A Survey of Longest Common Subsequence Algorithms

L. Bergroth
bergroth@cs.utu.fi

H. Hakonen
hat@cs.utu.fi

T. Raita
raita@cs.utu.fi

Department of Computer Science
University of Turku
20520 Turku
Finland

Abstract

The aim of this paper is to give a comprehensive comparison of well-known longest common subsequence algorithms (for two input strings) and study their behaviour in various application environments. The performance of the methods depends heavily on the properties of the problem instance as well as the supporting data structures used in the implementation. We want to make also a clear distinction between methods that determine the actual lcs and those calculating only its length, since the execution time and more importantly, the space demand depends crucially on the type of the task. To our knowledge, this is the first time this kind of survey has been done. Due to the page limits, we can give here only a coarse overview of the performance of the algorithms; more detailed studies are reported elsewhere [11].

1. Introduction

String comparison is a central operation in various environments: a spelling error correction program tries to find the dictionary entry which resembles most a given word, in molecular biology [2, 34] we want to compare two DNA or protein sequences to learn how homologous they are, in a file archive we want to store several versions of a source program compactly by storing only the original version as such; all other versions are constructed from the differences to the previous one, etc. An obvious measure for the closeness of two strings is to find the maximum number of identical symbols in them (preserving the symbol order). This, by definition, is the *longest common subsequence* of the strings. Formally, we are comparing two input strings, $X[1..m]$ and $Y[1..n]$, which are elements of the set Σ^*; here Σ denotes the *input alphabet* containing σ symbols. A *subsequence* $S[1..s]$ of $X[1..m]$ is obtained by deleting $m-s$ symbols from X. A *common subsequence (cs)* of $X[1..m]$ and $Y[1..n]$ – designated $cs(X,Y)$ – is a subsequence which occurs in both strings (w.l.o.g we will assume that $m \leq n$ in what follows). The *longest common subsequence (lcs)* of strings X and Y, $lcs(X,Y)$, is a common subsequence of maximal length. The length of $lcs(X,Y)$ is denoted by $r(X,Y)$, or when the input strings are known, by r. An introduction to the most important lcs methods can be found in books on text algorithms (see e.g. [6, 14, 37]) and on molecular biology (see e.g. [17, 32, 36]).

The lcs problem is a special case of the *edit distance problem*. The distance between X and Y, $Dist(X,Y)$, is defined as the minimal number of *elementary operations* needed to transform the source string X to the target string Y. In practical applications, the operations are restricted to insertions, deletions and substitutions subjected to single input symbols. For each operation, an application dependent *cost* is assigned, whence the distance is defined as the minimum sum of transformation costs. Normalizing the costs of insertion and deletion to 1 and excluding the substitution operation (which can be accomplished by defining its cost to be ≥ 2), we have in fact the lcs problem: in order to transform X to Y, we first delete $m-r$ symbols from the source to obtain $lcs(X,Y)$. To this string we insert $n-r$ symbols to form the target. Therefore,

$$Dist(X,Y) = n + m - 2 \cdot r(X,Y)$$

For a more detailed discussion of the edit distance, its properties and applications, see [17, 36].

Another closely related problem is the *shortest common supersequence problem (scs)*, in which we want to find the shortest string containing X and Y as subsequences. In this case,

$$r(X,Y) = m + n - r(scs(X,Y))$$

where $r(scs(X,Y))$ denotes the length of the shortest common supersequence. After constructing the supersequence,

0-7695-0746-8/00 $10.00 © 2000 IEEE

we can extract the lcs in a linear scan through the scs. However, if the input consists of more than two strings (*N-lcs/N-scs*), there is no obvious symmetry between the problems anymore [15].

The lcs problem can be reduced to two other well-known problems also. $Lcs(X,Y)$ is typically solved with the dynamic programming technique and filling an $m \times n$ table. The table elements can be regarded as vertices in a graph and the simple dependencies between the table values define the edges. The task is to find the longest path between the vertices in the upper left and lower right corner of the table. Another reduction can be done to the *longest increasing subsequence problem (lis)*. In the lis problem, we are given a sequence Z of z integers, and our task is to find longest subsequence of Z which is strictly increasing. The input to the lis problem is obtained from the lcs input as follows: take $X[1]$ and find all positions j in Y, for which $X[1] = Y[j]$. Sort these indices into decreasing order. Build a sorted list for $X[2]$ in a similar way and append it to the one we got for $X[1]$. Repeat this for all symbols in X. As a result, we have a sequence of integers having decreasing runs ('saw-tooth' form). Solving the lis of this sequence gives the Y-indices of those symbols which belong to the lcs.

The studies on theoretical complexity of the lcs problem [1, 16, 20, 22, 27, 39] give the lower bound $\Omega(n^2)$, if the elementary comparison operation is of type 'equal/unequal' and the alphabet size is unrestricted. However, if the input alphabet is fixed, we reach the lower bound $\Omega(\sigma n)$. In practice, the underlying encoding scheme for the symbols of the input alphabet implies a topological ordering between them and the $\leq$-comparison gives us more information reducing the lower bound to $\Omega(n \log m)$. Of the current methods, the $O(n^2/\log n)$ algorithm of Masek and Paterson [28] is theoretically the fastest known.

Although the time and space complexity of the dynamic programming approach is 'only' quadratic, it tends to be too large for many applications. Due to this, several heuristics [2, 9, 10, 12, 15, 23] as well as exact algorithms handling special inputs [5, 13, 19, 31, 33, 40] have been devised. Often, it is not the time complexity that is crucial but space becomes the limiting resource [17]. The basic dynamic programming method is easily implemented using (m) space [19], if only the length of the lcs is needed. The same bound can be reached also when the actual sequence has to be found [5, 7, 18, 25] at the cost of roughly doubling the time of the basic algorithm from which the variant was derived (but preserving the same asymptotic bounds). In this survey, the speed of the algorithms is the most important quality and we have excluded the linear-space variants. Still, it must be emphasized that the performance figures are very different for the algorithms included, whether they search the lcs or only its length.

The paper is organized as follows. First, we define some basic concepts in Section 2 and describe then the various approaches to solve the lcs problem in Section 3. After that, we discuss the influence of the data structures selected to the heart of the algorithm and the influence of lower level decisions, such as memory management. The actual comparison is given in Section 5 and the paper is concluded with a discussion.

2. Basic Concepts

The traditional technique for solving $lcs(X[1..m], Y[1..n])$ is to determine the longest common subsequence for all possible prefix combinations of the input strings. The recurrence relation for extending the length of the lcs for each prefix pair $(X[1..i], Y[1..j])$ is the following [38]:

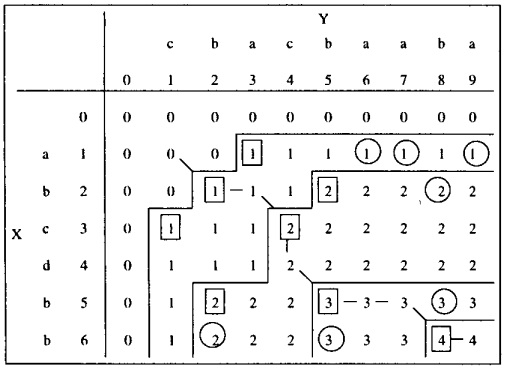
$$R[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ R[i-1,j-1]+1 & \text{if } X[i] = Y[j] \\ \max\{R[i-1,j], R[i,j-1]\} & \text{if } X[i] \neq Y[j] \end{cases}$$

where we have used the tabular notation $R[i,j] = r(X[1..i], Y[1..j])$. Intrinsic to this recurrence is that a single R-value depends only on three neighbouring values in the table, i.e. the calculation is very localized. After the table has been filled, the length is found in $R[m,n] = r(X[1..m], Y[1..n])$. The common subsequence is found by backtracking from $R[m,n]$ and at each step, either (a) follow pointers which were set during the calculation of the values or (b) recalculate the predecessor which yielded the value to the current table entry. Each time a match is found (the middle rule applies), we have found a symbol to the lcs. In this way, we can traverse a path through the table until a length of zero is found. In general, there may be several such paths because the lcs is not necessarily unique. To find all of them, we traverse systematically through the graph induced by the pointers (using e.g. breadth-first search). As an example, the table corresponding to $r(abcdbb, cbacbaaba)$ is shown on the next page. In this case, $r(X,Y) = 4$ and the longest common subsequence corresponding to the depicted path is *bcbb* (*acbb* is another).

Let us then introduce some concepts which are needed for the algorithmic descriptions to follow. A pair (i,j) defines a *match*, if $X[i] = Y[j]$. The set of all matches is

$$M = \{(i,j) \mid X[i] = Y[j],\ 1 \leq i \leq m,\ 1 \leq j \leq n\}.$$

Each match belongs to a class $C_k = \{(i,j) \mid (i,j) \in M \text{ and } R[i,j] = k\}$, $1 \leq k \leq r$. Often, it is also convenient to define a pseudoclass $C_0 = \{(0,0)\}$. A match which belongs to C_k is called a *k-match*. In the preceding figure, the encircled and the boxed matches having value k define the class C_k. Evidently, each match belongs exactly to one class. Thus, the classes partition all matches of M. Some of the



k-matches are more important (the boxed ones) than others (the encircled ones). To see this, consider matches (i,j) and (i',j') of C_k for which $(i = i'$ and $j \leq j')$ or $(i \leq i'$ and $j = j')$. Each element of C_{k+1} which can follow (i',j'), can also follow (i,j) in the lcs. Thus, it is enough to concentrate only on these *dominant matches* (i,j) when solving the problem. Let D_k denote the set of all dominant k-matches. Furthermore, let $D = \bigcup_{k>0} D_k$ and $d = |D|$.

In the above figure, the regions where the $R[i,j]$ -values are equal have been bounded by broken lines, so-called *contours*. The k-matches lie immediately below the k'th contour (1-matches are below the uppermost line, 2-matches below the next and so on). The corners of the lines are defined by the locations of the dominant matches, which obey the *ordering property*: if $D_k = (i_1,j_1),(i_2,j_2),\ldots,(i_\ell,j_\ell)$, the matches can be renumbered so that $i_1 < i_2 < \ldots < i_\ell$ and $j_1 > j_2 > \ldots > j_\ell$.

There are three strategies for locating the dominant matches. First, we can process the table row-wise looking for all dominant matches, one for each contour, on the current row. Using auxiliary data structures this can be done efficiently, concentrating only on dominant matches. Another strategy is to advance from contour to contour. In this approach, we may have to check a large region (several rows) of the table in every step and the regions of two consecutive steps may overlap significantly. In order to avoid this, a sophisticated trick is used to restrict the study only to small non-overlapping regions. The third approach tries to advance diagonalwise starting from $R[0,0]$ and trying to reach the lower right corner as greedily as possible.

3. Three approaches to solve the problem

3.1. Processing the Table Row by Row

The row-wise processing is inherited from the traditional approach for filling the dynamic programming table. However, this time we concentrate only on those table entries which correspond to a match. Each dominant match defines a new corner to a contour line. To maintain the columns where all contour lines cross the current row, we use the array $MinYPrefix[1..p]$, where $MinYPrefix[\ell]$ gives the Y-index where the ℓ'th contour line is located. As the name of the array suggests, the value of $MinYPrefix[\ell]$ may be regarded as a cursor, which indicates the minimum length prefix of Y that is needed to produce a common subsequence of length ℓ with the first i elements of X. Value p denotes $r(X[1..i],Y[1..n])$, that is, the number of contour lines crossing row i. Initially, the values of $MinYPrefix$ are initialized to 'undefined'.

Given the example strings $X = abcdbb$ and $Y = cbacbaaba$, the values of the array change as follows (undefined values are represented by $n+1$; the leftmost entry acts as a sentinel and is set to zero):

	MinYPrefix						
Row	0	1	2	3	4	5	6
1	0	3	10	10	10	10	10
2	0	2	5	10	10	10	10
3	0	1	4	10	10	10	10
4	0	1	4	10	10	10	10
5	0	1	2	5	10	10	10
6	0	1	2	5	8	10	10

To maintain the $MinYPrefix$ values when moving from row to row, we need the following result given in [22].

Update rule. Let us assume that we are processing row i. For each open interval $MinYPrefix[\ell]..MinYPrefix[\ell+1]$, $(\ell = 0,..,r)$, find the matches (i,j) which fall into it (i.e. matches for which the j value is in the interval). The right boundary of the interval is kept unchanged, if no such match exists. Otherwise, it is updated to the smallest such j value (leftmost match in the interval). Note that the updates are simultaneous.

For example, when moving from row 2 to row 3 in the above example, we notice that $X[3] = Y[4]$ and $MinYPrefix[1] < 4 \leq MinYPrefix[2]$, so we update $MinYPrefix[2]$ to 4. The general scheme for advancing in the dynamic programming table is the following.

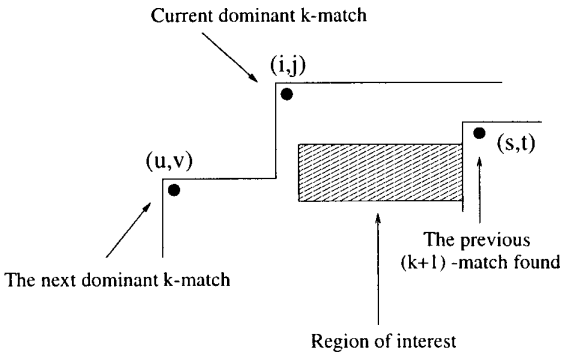
```
begin
(1) for i := 1 to m do MinYPrefix[i] := n+1;
(2) MinYPrefix[0] := 0;  r := 0;
(3) for i := 1 to m do
      /* Update the array values for row i. */
(4)   for j := 0 to r pardo   /* Do in parallel for each j. */
(5)     if range [MinYPrefix[j]+1..MinYPrefix[j+1]-1]
                contains matches then
(6)     begin
          MinYPrefix[j+1] :=
                min { ℓ | (i,ℓ) is a match in this range};
(7)       if j = r then r := r+1;
        end;
    return r;
end;
```

Several ways to perform the serialization of loop (4) and the minimization on line (6) have been devised. In the seminal paper of Hunt and Szymanski [22], each row is processed from right to left scanning through all matches and updating the *MinYPrefix* for each of them (the leftmost in the range remains as the final *MinYPrefix* value as imposed by the update rule). For each match, the corresponding range can be found using binary search because the values in the array *MinYPrefix* are in strictly increasing order. The time complexity is then $O(|M| \log n)$, where $|M|$ denotes the number of all matches. Independently, Mukhopadhyay [30] constructed a very similar implementation. The only basic difference is the processing order: from bottom row up, each row from left to right. Later, Hsu and Du noticed that we are actually performing a special kind of merge between *MinYPrefix* and the list of matches on the current row [21]. Due to this symmetry, we can perform the binary search also to the match list. This results in $O(rm \log(n/r) + rm)$ time complexity, if the merging process is performed using a sophisticated technique. In [26], Kuo and Cross process the matches from left to right, thus avoiding unnecessary updates to *MinYPrefix*. The same idea was also introduced by Apostolico and Guerra [3, 8]. In the same paper they show how we can concentrate only on the dominant matches while processing each row. In [35], Rick describes another implementation for the Apostolico and Guerra algorithm, one which runs in time $O(\sigma n + rm)$.

If X is long or σ is small, the values of the first entries in *MinYPrefix* form quite quickly a sequence of consecutive integers $1,2,..,k$. The reason for this is that we soon reach a state where the first k contours go through columns $1,2,...,k$. When this happens, we don't have to consider these index positions during the update anymore, restricting the active region of the array from below. Clearly, the number of contours traversing through the current row give an upper bound for the region. A sophisticated way to maintain the relevant index range is to study rows and columns at the same time [35]. That is, at step i, we update the *MinYPrefix* arrays both for the i'th row and the i'th column. Essentially, we are traversing along the diagonal of the table and when we detect, during the row-wise examination, a dominant match found by the columnwise computation (or vice versa), we know that all dominant matches of that contour have been found. Although a bit tricky, the implementation of this idea has proved to be very fast in practice. Its theoretical running time is $O(\sigma n + r(n-r))$.

3.2. Advancing from Contour to Contour

A disadvantage of the row-wise approach is that we have to scan the matches corresponding to the entire Y string for each row. Rick's method [35] avoids this (partly) by trying to compose whole contours as soon as possible. In the genuine contour-to-contour approach, we scan through all dominant matches on contour k and for each such match, determine all possible dominant $k+1$ -matches as 'seen' by the current dominant k-match. Clearly, there are at most σ of those. Furthermore, we can restrict the region where the dominant $k+1$-matches are to be found (shaded area):



The *region of interest* is defined by index ranges $[s+1..u] \times [j+1..t-1]$. The boundaries are defined as follows. First, the maximum column and minimum row are determined by the previous $(k+1)$-match and the ordering property of the dominant matches. If no previous $(k+1)$-match exists, the column is n and the row is $i+1$. Second, the minimum column of region of interest is defined by the position of the current dominant k-match. And finally, the maximum row is given by the next dominant k-match, since all dominant $(k+1)$-matches which are below row u can be seen by the match (u,v). If no next $(k+1)$- match exists, the boundary is m. Once the region of interest has been determined, it can be processed using the row-wise approach restricted to this area. Let us assume that the leftmost dominant match in this region lies on column ℓ. Then the next k-match (u,v) checks the area $[u+1..w] \times [v+1..\ell-1]$, where w is the row index of the next k-match after the (u,v), if any. When all k-matches have been processed this way, we move to contour $k+1$ and repeat, until an empty contour is found.

```
    begin
(1)   D_0 := {(0,0)};   /* Imaginary 0-match. */
(2)   k := 0;
(3)   while D_k is non-empty do
      begin
(4)     while there are dominant matches (i,j) in D_k do
        begin
          /* Process dominant matches from right to left */
(5)       Find the region of interest for (i,j);
(6)       for each dominant (k+1)-match (s,t) in
                the region of interest do add (s,t) to D_{k+1};
        end;
        k := k+1;
      end;
      return k-1;
    end;
```

Hirschberg (1977) gave two algorithms for going from contour to contour in [19]. The first one uses the ordering property of the dominant matches to find the k'th contour line. For this, it processes the table row-wise from index $i+1$ to m, where i denotes the row where contour $k-1$ started. The complexity of this algorithm is $O(rn+n\log n)$. The second algorithm of [19] is basically the same as the first except that now a lower bound for the length of the lcs is fixed in advance. Because of this, the table area where the contours lie is bounded. The time complexity is $O(r(m-r)\log n)$ showing that the method is fast when the inputs are similar.

Hsu and Du (1984) noted that we can use binary search to the list of matches in order to find the dominant matches [21]. The search is effectively limited to a range, the lower bound of which is defined by the Y-index of the next match on the previous contour. The time complexity of their algorithm is $O(rm+\log n/m+rm)$ [4]. In [8], Apostolico and Guerra (1987) introduce alternative data structures to support the forming of the lcs: *Close* vector is a compact representation of the *Closest* table defining for each symbol of Σ its nearest occurrence in Y after a given position j (these are discussed in the next section in more detail). The time complexity of their algorithm is $O(rm\log\min\{\sigma,m,2n/m\})$ when using *Close* vectors. Chin and Poon (1990) take advantage of the *Closest* table also [13]. Moreover, they introduce another table of size σn; this helps in finding, at any given row i, the positions of the next *different* symbols in X. This data structure is consulted when searching for dominant matches in the region of interest. The resulting method works nicely for small alphabets (since the construction of the tables takes $O(\sigma n)$ time) giving time complexity $O(\sigma n+\min\{\sigma d,rm\})$.

3.3. Filling the Table Diagonalwise

In this section we study methods which fill the dynamic programming table greedily, trying to reach the goal entry $R[m,n]$ as fast as possible. Most of these algorithms are based on the calculation of the edit distance between the input strings. To accomplish this, the diagonals are numbered so that entry (i,j) is on diagonal $j-i$. Thus, elements $(0,0),(1,1),...,(m,m)$ are on diagonal zero, diagonals $1,2,3,...,n$ lie above it and diagonals $-1,-2,...,-m$ below it. Diagonal $n-m$ ends in the lower right corner (m,n). The diagonals are processed in a systematic manner and each of them is extended in two steps. First, we check the diagonals immediately above and below (if they exist), which one reaches lower down in the table. The initial position (i,j) for the current diagonal is determined either by moving one row down from the diagonal above or one column right from the diagonal below. Once the initial position (and the edit distance value) has been determined, we move further down along the current diagonal as long as the symbols of X and Y match. If $X[i..i+\ell]=Y[j..j+\ell]$ and $X[i+\ell+1]\neq Y[j+\ell+1]$, we know that the edit distance remains the same up to position $(i+\ell,j+\ell)$. This row value is recorded in an array indexed by the diagonals (no dynamic programming table is needed).

```
    begin
(1) currentDist := 0;
(2) while currentDist ≤ m+n do
    begin
(3)   while there exist diagonals which can be extended
        with value currentDist do
      begin
(4)     Extend the diagonal: use the two-step process above;
(5)     if the diagonal reaches the lower right corner
            then return (n+m-currentDist)/2;
      end;
      currentDist := currentDist+1;
    end;
  end;
```

The above method is almost identical to that given in 1985 by Miller and Myers [29]. The processing of the diagonals can be made more sophisticated using two maneuvers: first, it is advantageous to extend the diagonals from the opposite corners of the table [31]. In this way, the central diagonals touch sooner each other and the number of studied diagonals is reduced. Second, we can stick close to the main diagonal by using the concept of *compressed distance* [40]: instead of calculating the true edit distance, we can concentrate on the number of deletions only. This reduces the width of the band that we have to check resulting in $O(pn)$ algorithm, where p is the number of deletions needed. Thus, the algorithm executes quickly when the shorter input string is a substring of the other.

A quite different solution to the diagonalwise approach was given by Nakatsu, Kambayashi and Yajima [33]. In this case, the lcs is found by checking systematically, how long common subsequence can be found for Y and the *substring* $X[\ell..i]$. That is, starting from $X[i]$, we select consecutive symbols from X and traverse Y from right to left searching for matches until all of Y has been 'used up' ($X[\ell-1]$ can not be found). These results are then combined cleverly to obtain the lcs. The search is terminated as soon as we know that the suffix of $X[1..i]$ can not give us any better results. Because of this, the method executes fast if the lcs is long; its theoretical time complexity is $O(n(m-r))$.

4. Supporting Data Structures

The most essential operation in practically all lcs algorithms is the following: given an index j and a symbol a, find the next/previous occurrence of this symbol in the input

string starting from position j. The data structure supporting this operation can be chosen from several alternatives:

(a) no data structures; use linear search along the input string from position j on,

(b) use a linked list for each input symbol a, containing the positions of all occurrences of a in the input string. Using the traditional naming convention [22], this structure is called a *MatchList*. It is easy to see that all lists can be formed in time proportional to the length of the input string. This structure is adequate, if the requests are issued in a highly systematic way and no random access is needed,

(c) collect the information of *MatchLists* into a contiguous memory area (array) to facilitate binary search. If all *MatchLists* have been stored into a shared memory area, an index giving the start position of each list has to be built,

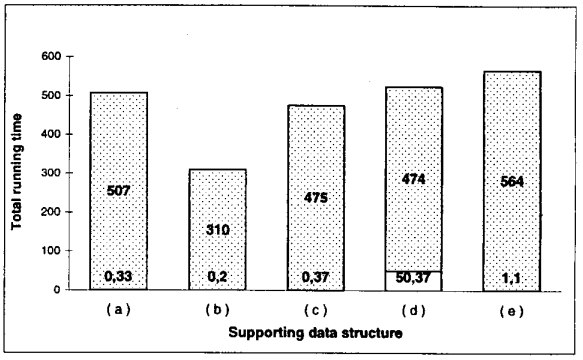
(d) construct a table of size $\sigma \cdot$*(length of input string)* with which the result is obtained in constant time independently of the arguments j and a, see e.g. [8]. The time and space needed for this *Closest*-table, may be unacceptably large in some situations,

(e) build a compromise between (c) and (d). This data structure, called *Close* vector, and introduced in [8], gives the result in time $O(\log\sigma)$ by using space which is proportional to the length of the input string.

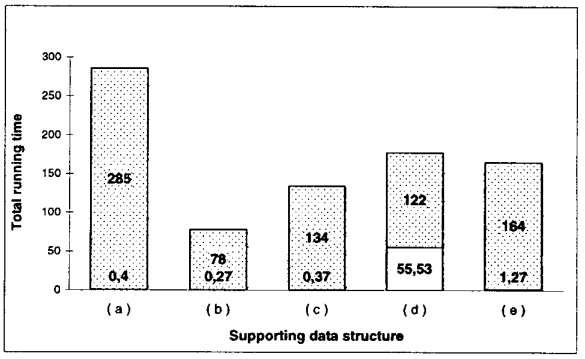
To find out the effect of the choice of the data structure we implemented the algorithm of Apostolico and Guerra (called Algorithm 4 in [8]) using all of the above schemes. The algorithm was simplified to search only the length of the lcs (consequently, most of the memory management could be discarded). Since the method is a variant of [22] and belongs to the category of row-wise methods, the positions of the input symbols are accessed regularly from left to right in Y. Therefore, linear data structures are expected to be adequate in this situation.

The execution time using the five different techniques is shown in Figure 1. The dotted area represents the actual search time and the empty area shows the preprocessing time. The input parameters were $n = m = 4000$, $\sigma = 128$ and the symbol distribution was uniform. To see the significance of the data structure during the search process, we selected $r = 0.9m$ (Figure 1(a)). On the other hand, when r is very small, the time needed to initialize the data structure becomes more pronounced. This is shown in Figure 1(b), where $r = 0.15m$.

From the results of 1(a) and 1(b) we can see that the only data structure which has a significant construction cost is the *Closest* table. When the problem instance is hard



(a) Performance when $r = 0.9m$.



(b) Performance when $r = 0.15m$.

Figure 1: Execution times using different elementary data structures.

and the searching phase dominates (Figure 1(a)), the construction cost is relatively small (roughly 10% for *Closest*). Somewhat counterintuitively, the search phase becomes only slightly faster with *Closest*, if compared to the search using linear representations. The best overall performance is obtained using linked lists. Even the linear scan (without any data structures) is a good alternative when r is large. This is partially due to the property of the actual search algorithm which terminates the search of matches on the current row once the length of the lcs has increased by one (and this happens frequently when r is large).

In Figure 1(b), the construction of the *Closest* table takes already 1/3 of the total running time. As expected, the linear scan results now in the slowest search time, since it has to traverse most of the input string on each row. Again, the benefit of forming the *Closest* table is not evident when comparing the search phase timings for the table and the

Row-by-row methods

Tag	Source	Original name	Time Complexity
WF	Wagner and Fischer [38]		$O(mn)$
HS	Hunt and Szymanski [22]	Alg. 2	$O(\mid M \mid \log n)$
Mu	Mukhopadhyay [30]	Alg. 2	$O(\mid M \mid \log n)$
HD-II	Hsu and Du [21]	Alg. 2	$O(rm\log(n/r)+rm)$
AG-II	Apostolico and Guerra [8]	Alg. 4	$O(m\log n+d\log(2mn/d))$
KC	Kuo and Cross [26]	HS*	$O(\mid M \mid +n(r+logn))$
Ri-I	Rick [35]	New $O(r(n-r))$	$O(\sigma n+\min\{r(n-r),rm\})$
Ri-II	Rick [35]	New $O(\sigma d)$	$O(\sigma n+\min\{\sigma d,rm\})$

Contour methods

Tag	Source	Original name	Time Complexity
Hi	Hirschberg [19]	ALGD	$O(rn+n\log n)$
HD-I	Hsu and Du [21]	Alg. 1	$O(rm\log(n/m)+rm)$
AG-I	Apostolico and Guerra [8]	Alg. 2	$O(rm+\sigma n+n\log\sigma)$
CP	Chin and Poon [13]		$O(\sigma n+\min\{\sigma d,rm\})$

Diagonal methods

Tag	Source	Original name	Time Complexity
NKY	Nakatsu et al. [33]		$O(n(m-r))$
MM	Miller and Myers [29]	fcomp	$O(m(m-r))$
WMMM	Wu et al. [40]	Compare	$O(n(m-r))$

LIST OF STUDIED ALGORITHMS IN THE THREE CATEGORIES

MatchList representations. In fact, considering the overall execution time, the simple linked list outperforms all others.

It should be noted that this experiment is related to a specific row-wise method and that the situation may change in contour and diagonal approaches. Moreover, we have concentrated here only on the most basic operation found in the lcs algorithms. For the row-wise methods, for example, an important operation is also a special kind of merge [21] introduced in section 3.1.

The underlying run-time memory management strategy may have a surprisingly large effect on the total running time too, especially, when we search for the actual subsequence (and not just its length). Implementing the memory management carefully we have tried to eliminate factors related to these issues. In all test runs reported in this paper, we used a uniform approach to allocate space (more details of this in [11]).

5. Comparison results

When implementing the algorithms, we tried to preserve the spirit of the original description (correcting inaccuracies, errors and augmenting them with a stage gathering the actual subsequence, if the source does not give it), including the choice of the auxiliary data structures. To see the differences in time and memory consumption, we implemented two different versions of each method: one which finds only the length of the lcs and another which determines the actual sequence also.

Parameters controlling the input generation process can have a substantial effect on the performance of the methods. For example, if the strings are short, the advantage gained by constructing large and sophisticated data structures in the initialization phase of the scheme is lost. Here we want to concentrate on the asymptotic behaviour of the algorithms. For this purpose, the length of the input strings was chosen to be 'large enough' ($n = m = 4000$). Since X and Y are of equal length, the anomalies caused by the situation $m \ll n$ are not detected in these tests (unfortunately, this important topic would require a thorough discussion so we have to defer it to [11]).

The theoretical time complexity of a method helps to deduce how fast it solves the extreme cases where r is close to zero or m. Therefore, we concentrate only on the most difficult problem instances, those which yield a lcs of length $m/2$ (roughly). For the input generation process, the reader is referred to [24]. Two distribution types and two alphabet sizes were used: (a) to model the properties of biosequences we chose a uniform distribution using $\sigma = 8$ and (b) to simulate natural language, we used a Zipfian distribution with $\sigma = 256$. In the latter, the probability of the i'th most probable symbol was chosen to be $p_i = \frac{1}{i \cdot H_\sigma}$, $i = 1,\ldots,\sigma$, where H_n denotes the n'th harmonic number.

The algorithms selected for a detailed study are given above. All implementations were written in C, they were not optimized at the source code level but the -O3 switch was used in the compilation (gcc compiler, version 2.7.2.3.f.1). The tests were performed using a Sparc Work-

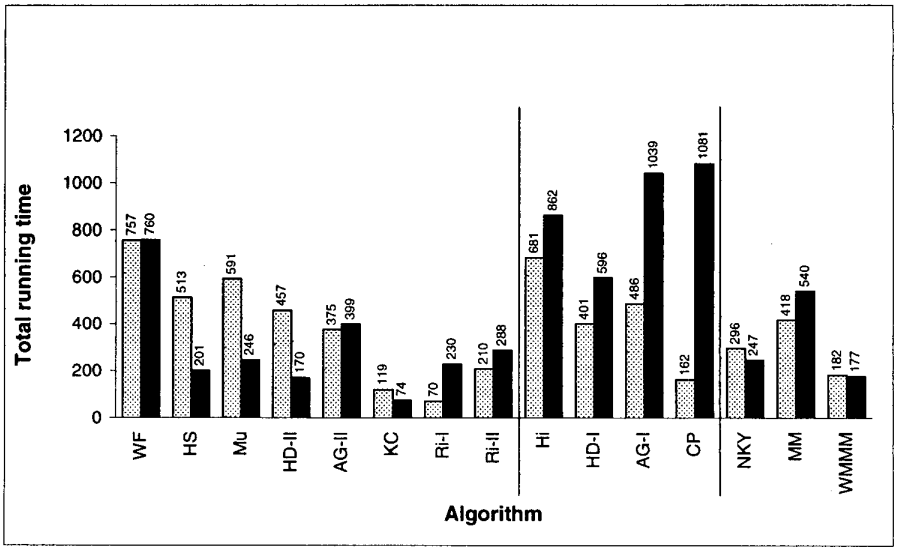


Figure 2: Execution times of the algorithms when solving r.
Shaded bar: $\sigma = 8$, *uniform distribution; Black bar:* $\sigma = 256$, *skew distribution*

Station under Unix.

Figure 2 gives the running times of these algorithms when the lcs length is calculated. The results have been organized by the three algorithmic approaches: the row-wise methods are on the left, contour-based methods in the middle and methods advancing diagonally on the right. Studying the results of the different categories, we note that no approach is clearly superior to others. The fact that the results of the contour methods are only moderate seem to be more due to algorithmic deficiencies than the actual approach itself. When comparing the results for the two input types, we see that there is no general trend even inside any category; the only exception is the contour approach which makes more work when the alphabet size is large and the distribution is skew. Considering all methods, it is evident that the ones which form the two-dimensional *Closest* table (AG-II, Ri-I, Ri-II, AG-I, CP) are more appropriate for cases where σ is small.

Somewhat surprisingly, a less known variant of the Hunt-Szymanski algorithm, KC, seems to perform very well. Also Ri-I has good performance for both input types. These two row-wise methods stand out from the others. The diagonal methods are inherently customized for cases where r is known to be close to m, so their full power is not shown in the figure. In spite of that, the results of WMMM can be considered very competitive to KC and Ri-I. We also gathered statistics about the memory consumption used by the algorithms. Summarizing these results, WF and Hi needed around 60MB space while, in the other end of the scale, KC, HS, HD-II, MM, NKY and WMMM used only about 40KB.

When also the actual subsequences are solved, the running times (and memory consumption) increase as shown in Figure 3. In general, the increase is around 10% – 30% but for some algorithms it is even 150% (NKY, Ri-I). As to the individual execution times, there are no big surprises when the results of Figure 2 are known: the methods of Kuo & Cross (KC), Rick (Ri-I) and Wu et al. (WMMM) are the fastest ones. For the uniform (skew) distribution, these methods used space 16MB, 9MB and 7MB (5MB, 10MB and 1MB), respectively.

6. Discussion

A fair and comprehensive comparison of the lcs algorithms is a demanding task for several reasons: the proposed algorithms seem to have inaccuracies, if not actual errors; the procedure to recover the longest common subsequence

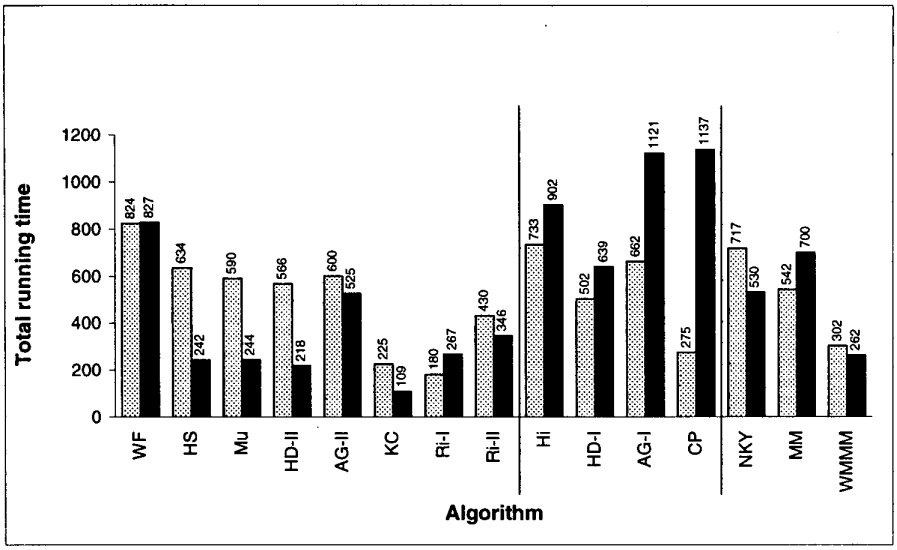


Figure 3: Execution times of the algorithms when solving *lcs*.
Shaded bar: $\sigma = 8$, *uniform distribution; Black bar:* $\sigma = 256$, *skew distribution*

has often been omitted; the supporting data structures have not been discussed thoroughly; the memory management has been ignored etc. In this paper we have tried to discuss these issues and bring out the benefits and disadvantages of various decisions in a unified framework.

The results of the comparison show that for 'typical' situations, no single lcs algorithm outperforms all others. However, as a rule of thumb, we can say that the algorithms of Kuo-Cross and Rick seem to be among the most plausible ones for a wide range of input parameters and independently of the task type (only length needed, also sequence needed). As to the space requirements, we note that the differences can be substantial among the implementations. However, for the two fastest ones the variance in the maximum amount of space is small.

The literature on lcs algorithms provides a wide range of facilities to tune almost any method for better performance. Typically, this is accomplished by restricting the search space of the algorithm. Also, the role of the data structure(s) assisting in the crucial operations during the search stage is central for the overall performance. It seems that making proper choices it is possible to construct variants which outperform the fundamental algorithms presented here [11].

References

[1] **Aho, A.V., Hirschberg, D.S. & Ullman, J.D.:** Bounds on the Complexity of the Longest Common Subsequence Problem, *Journal of the ACM, Vol. 23, No. 1, 1976, pp. 1-12*

[2] **Altschul, S.F., Gish, W., Miller, W. Myers, E.W & Lipman, D.J.:** Basic Local Alignment Search Tool, *J. Mol. Biol., Vol. 215, 1990, pp. 403–410*

[3] **Apostolico, A.:** Improving the Worst-Case Performance of the Hunt-Szymanski Strategy for the Longest Common Subsequence of Two Strings, *Inform. Proc. Lett., Vol. 23, 1986, pp. 63-69*

[4] **Apostolico, A.:** Remark on the Hsu-Du New Algorithm for the Longest Common Subsequence Problem, *Inf. Proc. Lett., Vol. 25, June 1987, pp. 235-236*

[5] **Apostolico, A., Browne, S. & Guerra, C.:** Fast Linear-Space Computations of Longest Common Subsequences, *Theor. Comp. Sc., 1992, pp. 3-17*

[6] **Apostolico, A. & Galil. Z. (eds.):** Pattern Matching Algorithms, *Oxford University Press, 1997*

[7] **Apostolico, A. & Guerra, C.:** A Fast Linear Space Algorithm for Computing Longest Common Subsequences, *Proc. of the 23rd Allerton Conf., Monticello, IL, 1985, pp. 76-84*

[8] **Apostolico, A. & Guerra, C.:** The Longest Common Subsequence Problem Revisited, *Algorithmica, No. 2, 1987, pp. 315-336*

[9] **Argos, P., Vingron, M. & Vogt, G.:** Protein Sequence Comparison: Methods and Significance, *Protein Engineering, Vol. 4, No. 4, 1991, pp. 375–383*

[10] **Bergroth, L., Hakonen, H. & Raita, T.:** New Approximation Algorithms for Longest Common Subsequences, *Proc. of SPIRE'98, String Processing and Information Retrieval: A South American Symposium*, Santa Cruz de La Sierra, Bolivia September 9 – 11, 1998, pp. 32–40.

[11] **Bergroth, L., Hakonen, H. & Raita, T.:** Performance Evaluation of the Longest Common Subsequence Algorithms, *in preparation*

[12] **Chin, F. & Poon, C.K.:** Performance Analysis of Some Simple Heuristics for Computing Longest Common Subsequences, *Algorithmica, Vol. 12, 1994, pp. 293–311*

[13] **Chin, F.Y.L. & Poon, C.K.:** A Fast Algorithm for Computing Longest Common Subsequences of Small Alphabets Size, *J. of Inf. Proc., Vol. 13, No. 4, 1990, pp. 463–469*

[14] **Crochemore, C. & Rytter, W.:** Text Algorithms, *Oxford University Press*, 1994

[15] **Fraser, C.B.:** Subsequences and Supersequences of Strings, *Ph.D. Thesis, University of Glasgow, 1995*

[16] **Fredman, M.L.:** On Computing the Length of Longest Increasing Subsequences, *Disc. Math., Vol. 11, 1975, pp. 29-35*

[17] **Gusfield, D.:** Algorithms on Strings, Trees and Sequences, *Cambridge University Press*, New York, 1997

[18] **Hirschberg, D.S.:** A Linear Space Algorithm for Computing Maximal Common Subsequences, *Comm. of the ACM, Vol. 18, No. 6, 1975, pp. 341-343*

[19] **Hirschberg, D.S.:** Algorithms for the Longest Common Subsequence Problem, *Journal of the ACM, Vol. 24, No. 4, 1977, pp. 664-675*

[20] **Hirschberg, D.S.:** An Information Theoretic Lower Bound for the Longest Common Subsequence Problem, *Inf. Proc. Letters, Vol. 7, No. 1, 1978, pp. 40-41*

[21] **Hsu, W.J. & Du, M.W.:** New Algorithms for the LCS Problem, *J. Comp. and System Sc., Vol. 29, 1984, pp. 133-152*

[22] **Hunt, J.W. & Szymanski, T.G.:** A Fast Algorithm for Computing Longest Common Subsequences, *Comm. of the ACM, Vol. 20, No. 5, 1977, pp. 350-353*

[23] **Jiang, T. & Li, M.:** On the Approximation of Shortest Common Supersequences and Longest Common Subsequences, *SIAM J. of Comp., Vol. 24, No. 5, 1995, pp. 1122-1139*

[24] **Johtela, T., Smed, J., Hakonen, H. & Raita, T.:** An Efficient Heuristic for the LCS Problem, *Proc. of the Third South American Workshop on String Processing, WSP'96, Recife, Brazil, August 1996, pp.126–140*

[25] **Kumar, S.K. & Rangan, C.P.:** A Linear-Space Algorithm for the LCS problem, *Acta Informatica, vol. 24, 1987, pp. 353-362*

[26] **Kuo, S. & Cross, G.R.:** An Improved Algorithm to Find the Length of the Longest Common Subsequence of Two Strings, *ACM SIGIR Forum, Vol. 23, No. 3–4, 1989, pp. 89–99*

[27] **Maier, D.:** The Complexity of Some Problems on Subsequences and Supersequences, *JACM, Vol. 25, No. 2, 1978, pp. 322-336*

[28] **Masek, W.J. & Paterson, M.S.:** A Faster Algorithm for Computing String-Edit Distances, *Journal of Computer and System Sciences, Vol. 20, No. 1, pp. 18–31*

[29] **Miller, W. & Myers, E.W.:** A File Comparison Program, *Softw. Pract. Exp., Vol. 15, No. 11, November 1985, pp. 1025-1040*

[30] **Mukhopadhyay, A.:** A Fast Algorithm for the Longest-Common-Subsequence Problem, *Information Sciences, Vol. 20, 1980, pp. 69-82*

[31] **Myers, E.W.:** An O(ND) Difference Algorithm and its Variations, *Algorithmica, Vol. 1, 1986, pp. 251-266*

[32] **Myers, E.W.:** An Overview of Sequence Comparison Algorithms in Molecular Biology, *Technical Report, TR 91-29, Dept. of CS, The University of Arizona, Tucson, Arizona*

[33] **Nakatsu, N., Kambayashi, Y. & Yajima, S.:** A Longest Common Subsequence Algorithm Suitable for Similar Texts, *Acta Informatica, Vol. 18, 1982, pp. 171-179*

[34] **Pearson, W.R. & Lipman, D.J.:** Improved Tools for Biological Sequence Comparison, *Proc. Natl. Acad. Sci. USA, Vol. 85, April 1988, pp. 2444–2448*

[35] **Rick, C.:** A New Flexible Algorithm for the Longest Common Subsequence Problem, *in Galil, Z. & Ukkonen, E. (eds): Proc. of Combinatorial Pattern Matching*, 6th Annual Symposium, Espoo, Finland, July 1995, pp. 340–351. Appeared also as *Lecture Notes in Computer Science*, Vol. 937

[36] **Sankoff, D. & Kruskal, J.B. (eds.):** Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, *Addison-Wesley, 1983*

[37] **Stephen, G.A.:** String Searching Algorithms, *World Scientific*, 1994

[38] **Wagner, R.A. & Fischer, M.J.:** The String-to-String Correction Problem, *Journal of the ACM, Vol. 21, No. 1, January 1975, pp. 168-173*

[39] **Wong, C.K. & Chandra, A.K.:** The String-to-String Correction Problem, *Journal of the ACM, Vol. 21, 1976, pp. 13-16*

[40] **Wu, S., Manber, U., Myers, G. & Miller, W.:** An O(NP) Sequence Comparison Algorithm, *Inf. Proc. Lett., Vol. 35, September 1990, pp. 317-323*