RESEARCH REPORT

INSITITUT FÜR INFORMATIK DER UNIVERSITÄT BONN
Report No. 85123-CS

New Algorithms for the Longest Common Subsequence Problem

CLAUS RICK

ISSN 0177-3623

Report No. 85123-CS

New Algorithms for the Longest Common Subsequence Problem

Claus Rick

October 1994

Institut für Informatik, Universität Bonn, Römerstrasse 164, 53117 Bonn

New Algorithms for the Longest Common Subsequence Problem

Claus Rick
University of Bonn
Computer Science Department IV
Römerstr. 164
53117 Bonn
GERMANY
E-Mail: rick@cs.uni-bonn.de

October 1994

Abstract

Given two sequences $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$, $m \leq n$, over some alphabet Σ, a common subsequence $C = c_1c_2 \ldots c_l$ of A and B is a sequence that can be obtained from both A and B by deleting zero or more (not necessarily adjacent) symbols. Finding a common subsequence of maximal length is called the Longest Common Subsequence (LCS) Problem. Two new algorithms based on the well-known paradigm of computing minimal matches are presented. One runs in time $O(ns + \min\{ds, pm\})$ and the other runs in time $O(ns + \min\{p(n-p), pm\})$ where $s = |\Sigma|$ is the alphabet size, p is the length of a longest common subsequence and d is the number of minimal matches. The ns term is charged by a standard preprocessing phase. When $m \approx n$ both algorithms are fast in situations when a LCS is expected to be short as well as in situations when a LCS is expected to be long. Further they show a much smaller degeneration in intermediate situations, especially the second algorithm. Thus, they are more flexible than previous algorithms which, in general, specialized either on short or long longest common subsequences.

1 Introduction

1.1 Basic definitions

Let $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n, m \leq n$, be two sequences over an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}$ of size s. It is assumed that two symbols can be compared in constant time. A sequence $C = c_1c_2 \ldots c_l$ is called a *subsequence* of A if C can be obtained from A by deleting a number of (not necessarily consecutive) symbols from A, that is it exists a mapping $F : [1, 2, \ldots, l] \longrightarrow [1, 2, \ldots, m]$ having the two properties

1. $F(i) = k$ only if $c_i = a_k$
2. $i < j \Longrightarrow F(i) < F(j)$ (F is monotone and strictly increasing)

C is said to be a *common subsequence* of A and B iff C is a subsequence of A and also a subsequence of B. The *Longest Common Subsequence* (LCS) *Problem* then is to determine a common subsequence of A and B of maximal length. In general C is not unique.

An ordered pair (i, j) of positions in A and B is called a *match* iff $a_i = b_j$. The set of all matches is $R = \{(i, j) \mid 1 \le i \le m, 1 \le j \le n \wedge a_i = b_j\}$ and $r = |R|$. Now the LCS Problem can be viewed as finding a longest sequence of matches that is strictly increasing in both components.

Let $|A|$ denote the length of a string A and let $A_i, 0 \le i \le |A|$, denote the length i prefix of A, i. e. $A_i = a_1 a_2 \ldots a_i$. In particular $A_0 = \epsilon$ is the empty string of length zero. Define $L_{i,j} = \max\{|C| \ : \ C$ is a common subsequence of A_i and $B_j\}$, $0 \le i \le m, 0 \le j \le n$, to be the length of a LCS between prefixes A_i and B_j. As a special case $L_{m,n}$ is the length of a LCS between A and B.

1.2 Previous results

The first algorithm to solve the LCS Problem has been a dynamic programming approach that was discovered by several different scientist independently [19, 21]. They observed that the following recursion holds for $L_{i,j}$ (a detailed proof can be found in [7]).

$$L_{i,j} = \begin{cases} 0 & \text{if} \quad i = 0 \text{ or } j = 0 \\ L_{i-1,j-1} + 1 & \text{if} \quad a_i = b_j \\ \max\{L_{i-1,j}, L_{i,j-1}\} & \text{if} \quad a_i \neq b_j \end{cases}$$

By employing an array $L[0..m, 0..n]$ initially filled with zeros and executing

for $i := 1$ **to** m **do**
  for $j := 1$ **to** n **do** **if** $a_i = b_j$ **then** $L[i,j] := L[i-1, j-1] + 1$
    else $L[i,j] := \max\{L[i-1,j], L[i,j-1]\}$;

all values $L_{i,j}$ can be computed in time and space $\Theta(mn)$. In order to reconstruct a LCS we can save a pointer with each cell $L[i,j]$ that indicates which of the terms $L[i-1, j-1] + 1, L[i-1,j], L[i,j-1]$ was used to define $L[i,j]$. It was shown in [9] that linear space suffices to compute both the length of a LCS and a LCS while maintaining the time bound.

Example: Let $\Sigma = \{a, b, c, d\}$ be an alphabet and let $A = abcdbb$ and $B = cbacbaaba$ be two input sequences over Σ. The above code computes the L-Matrix shown in Figure 1.

The asymptotically fastest general solution takes time $O(n^2/\log n)$ [15] and uses the "Four Russians" trick. A lot of algorithms have been developed that, although not improving the general $O(mn)$ time bound of the dynamic programming approach, exhibit a much better performance by specializing on certain classes of pairs of sequences. These algorithms are sensitive for other problem parameters, e.g. the length p of a LCS (the output parameter), the number of (minimal) matches or the alphabet size s.

There are two main paradigms that are used to solve the LCS Problem. One is to construct a special directed graph, called an editgraph, that consists of the intersection points on a $(m+1) \times (n+1)$ grid. In addition to the horizontal and vertical edges, there is a diagonal edge from node $(i-1, j-1)$ to node (i, j) if $a_i = b_j$. By assigning a cost of 1 to each horizontal and vertical edge and a cost of 0 to each diagonal edge the LCS Problem reduces to searching for a cheapest-cost path from node $(0, 0)$ to node (m, n) in this graph. The tails of the diagonal edges on a cheapest path then define a LCS for the two strings. This approach

						B					
			c	b	a	c	b	a	a	b	a
		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	1	1	1	1	1	1	1
b	2	0	0	1	1	1	2	2	2	2	2
c	3	0	1	1	1	2	2	2	2	2	2
A d	4	0	1	1	1	2	2	2	2	2	2
b	5	0	1	2	2	2	3	3	3	3	3
b	6	0	1	2	2	2	3	3	3	4	4

Figure 1: L-Matrix computed by the dynamic programming algorithm. Matches are encircled and regions with identical $L_{i,j}$ value are separated through contours.

was first suggested independently by Myers [16] and Ukkonen [20]. It takes time $O(n(n-p))$ and was later improved by Wu et al. [23] to $O(n(m-p))$. Thus, these algorithms suit with a class of sequences where the length p of a LCS is expected to be long.

A lot more algorithms use the paradigm of computing a longest and strictly increasing sequence of (minimal) matches. This approach will be reviewed in Section 2 since it is the basis for the new algorithms presented in this paper, too. The first algorithms using this approach have been invented by Hirschberg [11] and Hunt/Szymanski [13] with processing time $O(pn)$ and $O(m+r\log p)$ respectively. An additional $O(n\log s)$ term has to be added for both methods for a standard preprocessing phase. Later, both algorithms have been refined by Hsu and Du [12] and by Apostolico and Guerra [3] to take time $O(pm\log\frac{n}{p}+pm)$ and $O(pm\log(\min\{s,m,\frac{2n}{m}\}))$ respectively. They perform well when a LCS is expected to be short. There is also an algorithm from Nakatsu et al. [17] that has running time $O(n(m-p))$ and that can be used if longest common subsequences of great length are expected. Other algorithms are from Chin and Poon [5] ($O(ns+\min\{pm,ds\})$) and Apostolico and Guerra [3] ($O(m\log n+d\log(\frac{2mn}{d}))$). Here d denotes the number of minimal matches (see Section 2 for a definition). Algorithms that use linear space are discussed in [9, 4, 14]. Table 1 gives a chronological survey of algorithms for the LCS Problem. Lower bounds on the complexity of the LCS Problem can be found in [1, 10, 22]. For a fixed alphabet of size s there is still a big gap between the linear lower bound $\Omega(ns)$ and the worst case upper bounds of the various algorithms.

1.3 Mission statement

As pointed out in [17] we have to select one of the algorithms *a priori* depending on the kind of sequences we wish to compare. However, this might be very difficult to do, because either we have insufficient knowledge of the nature of the sequences to be compared or the class of sequences is so heterogeneous that both short and long longest common subsequences are

No.	Year	Author(s)	Time	Ref.
1	1974	Wagner, Fischer	$O(mn)$	[21]
2	1977	Hunt, Szymanski	$O(m + r \log p)$	[13]
3	1977	Hirschberg	$O(pn)$ $O((m-p)p \log n)$	[11]
4	1980	Masek, Paterson	$O(n^2/\log n)$	[15]
5	1982	Nakatsu et al.	$O(n(m-p))$	[17]
6	1984	Hsu, Du	$O(pm \log(\frac{n}{p}) + pm)$	[12]
7	1986	Myers	$O(n(n-p))$	[16]
8	1987	Apostolico, Guerra	$O(pm \log(\min\{s, m, \frac{2n}{m}\}))$ $O(m \log n + d \log(\frac{2mn}{d}))$	[3]
9	1990	Chin, Poon	$O(ns + \min\{ds, pm\})$	[5]
10	1990	Wu, Manber, Myers	$O(n(m-p))$	[23]
11	1992	Apostolico et al.	$O(n(m-p))$	[4]

Table 1: Algorithms for the LCS Problem. Time $O(n \log s)$ has to be added to algorithms 2, 3, 6 and 8 for a preprocessing phase.

likely to occur. The later might be the case when searching a data base for entries similar to a query. Moreover, when dealing with sequences of approximately equal length consisting of symbols drawn from a small alphabet in a more or less uniform manner the length of a LCS can be expected to lie in the range between $\frac{1}{3}m$ and $\frac{2}{3}m$, depending on the alphabet size [19]. In this case none of the the above specialized algorithms seems to be well-suited. The later situation seems to be typical for the analysis of macromolecular sequences in biology, where sequences are formed over an alphabet of size four for nuleic acids or over an alphabet of size 20 for proteins. Although the distribution of symbols is not uniform, it can be viewed as more or less random implying longest common subsequences of intermediate length when comparing arbitrary sequences.

Therefore it seemed worth while to look for an algorithm that exhibits a good performance for both short and long longest common subsequences and that can deal with longest common subsequences of intermediate length in a more efficient manner. Indeed it was possible to develop such an algorithm that has time complexity $O(ns+\min\{pm, p(n-p)\})$. It is presented in Section 4.

In Section 2 the paradigm of computing minimal matches is reviewed. Section 3 presents previous algorithms based on this paradigm, namely the algorithm by Hunt and Szymanski [13] and a refined one by Apostolico and Guerra [3]. Then in Section 4 first the new $O(ns + \min\{pm, p(n-p)\})$ algorithm will be developed. Subsequently another $O(ns + \min\{pm, ds\})$ algorithm is given which essentially does the same as the algorithm by Chin and Poon [5], but in a rather different manner. Some experimental results comparing various algorithms for the LCS Problem are presented in Section 5. Section 6 summarizes the achieved results.

2 The Paradigm of Computing Minimal Matches

To improve on the $\Theta(mn)$ running time of the dynamic programming algorithm we have to reduce the number of entries to be computed in the L-Matrix. Since a LCS consists exclusively of matches we loose no necessary information if we restrict ourselves to determine $L_{i,j}$ for matches $(i, j) \in R$ only. A match with greatest value $L_{i,j}$ then gives the length of a LCS.

There are two problems that have to be solved in order to make this approach work. First we must be able to determine the matches in time less than $\Theta(mn)$. This can be achieved easily by the introduction of a preprocessing phase. Second we have to find a way to pass on the information about known longest common subsequences of prefixes. The value of $L_{i,j}$ for a match (i, j) can not, in general, be determined by the inspection of $L_{i-1,j-1}$, because $(i-1, j-1)$ may not be a match and thus the value $L_{i-1,j-1}$ is not available. The regularities observed in the L-matrix (see Figure 1) suggest useful structural properties that could be exploited to solve this problem. The remainder of this section investigates these structural properties.

2.1 Matches

A match (i, j) is called a *successor (predecessor)* of match (i', j') if $i' < i \ \wedge \ j' < j$ $(i' > i \ \wedge \ j' > j)$. We say a match (i, j) appears at position k in a common subsequence of A and B if the k-th symbol of a common subsequence C corresponds to the i-th symbol in A and the j-th symbol in B. Obviously if (i', j') is a successor of (i, j) then (i, j) and (i', j') may appear at directly successive positions in a common subsequence. The *rank* of a match (i, j) is given by the value $L_{i,j}$, i. e. it is the length of a longest and strictly increasing sequence of matches between prefixes A_i and B_j. So a match with rank k may only appear at positions $\leq k$ in a common subsequence of A and B. Now collect matches of the same rank k in *classes* $C_k := \{(i, j) \mid (i, j) \in R \wedge L_{i,j} = k\}$, $k \geq 1$, and let $C_0 := \{(0, 0)\}$. A match $(i, j) \in C_k$ is also called a k-match. It is easy to see that $C_k \cap C_{k'} = \emptyset$, $k \neq k'$, and that $R = \bigcup_{k>0} C_k$.

Our goal will be, starting with class C_0, to determine the rank of more and more matches with the aid of matches for which the rank is known already. First we state some simple observations as

Lemma 2.1

1. *Let (i, j) be a predecessor of (i', j'). Then $(i, j) \in C_k \Longrightarrow (i', j') \in C_{k'}$, $k' \geq k + 1$.*
2. *$\forall (i, j) \in C_k \ \exists (i', j') \in C_{k-1} \ : \ (i', j')$ is a predecessor of (i, j).*
3. *Let p be the length of a LCS. Then $p = \max\{k \mid C_k \neq \emptyset\}$.*
4. *Let $(i, j), (i', j') \in C_k$. Then $i \leq i' \wedge j \geq j'$ or $i \geq i' \wedge j \leq j'$.*

The lemma states that a successor must have greater rank than each of his predecessors (1.), that each match has at least one *generating* predecessor in the directly preceeding class (2.) and that a match with maximal rank gives the length of a LCS (3.). As a direct consequence of (1.) matches belonging to the same class can not be predecessors (successors) from each other (4.).

So if $(i_1, j_1), (i_2, j_2), \ldots, (i_l, j_l) \in C_k$ are all the matches of rank k we can renumber them in such a way that $i_1 \leq i_2 \leq \ldots \leq i_l$ and $j_1 \geq j_2 \geq \ldots \geq j_l$ holds, i.e. k-matches shift from right to left for increasing rows or, equivalently, from bottom to top for increasing columns (see Figure 1). A lot of the matches in a class C_k are superfluous in the sense that they are not necessary for the generation of matches in the next class C_{k+1}. Let $(i,j),(i',j') \in C_k, (i,j) \neq (i',j')$. We say (i,j) *excludes* (i',j') if $i \leq i' \wedge j \leq j'$. Each successor of (i',j') is also a successor of (i,j) and therefore each match in C_{k+1} that is generated by (i',j') may alternatively be generated by (i,j). Obviously, by a pairwise comparison of matches in C_k we may exclude superfluous matches until the remaining matches $(i_1,j_1),(i_2,j_2),\ldots,(i_l,j_l)$, possibly after a renumbering, satisfy $i_1 < i_2 < \ldots < i_l$ and $j_1 > j_2 > \ldots > j_l$. Each of these k-matches excludes other k-matches located in the same row or column. Therefore they can also be characterized by the following definition.

Definition 2.1 (Minimal Match) *A match $(i,j) \in C_k$ is called a* minimal (dominant) k-match *if*

$$\forall (i',j') \in C_k \;:\; (i' > i \wedge j' \leq j) \text{ or } (i' \leq i \wedge j' > j).$$

That is (i,j) is a minimal element in C_k according to the standard ordering defined on vectors. Let $D_k := \{(i,j) \in C_k \mid (i,j)$ is a minimal match$\}$ be the set of minimal matches of rank k. $D := \bigcup_{k>0} D_k$ and $d := |D|$.

Since $D_k \subseteq C_k,\ k \geq 1$, we have $d \leq r$. The minimal matches constitute the core of matches on which we can focus our attention without being in danger to oversee a LCS, i. e. solving the problem incorrect. In Figure 1 minimal matches are indicated by embolded circles. From the stepwise exclusion of matches we have

Lemma 2.2

1. *$C_k \neq \emptyset \Longrightarrow D_k \neq \emptyset$.*
2. *$\forall (i',j') \in C_k \backslash D_k\ \exists (i,j) \in D_k \;:\; (i,j)$ excludes (i',j').*

Theorem 2.1 *Let p be the length of a LCS between sequences A and B. Then there is a LCS consisting exclusively of minimal matches, i. e. $\exists d_1, d_2, \ldots, d_p$ such that $d_k = (i_k, j_k) \in D_k$, $1 \leq k \leq p$, and d_{k+1} is a successor of d_k, $1 \leq k \leq p-1$.*

Proof: From Lemma 2.1 (3.) we have $C_p \neq \emptyset$ and from Lemma 2.2 (1.) we also have $D_p \neq \emptyset$. So choose any match $d_p \in D_p$. Lemma 2.1 (2.) guarantees that there is a predecessor of d_p in C_{p-1} and Lemma 2.2 (2.) even guarantees one in D_{p-1}. Take this one as d_{p-1}. We iterate this procedure until we have constructed a LCS consisting exclusively of minimal matches. □

Hence, to solve the LCS Problem it is sufficient to determine the minimal matches and their ranks. Again we have cut the set of values $L_{i,j}$ to be computed. At this point we still have no means to determine in an easy way the class to which a particular minimal match belongs.

2.2 Contours

Our goal in this paragraph is to formulate a sufficient condition for a match (i,j) to belong to a certain class C_k. We already have the necessary condition $(i,j),(i',j') \in C_k : i' \geq i \Longrightarrow j' \leq j$ (Lemma 2.1 (4.)). In other words, for increasing prefixes of sequence A ($i' \geq i$) we need monotonously less symbols from sequence B in order to be able to obtain a common subsequence of fixed length k. Assume that we have computed all k-matches up to the i-th row. The question, how many symbols from B are needed to obtain a common subsequence of length k with A_i, i. e. what is the left border of values k in row i of the L-matrix, is answered by a leftmost k-match found so far, i.e. by its second component (column). If we consider this value for each row we obtain the contour of the k-th region. Formally we can define the contours as follows.

Definition 2.2 *Let $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$ sequences. Then for $i = 0, 1, 2, \dots, m$ and $k \geq 0$ define*

$$C_{i,k} := \{(s,t) \in C_k \mid 0 \leq s \leq i\}$$

and

$$T_{i,k} := \begin{cases} \min\{j \mid (i,j) \in C_{i,k}\} & \textit{if } C_{i,k} \neq \emptyset \\ \infty & \textit{if } C_{i,k} = \emptyset \end{cases}$$

In $C_{i,k}$ we collect all k-matches up to the i-th row and then determine the least second component. In particular we have $T_{0,k} = \infty, k > 0$, and $T_{i,0} = 0, 0 \leq i \leq m$.

Apart from the description of the left border of region k in row i $T_{i,k}$ also gives an upper bound for the second component of k-matches (i', j') in rows $i' > i$ (in particular for $i' = i+1$). It is necessary that $j' \leq T_{i,k}$. This implies immediately that $T_{i+1,k} \leq T_{i,k}$ and in general we have the so-called left-shifting property of the contours.

Lemma 2.3 $\infty = T_{0,k} \geq T_{1,k} \geq T_{2,k} \geq \dots \geq T_{m,k}$

Because each k-match must have at least one predecessor in C_{k-1} we have the lower bound $T_{i,k-1} + 1$ for k-matches in row $i+1$ and thus

Lemma 2.4 $T_{i,k-1} < T_{i+1,k} \leq T_{i,k}$

As an easy corollary we see that the left borders of the regions are strictly increasing in each row and that contours of different classes never cross or touch.

Lemma 2.5 $T_{i,0} < T_{i,1} < T_{i,2} < \dots < T_{i,p}$ *where* $T_{i,p'} = \infty \ \forall p' > p$

It is easy to see that the necessary condition $T_{i-1,k-1} < j \leq T_{i-1,k}$ for $(i,j) \in C_k$ is also sufficient. So we have

Theorem 2.2 *A Match (i,j) is in C_k if and only if $T_{i-1,k-1} < j \leq T_{i-1,k}$.*

Having computed the k-matches in row i with the aid of $T_{i-1,k-1}$ and $T_{i-1,k}$ we are able to determine the value of $T_{i,k}$. If there has been no match in the relevant interval at all then $T_{i,k} = T_{i-1,k}$ since $C_{i,k}$ has not changed in comparison to $C_{i-1,k}$. On the other hand, if there are several matches (i,j) in row i with $T_{i-1,k-1} < j \leq T_{i-1,k}$ then, according to the definition, $T_{i,k}$ has to be the minimum of the second components of these new matches. So we can formulate the following recursion as a corollary to Theorem 2.2.

Lemma 2.6 (Recursion)

$$T_{i,k} = \begin{cases} \min\{j \mid a_i = b_j \wedge T_{i-1,k-1} < j \leq T_{i-1,k}\} \\ T_{i-1,k} \textit{ if there is no such } j \end{cases}$$

The determination of $T_{i,k}$ and of k-matches in row i can be done simultaneously. With the help of $T_{i,k}$ we are also able to express a simple necessary and sufficient condition for (i,j) to be a minimal k-match.

Theorem 2.3 *A match (i,j) is minimal in C_k (i. e. $(i,j) \in D_k$) if and only if $T_{i-1,k-1} < j < T_{i-1,k}$ and $j = T_{i,k}$.*

So if $T_{i,k} < T_{i-1,k}$ then $(i, T_{i,k})$ is a minimal match in C_k. The minimal matches are those that are responsible for a change of the border of a region and therefore suffice to specify the contours. Lemma 2.6 allows the successive computation of values $T_{i,k}$ – and thus of minimal matches and their ranks – for more and more increasing i and k until we have found a LCS.

3 The Hunt/Szymanski Approach

According to the recursion formulated in Lemma 2.6 the values $T_{i,k}$ can be computed in three different obvious orders.

1. For a fixed k compute all $T_{i,k}$ for $1 \leq i \leq m$, then advance to $k+1$. This is the approach used by Hirschberg [11] that considers the matches class by class.

2. For a fixed i compute all $T_{i,k}$ for appropriate k, then advance to $i+1$. This is the approach used by Hunt/Szymanski [13] that considers matches row by row in the L-matrix. It is reviewed in the next paragraph.

3. Compute $T_{i,k}$ in a diagonal order, that is use the order $T_{1,1}$, $T_{2,2}$, $T_{3,3}, \ldots$, $T_{2,1}$, $T_{3,2}$, $T_{4,3}, \ldots$, $T_{3,1}$, $T_{4,2}$, $T_{5,3}, \ldots$ This is the approach used by Nakatsu et al. [17].

In addition to the different orders of computation different methods for determining the minimum on the right hand side of the recursion, i.e. finding a minimal match, may be used. We will look at two such methods while using the Hunt/Szymanski order of computation.

3.1 Original method

The algorithm of Hunt/Szymanski proceeds row by row and keeps track of all partial determined classes found so far. Since i is fixed when looking at a particular row the task is to determine efficiently the rank for each match found in this row. Assume that we wish to compute the rank of a match $(i+1, j)$ in row $i+1$. From the consideration of the previous row we are given a strictly increasing list (Lemma 2.5)

$$0 = T_{i,0} < T_{i,1} < T_{i,2} < \ldots < T_{i,k} < T_{i,k+1} = \infty.$$

By finding an x such that $T_{i,x-1} < j \leq T_{i,x}$ we can conclude from Theorem 2.2 that $(i+1, j)$ must have rank x. If, in addition, $j < T_{i,x}$ and there is no $j' > T_{i,x-1}$ with $j' < j$ then $(i+1, j)$ is a minimal x-match.

What is still needed is a way to make available matches in a row quickly. This can be achieved by the introduction of a preprocessing phase that computes for each $\sigma \in \Sigma$ an ordered list of positions in B which contain the symbol σ. This can be done in time $O(|B|)$ or $O(|B| \log s)$ depending on whether the alphabet is known in advance or not. Then, when looking at row $i+1$, we can follow the precomputed list for symbol a_{i+1} to obtain all positions j in B with $a_{i+1} = b_j$.

This is exactly what the algorithm by Hunt/Szymanski does when processing a row; in doing a binary search on the ordered list of values $T_{i,k}$ $\forall k$ the rank for each match $(i+1, j)$ in row $i+1$ is computed. Each time a new k-match $(i+1, j)$ is found, $T_{i+1,k}$ is set to $\min\{j, T_{i+1,k}\}$. Thus, after having considered each match in row $i+1$ the $T_{i+1,k}$, $\forall k$, satisfy Definition 2.2 and we can proceed to the next row.

Instead of creating a new list of values $T_{i,k}$ in each row the algorithm of Hunt/Szymanski simply modifies the values of the preceeding row. So one vector $Thresh$ of size m that holds the current bound for each class, i.e. the column of the leftmost minimal match found so far, is sufficient. This method, therefore, is often viewed as trying to reduce the current threshold value of each class. To reconstruct a LCS we save additional pointers to the last minimal match found for each class C_k which enables us to identify a generating match for each newly found match. This allows the dynamic construction of linked lists representing longest common subsequences.

Algorithm 1 (Hunt/Szymanski [13], 1977)

Input: Sequences $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$, $m \leq n$.

Output: A LCS $C = c_1 c_2 \ldots c_p$.

Method:

(1) $Thresh[0] := 0$;
    for $k := 1$ to m
        do $Thresh[k] := n + 1$ **od**;
    $Link[0] :=$ nil;
(2) **for** $i := 1$ to m **do**
        for all decreasing j with $b_j = a_i$ **do**
(3)   find k such that $Thresh[k-1] < j \leq Thresh[k]$;
            if $j < Thresh[k]$ **then**
                $Thresh[k] := j$;
                $Link[k] :=$ newnode $(i, j, Link[k-1])$;
            fi;
        od;
    od;
(4) $k := m$;
    while $Thresh[k] = n + 1$
        do $k := k - 1$ **od**;
    $PTR := Link[k]$;
    while $PTR \neq$ nil **do**
        $c_k :=$ character corresponding to match pointed to by PTR;
        $k := k - 1$;
        advance PTR;
    od

The correctness follows immediately from Lemma 2.6 by observing the invariant

- $Thresh[k] = T_{i-1,k}$ $\forall k$ at the start of the i-th iteration
- $Thresh[k] = T_{i,k}$ $\forall k$ at the end of the i-th iteration

Note that it is crucial for this invariant to hold that the algorithm scans the list of matches in order of decreasing positions in a row. The two for loops scan all matches between A and B and for each match a binary search is performed on a list of length p at most, where p is the length of a LCS. Steps (1) and (4) need $O(m)$ steps. The processing time therefore is $O(m + r \log p)$. We have to add the time for the preprocessing. In many applications the performance may degrade to $O(mn \log m)$ which is even worse than the running time of the dynamic programming approach. But fortunately this basic approach can be improved.

3.2 Improvement

There are two sources of inefficiency in the original Hunt/Szymanski approach. First we are only interested in computing minimal matches but the algorithm scans all occurring matches. And second a binary search is used for each match considered to identify its rank. The key observation made by Apostolico/Guerra [3] is that what we really need to know to compute a minimal match is the first occurrence of a symbol in B to the right of a certain position. For example, to obtain a minimal match (i, j) in row i it is sufficient to know the first position j to the right of some $T_{i-1,k-1}$ with $b_j = a_i$. Then we have three possible cases.

1. $(j < T_{i-1,k})$: then (i, j) is a minimal k-match.
2. $(j = T_{i-1,k})$: then (i, j) is a k-match, but no minimal k-match.
3. $(j > T_{i-1,k})$: then (i, j) is a (minimal) match in the first class $C_{k'}, k' > k$, with $(T_{i-1,k'} > j)$ $T_{i-1,k'} = j$.

Indeed it is very easy to precompute information about the first occurrence of symbols to the right of a specific position and to improve the algorithm of Hunt/Szymanski considerably in this way.

Definition 3.1 *Given a sequence $B = b_1 b_2 \ldots b_n$ over some alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_s\}$ define the function $CLOSEST$ as follows:*

$$CLOSEST : [\sigma_1, \ldots, \sigma_s] \times [1, \ldots, n+1] \longrightarrow [1, \ldots, n+1]$$

$$CLOSEST[\sigma_i, j] := \begin{cases} n+1 & \textit{if } j = n+1 \\ \min\{\{j' \geq j \mid b_{j'} = \sigma_i\} \cup \{n+1\}\} & \textit{if } j = 1, 2, \ldots, n \end{cases}$$

When s is fixed and small in comparison to n, like for example in the analysis of molecular sequences, this function can be made available in form of a $s \times (n+1)$ matrix which is computed in time $\Theta(sn)$. For great s $(s \rightarrow m)$ time and space complexity $\Theta(sn)$ becomes too expensive. Fortunately it is possible to store the same information in a vector of size $n+1$ that can be computed in time $\Theta(n)$. But then time $O(\log s)$ is needed to retrieve a value $CLOSEST[\sigma_i, j]$ from the compact representation [3]. The Hunt/Szymanski approach can now be restated as

Algorithm 2 (Apostolico/Guerra [3], 1987)

Input: Sequences $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n, m \leq n$.

Output: A LCS $C = c_1 c_2 \ldots c_p$.

Method:

(1) $Thresh[0] := 0$;
  for $k := 1$ **to** m
    do $Thresh[k] := n + 1$ **od**;
(2) **for** $i := 1$ **to** m **do**
    $j := CLOSEST[a_i, 1]$; (* first match in row i *)
    $k := 1$;
(3)   **while** $j \neq n + 1$ **do**
      if $j < Thresh[k]$ **then**
        $temp := Thresh[k]$;
        $Thresh[k] := j$;
(4)   record new minimal match (i, j);
        $j := CLOSEST[a_i, temp + 1]$; (* next potential min. match *)
      fi;
      if $j = Thresh[k]$ **then**
        $j := CLOSEST[a_i, j + 1]$
      fi;
      $k := k + 1$;
    od;
  od;
(5) recover LCS

The correctness follows from the same invariant as before. In each row we start with the first match and the first class. The while-loop (3) increments the class until one of the if-statements is true (Case 3). If the first if-statement is true for some k for the first time then, from the definition of $CLOSEST$, we can conclude that we have found a new minimal k-match (Case 1). Since in one row we can have at most one minimal k-match the next match in this row to be considered must lie to the right of the old value $Thresh[k]$. If the second if-statement is true we deal with Case 2 by considering the next match in the current row.

Because we are now considering a subset of the matches in one row in *increasing* order a little care has to be taken in building up the linked lists of minimal matches for later reconstruction of a LCS. But this is only a technical aspect that does not influence the complexity of the algorithm and we will not elaborate on this.

During one iteration of the for-loop (2) the while-loop (3) will be executed at most $O(p)$ times ($p = |LCS|$) because k is increased by one in each iteration and $Thresh[p+1]$ must be equal to $n+1$. This implies the first if-statement to be true and then j would be set to $n+1$ which in turn causes the while-loop to terminate. The initialization (1) and the recovery of a LCS (5) need time $O(m)$. This gives a total running time of $O(pm)$. Taking into account the preprocessing time of $\Theta(ns)$ we have an $O(ns + pm)$ algorithm. When using the compact representation for $CLOSEST$ the time complexity is $O(n + pm \log s)$. The space complexity is $O(ns + d)$ and $O(n + d)$, respectively, where d is the number of minimal matches.

4 New Algorithms

By employing the function $CLOSEST$ it is possible to determine a minimal match in an interval $[Thresh[k]+1, Thresh[k+1]]$ in time $O(1)$. Therefore it seems to be a good strategy to reduce the total number of these searches, i.e. checks of $Thresh$-values, as far as possible. New minimal matches – the ones we are interested in – are found if and only if a certain $Thresh$-value is actually improved. So we should try to avoid checks that do not improve a $Thresh$-value. Two new approaches following this general strategy are presented subsequently in this section. Subsections 4.1 – 4.4 describe the first approach, Subsection 4.5 describes the second.

4.1 Witnesses for longest common subsequences

The disadvantage of all methods based on the Hunt/Szymanski approach is the fact that a class C_k is never identified as being completely determined. Once a class is recognized not to be empty we have to check in each subsequent row whether the $Thresh$-value corresponding to the class may be improved. For short longest common subsequences this doesn't matter too much because we have to check only a few $Thresh$-values in each row. But as the longest common subsequences are getting longer these algorithms are getting inefficient. We now examine how the basic concept of the Hunt/Szymanski approach can be refined in order to make it applicable for long longest common subsequences ($p \approx m$), too.

Lemma 4.1 *Let $S = s_1 s_2 \dots s_p$ be a LCS of length p between the two sequences $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$. Let $s_k = (i, j) \in C_k$ be the k-th match in the LCS S. Then we have*

1. $k \le i \le k + m - p$
2. $k \le j \le k + n - p$

Proof: 1. Assume we have $i < k$. Then (i, j) can only continue a common subsequence of length at most $k - 2$. But this is a contradiction to our assumption $(i, j) \in C_k$.

Assume now we have $i > k + m - p$. In S s_k has to be followed by $p - k$ further matches which continue $S_k = s_1 \dots s_k$ to form a LCS of length p. In particular we have that the first components (positions in A) of the following matches must strictly increasing. But since $i > k + m - p$ there are only less than $m - [k + m - p] = p - k$ positions i' in A with $i' > i$. Thus S_k can not be continued to form a common subsequence of length p. This contradicts the assumption that s_k is a match in a LCS of length p.

2. analog □

This lemma implies that we do not have to check $Thresh[k]$ in rows $i > k + m - p$ because k-matches possibly found in these rows could never be part of a LCS. So in Algorithm 2 we may cancel the consideration of class C_k after the $(k + m - p)$-th iteration of the for-loop. $Thresh[k]$ is not checked before the k-th iteration of the for-loop and exactly once during one

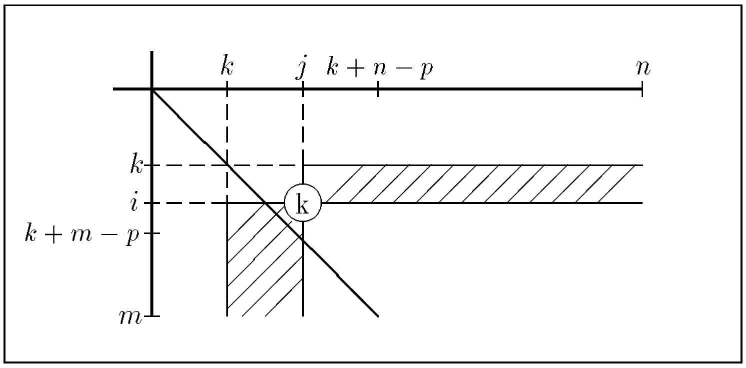

Figure 2: Assuming long common subsequences there are only two small stripes where we have to search for k-matches.

iteration. This means that $Thresh[k]$ is checked at most $(k+m-p)-k+1 = m-p+1$ times. When p is getting big ($p \to m$), i. e. there is a long LCS, $Thresh[k]$ would be considered only during a few iterations. The total number of $Thresh$-values checked – or equivalently the total number of iterations of the while-loop – would then be of order $O(p(m-p)+p)$. Statements outside the while-loop but inside the for-loop are executed m times. This gives a total running time of $O(m+p(m-p))$ which means that the algorithm could be used for both short and long longest common subsequences. But there is one big problem: the length p of a LCS is not known *a priori* (this is the value we want to compute) meaning that $i > k+m-p$ can not be used as a termination criterion for the consideration of class C_k.

Our goal will be to formulate another criterion that does not depend on the knowledge of p but that allows us the identification of classes that need not be considered further. There are two facts from the previous lemma we have not used yet. The first is that for a k-match (i,j) appearing in a LCS of length p we not only have $i \leq k+m-p$ but also $j \leq k+n-p$. Second it makes a statement of existence: if there is a LCS of length p then in *each* class C_k *there has to be at least one* k-match that satisfies the statement of the lemma and which may be extended to a LCS. Such a match is called a *k-witness*. If k-witness (i,j) is not a minimal match than the minimal match (i',j') that excludes (i,j) is also a k-witness, because it satisfies the statement of the lemma and can be extended to a LCS of length p by the same sequence of matches as (i,j). Thus, there is a minimal k-witness in each class.

Observation: Suppose the two input sequences are roughly of the same length ($n \approx m$) and there is a long LCS ($p \to m$). Then the positions of a k-witness (i,j) must be close to k – and therefore close to each other – because of $k \leq i \leq k+m-p$ and $k \leq j \leq k+n-p$. So for each class C_k there has to be a minimal k-witness (i,j) close to the point (k,k). But then all the other matches $(i',j') \in C_k$ must lie to the top right or to the bottom left of this match, i.e. we have either $i' \leq i$ or $j' \leq j$. Of course it must also hold that $i' \geq k$ and $j' \geq k$. Therefore all matches in C_k must lie in the two small stripes indicated by Figure 2. This fact can be exploited to devise a new efficient algorithm. From this observation we also have an upper bound on the number d of minimal matches.

Lemma 4.2 *Let d be the number of minimal matches for two sequences of length n and m that contain a LCS of length p. Then*

$$d \leq p(m + n - 2p + 1).$$

Proof: Let $(i, j) \in C_k$ be a minimal k-witness. For all other minimal k-matches (i', j') we then have either $k \leq i' < i$ or $k \leq j' < j$. This implies that there are at most $(i-k)+(j-k)+1$ minimal matches in class C_k. This value is greatest for $i = k + m - p$ and $j = k + n - p$. Thus from a LCS of length p we have an upper bound on the number of minimal matches in class C_k of $(i - k) + (j - k) + 1 \leq ([k + m - p] - k) + ([k + n - p] - k) + 1 = (n - p) + (m - p) + 1 = n + m - 2p + 1$. Since each of the p classes has a minimal witness the lemma is proven. $\Box$

4.2 A New Order of Computation

Our goal was to find a new criterion to detect that some class need not be considered any further. Obviously this is the case if all matches of a class have been computed. Matches in class C_k have been determined completely if both stripes in Figure 2 have been scanned.

Using the algorithm of Apostolico/Guerra which employs a row-by-row order of computation, only matches located in the horizontal stripe are considered speedily. The vertical stripe will be considered completely only after having scanned the very last row. Minimal matches are computed in order of increasing first components.

Obviously it makes sense to change the order of computation in such a way that matches located in the vertical stripe are determined speedily, too. This can be achieved by computing the matches of a class C_k in order of increasing smaller components, i.e. first we consider k-matches in the first row of the horizontal stripe (row k) and in the first column of the vertical stripe (column k), then those in the second row of the horizontal stripe and the second column of the vertical stripe, etc. Let (i_1, j_1), (i_2, j_2), ..., $(i_l, j_l) \in C_k$ be the minimal k-matches considered in this order. Then we have $k \leq \min\{i_1, j_1\} \leq \min\{i_2, j_2\} \leq \ldots \leq \min\{i_l, j_l\}$.

When determining minimal k-matches in this order of computation, how can we detect the fact that all minimal k-matches have been computed, i.e. the search for further minimal k-matches may be stopped? Obviously this event occurs when, at a certain point of the computation, there is a predecessor among the already computed k-matches for each match that will be considered subsequently implying that none of these matches may be in C_k. In order to specify this point exactly introduce the following definition.

Definition 4.1 *Collect all minimal k-matches in rows $\leq l$ or columns $\leq l$ in*

$$D_{l,k} := \{(i, j) \in D_k \mid \min\{i, j\} \leq l\}.$$

Among the greater components of each of these matches determine the minimum as

$$MinMax_{l,k} := \min\{\max\{i, j\} \mid (i, j) \in D_{l,k}\}$$

and then the smallest l, such that at least one match $(i, j) \in D_{l,k}$ has both components $\leq l$ as

$$l'_k := \min\{l \mid l = MinMax_{l,k}\}.$$

Lemma 4.3 *Let p be the length of a LCS. Then*

1. *there is no minimal k-match (i', j') with $\min\{i', j'\} > l'_k$, i.e. class C_k is completely determined upon the consideration of row and column l'_k;*

2. *$l'_k \leq k + n - p$.*

Proof:

1. From the definition we have at least one minimal k-match $(i, j) \in D_{l'_k, k}$ with $i \leq l'_k$ and $j \leq l'_k$. This match is a predecessor for all matches (i', j') with $\min\{i', j'\} > l'_k$ which can be no k-matches for this reason. But according to the order of computation only such matches are considered subsequently.

2. Would it hold that $l'_k > k + n - p$ there would be no minimal k-match (i, j) with $i \leq k + m - p \leq k + n - p$ and $j \leq k + n - p$, because at least one component of each minimal k-match would be $> k + n - p$. But this would imply that there is no k-witness and therefore no LCS of length p. A contradiction. $\square$

Now we have a sufficient criterion to detect that all matches of a certain class have been computed. This criterion does not depend on p but only on the coordinates of the minimal matches found so far. Knowing that there is a minimal k-witness in each class C_k we have an upper bound on the number of rows and columns to be considered for the determination of all minimal k-matches, namely $(k + n - p) - k + 1 = n - p + 1$. Assuming that a minimal match can be computed in time $O(1)$ in each row or column this gives time $O(n - p + 1)$ to determine the minimal matches of a certain class and time $O(p(n - p) + p)$ to determine all the minimal matches in p classes.

4.3 Dualization

The remaining question is how to compute minimal matches in this new order of increasing smaller components. The idea to solve this problem consists in a kind of dualization of the Hunt/ Szymanski strategy and in particular in the method proposed by Apostolico/Guerra.

One lack of most previously devised algorithms is that the LCS Problem is considered in an unsymmetric way in the sense that only one of the input sequences is analysed in a preprocessing phase. If this analysis is performed for sequence B we are limited to proceed row by row. If instead we compute a table $CLOSEST$ for the occurrences of symbols in A we may proceed in an analogous fashion column by column. All structural properties found in Section 2 hold in a symmetric way; we only changed the roles of the sequences A and B. For example, when proceeding from one column to the next, to find a new minimal k-match we only have to consider positions above the last minimal k-match found in a previous column.

Now, if we precompute tables $CLOSEST\text{-}A$ and $CLOSEST\text{-}B$ for both A and B respectively then we are in fact prepared to proceed row by row as well as column by column. To do this divide the set D of minimal matches into the two disjoint subsets

- $D^a = \{(i, j) \mid (i, j) \in D \wedge i \leq j\}$ (on and above the main diagonal)
- $D^b = \{(i, j) \mid (i, j) \in D \wedge i > j\}$ (below the main diagonal)

and add the pseudo-match $(0, 0)$ to both sets. Minimal matches in D^a will be computed with the aid of table $CLOSEST\text{-}B$ and a row-by-row scan whereas minimal matches in D^b

are now computed in a symmetric fashion with the aid of table $CLOSEST\text{-}A$ and a column by column scan. Intuitively one can imagine to slide down the main diagonal, starting with point (1,1) and ending with point (m, m), considering only points on and to the right of the diagonal in each row and points below the diagonal in each column, thereby devising the new order of computation of the minimal matches and their ranks. Within a certain row or column minimal matches of the different classes are determined by the method proposed by Apostolico/Guerra. This completes the description of the principle of the first new algorithm and the experienced reader should have no problem to set up the code for it. However, to formally prove the correctness of this principle some additional thoughts are necessary. The remainder of this paragraph is therefore devoted to the explicit dualization.

From Section 2 we know that contours of different classes do not intersect or touch. Therefore, when sliding down the diagonal, we will cut at most one contour at a time, i.e. in a certain point on the diagonal. Moreover the classes will be cut in order $C_1, C_2, C_3, \ldots$ implying that there is always a unique class C_{low} whose contour will be cut next. We know that the contour of some class C_k is specified completely through minimal k-matches. Assume now that the contour of class C_k is cut by the diagonal in point (l, l). Then there must be a minimal k-match (i, j) in row l to the left of the diagonal (i.e. $i = l, j < l$) or in column l above (or on) the diagonal (i.e. $i \leq l, j = l$). In the former case the contour is cut within a horizontal section in the later within a vertical section. Note that also both cases may apply and that this is exactly the graphical interpretation of Lemma 4.3 (1.). When determining minimal k-matches successively, according to the new order of computation, a minimal match responsible for a cut in point (l, l) must have been computed when considering a column $< l$ (if it is in row l) or a row $\leq l$ (if it is in column l). From Section 2 we know that minimal k-matches shift from right to left for increasing rows. For reasons of symmetry we now also have that minimal matches shift from bottom to top for increasing columns. Thus we can imagine contours to grow towards each other simultaneously from the bottom left and from the top right when proceeding in the computation. This implies that a minimal k-match responsible for a cut will always be the last one, and therefore leftmost (topmost), found during the consideration of previous rows (columns). So what we have to do is to keep track for each class of the current leftmost (topmost) minimal match in rows (columns).

In rows it is sufficient to store the column of the current leftmost minimal match which, in analogy to Section 2, leads to the following description of contours above the diagonal:

$$row\text{-}T_{i,k} := \min\{t \mid (s,t) \in D^a \cap C_k \wedge 0 \leq s \leq i\} \cup \{\infty\}.$$

Symmetrically it is sufficient in columns to store the row of the current topmost minimal match of each class, leading to the description of contours below the diagonal as:

$$col\text{-}T_{i,k} := \min\{s \mid (s,t) \in D^b \cap C_k \wedge 0 \leq t \leq i\} \cup \{\infty\}.$$

From this definition it is clear that $MinMax_{l,k} = \min\{row\text{-}T_{l,k}, col\text{-}T_{l,k}\}$, i.e. the contour of C_k is cut in point (l, l) for the smallest l such that $col\text{-}T_{l,k} = l$ or $row\text{-}T_{l,k} = l$.

When looking at a specific row and column, say l (i.e. having reached point (l, l) on the diagonal), we have to determine minimal matches not only for one class, but for all appropriate classes. Since classes whose contours have been cut already are determined completely we have only to look for minimal matches in classes C_k, $k \geq low$. From the consideration of previous rows (columns) $l' < l$ we can assume to know the leftmost (topmost) minimal match

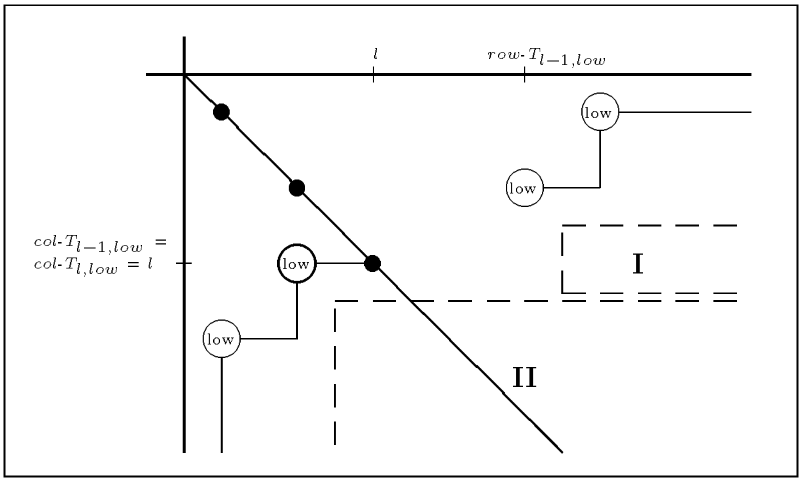

Figure 3: The diagonal cuts a horizontal contour.

of these classes found so far, i.e. $row\text{-}T_{l-1,k}$ ($col\text{-}T_{l-1,k}$), and the class C_{low} which will be completed next. Again from Section 2 we know that these leftmost (topmost) positions of minimal matches increase strictly for increasing classes. Also the current leftmost (topmost) minimal low-match must lie to the right (below) the diagonal, or else the contour of C_{low} would have been cut already.

Thus we have $l \leq row\text{-}T_{l-1,low} < row\text{-}T_{l-1,low+1} < \ldots < row\text{-}T_{l-1,max} < row\text{-}T_{l-1,max+1} = \infty$, where max denotes the highest class for which a minimal match has been found in rows so far, and $l \leq col\text{-}T_{l-1,low} < col\text{-}T_{l-1,low+1} < \ldots < col\text{-}T_{l-1,max} < col\text{-}T_{l-1,max+1} = \infty$.

So to find minimal matches in row l to the right of the diagonal in classes C_k, $k > low$, we have to find the first position strictly to the right of the leftmost minimal $(k-1)$-match found in previous rows ($row\text{-}T_{l-1,k-1}$). If this position improves on the position of the leftmost minimal k-match found so far ($row\text{-}T_{l-1,k}$), then we have identified a new minimal k-match. Remember that this necessary and sufficient condition is used in the algorithm by Apostolico and Guerra in exactly the same way. Again for reasons of symmetry an analogous argument applies for the identification of minimal matches in column l in classes C_k, $k > low$.

In order to determine minimal matches in C_{low} in row and column l a little care has to be taken. First observe that the first match (l, j) in row l to the right of (or on) the diagonal which improves on the current position of the leftmost low-match is guaranteed to be a low-match, because the contour of C_{low-1} has been cut already which implies the existance of a predecessor in C_{low-1} for all matches considered later on and therefore in particular for (l, j). The same argument applies to the first match (i, l) in column l below the diagonal which improves on the current position of the topmost low-match. However, (l, j) ((i, l)) is a minimal low-match if and only if there is no low-match in row l (column l) which lies to the left (top) of this match. But, as we have seen above, this is the case if and only if the diagonal cuts the contour of C_{low} in point (l, l) in a horizontal (vertical) section.

Therefore we first check whether there is a low-match (l, j') to the left of the diagonal in row l, i.e. whether the diagonal cuts the contour of C_{low} in a horizontal section. If such a match exists it must have been computed during the consideration of some column $l' < l$ because in columns we only inspect positions strictly below the diagonal. No minimal low-

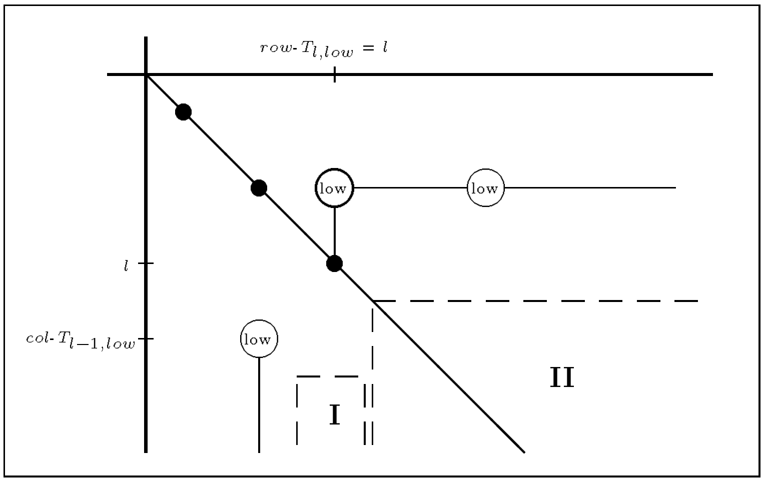

Figure 4: The diagonal cuts a vertical contour.

match in column l might cause a cut with the contour of C_{low} in point (l, l). Formally we have $col\text{-}T_{l,low} = l \Longleftrightarrow col\text{-}T_{l-1,low} = l$. Thus we don't have to know minimal matches in column l in order to be able to decide safely whether a low-match to the left of the diagonal exists in row l. In case we find such a match, i.e. $col\text{-}T_{l-1,low} = l$, we know two things. First the minimal matches in C_{low} have been determined completely and (l, j') is a predecessor for all matches in region II of Figure 3, i.e. for all matches to be considered subsequently. Second there is no minimal low-match in row l to the right of the diagonal. Therefore we can start directly to search minimal matches in classes $C_k,\ k > low$, in row l to the right of the current leftmost minimal low-match (i.e. $j > row\text{-}T_{l-1,low}$; region I in Figure 3). Otherwise we search for minimal matches in classes $C_k,\ k \geq low$, in row l to the right of (and on) the diagonal (i.e. $j \geq l$). In any case we can compute exactly the minimal matches in classes $C_k,\ k \geq low$, in row l.

Having done this we can check whether there is a low-match (i', l) to the top of (or on) the diagonal in column l ($row\text{-}T_{l,low} = l$), i.e. whether the diagonal cuts the contour of C_{low} in a vertical section. Symmetric implications follow if this is the case (see Figure 4) and we are thus able to compute exactly the minimal matches in classes $C_k,\ k \geq low$, in column l.

4.4 The $O(p(n-p))$ algorithm

The algorithm implementing the dualization now uses two vectors $row\text{-}Thresh$ and $col\text{-}Thresh$ of size m; $row\text{-}Thresh[k]$ is used to hold the current leftmost minimal k-match (i.e. its column) and $col\text{-}Thresh[k]$ to hold the current topmost minimal k-match (i.e. its row). The variable low is used to hold the class which will be completed next.

Algorithm 3

Input: Sequences $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n, m \leq n$.

Output: A LCS $C = c_1 c_2 \ldots c_p$.

Method:

```
(0)  Preprocessing:
     Compute CLOSEST-A and CLOSEST-B
(1)  Initialization:
     row-Thresh[0] := 0; col-Thresh[0] := 0;
     for k := 1 to m
         do col-Thresh[k] := m + 1; row-Thresh[k] := n + 1; od;
     low := 1;
(2)  for l := 1 to m do                                   (* slide down diagonal *)
(3)      if col-Thresh[low] = l then                      (* check for cut with contour *)
             j := CLOSEST-B[a_l, row-Thresh[low] + 1];
             low := low + 1;
         else
             j := CLOSEST-B[a_l, l];
         fi;
         k := low;
(4)      while j ≠ n + 1 do                               (* find min. matches in row l *)
             if j < row-Thresh[k] then
                 temp := row-Thresh[k];
                 row-Thresh[k] := j;
                 record new minimal match (l, j);
                 j := CLOSEST-B[a_l, temp + 1];
             fi;
             if j = row-Thresh[k] then j := CLOSEST-B[a_l, j + 1] fi;
             k := k + 1;
         od;
(5)      if row-Thresh[low] = l then                      (* check for cut with contour *)
             i := CLOSEST-A[b_l, col-Thresh[low] + 1];
             low := low + 1;
         else
             i := CLOSEST-A[b_l, l + 1];
         fi;
         k := low;
(6)      while i ≠ m + 1 do                               (* find min. matches in column l *)
             if i < col-Thresh[k] then
                 temp := col-Thresh[k];
                 col-Thresh[k] := i;
                 record new minimal match (i, l);
                 i := CLOSEST-A[b_l, temp + 1];
             fi;
             if i = col-Thresh[k] then i := CLOSEST-A[b_l, i + 1] fi;
             k := k + 1;
         od;
     od;
(7)  recover LCS
```

Theorem 4.1 *Algorithm 3 is correct, i. e. it computes exactly the minimal matches and their ranks and therefore a longest common subsequence.*

Proof: Again the correctness follows from the invariant

- $col\text{-}Thresh[k] = col\text{-}T_{l-1,k}\ \forall k \geq 0$
- $row\text{-}Thresh[k] = row\text{-}T_{l-1,k}\ \forall k \geq 0$

at the beginning of the the l-th iteration of the for-loop which is proved through an inductive argument. From the initialization it holds at the beginning of the first iteration and is then restored by the operations within the for-loop for the next iteration. The correctness of these operations is immediate from the discussion preceding the algorithm.

The algorithm terminates, because the for-loop is executed a fixed number of times and the two while-loops terminate for the same reason that was given for the termination of the algorithm by Apostolico/Guerra. □

Remark: For reasons of easy survey the linkage of minimal matches for later reconstruction of a LCS was not included in Algorithm 3. It can be done in much the same way as before except that now two vectors *row-Link* and *col-Link* are employed for minimal matches in D^a and D^b respectively. However, when we detect a cut of the diagonal with some contour the unique minimal match $(i,j) \in C_{low}$ responsible for the cut (embolded circle in Figures 3 or 4) is a generating predecessor for succeeding matches on, above and below the diagonal (region II). Therefore we have to update also *col-Link*[*low*] or *row-Link*[*low*] accordingly depending on whether (i,j) was in D^a or in D^b. But this is only a technical detail that does not influence the correctness or complexity of Algorithm 3. The reader is encouraged to add the appropriate statements to the algorithm, but beware of not overwriting a value that may still be needed!

Theorem 4.2 *Algorithm 3 has space complexity $O(ns + d)$ and time complexity $O(ns + \min\{pm, p(n-p)\})$, where s denotes the alphabet size, d the number of minimal matches and p the length of a LCS.*

Proof: Tables *CLOSEST-A* and *CLOSEST-B* need space $O(ns)$ and vectors *col-Thresh*, *row-Thresh*, *col-Link*, *col-Link* need space $O(m)$. One node is created for each of the d minimal matches. This gives a total space complexity of $O(ns + d)$.

Let p be the length of a LCS and assume $m \leq n$. Preprocessing (0) takes time $O(ns)$, initialization (1) and reconstruction of a LCS take time $O(n)$. All statements of the for-loop, with the exception of the two while-loops, take time $O(1)$ giving a total of $O(m)$ for all iterations. We now give two upper bounds for the total number of iterations of the two while-loops.

1. Within one of the m iterations of the for-loop each while-loop is executed at most $O(p)$ times. This is the same bound we obtained for Algorithm 2. Thus the while-loop will be executed $O(pm)$ times.

2. We obtain a second bound by counting the number of iterations of the for-loop during which *row-Thresh*[k] and *col-Thresh*[k] are checked for a fixed k. In one iteration of the for-loop each value is checked exactly once; *row-Thresh*[k] and *col-Thresh*[k] are not checked before the k-th iteration of the for-loop. From Lemma 4.1 we know that there has to be at least one minimal k-match (i,j) with $i \leq k + m - p$ and $j \leq k + n - p$.

Assume $k+n-p \leq m$. Then the diagonal cuts the contour of C_k at the latest in iteration $k+n-p$ (Lemma 4.3 (2.)). In subsequent iterations *row-Thresh*$[k]$ and *col-Thresh*$[k]$ are not checked any more implying that they are checked at most $n-p+1$ times each.

Assume $k+n-p > m$. Then the diagonal possibly will not cut the contour of C_k, because the for-loop is executed at most m times. But in this case *row-Thresh*$[k]$ and *col-Thresh*$[k]$ are checked at most $m-k$ times. From $k+n-p > m \iff k > m-(n-p)$ we have $m-k < m-(m-n+p) = n-p$.

Thus, for each of the p classes the corresponding values *row-Thresh*$[k]$ and *col-Thresh*$[k]$ are checked at most $n-p+1$ times, i. e. the while-loops are executed $O(p(n-p)+p)$ times. The time complexity of Algorithm 3 can now be expressed as claimed as $O(ns+\min\{pm, p(n-p)\})$.
□

This running time suggests that in cases when $m \approx n$ Algorithm 3 will be efficient for both short and long longest common subsequences. Moreover the running time increases more smoothly when leaving these two special cases and approaching longest common subsequences of intermediate length. The main advantage of this algorithm is that it makes superfluous an *a priori* decision for one of the specialized algorithms and therefore can be used more flexible. One typical application might be the search of data bases for records that are similar to a query. During the search process it is probable to encounter very different stored records and therefore we compare both records that are similar to the query and records that differ from it considerably.

From the upper bound on the number d of minimal matches ($d \leq p(m+n-2p+1)$) proved in Lemma 4.2 we can conclude that the algorithm presented by Chin/Poon [5] which has time complexity $O(ns+\min\{ds, pm\})$ should also exhibit a good performance in both situations. However, the alphabet size has to be reasonably small for this algorithm. The next paragraph introduces a new algorithm exhibiting exactly the same running time but using a rather different algorithmic realization.

4.5 The $O(ds)$ algorithm

The introduction of a symmetric concept for the Hunt/Szymanski strategy allowed the avoidance of many unnecessary checks of *Thresh*-values, i.e. current left borders of the individual classes. It is also possible to reduce the number of checks of *Thresh*-values without preprocessing the second sequence. The main idea is not only to store the current *Thresh*-value for each class, i. e. the column of the last (leftmost) minimal match found in each class, but to record the row of this minimal match responsible for the current value, too. This is useful because of the following observation (see Figure 5): Consider two rows i_1 and i_2, $i_1 < i_2$, such that the corresponding positions in A contain the same symbol σ ($a_{i_1} = \sigma = a_{i_2}$) and there is no position between i_1 and i_2 that contains symbol σ ($a_{i'} \neq \sigma, i_1 < i' < i2$). From the calculation of minimal matches in row i_1 we can conclude that certain positions in B can not contain symbol σ, namely positions in intervals whose upper bound is given by the border of some class C_{k+1} in row i_1 ($T_{i_1,k+1}$) and whose lower bound is given by the border of class C_k in row i_1-1 ($T_{i_1-1,k}$). This is because we search for minimal $(k+1)$-matches in row i_1 to the right of the border of class C_k in row i_1-1 and a minimal $(k+1)$-match in row i_1 gives the border of C_{k+1} in row i_1. Since a minimal match always occurs at the first position matching the symbol of a row there can be no matching position in the interval $[T_{i_1-1,k}+1$, $T_{i_1,k+1}-1]$. Now, if the lower bound, i.e. the border of C_k in row i_1-1, does not change up to row $i_2 - 1$, then we can be sure that there is no position matching the symbol of row i_2, which is the same as in row i_1, at positions below the border of class C_{k+1} in row $i_2 - 1$ ($T_{i_2-1,k+1}$), which is less than this border in row i_1, and above the unchanged border of C_k in row $i_2 - 1$. This implies that the border of C_{k+1} cannot change in row i_2 and hence remains the same as in row $i_2 - 1$. This observation can be stated formally as

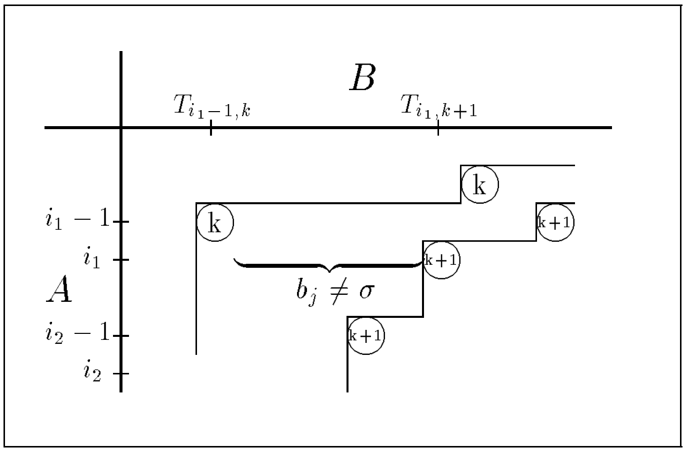

Figure 5: The border of C_{k+1} cannot change in row i_2.

Lemma 4.4 *Let $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$, $n \geq m$, be the two input sequences. Let i_1 and i_2, $i_1 < i_2$, be two positions in A such that $a_{i_1} = a_{i_2}$ and $a_i \neq a_{i_1}$, $\forall i_1 < i < i_2$. Then we have for row i_2:*

$$T_{i_2-1,k} = T_{i_1-1,k} \Longrightarrow T_{i_2,k+1} = T_{i_2-1,k+1}, \;\; k \geq 1$$

The consequence from this lemma is that some value $Thresh[k+1]$ may possibly change in row i_2 only if $Thresh[k]$ has been reduced in at least one of the rows $i_1, i_1+1, \ldots, i_2-1$. Our goal is to select a suitable data structure that enables us to avoid checks of $Thresh$-values that necessarily do not change, i.e. to select only those $Thresh$-values that might change in the given row. To do this we have to remember in which row the individual $Thresh$-values changed for the last time. As mentioned above this can be achieved by associating a value $Thresh[k]$ to the unique row which contains the minimal k-match last found, i. e. the match that caused $Thresh[k]$ to take on its current value. If we improve $Thresh[k]$ in row i then $Thresh[k]$ will remain associated with row i until it will be reduced again in some row $i' > i$.

When considering row i_2, in order to have an efficient access to those $Thresh$-values modified in previous rows a linked list is employed that contains $Thresh$-values in decreasing order of rows to which they are associated. To keep this list up to date we have to delete those $Thresh$-values that have been reduced in the current row at their former position and insert them at the beginning of the list. Therefore a doubly linked list will be used. We still need a random access to particular $Thresh$-values, too. These two requirements can be met by a simple augmentation of the elements of the vector $Thresh$ introducing fields that allow the building of the list structure and the association to a row. To be concrete, an element of the vector $Thresh$ will have the following components:

- *key* : stores the actual *Thresh*-value
- *nkey* : temporary storage for new *Thresh*-value
- *col* : contains the row to which this *Thresh*-value is associated
- *next* : pointer to next element in doubly linked list of *Thresh*-values
- *prev* : pointer to previous element in doubly linked list

If in row i_2 we would like to determine those *Thresh*-values that could possibly be reduced we can simply scan the list from the beginning as long as we encounter *Thresh*-values $Thresh[k_1]$, $Thresh[k_2]$, ..., $Thresh[k_l]$ associated with rows $i_2 - 1$, $i_2 - 2$, ..., i_1, where i_1 is the greatest preceding row of i_2 with $a_{i_1} = a_{i_2}$. Then $Thresh[k_1 + 1]$, $Thresh[k_2 + 1]$, ..., $Thresh[k_l + 1]$ are those values that may possibly be improved. All other *Thresh*-values in the current row need not be checked. In order to be able to check how long we have to scan the list, i.e. when we have found row i_1, we can use a vector *Lastocc* of size $|\Sigma|$ to record the last row that contained a particular symbol.

Algorithm 4

Input: Sequences $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n, m \leq n$.

Output: The length of a LCS.

Method:

```
(0)   Preprocessing:
      Compute CLOSEST
(1)   Initialization:
      for i := 0 to m do
          Thresh[i].next := i − 1; Thresh[i].prev := i + 1;
          Thresh[i].key := n + 1; Thresh[i].nkey := n + 1;
          Thresh[i].col := −2;
      od;
      Thresh[0].next := m; Thresh[m].prev := 0;
      Thresh[0].key := 0; Thresh[0].nkey := 0;
      Thresh[0].col := 0;
      for i := 1 to s do Lastocc[σ_i] := −1;
      max := 0; (* length of common subsequence found so far *)
      head := 0; (* points to Thresh-record last modified *)
(2)   for i := 1 to m do
          q := head;
          p := 1;
(3)       while Thresh[q].col ≥ Lastocc[a_i] do          (* scan list *)
              j := CLOSEST[a_i, Thresh[q].key + 1];
              if j < Thresh[q + 1].key then               (* new minimal match *)
                  Thresh[q + 1].nkey := j;
                  Change[p] := q + 1;
                  p := p + 1;
              fi;
              q := Thresh[q].next;
          od;
```

```
(4)      for j := 1 to p − 1 do                          (* update list *)
            q := Change[j];
            if head ≠ q then
(5)            Thresh[Thresh[q].next].prev := Thresh[q].prev;
               Thresh[Thresh[q].prev].next := Thresh[q].next;
(6)            Thresh[q].next := head;
               Thresh[q].prev := Thresh[head].prev;
               Thresh[Thresh[head].prev].next := q;
               Thresh[head].prev := q;
               head := q;
            fi;
(7)         Thresh[q].key := Thresh[q].nkey;
            Thresh[q].col := i;
         od;
         if Thresh[max + 1].key < n + 1 then max := max + 1;
         Lastocc[σ_i] := i;
      od;
(8)   print(max)
```

Theorem 4.3 *Algorithm 4 is correct, i. e. it computes exactly the minimal matches and their ranks and therefore the length of a LCS.*

Proof: The correctness of Algorithm 4 again follows from the invariant

- $Thresh[k].key = T_{i-1,k}\ \forall k$ at the beginning of the i-ten iteration
- $Thresh[k].key = T_{i,k}\ \forall k$ at the end of the i-ten iteration

From the initialization (1), which also builds up a doubly linked list, the invariant holds at the beginning of the first iteration. Because of Lemma 4.4 it is correct for the the while-loop (3) to check only those *Thresh*-values that might change in the current row. This is accomplished by scanning the list of *Thresh*-records as long as they are associated with rows $\geq$ the last occurrence of a_i. If it is observed that a value $Thresh[k+1].key$ may be reduced a new minimal match is found but we are not allowed to overwrite this value, because $Thresh[k+1]$ may occur backwards in the list and then the old value is needed. For the same reason $Thresh[k+1]$ must not be deleted at its current position in the list and inserted at the beginning. The vector *Change* is used to remember those records that have to be updated in a second step that is performed by the for-loop (4). This for-loop deletes a *Thresh*-record at its former position (5) and inserts it at the beginning of the list (6). Finally the *Thresh*-value is updated and associated with the current row (7). This restores the invariant. □

In order to reconstruct an actual LCS we can use a vector *Link* in the same way as in previous algorithms to store pointers to minimal matches. For each minimal match a new node is created dynamically. When updating this vector we have to proceed in two steps for the same reasons mentioned in the above proof of correctness. We now address the complexity of Algorithm 4.

Definition 4.2 (reachable) *Let $A = a_1 a_2 \ldots a_n$ be a sequence over some alphabet Σ. We say a_i is* reachable *from some symbol a_j, $j > i$, if there is no i' such that $i < i' < j$ and $a_{i'} = a_j$.*

Lemma 4.5 *Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}$ be an alphabet of size s and let $A = a_1 a_2 \ldots a_n$ be a sequence over Σ. Then a symbol a_i is reachable from at most s positions $j > i$.*

Proof: The occurrence of symbol $\sigma_t \in \Sigma$ at position $i' > i$ prevents that a_i is reachable from positions $j > i'$ where $a_j = a_{i'}$. Therefore for each $\sigma_t \in \Sigma$ a_i is reachable from at most one position $j > i$, that is from at most s different positions. □

Lemma 4.6 *A value $Thresh[k]$ associated with a specific row is visited at most s times when scanning the list of $Thresh$-records (in while-loop (3)).*

Proof: This is a simple corollary to the previous lemma, because in each row the while-loop scans the list of $Thresh$-records only up to the last occurrence of the symbol of the current row. □

Theorem 4.4 *Algorithm 4 has space complexity $O(ns + d)$ and time complexity $O(ns + \min\{ds, pm\})$.*

Proof: The table $CLOSEST$ needs space $O(ns)$ and vectors $Thresh$ and $Change$ need space $O(m)$. Vector *Lastocc* needs space $O(s)$ and nodes created for minimal matches need space $O(d)$. This gives a total space complexity of $O(ns + d)$.

Again time $O(ns)$ is needed for the preprocessing phase. Initialization (1) and the for-loop (2) take time $O(m)$, not taking into account the inner while-loop (3), and the inner for-loop (4). The for-loop is executed less times than the while-loop. To get an upper bound on the iterations of the while-loop observe that $Thresh$-values associated with different rows correspond exactly to the d minimal matches. Since by the previous lemma each of these $Thesh$-values is visited at most s times, while associated with a certain row, the while-loop will be executed $O(ds)$ times giving a total running time of $O(ns + ds)$.

Like in the previous algorithms we also have that during one iteration of the outer for-loop the while-loop is executed at most p times where p is the length of a LCS. Thus we have time complexity $O(ns + \min\{ds, pm\})$ for Algorithm 4. □

Remark: It is not by chance that the time bound of Algorithm 4 coincides exactly with the time bound given by Chin/Poon [5] for their algorithm. Indeed both algorithms check exactly the same $Thresh$-values. But they differ considerably with respect to the order in which this is done and to the strategy that is used to determine the $Thresh$-values that have to be checked. While Chin/Poon, in analogy to Hirschberg [11], proceed class by class and construct minimal matches in C_{k+1} from matches in C_k Algorithm 4 proceeds row by row in a Hunt/Szymanski strategy looking back to some of the previously determined minimal matches. Moreover the two algorithms use very different data structures. Chin/Poon compute necessary information in a preprocessing phase for both sequences storing it in two tables whereas Algorithm 4 uses a doubly linked list that makes available essentially the same information stored in the tables dynamically during the main processing phase. It is emphasized that this does not constitute a drawback in this particular case.

Remark: The concepts of the two new methods introduced in this paper can be viewed as orthogonal. The second concept provided for a method that allows to check only those $Thresh$-values that have a chance to change. This method can be used for both vectors of $Thresh$-values used by the first algorithm thereby combining the two approaches. It should be emphasized that this combination is not possible with the Chin/Poon algorithm.

5 Experimental Results

In order to validate the efficiency of the new algorithms some experiments were performed that compared them to previous methods. Time bounds for algorithms using the paradigm of computing minimal matches are obtained, at least for the main processing phase, by giving an upper bound on the number of $Thresh$-values checked, a value comparable to the number of comparisons in sorting algorithms. Therefore this number constitutes a good criterion for the comparison of the efficiency of these algorithms that is independent of the actual implementation. Six algorithms have been investigated with respect to this characteristic value, namely

1. A revised version of the Hunt/Szymanski algorithm developed by Apostolico/Guerra [3] that has running time $O(ns+pm)$. This algorithm is efficient for short longest common subsequences.
2. The algorithm by Nakatsu et al. [17] which has running time $O(n(m-p))$ and thus should be preferred if long longest common subsequences are expected.
3. The algorithm by Chin/Poon [5] which has a time complexity of $O(ns+\min\{ds, pm\})$.
4. Algorithm 4 which has the same time complexity of $O(ns+\min\{ds, pm\})$.
5. Algorithm 3 which has time complexity $O(ns+\min\{pm, p(n-p)\})$.
6. An algorithm that combines the new concepts of Algorithms 3 and 4.

Remark: When implementing the algorithm by Nakatsu et al. it was observed that a time bound of $O(ns+p(m-p))$ can be obtained by the use of the table $CLOSEST$. A modified version was used for the comparison accordingly.

All algorithms were tested for longest common subsequences of short, long and intermediate length. Sample sequences of length 4000 over an alphabet of size 16 were generated randomly (each symbol equally likely) as follows.

Short LCS: In order to obtain pairs of sequences that contain only a short LCS first the two sequences were generated over two disjoint alphabets of size 8 which implies a LCS of length zero. To get longest common subsequences of greater length new symbols were drawn from the whole alphabet of size 16 for more and more randomly chosen positions in both sequences.

Long LCS: Identical random sequences for A and B were generated first to get a LCS of length 4000. The length of a LCS between these two sequences was then reduced by redrawing symbols for randomly chosen positions in both sequences.

In both cases redrawing of more and more symbols leads to longest common subsequences of intermediate length. Results in Table 2 were obtained as the mean over 100 pairs of input sequences. Only the critical time complexities of the algorithms are given in the the headlines.

It can be seen clearly that the new $O(p(n-p))$ algorithm checks much fewer $Thresh$-values than the first two algorithms by Apostolico et al. and Nakatsu et al. Only the $O(ds)$ algorithms are superior for short longest common subsequences for quiet a long time. Notice the very close approximation of minimal matches by the $O(p(n-p))$ algorithm for long longest common subsequences. There are almost no superfluous checks of $Thresh$-values in this case.

$\|LCS\|$	minimal matches	Apostolico/ Guerra $O(pm)$	Nakatsu et al. $O(p(m-p))$	Chin/Poon $O(ds)$	New $O(ds)$	New $O(p(n-p))$	Combined New $O(p(n-p))$
50	1,435	120,679	147,361	9,638	9,638	60,112	16,904
97	4,989	262,925	284,614	32,499	32,499	141,838	37,979
195	17,602	543,821	549,565	109,935	109,935	307,136	107,627
378	53,424	1,046,973	995,970	313,505	313,505	579,587	278,950
684	133,360	1,816,673	1,598,142	718,009	718,009	906,427	573,656
1,034	257,687	2,692,127	2,166,799	1,277,350	1,277,350	1,255,101	931,968
1,352	401,620	3,475,230	2,567,138	1,868,036	1,868,036	1,569,218	1,288,738
1,572	513,599	4,024,663	2,792,765	2,293,083	2,293,083	1,777,573	1,539,632
1,663	506,354	4,094,862	2,715,082	2,300,281	2,300,281	1,532,089	1,367,447
2,161	423,878	4,748,025	2,414,038	2,236,754	2,236,754	859,494	814,394
2,830	282,017	5,867,671	1,861,615	1,819,923	1,819,923	410,560	402,579
3,337	165,328	6,772,005	1,203,500	1,224,247	1,224,247	200,006	199,103
3,648	90,828	7,344,405	689,941	735,664	735,664	99,845	100,167
3,818	48,939	7,663,327	373,510	424,754	424,754	51,206	51,774
3,907	26,783	7,830,310	197,717	250,206	250,206	27,353	28,148

Table 2: Frequency of checks of $Thresh$-values for six different algorithms. $|A| = 4000$, $|B| = 4000$, $|\Sigma| = 16$.

The algorithm combining the two new concepts constitutes a compromise in most cases and performs the fewest checks of all in intermediate situations.

How this extends to the actual running time depends very much on the implementation. All algorithms were implemented in a straight forward manner without any special tuning efforts. The experiments show that the $O(p(n-p))$ algorithm is very efficient with regard to the running time, too.

One goal, among others, was to develop an algorithm that is efficient for long longest common subsequences while retaining the efficiency for short longest common subsequences. It seems that the most efficient algorithms specialized on long longest common subsequences are those based on the paradigm of computing shortest paths, e. g. the algorithms by Myers [16] and by Wu et al. [23] the latter of which was also included in the comparison of running times. The results are shown in Table 3.

$\|LCS\|$	Apostolico/ Guerra $O(pm)$	Nakatsu et al. $O(p(m-p))$	Chin/Poon $O(ds)$	New $O(ds)$	New $O(p(n-p))$	Combined New $O(p(n-p))$	Wu et al. $O(n(m-p))$
50	644	1,350	1,044	371	745	685	123,388
97	1,074	2,350	1,229	550	1,011	849	120,120
195	1,945	4,293	1,875	1,187	1,554	1,417	114,183
378	3,539	7,604	3,637	2,910	2,505	2,892	103,586
684	6,067	12,159	7,267	6,527	3,770	5,680	87,605
1,034	9,084	16,598	12,447	11,441	5,285	9,456	70,496
1,352	11,920	19,839	18,043	16,903	6,738	13,511	56,835
1,572	14,058	21,702	22,111	21,282	7,771	16,471	48,094
1,663	14,163	21,095	22,069	21,487	6,908	15,342	45,383
2,161	15,841	18,731	20,607	20,578	4,679	10,933	28,329
2,830	18,668	14,476	16,071	16,007	2,791	6,641	11,482
3,337	20,947	9,451	10,836	10,681	1,737	3,915	3,726
3,648	22,386	5,533	6,827	6,389	1,164	2,358	1,074
3,818	23,187	3,117	4,344	3,646	865	1,520	309
3,907	23,606	1,774	2,969	2,163	708	1,091	100

Table 3: Running times for seven algorithms in milliseconds. $|A| = 4000$, $|B| = 4000$, $|\Sigma| = 16$.

Again it can be seen that the new $O(p(n-p))$ algorithm is superior to the first two algorithms. The algorithm of Wu et al. proves to be very specialized. It is extremely fast for very long longest common subsequences but its running time increases dramatically for shorter ones. The new $O(ds)$ algorithm compares favorably with the Chin/Poon algorithm. Probably this is due to the preprocessing of the second sequence performed by the latter whereas the former makes available dynamically only the information actually needed. When comparing $O(ds)$ algorithms to the $O(p(n-p))$ algorithm we find the same as above.

In particular the $O(p(n-p))$ algorithm does not suffer from a degeneration of the extent that is inherent in the specialized algorithms. Especially the slight losses in extreme cases, i.e. for very long or very short longest common subsequences, are more than compensated by the gain achieved in intermediate cases for most applications. Thus it can be used more flexible. When longest common subsequences of intermediate length are expected, it is clearly the best algorithm of all.

6 Conclusion

The problem of developing algorithms for the LCS Problem that make superfluous a *a-priori*-decision for a specialized method, i. e. one for short or one for long longest common subsequences, has been investigated. Two new algorithms have been developed which should solve this problem for a wide range of applications. Both methods built on the Hunt/Szymanski strategy of computing minimal matches.

It was observed that avoiding checks of $Thresh$-values that do not result in an actual improvement of that value, i.e. do not lead to new minimal matches, is the key to speed up the computations. Two different approaches achieved this goal.

The first approach uses a new order of computation for minimal matches which allowed the formulation of a simple and sufficient criterion to detect that all minimal matches of a class have been determined thereby saving a lot of unnecessary computational steps. This new order of computation was implemented through a kind of dualization of the refinement of the Hunt/Szymanski approach by Apostolico/Guerra. The time bound obtained for the resulting algorithm is $O(ns + \min\{pm, p(n-p)\})$.

An upper bound on the number d of minimal matches ($d \leq p(m+n-2p+1)$), a key value for the LCS Problem, implied that the algorithm devised by Chin/Poon [5] which has running time $O(ns+\min\{pm, ds\})$ should be efficient for both long and short longest common subsequences, too. From a simple observation a second new algorithm was developed that exhibits the same $O(ns + \min\{pm, ds\})$ time bound through identifying $Thresh$-values that necessarily cannot change in a specific row. In fact both algorithms compute exactly the same information but in a rather different manner.

Empirical experiments proved the efficiency of the new algorithms which compete well with previous methods. Moreover they are not limited for use in special cases and thus solve the initial problem for most applications. Because of its high efficiency for longest common subsequences of intermediate length the first new algorithm seems especially fitted for the analysis of macromolecular sequences. Finally it was noticed that the two new methods may be combined because of their orthogonal concepts leading to some kind of compromise.

Acknowledgement: I would like to thank Prof. Dr. N. Blum for drawing my attention to the LCS Problem.

References

[1] A. V. Aho, D. S. Hirschberg, J. D. Ullman: *Bounds on the Complexity of the Longest Common Subsequence Problem*, Journal ACM, Vol. **23**, Jan. 1976, 1–12.

[2] A. Apostolico: *Improving the Worst-Case Performance of the Hunt-Szymanski Strategy for the Longest Common Subsequence of Two Strings*, Information Processing Letters **23**, 1986, 63–69.

[3] A. Apostolico, C. Guerra: *The Longest Common Subsequence Problem Revisited*, Algorithmica, 1987, 315–336.

[4] A. Apostolico, S. Browne, C. Guerra: *Fast linear-space computations of longest common subsequences*, Theoretical Computer Science **92**, 1992, 3–17.

[5] F. Y. L. Chin, C. K. Poon: *A Fast Algorithm for Computing Longest Common Subsequences of Small Alphabet Size*, Journal of Information Processing, Vol. **13**, No. 4, 1990, 463–469.

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

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest: *Introduction to Algorithms*, MIT Press / Mc Graw-Hill, 1990.

[8] D. Eppstein, Z. Galil, R. Giancarlo, G. F. Italiano: *Sparse Dynamic Programming I: Linear Cost Functions*, Journal of the ACM, Vol. **39**, No. 3, 1992, 519–545.

[9] D. S. Hirschberg: *A Linear Space Algorithm for Computing Maximal Common Subsequences*, Comm. ACM, Vol. **18**, June 1975, 341–343.

[10] D. S. Hirschberg: *An Information-Theoretic Lower Bound for the Longest Common Subsequence Problem*, Information Processing Letters, Vol. **7**, Jan. 1977, 40–41.

[11] D. S. Hirschberg: *Algorithms for the Longest Common Subsequence Problem*, Journal ACM, Vol. **24**, Oct. 1977, 664–675.

[12] W. J. Hsu, M. W. Du: *New Algorithms for the LCS Problem*, Journal of Computer and System Sciences **29**, 1984, 133–152.

[13] J. W. Hunt, T. G. Szymanski: *A Fast Algorithm for Computing Longest Common Subsequences*, Comm. ACM, Vol. **20**, May 1977, 350–353.

[14] S. K. Kumar, C. P. Rangan: *A Linear Space Algorithm for the LCS Problem*, Acta Informatica **24**, 1987, 353–363.

[15] W. J. Masek, M. S. Paterson: *A Faster Algorithm Computing String Edit Distances*, Journal of Computer and System Sciences **20**, 1980, 18–31.

[16] E. W. Myers: *An $O(ND)$ Difference Algorithm and Its Variations*, Algorithmica, 1986, 251–266.

[17] N. Nakatsu, Y. Kambayashi, S. Yajima: *A Longest Common Subsequence Algorithm Suitable for Similar Text Strings*, Acta Informatica **18**, 1982, 171–179.

[18] M. Paterson, V. Dancik: *Longest Common Subsequences*, Proceedings of the 19th Intern. Symp. on Mathematical Foundations of Computer Science, Vol. **841** of LNCS, 1994, 127–142.

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

[20] E. Ukkonen: *Algorithms for Approximate String Matching*, Inform. and Control **64**, 1985, 100–118.

[21] R. A. Wagner, M. J. Fischer: *The String to String Correction Problem*, Journal ACM, **21**, 1974, 168–173.

[22] C. K. Wong, A. K. Chandra: *Bounds For the String Editing Problem*, Journal ACM, Vol. **23**, Jan. 1976, 13–16.

[23] S. Wu, U. Manber, G. Myers, W. Miller: *An $O(NP)$ Sequence Comparison Algorithm*, Information Processing Letters **35**, 1990, 317–323.