View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Journal of Computer and System Sciences 73 (2007) 1095–1117

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

www.elsevier.com/locate/jcss

Two algorithms for LCS Consecutive Suffix Alignment

Gad M. Landau [a,b,*,1], Eugene Myers [c,2], Michal Ziv-Ukelson [d,3]

[a] *Department of Computer Science, Haifa University, Haifa 31905, Israel*
[b] *Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840, USA*
[c] *Department of Computer Science at the University of California, Berkeley, CA, USA*
[d] *Department of Computer Science, Technion—Israel Institute of Technology, Technion City, Haifa 32000, Israel*

Received 10 September 2005; received in revised form 10 November 2005

Available online 13 March 2007

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 (Longest Common Subsequence) metric, where the LCS metric measures the subsequence of maximal length common to A and B. The first solution is an $O(nL)$ time and space algorithm for constant alphabets, where the size of the compared strings is $O(n)$ and $L \leqslant n$ denotes the size of the LCS of A and B.

The second solution is an $O(nL + n\log|\Sigma|)$ time and $O(n)$ space algorithm for general alphabets, where Σ denotes the alphabet of the compared strings.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Dynamic programming; Longest common subsequence; Match point arithmetic; Incremental algorithms

1. Introduction

The problem of comparing two sequences A of size m and B of size n to determine their similarity is one of the fundamental problems in pattern matching. Throughout this paper we will assume, for the sake of complexity analysis simplification, that $m = O(n)$. Standard dynamic programming sequence comparison algorithms compute an $(n+1)\cdot(m+1)$ 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 [9, 13], Common Substring Alignment Encoding [10–12], Approximate Overlap for DNA Sequencing [9] and more [14],

* Corresponding author. Fax: +972 4 824 9331.
E-mail addresses: landau@poly.edu (G.M. Landau), gene@eecs.berkeley.edu (E. Myers), michalz@cs.technion.ac.il (M. Ziv-Ukelson).
1 Partially supported by NSF grant CCR-0104307, and by the Israel Science Foundation grants 282/01 and 35/05.
2 Fax: +510 643 8443.
3 Partially supported by the Aly Kaufman Post Doctoral Fellowship and by the Bar-Nir Bergreen Software Technology Center of Excellence.

0022-0000/$ – see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2007.03.004

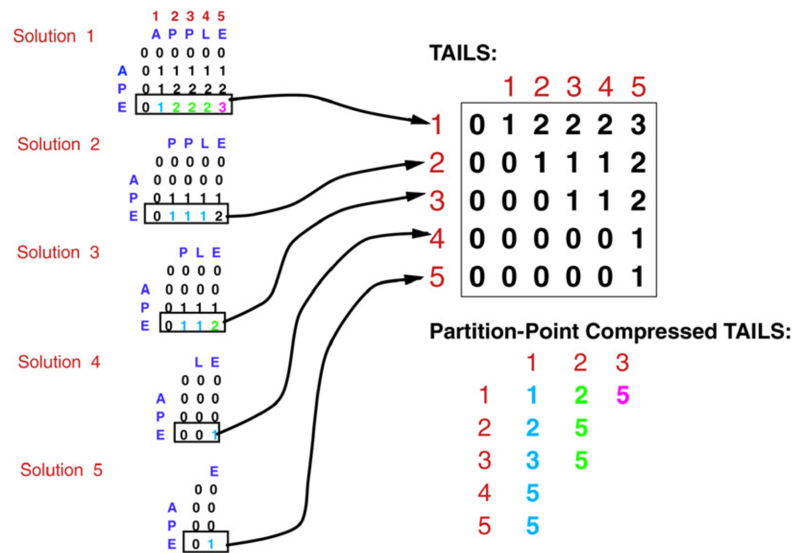

Fig. 1. The Consecutive Suffix Alignment Problem over the two input strings "*APE*" versus "*APPLE*," and an $O(nL)$ encoding of a solution to the problem in the form of a partition-point compressed *TAILS* table.

which require the computation of the solution for the comparison of B with progressively longer suffixes of A, as defined below.

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

We use Fig. 1 to exemplify what we mean by the term "alignment solution" in the above definition. Consider the dynamic programming table *DP* for the alignment of the string "*APE*" versus the four character string "*APPL*." Note that the string "*APPLE*" can be obtained from string "*APPL*" by *appending* the single character $'E'$ to it, and therefore the *DP* table for string "*APE*" versus "*APPLE*" (the table denoted "Solution 1" in Fig. 1) can be obtained from the table for string "*APE*" versus "*APPL*" (the table which consists of the first four columns of the table denoted "Solution 1" in Fig. 1) by computing only one additional column, i.e. $O(n)$ entries. On the other hand, consider the comparison of the *DP* table for string "*APE*" versus the four-character suffix "*PPLE*" of "*APPLE*" (Solution 2 in the figure) with the *DP* table for "*APE*" versus the full string "*APPLE*" (Solution 1 in the figure). Note that the string "*APPLE*" can be obtained from string "*PPLE*" by *pre-pending* a single character ($'A'$) to it, and that the comparison between Solution 2 and Solution 1 in Fig. 1 demonstrates that the *DP* matrix for string B versus string A and the matrix for B versus aA can differ in $O(n^2)$ entries. These potentially changing entries in the *DP* matrix correspond to the $O(n^2)$ values for the comparisons of all prefixes of B with all prefixes of A. In order to reduce this quadratic difference from one suffix alignment solution to the next, we will simplify the computations by replacing the full *DP* table with an encoding of the comparison of all (relevant) prefixes of B with all (relevant) prefixes of A. This encoding will be utilized later in this paper to compute all n solutions to the Consecutive Suffix Alignment of B versus A in $O(nL)$ time complexity instead of the naive $O(n^3)$.

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 [9] 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 [8,9]. Schmidt [13] describes an $O(nm)$ incremental comparison algorithm for metrics whose scoring table values are restricted to rational numbers. Here, we will focus on incremental alignment algorithms for the *Longest Common Subsequence* (LCS) metric [1].

Definition 2 *(LCS).* A *subsequence* of a given string is any string obtained by deleting zero or more symbols from this string. Given two sequences A and B, the *Longest Common Subsequence* of A and B, denoted $LCS[A, B]$, is the subsequence of maximal length which is common to both A and B. Correspondingly, $|LCS[A, B]|$ denotes the size of the LCS of A and B.

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 $O(n^2)$ time, assuming both strings A and B are of size $O(n)$, 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 Hirschberg [5] or Hunt–Szymanski [6]. All Sparse LCS algorithms are preceded by an $O(n \log |\Sigma|)$ preprocessing [1]. The Hirschberg 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 match-points 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 $m \leqslant n$ denotes the size of the shortest string among A and B, and another $O(m \log n + d \log(nm/d))$ algorithm, where $d \leqslant r$ is the number of dominant match-points (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 the size of both A and B is $O(n)$ and $L \leqslant n$ 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, $O(n)$ 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.

2. Preliminaries

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. More formally, let A_k^j denote the substring of A which begins at index k of A and ends at index j of A. At iteration i, A_{n-i}^n will be compared with the full string B. Correspondingly, we formally define the *DP* table which corresponds to the comparison of B with the k-sized suffix of A as follows (see Fig. 2).

Definition 3 *(DP^k).* DP^k denotes the dynamic programming table for comparing string B with string A_k^n, such that $DP^k[i, j]$, for $i = 1 \ldots n$, $j = k \ldots n$, stores $|LCS[B_1^i, A_k^j]|$.

Based on the observation that each column of the LCS *DP* is a monotone staircase with unit-steps, one can 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 Fig. 4).

Definition 4 *(P^k).* 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. If no entry in column j of DP^k bears the value of v, then $P^k[j, v]$ is set to *NULL*.

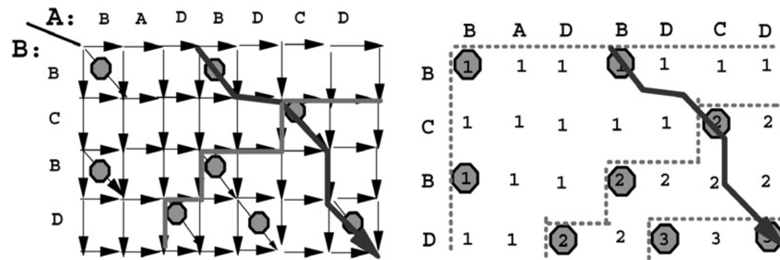

Fig. 2. Chains and anti-chains in G^0 and in DP^0.

In the literature the *DP* table used for computing the LCS of two strings is also viewed as a directed acyclic graph (DAG), called the LCS graph. An *LCS graph* [13] for A and B is a directed, acyclic, weighted graph containing $(|B|+1)(|A|+1)$ nodes, each labeled with a distinct pair (x, y) $(0 \leqslant x \leqslant |B|, 0 \leqslant y \leqslant |A|)$. The nodes are organized in a matrix of $(|B|+1)$ rows and $(|A|+1)$ columns. 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 $B[x+1] = A[y+1]$. 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. 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.

Definition 5 *(G^k). Generation k (G^k for short)* denotes the LCS graph for comparing B with A_k^n. Correspondingly, L_k denotes $|LCS[B, A_k^n]|$.

2.1. Match-points, chains and anti-chains in the LCS graph

The LCS graph of A versus B can be analyzed as a sparse graph of match-points, and the alignment problem as that of finding longest chains in a sparse graph of match-points. An ordered pair of positions (x, y) in the LCS graph where $A[x] = B[y]$ is called a *match-point*. We use r to denote the total number of match-points between A and B. We define the following partial order relation R on the set of match-points between A and B: match-point $[x, y]$ *precedes* match-point $[x_1, y_1]$ in R if $x < x_1$ and $y < y_1$. A set of match-points such that in any pair one of the match-points always precedes the other in R constitutes a *chain* relative to the partial order relation R [1]. A set of match-points such that in any pair neither element of the pair precedes the other in R is an *anti-chain* [3]. Then, the LCS problem translates into the problem of finding a longest chain in the graph of match-points induced by R.

Theorem 1. *(See Folklore [7].) G^k can be partitioned into L^k anti-chains.*

Proof. For each $i = 1, 2, \ldots, L_k$ let A_i consist of those match points x in G^k for which the longest chain in G^k having x as its minimal element contains i match-points (see Fig. 2). □

Lemma 1. *$r \leqslant 2nL$.*

Proof. The size of any anti-chain is bounded by $2n$. The size of a maximal chain in the graph G^0 is L. Each match-point participates in exactly one anti-chain. Therefore, by Theorem 1, we get $r \leqslant 2nL$. □

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.

Definition 6 *(MatchList). MatchList(j)* stores the list of indices of match-points in column j of *DP*, sorted in increasing row index order.

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

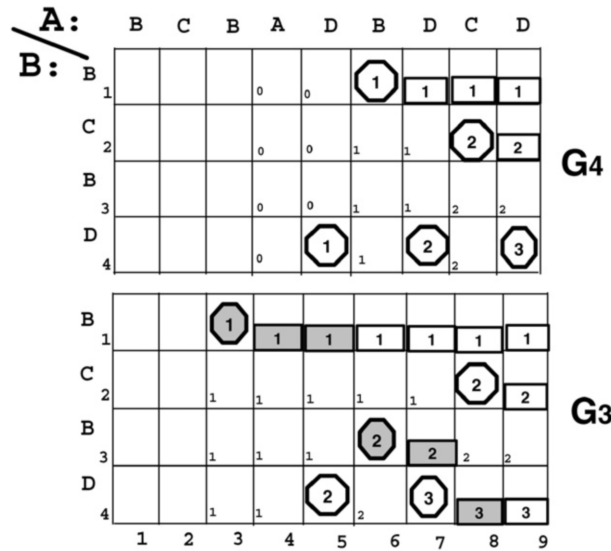

Fig. 3. The update operations applied by the first Consecutive Suffix Alignment algorithm, during the computation of the partition points of generation G^3 from the partition points of generation G^4. Partition points are indicated as rectangles and octagons, and the numbers inside stand for their value. The octagons represent partition points that are both partition points and match-points. The gray rectangles and octagons represent the new partition points in generation G^3.

Definition 7 *(NextMatch).* For each $\alpha \in \Sigma$, $NextMatch(i, \alpha)$ denotes a function that returns the smallest index $i' > i$ such that $B[i'] = \alpha$, if such a match-point exists (i.e., $NextMatch[i, A[j]]$ gives the row index of first match-point in column j below row i of *DP*, 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.

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 is constructed, based on string B.

Then, a main stage is executed, whose task is formally defined as follows: *compute* P^k *for each* $k \in [1, n]$.

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

At stage k of the algorithm, 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. First, P^k is updated by computing and adding the single new partition point of the newly appended column k, which corresponds to the first match-point in column k (see column 3 of G^3 in Fig. 3). 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 3. 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.

In the rest of this paper, we will alternatively use the term *path* instead of chain. A path from vertex (i, j) of the dynamic programming graph G to vertex (i', j') of G is a consecutive set of edges in the dynamic programming graph connecting the two vertices. Note that edges in the LCS dynamic programming graph G can assume only one of three directions: left-to-right, top-to-bottom and diagonal. The diagonal edges correspond to match-points and are assigned a weight of 1. The non-diagonal edges are assigned a weight of zero. Therefore, the weight of path p in G, denoted $|p|$, is computed as the sum of its diagonal edges.

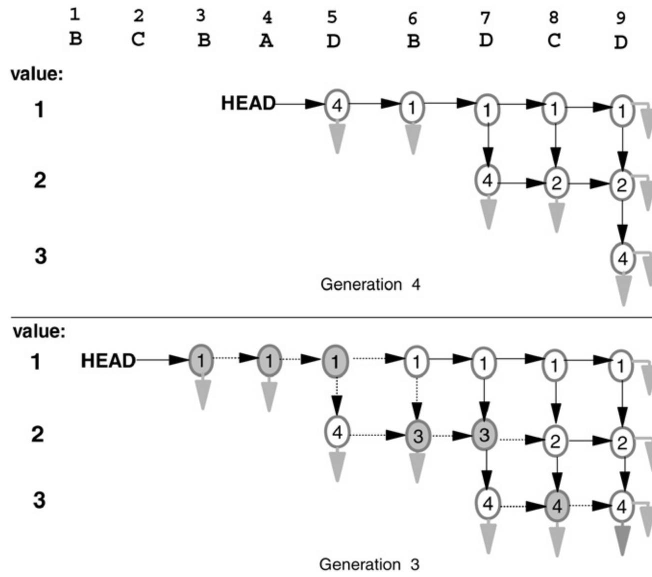

Fig. 4. The implementation of the partition-point data structure as a doubly-linked list. The gray circles represent the new partition points in generation G^3.

Definition 8 *(Crossing paths).* Two paths $p_1 \in G^k$ and $p_2 \in G^k$, *cross* in G^k *iff* there exist 4 points $(i_1, j_1) \in p_1$, $(i'_1, j'_1) \in p_1$, $(i_2, j_2) \in p_2$ and $(i'_2, j'_2) \in p_2$, such that $i_1 < i_2$ and $i'_1 > i'_2$.

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 2. *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 $i \geqslant I_j$.*

Proof. This follows from the monotonicity and unit-step properties of DP. Consider the series of differences, obtained by subtracting column j of DP^{k+1} from column j of DP^k: $delta[i] = DP^k[i,j] - DP^{k+1}[i,j]$, for $i = 0 \ldots m$.

Claim 1. *$delta[i]$ can assume one of two values: either zero or one.*

This is immediate from the unit step properties of LCS. The additional character A_k can either extend the common subsequence by one, or not extend it at all.

Claim 2. $delta[i] \leqslant delta[i+1]$.

This can be proven using the crossing paths argument, as follows. Figure 5 shows the four paths of interest. Path t corresponds to $DP^k[i+1,j]$, path w corresponds to $DP^k[i,j]$, path z corresponds to $DP^{k+1}[i+1,j]$, and path y corresponds to $DP^{k+1}[i,j]$.

Since path y is optimal, $|y| \geqslant A + D \Rightarrow |w| - |y| \leqslant C - A$.

Since path t is optimal, $|t| \geqslant C + B \Rightarrow |t| - |z| \geqslant C - A$.

Therefore,

$$\begin{aligned} delta[i] &= |w| - |y| = DP^k[i,j] - DP^{k+1}[i,j] \\ &\leqslant DP^k[i+1,j] - DP^{k+1}[i+1,j] = |t| - |z| = delta[i+1]. \qquad \square \end{aligned}$$

The next lemma immediately follows.

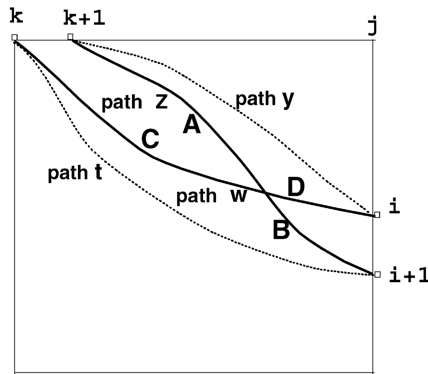

Fig. 5. Optimal paths that must cross.

Lemma 3. *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 $delta[I_j] = DP^k[I_j, j] - DP^{k+1}[I_j, j] = 1$.*

Proof. Following Lemma 2, I_j is a partition point in column j, and all partition points above and below I_j remain intact. Now, suppose that I_j was already a partition point in generation G^{k+1}, $DP^{k+1}[I_j, j] - DP^{k+1}[I_j - 1, j] = 1$. By Lemma 2, all entries in column j of DP with row index I_j or greater raise their value by one from generation G^{k+1} to generation G^k. Therefore, $DP^k[I_j, j] = DP^{k+1}[I_j, j] + 1$. All entries in column j of DP with row index smaller than I_j maintain their values from generation G^{k+1} to generation G^k. Therefore, $DP^k[I_j - 1, j] = DP^{k+1}[I_j - 1, j]$. From this it follows that $DP^k[I_j, j] - DP^k[I_j - 1, j] = 2$, in contradiction to the unit-step properties of LCS. □

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.*

The correctness of the above claim is immediate from the basic rules of dynamic programming.

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.

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.

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. Throughout this section, let I_j denote the index of the new partition point in column j of P^k, let $I_{j-1} = P^k_{j-1}[v]$ denote the new partition point in column $j-1$ of P^k, and let $Y = P^k_{j-1}[v+1]$. We start by constraining the range of row indices of column j in which the new partition point will be searched.

Lemma 4. $I_{j-1} \leqslant I_j \leqslant Y$.

Proof.

- $I_{j-1} \leqslant I_j$.
  This is immediate from Claim 3.
- $I_j \leqslant Y$.
  $DP^k[Y, j-1] = v+1$, and therefore by monotonicity of LCS it follows that $DP^k[Y, j] \geqslant v+1$. On the other hand, $DP^{k+1}[Y, j-1] = v$ (by Lemma 3 and the definitions of I_{j-1} and Y), that is, Y is the first index to reach a value of v in column $j-1$ of P^{k+1}, and therefore by the unit step property of LCS it follows that $DP^{k+1}[Y, j] = v$. Thus, index Y in column j falls in the interval of entries which raised their value from generation G^{k+1} to generation G^k, and by Claim 3 we conclude that $I_j \leqslant Y$. □

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.

Lemma 5. *I_j can assume one of two values, according to the following two cases.*

Case 1. $I_{j-1} \leqslant P_j^{k+1}[v]$, *in which case* $I_j = I_{j-1}$.
Case 2. $I_{j-1} > P_j^{k+1}[v]$, *in which case* $I_j = \min\{Y, \mathit{NextMatch}(I_{j-1}, A[j])\}$.

Proof.

Case 1. $I_{j-1} \leqslant P_j^{k+1}[v]$.

From Lemmas 4 and the monotonicity of LCS it is immediate that in this case $I_j = I_{j-1}$.

Case 2. $I_{j-1} > P_j^{k+1}[v]$.

This means, by Claim 3, that $P_j^k[v] = P_j^{k+1}[v] \neq I_j$. By Lemma 4 we know that $I_j \leqslant Y$. Therefore, the index of I_j is determined based on whether or not there are any match-points in column j of DP^k with row indices greater than I_{j-1} and smaller than Y, as follows.

- If there are no such match-points then, by Lemma 4, Y will be the index of the new partition point in column j of P^k.
- If there is at least one such match-point in column j of DP^k, then let X denote the row index of the first (smallest row index) of such match-point. Then match-point (X, j) will extend the v-sized chain that ends in $P_{j-1}^k[v]$ to a $(v+1)$-sized chain, and therefore, by Lemma 3, X will be the index of the new partition point in column j of P^k.

Note that a special case of this scenario occurs when v is the highest value in column $j-1$ of DP^k. In this case, $P_{j-1}^k[v+1]$ is set to the dummy index $n+1$. Then, if the query $\mathit{NextMatch}(I_{j-1}, n+1)$ returns *NULL*, we conclude that there is no new partition point in column j of P^k and therefore, by Conclusion 1, the algorithm exits the loop of P^k updates. □

Conclusion 2. *The new partition point in column j of P^k, if such exists, is one of five options*:

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

Conclusion 2 greatly simplifies the implementation of the algorithm, as will be seen in the next subsection.

3.3. An efficient implementation of the first algorithm

An efficient algorithm for the Consecutive Suffix Alignments Problem requires a data structure modeling the current partition that can be quickly updated in accordance with Lemma 5. To insert new partition points in $O(1)$ time suggests modeling 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 2, 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 ith cell in a column partition to the ith cell in the next column (see Fig. 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, we next show that the pointer updates which result from each new partition-point insertion can be correctly completed in $O(1)$ time. In order to explain this we refer the reader to the inserted partition points in generation G_3 of Fig. 4. The new partition point $I_j = P^k[j, v]$ is inserted into column j of the mesh, in between the two partition points $P^{k+1}[j, v-1]$, which is also $P^k[j, v-1]$, and $P^{k+1}[j, v]$, which now turns into $P^k[j, v+1]$. Therefore, the $\uparrow$ *down* pointer of the new partition point $I_j = P^k[j, v]$ is correctly set to the next in value, $P^{k+1}[j, v] = P^k[j, v+1]$, and the $\uparrow$ *down* pointer of the previous partition point in value, $P^{k+1}[j, v-1] = P^k[j, v-1]$, is correctly reset to the newly inserted partition point of column j, whose value is v. By Conclusion 2, the $\uparrow$ *right* pointer of any partition point of value smaller than v remains intact, and the $\uparrow$ *right* pointer of any partition point of value $v+2$ or larger remains intact as well. So, the only two partition points whose $\uparrow$ *right* pointer may need re-setting are the two consecutive cells in column j whose values are v and $v+1$. The $\uparrow$ *right* pointer of the new partition point should be set to the partition point of same value v in the next column, $P^{k+1}[j+1, v]$. This pointer can be copied from $P^{k+1}[j, v] = P^k[j, v+1]$, which pointed to the partition point of value v in column $j+1$ in the previous iteration $k+1$. The only problem is that, at this point, the $\uparrow$ *right* pointers of two cells from column j, $P^k[j, v]$ as well as $P^k[j, v+1]$, are both pointing to $P^{k+1}[j+1, v]$. It is too early to resolve this redundancy at this point, since the columns are traversed and updated left-to-right, and therefore when we insert the new partition point of column j we do not know yet which partition point in column $j+1$ will bear the value of v in generation G_k. Therefore, the task of correcting the value of the $\uparrow$ *right* pointer for one of these two consecutive partition points from column j is delayed until the new partition point of column $j+1$ has been computed, and is then decided according to Lemma 5, as follows.

Case 1. If $I_{j+1} = I_j$. (See, for example, the new partition point in column 7 in Fig. 4.) In this case the partition point in column $j+1$ which is pointed to by the $\uparrow$ *right* pointers of $P^k[j, v]$ and $P^k[j, v+1]$ rises in value from v to $v+1$. Therefore, $P^k[j, v+1]$ now correctly points to it, while the $\uparrow$ *right* pointer of $P^k[j, v]$ should be reset to the new partition point of column $j+1$, whose value is v.

Case 2. $P^k_{j+1}[v] = \min\{\mathit{NextMatch}(P^k_j[v], j+1), P^k_j[v+1]\}$. (See, for example, the new partition point in column 6 in Fig. 4.) In this case the partition point in column $j+1$ which is pointed to by the $\uparrow$ *right* pointers of $P^k[j, v]$ and $P^k[j, v+1]$ maintains its value of v. Therefore, $P^k[j, v+1]$ now correctly points to $P^k[j+1, v+1]$, while the $\uparrow$ *right* pointer of $P^k[j, v]$ should be reset to the next, new partition point of column $j+1$, whose value is $v+1$.

3.4. Time and space complexity of the first algorithm

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

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. Therefore, 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, which is $O(nL)$. Given the *NextMatch* table which was prepared in the preprocessing stage, the computation

of the next partition point, according to Lemma 5, can be executed in constant time. Navigation and insertion of a new partition point can also be done in constant time according to Conclusion 2 (see Fig. 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 [9,13], Common Substring Alignment Encoding [10–12], Approximate Overlap for DNA Sequencing [9] 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 Definition 6 in Section 2.1). The *MatchList* for strings A and B is constructed during the preprocessing stage.

A Comment Regarding Row Indexing: Note that, for the sake of clarity, throughout this section we assume that the rows in both DP^k and in G^k are numbered in a bottom–up order (see Fig. 7). That is, the lowest row in the bottom of the DP table is row number 1. Thus, when we say "a higher/lower match-point" we mean a match-point with a greater/smaller row index, correspondingly. We also assume that the first match-point in a given chain is the match-point with the greatest row index and the smallest column index.

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 first 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 9 *(TAILS).* $TAILS[k, j]$ is the column index of the jth partition point in the first row of G^k. In other words, $TAILS[k, j] = t$ if t is the smallest column index such that $|LCS[A_k^t, B]| = j$. That is, t is the index of the smallest column index to end a j-sized chain in G^k.

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

Definition 10 *(Tail).* Let t denote the value at entry j of row k of *TAILS*.

1. t is considered a *tail* in generation G^k (see Figs. 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 6. *If column k of the LCS graph contains at least one match point, then the following changes are observed when comparing row $k+1$ of TAILS to row k of TAILS. Otherwise, there are no changes.*

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: if $|LCS[B, A_k^n]| = |LCS[B, A_{k+1}^n]|$, then one entry value, which appeared in row $k+1$, disappears in row k. In this case, let j denote the index of the disappearing entry. Otherwise, if $|LCS[B, A_k^n]| = |LCS[B, A_{k+1}^n]| + 1$, then no entry disappears. In this case, let $j = |LCS[B, A_k^n]|$.*

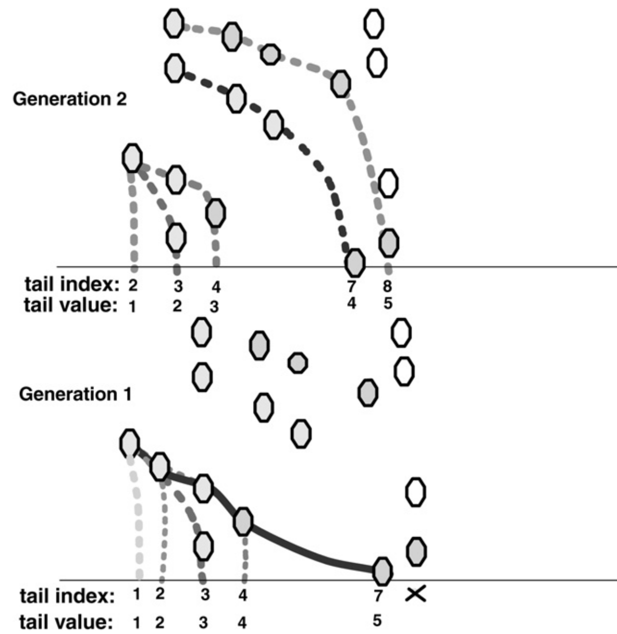

Fig. 6. The evolution of leftmost chains from chains that are not necessarily leftmost.

- *All values from row $k+1$ of TAILS up to index j 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 j remain intact in row k of TAILS.*

Proof. This is immediate from Lemmas 2, 3 and Conclusion 1. □

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*.

4.2. The $O(L^2)$ active chains in a given generation

In this section we will show that some chains dominate others as candidates to evolve into the L_k leftmost ending chains (corresponding to the L_k tails), and use this observation to define a core set of chains that needs to be considered throughout the algorithm. 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 that 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 Fig. 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.

At this stage it is 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 match-points 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.

Definition 11 *(Active chain).* 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 c_2 starts lower than c_1 and c_2 ends earlier than c_1.

For the purpose of tail computation, it is sufficient to maintain for each chain the row index of its first match-point and the column index of its last match-point. By Lemma 6 we know that any chain in generation k which is a candidate to become a leftmost chain (by left-extension via match-points contributed by future generations) must end in one of

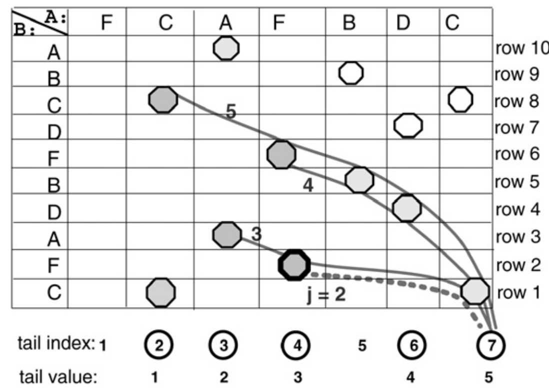

Fig. 7. The set of chains H_7 for the tail with value 5 and index 7 in generation G^2 of the Consecutive Suffix Alignment of strings $A =$ "*BCBADBCDC*" versus $B =$ "*BCBD*." The head which is h_7 is highlighted with a thicker border, and the corresponding shortest chain of size 2 is dotted. The dark circles around column indices directly below the bottom row of the graph mark the active tails in G^2. Row numbers, in bottom–up order, are marked on the left side of the table.

the L_k tails of generation k (the interested reader could reach the same conclusion via a crossing paths argument). Therefore, from now on we will use the term *tail* when referring to end-points of active chains. From this we conclude that there are $O(L_k^2)$ active chains in any given generation k. We next turn to address the first match-points of active chains.

Definition 12 *(Head).* The row index of the first match-point in an active chain is denoted a *head*.

4.3. Static properties of H_t (and T_h)

In the rest of this paper we will consider two active chains of identical sizes which have the same head and the same tail as one. This representation of chains, which is developed in this section, will be used to further compress the set of key values that needs to be maintained from one generation to the next. In order to organize and further analyze the minimal set of key values which suffices for our computations, we associate with each tail a set of relevant heads, as follows (see Fig. 7).

Definition 13 *(H_t).* H_t denotes the set of heads of active chains in G^k that end in tail t.

Symmetrically, for each head we associate a set of relevant tails, as follows.

Definition 14 *(T_h).* T_h denotes the set of tails of active chains in G^k that start in head h.

In the next Lemmas 7 to 10 we formalize some properties of H_t (and T_h) per a given generation. The changes to H_t (and T_h) from one generation to the next will be discussed is Section 4.4.

Lemma 7. *The heads of H_t are ordered in increasing row index and increasing chain size* (*see Fig.* 7). *Symmetrically, the tails of T_h are ordered in increasing column index and increasing chain size.*

Proof. Suppose there were two heads a, b in H_t, such that b is higher than a and yet the chain headed by a has a size s_a which is greater than the size s_b of the chain headed by b. Then the s_b-sized suffix of the chain headed by a would form a chain of size s_b which starts lower than b yet ends before or at t, in contradiction. □

Lemma 8. *For any tail t* (*symmetrically head h*), *the sizes of active chains which correspond to the heads in H_t* (*symmetrically tails in T_h*) *form a consecutive span.*

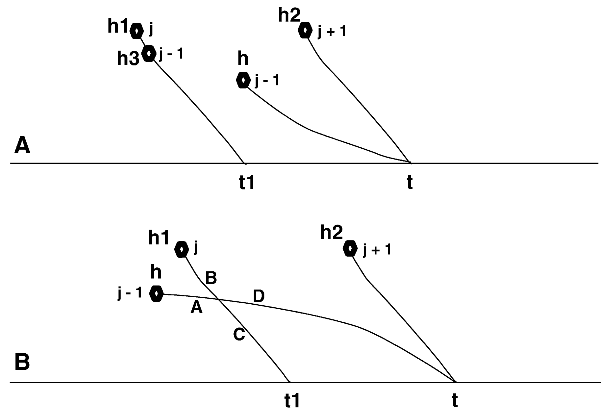

Fig. 8. A crossing paths argument for proving that sizes of active chains in H_t form a consecutive span.

Proof *(Crossing paths argument).* Consider tail t in Fig. 8. By definition, H_t must include the head of the lowest, leftmost chain of size val_t. By Lemma 7 this head is the highest head in H_t. We prove that if H_t includes an active $j+1$ head and an active $j-1$ head, then it must also include an active j head. Suppose that the lowest j-chain to end in t was inactive. Then, by definition, there must be another j-chain which starts lower and ends to its left in some tail t_1. Consider the two cases shown in the figure.

Case 1. (See Fig. 8A.) The $j-1$-path that ends in t does not cross the j-path that ends in t_1. Let h_3 denote the second match of the j-path that ends in t_1 and let (i,k) denote the indices of h_3. Let h denote the first match in the $j-1$-path that ends in t and let (i',k') denote the indices of h. We claim that h_3 precedes h in G, i.e. $i > i'$ and $k < k'$:

- Suppose $i \leqslant i'$. This would imply that there is a $j-1$-path that starts in h_3 and ends in t_1, to the left of t, in contradiction to the assumption that the $j-1$-chain from h to t is an active chain.
- Suppose $k' \leqslant k$. We are given that $t_1 < t$ and have also asserted above that $i > i'$. By the precedence order defined for match-points in a chain, it is clear that $k < t_1$. Therefore, $k' \leqslant k < t_1$ and thus the two paths (the path from h_3 to t_1 and the path from h to t) must cross, in contradiction to the case assumption.

Since, as asserted above, match-point h_3 precedes match-point h, it could be appended to the $j-1$-chain that ends in t to form a j-chain—in contradiction to the assumption that the j-chain that ends in t_1 starts lower.

Case 2. (See Fig. 8B.) The $j-1$-chain that ends in t does cross the j-chain that ends in t_1.

- if $C \geqslant D$. Then there is another $(j-1)$-sized chain that starts at the same row index and ends in a smaller column index—in contradiction to the assumption that the $j-1$-chain that ends in t is active.
- if $C < D$. Then the j-suffix of BD forms a j-chain that ends in t and starts lower than chain BC, in contradiction to the assumption that the j-chain which ends in t_1 is responsible for de-activating the j-chain which ends in t. □

The next definition, which applies to the heads and tails of chains that are active in a given generation, will be used in Lemma 9.

Definition 15 *(New head).* For any head h and tail t which correspond to active chains in a given generation of the LCS alignment graph. We say that "head h is new at tail t" iff the following two conditions are met:

1. There is an active chain which originates in head h and ends in tail t.
2. All active chains which originate in head h end in a tail that is greater than or equal to t.

A "new tail t at head h" is symmetrically defined.

Lemma 9.

1. *The head h_1 of the smallest chain in H_t is new at tail t.*
2. *All other heads in H_t are not new at tail t.*
   Symmetrically,
3. *the tail t_1 of the smallest chain in T_h is new at head h.*
4. *All other tails in T_h are not new at head h.*

Proof. We prove 1 and 2. The correctness of 3 and 4 follows by symmetry.

1. *The head h_1 of the smallest chain in H_t is new at tail t.* Consider tail t from Fig. 12(A and B). If h_1 heads the smallest active chain that ends in t, and if u denotes its size, then the $(u-1)$-sized chain that ends in t has been de-activated. By Lemma 7 we know that the head of the $(u-1)$-sized chain in H_t, prior to its de-activation, was lower than h_1. This means that there is now a $(u-1)$-sized chain that starts in some head h_2 lower than h_1 and ends to its left in some tail $t_2 < t$. Suppose that h_1 is not new in H_t. Then there is some active j-chain that starts in h_1 and ends in some tail t_1 to the left of t (note that $j \leqslant u-1$). If $t_2 \leqslant t_1$ (see Fig. 12A)—then the j-prefix of the $(u-1)$-sized chain from h_2 to t_2 would form a lower-origin, earlier-ending j-chain, in contradiction to the livelihood of the j-chain from h_1 to t_1. Therefore we know that $t_1 < t_2 < t$ and the two chains must cross (see the crossing paths argument in Fig. 12B).

2. *All other heads in H_t are not new at tail t.* Consider tail t from Fig. 12C. Suppose that the head h_0 of the u-sized chain in H_t is new, yet u is not the shortest chain in H_t. This means that none of the smaller chains which originate in h_0 is active. In particular, the $(u-1)$-sized prefix of this u-sized chain has been deactivated by some $(u-1)$-sized chain that starts in some head $h_2 < h_0$ and ends in some tail $t_2 < t$. Let h_3 denote the head of the $(u-1)$-sized chain in H_t (by Lemma 8 and the fact that h_0 is not the head of the shortest chain in H_t we know that there is indeed a $(u-1)$-sized chain in H_t). Suppose that $h_3 \geqslant h_2$. Then the livelihood of the $(u-1)$-sized chain in H_t would be contradicted. Thus, $h_3 < h_2$, and therefore the active chain from h_2 to t_2 crosses the active chain from h_3 to t (see the crossing paths argument in Fig. 12C). □

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

Lemma 10. *H_{t_i} includes all heads that are higher than h_{t_i} and start at least one active chain which ends in some tail $t < t_i$.*

Symmetrically, T_{h_i} includes all tails that are to the right of t_{h_i} and end at least one active chain which starts in some head $h < h_i$.

Proof. By crossing paths argument (similarly to the proof of Lemma 9). □

4.4. Changes in H_t (and T_h) from one generation to the next

In this section we discuss the changes in H_t (and T_h) as the graph of match-points for generation G^{k+1} is extended with the match-points of column k. The next lemma is key to the efficiency of the incremental algorithm which is to be described in the next section.

Lemma 11. *From one generation to the next, the number of active heads in H_t can decrease by at most one. Furthermore, of all the chains that start in some head in H_t and end in t, only the shortest chain could be de-activated in G^k without being replaced by a lower head of a similar-sized active chain to t.*

Proof. Let H_t include chains of sizes $j \ldots val_t$ in generation G^{k+1}. We will show that, in generation G^k, H_t will either keep the same chain sizes, or loose its shortest chain of size j. All other chains of sizes $j+1$ to val_t are still active, though they may have been lowered.

Consider the head of the shortest, j-sized chain that ends in t. We know that there is a chain of size $j-1$ that starts lower than the $(j-1)$-sized prefix of this chain and that ends to the left of t. This chain could be extended by a

match-point in column k that is lower than the head of the j-sized chain in H_t, as a result of which H_t will loose its j-sized chain.

Next, let C denote a chain of any size $u+1 > j$ in H_t. By Lemma 8 we know that if a chain of size $u+1$ is active in H_t in G^{k+1}, and this chain is not the shortest active chain to t, then there is also some active chain of size u in H_t. Let D denote this u-sized chain. Since chain D is active in G^{k+1}, there could not possibly be another chain of size u in G^{k+1} that starts lower than (or equal to) to chain D and ends to its left, and that could therefore be extended to form a new chain, in G_k, of size $(u+1)$, that would dominate chain C. On the other hand, there could be another active chain of size u in G^{k+1}, denoted E, that starts higher than chain D in G^{k+1} and ends to its left. However, the first match-point of this chain E already extends the u-sized chain D of H_t in G^{k+1} to a $(u+1)$-sized chain in G^{k+1} that ends in H_t, and therefore we know that any extension of chain E to a $(u+1)$-sized chain in G^k will definitely start higher than the $(u+1)$-sized chain of H_t in G^k and thus will not dominate it. Therefore, the $(u+1)$-sized chain of H_t in G^{k+1}, chain C, either stays as is in G^k or is replaced with a lower $u+1$ sized chain that is obtained by extending chain D, the u-sized chain of H_t in G^{k+1}. □

4.5. An algorithm based on an $O(L_k)$ PAIRS state encoding for G^k

In this section we describe the new data structure which is the core of our algorithm. Note that two or more different chains could share the same head in a given generation. For example, see Fig. 6. Based on this observation, we decided to count the number of different match-points which serve as heads in a given generation. Therefore, let G_r^k denote the graph of match-points obtained by turning the visual representation of graph G^k around by 90 degrees counterclockwise (see Fig. 9), and then considering chains of match-points in a top-down, left-to-right precedence order. Clearly, G_r^k is the graph of match-points obtained for the comparison of the reversed string A_k^n versus the reversed string B. It is easy to see that, by symmetry, the active chains of G^k are also the active chains of G_r^k, and the L_k tails of G_r^k are the L_k heads of G^k. This means that all active chains of G^k must originate in one of the L_k tails of G_r^k (i.e. heads of G^k) and end in one of the L_k heads of G^k (i.e. tails of G_r^k). Furthermore, we note a one-to-one mapping between the L_k heads of G^k and the L_k heads of G_r^k, which induces a set of pairs (t_i, h_i), $i = 1 \ldots L$, such that h_i is the head of the shortest chain to end in t_i in G_k, and t_i is the head of the shortest chain to end in h_i in G_r^k (see Fig. 10). These observations are formalized as follows,

Observation 1. *There is a one-to-one correspondence between the L_k heads and the L_k tails of G^k.*

Observation 1 is the key to the efficient $O(L_k)$ state encoding used in the second algorithm. Based on this observation, the heads of G^k and the tails of G^k (i.e. heads of G_r^k) can be paired as follows.

Definition 16 *(Head-to-tail pair).* For any active tail t_i in G^k, let h_i denote the head of the shortest active chain which ends in H_{t_i}. Symmetrically, t_i is the tail of the shortest active chain in T_{h_i}. (h_i, t_i) is denoted a *head-to-tail pair in G^k*.

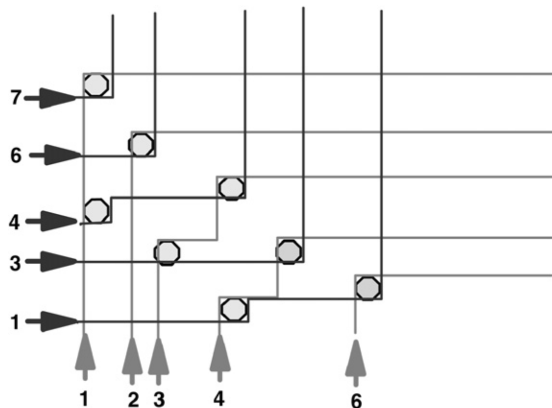

Fig. 9. The tails of G^k are the column indices of first match-points in the anti-chains of A_k^n versus B. Symmetrically, the heads of G^k are the column indices of first match-points in the anti-chains of the reversed string A_k^n versus the reversed string B.

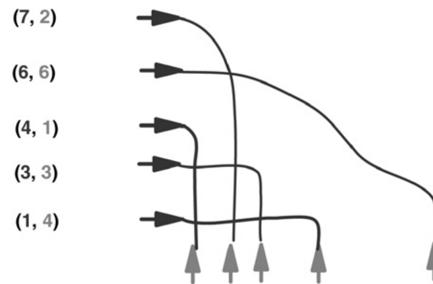

Fig. 10. The head-to-tail pairings in G^k. This figure continues the example of Fig. 9.

Throughout the algorithm, the relevant state information will be represented by a dynamic list *PAIRS* of head-to-tail pairs: $(h_1, t_1), (h_2, t_2) \dots (h_{L_k}, t_{L_k})$. (See Fig. 10.)

Definition 17 *($PAIRS_k$). $PAIRS_k$* denotes the set of active head-to-tail pairs in generation G^k, maintained as a list which is sorted in increasing head row index. Each pair (h_i, t_i) in the list is annotated with two values. One is the row index of h_i, and the other is the column index of t_i.

In the suggested algorithm, which is formally described and analyzed in Section 4.7, the *PAIRS* list will be modified for each consecutive suffix generation G^k, based on the match-points in column k of *DP*, as demonstrated in Fig. 11. The row indices of the heads in $PAIRS_{k+1}$ will be merged with the row indices of match-points in column k, and then the *PAIRS* list will be traversed once, in a bottom–up order, and updated with the resulting modifications to the set of active heads, to the set of active tails, and to the head-to-tail pairing described above, according to the evolution of active chains from G^{k+1} to G^k.

Two objectives will be addressed in G^k. The first and main task is to compute the tail that becomes inactive in G^k, according to Lemma 6. Recall that in Lemma 11 we showed that, for any tail t that was active in G^{k+1}, the size of H_t can decrease by at most 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. In the rest of this paper, and in Lemma 16 in particular, we show how the tail to disappear in G^k can be identified via a single traversal of the list formed by the merge of the match points of column k with the structure $PAIRS_{k+1}$.

The second task is to update the state information $PAIRS_{k+1}$ with the match-points in column k of *DP*, in order to prepare $PAIRS_k$ for the upcoming computations of G^{k-1}. Figure 11 shows the active heads and active tails of generation $k+1$ and two consecutive match points (m_p and m_{p+1}) from column k. The chains (arcs) in the figure demonstrate the head-to-tail pairings of G_{k+1}, i.e. for each tail t_i we drew its shortest chain that connects it to its corresponding head h_i, such that $(h_i, t_i) \in PAIRS_{k+1}$. In Lemmas 13 and 14 we show that the (*head*, *tail*) pairs that need re-labeling in G_k (highlighted in black in the figure) form a series of increasing head row indices and decreasing tail column indices (i.e. a series of crossing chains). Furthermore, we show that the modifications of the (*head*, *tail*) pairs from $PAIRS_{k+1}$ that need to be updated in G_k can be applied in a single up-traversal of the list, in which each head h_i of a modified pair $(h_i, t_i) \in PAIRS_{k+1}$ replaces the head of the next pair in line to be modified, i.e. $(h_{i+1}, t_{i+1}) \in PAIRS_{k+1}$, yielding the resulting new pair $(h_i, t_{i+1}) \in PAIRS_k$.

The detailed description of how to identify the disappearing tail, as well as how to update the *PAIRS* data structure of G^k in preparation for the computations of G^{k-1} will be partitioned into four cases.

Case 1. *The lowest match-point in column k.* The first match-point in column k is a new head. It is the first chain, of size 1, of the tail k, and therefore is h_k. All pairs whose heads fall below this match-point are unaffected, since no new chain that starts lower than these heads could have possibly been created in G^k.

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

Case 3. *Match-points above the highest head in $PAIRS_{k+1}$.* The lowest match-point in column k which is above the highest active head in $PAIRS_{k+1}$, if such a match-point exists, becomes a new head. Consider the longest chain in

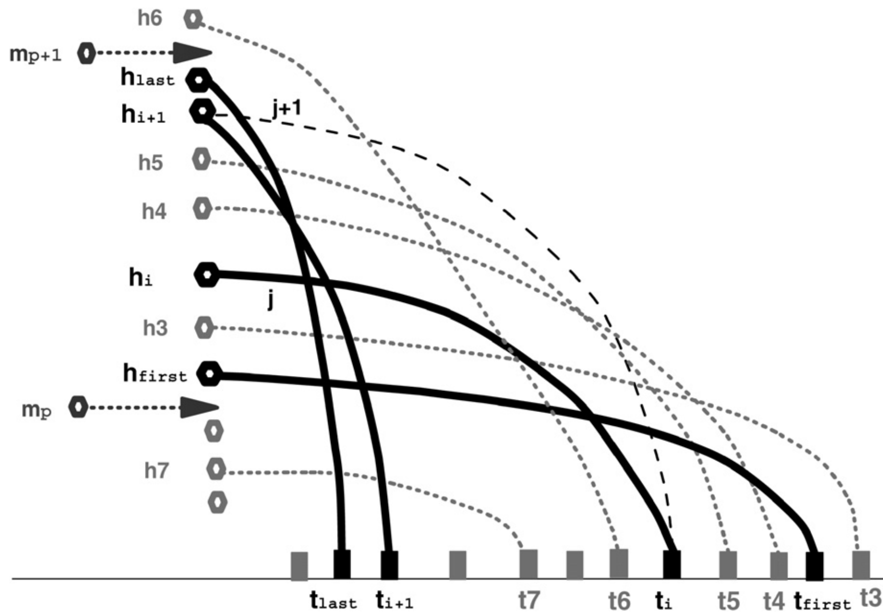

Fig. 11. The *PAIRS* list traversal in increasing row index during its update with the match-points of column k in generation G^k. The modified heads, as well as their corresponding tails and new chains, are highlighted in black.

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-point to a lowest, leftmost $L_k = L_{k+1} + 1$ chain, and therefore this match-point is a new head.

Case 4. *The series of pairs whose heads fall between two consecutive match-points* m_p *and* m_{p+1}. This case, which includes the series of remaining heads above the highest match-point 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 section.

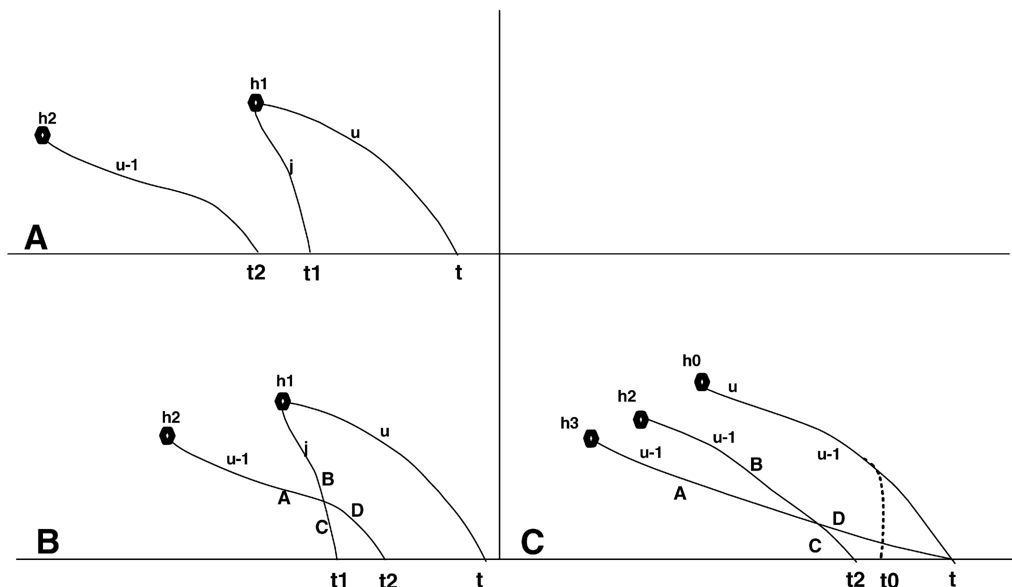

Fig. 12. A crossing paths argument for proving that the head of the smallest chain in H_t is new.

4.6. Heads that fall between two match-points m_p and m_{p+1} in column k

In this section we will show that some of the pairs whose heads fall between two consecutive match-points m_p and m_{p+1} (see Fig. 11) will change in G^k while others will remain unaffected. Therefore, let $UPDATED_{m_{p,k}}$ denote the series of pairs $(h_{\text{first}}, t_{\text{first}}), (h_{\text{first}}+1, t_{\text{first}}+1), (h_{\text{first}}+2, t_{\text{first}}+2)\ldots(h_{\text{last}}, t_{\text{last}})$ whose heads fall between m_p and m_{p+1} and whose head-to-tail association changed in generation k, ordered in increasing head row index. In Lemma 12 we will show that the first head that gets modified in the transition from $PAIRS_{k+1}$ to $PAIRS_k$, i.e. the pair denoted $(h_{\text{first}}, t_{\text{first}}) \in UPDATED_{m_{p,k}}$, is the pair from $PAIRS_{k+1}$ whose head is lowest above m_p. In Lemma 13 we will identify those pairs which are not included in $UPDATED_{m_{p,k}}$.

Lemma 12. *h_{first} is no longer an active head in G^k.*

Proof. Consider the tail t of any j-sized chain that started in h_{first}. There are two cases to consider. If j is the size of the smallest chain that ends in t then t_{first} is t. In this case, any of the $(j-1)$-sized chains which dominate the de-activated $(j-1)$-sized chain of H_t could be extended with m_p to yield a j-sized chain that starts lower than h_{first} and ends to its left.

In the second case, there exists an active $(j-1)$-sized chain which ends in t. The head of this chain is lower than m_p. Hence, the head of this chain can be extended by m_p to form a j-sized chain that starts lower than h_{first} and ends in t. □

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

Lemma 13. *Consider two consecutive pairs $(h_i, t_i), (h_{i+1}, t_{i+1}) \in PAIRS_{k+1}$. If $m_p < h_i < h_{i+1} < m_{p+1}$ and $t_i < t_{i+1}$, then the chain from h_{i+1} to t_{i+1} remains active in G^k.*

Proof. Consider Fig. 13. Let chain c_1 of size j denote the chain from h_i to t_i. Let chain c_2 of size u denote the chain from h_{i+1} to t_{i+1}. Suppose that c_2 becomes extinct in G^k. Then there is a $(u-1)$-sized chain in G^{k+1} which starts in some head h below m_p and ends in some tail t, such that $t \leqslant t_{i+1}$.

- $t \leqslant t_i$. We know that $u > j$, or c_2 would be extinct in G^{k+1}. Therefore $j \leqslant u-1$. The $u-1$-chain from h to t starts lower than c_1 and ends in the same column or to its left, in contradiction to the livelihood of c_1 in G^{k+1}.
- $t_i < t \leqslant t_{i+1}$. This case can be proven by crossing paths argument. □

In Lemma 13 we have identified head-to-tail pairs that remain unmodified in the transition from $PAIRS_{k+1}$ to $PAIRS_k$. Assume that all other pairs do get modified (this will be proven in Lemma 14). This means that the pairs

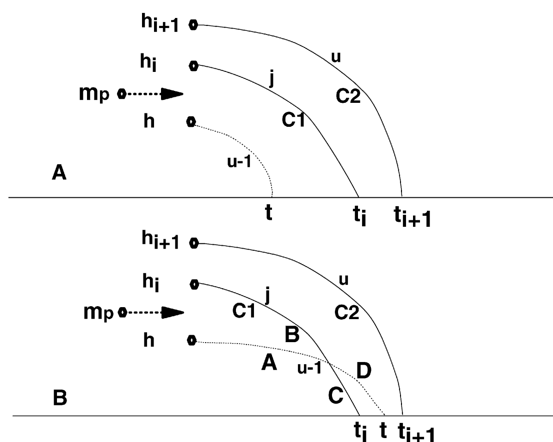

Fig. 13. A crossing paths argument for proving Lemma 13.

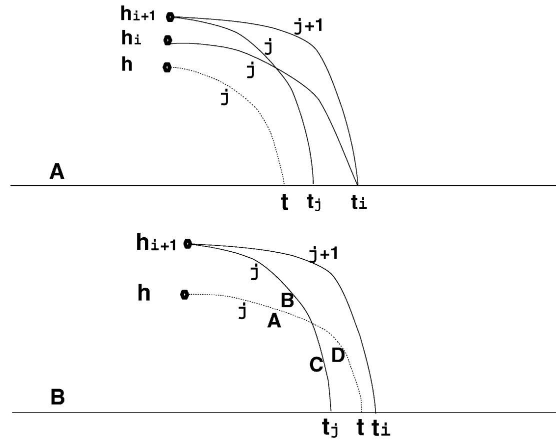

Fig. 14. A crossing paths argument for proving Lemma 14.

in $UPDATED_{m_p,k}$ form a series, starting with the first head above m_p, such that for any two consecutive pairs (h_i, t_i), (h_{i+1}, t_{i+1}) in this series, h_{i+1} is the lowest head above h_i such that $t_{i+1} < t_i$. (In other words, the pairs in $UPDATED_{m_p,k}$ and their corresponding tails form a series of increasing head row indices and decreasing tail column indices.) We will next show that the modifications to the pairs in the $UPDATED_{m_p,k}$ series are such that for any two consecutive pairs (h_i, t_i), (h_{i+1}, t_{i+1}) in this series, (t_i) is re-matched with (h_{i+1}). In other words, the first pair in the $UPDATED_{m_p,k}$ series, $(h_{\text{first}}, t_{\text{first}})$, disappears from $PAIRS_k$, and for any other pair (h_i, t_i) in the $UPDATED_{m_p,k}$ series h_i is re-matched, in $PAIRS_k$, to the tail t_{i-1} of the preceding pair in the $UPDATED_{m_p,k}$ series. According to this observation, the tail of the last pair in the $UPDATED_{m_p,k}$ remains headless. In the next subsection we will deal with this last, odd tail.

Lemma 14. *If (h_{i+1}, t_{i+1}) follows (h_i, t_i) in $UPDATED_{m_p,k}$, then $(h_{i+1}, t_i) \in PAIRS_k$.*

Proof. We need to show that, for each pair $(h_{i+1}, t_{i+1}) \in UPDATED_{m_p,k}$

1. The pair (h_{i+1}, t_{i+1}) becomes inactive in G^k.
2. h_{i+1} will be paired with t_i in G^k.

The proof is as follows.

1. We claim that for any two consecutive pairs (h_i, t_i), $(h_{i+1}, t_{i+1}) \in UPDATED_{m_p,k}$ the deactivation of the chain (h_i, t_i) implies the deactivation of the chain (h_{i+1}, t_{i+1}). Since the first chain $(h_{\text{first}}, t_{\text{first}})$ in $UPDATED_{m_p,k}$ is deactivated in G^k by Lemma 12, the above claim would imply that $(h_{\text{first}} + 1, t_{\text{first}} + 1)$ also gets deactivated in G^k, therefore $(h_{\text{first}} + 2, t_{\text{first}} + 2)$ also gets deactivated, and so on for all chains in $UPDATED_{m_p,k}$.

Let j denote the size of the chain (h_i, t_i). By Lemmas 8 and 10 there is an active chain of size $j + 1$ from h_{i+1} to t_i. Since $t_{i+1} < t_i$ we know by Lemma 8 that the size of the chain (h_{i+1}, t_{i+1}) is smaller than the size of the chain (h_{i+1}, t_i). Therefore, the size of (h_{i+1}, t_{i+1}) is either equal to or smaller than j (it is actually smaller than j if $T_{h_{i+1}}$ includes additional chains that end in tails between t_{i+1} and t_i).

This means that it suffices to show that the j-sized chain of $T_{h_{i+1}}$ dies in G^k, since then, by Lemma 8, so do chains of sizes smaller than j that originate in h_{i+1} and in particular chain (h_{i+1}, t_{i+1}). For this proof we refer the reader to Fig. 14. Let (h_{i+1}, t_j) denote the j-sized chain of $T_{h_{i+1}}$. We will show that if (h_i, t_i) becomes de-activated in G^k, then (h_{i+1}, t_j) also becomes de-activated in G^k. Let (h, t) denote the chain that made the j-sized chain from h_i to t_j extinct in G^k. If $t \leqslant t_j$ (see Fig. 14A), then of course the j-sized chain from h_{i+1} to t_{i+1} is not active. Otherwise, $t_j < t \leqslant t_i$, and we get the crossing paths argument of Fig. 14B.

2. From Lemma 11 we know that any tail can only lose one chain per generation, and therefore, since chain (h_{i+1}, t_{i+1}) whose size is $\leqslant j$ has already been deactivated in G^k, H_i will keep an active $(j + 1)$-sized chain in G^k.

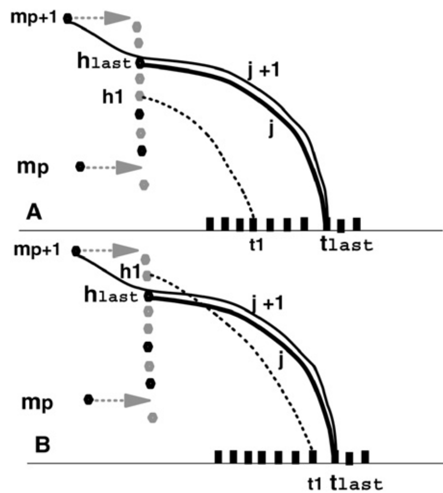

Fig. 15. The fate of a t_{last} tail that falls between two match-points from column k of *DP*.

We have shown in the previous item that h_{i+1} is the head of the $(j+1)$-sized chain of H_i in G^{k+1}. Chain (h_{i+1}, t_i) will remain active in G^k, unless a lower $(j+1)$-sized chain to t_i is created in G^k. But such a lower $(j+1)$-sized chain to t_i can only be created in G^k via an extension of the j-sized chain to t_i that was active in G^{k+1} by a match-point in column k. However, since h_i and h_{i+1} are two consecutive heads in $\mathit{UPDATED}_{m_p,k}$, the next match-point which could possibly extend the j-sized chain from h_i to t_i is m_{p+1}. Since m_{p+1} is higher than h_{i+1}, we conclude that, in G^k, $h_{i+1} = h_{t_i}$. □

4.6.1. The fate of tail t_{last} in $\mathit{UPDATED}_{m_p,k}$

In this section we will handle the tail $t_{\text{last}} \in \mathit{UPDATED}_{m_p,k}$ which remains headless at the end of the chain of modifications to the pairs in $\mathit{UPDATED}_{m_p,k}$. In the next two lemmas we will show that:

- Any tail that serves as t_{last} for some *UPDATED* series between two match-points remains active in G^k and is paired with m_{p+1}.
- The tail that serves as t_{last} for the last span of heads in PAIRS_{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 15. *The t_{last} of an $\mathit{UPDATED}_{m_p,k}$ series between two match-points stays alive and will be paired with m_{p+1}.*

Proof. Consider Fig. 15. Let j denote the size of the chain from h_{last} to t_{last}. Clearly, m_{p+1} extends the deactivated j-sized chain from h_{last} to t_{last} to a $(j+1)$-sized chain. We will prove that this chain is active. Suppose by contradiction that the $(j+1)$-sized chain from m_{p+1} to t_{last} is not active in G^k. This means that there is, in G^k, another $(j+1)$-sized chain, denoted c_1, from some head h_1 to some tail t_1, such that either one of the following two cases holds.

Case 1. $h_1 < m_{p+1}$, and $t_1 \leqslant t_{\text{last}}$. Note that h_1 could not possibly be below or at h_{last} (see h_1 in Fig. 15A), since the j-sized suffix of chain c_1 would then contradict the livelihood of the j-sized chain from h_{last} to t_{last} in G^{k+1}.

Therefore, consider Fig. 15B. Since there is no match point in column k of *DP* which falls between h_{last} and m_{p+1}, we conclude that chain c_1 was alive in G^{k+1}. We are given that $(h_{\text{last}}, t_{\text{last}}) \in \mathit{UPDATED}_{m_p,k}$, and therefore by Observation 1 $(h_1, t_{\text{last}}) \notin \mathit{PAIRS}_{k+1}$. Therefore, by Lemma 8, the shortest chain that originated in h_1 in G^{k+1} was of size smaller than $j+1$ and thus ended in some tail that is strictly smaller than t_{last}, such that $(h_1, t_1) \in \mathit{UPDATED}_{m_p,k}$. However, $h_1 > h_{\text{last}}$ implies that $(h_1, t_1) \in \mathit{UPDATED}_{m_p,k}$ in contradiction to the definition of h_{last}.

Case 2. $h_1 = m_{p+1}$ and $t_1 < t_{\text{last}}$. In this case let c_2 denote the j-sized prefix of c_1, and let h_2 denote the row index of the first match point of c_2. Similarly to the previous case, if $h_2 \leqslant h_{\text{last}}$ then the j-sized prefix of c_1, which was an active chain in G^{k+1}, would contradict the livelihood in G^{k+1} of the j-chain from h_{last} to t_{last}. If, on the other

hand, $h_2 > h_{\text{last}}$, then similarly to the previous case we get $(h_2, t_1) \in UPDATED_{m_p,k}$ in contradiction to the definition of h_{last}. □

Lemma 16. *The t_{last} of the last $UPDATED_{m_p,k}$ series (m_p is the highest match-point in column k) loses its single active chain and becomes inactive. This is the tail that disappears from row k of TAILS.*

Proof. Let j denote the size of the active chain from h_{last} to t_{last} in G^{k+1}. We claim that t_{last} is the tail that disappears in G^k. By Lemma 14 we know that h_{last} is de-activated in G^k. We will show that h_{last} was the only active head in H_{last} in G^{k+1}, and therefore, in G^k, t_{last} becomes inactive upon losing its last active head.

Suppose by contradiction, that t_{last} had ended another active chain in G^{k+1}. Then, by Lemma 8, the size of this additional chain would be $j+1$, and the head h_1 of this chain would therefore be higher than h_{last}. Since h_1 is not the *new* chain of t_{last}, then by Lemma 8 and the definition of h_t, the chain from h_1 to t_{last} was not the shortest chain in the span of t_{h_1} in G^{k+1}, and there was at least one additional chain of size shorter than j that started in h_1 and ended in some tail t_1 such that $t_1 < t_{\text{last}}$. This would imply, by Observation 1, that $(h_1, t_1) \in UPDATED_{m_p,k}$, in contradiction to the definition of h_{last}. □

4.7. Description and analysis of the second algorithm

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 the dynamic linked list *PAIRS* of active head-to-tail pairs. Each pair (h_i, t_i) is annotated with two fields: the row index of h_i and the column index of t_i. Upon the advancement of the computation from row $k+1$ of the *TAILS* table to row k, the graph of match-points is extended by one column to the left to include the match-points of column k of the LCS graph for A versus B. Given the list $PAIRS_{k+1}$, which is sorted by increasing head row index, the algorithm computes the new list $PAIRS_k$, obtained by merging and applying the match-points of column k to $PAIRS_{k+1}$ as described in the previous section, and the "disappearing entry" for row k of *TAILS* is finally realized.

The pseudo-code for the second algorithm is given in Fig. 16.

5. Time and space complexity

Since, by Lemma 1, $r \leqslant nL$, the total cost of merging r match-points with n lists of size L each is $O(nL)$. In iteration k, up to L_{k+1} new head row index 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 MatchLists. Thus, the second algorithm runs in $O(nL + n \log |\Sigma|)$ time. (Note that, since we only need to create MatchLists for characters appearing in B, an alphabet of size $|\Sigma| > n$ can be reduced in $O(n \log n)$ time to an n-sized alphabet. Therefore, throughout the paper we assume $|\Sigma| \leqslant n$.)

As for the space complexity, $O(L)$ key values are maintained throughout the algorithm, however the MatchLists data structures are of $O(n)$ size, and therefore the space complexity is $O(n)$.

6. Conclusions

In this paper we investigate the "evolution" of the LCS as longer suffixes are considered. In one line of investigation, the evolution of columnwise partition points is examined. This leads to a fairly simple $O(nL)$ time and space algorithm for strings A and B of length n over a constant alphabet, that computes a partition encoding of the dynamic programming matrices for the alignment of each suffix of A with B. The second line of investigation looks closely at the evolution of a well-characterized "basis set" of common subsequences, and shows that it can be tracked efficiently using $O(n)$ space. This leads to another algorithm that extends the results of the first algorithm to apply to general alphabets, and yields an $O(nL + n \log |\Sigma|)$ time, $O(n)$ space Consecutive Suffix Alignment algorithm, that 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.

```
Algorithm ConsecutiveSuffixAlignment:

input: For each character A[k], k = 1...n in A, a list of match-points
            with row index values reflecting the characters in B that match A[k].
output: The changes to TAILS in each generation.

Let M[k] denote the list of match-points for A[k] versus B, sorted in increasing row index.
Let PAIRS denote the list of (head, tail) pair items, initialized as an empty list, where each
pair item consists of two integer keys: a row index, denoted head and a column index, denoted tail.
The (head, tail) pair items in PAIRS are sorted in increasing row index.

For each column k, from n down to 1, such that M[k] is non-empty Do
1     Write: "Tail k joins TAILS in generation k";
2     Merge M[k] with PAIRS by row index key;
      For each match-point m_p ∈ M[k] in increasing row index,
          If ∃h_first such that (h_first, t_first) ∈ PAIRS
                              and m_p falls directly below h_first in the merge Then Do
3             Remove (h_first, t_first) from PAIRS;
              If m_p is the first match point in M[k] Then
4                 Create the pair item (m_p, k) and Insert it to PAIRS;
5             else Create the pair item (m_p, t_{i-1}) and Insert it to PAIRS;
6             t_{i-1} ← t_first;
              Let m_{p+1} denote the next match point above m_p in m[k];
              For each item (h_i, t_i) ∈ PAIRS in increasing order, Such That h_first < h_i < m_{p+1} Do
                  If t_i < t_{i-1} Then
7                     (h_i, t_i) ∈ PAIRS ← (h_i, t_{i-1});
8                     t_{i-1} ← t_i;
9                     h_{i-1} ← h_i;
      If there is no match-point in M[k] which falls above h_{i-1}, Then
10        Write: "Tail t_{i-1} disappears from TAILS in generation k."
      Else
          Let m_{p+1} denote the first match-point in m[k] such that m_{p+1} > h_{i-1};
11        Create the pair item (m_{p+1}, t_{i-1}) and Insert it to PAIRS;
```

Fig. 16. The pseudo-code for the second Consecutive Suffix Alignment algorithm.

We point out that both algorithms are extremely simple and practical, and use the most naive data structures. (The proof of correctness for second solution is complex and involves 11 lemmas, however, as can be seen from the pseudo-code, the algorithm itself is a simple iterative traversal of a linked list, combined with plain merge operations.)

It remains an interesting challenge to try and extend some of the new ideas and observations from this paper to other LCS-related application domains.

Acknowledgments

We thank the anonymous referees for their very thorough and helpful comments.

References

[1] A. Apostolico, String editing and longest common subsequences, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. 2, Springer-Verlag, Berlin, 1997, pp. 361–398.
[2] A. Apostolico, C. Guerra, The longest common subsequence problem revisited, Algorithmica 2 (1987) 315–336.
[3] R.P. Dilworth, A decomposition theorem for partially ordered sets, Ann. of Math. 51 (1950) 161–165.
[4] D. Eppstein, Z. Galil, R. Giancarlo, G.F. Italiano, Sparse dynamic programming I: Linear cost functions, J. ACM 39 (1992) 546–567.
[5] D.S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (4) (1977) 664–675.
[6] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Commun. ACM 20 (1977) 350–353.
[7] S. Jukna, Extremal Combinatorics with Applications in Computer Science, Texts Theoret. Comput. Sci. EATCS Ser., Springer-Verlag, ISBN 3-540-66313-4, 2001.
[8] S. Kim, K. Park, A dynamic edit distance table, in: Proc. 11th Annual Symposium on Combinatorial Pattern Matching, 2000, pp. 60–68.
[9] G.M. Landau, E.W. Myers, J.P. Schmidt, Incremental string comparison, SIAM J. Comput. 27 (2) (1998) 557–582.

[10] G.M. Landau, M. Ziv-Ukelson, On the shared substring alignment problem, Proc. Sympos. Discrete Algorithms (2000) 804–814.
[11] G.M. Landau, M. Ziv-Ukelson, On the common substring alignment problem, J. Algorithms 41 (2) (2001) 338–359.
[12] G.M. Landau, B. Schieber, M. Ziv-Ukelson, Sparse LCS common substring alignment, in: Proc. 14th Annual Symposium on Combinatorial Pattern Matching, 2003, pp. 225–236.
[13] J.P. Schmidt, All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings, SIAM J. Comput. 27 (4) (1998) 972–992.
[14] J.S. Sim, C.S. Iliopoulos, K. Park, Approximate periods of strings, in: Proc. 10th Annual Symposium on Combinatorial Pattern Matching, 1999, pp. 132–137.