=\min_{i'<i-1,j'<j}\{D[i',j']+w(i'+j',i-1+j)\}+a\\&\quad=E[i-1,j]+a.\end{aligned}$$

Since $E[i,j]$ is the minimum of the minima in three regions,

$$\begin{aligned}E[i,j]&=\min_{i'<i,j'<j}\{D[i',j']+w(i'+j',i+j)\}\\&=\min\{E[i-1,j]+a,\ E[i,j-1]+a,\ D[i-1,j-1]+w(i+j-2,i+j)\},\end{aligned}$$

which is a $2D/0D$ problem.

4. Sparsity

Dynamic programming problems may be sparse. Sparsity arises in the $2D/0D$ problem and $2D/2D$ problem [29, 7, 13]. In sparse problems we need to compute E only for a sparse set S of points. Let M be the number of points in the sparse set S. Since the table size is $O(n^2)$ in the two-dimensional problems, $M \leqslant n^2$. We assume that the sparse set S is given. (In the actual problems S is obtained by a preprocessing [23, 13].)

4.1. The 2D/0D problem

We define the *range* $R(i,j)$ of a point (i,j) to be the set of points (i',j') such that $(i,j) \prec (i',j')$. We also define two subregions of $R(i,j)$: the *upper range* $UR(i,j)$ and the *lower range* $LR(i,j)$. $UR(i,j)$ is the set of points $(i',j') \in R(i,j)$ such that $j'-i' \geqslant j-i$, and $LR(i,j)$ is the set of points $(i',j') \in R(i,j)$ such that $j'-i' \leqslant j-i$.

Consider the case of Recurrence 2 where $x_i = a$, $y_j = b$, $z_{i,j} = 0$ if $(i,j) \in S$; $z_{i,j} = a+b$ otherwise, for constants a and b, and $E[i,j] = D[i,j]$ (i.e. the edit distance problems when replacements are not allowed). Then Recurrence 2 becomes

$$D[i,j] = \min \begin{cases} D[i_1,j] + a(i-i_1) \\ D[i,j_1] + b(j-j_1) \\ \min\limits_{\substack{(i',j') \prec (i,j) \\ (i',j') \in S}} \{D[i',j'] + a(i-i'-1) + b(j-j'-1)\} \quad \text{for } (i,j) \in S, \end{cases} \tag{8}$$

where i_1 is the largest row index such that $i_1 < i$ and $(i_1,j) \in S$, and j_1 is the largest column index such that $j_1 < j$ and $(i,j_1) \in S$. The main computation in Recurrence 8 can be represented by

$$E[i,j] = \min_{\substack{(i',j') \prec (i,j) \\ (i',j') \in S}} \{D[i',j'] + f(i',j',i,j)\} \quad \text{for } (i,j) \in S, \tag{9}$$

where f is a linear function of i', j', i, and j for all points (i,j) in $R(i',j')$. This case will be called the *rectangle case* because $R(i',j')$ is a rectangular region. The rectangle case includes other sparse problems. In the linear $2D/2D$ problem, which is a $2D/0D$ problem, $f(i',j',i,j) = w(i'+j', i+j)$ is a linear function. Eppstein et al. [13] gave an $O(n + M \log\log \min(M, n^2/M))$ time algorithm for the linear $2D/2D$ problem. In the longest common subsequence (LCS) problem, $f(i',j',i,j) = 1$ and $E[i,j] = D[i,j]$, and we are looking for the maximum rather than the minimum. For the LCS problem where S is the set of matching points Hunt and Szymanski [29] gave an $O(M \log n)$ algorithm, which may also be implemented in $O(M \log\log n)$ time with van Emde Boas's data structure [46, 47]. Since f is so simple, the LCS problem can be made even more sparse by considering only *dominant matches* [23, 7]. Apostolico and Guerra [7] gave an algorithm whose time complexity is either $O(n \log n + M' \log(n^2/M'))$ or $O(M' \log\log n)$ depending on data structure, where M' is the number of dominant matches. It was observed in [13] that Apostolico and Guerra's algorithm can be implemented in $O(M' \log\log \min(M', n^2/M'))$ time using Johnson's data structure [30] which is an improvement upon van Emde Boas's data structure.

Consider the case of Recurrence 2 where $x_i = y_j = a$, $z_{i,j} = 0$ if $(i,j) \in S$; $z_{i,j} = a$ otherwise, and $E[i,j] = D[i,j]$ (i.e., the unit-cost edit distance problem [44] and approximate string matching [16]). Recurrence 2 becomes

$$D[i,j] = \min \begin{cases} D[i_1, j] + a(i - i_1) \\ D[i, j_1] + a(j - j_1) \\ \min\limits_{\substack{(i',j') < (i,j) \\ (i',j') \in S}} \{D[i',j'] + a \times \max(i - i' - 1, j - j' - 1)\} \quad \text{for } (i,j) \in S. \end{cases} \tag{10}$$

The main computation in Recurrence 10 can also be represented by Recurrence 9, where f is a linear function in each of $UR(i',j')$ and $LR(i',j')$. This case will be called the *triangle case* because $UR(i',j')$ and $LR(i',j')$ are triangular regions. The fragment alignment with linear gap costs [13] belongs to the triangle case, where $f(i',j',i,j) = w(|(j-i) - (j'-i')|)$. For the fragment alignment problem Wilbur and Lipman [53] proposed an $O(M^2)$ algorithm with M matching fragments. Eppstein et al. [13] improved it by obtaining an $O(n + M \log\log \min(M, n^2/M))$ time algorithm when the gap costs are linear. (They also obtained an $O(n + M \log M)$ algorithm for convex gap costs, and an $O(n + M \log M\alpha(M))$ algorithm for concave gap costs [14].)

4.2. *The rectangle case*

We solve Recurrence 9 when f is a linear function for all points in $R(i',j')$ [13].

Lemma 4.1. *Let P be the intersection of $R(p,q)$ and $R(r,s)$, and (i,j) be a point in P. If $D[p,q] + f(p,q,i,j) < D[r,s] + f(r,s,i,j)$ (i.e. (p,q)* is better than *(r,s) at one point), then (p,q) is better than (r,s) at all points in P.*

Proof. Immediate from the linearity of f. $\square$

By Lemma 4.1, whenever the ranges of two points overlap, we can decide by one comparison which point takes over the overlapping region. Thus the matrix E can be partitioned into regions such that for each region P there is a point (i,j) that is the best for points in P. Obviously, $R(i,j)$ includes P. We refer to (i,j) as the *owner* of P. The partition of E is not known in advance, but we discover it as we proceed row by row. A region P is *active* at row i if P intersects row i. At a particular row active regions are column intervals of that row, and the boundaries are the column indices of the owners of active regions. We maintain the owners of active regions in a list *ACTIVE*.

Let $i_1, i_2, \ldots, i_p$ $(p \leqslant M)$ be the nonempty rows in the sparse set S, and $ROW[s]$ be the sorted list of column indices representing points of row i_s in S. The algorithm consists of p steps, one for each nonempty row. During step s, it processes the points in

$ROW[s]$. Given $ACTIVE$ at step s, the processing of a point (i_s, j) consists of computing $E[i_s, j]$ and updating $ACTIVE$ with (i_s, j). Computing $E(i_s, j)$ simply involves looking up which region contains (i_s, j). Suppose that (i_r, j') is the owner of the region that contains (i_s, j). Then $E[i_s, j] = D[i_r, j'] + f(i_r, j', i_s, j)$. Now we update $ACTIVE$ to possibly include (i_s, j) as an owner. Since $R(i_r, j')$ contains (i_s, j), $R(i_r, j')$ includes $R(i_s, j)$. If (i_r, j') is better than (i_s, j) at $(i_s+1, j+1)$, then (i_s, j) will never be the owner of an active region by Lemma 4.1. Otherwise, we must end the region of (i_r, j') at column j, and add a new region of (i_s, j) starting at column $j+1$. Further, we must test (i_s, j) successively against the owners with larger column indices in $ACTIVE$, to see which is better in their regions. If (i_s, j) is better, the old region is no longer active, and (i_s, j) takes over the region. We continue to test against other owners. If (i_s, j) is worse, we have found the end of its region. Fig. 10 shows the outline of the algorithm.

For each point in S we perform a lookup operation and amortized constant number of insertion and deletion operations on the list $ACTIVE$; altogether $O(M)$ operations. The rest of the algorithm takes time $O(M)$. The total time of the algorithm is $O(M + T(M))$ where $T(M)$ is the time required to perform $O(M)$ insertion, deletion and lookup operations on $ACTIVE$. If $ACTIVE$ is implemented as a balanced search tree, we obtain $T(M) = O(M \log n)$. Column indices in $ACTIVE$ are integers in $[0, n]$, but they can be relabeled to integers in $[0, \min(n, M)]$ in $O(n)$ time. Since $ACTIVE$ contains integers in $[0, \min(n, M)]$, it may be implemented by van Emde Boas's data structure to give $T(M) = O(M \log\log M)$. Even better, by using the fact that for each nonempty row the operations on $ACTIVE$ may be reorganized so that we perform all the lookups first, then all the deletions, and then all the insertions [13], we can use Johnson's data structure to obtain $T(M) = O(M \log\log \min(M, n^2/M))$ as follows. When the numbers manipulated by the data structure are integers in $[1, n]$, Johnson's data structure supports insertion, deletion and lookup operations in $O(\log\log D)$ time, where D is the length of the interval between the nearest integers in the structure below and above the integer that is being inserted, deleted or looked up [30]. With Johnson's data structure $T(M)$ is obviously $O(M \log\log M)$. It remains to show that $T(M)$ is also bounded by $O(M \log\log(n^2/M))$.

Lemma 4.2 (Eppstein et al. [13]). *A homogeneous sequence of $k \leq n$ operations* (i.e. *all insertions, all deletions or all lookups*) *on Johnson's data structure requires $O(k \log\log(n/k))$ time.*

```
procedure sparse 2D/0D
  for $s \leftarrow 1$ to $p$ do
    for each $j$ in $ROW[s]$ do
      look up $ACTIVE$ to get $(i_r, j')$ whose region contains $(i_s, j)$;
      $E[i_s, j] \leftarrow D[i_r, j'] + f(i_r, j', i_s, j)$;
      update $ACTIVE$ with $(i_s, j)$;
    end do
  end do
end
```

Fig. 10. The algorithm of Eppstein et al.

Let m_s for $1 \leqslant s \leqslant p$ be the number of points in row i_s. By Lemma 4.2 the total time spent on row i_s is $O(m_s \log\log(n/m_s))$. The overall time is $O(\sum_{s=1}^{p} m_s \log\log(n/m_s))$. By introducing another summation the sum above is $\sum_{s=1}^{p}\sum_{j=1}^{m_s} \log\log(n/m_s)$. Since $\sum_{s=1}^{p}\sum_{j=1}^{m_s} n/m_s \leqslant n^2$ and $\sum_{s=1}^{p} m_s = M$, we have

$$\sum_{s=1}^{p}\sum_{j=1}^{m_s} \log\log(n/m_s) \leqslant M \log\log(n^2/M)$$

by the concavity of the loglog function. Hence, Recurrence 9 for the rectangle case is solved in $O(n + M \log\log\min(M, n^2/M))$ time. Note that the time is never worse than $O(n^2)$.

4.3. *The triangle case*

We solve Recurrence 9 when f is a linear function in each of $UR(i',j')$ and $LR(i',j')$ [13]. Let f_u and f_l be the linear functions for points in $UR(i',j')$ and $LR(i',j')$, respectively.

Lemma 4.3. *Let P be the intersection of $UR(p,q)$ and $UR(r,s)$, and (i,j) be a point in P. If $D[p,q] + f_u(p,q,i,j) < D[r,s] + f_u(r,s,i,j)$ (i.e. (p,q) is better than (r,s) at one point), then (p,q) is better than (r,s) at all points in P.*

Lemma 4.3 also holds for lower ranges. For each point in S, we divide its range into the upper range and the lower range, and we handle upper ranges and lower ranges separately. Recurrence 9 can be written as

$$E[i,j] = \min\{UI[i,j], LI[i,j]\},$$

where

$$UI[i,j] = \min_{\substack{(i',j') \prec (i,j) \\ j'-i' \leqslant j-i}} \{D[i',j'] + f_u(i',j',i,j)\}, \tag{11}$$

and

$$LI[i,j] = \min_{\substack{(i',j') \prec (i,j) \\ j'-i' \geqslant j-i}} \{D[i',j'] + f_l(i',j',i,j)\}. \tag{12}$$

We compute E row by row. The computation of Recurrence 11 is the same as that of the rectangle case except that regions here are bounded by forward diagonals ($d = j - i$) instead of columns. Recurrence 12 is more complicated since regions are bounded by columns and forward diagonals when we proceed by rows (see Fig. 11).

Let $i_1, i_2, \ldots, i_p$ ($p \leqslant M$) be the nonempty rows in the sparse set S. At step s, we compute LI for row i_s. Assume that we have active regions $P_1, \ldots, P_q$ listed in sorted order of their appearance on row i_s. We keep the owners of these regions in a doubly linked list *OWNER*. Since there are two types of boundaries, we maintain the boundaries of active regions by two lists *CBOUND* (column boundaries) and

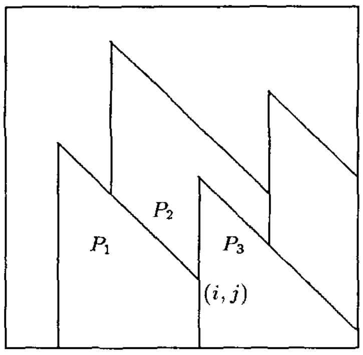

Fig. 11. Regions in the matrix LI.

DBOUND (diagonal boundaries). Each boundary in *CBOUND* and *DBOUND* has a pointer to an element on *OWNER*. Searching for the region containing a point (i_s, j) is then accomplished by finding the rightmost boundary to the left of (i_s, j) in each boundary list, and choosing among the two boundaries the one that is closer to (i_s, j). Suppose that (i_r, j') is the owner of the region that contains (i_s, j). Then $LI[i_s, j] = D[i_r, j'] + f_l(i_r, j', i_s, j)$. Again, $LR(i_r, j')$ includes $LR(i_s, j)$. If (i_r, j') is better than (i_s, j) at $(i_s+1, j+1)$, then (i_s, j) will never be the owner of an active region by Lemma 4.3. Otherwise, we insert (one or two) new regions into *OWNER*, and update *CBOUND* and *DBOUND*.

One complication in Recurrence 12 is that when we have a region bounded on the left by a diagonal and on the right by a column, we must remove it when the row on which these two boundaries meet is processed (P_2 in Fig. 11). A point is a *cut* point if such an active region ends at the point. We keep lists $CUT[i]$, $1 \leqslant i \leqslant n$. $CUT[i]$ contains the cut points of row i. To finish step s, we process all cut points in rows $i_s+1, \ldots, i_{s+1}$. Assume we have processed cut points in rows $i_s+1, \ldots, i-1$. We show how to process $CUT[i]$, $i \leqslant i_{s+1}$. If $CUT[i]$ is empty, we ignore it. Otherwise, we sort the points in $CUT[i]$ by column indices, and process one by one. Let (i, j) be a cut point in $CUT[i]$. Three active regions meet at (i, j). P_1, P_2, and P_3 be the three regions from left to right (see Fig. 11), and (i_1, j_1) and (i_3, j_3) be the owners of P_1 and P_3, respectively. Note that $j = j_3$ and $j - i = j_1 - i_1$. P_2 is no longer active. If (i_1, j_1) is better than (i_3, j_3) at $(i+1, j+1)$, P_1 takes over the intersection of P_1 and P_3. Otherwise, P_3 takes it over.

Lemma 4.4. *The total number of cut points is at most* $2M$.

Proof. Since each point in S creates two boundaries in the matrix LI, $2M$ boundaries are created. Whenever we have a cut point, two boundaries meet and one of them is removed. Therefore, there can be at most $2M$ cut points. □

CBOUND and *DBOUND* are implemented by Johnson's data structure. To sort $CUT[i]$, we also use Johnson's data structure. If there are c_i cut points in row i, sorting $CUT[i]$ can be done in $O(c_i \log\log \min(M, n/c_i))$ time [30]. The total time for sorting is $O(\sum_{i=1}^{n} c_i \log\log\min(M, n/c_i))$, which is $O(M \log\log\min(M, n^2/M))$ since $\sum_{i=1}^{n} c_i \leqslant 2M$. Therefore, Recurrence 12 and Recurrence 9 for the triangle case are solved in $O(n+M \log\log\min(M, n^2/M))$ time.

4.4. The 2D/2D problem

The $2D/2D$ problem is also represented by Recurrence 9, where $f(i',j',i,j)=w(i'+j', i+j)$, and w is either convex or concave. Eppstein et al. [14] introduced the *dynamic minimization* problem which arises as a subproblem in their algorithm: consider

$$E'[y]=\min_x \{D'[x]+w(x,y)\} \quad \text{for } 1 \leqslant x, y \leqslant 2n,$$

where w is either convex or concave. The values of D' are initially set to $+\infty$. We can perform the following two operations:

(a) Compute the value of $E'[y]$ for some y.

(b) Decrease the value of $D'[x]$ for some x.

Note that operation (b) involving one value of $D'[x]$ may simultaneously change $E'[y]$ for many y's.

Recall that the row indices giving the minima for E' are nondecreasing when w is convex and nonincreasing when w is concave. We call a row x *live* if row x supplies the minimum for some $E'[y]$. We maintain live rows and their intervals in which rows give the minima using a data structure. Computing $E'[y]$ is simply looking up which interval contains y. Decreasing $D'[x]$ involves updating the interval structure: deleting some neighboring live rows and finally performing a binary search at each end. Again with a balanced search tree the amortized time per operation is $O(\log n)$. The time may be reduced to $O(\log\log n)$ with van Emde Boas's data structure for simple w that satisfies the closest zero property.

We solve the $2D/2D$ problem by a divide-and-conquer recursion on the rows of the sparse set S. For each level of the recursion, having t points in the subprogram of that level, we choose a row r such that the numbers of points above r and below r are each at most $t/2$. Such a row always exists, and it can be found in $O(t)$ time. Thus we can partition the points into two sets: those above row r, and those on and below row r. Within each level of the recursion, we will need the points of each set to be sorted by their column indices. This can be achieved by initially sorting all points, and then at each level of the recursion performing a pass through the sorted list to divide it into the two sets. Thus the order we need will be achieved at a linear cost per level of the recursion. We compute all the minima by performing the following steps:

(1) recursively solve the problem above row r,

(2) compute the influence of the points above row r on the points on and below row r, and

(3) recursively solve the problem on and below row r.

Let S_1 be the set of points above row r, and S_2 be the set of points on and below row r. The influence of S_1 on S_2 is computed by the dynamic minimization problem. We process the points in S_1 and S_2 in order by their column indices. Within a given column we first process the points of S_2, and then the points of S_1. By proceeding along the sorted lists of points in each set, we only spend time on columns that actually contain points. If we use this order, then when we process a point (i,j) in S_2, the points (i',j') of S_1 that have been processed are exactly those with $j'<j$. To process a point (i,j) in S_1 (let $x=i+j$), we perform the operation of decreasing $D'[x]$ to $\min(D'[x], D[i,j])$. To process a point (i,j) in S_2 (let $y=i+j$), we perform the operation of computing $E'[y]$.

Note that the time per data structure operation can be taken to be $O(\log M)$ or $O(\log\log M)$ rather than $O(\log n)$ or $O(\log\log n)$ because we consider only diagonals that actually contain points in S. Thus the influence of S_1 on S_2 can be computed in $O(M\log M)$ or $O(M\log\log M)$ time depending on w. Since there are $O(\log M)$ levels of recursion, the total time is $O(n+M\log^2 M)$ in general or $O(n+M\log M\log\log M)$ for simple cost functions. We can further reduce the time bounds. We divide E alternately by rows and columns at the center of the matrix rather than the center of the sparse set, similarly to Aggarwal and Park's algorithm for the nonsparse $2D/2D$ problem. With Johnson's data structure and a special implementation of the binary search, $O(n+M\log M\log\min(M, n^2/M))$ or $O(n+M\log M\log\log\min(M, n^2/M))$ for simple w can be obtained [14].

Larmore and Schieber [38] extended their algorithm for the nonsparse $2D/2D$ problem to the sparse case. They achieved $O(n+M\log\min\{M, n^2/M\})$ time for convex w, and $O(n+M\alpha(M)\log\min\{M, n^2/M\})$ time for concave w.

5. Conclusion

We have discussed a number of algorithms for dynamic programming problems with convexity, concavity and sparsity conditions. These algorithms use two outstanding algorithmic techniques: one is matrix searching, and the other is maintaining possible candidates in a data structure. Sometimes combinations of these two techniques result in efficient algorithms. Divide-and-conquer is also useful with the two techniques.

As mentioned earlier, though the condition we have for convexity or concavity is the Monge condition, all algorithms are valid when the weaker condition of total monotonicity holds. It remains open whether there are better algorithms that actually use the Monge condition. Another notable open problem is whether there exists a linear time algorithm for the concave $1D/1D$ problem. Recently, Larmore [37] gave an algorithm for the concave $1D/1D$ problem which is optimal for the decision tree model, but whose time complexity is unknown.

The (single-source) shortest-path problem is a well-known dynamic programming problem. By the Bellman–Ford formulation it is a $2D/1D$ problem, and by the

Floyd–Warshall formulation it is a $3D/0D$ problem [39]. Though Dijkstra's algorithm takes advantage of sparsity of graphs, it is not covered in this paper since comprehensive treatments may be found in [43, 39]. We should also mention two important dynamic programming problems that do not share the conditions we considered: the matrix chain product [4] and the membership for context-free grammars [26]. The matrix chain product is a $2D/1D$ problem, so an $O(n^3)$ time algorithm is obvious [4]. Further improvement is possible by the observation that the problem is equivalent to the optimal triangulation of a polygon. An $O(n^2)$ algorithm is due to Yao [56], and an $O(n \log n)$ algorithm is due to Hu and Shing [27, 28]. The membership for context-free grammars can be formulated as a $2D/1D$ problem. The Cocke–Younger–Kasami algorithm [26] is an $O(n^3)$ time algorithm based on the dynamic programming formulation. Asymtotically better algorithms were developed: an $O(M(n))$ algorithm by Valiant [45] and an $O(n^3/\log n)$ algorithm by Graham, et al. [19], where $M(n)$ is the time complexity of matrix multiplication.

Since dynamic programming has numerous applications in various fields, we may find more problems which fit into our four dynamic programming problems. It may also be worthwhile to find other conditions that lead to efficient algorithms.

Acknowledgment

We thank David Eppstein and Pino Italiano for their helpful comments. We also thank Terry Boult and Henryk Wozniakowski for reading an early version of this paper.

References

[1] A. Aggarwal and M.M. Klawe, Applications of generalized matrix searching to geometric algorithms, *Discrete Applied Math.* **27** (1990) 3–24.
[2] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor and R. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica* **2** (1987) 195–208.
[3] A. Aggarwal and J. Park, Notes on searching in multidimensional monotone arrays, in: *Proc. 29th IEEE Symp. Found. Computer Science* (1988) 497–512.
[4] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).
[5] A.V. Aho, D.S. Hirschberg and J.D. Ullman, Bounds on the complexity of the longest common subsequence problem, *J. Assoc. Comput. Mach.* **23** (1976) 1–12.
[6] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms* (Addison-Wesley, Reading, MA, 1983).
[7] A. Apostolico and C. Guerra, The longest common subsequence problem revisited, *Algorithmica* **2** (1987) 315–336.
[8] R.E. Bellman, *Dynamic Programming* (Princeton University Press, Princeton, NJ, 1957).
[9] R.E. Bellman, Dynamic programming treatment of the traveling salesman problem, *J. Assoc. Comput. Mach.* **9** (1962) 61–63.
[10] R.S. Bird, Tabulation techniques for recursive programs, *Computing Surveys* **12** (1980) 403–417.

[11] D. Eppstein, Sequence comparison with mixed convex and concave costs, *J. Algorithms* **11** (1990) 85–101.

[12] D. Eppstein, Z. Galil and R. Giancarlo, Speeding up dynamic programming, in: *Proc. 29th IEEE Symp. Found. Computer Science* (1988) 488–496.

[13] D. Eppstein, Z. Galil, R. Giancarlo and G.F. Italiano, Sparse dynamic programming I: linear cost functions, *J. Assoc. Comput. Mach.*, to appear.

[14] D. Eppstein, Z. Galil, R. Giancarlo and G.F. Italiano, Sparse dynamic programming II: convex and concave cost functions, *J. Assoc. Comput. Mach.*, to appear.

[15] Z. Galil and R. Giancarlo, Speeding up dynamic programming with applications to molecular biology, *Theoret. Comput. Sci.* **64** (1989) 107–118.

[16] Z. Galil and K. Park, An improved algorithm for approximate string matching, in: *Proc. 16th ICALP*, Lecture Notes in Computer Science, Vol. 372 (Springer, Berlin, 1989) 394–404.

[17] Z. Galil and K. Park, A linear-time algorithm for concave one-dimensional dynamic programming, *Inform. Process. Lett.* **33** (1990) 309–311.

[18] O. Gotoh, An improved algorithm for matching biological sequences, *J. Mol. Biol.* **162** (1982) 705–708.

[19] S.L. Graham, M.A. Harrison and W.L. Ruzzo, On-line context-free language recognition in less than cubic time, in: *Proc. 8th ACM Symp. Theory of Computing* (1976) 112–120.

[20] M. Held and R.M. Karp, A dynamic programming approach to sequencing problems, *SIAM J. Applied Math.* **10** (1962) 196–210.

[21] P. Helman and A. Rosenthal, A comprehensive model of dynamic programming, *SIAM J. Alg. Dics. Meth.* **6** (1985) 319–334.

[22] D.S. Hirschberg, A linear-space algorithm for computing maximal common subsequences, *Comm. ACM* **18**(6) (1975) 341–343.

[23] D.S. Hirschberg, Algorithms for the longest common subsequence problem, *J. Assoc. Comput. Mach.* **24** (1977) 664–675.

[24] D.S. Hirschberg and L.L. Larmore, The least weight subsequence problem, *SIAM J. Comput.* **16**(4) (1987) 628–638.

[25] A.J. Hoffman, On simple linear programming problems, in: *Proc. Symposia in Pure Math., Vol. 7*: Convexity (Amer. Math. Soc., Providence, RI, 1963) 317–327.

[26] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, MA, 1979).

[27] T.C. Hu and M.T. Shing, Computation of matrix chain products. Part I, *SIAM J. Comput.* **11** (1982) 362–373.

[28] T.C. Hu and M.T. Shing, Computation of matrix chain products. Part II, *SIAM J. Comput.* **13** (1984) 228–251.

[29] J.W. Hunt and T.G. Szymanski, A fast algorithm for computing longest common subsequences, *Comm. ACM* **20** (1977) 350–353.

[30] D.B. Johnson, A priority queue in which initialization and queue operations take $O(\log\log D)$ time, *Math. Systems Theory* **15** (1982) 295–309.

[31] M.I. Kanehisa and W.B. Goad, Pattern recognition in nucleic acid sequences II: an efficient method for finding locally stable secondary structures, *Nucl. Acids Res.* **10** (1982) 265–277.

[32] R.M. Karp and M. Held, Finite-state processes and dynamic programming, *SIAM J. Applied Math.* **15** (1967) 693–718.

[33] M.M. Klawe, Speeding up dynamic programming, preprint, 1987.

[34] M.M. Klawe, A simple linear-time algorithm for concave one-dimensional dynamic programming, preprint, 1990.

[35] M.M. Klawe and D.J. Kleitman, An almost linear time algorithm for generalized matrix searching, Tech. Report, IBM Almaden Research Center, 1988.

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

[37] L.L. Larmore, An optimal algorithm with unknown time complexity for convex matrix searching, *Inform Process. Lett.*, to appear.

[38] L.L. Larmore and B. Schieber, On-line dynamic programming with applications to the prediction of RNA secondary structure, *J. Algorithms*, to appear.

[39] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids* (Holt, Rinehart and Winston, New York, 1976).
[40] W. Miller and E.W. Myers, Sequence comparison with concave weighting functions, *Bull. Math. Biol.* **50** (1988) 97–120.
[41] G. Monge, *Déblai et remblai* (Mémoires de l'Académie des Sciences, Paris, 1781).
[42] S.B. Needleman and C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, *J. Mol. Biol.* **48** (1970) 443–453.
[43] R.E. Tarjan, *Data Structures and Network Algorithms* (SIAM, Philadelphia, PA, 1983).
[44] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control* **64** (1985) 100–118.
[45] L.G. Valiant, General context-free recognition in less than cubic time, *J. Comput. Systems Sci.* **10** (1975) 308–315.
[46] P. van Emde Boas, Preserving order in a forest in less than logarithmic time, in: *Proc. 16th IEEE Symp. Found. Computer Science* (1975) 75–84.
[47] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* **6** (1977) 80–82.
[48] M.L. Wachs, On an efficient dynamic programming technique of F.F. Yao, *J. Algorithms* **10** (1989) 518–530.
[49] R.A. Wagner and M.J. Fischer, The string-to-string correction problem, *J. Assoc. Comput. Mach.* **21** (1974) 168–173.
[50] M.S. Waterman and T.F. Smith, RNA secondary structure: a complete mathematical analysis. *Math. Biosciences* **42** (1978) 257–266.
[51] M.S. Waterman and T.F. Smith, Rapid dynamic programming algorithms for RNA secondary structure, *Advances in Applied Math.* **7** (1986) 455–464.
[52] R. Wilber, The concave least-weight subsequence problem revisited, *J. Algorithms* **9** (1988) 418–425.
[53] W.J. Wilbur and D.J. Lipman, The context-dependent comparison of biological sequences, *SIAM J. Applied Math.* **44** (1984) 557–567.
[54] C.K. Wong and A.K. Chandra, Bounds for the string editing problem, *J. Assoc. Comput. Mach.* **23** (1976) 13–16.
[55] F.F. Yao, Efficient dynamic programming using quadrangle inequalities, in: *Proc. 12th ACM Symp. Theory of Computing* (1980) 429–435.
[56] F.F. Yao, Speed-up in dynamic programming, *SIAM J. Alg. Disc. Meth.* **3** (1982) 532–540.