Two Algorithms for LCS Consecutive Suffix Alignment

Gad M. Landau[1,2] *, Eugene Myers[3], and Michal Ziv-Ukelson[4] **

[1] Dept. of Computer Science, Haifa University, Haifa 31905, Israel
`landau@cs.haifa.ac.il`
[2] Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840, USA
`landau@poly.edu`
[3] Div. of Computer Science, UC Berkeley, Berkeley, CA 94720-1776, USA
`gene@eecs.berkeley.edu`
[4] Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel
`michalz@cs.technion.ac.il`

Abstract. The problem of aligning two sequences A and B to determine their similarity is one of the fundamental problems in pattern matching. A challenging, basic variation of the sequence similarity problem is the incremental string comparison problem, denoted **Consecutive Suffix Alignment**, which is, given two strings A and B, to compute the alignment solution of each suffix of A versus B.
Here, we present two solutions to the Consecutive Suffix Alignment Problem under the LCS metric. The first solution is an $O(nL)$ time and space algorithm for constant alphabets, where n is the size of the compared strings and $L \leq n$ denotes the size of the LCS of A and B.
The second solution is an $O(nL + n\log|\Sigma|)$ time and $O(L)$ space algorithm for general alphabets, where Σ denotes the alphabet of the compared strings. (Note that $|\Sigma| \leq n$.)

1 Introduction

The problem of comparing two sequences A of size n and B of size m to determine their similarity is one of the fundamental problems in pattern matching. Standard dynamic programming sequence comparison algorithms compute an $(m+1) \times (n+1)$ matrix DP, where entry $DP[i, j]$ is set to the best score for the problem of comparing A^i with B^j, and A^i is the prefix, $a_1, a_2, \ldots, a_i$ of A. However, there are various applications, such as Cyclic String Comparison [8,

* Research supported in part by NSF grant CCR-0104307, by NATO Science Programme grant PST.CLG.977017, by the Israel Science Foundation grant 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities, and by IBM Faculty Partnership Award.
** Research supported in part by the Aly Kaufman Post Doctoral Fellowship and by the Bar-Nir Bergreen Software Technology Center of Excellence.

14], Common Substring Alignment Encoding [9–11], Approximate Overlap for DNA Sequencing [8] and more, which require the computation of the solution for the comparison of B with progressively longer suffixes of A, as defined below.

Definition 1. *The* **Consecutive Suffix Alignment Problem** *is, given two strings A and B, to compute the alignment solution of each suffix of A versus B.*

By *solution* we mean some encoding of a relevant portion of the DP matrix computed in comparing A and B. As will be seen in detail later, the data-dependencies of the fundamental recurrence, used to compute an entry $DP[i,j]$, is such that it is easy to extend DP to a matrix DP' for B versus Aa by computing an additional column. However, efficiently computing a solution for B versus aA given DP is much more difficult, in essence requiring one to work against the "grain" of these data-dependencies. The further observation that the matrix for B versus A, and the matrix for B versus aA can differ in $O(n^2)$ entries suggests that the relationship between such adjacent problems is non-trivial. One might immediately suggest that by comparing the reverse of A and B, prepending symbols becomes equivalent to appending symbols, and so the problem, as stated, is trivial. But in this case, we would ask for the delivery of a solution for B versus Aa. To simplify matters, we will focus on the core problem of computing a solution for B versus aA, given a "forward" solution for B versus A. A "forward" solution of the problem contains an encoding of the comparison of all (relevant) prefixes of B with all (relevant) prefixes of A. It turns out that the ability to efficiently prepend a symbol to A when given all the information contained in a "forward" solution allows one to solve the applications mentioned above with greater asymptotic efficiency then heretofore possible.

There are known solutions to the Consecutive Suffix Alignment problem for various string comparison metrics. For the LCS and Levenshtein distance metrics, the best previously published algorithm [8] for incremental string comparison computes all suffix comparisons in $O(nk)$ time, provided the number of differences in the alignment is bounded by parameter k. When the number of differences in the best alignment is not bounded, one could use the $O(n(n+m))$ results for incremental Levenshtein distance computation described in [8, 7]. Schmidt [14] describes an $O(nm)$ incremental comparison algorithm for metrics whose scoring table values are restricted to the interval $[-S, M]$. Here, we will focus on incremental alignment algorithms for the LCS metric.

The simplest form of sequence alignment is the problem of computing the *Longest Common Subsequence* (LCS) between strings A and B [1]. A *subsequence* of a string is any string obtained by deleting zero or more symbols from the given string. A *Common Subsequence* of A and B is a subsequence of both, and an LCS is one of greatest length. Longest Common Subsequences have many applications, including sequence comparison in molecular biology as well as the widely used *diff* file comparison program. The LCS problem can be solved in $O(mn)$ time, where m and n are the lengths of strings A and B, using dynamic programming [5]. More efficient LCS algorithms, which are based on the observation that the LCS solution space is highly redundant, try to limit the computation only to those entries of the DP table which convey essential information, and exploit in various ways the *sparsity* inherent to the LCS problem. Sparsity allows us to relate algorithmic performances to parameters other than the lengths of the input strings. Most LCS algorithms that exploit sparsity have their natural predecessors in either Hirshberg [5] or Hunt-Szymanski [6]. All Sparse LCS algorithms are preceded by an $O(n \log |\Sigma|)$ preprocessing [1]. The Hirshberg algorithm uses $L = |LCS[A, B]|$ as a parameter, and achieves an $O(nL)$ complexity. The Hunt-Szymanski algorithm utilizes as parameter the number of matches between A and B, denoted r, and achieves an $O(r \log L)$ complexity. Apostolico and Guerra [2] achieve an $O(L \cdot m \cdot \min(\log |\Sigma|, \log m, \log(2n/m))$ algorithm, where $m \leq n$ denotes the size of the shortest string among A and B, and another $O(m \log n + d \log(nm/d))$ algorithm, where $d \leq r$ is the number of dominant matches (as defined by Hirschberg [5]). This algorithm can also be implemented in time $O(d \log \log \min(d, nm/d))$ [4]. Note that in the worst case both d and r are $\Omega(n^2)$, while L is always bounded by n.

Note that the algorithms mentioned in the above paragraph compute the LCS between two strings A and B, however the objective of this paper is to compute all LCS solutions for each of the n suffixes of A versus B, according to Definition 1.

1.1 Results

In this paper we present two solutions to the Consecutive Suffix Alignment Problem under the LCS metric. The first solution (Section 3) is an $O(nL)$ time and space algorithm for constant alphabets, where n is the size of A, m is the size of B and $L \leq n$ denotes the size of the LCS of A and B. This algorithm computes a representation of the Dynamic Programming matrix for the alignment of each suffix of A with B.

The second solution (Section 4) is an $O(nL + n \log |\Sigma|)$ time, $O(L)$ space incremental algorithm for general alphabets, that computes the comparison solutions to $O(n)$ "consecutive" problems in the same asymptotic time as its standard counterpart [5] solves a single problem. This algorithm computes a representation of the last row of each of the Dynamic Programming matrices that are computed during the alignment of each suffix of A with B.

Both algorithms are extremely simple and practical, and use the most naive data structures.

Note that, due to lack of space, all proofs are omitted. A full version of the paper, including proofs to all lemmas, can be found in:

$$http://www.cs.technion.ac.il/\sim michalz/lcscsa.pdf$$

2 Preliminaries

An *LCS graph* [14] for A and B is a directed, acyclic, weighted graph containing $(|A|+1)(|B|+1)$ nodes, each labeled with a distinct pair $(x, y)(0 \leq x \leq |A|, 0 \leq$

$y \leq |B|$). The nodes are organized in a matrix of $(|A|+1)$ rows and $(|B|+1)$ columns. An index pair (x, y) in the graph where $A[x] = B[y]$ is called a *match*. The LCS graph contains a directed edge with a weight of zero from each node (x, y) to each of the nodes $(x, y+1)$, $(x+1, y)$. Node (x, y) will contain a diagonal edge with a weight of one to node $(x+1, y+1)$, if $(x+1, y+1)$ is a match.

Maximal-score paths in the LCS graph represent optimal alignments of A and B, and can be computed in $O(n^2)$ time and space complexity using dynamic programming. Alternatively, the LCS graph of A versus B can be viewed as a sparse graph of matches, and the alignment problem as that of finding highest scoring paths in a sparse graph of matches. Therefore, paths in the LCS Graph can be viewed as chains of matches.

Definition 2. *A* **k-sized chain** *is a path of score k in the LCS graph, going through a sequence of k matches $(x_1, y_1)(x_2, y_2)\ldots(x_k, y_k)$, such that $x_j < x_{j+1}$ and $y_j < y_{j+1}$ for successive matches (x_j, y_j) and (x_{j+1}, y_{j+1}).*

Both algorithms suggested in this paper will execute a series of n iterations numbered from n down to 1. At each iteration, an increased sized suffix of string A will be compared with the full string B. Increasing the suffix of A by one character corresponds to the extension of the LCS graph to the left by adding one column. Therefore, we define the growing LCS graph in terms of generations, as follows (see Figure 2).

Definition 3. Generation k *(G_k for short) denotes the LCS graph for comparing B with A_k^n. Correspondingly, L_k denotes LCS[B, A_k^n], and reflects the size of the longest chain in G_k.*

We define two data structures, to be constructed during a preprocessing stage, that will be used by the consecutive suffix alignment algorithms for the incremental construction and navigation of the representation of the LCS graph for each generation (see Figure 1).

Definition 4. *$MatchList(j)$ stores the list of indices of match points in column j of DP, sorted in increasing row index order.*

$MatchLists$ can be computed in $O(n \log |\Sigma|)$ preprocessing time.

Definition 5. *$NextMatch(i, A[j])$ denotes a function which returns the index of the next match point in column j of DP with row index greater than i, if such a match point exists. If no such match point exists, the function returns $NULL$.*

A $NextMatch[i, \alpha]$ table, for all $\alpha \in \Sigma$, can be constructed in $O(n|\Sigma|)$ time and space. When the alphabet is constant, a $NextMatch$ table can be constructed in $O(n)$ time and space.

3 The First Algorithm

The first algorithm consists of a preprocessing stage that is followed by a main stage. During the preprocessing stage, the $NextMatch$ table for strings A and B is constructed.

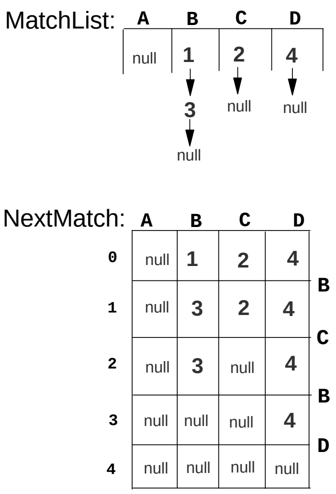

Fig. 1. The MatchList and NextMatch data structures.

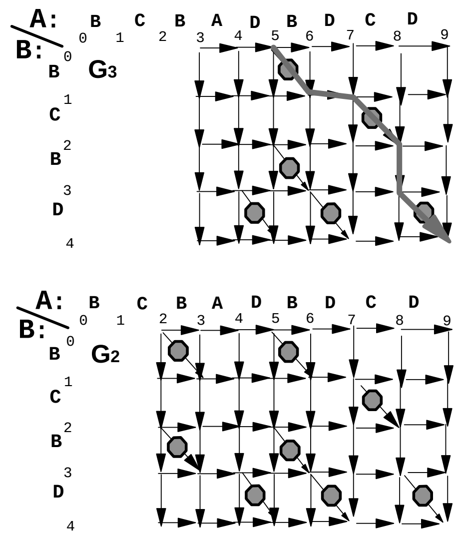

Fig. 2. The LCS Graphs G_3 and G_2 for the comparison of strings A = "$BCBADBCDC$" versus B = "$BCBD$". Grey octagons denote match points. A chain of size 3 is demonstrated in G_3, corresponding to the common subsequence "BCD".

During the main stage, the first algorithm will interpret the LCS graph for each generation as a dynamic programming graph, where node $[i, j]$ in G_k stores the value of the longest chain from the upper, leftmost corner of the graph up to node $[i, j]$. Therefore, we will formally define the graph which corresponds to each generation, as follows (see Figure 3).

Definition 6. *DP^k denotes the dynamic programming table for comparing string B with string A_k^n, such that $DP^k[i, j]$, for $i = 1 \ldots m$, $j = k \ldots m$, stores $LCS[B_1^i, A_k^j]$. DP^k corresponds to G^k as follows. $DP^k[i, j] = v$ if v is the size of the longest chain that starts in some match in the upper, left corner of G^k and ends in some match with row index $\leq i$ and column index $\leq j$.*

Using Definition 5, the objective of the first Consecutive Alignments algorithm could be formally defined as follows: **compute DP^k for each $k \in [1, n]$.**

Applying the dynamic programming algorithm to each of the n problems gives an $O(n^3)$ algorithm. It would be better if one could improve efficiency by incrementally computing DP^k from either DP^{k-1} or DP^{k+1}. At first glance this appears impossible, since computing DP^k from DP^{k+1} may require recomputing every value. Thus, attempting to compute the DP table for each problem incrementally appears doomed because the *absolute* value of as many as $O(n^2)$ elements can change between successive problem instances. However, based on the observation that each column of the LCS DP is a monotone staircase with

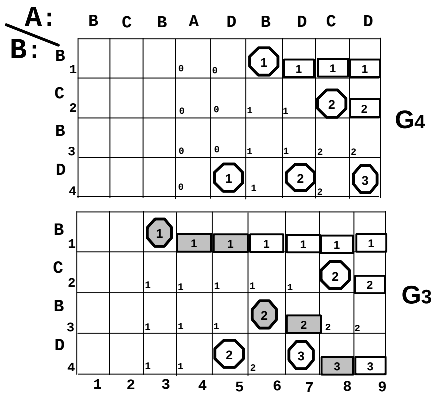

Fig. 3. The update operations applied by the first Consecutive Alignments algorithm, during the computation of the partition points of generation G_3 from the partition points of generation G_4. Partition points are indicated as rectangles and hexagons, and the numbers inside stand for their value. The hexagons represent partition points that are both partition points and match points. The grey rectangles and hexagons represent the new partition points in generation G_3.

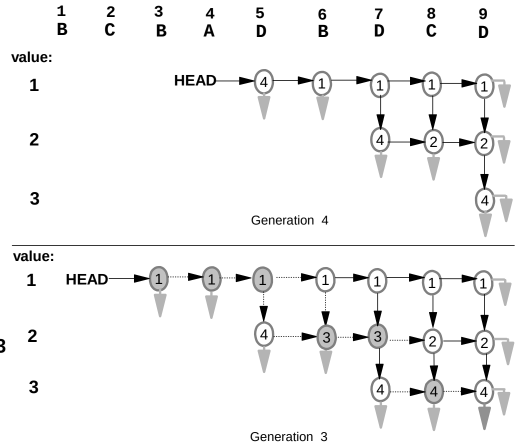

Fig. 4. The implementation of the partition-point data structure as a doubly-linked list. The grey circles represent the new partition points in generation G_3.

unit-steps, we will apply *partition encoding* [5] to the DP table, and represent each column of DP^k by its $O(L)$ partition points (steps), defined as follows (see Figure 4).

Definition 7. *P^k denotes the set of partition points of DP^k, where partition point $P^k[j, v]$, for $k = 1 \ldots n, j = k \ldots n, v = 0 \ldots L_k$, denotes the first entry in column j of DP^k which bears the value of v.*

In terms of chains in the LCS graph, $P^k[j, v] = i$ if i is the lowest row index to end a chain that is contained in the first j columns of G_k. It is now clear that instead of computing DP^k it suffices to compute P^k for $k = n \ldots 1$.

3.1 Computing P^k from P^{k+1}

The consecutive alignments algorithm consists of n stages. The LCS graph for comparing strings B and A is grown from right to left in n stages. At stage k, column k is appended to the considered LCS graph. Correspondingly, P^k is obtained by inheriting the partition points of P^{k+1} and updating them as follows. The first, newly appended column of P^k has only one partition point - which is the first match point $[i, k]$ in column k (see column 3 of G_3 in Figure 3). This match point corresponds to a chain of size 1, and indicates the index i such that all entries in column k of DP^k of row index smaller than i are zero, and all entries from index i and up are one. Therefore, stage k of the algorithm starts by computing, creating and appending the one partition point, which corresponds to the newly appended column k, to the partition points of P^k.

Then, the columns inherited from P^{k+1} are traversed in a left-to-right order, and updated with new partition points.

We will use two important observations in simplifying the update process. First, in each traversed column of P^k, at most one additional partition point is inserted, as will be shown in Lemma 2. We will show how to efficiently compute this new partition point. The second observation, which will be asserted in Conclusion 1, is that once the leftmost column j is encountered, such that no new partition point is inserted to column j of P^k, the update work for stage k of the algorithm is complete. Therefore, the algorithm will quit the column traversal and exit stage k when it hits the first, leftmost column j in P^k that is identical to column j of P^{k+1}.

The incremental approach applied in the first algorithm is based in the following lemma, which analyzes the differences in a given column from one generation of DP to the next.

Lemma 1. *Column j of DP^k is column j of DP^{k+1} except that all elements that start in some row I_j are greater by one. Formally, for column j of DP^k there is an index I_j such that $DP^k[i,j] = DP^{k+1}[i,j]$ for $i < I_j$ and $DP^k[i,j] = DP^{k+1}[i,j] + 1$ for $i \geq I_j$.*

The next Lemma immediately follows.

Lemma 2. *Column j in P^k consists of all the partition points which appear in column j of P^{k+1}, plus at most one new partition point. The new partition point is the smallest row index I_j, such that $delta[I_j] = DP^k[I_j,j] - DP^{k+1}[I_j,j] = 1$.*

Claim 3. *For any two rectangles in a DP table, given that the values of the entries in vertices in the upper and left border of the rectangles are the same and that the underlying LCS subgraphs for the rectangles are identical - the internal values of entries in the rectangles will be the same. Furthermore, adding a constant c to each entry of the left and top borders of a given rectangle in the DP table would result in an increment by c of the value of each entry internal to the rectangle.*

Conclusion 1: *If column j of DP^k is identical to column j of DP^{k+1}, then all columns greater than j of DP^k are also identical to the corresponding columns of DP^{k+1}.*

The correctness of Conclusion 1 is immediate from Claim 3. Given that the structure of the LCS graph in column $j+1$ does not change from DP^{k+1} to DP^k, that the value of the first entry in the column remains zero, and that all values in its left border (column j of DP^k) remain the same as in DP^{k+1}, it is clear that the dynamic programming algorithm will compute the exact same values for column $j+1$ of DP^{k+1} and for column $j+1$ of DP^k. The same claim follows inductively when computing the values of column $j+2$ of DP^k from the values of column $j+1$, and so on.

The suggested algorithm will traverse the columns of P^k from left to right. In each of the traversed columns it will either insert a new partition point or halt according to Conclusion 1.

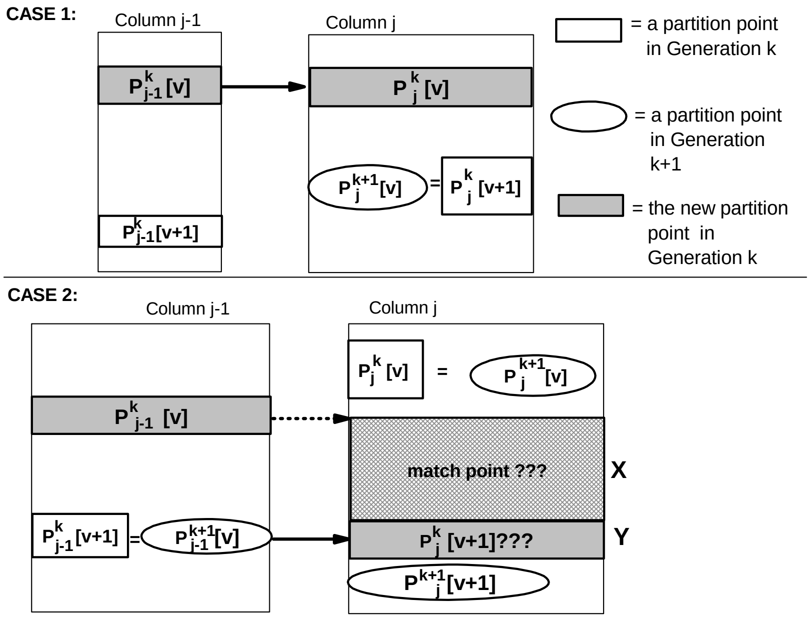

Fig. 5. The three possible scenarios to be considered when computing the new partition point of column j in generation G_k.

3.2 Computing the New Partition Points of Column j of P^k

In this section we will show how to compute the new partition points of any column $j > k$ of P^k, using the partition points of column j of P^{k+1}, the partition points of column $j-1$ of P^k, and the match points of column j of P^k. We start by constraining the range of row indices of column j in which the new partition point will be searched.

Lemma 3. *Let $I_{j-1} = P^k_{j-1}[v]$ denote the new partition point in column $j-1$ of P^k, and let I_j denote the index of the new partition point in column j of P^k. $P^k_{j-1}[v] \leq I_j \leq P^k_{j-1}[v+1]$.*

We will next show that there are two possible cases to consider when computing the new partition point of column j, as specified in the lemma below (see Figure 5).

Lemma 4. *Let $I_{j-1} = P^k_{j-1}[v]$ denote the row index of the new partition point in column $j-1$ of P^k. Let I_j denote the row index of the new partition point in column j of P^k. I_j can assume one of two values, according to the following two cases.*

case 1. $I_{j-1} = P^k_{j-1}[v] \leq P^{k+1}_j[v]$, *in which case* $I_j = P^k_j[v] = I_{j-1}$.
case 2. $I_{j-1} = P^k_{j-1}[v] > P^{k+1}_j[v]$, *in which case*

$$I_j = P^k_j[v+1] = min\{P^k_{j-1}[v+1], NextMatch(I_{j-1} = P^k_{j-1}[v], j)\}$$

.

Conclusion 2: *At each of the columns traversed by the algorithm, during the computation of P^k from the partition points of P^{k+1}, except for the last column that is considered for update, a single partition point is inserted. As for the last column considered in generation G_k, the algorithm quits the update of P^k, following Conclusion 1, upon realizing that there is no partition point to insert to this column, and it is therefore similar to the previous column.*

Conclusion 3: *The new partition point in column j of P^k, if such exists, is one of four options:*

1. *The new partition point of column $j-1$.*
2. *The partition point that immediately follows the new partition point of column $j-1$.*
3. *Some match point at an index that falls between the new partition point of column $j-1$ and the match point that immediately follows in column j.*
4. *Some match point at an index that falls between the last partition point of column $j-1$ and index $m+1$.*

3.3 An efficient Implementation of the First Algorithm

An efficient algorithm for the consecutive suffix alignments problem requires a data structure modelling the current partition that can be quickly updated in accordance with Lemma 4. To insert new partition points in $O(1)$ time suggests modelling each column partition with a singly-linked list of partition points. However, it is also desired that successive insertion locations be accessed in $O(1)$ time. Fortunately, by Conclusion 3, the update position in the current column is either the update position in the previous column or one greater than this position, and the update position in the first column in each generation is the first position. Thus, it suffices to add a pointer from the i-th cell in a column partition to the i-th cell in the next column (see Figure 4). Therefore, each cell in the mesh which represents the partition points of a given generation is annotated with its index, as well as with two pointers, one pointing to the next partition point in the same column and the other set to the cell for the partition point of the same value in the next column. Furthermore, it is easy to show, following Lemma 4, that the pointer updates which result from each new partition-point insertion can be correctly completed in $O(1)$ time.

Time and Space Complexity of the First Algorithm.

During the preprocessing stage, the $NextMatch$ table for strings A and B is constructed in $O(n|\Sigma|)$ time and space.

By conclusion 2, the number of times the algorithm needs to compute and insert a new partition point is linear with the final number of partition points in P^1. Given the $NextMatch$ table which was prepared in the preprocessing stage, the computation of the next partition point, according to Lemma 4, can be executed in constant time. Navigation and insertion of a new partition point can also be done in constant time according to Conclusion 3 (see Figure 4).

This yields an $O(nL)$ time and space complexity algorithm for constant alphabets.

4 The Second Algorithm

The second algorithm takes advantage of the fact that many of the Consecutive Suffix Alignment applications we have in mind, such as Cyclic String Comparison [8, 14], Common Substring Alignment Encoding [9–11], Approximate Overlap for DNA Sequencing [8] and more, actually require the computation of the last row of the LCS graph for the comparison of each suffix of A with B. Therefore, the objective of the second algorithm is to compute the partition encoding of the last row of the LCS graph for each generation. This allows to compress the space requirement to $O(L)$. Similarly to the first algorithm, the second algorithm also consists of a preprocessing stage and a main stage. This second algorithm performs better than the first algorithm when the alphabet size is not constant. This advantage is achieved by a main stage that allows the replacement of the $NextMatch$ table with a $MatchList$ data structure (see Figure 1). The $MatchList$ for strings A and B is constructed during the preprocessing stage.

4.1 An $O(L_k)$ Size $TAILS$ Encoding of the Solution for G_k

In this section we will examine the solution that is constructed from all the partition-point encodings of the last rows of DP^k, for $k = n \dots 1$. We will apply some definitions and point out some observations which lead to the conclusion that the changes in the encoded solution, from one generation to the next, are constant. The main output of the second algorithm will be a table, denoted $TAILS$, that is defined as follows.

Definition 8. *$TAILS[k, j]$ is the column index of the j-th partition point in the last row of G_k. In other words, $TAILS[k, j] = t$ if t is the smallest column index such that $LCS[B, A_k^t] = j$.*

Correspondingly, the term *tail* is defined as follows.

Definition 9. *Let t denote the value at entry j of row k of $TAILS$.*

1. *t is considered a **tail** in generation G_k (see Figures 6, 7).*
2. *The **value** of tail t in generation G_k, denoted val_t, is j. That is, $LCS[A_k^t, B] = j$.*

It is easy to see that, in a given generation, tails are ordered in left to right *column* order and increasing *size*.

In the next lemma we analyze the changes in the set of values from row $k+1$ to row k of $TAILS$, and show that this change is $O(1)$.

Lemma 5. *If column k of the LCS graph contains at least one match, then the following changes are observed when comparing row $k+1$ of $TAILS$ to row k of $TAILS$:*

1. *$TAILS[k, 1] = k$.*
2. *All other entries from row $k+1$ are inherited by row k, except for at most one entry which could be lost:*

Case 1. *All entries are passed from row $k+1$ to row k of tails and shifted by one index to the right. In this case* $LCS[B, A_k^n] = LCS[B, A_{k+1}^n] + 1$.

Case 2. *One entry value, which appeared in row $k+1$ disappears in row k. In this case* $LCS[B, A_k^n] = LCS[B, A_{k+1}^n]$.

- *All values from row $k+1$ of $TAILS$ up to the disappearing entry are shifted by one index to the right in row k of $TAILS$.*
- *All values from row $k+1$ of $TAILS$ which are greater than the disappearing entry remain intact in row k of $TAILS$.*

From the above lemma we conclude that, in order to compute row k of $TAILS$, it is sufficient to find out whether or not column k of G contains at least one match point, and if so to compute the entry which disappeared from row $k+1$ of $TAILS$. Hence, from now on the algorithm will focus only on columns where there is at least one match point and on discovering, for these columns, which entry (if at all) disappears in the corresponding row of $TAILS$.

From now on we will focus on the work necessary for updating the set of L_k values from row $k+1$ of $TAILS$ to row k of $TAILS$. Therefore, we simplify the notation to focus on the L_k values in row k of $TAILS$. We note that these L_k values denote column indices of leftmost-ending chains of sizes $1 \dots L_k$ in G_k. We will refer to these values from now on as the set of *tails* of generation G_k.

4.2 The $O(L^2)$ Active Chains in a Given Generation

In this section we will describe the new data structure which is the core of our algorithm. Note that $TAILS[k, j] = t$ if t is the index of the smallest column index to end a j-sized chain in G_k. So, in essence, in iteration k of the algorithm we seek all leftmost-ending chains of sizes $1 \dots L_k$ in the LCS graph G_k.

Recall that, in addition to the output computation for G_k, we have to prepare the relevant information for the output computation in future generations. Therefore, in addition to the $O(L_k)$ leftmost ending chains we also wish to keep track of chains which have the potential to become leftmost chains in some future generation. Note that a leftmost chain of size j in a given generation does not necessarily evolve from a leftmost chain of size $j-1$ in some previous generation (see Figure 6). This fact brings up the need to carefully define the minimal set of chains which need to be maintained as candidates to become leftmost chains in some future generation.

By definition, each chain starts in a match (the match of smallest row index and leftmost column index in the chain) and ends in a match. At this stage it is already clear that an earlier (left) last-match is an advantage in a chain, according to the tail definition. It is quite intuitive that a lower first-match is an advantage as well, since it will be easier to extend it by matches in future columns. Hence, a chain of size k is redundant if there exists another chain of size k that starts lower and ends to its left. Therefore, we will maintain as candidates for expansion only the non-redundant chains, defined as follows.

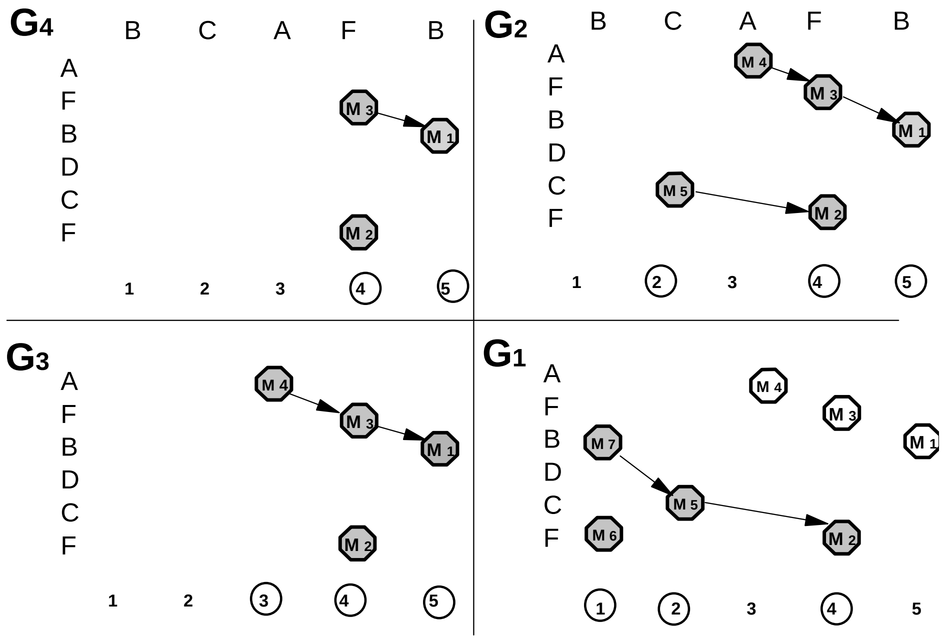

Fig. 6. The evolution of leftmost chains from chains that are not necessarily leftmost. For each generation, the dark circles around column indices directly below the bottom row of the graph mark active tails.

Definition 10. *A chain c_1 of size j is an* **active chain** *in generation G_k, if there does not exist another chain c_2 of size j in G_k, such that both conditions below hold:*

1. *c_2 starts lower than c_1.*
2. *c_2 ends earlier than c_1.*

For the purpose of tail computation, it is sufficient to maintain the row index of the first match in each chain and the column index of the last match in each chain.
-The row number of the first match in an active chain is denoted a *head.*
-The column index of a last match in an active chain is denoted an *end-point.*

Note that two or more different chains could share the same head in a given generation. For example, match m_7, corresponding to a head of row index 3, is the origin of active chains of sizes $2-3$ in generation G_1 of Figure 6. Based on this observation, we decided to count the number of different matches which serve as heads and end-points in a given generation. To our surprise, we discovered that in a given generation G_k, the number of distinct heads is only L_k (see Conclusion 4), and the number of distinct end-points in G_k is only L_k (see Lemma 6 which comes up next). This observation is the key to the efficient state encoding in our algorithm.

Lemma 6. *Each active chain ends in a tail.*

We have shown in Lemma 6 that each end-point is a tail. Therefore, from now on we will use the term *tail* when referring to end-points of active chains. We consider two active chains of identical sizes which have the same head and the same tail as one.

4.3 An $O(L_k)$ $HEADS$ Representation of the State Information for G_k

In this section we will show that the number of distinct heads in generation G_k is exactly L_k. In order to count the distinct heads, we associate with each tail a set of relevant heads, as follows.

Definition 11. *H_t denotes the set of heads of active chains that end in tail t.*

The active heads in G_k are counted as follows. The tails are traversed left to right, in increasing size and index, and the new heads contributed by each tail are noted (a head h is contributed by tail t if $h \in H_t$ and $h \notin H_{t_1}$ for any $t_1 < t$). The counting of the active heads which are contributed by each tail t will be based on the following two observed properties of H_t. These properties, given below, will be proven in the rest of this section.

Property 1 of H_t. Let j denote the size of the smallest chain in H_t. The chains headed by H_t form a consecutive span, ordered by increasing head height and increasing chain size, starting with the lowest head which is the origin of the j-chain of H_t, and ending with the highest head which is the origin of the val_t-chain which ends in t (see Figure 7).

Property 2 of H_t. The head which is the origin of the smallest chain (size j) of H_t is the one and only new head in H_t. All other heads are included in H_{t_1} for some $t_1 < t$.

The following Lemmas 7 to 9 formally assert the two observed properties of H_t.

Lemma 7. *The heads of H_t are ordered in increasing height and increasing chain size.*

Lemma 8. *For any tail t, the sizes of active chains which correspond to the heads in H_t form a consecutive span.*

Lemma 9. *The head h_1 of the smallest chain in H_t is new. That is, there is no active chain that originates in h_1 and ends in some tail to the left of t.*

Conclusion 4: *From Lemmas 8 and 9 we conclude that as we scan the tails for generation G_k from left to right, each tail contributes exactly one new head to the expanding list of active heads. Therefore, there are exactly L_k different row indices which serve as active heads (to one or more active chains) in generation G_k.*

The new head that is contributed by each tail t is a key value which will represent the full H_t set for tail t in our algorithm, and therefore it is formally defined and named below.

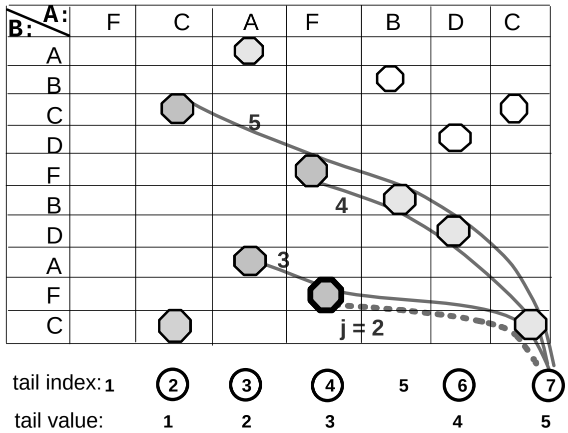

Fig. 7. The set of chains H_7 for the tail with value 5 and index 7 in generation G_2 of the consecutive suffix alignment of strings $A =$ "$BCBADBCDC$" versus $B =$ "$BCBD$". The head which is new_7 is highlighted with a thicker border, and the corresponding shortest chain of size 2 is dotted. The dark circles around column indices directly below the bottom row of the graph mark the active tails in G_2.

Definition 12. *new_t is the head of the smallest chain in H_t.*

We have found one more property of H_t which will be relevant to our algorithm, as proven in the next lemma.

Lemma 10. *H_t includes all heads that are higher than new_t and start at least one active chain which ends in some tail $t_3 < t$.*

Up till now we have analyzed the set H_t of heads that start all active chains which end in tail t. Next, we will symmetrically analyze the set of tails that end chains which originate in a given active head h.

We associate with each head a set of relevant tails, as follows.

Definition 13. *T_h denotes the set of tails of active chains that start in head h.*

Lemma 11. *For any head h, the sizes of active chains which correspond to the tails in T_h form a consecutive span.*

4.4 Changes in $HEADS$ and $TAILS$ from One Generation to the Next

In this section we discuss the changes in the sets of active heads and tails as the poset of matches for generation G_{k+1} is extended with the matches of column k. Following the update, some changes are observed in the set of active heads, in the set of active tails, and in the head-to-tail correspondence which was analyzed in the previous section. (When we say head-to-tail correspondence, we mean the pairing of head h_i with a tail t_i such that $h_i = new_{t_i}$).

Throughout the algorithm, the relevant state information will be represented by a dynamic list $HEADS$ of active heads, which is modified in each generation G_k, based on the match points in column k of DP.

Definition 14. *$HEADS_k$ denotes the set of active heads in generation G_k, maintained as a list which is sorted in increasing height (decreasing row index). Each head $h_{first} \in HEADS_k$ is annotated with two values. One is its height, and the second is the tail t such that $h_{first} = new_t$.*

In iteration k of the algorithm, two objectives will be addressed.

1. The first and main objective is to compute the tail that dies in G_k. In Lemma 12 we will show that, for any tail t that was active in G_{k+1}, the size of H_t can only decrease by one in G_k. Therefore, the tail to disappear in G_k is the tail t such that the size of H_t decreases from one to zero in G_k.
2. The second objective is to update the state information ($HEADS_{k+1}$ list) so it is ready for the upcoming computations of G_{k-1} ($HEADS_k$ list).

In this section we will show that both of the objectives above can be achieved by first merging $HEADS_{k+1}$ with the heights of matches in column k, and then traversing the list of heads once, in a bottom-up order, and modifying, in constant time, the head-to-tail association between active tails and their *new* head representative, if such an association indeed changes in G_k. (That is, if a given head h was new_t for some tail t in G_{k+1} and h is no longer new_t in G_k).

Lemma 12. *From one generation to the next, the number of active heads in H_t can only decrease by one. Furthermore, of all the chains that start in some head in H_t and end in t, only the shortest chain, the one headed by new_t in G_{k+1}, could be de-activated in G_k without being replaced by a lower head of a similar-sized active chain to t.*

From this we conclude that the tail to disappear from row k of $TAILS$ is the tail t such that the number of heads in H_t went down from one to zero in generation G_k. It remains to show how this dying tail t can be identified during the single, bottom up traversal of the list $HEADS_{k+1}$, following the merge with the matches of column k.

We are now ready to address the merging of the match points from column k with $HEADS_{k+1}$. The discussion of how the matches that are merged with $HEADS_{k+1}$ affect its transformation into $HEADS_k$ will be partitioned into four cases. First we discuss the first (lowest) match and the heads which fall below it. We then explain what happens to two consecutive matches with no head in between. The third case deals with the matches above the highest head in $HEADS_{k+1}$, if such exist. The fourth and main part of our discussion, deals with the changes to the "slice" of heads in $HEADS_{k+1}$ which fall either between two consecutive new matches in column k, or above the highest match in column k.

Case 1: The lowest match in column k. The first match in column k is a new head. It is the first chain, of size 1, of the tail k, and therefore is new_k. All heads below this match are unaffected, since no new chain that starts lower than these heads could have possibly been created in G_k.

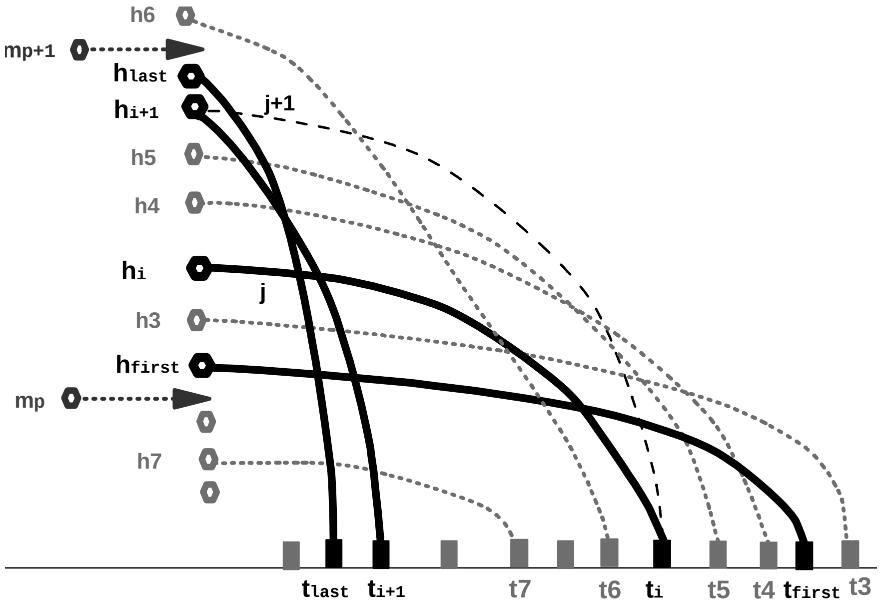

Fig. 8. The $HEADS$ list traversal in increasing height during its update with the match points of column k in generation G_k. The modified heads, as well as their corresponding tails and *new* chains are highlighted in black.

Case 2: Two consecutive matches with no heads in between. For any sequence of consecutive matches in column k with no head in between, all match points, except for the lowest match point in the sequence, are redundant.

Case 3. Matches above the highest head in $HEADS_{k+1}$. The lowest match in column k which is above the highest active head in $HEADS_{k+1}$, if such a match exists, becomes a new head. Consider the longest chain in G_{k+1}, of size L_{k+1}, that ends in tail L_{k+1}. Clearly, this chain's head is the highest head in the list. This chain will be extended by the new match to a lowest, leftmost $L_k = L_{k+1} + 1$ chain, and therefore this match is a new head.

Case 4. The series of heads that fall between two consecutive matches. This case, which includes the series of remaining heads above the highest match in column k, is the most complex case and covers the identification of the disappearing tail. It will therefore be discussed in detail in the next subsection.

4.5 Heads that Fall Between Two Matchpoints in Column k

Throughout this subsection we will use the following notation, as demonstrated in Figure 8.

- Let m_p, m_{p+1} denote two consecutive matches in column k, such that m_{p+1} is higher than m_p.

- Let h_{first} denote the first head in $HEADS_{k+1}$ which is higher than or equal to m_p and lower than m_{p+1}.
- Let $UPDATED_{m_{p,k}}$ denote the series of heads $h_i = h_{first} \dots h_{last}$ which fall between m_p and m_{p+1} and whose head-to-tail association changed in generation k, ordered in increasing height. Let $t_i = t_{first} \dots t_{last}$ denote the series of corresponding tails.
- Let h_{i+1} and h_i denote any pair of consecutive heads in $UPDATED_{m_{p,k}}$.

Consider the set of heads from $HEADS_{k+1}$ that fall between m_p and m_{p+1}. In this subsection we will show that some of the heads in this set remain unmodified in $HEADS_{k+1}$ (see, for example, heads h_3, h_4 and h_5 in Figure 8) while others change (see, heads h_{first}, h_i, h_{i+1} and h_{last} in Figure 8).

Lemma 14 shows that all heads in $HEADS_{k+1}$ that fall between m_p and m_{p+1} and are not in $UPDATED_{m_{p,k}}$ remain unchanged in G_k. Lemma 13 claims that $h_{first} \in UPDATED_{m_{p,k}}$, since it dies in generation G_k and is replaced with m_p. In Conclusion 8 we assert that all heads in $UPDATED_{m_{p,k}}$, except for h_{first}, remain active in G_k and survive the transformation of $HEADS_{k+1}$ to $HEADS_k$. Additional characteristics of the heads in the set $UPDATED_{m_{p,k}}$ are then investigated. In Lemma 14 we learn that the heads in $UPDATED_{m_p,k}$ and their corresponding tails form a series of increasing head heights and decreasing tail column indices, such that the *new* chains from any two consecutive heads in the series must cross. (See the dark chains in Figure 8). The challenge of modifying the head-to-tail association of the heads in $UPDATED_{m_{p,k}}$ is addressed in Lemma 16, and an interesting chain-reaction is observed. We show that, for any two consecutive heads $h_i, h_{i+1} \in UPDATED_{m_{p,k}}$, head h_{i+1} replaces h_i as the new new_{t_i} in G_k.

Next, we consider the tails that are active in G_{k+1} in order to try and find the tail which becomes extinct in G_k. Clearly, for some of these tails (see, for example t_6 in Figure 8), the corresponding *new* head falls above m_{p+1} and therefore the decision as to whether or not they survive the transition to G_k is delayed till later when the corresponding span is traversed and analyzed. For others (see, for example t_7 in Figure 8), the corresponding *new* head falls below m_p and therefore it has already been treated during the analysis of some previous span. For some tails (such as t_3, t_4 and t_5), the corresponding *new* heads indeed fall between m_p and m_{p+1} but are not included in $UPDATED_{m_{p,k}}$, and therefore these tails keep their shortest chain as is, and will also survive the transition to G_k. In Conclusion 8 we assert that the tails which correspond to all heads in $UPDATED_{m_{p,k}}$, except for t_{last}, are kept alive in G_k. As for t_{last}, this is the only candidate for extinction in G_k, and in Lemma 17 we show that in the last span of traversed heads, if the highest match in column k falls below the highest head in $HEADS_{k+1}$, then t_{last} of this last span will finally be identified as the dying tail in G_k.

Lemma 13. *h_{first} is no longer an active head in G_k. Instead, the height of match m_p replaces h_{first} in $HEADS_k$.*

We have shown that h_{first} dies in generation G_k and is replaced with the height of m_p in $HEADS_k$. From this we conclude that h_{first} is the first head in

$UPDATED_{m_{p,k}}$. The next lemma will help further separate the heads which participate in $UPDATED_{m_{p,k}}$ from the heads that remain unmodified from $HEADS_{k+1}$ to $HEADS_k$.

Lemma 14. *Consider two heads $h_1, h_2 \in HEADS_{k+1}$, such that h_2 is higher than h_1, and given that there is no match point in column k which falls between h_1 and h_2. Let $h_1 = new_{t_1}$ and $h_2 = new_{t_2}$. If $t_1 < t_2$, then the chain from h_2 to t_2 remains active in G_k.*

The above Lemma immediately leads to the following conclusion.

Conclusion 5 *The heads in $UPDATED_{m_p,k}$ and their corresponding tails form a series of increasing head heights and decreasing tail column indices.*

We have shown that all heads which are not in $UPDATED_{m_p,k}$ remain unmodified. We will next show that all heads in $UPDATED_{m_p,k}$, even though modified, survive the transformation from $HEADS_{k+1}$ to $HEADS_k$. In order to do so we first prove the following two lemmas, which lead to the conclusion that the modifications in the head-to-tail associations of heads in $UPDATED_{m_p,k}$ consist of a chain reaction in which each head is re-labelled with the tail of the head below.

Lemma 15. *Let h_i and h_{i+1} denote two heads in $HEADS_{k+1}$, such that $h_i = new_{t_i}$, $h_{i+1} = new_{t_{i+1}}$, and h_{i+1} is the first head above h_i such that its corresponding new tail t_{i+1} falls to the left of t_i. Let j denote the size of the chain from h_i to t_i. If the chain from h_i to t_i becomes de-activated in G_k, then all chains that originate in h_{i+1} and are of sizes smaller than or equal to j will be de-activated in G_k.*

The following conclusion is immediate from the above Lemma 15 and the definition of $UPDATED_{m_p,k}$.

Conclusion 6. *Let h_i and h_{i+1} denote two heads in $HEADS_{k+1}$, such $h_i = new_{t_i}$, $h_{i+1} = new_{t_{i+1}}$, and h_{i+1} is the first head above h_i such that its corresponding new tail t_{i+1} falls to the left of t_i. If $h_i \in UPDATED_{m_p,k}$, then it follows that $h_{i+1} \in UPDATED_{m_p,k}$.*

Observation 1.
For any tail t_i, if the new chain from new_{t_i} to t_i becomes de-activated in G_k, and let j denote the size of the active chain from new_{t_i} to t_i in G_{k+1}. In generation G_k H_{t_i} will no longer include any head of a j-sized chain to t_i.

The above observation is correct since, by definition, an active chain to a given tail can only evolve in G_k by extending a chain shorter by one to the same tail, a chain that was active in G_{k+1}, with a new match from column k. However the fact that the j-sized chain to t_i was the *new* chain of H_{t_i} in G_{k+1} implies, by definition, that there was no shorter active chain to t_i in G_{k+1}. Therefore, H_{t_i} no longer includes any head of a j-sized chain to t_i in G_k.

Lemma 16. *[Chain Reaction.] For any two consecutive heads $h_i, h_{i+1} \in UPDATED_{m_p,k}$, such that $h_i = new_{t_i}$ in G_{k+1}. In generation G_k, h_{i+1} becomes new_{t_i}.*

The above lemma implies that, for any two consecutive heads $h_i, h_{i+1} \in UPDATED_{m_p,k}$, there is, in G_k, an active chain from t_i to h_{i+1}. Since all it takes is one active chain per generation to keep the head that starts this chain and the tail that ends this chain alive, this leads to the the following conclusion.

Conclusion 7.

- *The heads $h_{first+1} \dots h_{last} \in UPDATED_{m_p,k}$ remain active in G_k.*
- *The corresponding tails $t_{first} \dots t_{last-1}$ remain active in G_k.*

The only two critical indices in any $UPDATED$ series, which are not included in the above lists of heads and tails that remain active in G_k, are h_{first} and t_{last}. Since we already know, by Lemma 13, that any head that serves as h_{first} to some $UPDATED$ series becomes extinct in G_k and is replaced with the height of the highest match point below, the only remaining issue to settle is what happens to tails that serve as t_{last} in G_k. This is done in the next Lemma, which claims that all tails that serve as t_{last} for some $UPDATED$ series between two match points remain active in G_k, and the tail that serves as t_{last} for the last span of heads in $HEADS_{k+1}$, if there is indeed no match point in column k above the highest head in this span, is the tail that becomes extinct in G_k.

Lemma 17. *The tail to disappear from row k of $TAILS$, i.e. the tail which becomes inactive in generation G_k, is the t_{last} of the last $UPDATED$ series of $HEADS_{k+1}$.*

The second algorithm computes the rows of $TAILS$ incrementally, in decreasing row order. Row k of $TAILS$ will be computed from row $k+1$ of $TAILS$ by inserting the new tail k, (if such exists) and by removing the "disappearing" tail (if such exists). The algorithm maintains a dynamic list $HEADS$ of active heads. Each head is annotated with two fields: its height and a label associating it with one of the active tails t for which it is new_t. Upon the advancement of the computation from row $k+1$ of the $TAILS$ table to row k, the poset of matches is extended by one column to the left to include the matches of column k of the LCS graph for A versus B. Given the list $HEADS_{k+1}$, sorted by increasing height, the algorithm computes the new list $HEADS_k$, obtained by merging and applying the matches of column k to $HEADS_{k+1}$, and the "disappearing entry" for row k of $TAILS$ is finally realized.

Lemma 18. $r \leq nL$

Time and Space Complexity of the Second Algorithm.
Since $r \leq nL$, the total cost of merging r matches with n lists of size L each is $O(nL)$. In iteration k, up to L_{k+1} new head height values may be updated,

and up to one new head created. The linked list of L_{k+1} heads is then traversed once, and for each item on the list up to one, constant time, swap operation is executed. Therefore, the total work for n iterations is $O(nL)$. There is an additional $O(n \log |\Sigma|)$ preprocessing term for the construction of Match Lists. (Note that we only need to create match lists for characters appearing in B, and that $|\Sigma| \leq n$). Thus, the second algorithm runs in $O(nL + n \log |\Sigma|)$ time, and requires $O(L)$ space.

References

1. A. Apostolico, String editing and longest common subsequences. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, Vol. 2, 361–398, Berlin, 1997. Springer Verlag.
2. Apostolico A., and C. Guerra, The longest common subsequence problem revisited. *Algorithmica*, **2**, 315–336 (1987).
3. Carmel, D.,N. Efraty , G.M. Landau, Y.S. Maarek and Y. Mass, An Extension of the Vector Space Model for Querying XML Documents via XML Fragments, *ACM SIGIR'2002 Workshop on XML and IR*, Tampere, Finland, Aug 2002.
4. Eppstein, D., Z. Galil, R. Giancarlo, and G.F. Italiano, Sparse Dynamic Programming I: Linear Cost Functions, *JACM*, **39**, 546–567 (1992).
5. Hirshberg, D.S., "Algorithms for the longest common subsequence problem", *JACM*, **24**(4), 664–675 (1977).
6. Hunt, J. W. and T. G. Szymanski. "A fast algorithm for computing longest common subsequences." *Communications of the ACM* , **20** 350–353 (1977).
7. Kim, S., and K. Park, "A Dynamic Edit Distance Table.", *Proc. 11th Annual Symposium On Combinatorial Pattern Matching*, 60–68 (2000).
8. Landau, G.M., E.W. Myers and J.P. Schmidt, Incremental string comparison, *SIAM J. Comput.*, **27**, 2, 557–582 (1998).
9. Landau, G.M. and M. Ziv-Ukelson, On the Shared Substring Alignment Problem, *Proc. Symposium On Discrete Algorithms*, 804–814 (2000).
10. Landau, G.M., and M. Ziv-Ukelson, On the Common Substring Alignment Problem, *Journal of Algorithms*, **41**(2), 338–359 (2001)
11. G. M. Landau, B. Schieber and M. Ziv-Ukelson, Sparse LCS Common Substring Alignment, *CPM 2003*, 225-236
12. Myers, E. W., "Incremental Alignment Algorithms and their Applications," *Tech. Rep. 86-22, Dept. of Computer Science, U. of Arizona.* (1986).
13. Myers, E. W., "An O(ND) Difference Algorithm and its Variants," *Algorithmica*, **1**(2): 251-266 (1986).
14. Schmidt, J.P., All Highest Scoring Paths In Weighted Grid Graphs and Their Application To Finding All Approximate Repeats In Strings, *SIAM J. Comput*, **27**(4), 972–992 (1998).
15. Sim, J.S., C.S. Iliopoulos and K. Park, "Approximate Periods of Strings." *Proc. 10th Annual Symposium On Combinatorial Pattern Matching*, 132-137 (1999).