

---

## 4

# Serial Computations of Levenshtein Distances

---

In the previous chapters, we discussed problems involving an exact match of string patterns. We now turn to problems involving similar but not necessarily exact pattern matches.

There are a number of similarity or distance measures, and many of them are special cases or generalizations of the Levenshtein metric. The problem of evaluating the measure of string similarity has numerous applications, including one arising in the study of the evolution of long molecules such as proteins. In this chapter, we focus on the problem of evaluating a longest common subsequence, which is expressively equivalent to the simple form of the Levenshtein distance.

### 4.1 Levenshtein distance and the LCS problem

The Levenshtein distance is a metric that measures the similarity of two strings. In its simple form, the Levenshtein distance,  $D(x, y)$ , between strings  $x$  and  $y$  is the minimum number of character insertions and/or deletions (indels) required to transform string  $x$  into string  $y$ . A commonly used generalization of the Levenshtein distance is the minimum cost of transforming  $x$  into  $y$  when the allowable operations are character insertion, deletion, and substitution, with costs  $\delta(\lambda, \sigma)$ ,  $\delta(\sigma, \lambda)$ , and  $\delta(\sigma_1, \sigma_2)$ , that are functions of the involved character(s).

There are direct correspondences between the Levenshtein distance of two strings, the length of the shortest *edit sequence* from one string to the other, and the length of the *longest common subsequence* (LCS) of those strings. If  $D$  is the simple Levenshtein distance between two strings having lengths  $m$  and  $n$ ,  $SES$  is the length of the shortest edit sequence between the strings, and  $L$  is the length of an LCS of the strings, then  $SES = D$  and  $L = (m + n - D)/2$ . We will focus on the problem of determining the length of an LCS and also on the related problem of recovering an LCS.

Another related problem, which will be discussed in Chapter 7, is that of approximate string matching, in which it is desired to locate all positions within string  $y$  which begin an approximation to string  $x$  containing at most  $D$  errors (insertions or deletions).

---

```

procedure CLASSIC(  $x, m, y, n, C, p$  ):
begin
   $L[0:m, 0] \leftarrow 0$ ;
   $P[0:m, 0] \leftarrow 1$ ;
   $L[0, 0:n] \leftarrow 0$ ;
   $P[0, 0:n] \leftarrow 2$ ;
  for  $i \leftarrow 1$  until  $m$  do
    for  $j \leftarrow 1$  until  $n$  do
      if  $x_i = y_j$  then  $L[i, j] \leftarrow 1 + L[i - 1, j - 1]$ 
      else if  $L[i - 1, j] > L[i, j - 1]$  then  $L[i, j] \leftarrow L[i - 1, j]$ 
      else  $L[i, j] \leftarrow L[i, j - 1]$ ;
      if  $x_i = y_j$  then  $P[i, j] \leftarrow 3$ 
      else if  $L[i - 1, j] > L[i, j - 1]$  then  $P[i, j] \leftarrow 1$ 
      else  $P[i, j] \leftarrow 2$ ;
   $p \leftarrow L[m, n]$ ;
   $(i, j) \leftarrow (m, n)$ ;
   $k \leftarrow p$ ;
  while  $k > 0$  do
    if  $P[i, j] = 3$  then
      begin
         $C[k] \leftarrow x_i$ ;
         $k \leftarrow k - 1$ ;
         $(i, j) \leftarrow (i - 1, j - 1)$ ;
      end
    else
      if  $P[i, j] = 1$  then  $i \leftarrow i - 1$ 
      else  $j \leftarrow j - 1$ 
end

```

---

**Fig. 4.1.** Classic LCS algorithm

## 4.2 Classical algorithm

Let the two input strings be  $x = x_1x_2\dots x_m$  and  $y = y_1y_2\dots y_n$  and let  $L(i, j)$  denote the length of an LCS of  $x[1:i]$  and  $y[1:j]$ . A simple recurrence relation exists on  $L$ :

$$L(i, j) = \begin{cases} 0, & \text{if either } i = 0 \text{ or } j = 0 \\ 1 + L(i - 1, j - 1), & \text{if } x_i = y_j \\ \max\{L(i - 1, j), L(i, j - 1)\}, & \text{if } x_i \neq y_j \end{cases} \quad (4.1)$$

This forms the basis for a dynamic programming algorithm that determines the length of an LCS. We fill matrix  $L[0:m, 0:n]$  with the values of

the  $L$ -function. We first set the values stored in the boundary cells ( $L[0, \bullet]$  and  $L[\bullet, 0]$ ) to 0. By sweeping the matrix in an order calculated to visit a cell only when its precedents (as defined by the recurrence) have already been valuated, we iterate setting the value stored in  $L[i, j]$  to the value of  $L(i, j)$  using the recurrence relation. (See Figure 4.1.) Storing pointers  $P[i, j]$  that indicate which  $L$ -entry contributed to the value of  $L(i, j)$  (change of coordinate 1, 2, or both), enables the recovery of an LCS by tracing these threads from  $(m, n)$  back to  $(0, 0)$ . A solution LCS, having length  $p = L(m, n)$ , will be placed in array  $C$ . This method requires  $O(mn)$  time and space.

The recurrence relation on  $L$  can be revised to enable computation of the generalized Levenshtein distance,  $D(i, j)$  between  $x[1:i]$  and  $y[1:j]$ , where insertions, deletions and substitutions have costs that are a function,  $\delta$ , of the symbols involved. A similar dynamic programming algorithm will evaluate  $D$ .

$$D(i, j) = \begin{cases} 0, & \text{if } i = 0 \text{ and } j = 0 \\ D(0, j - 1) + \delta(\lambda, y_j), & \text{if } i = 0 \text{ and } j > 0 \\ D(i - 1, 0) + \delta(x_i, \lambda), & \text{if } i > 0 \text{ and } j = 0 \\ D(i - 1, j - 1), & \text{if } x_i = y_j \\ \min \left\{ \begin{array}{l} D(i, j - 1) + \delta(\lambda, y_j), \\ D(i - 1, j) + \delta(x_i, \lambda), \\ D(i - 1, j - 1) + \delta(x_i, y_j) \end{array} \right\}, & \text{if } x_i \neq y_j \end{cases}, \quad (4.2)$$

### 4.3 Non-parameterized algorithms

A number of algorithms developed for solving the LCS problem have execution times dependent upon either the nature of the input (beyond merely the sizes of the two input strings) or the nature of the output. Such algorithms are referred to as input- or output-sensitive. Before discussing such algorithms, we first describe two LCS algorithms of general applicability whose performance is not parameterized by other variables.

#### 4.3.1 LINEAR SPACE ALGORITHM

The space complexity of determining the length of an LCS can be reduced to  $O(n)$  by noting that each row of  $L$  depends only on the one immediately preceding row of  $L$ . The length of an LCS of strings  $x[1:m]$  and  $y[1:n]$  will be returned in  $L[n]$  after invoking  $\text{FINDROW}(x, m, y, n, L)$ . (See Figure 4.2.)

Recovering an LCS using only linear space is not as simple. The “curve” that recovers an LCS was obtained by following threads through the  $L$  matrix after it was computed in its entirety. Instead, we first determine the middle point of an LCS curve and then, applying the procedure recursively, we determine the quartile points, etc.

---

```

procedure FINDROW(  $x, m, y, n, L$  ):
begin
   $L[0 : n] \leftarrow 0$ ;
  for  $i \leftarrow 1$  until  $m$  do
    begin
      for  $j \leftarrow 1$  until  $n$  do
        if  $x_i = y_j$  then  $L_{new}[j] \leftarrow 1 + L[j - 1]$ 
        else  $L_{new}[j] \leftarrow \max\{L_{new}[j - 1], L[j]\}$ ;
      for  $j \leftarrow 1$  until  $n$  do  $L[j] \leftarrow L_{new}[j]$ 
    end
  end

```

---

**Fig. 4.2.** FINDROW algorithm

There are several ways to determine the middle point of an LCS curve. We outline two methods, one that is conceptually simple and the other that is easier and more efficient to implement.

The simple method computes the middle row of  $L$  and then continues computing additional rows  $L[i, \bullet]$ , retaining for each element  $(i, j)$  a pointer to that element of the middle row through which the LCS curve from  $(0, 0)$  to  $(i, j)$  passes. The pointer retained by  $(m, n)$  indicates the middle point of an LCS. (See Figure 4.3.)

A more efficient method is to use the linear space FINDROW algorithm to compute the middle row of  $L$  and also the middle row of the solution matrix  $L^R$  for the problem of the reverses of strings  $x$  and  $y$ . It can be shown that their sum is maximized at points where LCS curves intersect with the middle row. It is in this manner that the middle point of an LCS curve is determined. Applying this procedure recursively, the quartile intersections can be recovered, etc. (See Figure 4.4.) Each iteration uses linear space, and it can be shown that the total time used is still quadratic, though about double what it was before.

This paradigm to recover an LCS, requiring only linear space, by using divide-and-conquer with algorithms that only evaluate the length of an LCS, can also be applied to many other algorithms for the LCS problem.

#### 4.3.2 SUBQUADRATIC ALGORITHM

A subquadratic time algorithm for this problem, that applies to the case of a finite alphabet of size  $s$ , uses a “Four Russians” approach. Essentially, instead of calculating the matrix  $L$ , the matrix is broken up into boxes of some appropriate size,  $k$ . The “high” sides of a box (the  $2k - 1$  elements of  $L$  on the edges of the box with largest indices) are computed from  $L$ -values known for boxes adjacent to it on the “low” side and from the relevant

---

```

function FINDMID(  $x, m, y, n$  ):
begin
   $L[0 : n] \leftarrow 0$ ;
   $mid \leftarrow \lceil m/2 \rceil$ ;
  for  $i \leftarrow 1$  until  $mid$  do
    begin
      for  $j \leftarrow 1$  until  $n$  do
        if  $x_i = y_j$  then  $L_{new}[j] \leftarrow 1 + L[j - 1]$ 
        else  $L_{new}[j] \leftarrow \max\{L_{new}[j - 1], L[j]\}$ ;
      for  $j \leftarrow 1$  until  $n$  do  $L[j] \leftarrow L_{new}[j]$ 
    end;
  for  $j \leftarrow 0$  until  $n$  do  $P[j] \leftarrow j$ ;
  for  $i \leftarrow mid + 1$  until  $m$  do
    begin
      for  $j \leftarrow 1$  until  $n$  do
        begin
          if  $x_i = y_j$  then  $L_{new}[j] \leftarrow 1 + L[j - 1]$ 
          else  $L_{new}[j] \leftarrow \max\{L_{new}[j - 1], L[j]\}$ ;
          if  $x_i = y_j$  then  $P_{new}[j] \leftarrow P[j - 1]$ 
          else if  $L_{new}[j - 1] > L[j]$  then
             $P_{new}[j] \leftarrow P_{new}[j - 1]$ 
          else  $P_{new}[j] \leftarrow P[j]$ 
        end;
      for  $j \leftarrow 1$  until  $n$  do  $L[j] \leftarrow L_{new}[j]$ ;
      for  $j \leftarrow 1$  until  $n$  do  $P[j] \leftarrow P_{new}[j]$ 
    end;
  return  $P[n]$ 
end

```

---

Fig. 4.3. FINDMID algorithm

symbols of  $x$  and  $y$  by using a lookup table that was precomputed.

There are  $2k + 1$  elements of  $L$  adjacent to a box on the “low” side. Two adjacent  $L$ -elements can differ by either zero or one. There are thus  $2^{2k}$  possibilities in this respect. The symbols of  $x$  and  $y$  range over an alphabet of size  $s$  for each of the  $2k$  elements, yielding a multiplicative factor of  $s^{2k}$  and the total number of boxes to be precomputed is therefore  $2^{2k(1+\log s)}$ . Each such box can be precomputed in time  $O(k^2)$  for a total precomputing time of  $O(k^2 2^{2k(1+\log s)})$ .

The sides of a box can be stored as “steps” consisting of 0’s and 1’s indicating whether adjacent elements of the side differ by 0 or 1. A box can therefore be looked up in time  $O(2k)$ . There are  $(n/k)^2$  boxes to be

---

```

procedure LINEARSPACE(  $x, m, y, n, C, p$  ):
begin
  if  $n = 0$  then  $p \leftarrow 0$ 
  else
    if  $m = 1$  then
      if  $\exists j \leq n$  with  $y_j = x_1$  then
        begin
           $p \leftarrow 1$ ;
           $C[1] \leftarrow x_1$ 
        end
      else  $p \leftarrow 0$ 
    else
      begin
         $i \leftarrow \lceil m/2 \rceil$ ;
        FINDROW(  $x, i, y, n, L$  );
        let  $x^R$  be the reverse of string  $x$ ;
        let  $y^R$  be the reverse of string  $y$ ;
        FINDROW(  $x^R, m - i, y^R, n, L^R$  );
        determine a  $k$  in the range  $0 \dots n$ 
          that maximizes  $L[k] + L^R[n - k]$ ;
        LINEARSPACE(  $x, i, y, k, C, q$  );
        let  $x'[1 : m - i]$  consist of elements  $x[i + 1 : m]$ ;
        let  $y'[1 : n - k]$  consist of elements  $y[k + 1 : n]$ ;
        LINEARSPACE(  $x', m - i, y', n - k, C', r$  );
         $p \leftarrow q + r$ ;
        let  $C'[q + 1 : p]$  consist of elements  $C'[1 : r]$ 
      end
    end
  end

```

---

**Fig. 4.4.** Linear space LCS algorithm

looked up, for a total time of  $O(n^2/k)$ .

The total execution time will therefore be  $O(k^2 2^{2k(1+\log s)} + n^2/k)$ . If we let  $k = (\log n)/(2 + 2 \log s)$ , we see that the total execution time will be  $O(n^2/\log n)$ . This algorithm can be modified for the case when the alphabet is of unrestricted size, with the resulting time complexity of  $O(n^2(\log \log n)/\log n)$ .

We note that this method works only for the classical Levenshtein distance metric but not for generalized cost matrices.

## 4.3.3 LOWER BOUNDS

The  $O(n^2/\log n)$  algorithm is the asymptotically fastest known. It is an open question as to whether this algorithm is asymptotically best possible.

If we consider algorithms for solving the LCS problem that are restricted to making symbol comparisons of the form “ $\sigma_1 = \sigma_2?$ ” then any such algorithm must make  $\Omega(n^2)$  comparisons for alphabets of unrestricted size and  $\Omega(ns)$  comparisons for alphabets of size restricted to  $s$ . In particular, if  $T(n, s)$  is the minimum number of “equal-unequal” comparisons under the decision tree model needed to find an LCS of two strings of length  $n$  when the total number of distinct symbols that can appear in the strings is  $s$ , then

$$\begin{aligned} T(n, 2) &= 2n - 1 \\ T(n, s) &\geq ns/2 + s^2/4, & \text{for } s \leq n \\ T(n, s) &\geq 3ns/4, & \text{for } n \leq s \leq 4n/3 \\ T(n, s) &= n^2, & \text{for } s \geq 4n/3 \end{aligned}$$

If we consider algorithms that may make symbol comparisons of the form “ $\sigma_1 \leq \sigma_2?$ ” then any such algorithm that solves the LCS problem must make  $\Omega(n \log m)$  symbol comparisons for alphabets of unrestricted size. The proofs of these lower bounds generally have relied on exhibiting a path of requisite length in decision trees that support such algorithms by using adversary arguments. We present a sketch of the  $\Omega(n \log m)$  lower bound.

Let the adversary’s response to a comparison  $p : q$  be as follows. If  $p$  and  $q$  are both positions in string  $x$  (say,  $x_i$  and  $x_j$ ) then if  $i < j$  return “less than”; otherwise, return “greater than”.

If  $p$  and  $q$  are not both positions in string  $x$  then let  $R$  be the number of relative orderings of positions of strings  $x$  and  $y$  that are consistent with the results of comparisons made thus far and that are consistent with  $x_1 < x_2 < \dots < x_m$ . Let  $R_1$  be the subset of  $R$  consistent with  $p < q$  and let  $R_2$  be the subset of  $R$  consistent with  $p > q$ . If  $|R_1| > |R_2|$  then return “less than”; otherwise, return “greater than”.

Define positions  $p$  and  $q$  to be *comparable* with respect to a sequence of comparisons if it can be logically deduced from the results of the comparisons that  $p < q$  or that  $p > q$ .

**Lemma 4.1.** *The algorithm must perform sufficient comparisons so that all positions in  $x$  are comparable to all positions in  $y$ .*

Each  $y_j$  in string  $y$  can be in any one of  $m + 1$  distinct states:

$$\begin{aligned} y_j &\leq x_1, \\ x_i &< y_j \leq x_{i+1}, [i = 1, \dots, m - 1], \\ x_m &< y_j \end{aligned}$$

Thus, there are  $(m + 1)^n$  possible relative orderings of the elements of  $y$  with respect to the elements of  $x$  and it will take  $\log(m + 1)^n \geq n \log m$  comparisons to distinguish which states the elements of  $y$  are in.

There are many algorithmic techniques that are not modeled by decision trees and for which these lower bounds would not apply. Examples of such techniques include array indexing (as used by the subquadratic algorithm of Section 4.3.2) and hashing.

#### 4.4 Parameterized algorithms

In this section, we discuss several algorithms for the LCS problem whose performance is parameterized by variables other than the sizes of the two input strings. Before describing these algorithms, we define some notation.

Consider the  $(m + 1) \times (n + 1)$  lattice of points corresponding to the set of prefix pairs of  $x$  and  $y$ , allowing for empty prefixes. We refer to the first coordinate of a point as its  $i$ -value and to the second coordinate as its  $j$ -value. We say that point  $(i, j)$  *dominates* point  $(i', j')$  if  $i' \leq i$  and  $j' \leq j$ . A *match* point is a point  $(i, j)$  such that  $x_i = y_j$ . The point  $(0, 0)$  is specially designated as also being a match point. Point  $(i, j)$  has rank  $k$  if  $L(i, j) = k$ . Point  $(i, j)$  is  $k$ -dominant if it has rank  $k$  and it is not dominated by any other point of rank  $k$ . Analogously, a point  $(i, j)$  is  $k$ -minimal if it has rank  $k$  and it does not dominate any other point of rank  $k$ . Note that if a point is  $k$ -dominant or  $k$ -minimal then it must be a match point.

The dominance relation defines a partial order on the set of match points. The LCS problem can be expressed as the problem of finding a longest chain in the poset of match points, modified to exclude links between match points that share the same  $i$ -value or  $j$ -value. Most known approaches to the LCS problem compute a minimal antichain decomposition for this poset, where a set of match points having equal rank is an antichain. These approaches typically either compute the antichains one at a time, or extend partial antichains relative to all ranks already discovered.

Let  $r$  be the number of match points, excluding  $(0, 0)$ , and let  $d$  be the total number of dominant points (all ranks). Then  $0 \leq p \leq d \leq r \leq mn$ .

##### 4.4.1 $PN$ ALGORITHM

We describe an algorithm that solves the LCS problem by computing the poset antichains one at a time, iteratively determining the set of  $k$ -minimal points for successively larger values of  $k$ . This algorithm requires time  $O(pn + n \log n)$ . If the expected length of an LCS is small, this algorithm will be faster than the classic algorithm.

The  $k$ -minimal points, if ordered by increasing  $i$ -value, will have their  $j$ -values in decreasing order. The algorithm detects all minimal match points of one rank by processing the match points across rows.

---

```

procedure PN(  $x, m, y, n, C, p$  ):
begin
   $\forall \sigma \in x$ , build ordered MATCHLIST( $\sigma$ ) of  $y$ -positions containing  $\sigma$ ;
   $M[0, 0:m] \leftarrow 0$ ;
   $first \leftarrow 0$ ;
  for  $k \leftarrow 1$  step 1 do
    begin
       $prev \leftarrow first$ ;
       $low \leftarrow M[k-1, prev]$ ;
       $high \leftarrow n + 1$ ;
      for  $i \leftarrow prev + 1$  until  $m$  do
        begin
           $t \leftarrow \min\{j \in \text{MATCHLIST}(x_i) \mid j > low\}$ ;
          if  $t < high$  then
             $M[k, i] \leftarrow high \leftarrow t$ 
          else  $M[k, i] \leftarrow 0$ ;
          if  $M[k, i] > 0$  and  $first = prev$  then  $first \leftarrow i$ ;
          if  $M[k-1, i] > 0$  then  $low \leftarrow M[k-1, i]$ 
        end;
        comment  $M[k, 0:m]$  contains the set of  $k$ -minimal points;
        if  $first = prev$  then goto recover
      end;
    end;
  recover:
     $p \leftarrow k - 1$ ;
     $k \leftarrow p$ ;
    for  $i \leftarrow m$  step  $-1$  until  $0$  do
      if  $M[k, i] > 0$  then
        begin
           $C[k] \leftarrow x_i$ ;
           $k \leftarrow k - 1$ 
        end
    end
end

```

---

**Fig. 4.5.** Sketch of  $pn$  LCS algorithm

Define  $low_k(i)$  to be the minimum  $j$ -value of match points having rank  $k-1$  whose  $i$ -value is less than  $i$ . Define  $high_k(i)$  to be the minimum  $j$ -value of match points having rank  $k$  whose  $i$ -value is less than  $i$  ( $n+1$  if there are no such points). The following lemma is essential to the algorithm.

**Lemma 4.2.**  $(i, j)$  is a  $k$ -minimal point iff  $j$  is the minimum value such that  $x_i = y_j$  and  $low_k(i) < j < high_k(i)$ .

A sketch of the  $O(pn)$  algorithm is given in Figure 4.5. The algorithm

obtains its efficiency by the use of three simple data structures. First, a collection of balanced binary search trees provides a mapping of alphabet symbols to the integers  $\{1, \dots, |\Sigma|\}$ . Second, for each  $\sigma \in x$ ,  $\text{MATCHLIST}(\sigma)$  contains the ordered list of positions in  $y$  in which symbol  $\sigma$  occurs. Third, array  $M$  maintains the set of  $k$ -minimal points ordered by the  $i$ -values of the points. For each value of  $k$ , an iteration of the outer loop determines the set of  $k$ -minimal points in linear time. A crucial observation is that the evaluation of variable  $t$ , the minimum element in  $\text{MATCHLIST}(x_i)$  satisfying the *low* and *high* bounds, can be accomplished by iteratively decrementing a pointer to that  $\text{MATCHLIST}$ . The total number of decrements to that  $\text{MATCHLIST}$  cannot exceed the size of that  $\text{MATCHLIST}$ , and the sum of the lengths of all  $\text{MATCHLIST}$ s is  $n$ .

#### 4.4.2 HUNT-SZYMANSKI ALGORITHM

We now describe an algorithm, due to J. Hunt and T. Szymanski, for solving the LCS problem in  $O((r+n)\log n)$  time and  $O(r+n)$  space. (See Figure 4.6.) This algorithm is particularly efficient for applications where most positions of one sequence match relatively few positions in the other sequence. Examples of such applications include finding the longest ascending subsequence of a permutation of the integers  $\{1 \dots n\}$  and file differencing in which a line of prose is considered atomic.

The algorithm detects dominant match points across all ranks by processing the match points row by row. For each  $i$ , the ordered list  $\text{MATCHLIST}(i)$  is set to contain the descending sequence of positions  $j$  for which  $x_i = y_j$ . This initializing process can be performed in time  $O(n \log n)$  by stably sorting a copy of sequence  $y$  while keeping track of each element's original position, and counting the number of elements of each symbol value.  $\text{MATCHLIST}(i)$  can be implemented with a count of the size and a pointer to the last of the now contiguous subset of elements having symbol  $x_i$ . Then, iteratively for each row  $i$ , the algorithm evaluates the *threshold* function  $T(i, k)$  defined to be the smallest  $j$  such that  $L(i, j) \geq k$ . This function satisfies the recurrence relation

$$T(i, k) = \begin{cases} \text{smallest } j \text{ such that } x_i = y_j \\ \quad \text{and } T(i-1, k-1) < j \leq T(i-1, k) & (4.3) \\ T(i-1, k), \text{ if no such } j \text{ exists} \end{cases}$$

By maintaining the  $T$  values in a one-dimensional array  $\text{THRESH}$  and considering the  $j$  in  $\text{MATCHLIST}(i)$  in descending order, the  $k$  for which  $T(i, k)$  differs from  $T(i-1, k)$  can be determined in  $O(\log n)$  time by using binary search on the  $\text{THRESH}$  array.

Variations of the Hunt-Szymanski algorithm have improved complexity. The basic algorithm can be implemented with flat trees to achieve time

---

```

procedure HUNT(  $x, m, y, n, C, p$  ):
begin
  for  $i \leftarrow 1$  until  $m$  do
    begin
      comment initialize THRESH values;
      THRESH[ $i$ ]  $\leftarrow n + 1$ ;
      set MATCHLIST[ $i$ ] to be the descending sequence
        of positions  $j$  s.t.  $x_i = y_j$ ;
    end
    THRESH[0]  $\leftarrow 0$ ;
    LINK[0]  $\leftarrow \lambda$ ;
    comment compute successive THRESH values
      THRESH[ $k$ ] =  $T(i - 1, k)$  (initially) and  $T(i, k)$  (finally);
    for  $i \leftarrow 1$  until  $m$  do
      for each  $j$  in MATCHLIST[ $i$ ] do
        begin
          use binary search on the THRESH array to find  $k$ 
            such that THRESH[ $k - 1$ ] <  $j \leq$  THRESH[ $k$ ];
          if  $j <$  THRESH[ $k$ ] then
            begin
              THRESH[ $k$ ]  $\leftarrow j$ ;
              create a list node new whose fields contain:
                 $i, \text{LINK}[k - 1]$ ;
              LINK[ $k$ ]  $\leftarrow \textit{new}$ 
            end
          end
        end
       $t \leftarrow$  largest  $k$  such that THRESH[ $k$ ]  $\neq n + 1$ ;
       $\textit{last} \leftarrow$  LINK[ $t$ ];
       $p \leftarrow 0$ ;
      while  $\textit{last} \neq \lambda$  do
        begin
          comment recover LCS in reverse order;
           $(i, \textit{prev}) \leftarrow$  fields of list node  $\textit{last}$ ;
           $p \leftarrow p + 1$ ;
           $S[p] \leftarrow x_i$ ;
           $\textit{last} \leftarrow \textit{prev}$ 
        end;
       $C[1 : p] \leftarrow$  the reverse of the sequence of elements  $S[1 : p]$ 
    end
  end

```

---

Fig. 4.6. Hunt-Szymanski LCS algorithm

complexity  $O(r \log \log n + n \log n)$  over an unbounded alphabet and  $O((r + n) \log \log n)$  over a fixed-size alphabet. However, since the use of flat trees imposes a large multiplicative constant, this improvement is of theoretical interest only.

By concentrating attention on the  $d$  dominant points (a subset of the  $r$  match points), the basic algorithm can be modified to have  $O(m \log n + d \log(mn/d))$  time complexity and  $O(d + n)$  space complexity. The time complexity can be theoretically further improved to  $O(n + d \log \log(mn/d))$  by application of Johnson's improvement to flat trees.

#### 4.4.3 $ND$ ALGORITHM

Let  $D$  be the difference in length between  $x$  and  $\text{LCS}(x, y)$ ;  $D = m - p$ . We now describe an  $O(nD)$ -time LCS algorithm. (See Figure 4.7.) This algorithm is based on evaluating the function  $M(k, i)$  defined to be the largest  $j$  such that  $x[i : m]$  and  $y[j : n]$  have an LCS of size  $\geq k$ . This function is symmetric to the threshold function (Equation 4.3 of Section 4.4.2) and satisfies the following recurrence relation.

$$M(k, i) = \begin{cases} \text{largest } j > M(k, i + 1) \text{ such that } x_i = y_j, \text{ and} \\ \quad \quad \quad \text{if } k > 1, j < M(k - 1, i + 1) \\ M(k, i + 1), \text{ if no such } j \text{ exists} \end{cases} \quad (4.4)$$

The efficiency of this algorithm derives from the procedure of avoiding calculating elements of  $M$  which cannot induce an LCS. The elements of  $M$  are evaluated along diagonals, one diagonal at a time. The first diagonal ( $diag = m$ ) is  $M[1, m]$  through  $M[m, 1]$ ; successive diagonals (smaller values of  $diag$ ) are  $M[1, diag]$  through  $M[diag, 1]$ . No further elements of  $M$  are evaluated beyond diagonal  $p$ . We know that we have encountered the last required diagonal when the length,  $p$ , of the longest found CS equals the diagonal number. Each diagonal requires only linear time since the  $y$  index,  $j$ , has range at most 1 to  $n$ . Therefore the total time required is  $O(n(m - p))$ .

The algorithm, as given, uses  $m^2$  space for the  $M$  array. However, by using the simple mapping of  $M[k, i]$  to an element of a one-dimensional array, it is straightforward to use space  $O(mD)$ . The space-saving technique, discussed earlier, can be applied to this algorithm, resulting in an algorithm with  $O(nD)$  time and linear space complexity.

#### 4.4.4 MYERS ALGORITHM

Myers developed an  $O(nD)$  time algorithm that can be executed using linear space. Under a basic stochastic model, his algorithm has expected time complexity  $O(n + D^2)$ . A non-practical variation, using suffix trees, has  $O(n \log n + D^2)$  worst-case time complexity.

---

```

procedure NAKATSU(  $x, m, y, n, C, p$  ):
begin
     $diag \leftarrow m$ ;
     $p \leftarrow 0$ ;
    while  $p < diag$  do
        begin
             $i \leftarrow diag$ ;
             $len \leftarrow 1$ ;
             $j_{max} \leftarrow n + 1$ ;
            comment evaluate  $M[k, i]$  along one diagonal;
            while  $i \neq 0$  and  $j_{max} \neq 0$  do
                begin
                    comment clear an element of  $M$ 
                    for uniform handling;
                    if  $diag = m$  or  $len > p$  then  $M[diag, i + 1] \leftarrow 0$ ;
                     $j_{min} \leftarrow \max\{1, M[diag, i + 1]\}$ ;
                     $j \leftarrow j_{max} - 1$ ;
                    comment calculate one  $M[k, i]$ ;
                    while  $j \geq j_{min}$  and  $x_i \neq y_j$  do  $j \leftarrow j - 1$ ;
                    if  $j \geq j_{min}$  then  $j_{max} \leftarrow j$ 
                    else  $j_{max} \leftarrow M[diag, i + 1]$ ;
                     $M[diag, i] \leftarrow j_{max}$ ;
                    if  $j_{max} = 0$  then  $len \leftarrow len - 1$ ;
                    if  $len > p$  then  $p \leftarrow len$ ;
                     $len \leftarrow len + 1$ ;
                     $i \leftarrow i - 1$ 
                end;
                 $diag \leftarrow diag - 1$ 
            end;
            comment recover an LCS, the length of which is  $p$ ;
            if  $j_{max} = 0$  then  $i \leftarrow i + 2$ 
            else  $i \leftarrow i + 1$ ;
             $k \leftarrow p$ ;
            while  $k > 0$  do
                begin
                    while  $M[k, i] = M[k, i + 1]$  do
                         $i \leftarrow i + 1$ ;
                     $C[p + 1 - k] \leftarrow x_i$ ;
                     $i \leftarrow i + 1$ ;
                     $k \leftarrow k - 1$ 
                end
            end
        end
    end

```

---

Fig. 4.7.  $nD$  LCS algorithm

---

```

function MYERS(  $x, m, y, n$  ):
begin
  DOMI[1]  $\leftarrow$  0;
  comment look for dominant  $D$ -deviation points;
  for  $D \leftarrow 0$  until  $m + n$  do
    for  $k \leftarrow -D$  step 2 until  $D$  do
      begin
        comment diagonals are 2 apart;
        if  $k = -D$  or  $(k \neq D$  and
          DOMI[ $k - 1$ ] < DOMI[ $k + 1$ ]) then
           $i \leftarrow$  DOMI[ $k + 1$ ]
        else  $i \leftarrow$  DOMI[ $k - 1$ ] + 1;
         $j \leftarrow i - k$ ;
        comment until non-match is found
          increment both coordinates;
        while  $i < m$  and  $j < n$  and  $x[i + 1] = y[j + 1]$  do
           $(i, j) \leftarrow (i + 1, j + 1)$ ;
        comment store  $i$ -value
          of diagonal  $k$  dominant  $D$ -deviation;
        DOMI[ $k$ ]  $\leftarrow i$ ;
        comment if we found minimum adequate deviation
          then return length of LCS;
        if  $i = m$  and  $j = n$  then
          return  $(i + j - D)/2$ ;
      end
    end
end

```

---

**Fig. 4.8.** Myers LCS algorithm

The LCS trace will not deviate from the main diagonal more than the difference between the two input sequences. The essence of Myers' algorithm is to avoid evaluating unnecessary parts of the  $L$  matrix. Associated with each point  $(i, j)$  having rank  $k$  is its diagonal number,  $i - j$ , and its deviation,  $i + j - 2k$ . There is only one 0-dominant point,  $(0, 0)$ , and it has deviation 0. The set of dominant 0-deviation points lie on the 0-diagonal from  $(0, 0)$  through  $(i - 1, i - 1)$ , where  $i$  is the minimum index such that  $x_i \neq y[i]$ . The algorithm iterates calculating the set of dominant points having successively higher deviation. Each dominant  $(D + 1)$ -deviation point can be found by starting at a dominant  $D$ -deviation point, traversing one unit orthogonally (adding one to exactly one coordinate), and iteratively incrementing both coordinates until just before the first non-match point is encountered. The algorithm terminates when point  $(m, n)$  is reached.

Implementation is made easier by the fact that there will be exactly one dominant  $D$ -deviation point for every other diagonal (that is, half the diagonals) in the range  $-D$  to  $+D$ .

The code shown in Figure 4.8 returns the length of an LCS. In this code,  $\text{DOMI}[k]$  stores the  $i$ -value of the dominant  $D$ -deviation point located on diagonal  $k$ . Therefore, that point is  $(\text{DOMI}[k], \text{DOMI}[k] - k)$ .

In order to recover an LCS, either the sequence of all encountered dominant points (all successive values of array  $\text{DOMI}$ ) are retained, necessitating the usage of  $O(nD)$  space or, by using a space-saving method similar to that used earlier, linear space will suffice at a cost of increasing the time requirements by a factor of about two.

A linear space version is enabled by determining the midpoint of an SES (shortest edit sequence) curve. This can be done by alternately calculating dominant  $D$ -deviation points for the two reverse problems  $(x, y)$  and  $(x^R, y^R)$  for iteratively larger values of  $D$  until a member of one of the two sets of dominant deviation points meets or passes a member of the other set along their common diagonal. A first point of meeting or passage will be an SES midpoint.

#### 4.5 Exercises

1. Implement the classical algorithm to recover the sequence of edit operations (insert, delete, substitute) that will result in minimum total cost. Assume that substituting one symbol for another incurs cost 1.5 while insertion or deletion of a symbol incurs unit cost.
2. A string insertion (deletion) consists of inserting (deleting) a string of any length at one location. Implement an algorithm that determines the minimum cost sequence of character and string insertions and deletions required to transform string  $x$  into string  $y$  under the constraint that no substring of an inserted string may subsequently be deleted. Assume that single character insertions and deletions have unit cost, and that each string insertion and deletion has cost  $1 + \text{stringlength}/2$ .
3. What can you say about the complexity of the above problem without the constraint disallowing subsequent partial deletion of inserted strings.
4. Show that the sum of the middle rows of  $L$  and  $L^R$  is maximized at points where LCS curves intersect with the middle row.
5. How many nested levels of iterations are required by the Linear Space Algorithm? Show that the total time used is quadratic.
6. Implement the  $pn$  LCS algorithm.
7. Prove the recurrence relation on threshold values.
8. Show the relation between the  $T$  function defined in Section 4.4.2, and the  $M$  function defined in Section 4.4.3. What, if any, is the relation

between the  $T$  function and the *low* and *high* functions defined in Section 4.4.1?

9. Implement a linear space version of Myers algorithm that recovers an LCS.
10. Implement a cubic-time and quadratic-space algorithm that recovers a longest subsequence common to three strings.

#### 4.6 Bibliographic notes

Levenshtein [1966] introduced measures of distance between strings based on indels. Applications of the LCS and related problems are discussed in more detail in Sankoff and Kruskal [1983]. The string-to-string edit problem is described and solved in Wagner and Fisher [1974]. Approximate string matching is discussed in Ukkonen [1985], Galil and Giancarlo [1988], Landau and Vishkin [1988], and Galil and Park [1990]. Chin and Poon [1994] analyze some heuristics for computing an LCS. Gotoh [1982] exhibits an  $O(mn)$  time algorithm to compute the edit distance between two strings under a generalized Levenshtein distance in which indels of substrings have cost linear in the indel length. The linear space algorithm for the LCS problem is due to Hirschberg [1975]. The conceptually simple linear space method of determining the middle of an LCS curve is due to Eppstein (unpublished).

The “Four Russians” are Arlazarov, Dinic, Kronrod, and Faradzev [1970]. Their approach is also discussed in Aho, Hopcroft, and Ullman [1974]. The subquadratic time algorithm for restricted size alphabet is from Masek and Paterson [1980]. A discussion for the case of unrestricted size alphabet can be found in Hunt and Szymanski [1977].

Lower bounds for the LCS problem are proven in Aho, Hirschberg, and Ullman [1976], Wong and Chandra [1976], and Hirschberg [1978]. The description of LCS algorithmic approaches in terms of poset antichains was first explicated in Apostolico, Browne, and Guerra [1992]. An LCS algorithm with time complexity  $O(pn + n \log n)$  is in Hirschberg [1977]. The  $O((r + n) \log n)$  time Hunt-Szymanski algorithm is described in Hunt and Szymanski [1977]. Apostolico [1986] improves its worst-case performance. The notion of flat trees is from van Emde Boas [1975]; they are improved in Johnson [1982]. Modifications to the Hunt-Szymanski algorithm are discussed in Hsu and Du [1984a] (but see Apostolico [1987]), Apostolico and Guerra [1987] and Eppstein, Galil, Giancarlo, and Italiano [1990]. Other algorithms are discussed in Chin and Poon [1990] and Rick [1995].

The  $O(nD)$ -time algorithm is due to Nakatsu, Kambayashi, and Yajima [1982]. The code in Figure 4.7 is from their paper, with changes in the variable names. The linear space version of Nakatsu’s algorithm is shown in Kumar and Rangan [1987].

Myers algorithm is from Myers [1986]. The code in Figure 4.8 is from

Myers [1986], with changes in the variable names. Wu, Manber, Myers, and Miller [1990] obtain a slightly faster  $O(nP)$  algorithm, where  $P$  is the number of deletions in the shortest edit script.

The LCS problem can be generalized to the problem of determining a longest sequence common to  $N$  strings. Maier [1978] shows that if the number of strings,  $N$ , is not a constant then the  $N$ -LCS problem is NP-complete. However, for fixed values of  $N$ , the  $N$ -LCS problem can be solved using extensions of the algorithms in this chapter for the 2-LCS problem. Itoga [1981] shows that the extension of the classical algorithm has time and space complexity proportional to the product of the number of strings and the strings' lengths which, in the case of  $N$  strings each of length  $n$ , is  $\theta(Nn^N)$ . Other algorithms for the  $N$ -LCS problem are shown in Hsu and Du [1984b] and Irving and Fraser [1992].

#### 4.7 Bibliography

- AHO, A. V., D. S. HIRSCHBERG, AND J. D. ULLMAN [1986]. "Bounds on the complexity of the longest common subsequence problem," *Jour. ACM* **23**, 1-12.
- AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.
- APOSTOLICO, A. [1986]. "Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings," *Info. Processing Letters* **23**, 63-69.
- APOSTOLICO, A. [1987]. "Remark on Hsu-Du New Algorithm for the LCS Problem," *Info. Processing Letters* **25**, 235-236.
- APOSTOLICO, A., S. BROWNE, AND C. GUERRA [1992]. "Fast linear-space computations of longest common subsequences," *Theoretical Computer Science* **92**, 3-17.
- APOSTOLICO, A., AND C. GUERRA [1987]. "The longest common subsequence problem revisited," *Algorithmica* **2**, 315-336.
- ARLAZAROV, V. L., E. A. DINIC, M. A. KRONROD, AND I. A. FARADZEV [1970]. "On economical construction of the transitive closure of a directed graph," *Dokl. Akad. Nauk SSSR* **194**, 487-488 (in Russian). English translation in *Soviet Math. Dokl.* **11**, 1209-1210.
- CHIN, F. Y. L., AND C. K. POON [1990]. "A fast algorithm for computing longest common subsequences of small alphabet size," *Jour. of Info. Processing* **13**, 463-469.
- CHIN, F., AND C. K. POON [1994]. "Performance analysis of some simple heuristics for computing longest common subsequences," *Algorithmica* **12**, 293-311.
- VAN EMDE BOAS, P. [1975]. "Preserving order in a forest in less than logarithmic time," *Proc. 16th FOCS*, 75-84.

- EPPSTEIN, D., Z. GALIL, R. GIANCARLO, AND G. ITALIANO [1990]. "Sparse dynamic programming," *Proc. Symp. on Discrete Algorithms*, San Francisco CA, 513-522.
- GALIL, Z., AND R. GIANCARLO [1988]. "Data structures and algorithms for approximate string matching," *Jour. Complexity* **4**, 33-72.
- GALIL, Z., AND K. PARK [1990]. "An improved algorithm for approximate string matching," *SIAM Jour. Computing* **19**, 989-999.
- GOTOH, O. [1982]. "An improved algorithm for matching biological sequences," *Jour. Mol. Biol.* **162**, 705-708.
- HIRSCHBERG, D. S. [1975]. "A linear space algorithm for computing maximal common subsequences," *Commun. ACM* **18**, 341-343.
- HIRSCHBERG, D. S. [1977]. "Algorithms for the longest common subsequence problem," *Jour. ACM* **24**, 664-675.
- HIRSCHBERG, D. S. [1978]. "An information theoretic lower bound for the longest common subsequence problem," *Info. Processing Letters* **7**, 40-41.
- HSU, W. J., AND M. W. DU [1984a]. "New algorithms for the LCS problem," *Jour. of Computer and System Sciences* **29**, 133-152.
- HSU, W. J., AND M. W. DU [1984b]. "Computing a longest common subsequence for a set of strings," *BIT* **24**, 45-59.
- HUNT, J. W., AND T. G. SZYMANSKI [1977]. "A fast algorithm for computing longest common subsequences," *Commun. ACM* **20**, 350-353.
- IRVING, R. W., AND C. B. FRASER [1992]. "Two algorithms for the longest common subsequence of three (or more) strings," Proc. of the 3rd Annual Symp. on Combinatorial Pattern Matching, *Lecture Notes in Computer Science* **644**, 214-229.
- ITOGA, S. Y. [1981]. "The string merging problem," *BIT* **21**, 20-30.
- JOHNSON, D. B. [1982]. "A priority queue in which initialization and queue operations take  $O(\log \log D)$  time," *Math. Systems Theory* **15**, 295-309.
- KUMAR, S. K., AND C. P. RANGAN [1987]. "A linear space algorithm for the LCS problem," *Acta Informatica* **24**, 353-362.
- LANDAU, G. M., AND U. VISHKIN [1988]. "Fast string matching with k differences," *Jour. Comp. and System Sci.* **37**, 63-78.
- LEVENSHTEIN, V. I. [1966]. "Binary codes capable of correcting deletions, insertions, and reversals," *Cybernetics and Control Theory* **10** (1966), 707-710.
- MAIER, D. [1978]. "The complexity of some problems on subsequences and supersequences," *Jour. ACM* **25**, 322-336.
- MASEK, W. J., AND M. S. PATERSON [1980]. "A faster algorithm for computing string edit distances," *Jour. Comput. System Sci.* **20**, 18-31.
- MYERS, E. W. [1986]. "An  $O(ND)$  difference algorithm and its variations," *Algorithmica* **1**, 251-266.

- NAKATSU, N., Y. KAMBAYASHI, AND S. YAJIMA [1982]. "A longest common subsequence algorithm suitable for similar text strings," *Acta Informatica* **18**, 171-179.
- RICK, C. [1995]. "A new flexible algorithm for the longest common subsequence problem," Proc. of the 6th Annual Symp. on Combinatorial Pattern Matching, *Lecture Notes in Computer Science* **937**, 340-351.
- SANKOFF, D., AND J. B. KRUSKAL [1983]. *Time Warps, String Edits, and Macromolecules*, Addison-Wesley, Reading, Mass.
- UKKONEN, E. [1985]. "Finding approximate patterns in strings," *Jour. Algorithms* **6**, 132-137.
- WAGNER, R. A., AND M. J. FISCHER [1974]. "The string-to-string correction problem," *Jour. ACM* **21**, 168-173.
- WONG, C. K., AND A. K. CHANDRA [1976]. "Bounds for the string editing problem," *Jour. ACM* **23**, 13-16.
- WU, S., U. MANBER, E. W. MYERS, AND W. MILLER [1990]. "An  $O(NP)$  sequence comparison algorithm," *Info. Processing Letters* **35**, 317-323.