Small Longest Tandem Scattered Subsequences

Luís M. S. Russo[a,*], Alexandre P. Francisco[a]

[a]*INESC-ID and the Department of Computer Science and Engineering, Instituto Superior Técnico, Universidade de Lisboa.*

Abstract

We consider the problem of identifying tandem scattered subsequences within a string. Our algorithm identifies a longest subsequence which occurs twice without overlap in a string. This algorithm is based on the Hunt-Szymanski algorithm, therefore its performance improves if the string is not self similar. This occurs naturally on strings over large alphabets. Our algorithm relies on new results for data structures that support dynamic longest increasing sub-sequences. In the process we also obtain improved algorithms for the decremental string comparison problem.

Keywords: dynamic programming, Hunt-Szymanski algorithm, longest increasing sub-sequence
2000 MSC: 68W32, 68W40, 68Q25

1. Introduction

In this paper we study longest common scattered sub-sequences (LCSS). Given two strings P and S the LCSS is used extensively as a measure of similarity. In particular we consider a variant of this problem, where the LCSS must occur twice without overlap in an initial string F. We study algorithms and data structures that are relevant for this goal. Namely we use the Hunt-Szymanski algorithm (1977) and present new results for data structures that maintain information about the longest increasing sub-sequence of a dynamic sequence of numbers and new algorithms for the decremental string comparison problem. Specifically we obtain the following results:

---

*Corresponding author.
   Email addresses:* `luis.russo@tecnico.ulisboa.pt` (Luís M. S. Russo), `aplf@tecnico.ulisboa.pt` (Alexandre P. Francisco)

1. A data structure to maintain the longest increasing subsequence (LIS) of a dynamic list of numbers. This structure can be used to efficiently: **Append** a number at the end of the list; remove the current minimum value from the sequence (**ExtractMin**); obtain a current longest increasing subsequence (**GetLIS**). When the list contains $\ell \geq 2$ elements, the operation **Append** requires $O(\log \ell)$ time[1]. If the size of the LIS is $\lambda \geq 2$ then the **ExtractMin** operation requires $O(\lambda \log \ell)$ time. See Theorem 1. We further improve these bounds by analyzing batches of operations and assuming the final sequence is empty. In this case **ExtractMin** requires $O(\lambda(1 + \log(\min\{\lambda, \ell/\lambda\})))$ amortized time per operation and **Append** requires $O(1)$ amortized time, when the numbers are inserted in decreasing sequences of size at least λ elements. See Theorem 2. This structure uses optimal $O(\ell)$ space.

2. Using the Hunt and Szymanski (1977) reduction from LCSS to LIS we obtain new bounds for the decremental string comparison problem. In particular, for a given string F of size $n > 1$, we show that it is possible to obtain all the LCSS values for all the pairs of strings P and S such that $F = P.S$ in $O(\min\{n, \ell\}\lambda(1 + \log(\min\{\lambda, \ell/\lambda\})) + n\lambda + \ell)$ time, where $\lambda \geq 2$ is the size of the LCSS and $\ell \geq 2$ is the number of pairs of positions in F that contain the same letter. Therefore it is possible to determine the LTSS within this time, i.e., the LCSS which occurs twice without overlap in a string F.

2. The problem

Let us start by describing the longest tandem scattered sub-sequence (LTSS) problem of a given string F. We will use a running example with $F = \texttt{AGCGAACGGGTA}$. The meaning of tandem is that the sub-sequence needs to occur twice without overlap in F. Therefore F can be partitioned into a prefix P and a suffix S, i.e., $F = P.S$, such that the desired scattered sub-sequence is a longest common scattered sub-sequence (LCSS) between P and S. To determine which partition yields the overall largest sub-sequence it is necessary to test all such partitions.

---

[1]Note that to simplify expressions as $O(1 + \log \ell)$ we impose restrictions on parameters such as $\ell \geq 2$. This also avoids invalid expressions such as when $\ell = 0$. In general the complexity of the excluded cases is $O(1)$.

Let us consider the partition with $P = $ AGCG and $S = $ AACGGGTA. The LCSS is the longest string that occurs as a scattered sub-sequence of P and S. Figure 1 illustrates that the string ACG is a longest common scatted sub-sequence of P and S. A common sub-sequence can be defined as a set of pairs (i, j) where i is an index over P and j an index over S and the i-th letter of P is equal to the j-th letter of S, represented as $P(i) = S(j)$. All numbers i must be distinct among themselves and all numbers j must be distinct among themselves. Moreover sorting the pairs by i must also yield a sorted sequence by j. In our example the LCSS for P and S can be represented by the set $\{(1, 1), (3, 3), (4, 4)\}$.

Figure 1 also shows an LCSS for a second prefix suffix decomposition of $F = P'.S'$. This second decomposition is related to the first as $P' = P.$A and in fact the LCSS is similar to the previous LCSS, with the extra character A, i.e., ACGA.

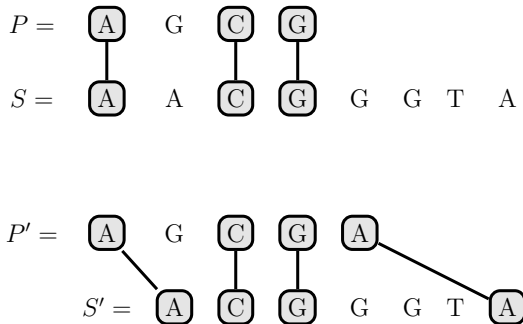In this example the LCSS between P' and S' is the desired overall LTSS.

Figure 1: Example of LCSS for two prefix suffix decompositions of F, $F = P.S$ and $F = P'.S'$

3. Reduction to decremental string comparison

In this section we present the main ideas of an algorithm that computes LTSS. Given a string F we can reduce this problem to computing the size of the LCSSs for all prefix and suffix decompositions of F, i.e., for all $P.S = F$, where P is a prefix and S is a suffix. The resulting tandem can be obtained from the overall largest LCSS.

This process involves n LCSS computations, when the size of F is n. Each LCSS can be determined with the classical dynamic programming table between P and S. Table D is a bi-dimensional array that stores integers. Each value $D[i, j]$ represents the size of the LCSS between the prefix of P with i letters and the prefix of S with j letters. The coordinate i ranges from 0 to the size of P, likewise coordinate j ranges between 0 and the size of S. The value 0 represents the empty prefix.

The values $D[i, j]$ can be computed locally according to the equalities bellow, where $P(i)$ denotes the i-th letter of P and $S(j)$ the j-th letter of S:

遞迴關係式

$$D[i, j] = 0 \qquad\qquad\qquad \text{if } i = 0 \text{ or } j = 0 \qquad (1)$$
$$D[i, j] = D[i - 1, j - 1] + 1 \qquad \text{if } i, j > 0 \text{ and } P(i) = S(j) \quad (2)$$
$$D[i, j] = \max\{D[i, j - 1], D[i - 1, j]\} \quad \text{if } i, j > 0 \text{ and } P(i) \neq S(j) \quad (3)$$

Let us consider a running example with $P = $ AGCG and $S = $ AACGGGTA. The values of table $D[i, j]$ are shown in the top portion of Figure 2. For example the value $D[4, 3]$ is 2, which means that the LCSS between AGCG and AAC has size 2. The D values can be computed with some local relations, which we review in Section 4. Hence this table requires $O(n^2)$ time to build, when P and S have $O(n)$ size.

The LCSSs can be recovered with tracebacks. A traceback is a pointer from a cell $D[i, j]$ to one of its neighboring cells $D[i - 1, j]$, $D[i, j - 1]$ or $D[i - 1, j - 1]$. The resulting paths represent the corresponding LCSSs. The diagonal tracebacks represent matches between the corresponding strings, we show only these tracebacks in Figure 2. In our example there is a diagonal traceback from $D[4, 4]$, representing the fact that both strings end with the letter G. Let us then consider $S' = $ ACGGGTA and $P' = P.$A. We also need to compute the D table for these strings, shown at the in the middle of Figure 2. To avoid confusion we refer to this table as D'.

目標是把O(n^2)降到O(n)   We aim to compute a representation of table D' in $O(n)$ time, instead of $O(n^2)$. First let us highlight the changes between D and D'. Column $D[i, 1]$ is removed, which corresponds to removing A from S. Row $D'[5, j]$ is inserted, which corresponds to appending A to P. Several values are maintained, $D[i, j + 1] = D'[i, j]$. The remaining values decrease by 1, i.e., $D'[i, j] = D[i, j + 1] - 1$. In our example only the values of column $D'[i, 0]$ decrease, the remaining values are maintained. To determine which cells change and which remain constant we consider another representation of table D. The representation used in the Hunt-Szymanski algorithm Hunt and Szymanski

使用D-table難以觀察出有規律變化，所以用另一個結構

$$\begin{array}{c|ccccccccc}
\xrightarrow{j} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
i\downarrow P & & \text{A} & \text{A} & \text{C} & \text{G} & \text{G} & \text{G} & \text{T} & \text{A} = S \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1\ \text{A} & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2\ \text{G} & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 \\
3\ \text{C} & 0 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\
4\ \text{G} & 0 & 1 & 1 & 2 & 3 & 3 & 3 & 3 & 3 \\
\end{array}$$

$$\begin{array}{c|cccccccc}
i\downarrow P' & & \text{A} & \text{C} & \text{G} & \text{G} & \text{G} & \text{T} & \text{A} = S' \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1\ \text{A} & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2\ \text{G} & 0 & 1 & 1 & 2 & 2 & 2 & 2 & 2 \\
3\ \text{C} & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\
4\ \text{G} & 0 & 1 & 2 & 3 & 3 & 3 & 3 & 3 \\
5\ \text{A} & 0 & 1 & 2 & 3 & 3 & 3 & 3 & 4 \\
\end{array}$$

$$\begin{array}{c|ccccccc}
i\downarrow P'' & & \text{C} & \text{G} & \text{G} & \text{G} & \text{T} & \text{A} = S'' \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1\ \text{A} & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
2\ \text{G} & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
3\ \text{C} & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
4\ \text{G} & 0 & 1 & 2 & 2 & 2 & 2 & 2 \\
5\ \text{A} & 0 & 1 & 2 & 2 & 2 & 2 & 3 \\
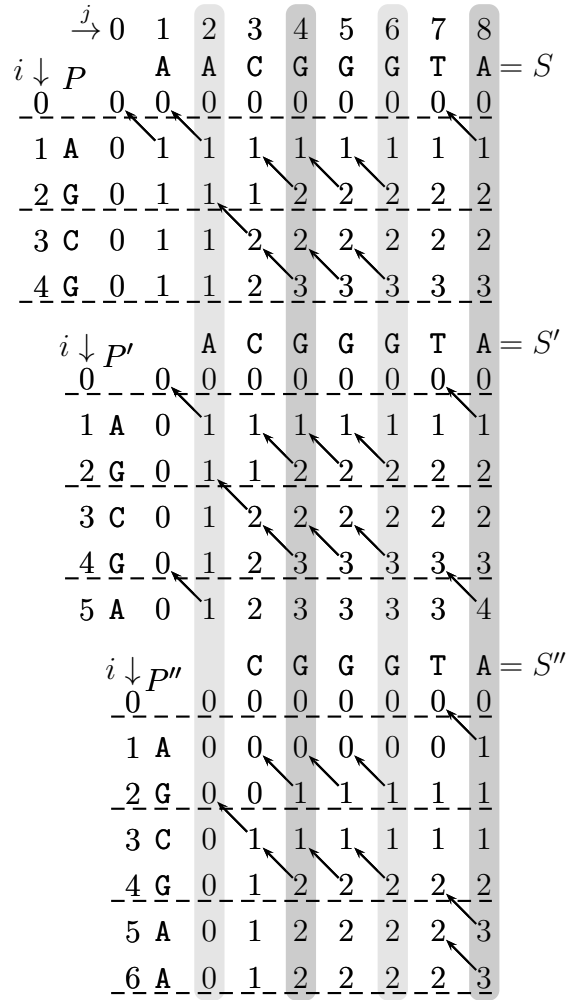6\ \text{A} & 0 & 1 & 2 & 2 & 2 & 2 & 3 \\
\end{array}$$

Figure 2: Illustration of tables D, D' and D'' with diagonal tracebacks.

(1977).

4. Dynamic Hunt-Szymanski Algorithm

In this section we show to efficiently compute decremental string comparison, i.e., a simple and efficient way to obtain table D' from table D. Let us start by reviewing and augmenting the Hunt-Szymanski algorithm (Hunt and Szymanski, 1977). The algorithm works by reducing the LCSS to the problem of determining a longest increasing sub-sequence of numbers (LIS). This reduction is illustrated in Figure 3. It works in two steps. In the first step it processes S.
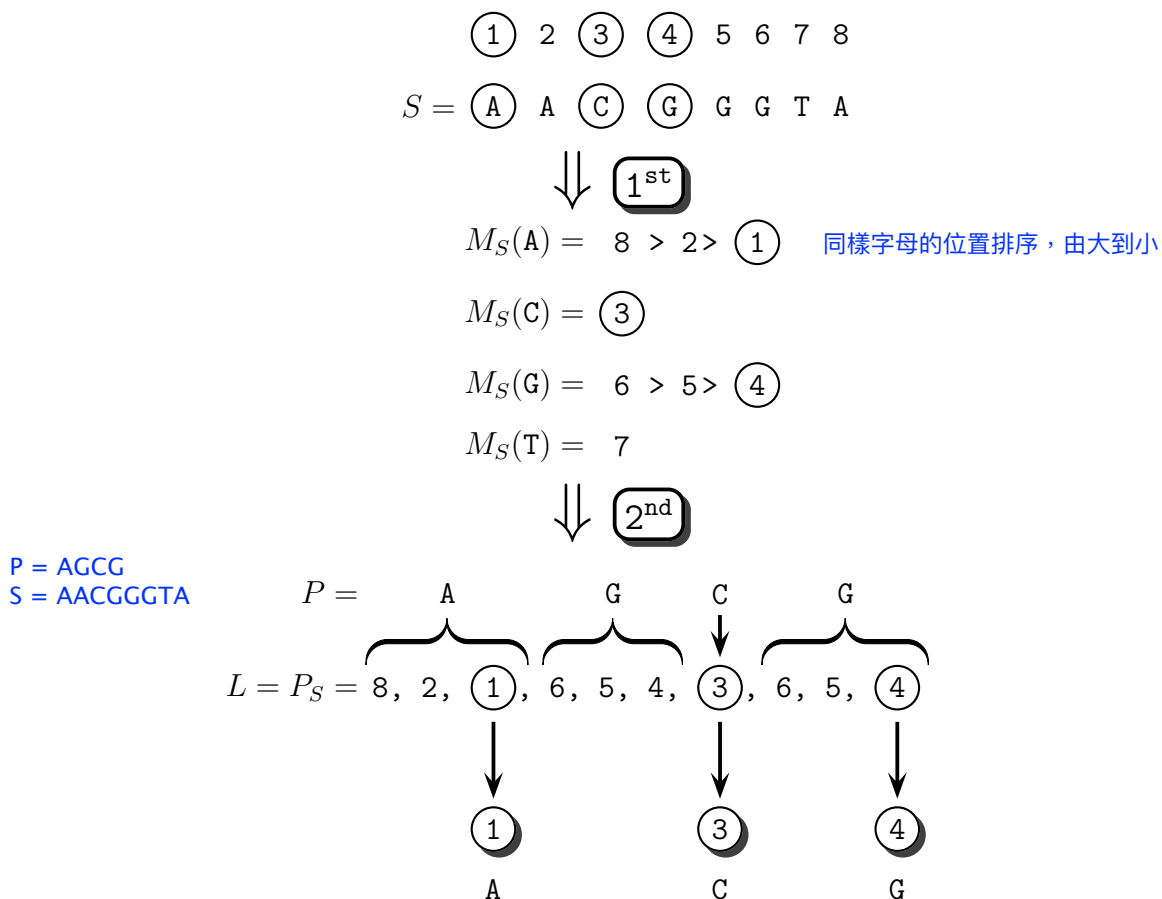
5

$$\begin{array}{cccccccc}①&2&③&④&5&6&7&8\end{array}$$

$$S = \begin{array}{cccccccc}Ⓐ&\texttt{A}&Ⓒ&Ⓖ&\texttt{G}&\texttt{G}&\texttt{T}&\texttt{A}\end{array}$$

$\Downarrow$ [1ˢᵗ] → $\Downarrow$ **1st**

$M_S(\texttt{A}) = \quad 8 > 2 > ①$ 同樣字母的位置排序，由大到小

$M_S(\texttt{C}) = ③$

$M_S(\texttt{G}) = \quad 6 > 5 > ④$

$M_S(\texttt{T}) = \quad 7$

$\Downarrow$ **2nd**

P = AGCG
S = AACGGGTA

$$P = \quad \texttt{A} \qquad \texttt{G} \qquad \texttt{C} \qquad \texttt{G}$$

$$L = P_S = \; 8,\; 2,\; ①,\; 6,\; 5,\; 4,\; ③,\; 6,\; 5,\; ④$$

$$① \qquad ③ \qquad ④$$

$$\texttt{A} \qquad \texttt{C} \qquad \texttt{G}$$

Figure 3: Reduction from LCSS to LIS.

For every letter b in S it computes the list of positions where b occurs in S, represented by $M_S(b)$. In the second step it processes P, from left to right, and produces a list of numbers P_S. For every letter b of P the list $M_S(b)$ is appended to the current list of numbers.

The resulting list P_S consists of a list of positions of S, where the same position may appear several times. Hence selecting a subsequence from P_S is equivalent to choosing letters from S. In our example a resulting longest increasing subsequence is $1 < 3 < 4$. Selecting these letters from S yields the desired common subsequence ACG. To avoid selecting the same letter from S repeatedly the LIS needs to be strictly increasing. Moreover, to guarantee that a letter from P is selected only once the lists $M_S(b)$ are

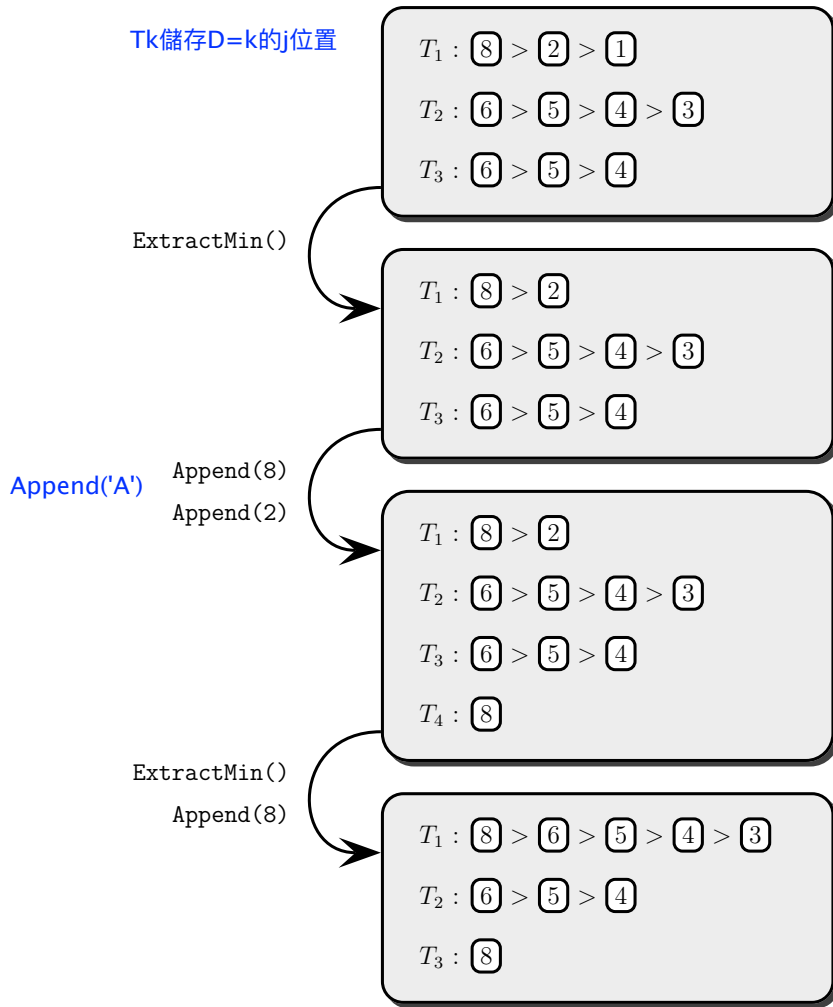Figure 4: Dynamic LIS computation. The top box shows the T_k lists for P and S.

sorted in decreasing order and this order is used to build P_S.

The Hunt-Szymanski algorithm then proceeds to efficiently compute the LIS. In this context we represent the list of numbers by L, abstracting away the process that was used to produce it, i.e. $L = P_S$. To determine the LIS the algorithm uses a sequence of threshold lists T_k. List T_k contains the value i of L if the longest increasing subsequence of the first elements of L up to and including i, has size k. The top box of Figure 4 shows this threshold structure for the sequence L we are considering.

Now let us return to the decremental string problem and study how this data structure is affected when P changes to P' and S changes to S'. The

7

top part of Figure 2 shows the dynamic programming table D, for P and S. The figure also illustrates D' and D'' for the consecutive decompositions that append letters to P and remove letters from the beginning of S. This figure serves to illustrate the relation between the D table and the T_k lists. Figure 2 shows only diagonal tracebacks, as these are the only ones that appear in the T_k lists. For example consider the cells in D that are equal to 2, the list T_2 gives a representation of this set. The cells $(2, 6), (2, 5), (2, 4)$ and $(3, 3)$ are the respective diagonal tracebacks. The T_k lists store only the j coordinates, therefore the list T_2 contains $6 < 5 < 4 < 3$. In general list T_k stores the decreasing j coordinates of the cells with D value k and diagonal tracebacks.

Computing the threshold structure is done incrementally by processing the elements of L from left to right. Therefore the original Hunt-Szymanski algorithm already supports the **Append** operation, which updates the structure when a new number is appended. Each list T_k is stored in decreasing order. With this organization the sequence of tail elements is kept in increasing order throughout the execution of the algorithm. After processing our sample list L the resulting sequence of tail elements is $1 < 3 < 4$. This order is used to determine in which T_k a given element of L should be inserted. Let us consider our running example and start with all the T_k lists empty. The first 8 initializes T_1. The 2 is also append to T_1 because $8 > 2$, likewise 1 is also appended to T_1 because $2 > 1$. Number 6 initializes list T_2, because $1 < 6$, which becomes the current sequence of tail elements. Numbers $5, 4$ and 3 are also appended to T_2, as they are all greater than 1 and in decreasing order. The number 6 initializes list T_3, because $1 < 3 < 6$. Likewise 5 and 4 are also appended to T_3. Since list T_3 is not empty, we know that L contains an increasing subsequence of length 3. Since T_4 was left empty, there is no increasing subsequence of length 4. Therefore the size of a LIS in our example is 3.

We now focus on adapting the threshold structure to obtain the longest subsequence which occurs twice without overlap, in a string F. Consider $F = $ AGCGAACGGGTA, in which case the subsequence could be ACGA, which has size 4. To compute it we can divide $F = P.S$ into a prefix P and a suffix S and compute the corresponding LCSS. Notice that this process is guaranteed to obtain a non-overlapping subsequence, which occurs twice in F. However the resulting LCSS might not be the longest such subsequence. In the division we used before the resulting subsequence had only size 3. Again the straightforward approach is to compute the LCSS for all possible P and S pairs.

Our approach aims to avoid repeated computation by modifying the T_k lists, instead of recomputing them. We will append letters to P and delete them from to S. Assume that we have the T_k lists for the strings P and S. We aim to update this structure for strings $P' = P.\texttt{A}$ and $S' = \texttt{ACGGGTA}$, i.e., we want to append a letter to P and remove the first letter from S. Removing the first letter from S changes the $M_S(b)$ lists, in particular all the positions are offset by 1, for example $M_{S'}(\texttt{G}) = 5, 4, 3$. This offset does not alter the relative order of the numbers nor the shape of the threshold structure. Therefore we ignore this offset, to simplify the exposition. Instead assume that we start numbering the positions of S' at 2. Now the only change is to $M_{S'}(\texttt{A}) = 8, 2$, which looses position 1, as it is no longer part of S'. In general the position that gets removed is the overall minimum. Hence we need to apply an **ExtractMin** operation to the threshold structure. This operation should remove all the instances of the minimum in L, in this case all the instances of 1. In this particular example there is only one instance, but in general there can be several such occurrences. 刪除首位字母，相當於要刪掉所有最小值

因為是採用相對位置，所以所有位置會位移 1，但這不影響答案

Due to the decreasing order of the T_k lists and increasing order of the tail elements it is straightforward to locate the overall minimum element. The minimum is always the tail element of T_1. Notice that even if there are several instances of the minimum in L there is only one element in T_1, because of our approach of discarding duplicated elements. Now remove this element from T_1. The resulting structure still maintains the necessary orders as the sequence of the tail elements becomes $2 < 3 < 4$, and the internal order of the T_k's was not altered. This structure is shown in the second box of Figure 4. This is indeed the same structure that can be obtained from the sequence $8, 2, 6, 5, 4, 3, 6, 5, 4$. Thus, in this case, no further work is required.

Since P' contains an extra $\texttt{A}$. Hence we need to append the list $M_{S'}(\texttt{A})$ to L. Therefore we execute the operations **Append(8)** and **Append(2)**. This alters the T_k lists, as explained above. Appending the number 8, initializes T_4, because $2 < 3 < 4 < 8$. Appending the number 2 does not produce any change because it is already the tail of T_1. In this case we simply drop the element, Section 4.1 describes more a elaborated process that is used when the size of the LIS is not enough and we want to retrieve an actual such sequence. At this point we obtained a LIS of size 4 which identifies a LCSS of P' and S' that is our goal subsequence of F. However to make sure this is indeed the longest subsequence we must continue the process and update the T_k lists for $P'' = P'.\texttt{A}$ and $S'' = \texttt{CGGGTA}$. Again we begin by computing **ExtractMin**. Notice that the operation removes both instances

of the number 2. This time the operation is more elaborated because after removing the 2 from T_1 the resulting sequence of tail elements is no longer increasing. Note that 8 is the tail of T_1 and 3 is the tail of T_2 and $8 > 3$. To solve this problem we could transfer the 3 from T_2 to T_1, thus fixing the first inequality as $3 < 4$. However this is not correct. Note that at this point the sequence L we are considering is $8, 6, 5, 4, 3, 6, 5, 4$, in which case T_1 should be $8 > 6 > 5 > 4 > 3$. Therefore the correct procedure is to remove all the elements from T_2 and append them to T_1. Now T_2 becomes empty so all the elements from T_3 are moved to T_2, which leads T_3 to become empty and therefore all the elements from T_4 are moved to T_3. Hence T_4 becomes empty and the process terminates because T_4 was the last list.

The general procedure for `ExtractMin` is to remove the tail element from T_1 and then transfer from T_2 to T_1 all the elements that are larger than the current tail element. The process continues from T_{k+1} to T_k until there are no further elements to transfer, either because T_{k+1} is empty or all its elements are larger than the current tail element of T_k. In Section 4.1 we formalize, extend and analyze this data structure. Our example finishes by appending 8, which gets discarded and therefore does not alter the structure. For P'' and S'' the resulting LIS has size 3 and is therefore smaller than the subsequence `ACGA` obtained for P' and S'. This was in fact a desired subsequence, but the algorithm must scan the remaining pairs of prefixes and suffixes to certify this conclusion.

4.1. Implementation and Analysis

First let us discuss which data structures can be used to efficiently store the threshold data structure. In the classical Hunt-Szymanski algorithm each T_k list can be stored in a stack, where reading the `Top` element and pushing new elements can be achieved in constant time. There is no need to pop elements from the stacks, so it is enough to store the `Top` values. These values are stored in an array so that it is possible to perform a binary search on the top elements. The procedure to execute `Append(`i`)` is to execute a binary search on the array to find k such that `Top(`T_{k-1}`)` $< i \leq$ `Top(`T_k`)`. If T_k is empty assume its stack top is $+\infty$, also assume there is a sentinel list T_0 with `Top(`T_0`)` $= -\infty$. If for the resulting k we have `Top(`T_k`)` $= i$ then the procedure stops, otherwise it performs `Push(`T_k`, `i`)`.

To support the `ExtractMin` operation we prefer to use a different data structure. We represent the T_k lists using balanced binary search trees (BST), in particular red-black trees. This allows us to compute `Min(`T_k`)`,

`Insert`(T_k, i), `Remove`(T_k, i), `Predecessor`(T_k, i), `Split`(T_k, v) and `Concatenate`(T_k, $T_{k'}$) in $O(\log \ell)$ time, where ℓ is the size of L. Like the Hunt-Szymanski algorithm, we keep an array `Min`[k] that stores the tail element of T_k, so that it can be accessed in constant time. The `Min`(T_k) operation finds the smallest element in T_k. When the BST of T_k is empty it returns $+\infty$. The `Insert`(T_k, i) operation inserts the number i into the BST of T_k. The `Remove`(T_k, i) operation removes the number i from the BST of T_k, if key i does not exist then an error is reported and the current process is stopped. The `Predecessor`(T_k, i) operation finds the largest element of T_k that is less than or equal to i, i.e., $\max\{j \in T_k | j \leq i\}$, the result should be a pointer to the corresponding tree node v, if no such node exists the pointer should be `NULL`. The `Split`(T_k, v) operation divides the BST of T_k in two by keeping all the nodes with keys strictly larger than v in T_k and putting v and the remaining nodes in a new BST. The operation `Concatenate`(T_k, v) joins the BST containing node v into the BST of T_k, assuming that all the key values in T_k are larger than or equal to the key in v and v is the maximum key value in its BST. Recall that we assume that the values in T_k are not repeated, therefore the `Insert` and `Concatenate` operations drop duplicated elements when they occur. Algorithms 1 and 2 show the pseudo-code for the `Append` and `ExtractMin` operations, respectively. Note that for the `Append` procedure the `Min`[k] array plays the role of the `Top` operation in the classical version.

Let us now analyze the time performance of the `Append` procedure, Algorithm 1. Without the `Min`[k] array the overall time would be $O((\log \ell)(\log \lambda) + \log \ell)$, where the first term accounts for the binary search in lines 4 to 10. The second term accounts for the `Insert` operation in line 12. Using the `Min`[k] array this term reduces to $O(\log \lambda)$ and thus the overall time becomes $O(\log \ell)$ because $\lambda \leq \ell$, since λ is the size of a subsequence of L.

Now let us analyze the `ExtractMin` procedure, Algorithm 2. The **while** loop executes at most λ times. Each execution requires $O(\log \ell)$ time for the `Predecessor`, `Split` and `Concatenate` operations. Hence we obtain a bound of at most $O(\lambda \log \ell)$ time. However an even tighter bound is possible. This operation can be bounded by $O(\sum_{k=1}^{\lambda} \log(|T_k|))$, where $|T_k|$ is the size of the list T_k. Because the log function is concave and the size of all the lists adds up to ℓ we can use Jensen's inequality (1906) to obtain an $O(1 + \log(\ell/\lambda))$

Algorithm 1 Append(*i*)

Ensure: Updated threshold structure for L with i appended.

  1: $\ell \leftarrow (\ell + 1)$ ▷ Increase size of L.
  2: $j \leftarrow 0$
  3: $k \leftarrow (\lambda + 1)$
  4: **while** $j + 1 < k$ **do**
  5:     $m \leftarrow \lfloor (j + k)/2 \rfloor$
  6:     **if** $i > \texttt{Min}[m]$ **then**
  7:         $j \leftarrow m$
  8:     **else**
  9:         $k \leftarrow m$
10:     **end if**
11: **end while**
12: $\texttt{Insert}(T_k,\ i)$
13: $\texttt{Min}[k] \leftarrow i$
14: **if** $k = (\lambda + 1)$ **then**
15:     $\lambda \leftarrow (\lambda + 1)$ ▷ LIS grows
16: **end if**
---

bound. The following derivation justifies the bound.

$$\sum_{k=1}^{\lambda} \log(|T_k|) = \lambda \sum_{k=1}^{\lambda} \frac{\log(|T_k|)}{\lambda}$$

$$\leq \lambda \log \left(\sum_{k=1}^{\lambda} \frac{|T_k|}{\lambda} \right)$$

$$= \lambda \log(\ell/\lambda)$$

For our particular application of the LTSS we do not need to recover an actual sequence, at least not at the same time as identifying the $P.S$ partition of F. Still for a general dynamic LIS problem this may be useful. Hence we will now explain how to augment the data structure to support such a process.

Recall the L sequences that occur in our running example. In the top of Figure 5 we show these sequences, numbered L_1, L_2 and L_3, corresponding to the pairs of strings (P, S), (P', S') and (P'', S''). To retrieve the elements from the list L we need to index them. For L_1 this is straightforward to obtain,

12

Algorithm 2 ExtractMin()

Require: L is not empty
Ensure: Updated threshold structure for L without the current minimum.

 1: $\ell \leftarrow (\ell - 1)$ ▷ Decrease size of L.
 2: Remove(T_1, Min[1])
 3: $k \leftarrow 2$
 4: **if** $\ell > 0$ **then**
 5:     **while** Min(T_{k-1}) $\geq$ Min[k] **do**
 6:         $v \leftarrow$ Predecessor(T_k, Min(T_{k-1}))
 7:         Split(T_k, v)
 8:         Concatenate(T_{k-1}, v)
 9:         Min[$k - 1$] $\leftarrow$ Min[k]
10:         $k \leftarrow k + 1$
11:     **end while**
12: **end if**
13: Min[$k - 1$] $\leftarrow$ Min(T_{k-1})
14: **if** Min[λ] $= +\infty$ **then**
15:     $\lambda \leftarrow (\lambda - 1)$ ▷ LIS shrinks
16: **end if**

---

we simply number the elements from 1 to 10. However when L_1 changes to L_2 and the number 1 is removed, we do not re-index the sequence. A gap is left at position 3. Likewise when L_2 changes to L_3 a gap is left at position 2. Position 12 can be re-used because it was the last position of L_2.
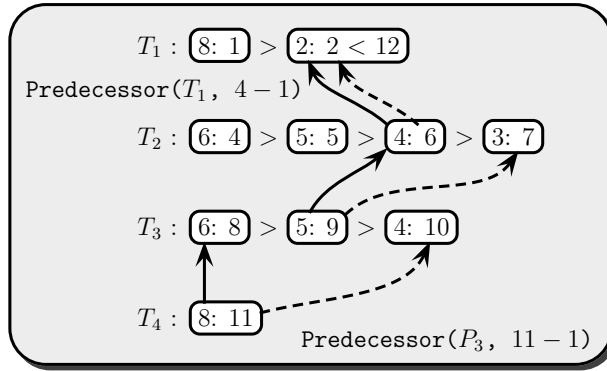
To retrieve the sequences we augment the elements inside each T_k. Each element stores a value of i of L and a list which contains positions where i occurs. These lists must contain at least one such position, but may contain more than one. The lists contain other positions precisely to avoid repeated elements in a T_k. To support LIS retrieval duplicated elements are not dropped, instead their positions are stored in lists. Figure 5 shows this structure for L_2 where the list T_1 contains the element 2 and the position list $2, 12$. Note that these position lists can be stored in increasing order, for each element. Moreover the concatenation of these lists, for a fixed T_k, is also sorted in increasing order. We refer to these global lists as P_k. Hence for L_2 we have $P_1 = 1, 2, 12$ and $P_2 = 4, 5, 6, 7$ and $P_3 = 8, 9, 10$ and $P_4 = 11$.

The possible sequences for our problem are shown in the bottom part of Figure 5. Each sequence is obtained by choosing one element from T_1, T_2,

```
                 1  2  3  4  5  6  7  8  9  10  11  12
L₁ = 8  2  1  6  5  4  3  6  5  4
L₂ = 8  2     6  5  4  3  6  5  4      8    2
L₃ = 8        6  5  4  3  6  5  4      8    8
```

$L_1 = 8\ 2\ 1\ 6\ 5\ 4\ 3\ 6\ 5\ 4$

$L_2 = 8\ 2\ \ 6\ 5\ 4\ 3\ 6\ 5\ 4\ \ 8\ \ 2$

$L_3 = 8\ \ 6\ 5\ 4\ 3\ 6\ 5\ 4\ \ 8\ \ 8$

L1->L2
移除最小值(1)
添加新增的match

每個元素儲存兩個
資料,分別為L的值
和index,index有
可能儲存多個以上

$T_1 :$ [8: 1] > [2: 2 < 12]
Predecessor(T_1, $4 - 1$)
$T_2 :$ [6: 4] > [5: 5] > [4: 6] > [3: 7]
$T_3 :$ [6: 8] > [5: 9] > [4: 10]
$T_4 :$ [8: 11]
Predecessor(P_3, $11 - 1$)

L2可能的LIS

T_1	T_2	T_3	T_4
2:2,	5:5,	6:8,	8:11
2:2,	4:6,	6:8,	8:11
2:2,	3:7,	6:8,	8:11
2:2,	4:6,	5:9,	8:11
2:2,	3:7,	5:9,	8:11
2:2,	3:7,	4:10,	8:11

Figure 5: Top: Example dynamic list L. Middle: Augmented threshold data structure for L_2. Bottom: Longest increasing sequences for L_2 in the order produced by Algorithm 3.

T_3 and T_4, in general one element from T_1 to T_λ. The sequences must be increasing in the values chosen from T_k and also in the values chosen from P_k. To guarantee that searching through these lists always yields a sequence of size λ the procedure starts with $k = \lambda$ and proceeds to decrease k. This is illustrated by the big arrow in Figure 5.

In our example we start at T_4 and choose the value 8 with corresponding position 11. Now we aim to determine which elements of T_3 may occur in LIS sequences that terminate at 8. To determine the first such element we can compute Predecessor(T_3, $8 - 1$), i.e., find the largest value in T_3 that is strictly smaller than 8. Likewise the last such element in P_3 should be

P是位置

`Predecessor(`P_3`, ` $11-1$`)`, i.e., it must occur in a position strictly smaller than 11. Recall that T_3 is stored in decreasing order and P_3 in increasing order. Therefore these predecessors define the interval of valid element choices for a LIS. In general the interval of interest for a given element (i, p) of T_{k+1} is between `Predecessor(`T_k`, ` $i - 1$`)` and `Predecessor(`P_k`, ` $p-1$`)`. The `Predecessor` on T_k is obtained from the BST for T_k in $O(\log \ell)$ time. The `Predecessor` on P_k is conceptual and it is enough to verify that $p' < p$, where p' is the current position in P_k. We illustrate these operations with arrows in Figure 5. The dashed lines are used for P_k and the filled lines for T_k. The figure also illustrates the interval for element 5 of T_3, i.e., `Predecessor(`T_2`,`$5-1$`)` and `Predecessor(`P_2`,`$9-1$`)`. Moreover it also shows the interval for element 4 of T_2, i.e., `Predecessor(`T_1`,`$4-1$`)` and `Predecessor(`P_1`,`$6-1$`)`. Therefore, iterating the `Predecessor(`T_k`, ` $i - 1$`)` operations, we can obtain the lexicographically largest LIS in $O(\lambda \log \ell)$. To obtain the remaining LIS we traverse these intervals, yielding a new LIS for each position that is visited. Algorithm 3 details this procedure.

The arguments for the RECURSIVEGETLIS are respectively, a stack S, which starts empty, a value for k, a value i of L in T_{k+1} and a corresponding position p in P_{k+1}. The `Predecessor` operation is extended to return the positions p, besides the i values. Since the operation is on T_k it returns the smallest p for the corresponding i, in our example `Predecessor(`T_1`, ` $4-1$`)` returns $(2, 2)$ instead of $(2, 12)$. If the corresponding T_k is empty then it returns $(-\infty, +\infty)$. Moreover for this algorithm we also use the `Next` operation, which behaves as an iterator and returns an (i', p') pair. It returns the next element, for example `Next(`P_1`)`, returns $(2, 12)$, assuming it is the first invocation after `Predecessor(`T_1`, ` $4-1$`)`. If there is no such element it returns $(+\infty, +\infty)$. Assume that T_k is represented as a BST and P_k is divided into lists, each inside a node of the BST as shown in the middle of Figure 5. The `Next` operation either moves to the next element in the current list, or to the next node on the BST, when it reaches the end of the current list. Note that by next on the BST we mean a smaller value of i, as the T_k are stored in decreasing order. Moving to the next element on a list requires constant time, but finding the next element on the BST may require $O(\log \ell)$ time. Hence Algorithm 3 obtains each LIS in $O(\lambda \log \ell)$ time, which again can be reduced to $O(1 + \lambda \log(\ell/\lambda))$ by Jensen's inequality. This finishes the dynamic LIS contribution.

Theorem 1. *It is possible to maintain a dynamic list with $\ell \geq 2$ num-*

Algorithm 3 GetLIS()

Require: L is not empty
 1: RECURSIVEGETLIS($\emptyset, \lambda, +\infty, +\infty$)

 2: **procedure** RECURSIVEGETLIS(S, k, i, p)
 3:    **if** $k = 0$ **then**
 4:       **return** S ▷ Found a LIS
 5:    **else**
 6:       $(i', p') \leftarrow$ Predecessor($T_k, i - 1$)
 7:       **while** $p' < p$ **do**
 8:          Push(S, p')
 9:          RECURSIVEGETLIS($S, k - 1, i', p'$)
10:          Pop(S)
11:          $(i', p') \leftarrow$ Next(P_k)
12:       **end while**
13:    **end if**
14: **end procedure**
---

bers such that the **Append** operation can be computed in $O(\log \ell)$ time and **ExtractMin** and **GetLIS** requires $O(1 + \lambda \log(\ell/\lambda))$ time, for a longest increasing sub-sequence, of size λ.

This result establishes some initial bounds of this data structure. However these bounds are fairly non competitive for our goals. To determine an LTSS we might generate a sequence with $\ell = O(n^2)$ elements and perform $O(\ell)$ **Append** operations and n **ExtractMin** operations. This yields an $O(n^2 \log n)$ time algorithm. Let us improve the performance of the dynamic LIS data structure. First we change the red-black BSTs to finger trees (D. Booth, 1996; HINZE and PATERSON, 2006; Guibas and Sedgewick, 1978). This means that Split and Concatenate operations that involve the $t_i \geq 2$ tail elements of T_k requires only $O(\log t_i)$ amortized time, instead of $O(\log |T_k|)$ time. Let us consider the overall algorithm, from the initial empty structure to the final one. We will analyze the overall time that is used to process a given list T_k. The following argument applies for any k but for simplicity consider that we are analyzing T_λ. We have the following inequality:

$$\sum_{i=1}^{n} t_i \leq n + (\lambda - 1)n \tag{4}$$

16

The left term in the inequality counts the number of elements that are moved from T_λ. The right side counts the number of elements that are removed from the data structure. The term n counts the number of elements that are actually removed from T_1, one for each `ExtractMin` operation. The term $(\lambda - 1)n$ accounts for the elements that are dropped in the middle of the data structure. In each `ExtractMin` operation at most $(\lambda - 1)$ elements are dropped, one for each T_k list, except for T_λ. Now the total time of these operations is $O(\sum_{i=1}^{n} \log t_i)$. We can obtain the such value, restricted to Equation (4), by using Lagrange multipliers. We consider only one Lagrange multiplier, represented by c, because we have only one restriction. Hence the resulting Lagrangian expression is the following:

$$\sum_{i=1}^{n} \log t_i - c(\sum_{i=1}^{n} t_i - \lambda n)$$

A derivative in order of t_i yields the following condition:

$$\frac{1}{t_i} = c \tag{5}$$

The derivative in order of c returns the original restriction:

$$\sum_{i=1}^{n} t_i = \lambda n \tag{6}$$

Combining both equations we obtain that $c = 1/\lambda$ and therefore $\log t_i = \log(\lambda)$. If we use the same upper bound for all the other T_k lists we obtain $O(n\lambda \log \lambda)$ total time for n `ExtractMin` operations. This yields an amortized time of $O(\lambda \log \lambda)$ per operation, provided the final structure is empty. This new bound for `ExtractMin` is not necessarily smaller than the previous $O(\lambda \log(\ell/\lambda))$, but the best of both applies.

Besides this bound for `ExtractMin` we also need a faster `Append` operation. Using the amortized performance of the finger tree data structure we obtain an $O(\log \lambda)$ amortized bound for the `Append` operation. This performance can be further improved by discarding the binary search process. Instead do a simple linear scan from T_λ down to the desired position. However we do not always reset the search, only if necessary. A sequence of decreasing numbers is therefore refereed to as a batch. During a batch the position k is not reset. This means that processing a batch containing m

Algorithm 4 AppendBatch(i)

Ensure: Updated threshold structure for L with i appended.
1: $\ell \leftarrow (\ell + 1)$ $\triangleright$ Increase size of L.
2: **if** Min[k] $< i$ **then** $\triangleright$ Value of k is maintained between calls.
3:      $k \leftarrow \lambda + 1$
4: **end if**
5: **while** $k > 1$ and Min[T_{k-1}] $> i$ **do**
6:      $k \leftarrow k - 1$
7: **end while**
8: Insert(T_k, i)
9: $\lambda \leftarrow \max(\lambda, k)$

---

numbers requires only $O(m + \lambda)$ time. Note that this is $O(1)$ amortized time per number, when the batch contains at least λ numbers.

A detailed description of the AppendBatch procedure is shown in Algorithm 4, where line 8 is computed in $O(1)$ amortized time with the finger tree data structure and line 5 accumulates to $O(\lambda)$ in a decreasing sequence. Note that the local variable k preserves its value among successive calls.

We can now summarize our dynamic LIS data structure in the following theorem:

Theorem 2. *It is possible to maintain information about the Longest Increasing Sub-Sequence of a dynamic list of numbers, which starts and finishes empty. Assuming that in total ℓ elements are inserted into the structure, in d batches of decreasing sequences and also that the* ExtractMin *operation is executed e times in total, then the overall time is bounded by $O(e\lambda(1 + \log(\min\{\lambda, \ell/\lambda\})) + \ell + d\lambda)$, where $\lambda \geq 1$ is the size of the largest overall LIS. At anytime the size of the current LIS can be obtained in $O(1)$ time.*

We can now combine the results of Theorem 1 and 2 to obtain our bounds for the decremental string comparison problem.

Theorem 3. *Given strings P and S there exists a data structure and can be used to obtain $\lambda \geq 2$, the size of LCSS between these strings. This structure requires $O(\ell)$ space, where ℓ is the number of matches between P and S. This structure can be updated for the strings $P.c$ and S in $O(\lambda + |S|)$ time, where c is any letter. It can also be updated for the strings P and S', where $S = c.S'$,*

in $O(\lambda(1 + \log(\ell/\lambda)))$ time. A sequence of operations that starts with an empty string and inserts letters to form the string P requires $O(|P|\lambda + \ell)$ time. A sequence of operations that decrements S until it becomes empty requires $O(\min\{|S|, \ell\}\lambda(1 + \log(\min\{\lambda, \ell/\lambda\})) + |S|)$ time.

The amortized complexities follow from Theorem 2, and the extra min that appears is a bound on the number of `ExtractMin` operations, e in Theorem 2. This number of operations is bounded simultaneously by $|S|$ and by ℓ, because we cannot remove more points than the ones that exist inside the structure. However in the case where $e < |S|$ it is necessary to add an $O(|S|)$ term. This corresponds to the case where the letter c that is being removed from S has no occurrences in P. In this case there is no call to `ExtractMin` operation but this verification still needs to be performed, which requires $O(1)$ time and must be accounted for.

Our application of computing the LTSS now follows from Theorem 3. The total amount of time the LTSS algorithm is therefore $O(\min\{n, \ell\}\lambda(1 + \log(\min\{\lambda, \ell/\lambda\})) + n + \ell)$, where $\lambda \geq 2$ is the size of the LTSS and $\ell \geq 2$ is the number of pairs of positions in F that contain the same letter.

5. Related work

An initial efficient algorithm to compute the LTSS, for the simple case of only one string F, was given by Kosowski (2004). This algorithm required optimal $O(n^2)$ time and $O(n)$ space. Tiskin (2008, Section 5.6) presented an algorithm which obtains the smallest worst case bound by exploring the Monge properties of the respective distance matrices. This property depends on the fact that the graph underlying the D table of two strings is planar. The resulting algorithm obtains the overall worst case time bound of $O(n^2(\log\log n)^2/(\log n)^2)$.

The work on incremental string comparison was initiated by Landau, Myers and Schmidt (1998), which obtained an $O(n)$ time algorithm to obtain D' from D. A simpler version, with the same performance was presented by Kim and Park (2000), which is simultaneously incremental and decremental. This is the first instance of the decremental variation of the problem. This solution was presented for the edit distance. Ishida, Inenaga, Shinohara and Takeda (2005) presented an algorithm which reduced the time complexity from $O(n)$ to $O(\lambda)$ and was fully incremental. The algorithm was presented for the LCSS and they also reduced the space requirements from $O(n^2)$ to $O(n\lambda)$.

Landau, Myers and Ziv-Ukelson (2004) studied the problem of consecutive suffix alignment problem, which obtained the size of the LCSS between all the suffixes of a string A and a string B, the final version of the paper appeared in (2007). The authors presented two algorithms for this problem, which required $O(n\lambda)$ and $O(n\lambda + n\log\sigma)$ time, where σ is the size of the alphabet of the underlying strings. Their approach uses a structure similar to the T_k lists from the Hunt-Szymanski algorithm, but contrary to our approach of Section 4 the elements are prepended to a variation of the T_k lists. Moreover their structure is not decremental. Because of these nuances the relation to LTSS is not immediate which justifies the algorithm of Kosowski (2004), in the same year.

A corner stone of all these results is the algorithm from Hunt and Szymanski (1977), whose crucial idea was the reduction from the

LCSS to the LIS, although this was not immediately clear in the original presentation. It was partially identified by Apostolico (1986), Apostolico and Guerra (1987) and made explicit by Jacobson and Vo (1992) and independently by Pevzner and Waterman (1992). Interestingly the original presentation of Hunt and Szymanski (1977) reported an $O((n+\ell)\log\ell)$ time bound, where ℓ is the size of the sequence L. This is a significant improvement over the plain dynamic programming algorithm, which always requires $O(n^2)$ time. Although in the worst case ℓ may be $O(n^2)$, in general it may be significantly smaller. The original complexity was not always faster than the plain algorithm, because ℓ may be $\Omega(n/\log n)$. This issue was addressed by Apostolico (1986) which obtained $O(n^2)$ time worst case guarantees. Their algorithm already considered using finger trees to represent the T_k lists. Improvements of the Hunt-Szymanski algorithm based on bitwise operations where proposed by Crochemore, Iliopoulous and Pinzon (2003).

A data structure that supports dynamic longest increasing sub-sequences was presented by Chen, Chu and Pinsker (2013). The focus is in supporting insertions anywhere in the sequence, which is achieved in $O(1 + \lambda\log(\ell/\lambda))$ time. The authors obtain one corresponding LIS in $O(\lambda + \log\ell)$ time. This is more efficient than the procedure we explain before Theorem 1, however our procedure can be used to obtain all the sub-sequences, whereas their approach obtains only one. Their data structure is similar to the one we present, which is expected as both are related to the Hunt-Szymanski algorithm. Chen et al. (2013) use level key lists L_k, which are similar to our T_k lists, but store index value pairs and are sorted by increasing index. This is similar to the structure we use for the `GetLIS` operation, but the lists are flattened,

instead of storing the indexes in a second structure. Moreover they also use red-black trees to split and concatenate lists and also mention exploring fingering properties of the structure. The presentation mentions deletions but the focus is on insertions. It seems plausible their representation could also support deletions efficiently.

The most recent approach for computing the LTSS was proposed by Inoue, Inenaga and Bannai (2020). Their algorithm is very similar to the one we present in this paper. They also reduce the problem to a dynamic LIS problem and used the data structure of Chen, Chu and Pinsker (2013) to obtain a complexity of $O(\min\{n, \ell\}\lambda(1 + \log(\ell/\lambda)) + n + \ell\log n)$. In the next section we explain how our algorithm improves upon their result and discuss future possible improvements.

6. Conclusions

In this section we recall and discuss the contributions of the paper in context. In this paper we presented a new algorithm to determine the longest tandem scattered sub-sequence of a string F. In the process we introduced the decremental string comparison problem and provided new data structures to support dynamic LIS sequences. We studied a dynamic version of the Hunt-Szymanski algorithm, which yielded several interesting results. Considering the LTSS problem itself the strongest work case bounds where obtained by Tiskin (2008) with an $O(n^2(\log\log n)^2/(\log n)^2)$ time bound. Both this algorithm and one by Kosowski (2004) seem to have the average case with the same bound as the worst case. The algorithm we obtain as a consequence of Theorem 2 obtains $O(\min\{n, \ell\}\lambda(1 + \log(\min\{\lambda, \ell/\lambda\})) + n + \ell)$ time, where $\lambda \geq 2$ is the size of the LTSS and $\ell \geq 2$ is the number of pairs of positions in F that contain the same letter. Hence when $\ell = o(n^2(\log\log n)^2/(\log n)^2)$ and $\lambda = o(n(\log\log n)^2/((\log(\min\{\lambda, \ell/\lambda\}))(\log n)^2))$ our algorithm becomes more efficient. Hence our algorithm is most efficient when the size of the LTSS is small, the extreme case in favor of our algorithm occurs when all the letters in F are distinct. In this case our algorithm is actually linear, i.e., $O(n)$ time and space. This particular case is trivial but a similar situation occurs when the alphabet size is large, i.e., polylog. This is also the case were the original Hunt-Szymanski algorithm obtains its best performance.

One important contribution of this work is the relation between the LTSS and the decremental and incremental string comparison algorithms. This

relation is straightforward but seems to have remained unnoticed[2], as the algorithm of Ishida et al. (2005) also depends on λ and could thus be used to compute the LTSS, if the structure was also decremental. On the other hand the structure of Kim and Park (2000) is decremental but does not depend on λ. This relation was indeed explored in the work of Tiskin (2008), but as mentioned above the resulting algorithm is also not dependent on λ. Hence for the case of a single string the algorithm we presented in Section 4 yields competitive results. In fact it is very interesting to compare the T_k lists of the Hunt-Szymanski algorithm to the incremental data structure of Ishida et al. (2005). In essence their structure consists in expanded T_k lists, where each element is repeated several times so that the list becomes size n. This increases the space requirements but makes navigating the lists and across lists more convenient. Also it forces one of the operations be $O(n)$ and thus the overall bound is always $O(n^2)$ instead of $O(n\lambda)$.

The recent work by Inoue, Inenaga and Bannai (2020) follows essentially the same approach as this paper. It solves the LTSS problem by resorting to the same decremental string comparison approach and solves this problem using the Hunt-Szymanski reduction to a LIS problem. A dynamic version of the LIS problem that supports the ExtractMin operation is also considered. In fact we were unaware of the similarity of their approach until recently. Still our approach contains several key insights which allow us to obtain a result that is competitive against their $O(\min\{n, \ell\}\lambda(1 + \log(\ell/\lambda)) + n + \ell \log n)$ time bound[3]. They use essentially the dynamic LIS structure of Chen, Chu and Pinsker (2013) and propose only one improvement, batched ExtractMin operations. This means that they obtain better performance for a sequence of ExtractMin operations. We obtain the same improvement by using our duplicate discarding approach. This means that a single ExtractMin operation on our data structure corresponds to several on theirs, because their ExtractMin operation removes one duplicate at a time whereas ours removes all. Hence our ExtractMin operation requires the same time as their batch of ExtractMin. One very important optimization of our approach are the finger trees to represent the T_k lists which lead to the improved performance of the AppendBatch operation, Algorithm 4. This implies

---

[2]It was also recently pointed out by Inoue, Inenaga and Bannai (2020).

[3]Note that we added an $O(n)$ term to their complexity result, because in the case that we considered when all the letters of F are distinct we have $\ell = \lambda = 0$, but their algorithm still requires $O(n)$ time, as does ours.

that there is no $O(\log n)$ factor associated to the ℓ term in our complexity. This term is most likely to dominate the overall complexity in several interesting cases and in our algorithm it is $O(\ell)$ whereas in theirs it is $O(\ell \log n)$. However this is a tradeoff as we obtain an extra $O(n\lambda)$ term, whereas theirs is only $O(n)$. Hence ignoring the first term the resulting comparison is between $O(n + \ell \log n)$ for their algorithm and $O(n\lambda + \ell)$ for ours. Hence our algorithm obtains better performance provided that $\ell = \Omega(n(\lambda-1)/((\log n)-1))$ and $n \leq \ell$ because of the first term. Let us consider a very simple example where this is likely to happen. Assume that the letters of F are obtained independently and uniformly at random from an alphabet of size σ. In this case ℓ is expected to be $(n/\sigma)^2$ and the distribution is highly concentrated around this value. Hence in order for our algorithm to obtain the best theoretical bound it is necessary to have an alphabet size σ that is smaller than $\sqrt{(n((\log n)-1))/(\lambda-1)}$. This is actually a very loose bound, much larger than poly logarithmic alphabets. Hence our algorithm's theoretical bound yields the best performance for most alphabets, except for exceedingly large ones. Even in extremely large alphabets there is the mitigating expectation that σ and λ have an inverse relation, meaning that larger values of σ should yield smaller values of λ.

The final improvement on the work of Inoue, Inenaga and Bannai (2020) is the analysis with Lagrange multipliers that yields the $\log(\min\{\lambda, \ell/\lambda\})$ bound that improves on the previous $\log(\ell/\lambda)$ complexity. Also we believe that future research on these data structures will focus precisely on this factor. One approach that seems promising is to store the T_k lists in a data structure that supports the dynamic fractional cascading technique of Chazelle and Guibas (1986), potentially reducing this factor to $O(\log \log n)$.

The final contribution of this paper is the data structure to maintain a dynamic LIS. Our approach uses a couple of nuances that allow us to obtain Theorem 2. Using the `AppendBatch` of Algorithm 4 the complexity of the original Hunt-Szymanski algorithm drops to $O(\ell + n\lambda)$, which is never more than $O(n^2)$ and sometimes much better. Given the importance of this algorithm, similar improvements have already been proposed by Apostolico (1986). Still our work provides a fairly simple alternative.

Acknowledgements

References

Apostolico, A., 1986. Improving the worst-case performance of the hunt-szymanski strategy for the longest common subsequence of two strings. Information Processing Letters 23, 63–69.

Apostolico, A., Guerra, C., 1987. The longest common subsequence problem revisited. Algorithmica 2, 315–336.

Chazelle, B., Guibas, L.J., 1986. Fractional cascading: I. a data structuring technique. Algorithmica 1, 133–162.

Chen, A., Chu, T., Pinsker, N., 2013. The dynamic longest increasing subsequence problem. arXiv preprint arXiv:1309.7724 .

Crochemore, M., Iliopoulous, C.S., Pinzon, Y.J., 2003. Speeding-up hirschberg and hunt-szymanski lcs algorithms. Fundamenta Informaticae 56, 89–103.

D. Booth, H., 1996. An overview over red-black and finger trees .

Guibas, L.J., Sedgewick, R., 1978. A dichromatic framework for balanced trees, in: Foundations of Computer Science, 1978., 19th Annual Symposium on, IEEE. pp. 8–21.

HINZE, R., PATERSON, R., 2006. Finger trees: a simple general-purpose data structure. Journal of Functional Programming 16, 197217.

Hunt, J.W., Szymanski, T.G., 1977. A fast algorithm for computing longest common subsequences. Communications of the ACM 20, 350–353.

Inoue, T., Inenaga, S., Bannai, H., 2020. Longest square subsequence problem revisited. arXiv:2006.00216.

Ishida, Y., Inenaga, S., Shinohara, A., Takeda, M., 2005. Fully incremental LCS computation, in: International Symposium on Fundamentals of Computation Theory, Springer. pp. 563–574.

Jacobson, G., Vo, K.P., 1992. Heaviest increasing/common subsequence problems, in: Annual Symposium on Combinatorial Pattern Matching, Springer. pp. 52–66.

Jensen, J.L.W.V., 1906. Sur les fonctions convexes et les ingalits entre les valeurs moyennes. Acta Math. 30, 175–193.

Kim, S.R., Park, K., 2000. A dynamic edit distance table, in: Annual Symposium on Combinatorial Pattern Matching, Springer. pp. 60–68.

Kosowski, A., 2004. An efficient algorithm for the longest tandem scattered subsequence problem, in: International Symposium on String Processing and Information Retrieval, Springer. pp. 93–100.

Landau, G.M., Myers, E., Ziv-Ukelson, M., 2004. Two algorithms for LCS consecutive suffix alignment, in: Annual Symposium on Combinatorial Pattern Matching, Springer. pp. 173–193.

Landau, G.M., Myers, E., Ziv-Ukelson, M., 2007. Two algorithms for LCS consecutive suffix alignment. Journal of Computer and System Sciences 73, 1095 – 1117. Bioinformatics III.

Landau, G.M., Myers, E.W., Schmidt, J.P., 1998. Incremental string comparison. SIAM Journal on Computing 27, 557–582.

Pevzner, P.A., Waterman, M.S., 1992. Matrix longest common subsequence problem, duality and hilbert bases, in: Annual Symposium on Combinatorial Pattern Matching, Springer. pp. 79–89.

Tiskin, A., 2008. Semi-local string comparison: Algorithmic techniques and applications. Mathematics in Computer Science 1, 571–603. doi:10.1007/s11786-007-0033-3.