Theoretical Computer Science 64 (1989) 107–118
North-Holland

NOTE

SPEEDING UP DYNAMIC PROGRAMMING WITH APPLICATIONS TO MOLECULAR BIOLOGY*

Zvi GALIL

Department of Computer Science, Columbia University, NY 10027, U.S.A., and Tel-Aviv University, Tel-Aviv, Israel

Raffaele GIANCARLO

Department of Computer Science, Columbia University, NY 10027, U.S.A., and University of Salerno, Salerno, Italy

Communicated by G. Ausiello
Received July 1987
Revised May 1988

Abstract. Consider the problem of computing

$$E[j]=\min_{0\le k\le j-1}\{D[k]+w(k,j)\},\quad j=1,\ldots,n,$$

where w is a given weight function, $D[0]$ is given and for every $k=1,\ldots,n$, $D[k]$ is easily computable from $E[k]$. This problem appears as a subproblem in dynamic programming solutions to various problems. Obviously, it can be solved in time $O(n^2)$, and for a general weight function no better algorithm is possible.

We consider two dual cases that arise in applications: in the *concave case*, the weight function satisfies the quadrangle inequality:

$$w(k,j)+w(l,j')\le w(l,j)+w(k,j')\quad \text{for all } k\le l\le j\le j'.$$

In the *convex case*, the weight function satisfies the inverse quadrangle inequality.

In both cases we show how to use the assumed property of w to derive an $O(n\log n)$ algorithm. Even better, linear-time algorithms are obtained if w satisfies the following additional *closest zero property*: for every two integers l and k, $l<k$, and real number a, the smallest zero of

$$f(x)=w(l,x)-w(k,x)-a$$

which is larger than k can be found in constant time.

The two algorithms speed up several dynamic programming routines that solve as a subproblem the problem above. The speed-up is from $O(n^3)$ to $O(n^2\log n)$ or $O(n^2)$. Applications include algorithms for comparing DNA sequences and algorithms used in speech recognition and geology.

One typical problem is the following: given the cost of substituting any pair of symbols and a convex cost function g for gaps (where $g(r)$ is the cost of a gap of size r), compute the modified edit distance between the two given sequences.

* Work supported in part by NSF Grants DCR-85-11713, CCR-86-05353 and by the Italian Ministry of Education, Project "Teoria degli Algoritmi".

1. Introduction

Dynamic programming is one of several widely used problem-solving techniques in computer science and operations research. In applying the technique, one always seeks to take advantage of special properties of the problem at hand and speed up the algorithm. There are very few general techniques for speeding up dynamic programming routines and ad hoc approaches seem to be characteristic.

The only general technique known to us is due to Yao [12]. She considered the following recurrence relations:

$$c(i, i) = 0; \qquad c(i, j) = w(i, j) + \min_{i<k\leq j} \{c(i, k-1) + c(k, j)\} \quad \text{for } i<j. \tag{1}$$

She proved that if the weight function satisfies the quadrangle inequality

$$w(k, j) + w(l, j') \leq w(l, j) + w(k, j'), \quad \text{for all } k \leq l \leq j \leq j', \tag{2}$$

then the obvious $O(n^3)$ algorithm can be sped up to $O(n^2)$. A corollary of this result is an $O(n^2)$ algorithm for computing optimum binary search trees, an earlier remarkable result of Knuth [5].

In this paper we consider the problem of computing

$$E[j] = \min_{0\leq k\leq j-1} \{D[k] + w(k, j)\}, \quad j = 1, \ldots, n, \tag{3}$$

where w is a given weight function; $D[0]$ is given and for every $k = 1, \ldots, n$, $D[k]$ is easily computable from $E[k]$. This problem appears as a subproblem in dynamic programming solutions to various problems. Obviously, it can be solved in time $O(n^2)$, and for a general weight function no better algorithm is possible.

We consider two dual cases that arise in applications: in the *concave case*, the weight function satisfies the quadrangle inequality above. In the *convex case*, the weight function satisfies the inverse quadrangle inequality.

In both cases we show how to use the assumed property of w to derive an $O(n \log n)$ algorithm. Even better, linear-time algorithms are obtained if w satisfies the following additional property, which we call the *closest zero property*: for every two integers l and k, $l<k$, and real number a, the smallest zero of

$$f(x) = w(l, x) - w(k, x) - a$$

which is larger than k can be found in constant time. The two algorithms are simple and short (a dozen lines of code each).

Surprisingly, the two algorithms are also dual in the following sense: both work in stages. In the jth stage they compute $E[j]$, which is viewed as a competition among indices $0, 1, \ldots, j-1$ for the minimum in (3). They maintain a set of candidates which satisfies the property that $E[j]$ depends only on $D[k] + w(k, j)$ for k's in the set. Moreover, each algorithm discards candidates from the set, and discarded candidates never rejoin the set. To be able to maintain such a set of candidates efficiently one uses the following "dual" data structures: a queue in the concave case and a stack in the convex case.

Despite the similarity, Yao's result does not seem to apply in our case. She uses the quadrangle inequality to derive tighter bounds on the ranges of indices for computing the minimum in (1) and does not need any data structure. Also, it is not clear what happens in her case when the inverse quadrangle inequality holds.

Notice that in the special case that $D[j]=E[j]$ and $w(j, j)=0$ for $j=1, \ldots, n$ our problem is the single source shortest path problem for the complete acyclic graph where edge lengths are given by the weight function w. However, neither the convex nor the concave case is interesting, since the quadrangle inequality implies the inverse triangle inequality and the inverse quadrangle inequality implies the triangle inequality. Thus in the convex case $E[j]=D[0]+w(0, j)$ and in the concave case

$$E[j]=D[0]+w(0,1)+w(1,2)+\cdots+w(j-1, j).$$

We use recurrence (3) to compute various versions of the "modified edit distance" defined below. Given two strings over alphabet Σ, $x=x_1 \ldots x_m$ and $y=y_1 \ldots y_n$, the *edit distance* of x and y is the minimal cost of an edit sequence that changes x into y. This sequence contains operations of the form **delete**(x_i) of cost $d(x_i)$, **insert**(y_j) of cost $i(y_j)$ and **substitute**(x_l, y_k) of cost $s(x_l, y_k)$. The edit distance can be easily computed by an obvious dynamic program in $O(mn)$ time.

Notice that a sequence of deletes (inserts) corresponds to a gap in x (y, respectively). In many applications the cost of such a gap is not linear. In the applications we list below the cost of deleting $x_{l+1} \ldots x_k$ is

$$w(l, k)=f^1(x_l, x_{l+1})+f^2(x_k, x_{k+1})+g(k-l), \tag{4}$$

where g is a convex function. The cost consists of charges for breaking x_{l+1} and x_k plus an additional cost that depends on the length of the gap. The dependence (the function g) is convex, since the incremental cost decreases as the size of the gap increases. The *modified edit distance* is defined to be the minimal cost of an edit sequence which changes x into y, where the costs of gaps in x are as in (4), and similarly the costs of gaps in y are derived from an analogous weight function w'.

To compute the modified edit distance, we consider a dynamic programming equation of the form

$$D[i, j]=\min\{D[i-1, j-1]+s(x_i, y_j), E[i, j], F[i, j]\} \tag{5}$$

where

$$E[i, j]=\min_{0 \leq k \leq j-1}\{D[i, k]+w(k, j)\}, \tag{6}$$

$$F[i, j]=\min_{0 \leq l \leq i-1}\{D[l, j]+w'(l, i)\}, \tag{7}$$

with initial conditions

$$D[i, 0]=w'(0, i),\ 1 \leq i \leq m \quad \text{and} \quad D[0, j]=w(0, j),\ 1 \leq j \leq n.$$

The obvious algorithm that computes this recurrence takes time $O(mn \cdot \max(m, n))$. Notice that the computation of $D[i, j]$ reduces to the computation of $E[i, j]$ and

$F[i, j]$, and the computation of a row of E and of a column of F are each just the same as the problem discussed above. But the weight functions w and w' are defined as in (4) and therefore satisfy the inverse quadrangle inequality. As a result, we obtain an algorithm that computes the matrix D in $O(mn(\log n + \log m))$ time, and even better $O(mn)$ time if the weight functions satisfy the closest zero property.

This dynamic programming scheme arises in the context of sequence comparison in molecular biology [10], geology [9] and in speech recognition [7]. In those fields, the most natural weight functions w are convex. In molecular biology, for instance, the motivation for the use of convex weight functions is the following. When a DNA sequence evolves into another by means of the deletion, say, of some contiguous bases, the event "deletion of contiguous bases" should be seen as a single event rather than as the combination of many "deletion" events. Accordingly, the cost of the multiple deletions event must be less than the total cost of the single deletion events composing it. Experimental evidence supports this theory [2]. In geology and speech recognition, analogous reasoning motivates the use of convex weight functions.

For the concave case, good algorithms were already known. Hirshberg and Larmore [3] assumed a restricted quadrangle inequality with $k \leq l < j \leq j'$ in (2) that does not imply the inverse triangle inequality. They solved the "least weight subsequence", with $D[j] = E[j]$, in time $O(n \log n)$. Such a time bound becomes $O(n)$ in case a certain condition (similar to our closest zero property) is satisfied by the weight function. They used this result to derive improved algorithms for several problems. Their main application is an $O(n \log n)$ algorithm for breaking a paragraph into lines with a concave penalty function. For quadratic and linear penalty functions they design a linear-time algorithm. This problem was considered first by Knuth and Plass [6] with general penalty functions. The algorithm of Hirshberg and Larmore, like our algorithm, uses a queue. Surprisingly, our algorithm, which solves a more general case, is slightly simpler and in many cases faster, as in our algorithm the queue is sometimes emptied in a single operation.

Wilber [11] obtained an ingenious $O(n)$ algorithm, also for the concave case, based on previous work by Aggarwal et al. [1]. His algorithm is recursive and its recursive calls use another recursive algorithm, so the constant factor in the time bound is quite large. Wilber claims that his algorithm is superior to our $O(n \log n)$ one only for n in the thousands. We remark that Wilber's technique does not seem to be useful in the convex case, which is the main subject of this paper.

2. The convex case

In this section we describe the convex case. That is, the weight function w satisfies the inverse quadrangle inequality:

$$w(l, j) + w(k, j') \geq w(k, j) + w(l, j'), \quad \text{for all } l \leq k \leq j \leq j'. \tag{8}$$

Let $C(k, r)$ denote $D[k]+w(k, r)$. We give an algorithm, denoted A, that computes $E[j]$, $0<j\leq n$, in $O(n \log n)$ time. The algorithm consists of n steps. We now describe step j. We need the following definition.

Definition 1. An index k, $k<j$, is dead (at step j) if for every j', $j\leq j'\leq n$, either there is $l\neq k$, $l<j$ (l may depend on j') with $C(l, j')<C(k, j')$ or there is $l<k$ with $C(l, j')=C(k, j')$.

Algorithm A implicitly maintains a list of candidates k, $k<j$. These candidates compete for the minimum in (3). Then the algorithm justifiably discards dead indices from the list since it is clear that for every j', $j\leq j'\leq n$, the minimum in the expression for $E[j']$ corresponding to (3) is achieved on some k that has not been discarded. There are two criteria that the algorithm uses to kill k's. In the first, an index k dies because of one of the candidates which will always dominate it ($l(j')=l$ for each j', $j\leq j'\leq n$), and in the second case k will be dominated by one of two candidates ($l(j')\in\{l_1, l_2\}$ for each j', $j\leq j'\leq n$).

A basic step of the algorithm consists of comparing two candidates l and k. Lemma 1 is the basis for such a comparison.

Lemma 1. *For given l and k, $l<k\leq n$, the function*

$$f(r)=C(l, r)-C(k, r)=w(l, r)-w(k, r)+D[l]-D[k]$$

is monotonically nonincreasing.

Proof. Straightforward from equation (8). □

Facts 1–4 are immediate consequences of Lemma 1.

Fact 1. *Given indices l and k, $l<k<j\leq n$, assume that $C(l, j)\leq C(k, j)$. Then $C(l, j')\leq C(k, j')$ for all j' satisfying $j\leq j'\leq n$.*

Fact 2. *Given indices l and k, $l<k<j\leq n$, assume that $C(l, j)>C(k, j)$. Let h be the minimal index, $k<h\leq n$, such that*

$$C(l, h)\leq C(k, h). \tag{9}$$

Then $C(l, j')>C(k, j')$, for all j' satisfying $k<j'<h$ and $C(l, j')\leq C(k, j')$, for all j' satisfying $h\leq j'\leq n$.

In what follows, we adopt the shorthand notation $h(l, k)$, for the minimal h, $k<h\leq n$, satisfying equation (9) for indices l and k, $l<k<n$ and we set $h(l, k)=n+1$ if no such h exists.

Fact 3. *For given l and k, $l<k<n$, $h(l,k)\leq \bar{h}$ if and only if $C(l,\bar{h})\leq C(k,\bar{h})$.*

Fact 4. *Given l and k, $l<k\leq n$, $h(l,k)$ can be computed in time $O(\log n)$ and, if w satisfies the closest zero property, it can be computed in constant time.*

Proof. By Fact 2 we can compute $h(l,k)$ by binary search. If the closest zero property holds, then one computes $h(l,k)$ by finding the closest zero, x_0, of

$$f(x)=w(l,x)-w(k,x)-a, \quad a=D[l]-D[k],$$

and $h(l,k)=\lceil x_0\rceil$. □

The meaning of Fact 1 is that if a newer candidate k is no better than an older candidate l, then k is dead. The meaning of Fact 2 is that if the newer candidate is better we know that it can be declared dead at step $h=h(l,k)$. Moreover, we also know that k is a better candidate than l for steps $j, j+1,\ldots,h-1$. Thus, there is no need to consider l in the competition for the minimum during these steps.

The algorithm compares the new candidate, namely $j-1$, with the best among the old candidates. If the new candidate is no better, then Fact 1 allows us to discard it. If it is better, Fact 2 tells us that it is better in an interval at the end of which the new candidate will die. This gives rise to the use of a stack.

The list of candidates at step j is represented in a stack S of pairs (corresponding to intervals) $(k_{\text{top}}, h_{\text{top}}), (k_{\text{top}-1}, h_{\text{top}-1}), \ldots, (k_0, h_0)$, where (k_0, h_0) is a dummy pair equal to $(k_0, n+1)$. At step j, the pairs in S satisfy the following conditions, for $0\leq s<\text{top}$,

(1) $h_{s+1}<h_s$, with $j<h_{\text{top}}$ and $h_0=n+1$,
(2) $k_s<k_{s+1}$, with $k_{\text{top}}<j-1$,
(3) $h_{s+1}=h(k_s,k_{s+1})$,
(4) if $k<j$ and k is not in any pair in S, then k is dead,
(5) the last element in S is always a dummy pair.

Conditions (1)–(5) are referred to as the *stack property*. Part (1) and (2) of the stack property mean that the stack consists of a sequence of open intervals, properly nested on both sides, all of which contain j. By Fact 2 and condition (3), the meaning of adjacent pairs (k_{s+1}, h_{s+1}) and (k_s, h_s) is that k_{s+1} is a better candidate than k_s up to step $h_{s+1}-1$. At step h_{s+1}, k_{s+1} can be declared dead since from that point on k_s is never worse than k_{s+1}. The meaning of the dummy pair $(k_0, n+1)$ is that A does not know yet how long k_0 is going to survive.

We now show that if the stack property holds at step j, we can easily compute $E[j]$. Then, we discuss how the algorithm updates S so that the stack property holds at step $j+1$.

Lemma 2. *Assume that $|S|>1$. The quantities $C(k_s, j)$ are monotonically increasing as we go down the stack.*

Proof. Fix s, $0\leq s<\text{top}$. From Fact 2 we have that for each adjacent pair (k_s, h_s) and (k_{s+1}, h_{s+1})

$$C(k_s, j')>C(k_{s+1}, j'),$$

for j' satisfying $k_{s+1} < j' < h_{s+1}$. By conditions (1) and (2) of the stack property $k_{s+1} < j < h_{s+1}$ and thus $C(k_s, j) > C(k_{s+1}, j)$. □

Lemma 2 implies that k_{top} is the best among the old candidates. Thus, $E[j]$ is the minimum of $C(j-1, j)$ and $C(k_{top}, j)$. (In the case that $|S| = 1$, the same result follows since $j-1$ and k_{top} are the only two candidates.)

We now describe the update of S that follows the computation of $E[j]$. Informally, it consists of the insertion of the new candidate $j-1$, if appropriate, and of the possible deletion of some "old" candidates. The update step depends on the outcome of the comparison between $C(k_{top}, j)$ and $C(j-1, j)$.

When $C(k_{top}, j) \leq C(j-1, j)$, the update of S is very simple. Indeed, algorithm A can kill $j-1$ by Fact 1. When $C(k_{top}, j) > C(j-1, j)$, the algorithm tries to push a pair corresponding to $j-1$ on the stack. However, the new interval (in which $j-1$ is better than the current k_{top}) may not be properly nested in (k_{top}, h_{top}), i.e. the new interval may end at $h > h_{top} - 1$. By Fact 3, we can check it by comparing $C(k_{top}, h_{top} - 1)$ and $C(j-1, h_{top} - 1)$ (i.e. comparing the two candidates at the last point of the interval of k_{top}). Fortunately, if the nesting property is violated, then k_{top} can be declared dead as shown in Fact 5.

Fact 5. *Let* (k_{top}, h_{top}) *be the pair on top of* S *at step* j *and assume that* $C(k_{top}, h_{top} - 1) > C(j-1, h_{top} - 1)$. *Then* k_{top} *can be declared dead at step* j.

Proof. By the assumption and Fact 1 we have $C(k_{top}, j') > C(j-1, j')$ for j' satisfying $j \leq j' \leq h_{top} - 1$. If $|S| = 1$, $h_{top} = n+1$ and the proof is complete. Otherwise, by Fact 1 (since $k_{top-1} < k_{top}$) $C(k_{top-1}, j') \leq C(k_{top}, j')$ for j' satisfying $h_{top} \leq j' \leq n$, and k_{top} is dead (it is dominated either by $j-1$ or by k_{top-1}). □

When $C(k_{top}, j) > C(j-1, j)$, the update of S is as follows. Algorithm A keeps popping pairs (k_s, h_s) when $C(k_s, h_s - 1) > C(j-1, h_s - 1)$. The deletion process stops either when the stack is empty or when the algorithm finds a pair (k_q, h_q) such that $C(k_q, h_q - 1) \leq C(j-1, h_q - 1)$. If the former case holds, the algorithm inserts the dummy pair $(j-1, n+1)$ and ends the update of S. If the latter case holds, the algorithm computes $\bar{h} = h(k_q, j-1)$ and pushes $(j-1, \bar{h})$ on top of S. (Note that, since

$$C(k_q, h_q - 1) \leq C(j-1, h_q - 1),$$

$\bar{h} \leq h_q - 1$ by Fact 3.) We notice that all the first components of the popped pairs are k's which are dead by Fact 5.

Once S has been updated as described above, we may have to pop one more pair from its top. Indeed, if $h_{top} = j+1$ we have by Fact 1 that k_{top} can be declared dead at step $j+1$. Thus, the algorithm kills k_{top} by popping the stack. If $j+1 < h_{top}$, the stack is not modified since k_{top} can still be a candidate.

In what follows let $K(r)$ and $H(r)$ denote the first and second component of the rth pair from the bottom in S. The algorithm described above can be formalized as follows.

Algorithm A
push $(0, n+1)$ on S;
for $j=1$ **to** n **do**
begin
  $l \leftarrow K(\text{top})$;
  if $C(j-1, j) \geqslant C(l, j)$ **then** $E[j] \leftarrow C(l, j)$;
  else
  begin
    $E[j] \leftarrow C(j-1, j)$;
    while $S \neq \emptyset$ **and** $C(j-1, H(\text{top})-1) < C(K(\text{top}), H(\text{top})-1)$ **do pop**
    if $S=\emptyset$ **then push** $(j-1, n+1)$
      else $h \leftarrow h(K(\text{top}), j-1)$; **push** $(j-1, h)$
  end
  if $H(\text{top})=j+1$ **then pop**
end.

Theorem 1. *Algorithm* **A** *is correct and runs in time* $O(n \log n)$. *If w satisfies the closest zero property, then one can implement the algorithm in linear time.*

Proof. The correctness of the algorithm can be easily proved by induction using the discussion on the update of the stack at step j. The time bound can be derived as follows. Notice that for each index j, $1 \leqslant j \leqslant n$, there may be a computation of $h(K(\text{top}), j)$ when a pair corresponding to j is pushed onto the stack. Since each index can be pushed on the stack only once and since, by Fact 4, the computation of $h(K(\text{top}), j)$ takes $O(\log n)$, we obtain a time bound of $O(n \log n)$. If w satisfies the closest zero property, then the computation $h(K(\text{top}), j)$ takes constant time per call and the above time bound reduces to $O(n)$. □

3. The concave case

In this section we describe the concave case, omitting the proofs since they are analogous to the ones given in the previous section. In the concave case, the weight function satisfies the quadrangle inequality (2). We given an algorithm, denoted **B**, that computes $E[j]$, $0 < j \leqslant n$, in $O(n \log n)$ time. The algorithm consists of n steps. We now describe step j. We need the following definition.

Definition 2. An index k, $k < j$, is dead (at step j) if for every j', $j \leqslant j' \leqslant n$, either there is $l \neq k$, $l < j$ (l may depend on j') with $C(l, j') < C(k, j')$, or there is $l > k$ with $C(l, j') = C(k, j')$.

Notice that Definitions 1 and 2 are the same except that now we break ties in favor of the larger index.

Algorithm B implicitly maintains a list of candidates k, $k<j$. These candidates compete for the minimum in expression (3). Then the algorithm discards dead indices from the list since it is clear that for every j', $j \leq j' \leq n$, the minimum in the expression for $E[j']$ corresponding to (3) is achieved on some k that has not been discarded. Again, there are two criteria that the algorithm uses to kill k's. These criteria are analogous to the ones given in the previous section.

A basic step of the algorithm consists of comparing two candidates l and k. Lemma 3 is the basis for such a comparison.

Lemma 3. *For given l and k, $k<l\leq n$,*

$$f(r)=C(l,r)-C(k,r)=w(l,r)-w(k,r)+D[l]-D[k]$$

is monotonically nondecreasing.

Facts 6–9 are an immediate consequence of Lemma 3.

Fact 6. *Given indices l and k, $k<l<j\leq n$, assume that $C(l,j)\leq C(k,j)$. Then $C(l,j')\leq C(k,j')$ for j' satisfying $j\leq j'\leq n$.*

Fact 7. *Given indices l and k, $k<l<j\leq n$, assume that $C(l,j)>C(k,j)$. Let h be the minimal index, $l<h\leq n$, such that*

$$C(l,h)\leq C(k,h). \tag{10}$$

Then $C(l,j')>C(k,j')$, for j' satisfying $l<j'<h$ and $C(l,j')\leq C(k,j')$ for j' satisfying $h\leq j'\leq n$.

In what follows, we adopt the shorthand notation $h(l,k)$, for the minimal h, $l<h\leq n$, satisfying equation (10) for indices l and k, $k<l<n$, and we set $h(l,k)=n+1$ if no such h exists.

Fact 8. *For given l and k, $k<l<n$, $h(l,k)\leq \bar{h}$ if and only if $C(l,\bar{h})\leq C(k,\bar{h})$.*

Fact 9. *Given l and k, $k<l\leq n$, $h(l,k)$ can be computed in time $O(\log n)$ and, if w satisfies the closest zero property, it can be computed in constant time.*

The meaning of Facts 6 and 7 is analogous to the meaning of Facts 1 and 2, respectively, with the role of l and k switched.

The algorithm compares the new candidate, namely $j-1$, with the best among the old candidates. If the new candidate is better, then Fact 6 allows us to discard all the old candidates. If it is no better, Fact 7 tells us that it will be better in an interval at the beginning of which the old candidate will die. This gives rise to the use of a queue.

The list of candidates at step j is represented in a queue Q of pairs

$$(k_{\text{front}}, h_{\text{front}}), (k_{\text{front}-1}, h_{\text{front}-1}), \ldots, (k_0, h_0),$$

where $(k_{\text{front}}, h_{\text{front}})$ is a dummy pair with $h_{\text{front}} = j$. At step j, the pairs in Q satisfy the following conditions, for $0 \leq s < \text{front}$:

(1) $h_{s+1} < h_s$, with $h_{\text{front}} = j$.
(2) $k_s > k_{s+1}$, with $k_0 < j-1$.
(3) $h_s = h(k_s, k_{s+1})$.
(4) If $k < j$ and k is not in any pair in Q, then k is dead.
(5) The first element in Q is always a dummy pair.

In what follows, we refer to conditions (1)–(5) as *the queue property*. Parts (1) and (2) of the queue property mean that the queue consists of a sequence of open intervals, properly nested on both sides, all containing j except for the dummy interval. The meaning of adjacent pairs (k_{s+1}, h_{s+1}) and (k_s, h_s) is that k_{s+1} is a better candidate than k_s up to step $h_s - 1$. At step h_s, k_{s+1} can be declared dead since from that point on k_s is never worse than k_{s+1}. The meaning of the dummy pairs (k_{front}, j) is that k_{front} has no index in front of it to kill.

We now show that if the queue property holds at step j, we can easily compute $E[j]$. Then, we discuss how the algorithm updates Q so as to preserve the queue property at step $j+1$.

Lemma 4. *Assume that $|Q| > 1$. The quantities $C(k_s, j)$ are monotonically increasing as we go along the queue from the front to the rear.*

Lemma 4 implies that k_{front} is the best among the old candidates. Thus, $E[j]$ is the minimum of $C(j-1, j)$ and $C(k_{\text{front}}, j)$.

Next, we describe the update of Q that follows the computation of $E[j]$. Informally, it consists of the insertion of the new candidate $j-1$, if appropriate, and of the possible deletion of some "old" candidates. The update step depends on the outcome of the comparison between $C(k_{\text{front}}, j)$ and $C(j-1, j)$.

When $C(k_{\text{front}}, j) \geq C(j-1, j)$, the update of Q is very simple. Indeed, $k_0 < j-1$, $k_s > k_{s+1}$ and, by Lemma 4, the quantities $C(k_s, j)$ are monotonically increasing as we go down the queue. By Fact 6, all the k_s in Q can be declared dead. The algorithm sets $Q = \emptyset$, i.e. it discards all elements in the queue, and inserts the dummy pair $(j-1, j+1)$. The operation $Q = \emptyset$ can obviously be implemented in constant time.

When $C(k_{\text{front}}, j) < C(j-1, j)$, the algorithm must insert a pair corresponding to $j-1$ at the rear of the queue. Again, such insertion may cause the departure of some pairs in Q. The following fact is useful in this respect.

Fact 10. *Let (k_0, h_0) be the pair at the end of Q at step j and assume that $C(k_0, h_0) \geq C(j-1, h_0)$. Then k_0 can be declared dead at step j.*

When $C(k_{\text{front}}, j) < C(j-1, j)$, the update of Q is as follows. The algorithm keeps on deleting pairs (k_s, h_s) from the rear of Q when $C(k_s, h_s) \geq C(j-1, h_s)$. The

deletion process stops when the algorithm finds a pair (k_q, h_q) such that $C(k_q, h_q) < C(j-1, h_q)$. (At least the dummy pair meets this condition.) Then, the algorithm computes $h(j-1, k_q)$. Notice that all the first components of the deleted pairs are k's that are dead by Fact 10. Moreover, if $j+1 = h_{\text{front}-1}$, the algorithm deletes the front of the queue.

In what follows, let $K(r)$ and $H(r)$ denote the first and second component, respectively, of the rth pair from the back of the queue and let *rear* denote the last element in Q. The algorithm outlined above can be formalized as follows. It uses the following operations: *delete* and *dequeue* to remove the last and first element of Q, respectively; and *enqueue* to insert a new element at the end of the queue.

Algorithm B
enqueue $(0, 1)$ in Q
for $j = 1$ **to** n **do**
begin
  $l \leftarrow K(\text{front})$;
  if $C(j-1, j) \leqslant C(l, j)$ **then**
  begin
    $E[j] \leftarrow C(j-1, j)$;
    $Q \leftarrow \emptyset$;
    enqueue $(j-1, j+1)$
  end
  else
  begin
    $E[j] \leftarrow C(l, j)$
    while $C(j-1, H(\text{rear})) \leqslant C(K(\text{rear}), H(\text{rear}))$ **do delete**
    $h \leftarrow h(j-1, K(\text{rear}))$; **enqueue** $(j-1, h)$
    if $j+1 = H(\text{front})$ **then dequeue**
    else $H(\text{front}) \leftarrow H(\text{front}) + 1$;
  end
end

Theorem 2. *Algorithm* B *is correct and runs in time* $O(n \log n)$. *If w satisfies the closest zero property, then one can implement the algorithm in linear time.*

4. Conclusion

We presented two algorithms for the computation of

$$E[j] = \min_{0 \leqslant k \leqslant j-1} \{D[k] + w(k, j)\}, \quad j = 1, \ldots, n, \tag{11}$$

knowing that the weight function satisfies either the quadrangle inequality or the inverse quadrangle inequality. The two algorithms have a time complexity of

$O(n \log n)$. This time bound reduces to $O(n)$ if the weight function satisfies the closest zero property. These algorithms can be used to speed up several dynamic programming routines. The speed-up is from $O(n^3)$ to $O(n^2 \log n)$ or $O(n^2)$. In particular, we obtain an efficient and practical algorithm for the computation of the "modified edit distance" between two strings.

Note added in proof

Webb Miller and Eugene W. Myers [8] independently discovered Algorithm A. The two algorithms are similar, except for boundary conditions. Maria Klawe [4] improved Algorithm A to run in time $O(n \log^* n)$. She later improved it even further obtaining an $O(n\alpha(n))$ time bound (personal communication). Her algorithm is very nice, although mainly of theoretical interest. Indeed, the constant in the O-notation is quite large since the algorithm is recursive and each recursive step consists of several calls to another recursive procedure.

References

[1] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor and R.E. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica* **2** (1987) 209–233.
[2] W.M. Fitch and T.F. Smith, Optimal sequence alignment, *Nat. Acad. Sci. U.S.A.* (1983) 1382–1385.
[3] D.S. Hirshberg and L.L. Larmore, The least weight subsequence problem, *SIAM J. Comput.* **16** (1987) 628–638.
[4] M.M. Klawe, Speeding up dynamic programming, Manuscript.
[5] D.E. Knuth, Optimum binary search trees, *Acta Inform.* **1** (1973) 14–25.
[6] D.E. Knuth and M.F. Plass, Breaking paragraphs into lines, *Software: Practice and Experience* **11** (1981) 1119–1184.
[7] J.B. Kruskal and D. Sankoff, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison* (New York, Addison-Wesley, 1983).
[8] W. Miller and E.W. Myers, Sequence comparison with concave weighting functions, *Bull. Math. Biology* **50** (1988) 97–120.
[9] T.F. Smith and M.S. Waterman, New stratigraphic correlation techniques, *J. Geology* **88** (1980) 451–457.
[10] M.S. Waterman, General methods of sequence comparison, *Bull. Math. Biology* **46** (1984) 473–501.
[11] R.E. Wilber, The concave least weight subsequence problem revisited, *J. Algorithms*, to appear.
[12] F.F. Yao, Speed-up in dynamic programming, *SIAM J. Alg. Discr. Meth.* **3** (1982) 532–540.