Information Processing Letters 75 (2000) 275–281

Information Processing Letters

www.elsevier.com/locate/ipl

Simple and fast linear space computation of longest common subsequences

Claus Rick

Universität Bonn, Institut für Informatik IV, Römerstr. 164, 53117 Bonn, Germany

Received 17 March 2000
Communicated by T. Lengauer

Keywords: Algorithms; Longest common subsequence; Linear space

1. Introduction

Given two sequences $A = a_1a_2 \dots a_m$ and $B = b_1b_2 \dots b_n$, $m \leqslant n$, over some alphabet Σ of size s the longest common subsequence (LCS) problem is to find a sequence of greatest possible length that can be obtained from both A and B by deleting zero or more (not necessarily adjacent) symbols. Applications for the LCS problem arise in many different areas since the length, p, of a longest common subsequence can be viewed as a simple measure of similarity between two sequences. There is a wide range of efficient algorithms, suiting different purposes, which can compute the length of an LCS using only linear space [7]. The space requirement of these algorithms usually rises to $\mathrm{O}(mn)$ when a longest common subsequence has to be constructed, and, as stated in [2], it is not obvious in general that an LCS algorithm can be adapted to run in linear space without substantial alteration of its time complexity.

In this paper we show how to maintain the $\mathrm{O}(\min\{pm, p(n-p)\})$ time complexity and linear space of the algorithm introduced in [11] which seems to have been widely accepted as very fast and flexible. All linear space implementations, including ours, rely on variations of a divide and conquer scheme first introduced by Hirschberg [5]. The main difficulty in keeping up the time complexity of an algorithm is to find a suitable partition into two subproblems such that the time to solve these is about half the time required for the original problem [2]. Although the asymptotic time complexity could be shown to remain unchanged for some algorithms, the application of the divide and conquer scheme introduces some overhead, which can be estimated theoretically for the worst-case, and which varies for different methods suggested so far. In its ideal form, i.e., when problems are evenly split into subproblems, one can expect a doubling of the original running time. But in order to split the problem evenly some methods have to calculate the length of an LCS in a separate stage preceding the divide and conquer scheme [3,8]. So their performance can be expected to be three times that of the original algorithm. Recently, Goeman and Clausen [4] suggested a variation of the algorithm from [11] and developed a linear space implementation. Although they report experiments indicating that their algorithm is fast in practice they had to add an $m \log m$ term with respect to the asymptotic running time, and they obtain a worst-case overhead factor of 5.25 since their algorithm is

E-mail address: rick@cs.uni-bonn.de (C. Rick).

Table 1
Linear space algorithms computing an LCS. c gives the theoretical worst-case overhead

Year	Author(s)	Time	c	Paradigm
1975	Hirschberg [5]	$O(mn)$	2	dyn. programming
1985	Apostolico, Guerra [1]	$O(m \log m + pm)$	$[2, \log m]$	contours
1986	Myers [9]	$O(n(n-p))$	2	shortest path
1987	Kumar, Rangan [8]	$O(n(m-p))$	3	contours
1990	Wu et al. [12]	$O(n(m-p))$	2	shortest path
1992	Apostolico et al. [3]	$O(n(m-p))$	3	contours
1992	Apostolico et al. [3]	$O(pm)$	3	contours
1999	Goeman, Clausen [4]	$O(\min\{pm, m \log m + p(n-p)\})$	$[5.25, \log m]$	contours
1999	this paper	$O(\min\{pm, p(n-p)\})$	2	contours

not guaranteed to split the LCS evenly. Therefore, depending on p, the overhead varies between 5.25 and $\log m$. The details of their method are quite involved. Table 1 gives a survey of linear space algorithms for the LCS problem. Most algorithms also have to perform some additional bookkeeping which is difficult to estimate theoretically. We omitted the times necessary for some standard preprocessing, solving the so-called string identification problem, since it does not affect the time complexity of the divide and conquer scheme for the main processing phase.

Our linear space implementation of the algorithm from [11] also employs the divide and conquer scheme and, like in most linear space implementations, it exploits the fact that the basic algorithm can equally well be applied to the reversed input strings. We highlight general structural facts concerning this symmetry and use these facts to develop a general method which

(1) eliminates the need for a separate stage to compute the length;
(2) splits an LCS evenly;
(3) supports a speed up of the computation in practice.

Using this method we are able to reduce the theoretical overhead factor to 2 for a variety of algorithms based on the same paradigm (contours) to compute an LCS [6,3,11]. We will shortly review this paradigm in Section 2. Before, such a factor was only known to be achievable for the basic dynamic programming algorithm [5], and for algorithms employing a different paradigm [9,12]. For these latter algorithms it was observed [10] that in practice finding a midpoint (and thus the length) of an LCS by attacking the problem from two sides was twice as efficient as the basic algorithm, which means that the time overhead of the divide and conquer scheme is negligible.

2. Computing contours

It is common to describe the LCS problem in the following way. An ordered pair (i, j), $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$ is called a *match* if $a_i = b_j$. The set M of all matches can be represented by a *matching matrix* of size $m \times n$ in which each match is identified by a circle. Two matches (i, j) and (i', j') may be part of the same common subsequence if and only if $i < i' \wedge j < j'$ or $i' < i \wedge j' < j$. A sequence of matches that is strictly increasing in both components is called a *chain*. The LCS problem can now be viewed as finding a longest chain.

For a match (i, j) we say that it is of *rank* k if the length of a longest chain ending at (i, j) is k. Thus, M can be partitioned into classes $C_1, C_2, \ldots, C_p$, each class containing matches of the same rank. It is well known that these classes exhibit a special structure in the matching matrix. If sorted with respect to the first component, matches belonging to the same class shift from right to left, and they form so-called *contours* when connected by lines as shown in Fig. 1(a). Contours of different classes may never cross or touch, and the contour of each class divides the matrix into a top/left part and a bottom/right part. Each contour can be completely specified by *dominant matches*, i.e., those matches (i, j) in a class for which there is no other match (i', j') in the same class with

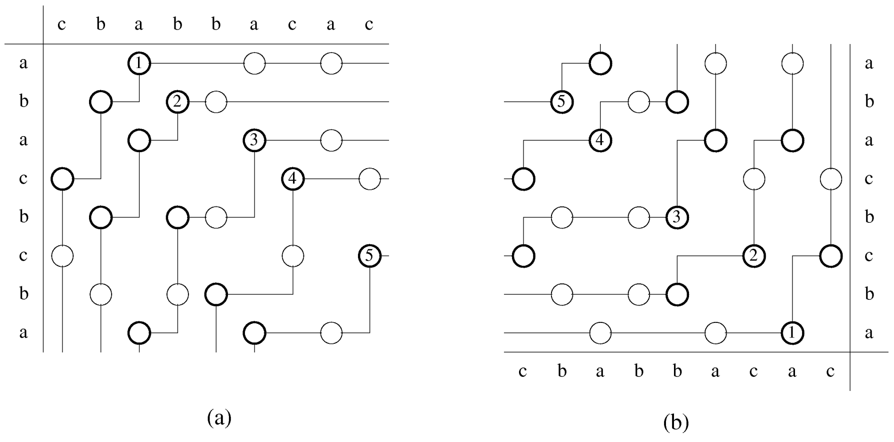


Fig. 1. Forward contours (a) and backward contours (b).

$i' = i \wedge j' < j$ or $i' < i \wedge j' = j$. They are indicated by bold circles in Fig. 1. Note that there will always be an LCS consisting only of dominant matches. For more background on this see, for example, [11].

One way to find a longest common subsequence is to compute these classes (or contours) rank by rank employing the mentioned structural properties. This was first suggested by Hirschberg [6]. Efficiency was later gained by concentrating on the computation of dominant matches [3]. We note that the algorithm from [11] can be easily modified to work in the same style while maintaining its time bound. It is this modified algorithm for which we describe a linear space implementation. Details on how these algorithms actually compute the contours are not important to understand our approach. However, we have to keep in mind that they will only determine the dominant matches.

3. The new method

Note that the strategy sketched in the previous section would equally well work to compute the length of an LCS if applied from the back of the strings (i.e., to the reversed input strings). In this way we may obtain *backward contours* as opposed to our standard *forward contours*. The number of forward contours and backward contours will be the same and for every match there will be a unique forward contour and a unique backward contour. We will denote the set of matches with *forward rank k* by FC_k and those with *backward rank k* by BC_k. Forward contours and backward contours might look very different as can be seen in Fig. 1. Nevertheless they are related in a special way which forms the basis of our new approach.

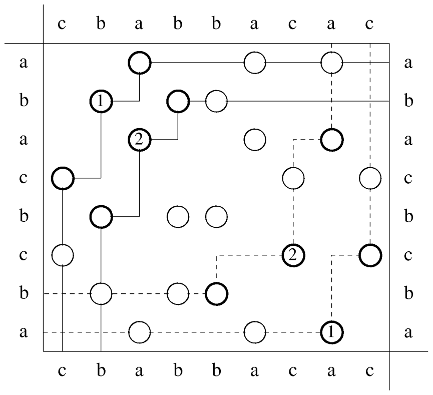
Lemma 1. *Let p be the length of an LCS between strings A and B. Then for every match (i, j) the following holds*:

There is an LCS containing (i, j) if and only if (i, j) lies on the kth forward contour and *on the $(p-k+1)$st backward contour for some $k \in [1, p]$.*

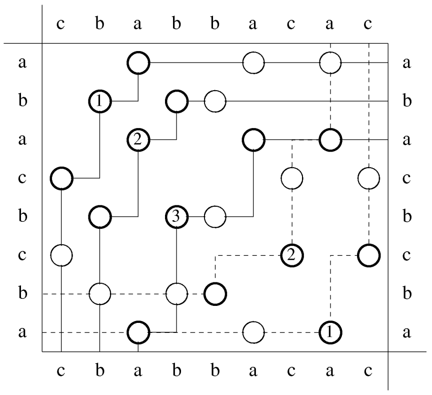
Proof. ($\Rightarrow$) If there is a forward chain of length k ending at (i, j) and a backward chain of length $p - k + 1$ ending at (i, j) we can glue them together to obtain a CS of length p.

($\Leftarrow$) Assume there is an LCS containing (i, j) as its kth element. Then (i, j) has forward rank at least k and backward rank at least $p - k + 1$ since matches of an LCS form a chain of length p. Would its forward rank or its backward rank be greater than k or $p - k + 1$, respectively we could construct a CS of length $> p$. But this would contradict p being the length of an LCS. □

Based on this lemma our idea to compute the length and the midpoint of a longest common subsequence is as follows. Start computing forward contours and backward contours in an alternating fashion, i.e., use



(a)



(b)

Fig. 2. Forward and backward contours overlayed.

the order $FC_1, BC_1, FC_2, BC_2, \ldots$. In this way more and more matches will be covered, i.e., assigned to a forward contour or to a backward contour. Sooner or later there will also be matches covered by both a forward contour and a backward contour. In particular this means that forward contours and backward contours will start to cross (see Fig. 2 for an example). Observing the last contour which covers matches not assigned to any previous contour we will have found the length of an LCS, and any of the matches covered by this last contour for the first time may serve as a midpoint of an LCS (when the length of an LCS is even, say $2k$, our "midpoint" will be the $(k+1)$st match on an LCS). Then we can apply the same algorithm recursively. We now prove that this method is in fact correct and that it can be implemented efficiently.

We define sets M_i containing the matches which remain uncovered by contours computed up to certain points in the computation: $M_0 := M$, $M_1 := M_0 \setminus FC_1$, $M_2 := M_1 \setminus BC_1$, and for $i \geqslant 2$ we set $M_{2i-1} := M_{2(i-1)} \setminus FC_i$ and $M_{2i} := M_{2i-1} \setminus BC_i$. For example, in Fig. 2(a) $M_4 = \{(3,6),(5,5),(5,4)\}$ while in Fig. 2(b) $M_5 = \emptyset$. Obviously, $M = M_0 \supset M_1 \supset M_2 \supset M_3 \supset \cdots$. Let $\hat{p}$ be the minimal index such that $M_{\hat{p}} = \emptyset$.

Lemma 2. *The length of an LCS is $\hat{p}$ and each match in $M_{\hat{p}-1}$ is a possible midpoint of an LCS.*

Proof. We only prove the case where $\hat{p}$ is odd. The other case is analogous. Our situation is as follows. Since $M_{\hat{p}}$ is the first set equal to $\emptyset$, the matches in $M_{\hat{p}-1}$ were covered by FC_k, $k = (\hat{p}+1)/2$, but not by any contour $FC_l, BC_l, l < k$. So their forward rank and their backward rank is at least k which means that a CS of length $2k-1 = \hat{p}$ exists. We have to show that there can be no longer CS. In order to prove this we will argue that there can be no match whose forward rank and backward rank adds up to more than $2k$.

We claim that all matches in $M_{\hat{p}-1}$ are also on BC_k. Assume some match $(i,j) \in M_{\hat{p}-1}$ has backward rank $> k$. This means that there has to be some match (i',j') of backward rank k which succeeds a match in BC_{k-1} and precedes (i,j) on a backward chain. But such a match would not have been covered by any FC_l, $l \leqslant k$, or any BC_l, $l < k$. This, however, would imply that $M_{\hat{p}} \neq \emptyset$. A contradiction. Also note that all matches on FC_k except those in $M_{\hat{p}-1}$ are covered by some BC_l, $l < k$, i.e., $M_{\hat{p}-1} = FC_k \setminus BC^*_{k-1}$ where $BC^*_t := \bigcup_{i=1}^{t} BC_i$ (FC^*_t is defined analogously). So no match on FC_k has backward rank $> k$.

This claim is the basis of an induction which, using similar arguments, shows that for $1 \leqslant l \leqslant k-1$ all matches in $FC_{k-l} \setminus BC^*_{k-1+l}$ are on BC_{k+l} and, symmetrically, that all matches in $BC_{k-l} \setminus FC^*_{k-1+l}$ are on FC_{k+l}. From this it follows that for $1 \leqslant l \leqslant k-1$, $BC_{k+l} \subseteq FC^*_{k-l}$ and that $FC_{k+l} \subseteq BC^*_{k-l}$. But then there is no match whose forward rank and backward rank sum up to more than $(k+l)+(k-l)=2k$ which implies that no CS of length $> 2k-1=\hat{p}$ exists.

Since matches in $M_{\hat{p}-1}$ are on both FC_k and BC_k they are possible midpoints of an LCS by Lemma 1. □

We can not, however, keep track of these sets explicitly in order to find $\hat{p}$. This would be too costly. We need a criterion to detect that all matches have been covered which is solely based on the knowledge of dominant matches (this is all the information we can expect to obtain from our algorithm). Further, we would like to identify at least one dominant match as midpoint of an LCS. The basic idea is to check how the two most recently computed contours interact. From our alternating order of computation this will always be contours FC_f and BC_b with $f-b \leqslant 1$. Let $\hat{f}$ and $\hat{b}$ be the minimal indices such that no dominant match in $BC_{\hat{b}}$ is to the bottom/right of any dominant match in $FC_{\hat{f}}$ (or, equivalently, no dominant match in $FC_{\hat{f}}$ is to the top/left of any dominant match match in $BC_{\hat{b}}$). Intuitively, this is the first time that the two contours do not cross anymore (they will, however, touch each other at least once). The following lemma shows that this is a suitable termination criterion for our computation.

Lemma 3.
(1) *The length of an LCS is* $\hat{b}+\hat{f}-1$.
(2) $FC_{\hat{f}}$ *and* $BC_{\hat{b}}$ *share at least one match which is dominant on* $FC_{\hat{f}}$ *and/or* $BC_{\hat{b}}$.

Proof. We only prove the case $\hat{b}=\hat{f}$, i.e., $BC_{\hat{b}}$ is the last computed contour (the case $\hat{b}+1=\hat{f}$ is symmetric).

First we prove (2), i.e., we show that at least one dominant match on $BC_{\hat{b}}$ is on $FC_{\hat{f}}$. Since $\hat{f}$ and $\hat{b}$ are minimal we can conclude that there is a dominant match $(i,j)\in FC_{\hat{f}}$ which is to the top/left of some dominant match on $BC_{\hat{b}-1}$ and which is covered by $BC_{\hat{b}}$. Consider the dominant match $(i',j')\in BC_{\hat{b}}$ which shares at least on component with (i,j). We claim that $(i',j')\in FC_{\hat{f}}$. From the shape of contours we know that either $i\leqslant i' \wedge j=j'$ or $i'=i \wedge j\leqslant j'$. So if $(i',j')\notin FC_{\hat{f}}$ it would have to be located to the bottom/right of some dominant match on $FC_{\hat{f}}$. But this contradicts $\hat{f}$ and $\hat{b}$ being minimal.

Now we prove that $\hat{b}+\hat{f}-1=\hat{p}$ (then (1) follows from Lemma 2). Since no dominant match on $BC_{\hat{b}}$ is to the bottom/right of any dominant match on $FC_{\hat{f}}$ we can also conclude that $BC_{\hat{b}}\subseteq FC^*_{\hat{f}}$. But this means that all matches will still be covered by $FC^*_{\hat{f}}$ and $BC^*_{\hat{b}-1}$ and therefore $\hat{p}\leqslant \hat{f}+\hat{b}-1$. On the other hand $FC_{\hat{f}}$ and $BC_{\hat{b}}$ share at least one match (by (2)). This match would not be covered by $FC^*_{\hat{f}-1}$ and $BC^*_{\hat{b}-1}$ and so $\hat{p}>\hat{f}+\hat{b}-2$. □

So all we must be able to do is to check whether dominant matches of the most recent contour are to the top/left, to the bottom/right or on the contour computed just before the most recent contour. With respect to the time complexity it is important to perform these tests in time proportional to the number of dominant matches of the contours involved. Assuming that dominant matches of both contours are available as sorted lists (e.g., from bottom/left to top/right) and noting the special shape of contours it is not difficult to see that a single appropriate scan of the lists is sufficient. The discussion above finally gives

Theorem 4. *A longest common subsequence can be constructed in* $O(ns+\min\{pm, p(n-p)\})$ *time and linear space.*

Proof. Time and space $O(ns)$ is needed for a standard preprocessing stage solving the so-called string identification problem. This supports certain queries during the main processing stage. Note, however, that we have to solve this problem only once before starting the divide and conquer scheme. So the crucial part is the analysis of this scheme for the main algorithm. Since we only need to retain information on the most recent forward contour and on the most recent backward contour space requirement is clearly bounded by $O(n)$. Computing the forward contours takes time $O(\min\{\frac{1}{2}pm, \frac{1}{2}p(n-p)\})$ as does the computation of the backward contours. The number of dominant matches on each contour is bounded by $O(\min\{m,(n-p)\})$. So checking for termination according to Lemma 3 is within our time bound. The total time to compute the length and the midpoint of an LCS therefore remains $O(ns+\min\{pm, p(n-p)\})$.

We split the problem at some match (m',n') on contour $\lceil \frac{1}{2}p\rceil$ which is the midpoint of the LCS to be constructed. Then we recursively determine an LCS of length $p_1 = \lceil \frac{1}{2}p \rceil - 1$, for the length $m' - 1$ and length $n' - 1$ prefixes of A and B, respectively, and an LCS of length $p_2 = \lfloor \frac{1}{2}p \rfloor$, for the length $m - m'$ and length $n - n'$ suffixes of A and B, respectively. So $p_1 + p_2 = p - 1$ and $p_1, p_2 \leqslant \frac{1}{2}p$. The recursion is stopped whenever we discover a subproblem having an LCS of length zero or when the length of the shorter string of the subproblem equals the length of an LCS. In the later case all characters of the shorter string must be part of the LCS. Note that we will always know the length of an LCS which has to be found for a certain subproblem.

The actual number of steps performed by the original algorithm is upper bounded by $2 \cdot \min\{pm, p(n-p)\}$. In calculating the time T_1 to find the midpoints of the two subproblems on the first level of the recursion we have to consider both bounds in turn. First we have

$$\begin{aligned} T_1 &\leqslant 2 \cdot p_1(m'-1) + 2 \cdot p_2(m-m') \\ &\leqslant p(m'-1+m-m') \\ &\leqslant pm. \end{aligned}$$

In the other case there is a subtlety since $n = \max\{m, n\}$ is assumed in the time bound of the basic algorithm. But after splitting at (m', n') it does not necessarily hold that

$$\begin{aligned} &\max\{m'-1, n'-1\} + \max\{m-m', n-n'\} \\ &\quad \leqslant \max\{m, n\}. \end{aligned}$$

However, we note that the time bound of the basic algorithm can be reduced to $p(n-p) + p(m-p) \leqslant 2 \cdot p(n-p)$ when p is known. Then we have

$$\begin{aligned} T_1 &\leqslant p_1(m'-1-p_1) + p_1(n'-1-p_1) \\ &\quad + p_2(m-m'-p_2) + p_2(n-n'-p_2) \\ &\leqslant \frac{p}{2}(m'-1-p_1+m-m'-p_2) \\ &\quad + \frac{p}{2}(n'-1-p_1+n-n'-p_2) \\ &= \frac{p}{2}\bigl(m-1-(p_1+p_2)\bigr) \\ &\quad + \frac{p}{2}\bigl(n-1-(p_1+p_2)\bigr) \\ &= \frac{p}{2}(m-p) + \frac{p}{2}(n-p). \end{aligned}$$

So $T_1 \leqslant \min\{pm, p(n-p)\}$. Continuing the recursion, the total time to compute an LCS will be bounded by

$$\begin{aligned} &2 \cdot \sum_{i=0}^{\log m} \left(\frac{1}{2}\right)^i \min\bigl\{pm, p(n-p)\bigr\} \\ &\quad \leqslant 4 \min\bigl\{pm, p(n-p)\bigr\} \end{aligned}$$

giving a worst-case overhead factor of 2. □

Apart from eliminating the need for an extra stage to compute the length of an LCS the alternating computation of forward contours and backward contours supports an additional speedup of the implementation. Observing the crossing of the two most recently computed contours, say FC_f and BC_b, we can identify parts of the next contour which need not be computed at all. In general there will be matches on $(i, j) \in FC_f$ which have been covered by some BC_l, $l \leqslant b$ (a symmetric observation holds for BC_b). Thus the sum of the forward rank and backward rank of (i, j) will be at most $f + b$. Any match (i', j') on FC_{f+1} which originates from (i, j) (i.e., $i < i' \wedge j < j'$) will be covered by some $BC_{l'}$, $l' < l$. This means that the sum of the forward rank and backward rank of (i', j') will not exceed $b + f$. But our goal is to identify some match maximizing this sum. The value $b + f$ may only be exceeded by matches on FC_{f+1} which have been uncovered so far or which have been covered by BC_b. Fig. 3 illustrates this situation. Thus we have the following fact.

Lemma 5. *It is sufficient to compute only those matches on FC_{f+1} which originate from* (*dominant*) *matches $(i, j) \in FC_f \setminus BC_b^*$.*

A similar statement holds for the computation of BC_{b+1}. An easy way to apply this observation in practice is to find the two "outer" matches in $FC_f \setminus BC_b^*$, i.e., the leftmost and the rightmost one (in Fig. 3 this would be matches a and b). We then only determine that part of FC_{f+1} which originates from matches on FC_f located between these two outer matches. Note that finding (one of) the outer matches is also necessary to check for termination. This is easily done by scanning the sorted lists of dominant matches of the involved contours. In practice we can expect those parts of the contours that really have to be computed to get smaller and smaller as we approach the middle of an LCS.

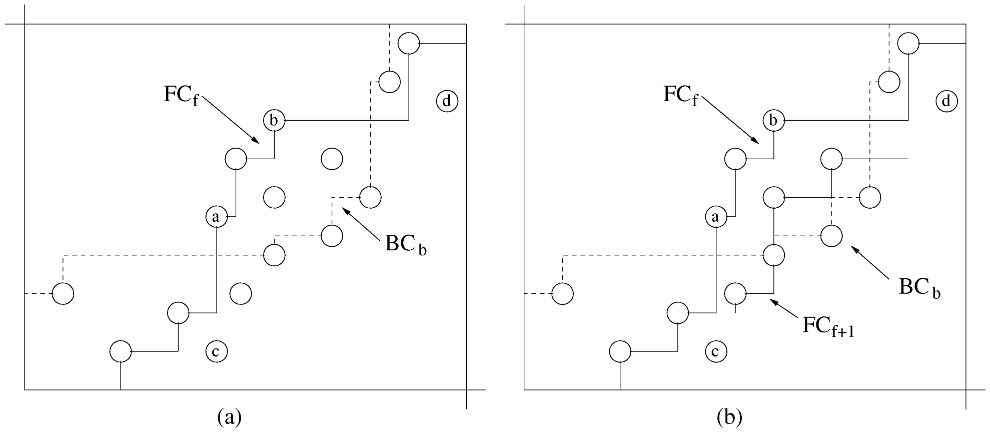


Fig. 3. Cutting of contours.

4. Conclusion

We presented a new simple method to obtain fast linear space implementations for some LCS algorithms. In particular we showed that an LCS can be constructed in $O(ns + \min\{pm, p(n-p)\})$ time and linear space.

References

[1] A. Apostolico, C. Guerra, A fast linear space algorithm for computing longest common subsequences, in: Proc. of the 23rd Allerton Conference on Communication, Control and Computing, Monticello, 1985, pp. 76–84.

[2] A. Apostolico, String editing and longest common subsequences, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Vol. 2, Springer, Berlin, 1997, pp. 361–398.

[3] A. Apostolico, S. Browne, C. Guerra, Fast linear-space computations of longest common subsequences, Theoret. Comput. Sci. 92 (1992) 3–17.

[4] H. Goeman, M. Clausen, A new practical linear space algorithm for the longest common subsequence problem, in: Proc. Prague Stringology Club Workshop'99, Report DC-99-05, Department of Computer Science and Engineering, Czech Technical University, 1999, pp. 40–60.

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

[6] D. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (1977) 664–675.

[7] D. Hirschberg, Serial computations of Levenshtein distances, in: A. Apostolico, Z. Galil (Eds.), Pattern Matching Algorithms, Oxford University Press, 1997, pp. 123–141.

[8] S. Kumar, C. Rangan, A linear space algorithm for the LCS problem, Acta Inform. 24 (1987) 353–363.

[9] E.W. Myers, An $O(nd)$ difference algorithm and its variations, Algorithmica 1 (1986) 251–266.

[10] E.W. Myers, W. Miller, Optimal alignments in linear space, CABIOS 4 (1988) 11–17.

[11] C. Rick, A new flexible algorithm for the longest common subsequence problem, in: Proc. CPM'95, Lecture Notes in Comput. Sci., Vol. 937, Springer, Berlin, 1995, pp. 340–351. Also in: Nordic J. Comput. 2 (1995) 444–461.

[12] S. Wu, U. Manber, G. Myers, W. Miller, An $O(np)$ sequence comparison algorithm, Inform. Process. Lett. 35 (1990) 317–323.