Fully Incremental LCS Computation

Yusuke Ishida[1], Shunsuke Inenaga[1],
Ayumi Shinohara[2,3], and Masayuki Takeda[1,4]

[1] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
{y-ishida, shunsuke.inenaga, takeda}@i.kyushu-u.ac.jp
[2] Graduate School of Information Sciences, Tohoku University,
Sendai 980-8579, Japan
ayumi@ecei.tohoku.ac.jp
[3] PRESTO, Japan Science and Technology Agency (JST)
[4] SORST, Japan Science and Technology Agency (JST)

Abstract. *Sequence comparison* is a fundamental task in pattern matching. Its applications include file comparison, spelling correction, information retrieval, and computing (dis)similarities between biological sequences. A common scheme for sequence comparison is the *longest common subsequence* (*LCS*) metric. This paper considers the *fully incremental LCS computation* problem as follows: For any strings A, B and characters a, b, compute $LCS(aA, B)$, $LCS(A, bB)$, $LCS(Aa, B)$, and $LCS(A, Bb)$, provided that $L = LCS(A, B)$ is already computed. We present an efficient algorithm that computes the four LCS values above, in $O(L)$ or $O(n)$ time depending on where a new character is added, where n is the length of A. Our algorithm is superior in both time and space complexities to the previous known methods.

1 Introduction

Pattern matching is one of the most extensively studied sub-areas of theoretical computer science [1, 2], and one example of the fundamental problems on pattern matching is *sequence comparison* [3]. There are a wide range of applications for sequence comparison, including file comparison [4], spelling correction [5], information retrieval [6], and computing (dis)similarities between biological sequences [7, 8]. Comparing two strings $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$ can be done by computing an *alignment* between these strings. Standard alignment algorithms compute a dynamic programming matrix DP for the optimal alignments between the consecutive prefixes of A and B. Namely, each entry $DP[i, j]$ stores the score of the alignment between $A[1..j] = a_1 \cdots a_j$ and $B[1..i] = b_1 \cdots b_i$.

A common scheme of sequence comparison is the *longest common subsequence* (*LCS*) metric [9]. A subsequence of string A is any string obtained by removing 0 or more characters from A, and the LCS of two strings A and B (denoted by $LCS(A, B)$) is the longest subsequence that commonly appears in both A and B. In the LCS measure, matched pairs of characters are assigned score 1 and

Table 1. Comparison of complexities for fully incremental LCS computation provided that $LCS(A, B)$ is already computed, where $n = |A|$ and $m = |B|$. Note that $L = LCS(A, B) \leq \min(n, m)$ always holds. The last row shows the total space requirement of each algorithm.

	Naive DP	Modified algorithm of [12]	Our algorithm
time for $LCS(aA, B)$	$O(mn)$	$O(m+n)$	$O(L)$
time for $LCS(Aa, B)$	$O(m)$	$O(m)$	$O(L)$
time for $LCS(A, bB)$	$O(mn)$	$O(m+n)$	$O(n)$
time for $LCS(A, Bb)$	$O(n)$	$O(n)$	$O(n)$
total space complexity	$O(mn)$	$O(mn)$	$O(nL+m)$

unaligned characters are assigned score 0, and the objective is to compute an optimal alignment that gives the maximum score corresponding to $LCS(A, B)$.

$LCS(A, B)$ can be obtained by computing the DP matrix in $O(mn)$ time. The DP approach is suitable for on-line incremental computation of the LCS, in such a situation where upcoming characters are appended to the tails of A and/or B. In fact, $LCS(Aa, B)$ and $LCS(A, Bb)$ can be easily computed in $O(m)$ and $O(n)$ time respectively, provided that $LCS(A, B)$ is already computed. This enables us an efficient processing of e.g. streaming data.

In recent years, the research of computing string alignments to the reversed direction (from right to left) has been a popular topic of pattern matching. Examples of motivations are to process log files backdating to the past, and to compute the alignments between not only the prefixes but also the suffixes of a biological sequence and another biological sequence [10]. However, a naive use of the DP approach is not efficient enough: Since prepending a character a to the head of A can change all the entries of the DP table, we have to recompute the whole DP table from scratch, and this obviously takes $O(mn)$ time. Significant improvement was given by Landau et al. [11] for the *edit distance* metric. For the edit distance metric, their algorithm performs in $O(m+n)$ time. Kim and Park [12] presented a simpler algorithm solving the same problem in the same complexity. Landau et al. [10] introduced the *consecutive suffix alignment problem* and showed two algorithms to solve this problem; the first one runs in $O(nL+m)$ time and space, and the second one in $O(nL)$ time and space, where $L = LCS(A, B)$, assuming that the alphabet is fixed. Note that $L \leq \min(n, m)$ always holds.

This paper treats *fully incremental LCS computation* where characters are added to any position of the heads and tails of A and B. In so doing, we pay our attention to the $O(nL+m)$ algorithm by Landau et al. in [10]. In this paper, we produce an algorithm for fast, flexible, and efficient computation of LCS. The result of this work is summarized in Table 1. It is actually possible to apply the algorithm of Kim and Park [12] to fully incremental LCS computation, which was originally designed for the edit distance metric. However, as seen in Table 1, our algorithm is superior to their algorithm in both time and space complexities.

2 Preliminaries

Let Σ be a finite *alphabet*. Throughout this paper we assume that Σ is fixed. An element of Σ^* is called a *string*. For string $A = a_1 a_2 \cdots a_n$, let $|A|$ denote its length, namely $|A| = n$. Let $A[i] = a_i$ and $A[i..j] = a_i \cdots a_j$, where $1 \le i \le j \le n$. Then $A[1..j]$ is called a *prefix*, $A[i..j]$ a *substring*, and $A[i..n]$ a *suffix* of A. Sequence $A[i_1]A[i_2] \cdots A[i_\ell]$ is called a *subsequence* of A of length ℓ, where $1 \le i_1 < i_2 \ldots < i_\ell \le n$. Note that any substring of A is a subsequence of A. Let $B = b_1 b_2 \cdots b_m$. A subsequence occurring in both A and B is called a *common subsequence* of A and B, and the longest such subsequence is called the *longest common subsequence* (*LCS*) of A and B, which is denoted by $LCS(A, B)$.

A standard technique for computing $LCS(A, B)$ is the *dynamic programming* method, where we compute the DP matrix of size $(m + 1) \times (n + 1)$ for which $DP[i, j] = LCS(A[1..j], B[1..i])$ for $1 \le j \le n$ and $1 \le i \le m$. The recurrence of the DP matrix is the following:

$$DP[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \max(DP[i-1,j], DP[i,j-1]) & \text{if } i,j > 0 \text{ and } A[j] \neq B[i], \\ DP[i-1,j-1] + 1 & \text{if } i,j > 0 \text{ and } A[j] = B[i]. \end{cases}$$

Therefore, to compute $LCS(A, B) = DP[m, n]$, we need $O(mn)$ time and space.

Pair (i, j) is said to be a *match point* between A and B, if $A[j] = B[i]$. Pair (i, j) is said to be a *partition point* of DP if $DP[i, j] = DP[i-1, j]+1$. P denotes the set of the partition points of DP. Let $(i, j) \in P$ and $DP[i, j] = v$. Then we write as $P[v, j] = i$, namely, $P[v, j]$ is the first row index i at column j of DP which bears v. See Fig. 1 for examples of match points and partition points.

3 The Landau Myers Ziv-Ukelson Algorithm

Assume that, given two strings A, B, we have already computed $L = LCS(A, B)$. In this section we recall the algorithm of [10] which, for any character a, computes $LCS(aA, B)$ in amortized $O(L)$ time. This algorithm computes only the partition points rather than the whole DP matrix, thus saving both time and space.

Let DP^{Ah} and P^{Ah} denote the DP matrix and the partition point set obtained from DP and P by adding a new character a to the head of A, respectively. Let $n = |A|$ and $m = |B|$.

Lemma 1 (Landau et al. [10]). *P^{Ah} is computed by inserting at most one new partition point at each column of P.*

See Fig. 1 for a concrete example of the above lemma.

In Lemma 2 we will show how to compute in $O(1)$ time the new partition point for each column. In so doing, we construct the *next match table* (*NM* table) as follows: $NM[i, a]$ returns $\min\{i' \mid i' > i \text{ and } B[i'] = a\}$, if such i' exists. Otherwise, it returns *null*. For fixed alphabet Σ the size of *NM* table is $O(m)$. An example of *NM* table is shown in Fig. 2.

	b	a	d	b	d	c	d
b		0	0	1	1	1	1
c		0	0	1	1	2	2
b		0	0	1	1	2	2
d		0	1	1	2	2	3

	b	a	d	b	d	c	d
b	1	1	1	1	1	1	1
c	1	1	1	1	1	2	2
b	1	1	1	2	2	2	2
d	1	1	2	2	3	3	3

Fig. 1. DP (left) and DP^{Ah} (right) with $A = \texttt{adbdcd}$, $B = \texttt{bcbd}$ and $a = \texttt{b}$. Cells marked with a circle and rectangle are match and partition points, respectively. Grey rectangles show the new partition points inserted into DP^{Ah}.

		a	b	c	d
	0	null	1	2	4
b	1	null	3	2	4
c	2	null	3	null	4
b	3	null	null	null	4
d	4	null	null	null	null

Fig. 2. NM table for string $B = \texttt{bcbd}$ with alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$.

Lemma 2 (Landau et al. [10]). *Let $I_{j-1} = P^{Ah}[v, j-1]$ denote the row index of the new partition point in column $j-1$ of P^{Ah}. Then, the new partition point I_j at column j of DP^{Ah} is computed as follows:*

$$I_j = \begin{cases} I_{j-1} & \text{if } P^{Ah}[v, j-1] \le P[v, j], \\ \min\{NM(P^{Ah}[v, j-1], A[j]), P^{Ah}[v+1, j-1]\} & \text{if } P^{Ah}[v, j-1] > P[v, j]. \end{cases}$$

Note that a special case occurs in Lemma 2 when v is the highest value in column $j-1$ of DP^{Ah}, and therefore partition point $P^{Ah}[v+1, j-1]$ does not exist. In this case, $P^{Ah}[v+1, j-1]$ is set to the dummy index $m+1$, so that we can proceed according to the above lemma.

The stop condition of the update procedure is as follows.

Lemma 3 (Landau et al. [10]). *If column j of DP^{Ah} is identical to colum j of DP, then all columns $j' > j$ of DP^{Ah} are also identical to columns j' of DP.*

The partition point set P is implemented by a double linked list in order that insertion of new partition points can be done in $O(1)$ time. The row indices correspond to the LCS values and the column indices correspond to the positions of string A, and each cell stores the corresponding row index of B. Fig. 3 shows an example of the update of P to P^{Ah}. It is obvious that the size of the partition point set is bounded by $O(nL)$. Since insertion of each new partition point can

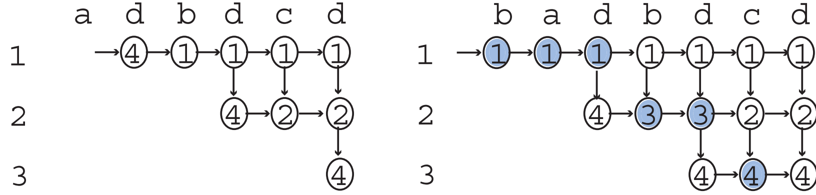

Fig. 3. Update of P with strings $A = $ `dbdcd` and $B = $ `bcbd` to P^{Ah} with new character $a = $ `b`. Grey circles are the new partition points inserted to P^{Ah}.

be done in $O(1)$ time, this set can be constructed in $O(nL)$ time. Since $|A| = n$, each incrementation of a new character to the head of A takes the following time.

Theorem 1 (Landau et al. [10]). *Provided that $L = LCS(A, B)$ is already computed, for any character a, $LCS(aA, B)$ is computable in amortized $O(L)$ time.*

4 A Fully Incremental LCS Computation Algorithm

In this section we produce an efficient algorithm to solve the *fully incremental LCS computation problem*, where the problem is to compute the LCS of given two strings under the condition that characters are added to any of the heads and tails of the two strings at any time. Namely, we are to compute $LCS(aA, B)$, $LCS(Aa, B)$, $LCS(A, bB)$, or $LCS(A, Bb)$. The first one, $LCS(aA, B)$, is computable in amortized $O(L)$ time due to Theorem 1 by Landau et al. [10], as recalled in Section 3. In what follows, we will show how to compute the three others.

4.1 Computing $LCS(A, bB)$

Assume we have already computed $LCS(A, B)$. Let DP and P be the DP table and the partition point set for $LCS(A, B)$, respectively. Let DP^{Bh} and P^{Bh} denote the DP matrix and the partition point set for $LCS(A, bB)$ with character b, respectively. Let $n = |A|$ and $m = |B|$.

Where partition points are updated. This subsection is devoted to clarifying where partition points are possibly changed in the DP table when computing $LCS(A, bB)$ from $LCS(A, B)$. Fig. 4 shows an example of updating DP to DP^{Bh}.

Let $\ell = \min\{j \mid A[j] = b\}$. Namely, ℓ is the smallest column index of DP^{Bh} in which a match point exists in the first row. Then we have the following proposition.

	a	a	a	a	b	a	c	b	a	b	c	a
b												
c	0	0	0	0	0	0	1	1	1	1	1	1
b	0	0	0	0	1	1	1	2	2	2	2	2
a	1	1	1	1	1	2	2	2	3	3	3	3
b	1	1	1	1	2	2	2	3	3	4	4	4
a	1	2	2	2	2	3	3	3	4	4	4	5
c	1	2	2	2	2	3	4	4	4	4	5	5

		a	a	a	a	b	a	c	b	a	b	c	a
b	0	0	0	0	0	1	1	1	1	1	1	1	1
c	0	0	0	0	0	1	1	2	2	2	2	2	2
b	0	0	0	0	0	1	1	2	3	3	3	3	3
a	0	1	1	1	1	1	2	2	3	4	4	4	4
b	0	1	1	1	1	2	2	2	3	4	5	5	5
a	0	1	2	2	2	2	3	3	3	4	5	5	6
c	0	1	2	2	2	2	3	4	4	4	5	6	6

Fig. 4. Update of DP to DP^{Bh} with A = `aaaabacbabca`, B = `cbabac`, and b = `b`. Rectangles show the partition points. In DP^{Bh} on right, dashed rectangles are new partition points inserted, and circles indicate partition points deleted in updating DP to DP^{Bh}.

Proposition 1. *All the entries of DP^{Bh} are identical to those of DP at the columns smaller than ℓ, except for the first row of DP^{Bh}. The scores in the first row of DP^{Bh} are 0 at columns smaller than ℓ, while the scores are 1 at the other columns.*

See Fig. 4 for concrete examples. This proposition means that we do not need to care about these entries of the DP table. In the following, we only consider the other entries than these.

Lemma 4. *For any column $j \geq \ell$, there exists row index E_j such that*

$$DP^{Bh}[i,j] = \begin{cases} DP[i,j]+1 & \text{if } i < E_j, \\ DP[i,j] & \text{if } i \geq E_j. \end{cases}$$

Proof. Similar to the proof of Lemma 1 in [10]. □

The following lemma is derived from Lemma 4.

Lemma 5. *Column j of P^{Bh} consists of the partition points in P except for one possibly eliminated partition point from P, plus the first row index of DP^{Bh} if it has score 1 at column j. Let E_j be the smallest row index such that $\delta_{E_j} = DP^{Bh}[E_j,j] - DP[E_j,j] = 0$. Then (E_j,j) is the only partition point eliminated at column j in updating P to P^{Bh}.*

Proof. It is obvious that the first row index of DP^{Bh} becomes a partition point at each column of P^{Bh}, if it has score 1.

In what follows, we will show that (1) (E_j,j) is a partition point of DP; (2) (E_j,j) is not a partition point of DP^{Bh}.

(1) For contrary, assume (E_j,j) is not a partition point of DP. Then $DP[E_j-1,j] = DP[E_j,j]$. Since E_j is the smallest row index such that $\delta_{E_j} = 0$, by Lemma 4 we get $\delta_{E_j-1} = 1$ which yields $DP^{Bh}[E_j-1,j] = DP^{Bh}[E_j,j]+1$ but this contradicts the monotonicity of LCS. Hence (E_j,j) is a partition point of DP.

(2) For contrary, assume (E_j, j) is a partition point of DP^{Bh}. Then $DP^{Bh}[E_j - 1, j] = DP^{Bh}[E_j, j] - 1$. Since E_j is the smallest row index such that $\delta_{E_j} = 0$, by Lemma 4 we get $\delta_{E_j - 1} = 1$ which yields $DP[E_j, j] = DP[E_j - 1, j]$ which contradicts (1) above. Hence (E_j, j) is not a partition point of DP^{Bh}.

For any row $i < E_j$ of column j, we have $DP^{Bh}[i, j] = DP[i, j] + 1$ by Lemma 4. Since the first row at column j of DP^{Bh} is a new partition point of P^{Bh} with score 1, the partition point in any rows smaller than E_j are inherited from P to P^{Bh}. Similar arguments hold for the rows greater than E_j. □

See Fig. 4 for concrete examples of Lemma 5. Each entry marked by a circle is the partition point eliminated at the column.

According to Lemma 5, at each column j of P^{Bh} at most one new partition point is inserted in the first row, and at most one partition point E_j is eliminated at a larger row. In updating P to P^{Bh}, P is processed from left column to right column. Now we show where E_j can exist at each column j.

Proposition 2. *For any column $j - 1$ of DP table, let $P[v, j - 1] = x$. At the next column j, we have $DP[x, j] = v$.*

Proof. Since $DP[x, j - 1]$ is the partition point of score v, we know that $DP[x - 1, j - 1] = v - 1$. There are two possible cases:

- when (x, j) is a match point.
  By the recursion of LCS computation, $DP[x, j] = DP[x - 1, j - 1] + 1 = v$.
- when (x, j) is not a match point.
  Since $DP[x - 1, j - 1] = v - 1$, $DP[x - 1, j]$ can assume $v - 1$ or v. Thus, $DP[x, j] = \max\{DP[x - 1, j], DP[x, j - 1]\} = v$.

□

Lemma 6. *Let $(E_{j-1}, j - 1)$ and (E_j, j) be the partition points eliminated at columns $j - 1$ and j in updating P to P^{Bh}, respectively. Let $DP^{Bh}[E_{j-1}, j - 1] = v$. Then we have*

$$E_{j-1} \le E_j \le P^{Bh}[v + 1, j - 1].$$

(see Fig. 5.)

Proof. Since $DP^{Bh}[E_{j-1}, j - 1] = v$, $P[v, j - 1] = E_{j-1}$. In what follows, we will consider three kinds of rows and show that E_j can exist in none of them. Recall that $P^{Bh}[v, j - 1] < P[v, j - 1] = E_{j-1}$.

- rows smaller than or equal to $P^{Bh}[v, j - 1]$.
  Consider any partition point $(x, j - 1)$ such that $x \le P^{Bh}[v, j - 1]$ and let $DP[x, j - 1] = v'$. By Proposition 2, $DP[x, j] = v'$. On the other hand, by Lemma 4, $P[v', j - 1] = P^{Bh}[v' + 1, j - 1] = x$. Since $DP[P^{Bh}[v' + 1, j - 1], j] = v' + 1$ by Proposition 2, we have $DP^{Bh}[x, j] = DP[x, j] + 1$ which means that, for any partition point $(x, j - 1)$, we have $(x, j - 1) \in P^{Bh}$, while increasing its score just by 1. Thus no partition point is eliminated in the range smaller than $P^{Bh}[v, j - 1]$ at column j.

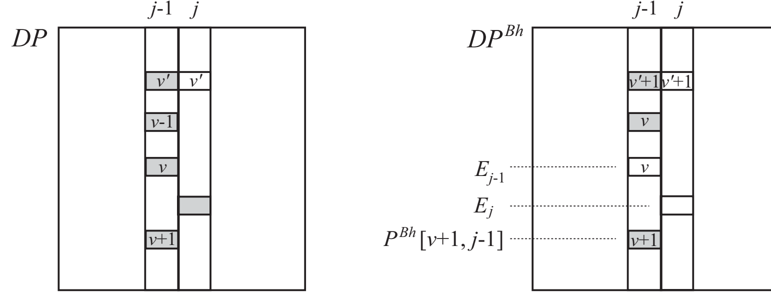

Fig. 5. The range where a partition point E_j at column j can exist, in updating P to P^{Bh}. Gray entries indicate partition points.

- rows greater than $P^{Bh}[v, j-1]$ and smaller than E_{j-1}.
  The scores of these rows in DP are all $v-1$, since $DP[E_{j-1}, j-1] = v$ and $(E_{j-1}, j-1)$ is a partition point. We have two cases.
  - when there are one or more match points in these rows at column j.
    Consider the highest such match point (of the smallest row index) and let its row index be i. Then there is a partition point (i, j) such that $P[v, j] = i$. For any row indices $P^{Bh}[v, j-1] < i' < i$, we have $DP[i', j-1] = DP[i', j] = v-1$ and $DP^{Bh}[i', j-1] = DP^{Bh}[i', j] = v$. For any row indices $i \leq i'' < E_{j-1}$, we have $DP[i'', j-1] + 1 = DP[i'', j] = v$ and $DP^{Bh}[i'', j-1] + 1 = DP^{Bh}[i'', j] = v+1$. Thus $P^{Bh}[v+1, j] = P[v, j] = i$.
  - when there are no match points in these rows at column j.
    In this case, there are no partition points in these rows of either DP or DP^{Bh}.
- rows greater than E_j.
  Similar to the first case.

Therefore we can conclude that $E_{j-1} \leq E_j \leq P^{Bh}[v+1, j-1]$. □

Eliminating partition points. In the last subsection we described where the partition points, which can possibly be eliminated, exist. In this section, we show how to quickly eliminate such partition points.

Lemma 7. *Let $(E_{j-1}, j-1)$ and (E_j, j) be the partition points eliminated at columns $j-1$ and j in updating P to P^{Bh}, respectively. Let $DP^{Bh}[E_{j-1}, j-1] = v$. Then we have*

$$E_j = \begin{cases} E_{j-1} & \textit{if there is no match point } (x, j) \textit{ s.t. } P^{Bh}[v, j{-}1] < x \leq E_{j-1}, \\ P[v{+}1, j] & \textit{otherwise.} \end{cases}$$

Proof. We begin with the first case (see Fig. 6). Since $(E_{j-1}, j-1)$ is the partition point in DP with score v, by the monotonicity of LCS we have $DP[P^{Bh}[v, j-$

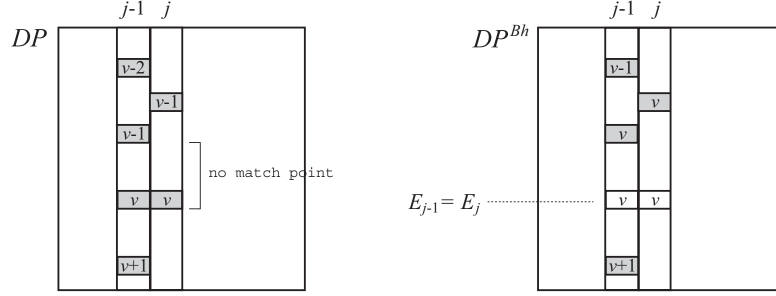

Fig. 6. $E_j = E_{j-1}$ if there is no match point between $P^{Bh}[v, j-1]$ and E_{j-1}.

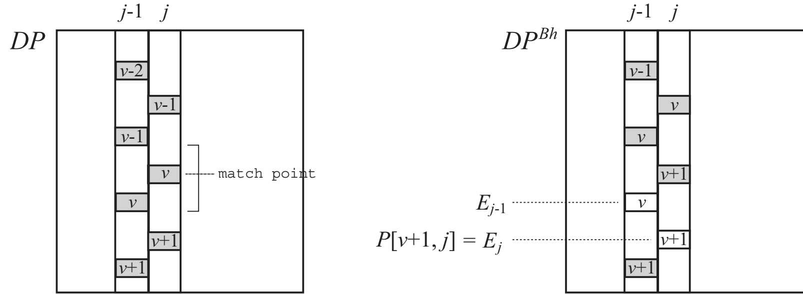

Fig. 7. $E_j = P[v+1, j]$ if there is a match point between $P^{Bh}[v, j-1]$ and E_{j-1}.

$1], j] = v-1$. Thus for any row index $P^{Bh}[v, j-1] \leq i < E_{j-1}$, $DP[i, j-1] = v-1$. By Lemma 4 $DP^{Bh}[i, j-1] = v$ for any such i. Recall $P^{Bh}[v, j] \leq P^{Bh}[v, j-1]$. Since there is no match point (x, j) such that $P^{Bh}[v, j-1] < x \leq E_{j-1}$, we get the three following properties:

- for any row index $P[v, j] \leq i' < E_j$, $DP[i', j] = v-1$,
- $P[v, j] = E_{j-1}$, and
- for any row index $P^{Bh}[v, j] \leq i'' \leq E_j$, $DP[i'', j] = v$,

which imply $E_j = E_{j-1}$.

Now we focus on the second case (see Fig. 7). Consider minimum row index x in range $P^{Bh}[v, j-1] < x \leq E_{j-1}$, such that (x, j) is a match point. Then we know that $P[v, j] = x$. By Lemmas 4 and 6, we have $DP^{Bh}[x, j] = v+1$. Hence $E_j = P[v+1, j]$ as it is no longer a partition point at column j of DP^{Bh}. □

Note that a spacial case occurs in Lemma 7 when v is the highest value at column j of DP, and therefore partition point $(P[v+1, j], j)$ does not exist. In this case, $P[v, j] = P^{Bh}[v+1, j]$ as usual, but no partition point is eliminated at column j. The update of P to P^{Bh} is stopped at this point, since Lemma 3 also stands for P^{Bh}.

The initial condition to determine the first column in which a partition point is eliminated, and in which row the partition point to be eliminated exists, is given in the following lemma.

Lemma 8. *Let* $\ell = \min\{j \mid A[j] = b\}$. *Then we have*

$$E_\ell = \begin{cases} null & \text{if there is no partition point at column } j \text{ in } DP, \\ P[1, \ell] & \text{otherwise.} \end{cases}$$

Proof. Trivial. □

In case $E_j = null$ in Lemma 8, there occurs no partition point elimination at the greater columns than ℓ, either.

Due to the above arguments, it is possible to update each column of P in constant time using the double-linked list implementation in Section 3. Since we have to update n columns in the worst case (For instance, consider $A = \mathtt{ba^n}$ and $B = \mathtt{b^m}$. Every time we add $b = \mathtt{b}$ to the head of B, n new partition points will be added, and n old partition points will be eliminated), we conclude that:

Theorem 2. *Provided that* $L = LCS(A, B)$ *is already computed, for any character* b, $LCS(A, bB)$ *is computable in* $O(n)$ *time.*

4.2 Computing $LCS(Aa, B)$

Let DP^{At} and P^{At} denote the DP matrix and the partition point set which we obtain in computing $LCS(Aa, B)$ with character a, respectively.

The following proposition is obvious.

Proposition 3. *For each partition point* $(P[v, j-1], j-1)$,

$$P[v, j] = \begin{cases} NM(P[v-1, j], A[j]) & \text{if } NM(P[v-1, j], A[j]) < P[v, j-1], \\ P[v, j-1] & \text{otherwise.} \end{cases}$$

It is clear that the scores of all the existing columns of DP are inherited to DP^{At} and thus we only need to compute the partition points in the last (new) column of P^{At}, which is computable based on Proposition 3. Therefore we obtain the following result.

Theorem 3. *Provided that* $L = LCS(A, B)$ *is already computed, for any character* a, $LCS(Aa, B)$ *is computable in* $O(L)$ *time.*

4.3 Computing $LCS(A, Bb)$

Let DP^{Bt} and P^{Bt} denote the DP matrix and the partition point set which we obtain in computing $LCS(A, Bb)$ with character b, respectively.

It is clear that in updating DP to DP^{Bt} the scores of all rows are preserved and thus we only need to examine whether or not the last (new) row becomes a new partition point at each column. Let $P[j]$ denote the set of the partition points at column j of DP. That is, $P[j]$ is a subset of P. Then we have the following proposition and theorem.

Proposition 4. *Let partition point* $\max(P[j-1])$ *have score* v*.*

- *If* $\max(P[j])$ *has score* $v+1$*, then the last row at column* j *is not in* P^{Bt}*.*
- *If* $\max(P[j])$ *has score* v*, then there are two sub-cases.*
  - *If the last row at column* j *of* DP^{Bt} *is a match point, then the last row at column* j *is in* P^{Bt} *with score* $v+1$*.*
  - *If the last row at column* j *of* DP^{Bt} *is not a match point, then there are two further sub-cases.*
    - *If* $\max(P^{Bt}[j-1])$ *has score* v*, then the last row at column* j *is not in* P^{Bt}*.*
    - *If* $\max(P^{Bt}[j-1])$ *has score* $v+1$*, then the last row at column* j *is in* P^{Bt} *with score* $v+1$*.*

Theorem 4. *Provided that* $L = LCS(A,B)$ *is already computed, for any character* b*,* $LCS(A,Bb)$ *is computable in* $O(n)$ *time.*

Proof. By Proposition 4 we can compute each partition point in the last row of DP^{Bt} in $O(1)$ time. Since there are n column indices at the last row of DP^{Bt}, it takes $O(n)$ time in total. □

4.4 Updating *NM* Table

The algorithms introduced in the last subsections use *NM* table. Recall that we construct *NM* table for alphabet Σ against string B. Thus, when a new character is added to the head or tail of B, *NM* table has to be updated accordingly. Let NM^{Bt} and NM^{Bh} denote the next match tables obtained by updating *NM* for $LCS(A,bB)$ and $LCS(A,Bb)$, respectively.

- computing NM^{Bh}.
  Let i be the position index of new character b added to the head of B. Then we have

$$NM^{Bh}[k,c] = \begin{cases} i & \text{if } k = i-1 \text{ and } c = b, \\ NM[k+1,c] & \text{if } k = i-1 \text{ and } c \neq b, \\ NM[k,c] & \text{otherwise.} \end{cases}$$

  This means that we only have to update the top row $i-1$ of NM^{Bh}. Since we have assumed that Σ is fixed, it takes $O(1)$ time.
- computing NM^{Bt}.
  Let i' be the position index of new character b appended to the tail of B. Also, let ℓ be the last occurrence of b in B. Then we have

$$NM^{Bt}[k,c] = \begin{cases} null & \text{if } k = i', \\ i' & \text{if } \ell \leq k < i' \text{ and } c = b, \\ NM[k,c] & \text{otherwise.} \end{cases}$$

Initializing row i' takes constant time as Σ is fixed. For row $\ell \leq k < i'$ at column b, in the worst case it takes linear time in the length of B. However, notice that once any entry is valued with a non-null position, its value will never change. Since the size of NM is linear in the length of B (once more recall Σ is fixed), the amortized time complexity for updating NM is $O(1)$.

In conclusion of this whole section, the following theorem stands.

Theorem 5. *Given strings A, B of length n, m respectively, and provided that $L = LCS(A, B)$ is already computed, we can compute, for any character a, b, $LCS(aA, B)$ in $O(L)$ time, $LCS(A, bB)$ in $O(n)$ time, $LCS(Aa, B)$ in $O(L)$ time, and $LCS(A, Bb)$ in $O(n)$ time. The total space complexity is $O(nL + m)$.*

References

1. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2002)
2. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
3. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In: Proc. 13th SIAM Symposium on Discrete Algorithms (SODA'02). (2002) 679–688
4. Hunt, J.W., Szymanski, T.G.: An algorithm for differential file comparison. Communications of the ACM **2** (1977) 417–439
5. Amir, A., Eisenberg, E., Porat, E.: Swap and mismatch edit distance. In: 12th Annual European Symposium on Algorithms (ESA'04). Volume 3221 of LNCS., Springer-Verlag (2004) 16–27
6. Wu, S., Manber, U.: Fast text searching allowing errors. Communications of the ACM **35** (1992) 83–91
7. Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research **25** (1997) 3389–3402
8. Li, M., Ma, B., Kisman, D., Tromp, J.: PatternHunter II: Highly sensitive and fast homology search. Journal of Bioinformatics and Computational Biology **2** (2004) 417–439
9. Apostolico, A.: String editing and longest common subsequences. In: Handbook of Formal Languages. Volume 2., Springer-Verlag (1997) 361–398
10. Landau, G.M., Myers, E., Ziv-Ukelson, M.: Two algorithms for LCS consecutive suffix alignment. In: Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM'04). Volume 3109 of LNCS., Springer-Verlag (2004) 173–193
11. Landau, G.M., Myers, E.W., Schmidt, J.P.: Incremental string comparison. SIAM Journal of Computing **27** (1998) 557–582
12. Kim, S.R., Park, K.: A dynamic edit distance table. In: Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00). Volume 1848 of LNCS., Springer-Verlag (2000) 60–68