Sparse Normalized Local Alignment

Nadav Efraty* Gad M. Landau†

April 18, 2005

Abstract

Given two strings, X and Y both of length $O(n)$ over alphabet Σ, a basic problem (*local alignment*) is to find pairs of similar substrings, one from X and one from Y. For substrings X' and Y' from X and Y, respectively, the metric we use to measure their similarity is *normalized alignment value*: $LCS(X', Y')/(|X'| + |Y'|)$. Given an integer M we consider only those substrings whose LCS length is at least M. We present an algorithm that reports the pairs of substrings with the highest normalized alignment value in $O(n \log |\Sigma| + rM \log \log n)$ time ($r-$ the number of matches between X and Y). We also present an $O(n \log |\Sigma| + rL \log \log n)$ algorithm ($L = LCS(X, Y)$) that reports all substring pairs with a normalized alignment value above a given threshold.

1 Introduction

Sequence comparison is an extensively studied topic. Many textbooks are devoted to the subject [5, 9, 10, 12, 19, 20]. Its applications are numerous and include areas such as file comparison, search for similarity between bio-sequences, information retrieval and XML querying, music retrieval, image comparison and an almost infinite number of other sequence comparison applications.

While for applications such as the comparison of protein sequences the methods of scoring can involve arbitrary scores for symbol pairs and for gaps among unaligned symbols, for uses in other contexts such as text comparison or screening sequences, simple unit score schemes suffice. Two of these, the *Longest Common Subsequence* (LCS) and the *edit distance* measures, have been studied extensively, for the unit cost nature of their scoring provides combinatorial leverage not found in the more general framework [6, 13, 14, 16].

The *Longest Common Subsequence* measures the length of the longest identical subsequence of the two strings. *Edit distance* measures the minimal number of operations that are required to transform one string into the other one, when the permitted operations are substitution, deletion, and insertion. The goal is to find such a sequence of operations of minimal length or of minimal

*Department of Computer Science, University of Haifa, Haifa 31905, Israel, phone: (972-4) 828-8367, FAX: (972-4) 824-9331; email: nadave@cs.haifa.ac.il; partially supported by the Israel Science Foundation grant 282/01, and by the FIRST Foundation of the Israel Academy of Science and Humanities.

†Department of Computer Science, University of Haifa, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840; email: landau@poly.edu; partially supported by NSF grant CCR-0104307, by the Israel Science Foundation grant 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities, and by the IBM Faculty Award.

value when each edit operation has an assigned value. These two algorithms basically rely on dynamic programming techniques.

As the LCS and *edit distance* algorithms evolved, the notion of the sparsity of the essential data in the dynamic programming table became the key to the acceleration of the algorithms.

The evolution of the LCS algorithms can be tracked by examining [4, 6, 11, 18], which can be regarded as the successors of the classic LCS algorithms of Hirschberg [13] and Hunt and Szymanski [14]. For example, Eppstein et. al. [11] presented an $O(n \log |\Sigma| + d \log \log (\min \{d, nm/d\}))$ algorithm, where n and m are the sizes of the input strings, $|\Sigma|$ is the size of the alphabet and d is the number of dominant matches.

While the LCS and *edit distance* algorithms are measures of the global similarity between two strings, in many applications, two strings may not be very similar in their entirety, but may contain regions that are very similar. The task is to find and extract a pair of regions, one from each of the two given strings, that exhibit a strong degree of similarity. This is called the *local similarity* or the *local alignment* problem, as defined in [12]:

Definition 1 *Given two strings, X and Y, find substrings X' and Y' of X and Y, respectively, whose similarity (optimal global alignment value) is maximum over all pairs of substrings from X and Y.*

This definition can be extended to finding not only the most similar substrings, but also, all pairs of substrings whose similarity value exceeds a certain similarity level (e.g. 80%).

The *local similarity* problem is, in many senses, more challenging than that of the *global similarity*. There are no clear starting and ending points, so any entry has the potential of being the first or the last of an optimal alignment. In addition, one single match is always a perfect local alignment. Thus, a local alignment algorithm might report all the matches as optimal alignments, while longer and more meaningful alignments that are imperfect will not be reported.

One of the most important and commonly used local comparison techniques was introduced by Smith and Waterman [21]. Their algorithm is broadly used in molecular biology, as well as in other fields where local sequence comparison is practiced.

According to a recent paper by Arslan, Eğecioğlu and Pevzner [7], the Smith Waterman algorithm has two weaknesses that make it non optimal as a similarity measure. The first weakness is called the *mosaic effect*. This term describes the algorithm's inability to discard poorly conserved intermediate segments, although it can discard poor prefixes or suffixes of a segment. The second weakness is known as the *shadow effect*. This term describes the tendency of the algorithm to lengthen long alignments with a high score rather than shorter alignments with a lower score and a higher degree of similarity.

Some of the extensive work on the *Smith Waterman* algorithm as a measure of similarity is detailed in [1, 2, 3, 7, 23]. According to these studies, the normalization of the values of the alignments by their length will yield a better measure of the similarity level of the local alignments.

Definition 2 *The normalized alignment value of two substrings, X' and Y', is $S(X', Y')/\ (|X'| + |Y'|)$, where S is the global alignment value (of X' and Y') according to one of the scoring schemes.*

Arslan et al. [7] suggested a measure designed for the purpose of finding the most similar pair of

substrings whose length is significant. The measure is based on the reformulation of the above definition of the normalized alignment's value. Their definition is as follows:

Definition 3 *The normalized alignment value of two substrings is* $S(X', Y')/(|X'| + |Y'| + L)$, *where* X' *and* Y' *are substrings of* X *and* Y, $S(X', Y')$ *is the global maximal score of the alignment of* X' *and* Y', *and* L *is a positive number that controls the amount of normalization.*

The ratio between L and $(|X'| + |Y'|)$ determines the influence of L on the value of the normalized sequence alignment under that metric. For short alignments it might lower the normalized sequence value dramatically, while for long alignments the effect on the value should be minor. Using this measure, it is less likely that short alignments will receive high normalized sequence alignment values.

The weakness in this, otherwise effective, approach for discarding alignments whose length is insufficient is the reformulation of the original definition of the normalized value (definition 2). By altering the definition, the outcomes will accordingly be different than the expected outcomes of the original problem under the original definition.

The time complexity of the measure, suggested by Arslan et al., is $O(n^2 \log n)$, where n is the size of the input strings. The space complexity of this measure, which is $O(n^2)$, is derived from the space complexity of the *Smith Waterman* algorithm, which is utilized repeatedly in order to compute the values of $S(X', Y')$.

In this paper, we present an algorithm designed for the computation of the local similarity normalized values of substrings of the two input strings whose lengths are not too short to be of significance. The presented algorithm utilizes the LCS metric for the computation of the normalized local alignment value and exploits the sparsity of the essential data in the dynamic programming tables.

Definition 4 *An entry* (i, j) *in the dynamic programming table of two sequences,* $|X| = n$ *and* $|Y| = m$, *is called a match if and only if* $X_i = Y_j$. *The number of such entries in the table is denoted by* r *where obviously,* $r \leq nm$ *[4].*

The LCS, which is a global measure rather than a local measure, is made into a local measure of similarity by dividing the LCS value of the two substrings by the sum of their lengths. Substrings that maximize that value are the most similar. Note that the alignment with the highest similarity level must begin and end in a match. Otherwise, there is a better alignment with the same LCS and a lower value of $|X| + |Y|$.

Though the LCS appears, at first glance, to be a less powerful metric if compared with other scoring schemes, it can be used to capture alignments whose matches density ratio is high, indicating that their similarity level is high. Clearly, the LCS metric with its simple scoring scheme is sufficient to solve numerous problems from a variety of domains. Indeed, even applications and problems that utilize more complicated scoring schemes, such as comparisons of protein sequences, may benefit from this algorithm.

Let X' and Y' be substrings of the input strings X and Y, respectively. A minimal length constraint, denoted herein by M, may be either a minimal length constraint on the sum of the lengths $|X'|+|Y'|$, or a minimal length constraint on the length of the longest common subsequence (LCS) of X' and

Y'. We chose to refer to the constraint on the LCS of X' and Y' because it better suits an algorithm that exploits the sparsity of the matches in the dynamic programming tables.

The minimal length constraint (M) is enforced in a straightforward fashion, without the need to reformulate the original problem that in the case of normalized LCS is $LCS(X', Y')/(|X'| + |Y'|)$. The value of that minimal constraint is expected to be problem related rather than input related, and it is expected to be of a much smaller scale than the lengths of the input strings.

Definition 5 *$Best(M)_Y^X$ - the highest normalized alignment value of any of the substrings pairs, of strings X and Y, with LCS value higher than M.*

Results:

Given two strings X and Y each of length n, and a length constraint M, we will introduce two algorithms that compute the value of $Best(M)_Y^X$. The first algorithm is discussed thoroughly in section 2. This normalized local LCS algorithm reports substring pairs that achieve the value $Best(M)_Y^X$ and whose common subsequence is longer than M. Alternatively, it may output substring pairs whose similarity is higher than a predetermined value and whose common sequence is longer than M. The time complexity of that algorithm is $O(n \log |\Sigma| + rL \log \log n)$ and its space complexity is $O(rL + nL)$ where $L = LCS(X, Y)$.

The second algorithm, discussed in section 3, is similar to the first in its ability to compute the normalized value of $Best(M)_Y^X$, as well as the substring pairs that achieve that value. The time and space complexity of that algorithm are $O(n \log |\Sigma| + rM \log \log n)$ and $O(rM + nM)$, respectively.

Since we expect M to be much smaller than L, the second algorithm is more efficient than the first one. But, it does not report long substring pairs whose similarity exceeds a predetermined value, if this value is lower than the normalized value of $Best(M)_Y^X$.

Note that for 100% similarity, we demand that $LCS(X, Y) = |X| \ \wedge \ LCS(X, Y) = |Y|$; thus, the normalized value is $\frac{1}{2}$. Similarly, any normalized value v represents a similarity level that is $200 \times v$.

Our algorithms avoid the shadow and mosaic effects. The shadow effect is avoided since for any number of matches, the shortest alignment is constructed. Longer alignments would be preferable over shorter alignments only if the longer ones contain more matches, and their normalized value is higher. The mosaic effect is avoided since the normalized value of a sufficiently long alignment with a poor intermediate segments would be lower than the normalized values of its prefix and suffix, which are computed separately.

2 The $O(rL \log \log n)$ normalized local LCS algorithm

In this section we discuss our basic algorithm for the computation of $Best(M)_Y^X$ and for the computation of the alignments that exceed a certain similarity level, using the LCS metric. The discussion begins with the definitions and lemmas that are needed for the understanding of the algorithm. Each of the major stages of the algorithm, as well as the complexity analysis, will be discussed in a separate subsection. Finally, an alternative algorithm would be discussed.

The input is two strings, $|X| = n$ and $|Y| = m$ ($m = O(n)$). As in [4], our algorithm constructs a data structure that substitutes the dynamic programming tables that are used by other local

similarity algorithms. Implicitly, many of the properties of the dynamic programming tables are maintained in our sparse representation of it. A match (i, j) is a match that will be in entry (i, j) in the analogous dynamic programming table.

A *chain* was defined in [8] as a sequence of matches that is strictly increasing in both components, i.e., two matches (i, j) and (i', j') may be part of the same chain if and only if $(i < i' \wedge j < j') \vee (i > i' \wedge j > j')$. Let us present the extended definition of a chain that will be used throughout this work.

Definition 6 *A k-$Chain_{(i',j')}^{(i,j)}$ denotes a sequence of k matches that is strictly increasing in both components, whose head is the match (i, j) and whose tail is the match (i', j').*

- *$k = LCS(X_{j...j'}, Y_{i...i'})$. $X_{j...j'}$ and $Y_{i...i'}$ are substrings of the input strings X and Y, respectively.*
- ***Length of** k-$Chain_{(i',j')}^{(i,j)}$**:** The length is the sum of the lengths of $X_{j...j'}$ and $Y_{i...i'}$ (i.e. $j' - j + i' - i$).*
- *k-$Chain^{(i,j)}$ denotes the best chain of k matches starting from (i, j), i.e., the chain of the shortest possible length that has k matches.*
- ***Normalize value of** k-$Chain_{(i',j')}^{(i,j)}$**:** The normalized value is $\frac{k}{j'-j+i'-i}$.*

For each match (i, j), the algorithm constructs k-$Chain^{(i,j)}$ for every possible value of k ($1 \leq k \leq LCS(X, Y)$). The algorithm starts by marking the positions of the matches between the input strings. Later, the matches are processed in decreasing row number order (bottom to top). The processing of each row has two stages.

1. First stage: The algorithm constructs the best k-*Chains* of any possible value of k, starting from each of the matches in the row. This is done using data structures that were prepared during the processing of previous rows.
2. Second stage: The matches of the processed row and additional information regarding their k-*Chains* are inserted into the data structures, in order to prepare them for future use during the processing of the succeeding rows.

A major obstacle in the process of constructing k-*Chains* is that any attempt to construct $(k+1)$-$Chain^{(i,j)}$ simply by tying another match to the tail of k-$Chain^{(i,j)}$ (which is the best chain of k matches starting from (i, j)) will not necessarily produce optimal results, as seen in figure 1 where the 2-*Chain*, 3-*Chain* and 4-*Chain* of the match $(2, 2)$ do not share common matches. One way to deal with that difficulty is to try to add one match to the tail of all the possible chains of k matches starting from (i, j). This solution would, indeed, construct $(k+1)$-$Chain^{(i,j)}$, but it may prove overly complex. We take the opposite approach. From among all of the k-*Chains* that start lower than and to the right of (i, j), we choose the one that, when concatenated to (i, j) as its head, creates $(k+1)$-$Chain^{(i,j)}$. The following lemma proves the correctness of this strategy.

Lemma 1 *For any given value of k, and for a match (i, j), $(k+1)$-$Chain^{(i,j)}$ is a chain that starts from (i, j) and continues with k-$Chain^{(i',j')}$, $i' > i \wedge j' > j$.*

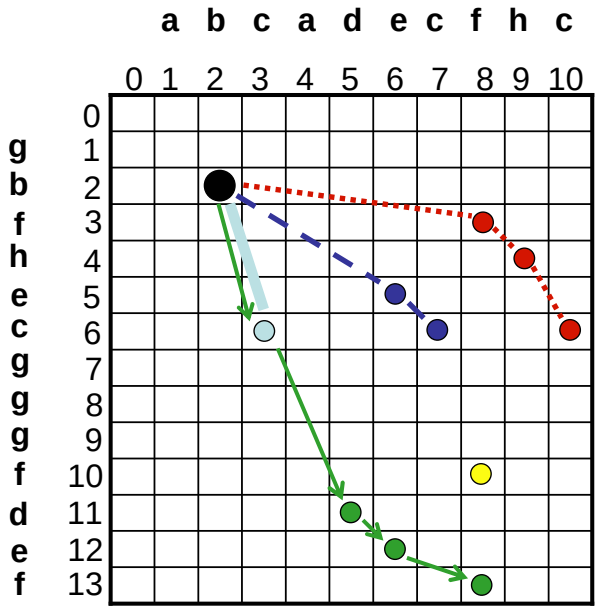

Figure 1: The dynamic programming table of the strings $abcadecfhc$ (X) and $gbfhecgggfdef$ (Y). The matches are marked as circles. 2-$Chain^{(2,2)}_{(6,3)}$, which is 2-$Chain^{(2,2)}$, is marked with a solid line. Its length is $6-2+3-2=5$ and its normalized value $\frac{2}{5}$. 3-$Chain^{(2,2)}$, 4-$Chain^{(2,2)}$ and 5-$Chain^{(2,2)}$ are marked with dashed lines, dotted lines, and arrowed lines, respectively.

Proof: Assume that instead of using k-$Chain^{(i',j')}$ we use another chain of k matches starting from (i',j') which yield a better chain of $k+1$ matches for (i,j). Since the length of the chain from (i,j) to (i',j') remains identical, regardless of the k matches' suffix starting from (i',j'), the difference in the length between two potential chains depends only on the length of the chain of k matches starting from (i',j'). Thus, if $(k+1)$-$Chain^{(i,j)}$ passes through (i',j'), but its suffix is different than k-$Chain^{(i',j')}$, it implies that we have constructed a better chain of k matches starting from (i',j'), thereby contradicting the definition of k-$Chain^{(i',j')}$ (definition 6). ■

The above lemma provides a simple $O(r^2L)$ time complexity algorithm for the problem. For each match (i,j), this algorithm would construct $(k+1)$-$Chain^{(i,j)}$, $1 \leq k < L$, by examining all $O(r)$ potential heads of k-$Chains$ and tying (i,j) to the most appropriate k-$Chain$. The next subsection will demonstrate how to improve that time complexity by narrowing the search performed by (i,j) to a single match which must be the head of the appropriate k-$Chain$.

Let us present the skeleton of the algorithm. The first stage, which is the preprocessing stage, is similar to the typical preprocessing of the sparse LCS algorithms [4]. Its output is a list of the different symbols of Σ, where each symbol has a list of the indices of its appearances in the input string X. After executing this stage, we can view the matches of each row i by examining the list of symbol $\sigma = Y_i$ ($\sigma \in \Sigma$). The two stages of the algorithm and the $Report\_Best(M)^X_Y$ procedure will be discussed in the following subsections.

$O(rL \log\log n)$ **normalized local alignment algorithm**

For each row, corresponding to a symbol Y_i, create an ordered list of the matches in the row.
$i \leftarrow m$

Repeat until $i = 0$
    $k \leftarrow 1$

    Stage one
    Repeat while chains with growing k values are constructed
        $Construct\_(k+1)\text{-}Chains(matches\ of\ row\ i, k)$
        $k \leftarrow k + 1$

    Stage two
    Repeat while $k > 0$
        $Insert\_Matches(matches\ of\ row\ i, k)$
        $k \leftarrow k - 1$

    $i \leftarrow i - 1$

$Report\_Best(M)_Y^X$

2.1 Stage two - The creation and updating of ranges

The purpose of this stage is to insert the chains that were constructed during the first stage into a data structure that will enable us to narrow the search performed by each of the succeeding matches to a single k-*Chain*. L data structures are maintained for k-*Chains* of each number of matches k ($1 \leq k \leq L$). Our discussion commences with formal definitions of the intuitive concepts of range and owner.

Definition 7 Range: *A range of a match (i, j) is an area of the dynamic programming table that stretches from column $j-1$ and to the left and from row $i-1$ and above, i.e., it is $(i'...i-1, j'...j-1)$ for each i' and j', $0 \leq i' < i \wedge 0 \leq j' < j$. Hence each match has $i \times j$ such ranges.*

Definition 8 Mutual range: *The range of one match may partially or fully contain a range of another match. The overlap area that is part of the range of both of the matches is called a mutual range.*

Definition 9 Owner of a range: *The match (i, j) is the owner of a range if k-$Chain^{(i,j)}$ is the suffix of all $(k+1)$-Chains that start inside the range.*

L separated lists of ranges and their owners are maintained by the algorithm. The following lemma provides the key to determining the correct ranges and their owners in each of these lists.

Lemma 2 *A mutual range of two matches is owned completely by one of them.*

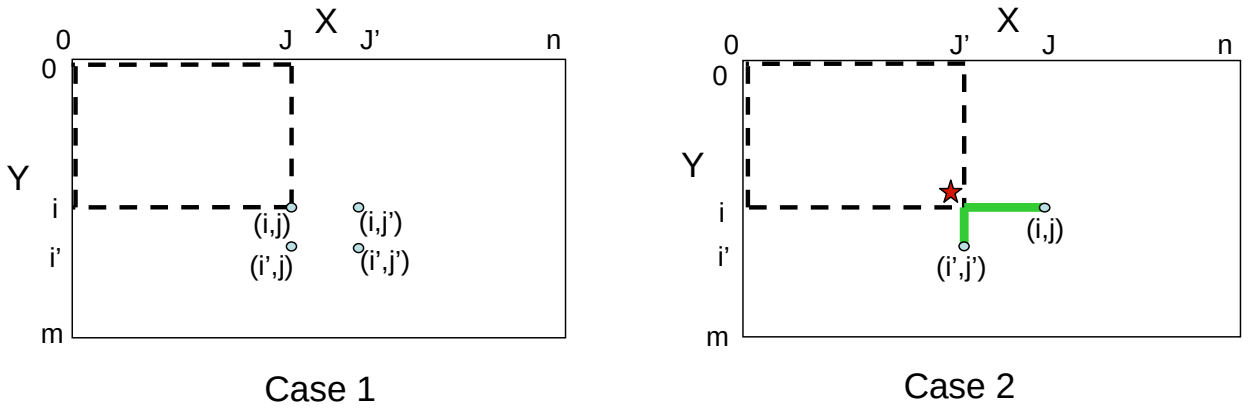

Figure 2: The two cases from lemma 2. In the figure representing case 1, the range that is surrounded by the dashed line is owned by (i, j). In the figure representing case 2, the mutual point is marked with a star and the mutual range is surrounded by a dashed line.

Proof: The k-*Chain* that is headed by a match (i, j) may be the suffix of any $k + 1$ matches chain starting from any of the matches in the ranges of (i, j). Note, however, that these chains are not necessarily the $(k + 1)$-*Chain* of these matches. For all matches that are in a range of a single match (i, j) (i.e., they are not in a mutual range), the only way to construct a $(k + 1)$-*Chain* is to pass through (i, j). Thus, (i, j) will be the owner of that range. Let us deal with the two different settings of two matches that share a mutual range. These matches will be p (i, j) and q (i', j').

1. $i \leq i' \wedge j \leq j'$**:** The mutual range of p and q is $(0...i-1, 0...j-1)$. According to their positions, p may use the $k-1$ suffix of k-$Chain^q$ as part of a possible k-*Chain* from it. Hence, for each match in the mutual range, a $(k + 1)$-*Chain* through p is either equal to or better than the chain through q. Thus, p owns the mutual range.

2. $i < i' \wedge j > j'$**:** The mutual range of p and q is $(0...i - 1, 0...j' - 1)$. Let us define the entry $(i - 1, j' - 1)$ as the **mutual point (*MP*)** of p and q. MP is the bottommost and rightmost entry of the mutual range, and it is not a match. The length of the chain from any match z in the mutual range to either p or q is equal to the length of the chain from z to MP (which is equal for both p and q) plus the length of the chain from MP to either p or q (for match z in coordinates (i'', j''), $i'' < i \wedge j'' < j'$, the length of the chain to p is $(i - i'') + (j - j'')$, and the length of the chain from z to p that passes through MP is $(i - (i - 1) + j - (j' - 1)) + ((i - 1) - i'' + (j' - 1) - j'') = (i - i'') + (j - j''))$. Since the distances from MP to both p and q are predetermined (they are $j - j' + 2$ and $i' - i + 2$ for p and q, respectively), the one whose tail is closer to MP also forms a shorter chain with any match z in the mutual range. Let the length of k-$Chain^p$ be L^p and the length of k-$Chain^q$ be L^q. p will be the owner of the mutual range if $L^p + (j - j') \leq L^q + (i' - i)$ and q will be its owner otherwise. ■

Observations:

1. For the given matches $p\ (i,j)$ and $q\ (i',j')$, such that $i < i' \wedge j > j'$, and for the given lengths L^p and L^q of $k\text{-}Chain^p$ and $k\text{-}Chain^q$, respectively, if $L^p + (j - j') > L^q + (i' - i)$ then the owner of the mutual range is q and the range owned by p is blocked from the left by the range of q. If $L^p + (j - j') < L^q + (i' - i)$, then the owner of the mutual range is p and the range owned by q is blocked from row i and above by the range of p. Since the algorithm processes the matches in decreasing row number order, matches whose row coordinate value is higher than i will not be processed later by the algorithm. Thus, the range owned by q (i.e., $(i...i'-1, j''...j')$, $j'' < j$) is no longer relevant, and it would not become relevant later. No range above row i would be owned by q, and therefore, it may be extracted from the data structure of the heads of $k\text{-}Chains$. In the case of an equality ($L^p + (j - j') = L^q + (i' - i)$), we prefer p over q as the owner of the mutual range because it gives us the opportunity to extract q from the data structure without the loss of important information.

2. The range owned by any match (i,j) is $(0...i-1, j'...j-1)$, $0 \leq j' < j$. The range always reaches row 0 because if the range is completely blocked from above at row $i' < i$, then for any match above it this range is no longer relevant. The range is extracted, and therefore ceases to exist. If the range is partially blocked from above at row $i' < i$ and column $j'' < j$ (see the first setting in the above lemma), the range $(0...i-1, j''...j-1)$ which is equal to the right part of the of the original range, still reaches row 0.

3. For a given group of matches that are the heads of $k\text{-}Chains$, the matches whose row number is the lowest (at a given time) must own (at that time) the ranges that stretch between their row and row 0.

The data structure: LRO^k denotes the list of ranges and their owners that are the heads of $k\text{-}Chains$. Such a list is maintained for each value of k, $1 \leq k \leq L$. Each such list of range owners is ordered by the column. The range of an owner in LRO^k, whose position is (i,j), is $(0...i-1, j'...j-1)$, where $j' < j$ is the column of the left neighbor of (i,j) in LRO^k. An example of an LRO^k is given in figure 3. In addition to each owner, we keep the length of the $k\text{-}Chain$ starting from it.

The LRO^ks are implemented as Johnson Trees [15]. Explicitly, LRO^k is held in data structures for integers in the range $[0,n]$. These data structures support the operations insert, extract and look for the range that a given match is in.

The algorithm processes the rows in decreasing row number order. Thus, row i is processed only after rows m to $i+1$ were processed and matches that are the heads of $k\text{-}Chains$ were inserted into LRO^k. When the match $p\ (i,j)$, which is the head of a $k\text{-}Chain$, is processed, then according to observation 3 above, it will always be inserted into LRO^k as the range whose right boundary is column $j-1$. Later, the following update operations are performed in LRO^k:

- Right boundary: If LRO^k has another match $q\ (i',j')$ such that $i < i' \wedge j = j'$, then by lemma 2, the range of q that is above row i is owned completely by p and thus, q is extracted from LRO^k.

- Left boundary: The left neighboring range, whose owner is $q\ (i',j')$ $(i' \geq i \wedge j' < j)$, is examined. If $i' = i$, the left boundary of the range of p is j' (lemma 2, case 1). If $i' > i$, we

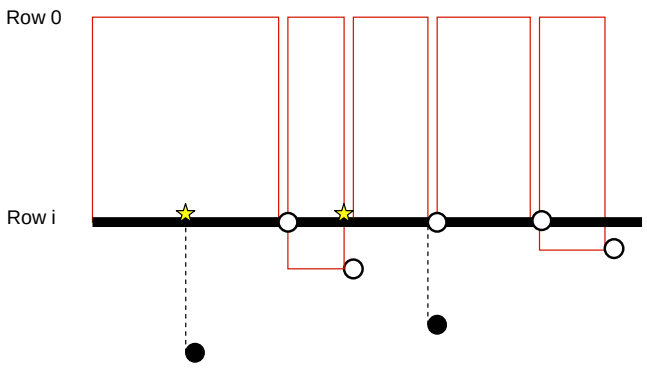

Figure 3: The LRO^k. Matches that are heads of k-$Chains$ are marked by circles. The white circles are the owners of the ranges that are in LRO^k. Each white circle is the owner of the range to its left. The black circles are owners that were extracted from LRO^k. The stars represent the mutual points, where the boundary of ranges were set according to lemma 2, case 2.

use observation 1 to determine the owner of the mutual range of p and q. If q is the owner of the mutual range, it sets the left boundary of the range of p. If p is the owner of the mutual range, q is extracted from the data structure (implicitly, the range of p was extended) and the left neighbor of q is examined in the same fashion.

$Insert\_Matches(matches\ of\ row\ i, k)$

Repeat until all matches of row i that are the heads of k-$Chains$ are inserted into LRO^k.

Insert the match (i, j) into LRO^k in the appropriate position for j.

If LRO^k has a previous match with column coordinate j, then extract it.

Repeat while for (i', j'), which is the left neighbor of (i, j) in LRO^k,
(the length of k-$Chain^{(i',j')} + i' - i$) $\geq$ (the length of k-$Chain^{(i,j)} + j - j'$)
Extract (i', j') from LRO^k.

2.2 Stage one - The construction of $(k+1)$-$Chains$

In this stage, we will compute the $(k+1)$-$Chains$ of all matches of row i, where $1 \leq k \leq L$ and $1 \leq i \leq m$. The input for this stage is the list of ranges and their owners (LRO^k) that were computed for rows m to $i+1$ and were discussed in the previous subsection.

For a match p, $(k+1)$-$Chain^p$ is constructed simply by concatenating p to the match q, which is the owner of the range containing p. Explicitly, q is the match in LRO^k whose column coordinate is the closest to that of p from the right.

The data structure: All the matches are ordered according to their positions. Every match has

information regarding all the k-$Chain$, $1 \leq k \leq L$, starting from it. For a given match p, the data structure maintains a record where for any given k value, the length of k-$Chain^p$ is recorded, along with a pointer to a match q, such that $(k-1)$-$Chain^q$ is the suffix of k-$Chain^p$. Owners of ranges that were extracted from LRO^k are not deleted from that data structure.

Construct_*$(k+1)$-Chains*$(matches\ of\ row\ i, k)$

Repeat until all matches of row i are processed.

Add the k's element of the list of (i, j).
- Its pointer points to the match (i', j'), the owner of the range of (i, j) in LRO^k
- Its length value = length of k-$Chain^{(i',j')} + (i' - i + j' - j)$

2.3 $Report\_Best(M)_Y^X$

After the matches of row 1 have been processed, the data structure wherein every match p has a record with all of the k-$Chain^p$ and their lengths, is completed. Now, the records of all of the matches are examined, the normalized value of any of the k-$Chains$, $k \geq M$, is computed, and the highest valued k-$Chain$, $Best(M)_Y^X$, and its normalized value are computed. $Best(M)_Y^X$ and its corresponding substrings X' and Y' of the input strings X and Y, respectively, may be reported by traversing the pointers of the data structure of matches.

Alternatively, it is possible to report all of the chains and the corresponding substrings whose normalized value is higher than a given normalized value, e.g. 80%. Such sequences may also be reported on the fly during the operation of the algorithm.

2.4 Complexity Analysis

Let us analyze the complexity of each of the stages of the algorithm.

Preprocessing stage: The complexity of the preprocessing stage is $O(n \log |\Sigma|)$, $|\Sigma| \leq m$, and the collective space consumed for the lists of all individual symbols is $O(n)$. This stage is similar to the typical preprocessing of the sparse LCS algorithms [4].

First stage: During the first stage of the processing of each match, attempts are made to construct k chains, $1 \leq k \leq L$, where $L = LCS(X, Y)$ is the highest possible number of matches in any of the chains. Each such attempt requires one query for the nearest neighbors on each of the corresponding LRO^ks. The LRO^ks are implemented as Johnson Trees [15]. The time complexity of each such query is $O(\log \log G)$, where G is the gap between the integer that was the subject of the operation (i.e., the column number of the processed match) and its right and left neighbors in the list. In such lists when a pointer to one of the owners of the ranges is given, its predecessor and successor are reported in $O(1)$ time complexity because a connected list of the owners of ranges is also maintained. The space complexity of such a tree is $O(n)$. Since it is difficult to assess the mean value of G because of the constant changes in LRO^k, we refer to it as n. For all practical purposes, however, the mean value of G is lower than n. Hence, the total complexity of all the iterations of all the r matches is $O(rL \log \log n)$.

Second stage: Each match is inserted and extracted no more than once from each of the LRO^ks. The total time complexity of this entire operation is again $O(rL \log\log n)$.

Report_Best$(M)_Y^X$: For the retrieval of the highest normalized value and for the construction of the optimal sequence (or the corresponding substrings), the algorithm must examine all the elements in the record of each match with a total time complexity of $O(rL)$.

Henceforth, the time complexity of the algorithm is $O(n \log|\Sigma| + rL \log\log n)$

The space complexity is $O(rL + nL)$. It is dictated by the size of the data structure for the matches where each match has a record with pointers to no more than L other matches, with one additional length value recorded with each such pointer, and the space needed for L LRO^k data structures that are, in fact, Johnson Trees of $O(n)$ space each.

2.5 An alternative algorithm for the management of the data structure

An alternative technique for managing the LRO^ks that enables both queries and update operations and does not defer the time complexity of the above algorithm was presented in Mäkinen [17]. As in the algorithm presented above, this technique is based on the insertion of matches that are the heads of k-*Chains* into an array $A^k[1...n]$ wherein the match in each column (if any) is the one with the lowest row coordinate among the matches of the column that have already been processed. This is done according to the first case of lemma 2.

In order to construct $(k+1)$-*Chains*, queries are made in the array of k-*Chains*. Each such query is, in fact, a range minimum query (RMQ) in the k's array, where the range for the query is $[j+1, n]$ for a query of a match (i, j).

$(k+1)$-$Chain^{(i,j)}$ is obtained by finding a match (i', j') that is the head of a k-*Chain*, such that the sum of the distance between (i, j) and (i', j'), plus the length of k-$Chain^{(i',j')}$, denoted by $L^{(i',j')}$, is the minimum possible. Formally, we wish to find a match (i', j') such that the expression $i' - i + j' - j + L^{(i',j')}$ is minimized.

Let us rearrange the expression $i' - i + j' - j + L^{(i',j')}$ to $[i' + j' + L^{(i',j')}] - [i + j]$. In the later expression, it is clear that the right (left) side of the expression depends only on the match (i, j) ((i', j')). The value of the right (left) side remains the same, regardless of the match (i', j') ((i, j)). Thus, to minimize the expression, all that is necessary is to find a match (i', j') from the array of matches that are the heads of k-*Chains* which minimizes the expression $[i' + j' + L^{(i',j')}]$. After that minimal value and its corresponding match (i', j') are found, we need only to sum the value of the left side of the expression with that of the right side in order to compute the length of $(k+1)$-$Chain^{(i,j)}$.

Let A^k denote the array of matches that are the heads of k-*Chains*, $A^k[j'] = [i' + j' + L^{(i',j')}]$. Finding the position in the array with the minimum value is analogous to finding the match (i', j') which minimizes the expression $i' - i + j' - j + L^{(i',j')}$.

Time complexity: According to [17], the position with the minimum value is reported through a one dimensional range minimum query. Such queries may be performed in $O(\log\log n)$ time if the data structure in use is a Johnson Tree. An insertion of a match into the Johnson Tree is also performed in $O(\log\log n)$ time.

To conclude, the complexity of the algorithm presented in this subsection is identical to that of the

algorithm presented in the previous subsections.

3 The $O(rM \log\log n)$ normalized local LCS algorithm

In this section we present an algorithm for the computation of the normalized value of $Best(M)_Y^X$. Such an algorithm may be ideal for screening input strings that do not reach a desired similarity level. Later, we will show that this algorithm may actually do more than just compute the normalized value of $Best(M)_Y^X$. It may also be used to construct the longest chain that is $Best(M)_Y^X$.

The algorithm that was presented in the previous section is capable of computing $Best(M)_Y^X$ and its corresponding normalized value by constructing the k-$Chains$, $1 \leq k \leq LCS(X,Y)$, starting from each of the matches. In this section we will prove that constructing k-$Chains$ for $k \leq 2M-1$ is sufficient for the computation of the value of $Best(M)_Y^X$.

Let us start with the definition of a *sub-chain*, that will be followed by the claim that the normalized value of a chain cannot be higher than the normalized value of its best sub-chain.

Definition 10 ***sub-chain:*** *A sub-chain of a k-Chain is a path that contains a sequence of $x \leq k$ consecutive matches of the k-Chain.*

Note that unlike a k-$Chain$, which always starts and ends with a match, any sub-chain, except the first and the last of a given k-$Chain$, may start and end at any entry of the chain, even if it is not a match. The first sub-chain, which is the prefix of the k-$Chain$, always starts at the head of the k-$Chain$, and the last sub-chain, which is its suffix, always ends at the tail of the k-$Chain$.

Note also that a sub-chain of x matches has a normalized value that is less than or equal to the normalized value of the x-$Chain$ comprised of the same matches, since the sub-chain may have an additional length (at its front and rear).

According to definition 6, the normalized value of a given k-$Chain$ whose length is ℓ is $\frac{k}{\ell}$. Let us split this k-$Chain$ into any number $\leq k$ of non overlapping consecutive sub-chains, such that $k = \sum k_i$ and $\ell = \sum \ell_i$. Hence, $\frac{k}{\ell} = \frac{\sum k_i}{\sum \ell_i}$. The normalized value of each such sub-chain is $\frac{k_i}{\ell_i}$.

Claim 1 $\frac{k}{\ell} \leq \max(\frac{k_i}{\ell_i})$.

Proof: Let $\frac{k_{i^*}}{\ell_{i^*}} = \max(\frac{k_i}{\ell_i})$. Thus, for any i, $\frac{k_i}{\ell_i} \leq \frac{k_{i^*}}{\ell_{i^*}}$. The value of ℓ_i that represents the length of the i's sub-chain must be positive, hence, $\frac{k_i}{\ell_i} \leq \frac{k_{i^*}}{\ell_{i^*}} \rightarrow k_i \times \ell_{i^*} \leq k_{i^*} \times \ell_i$. Since it holds for any i, we get $\sum(k_i \times \ell_{i^*}) \leq \sum(k_{i^*} \times \ell_i)$. Hence, $\frac{k}{\ell} = \frac{\sum k_i}{\sum \ell_i} \leq \frac{k_{i^*}}{\ell_{i^*}} = \max(\frac{k_i}{\ell_i})$. ■

Note that if $\frac{k}{\ell} = \max(\frac{k_i}{\ell_i})$, then for any sub-chain, $\frac{k_i}{\ell_i} = \frac{k}{\ell}$.

According to claim 1, constructing all of the short sub-chains is sufficient to find the value of $Best(M)_Y^X$. Very short sub-chains may have normalized values that are extremely high (e.g., if we consider 1-$Chains$, then each such chain would have a normalized value of $\frac{1}{2}$ which is equal to 100% similarity) but do not reflect significant similarity between the input strings. Thus, in order to compute the value of $Best(M)_Y^X$, it is necessary to construct sub-chains of at least M matches.

Lemma 3 *Constructing all* $(2M-1)$*-Chains is sufficient for the computation of the value of* $Best(M)_Y^X$.

Proof: Any k*-Chain* $(k \geq M)$ can be split into consecutive non overlapping sub-chains of M to $2M-1$ matches. Chains with less than M matches are not sufficient, and $(2M-1)$*-Chains* can not be split to sub-chains of at least M matches. According to claim 1, the normalized value of the k*-Chain* is not better than the normalized value of its best sub-chain. ■

This concludes our claim that by constructing chains of no more than $2M-1$ matches, the algorithm can report the value of $Best(M)_Y^X$. Now, let us turn to the claim that the $O(rM \log\log n)$ algorithm may also be used to report the longest chain that is $Best(M)_Y^X$.

When the normalized value of $Best(M)_Y^X$ equals $\frac{1}{2}$ (100% similarity), the $Best(M)_Y^X$ chains and the corresponding substring alignments can be found using the suffix tree of the two input strings. The construction of such a suffix tree is accomplished in $O(n \log(\Sigma))$ time [22]. In fact, it may be worthwhile to construct a suffix tree and check whether there is a substring of at least M matches that is common to both the input strings even before we turn to the $O(rM \log\log n)$ algorithm for the computation of the normalized value of $Best(M)_Y^X$.

We will prove that when the normalized value of $Best(M)_Y^X$ is lower than $\frac{1}{2}$, the longest $Best(M)_Y^X$ will be a chain of no more than $2M-1$ matches. This would imply that the $O(rM \log\log n)$ algorithm is also sufficient for the construction of the longest $Best(M)_Y^X$.

Lemma 4 *If the normalized value of* $Best(M)_Y^X$ *is lower than* $\frac{1}{2}$*, the longest* $Best(M)_Y^X$ *is a chain of no more than* $2M-1$ *matches.*

Proof: Consider a chain with more than $2M-1$ matches with normalized value $Best(M)_Y^X$, denoted by LB.

- According to lemma 3, we may split LB into a number of sub-chains of M matches, followed by a single sub-chain of between M and $2M-1$ matches.
- According to claim 1, the normalized value of each of these sub-chains must be equal to the normalized value of LB.
- According to the definition of a sub-chain (definition 10), if one of the above sub-chains of LB does not start or end with a match, the chain comprised of the same matches has a normalized value that is higher than that of the sub-chain, and thus, higher than the normalized value of LB itself. Hence, all of these sub-chains of LB must start and end with a match.

Let $M\text{-}Chain_{(i',j')}^{(i,j)}$ be one of these M matches sub-chains of LB. This sub-chain is, in fact, a chain because it starts and ends at a match. Let the length of $M\text{-}Chain_{(i',j')}^{(i,j)}$ be ℓ $(\ell = i'-i+j'-j)$. The normalized value of $M\text{-}Chain_{(i',j')}^{(i,j)}$, which is equal to the normalized value of LB, is $\frac{M}{\ell}$. The sub-chain next to $M\text{-}Chain_{(i',j')}^{(i,j)}$ must also start at a match. Thus, $(i'+1, j'+1)$, which is the position of the head of the next sub-chain, must be a match, and the length of $(M+1)\text{-}Chain_{(i'+1,j'+1)}^{(i,j)}$, which is comprised of the matches of $M\text{-}Chain_{(i',j')}^{(i,j)}$ and the match $(i'+1, j'+1)$, is $\ell+2$. Since

$\frac{M}{\ell} < \frac{1}{2} \rightarrow \frac{M}{\ell} < \frac{M+1}{\ell+2}$, the normalized value of $(M+1)$-$Chain^{(i,j)}_{(i'+1,j'+1)}$ is higher than that of M-$Chain^{(i,j)}_{(i',j')}$ alone, and thus, it is also higher than that of LB. Hence, if LB has more than $2M-1$ matches, and if its normalized value is lower than $\frac{1}{2}$, LB must have a sub-chain of at least M matches whose normalized value is higher than the normalized value of LB. Therefore, such LB cannot be $Best(M)^X_Y$. ■

This concludes our claim that the $O(rM \log\log n)$ algorithm may be used for the construction of the longest $Best(M)^X_Y$.

The $O(rM \log\log n)$ algorithm: The algorithm is identical to the $O(rL \log\log n)$ algorithm from the previous section in all aspects except one; it constructs k-*Chains* for $1 \leq k \leq 2M-1$. Thus, only $2M-1$ LRO^ks are maintained and updated, and the record of each match in the data structure of matches has at most $2M-1$ elements listed.

Complexity analysis: In order to construct chains of at most $2M-1$ matches, each match has to issue queries at $2M-1$ LRO^ks. Each match is inserted into and extracted from each LRO^k at most once. Thus, the total time complexity of the algorithm is $O(n \log|\Sigma| + rM \log\log n)$. The space complexity is $O(rM + nM)$. $O(rM)$ is also the time complexity of retrieving $Best(M)^X_Y$.

4 Conclusions and open problems

The normalized sequence alignment approach enables us to localize the LCS algorithm, which is global by its nature. This technique enabled us not only to design an algorithm that is both local and sparse, but also to eliminate the mosaic and the shadow effects from which non normalized local similarity algorithms suffer. In addition, the issue of minimal length constraint on the length of the output alignments, which is trivial in the non normalized algorithms, but tends to be problematic for normalized algorithms, was handled simply and without the reformulation of the original normalized alignment problem.

As proved in section 3, the $O(rM \log\log n)$ algorithm is capable of computing the normalized value of $Best(M)^X_Y$ and constructing the longest $Best(M)^X_Y$. Still, for many practical applications, such as local text similarity, the $O(rL \log\log n)$ algorithm that can compute all the substring pairs whose similarities are higher than a predefined value and whose length has no upper bound (except by the length of the input strings) may be the preferred algorithm. Nonetheless, it may be useful to use the $O(rM \log\log n)$ algorithm first to screen out input strings that do not achieve the desired local similarity values.

The modification of the scoring scheme of these algorithms from the LCS metric to other unit cost scorings schemes such as the *edit distance* remains an open problem.

Acknowledgment

The authors would like to thank Kunsoo Park for introducing the problem to us. We are also grateful to Alberto Apostolico, Klara Kedem, Yuri Rabinovich, Micha Sharir, Alek Vainshtein and Michal Ziv-Ukelson for fruitful discussions.

References

[1] Alexandrov, N.N., V.V. Solovyev. Statistical significance of ungapped alignments . in: *Pacific Symposium on Bioinformatics*, 463-472, R. Altman, A. Dunker, L. Hunter, T. Klein, editors, (1998).

[2] Altschul, S.F., B.W. Ericson. Locally optimal subalignments using nonlinear similarity functions. *Bull. Math. Biol.*, 48, 633-660, (1986).

[3] Altschul, S.F., B.W. Ericson. Significance levels for biological sequence comparison using nonlinear similarity functions. *Bull. Math. Biol.*, 50, 77-92, (1988).

[4] Apostolico, A. String editing and longest common subsequence. in: *Handbook of Formal Languages*, Vol. 2, 361-398, G. Rozenberg and A. Salomaa, editors, Springer Verlag, Berlin, (1997).

[5] Apostolico, A., Z. Galil. Pattern matching algorithms. Oxford University Press, 1997.

[6] Apostolico, A., C. Guerra. The Longest Common Subsequence Problem Revisited. *Algorithmica*, 2, 315-336, (1987).

[7] Arslan, A.N., Ö. Eğecioğlu, P.A. Pevzner. A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics*, 17(4), 327-337, (2001).

[8] Claus R. Efficient Computation of All Longest Common Subsequences. *SWAT 2000*, 407-418, (2000).

[9] Crochemore M., W. Rytter. Text Algorithms. Oxford University Press, 1994.

[10] Crochemore M., W. Rytter. Jewels of Stringology. World Scientific, 2002.

[11] Eppstein, D., Z. Galil, R. Giancarlo, G.F. Italiano. Sparse Dynamic Programming I: Linear Cost Functions. *JACM*, 39, 546-567, (1992).

[12] Gusfield, D., Algorithms on strings, trees, and sequences. Cambridge University Press (1997).

[13] Hirschberg, D.S. Algorithms for the longest common subsequence problem *JACM*, 24(4), 664-675 (1977).

[14] Hunt, J.W., T.G. Szymanski. A fast algorithm for computing longest common subsequence. *Communications of the ACM*, 20, 350-353 (1977).

[15] Johnson, D.B. A priority queue in which initialization and queue operations take O(loglog D) time. *Math. Syst. Theory*, 15, 295-309 (1982).

[16] Levenshtein, V.I., Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl*, 10, 707-710 (1966)

[17] Mäkinen, V., Parameterized approximate string matching and local similarity based point pattern matching. University of Helsinki, Finland, Report A-2003-6, 2003.

[18] Myers, E.W. Incremental Alignment Algorithms and their Applications. Tech. Rep. 86-22, Dept. of Computer Science, U. of Arizona (1986).

[19] Navarro G., M. Raffinot. Flexible pattern matching in strings practical on-line search algorithms for text and biological sequences. Cambridge University Press, 2002.

[20] Sankoff D., J.B. Kruskal, editors. Time warps, string edits, and macromolecules: The theory and practice of sequence comparison. Addison-Wesly Publishing Company, 1983.

[21] Smith, T.f., M.S. Waterman. The identification of common molecular subsequences. *J. Mol. Biol.*, 147, 195-197 (1981).

[22] Ukkonen E., On-line construction of suffix trees. Technical Report No A-1993- 1, Department of Computer Science, University of Helsinki, 1993

[23] Zhang, z., P. Berman, T. Wiehe, W. Miller. Post-processing long pairwise alignments *Bioinformatics*, 16, 1012-1019, (1999).