Computing and Informatics, Vol. 31, 2012, 1217–1233

A COVER-MERGING-BASED ALGORITHM FOR THE LONGEST INCREASING SUBSEQUENCE IN A SLIDING WINDOW PROBLEM

Sebastian Deorowicz

Institute of Informatics
Silesian University of Technology
Akademicka 16
44-100 Gliwice, Poland
e-mail: `sebastian.deorowicz@polsl.pl`

Communicated by Vladimír Kvasnička

Abstract. A longest increasing subsequence problem (LIS) is a well-known combinatorial problem with applications mainly in bioinformatics, where it is used in various projects on DNA sequences. Recently, a number of generalisations of this problem were proposed. One of them is to find an LIS among all fixed-size windows of the input sequence (LISW). We propose an algorithm for the LISW problem based on cover representation of the sequence that outperforms the existing methods for some class of the input sequences.

Keywords: Longest increasing subsequence, sliding window, pattern matching

Mathematics Subject Classification 2010: 68W05

1 INTRODUCTION

The problem of finding a longest increasing subsequence (LIS) in a sequence of integers is to find a longest subsequence (subsequence is obtained from a sequence by removing zero or more symbols) that symbols are increasing.[1] It is a well-known combinatorial problem with applications in bioinformatics. It was used in MUMmer

[1] Note that there may be many subsequences satisfying the problem requirements, since only the LIS length is unique.

project to make an alignment of whole genomes [12, 13, 24]. In [18] an application in preparing gene maps were given. The LIS problem was used in Celera Genomics to discover new genes [32]. In [25] a discussion on applications of this problem in preparing probes for virus identification can be found. The problem was also used to discover relationships in databases [19]. An algorithm for the LIS problem can be also used to solve the longest common subsequence (LCS) problem [21] and to compute cliques in permutation graphs [20, p. 159].

There are a lot of research on LIS properties. In 1935, Erdös and Szekeres proved that in any integer sequence of unique symbols of length n there is a decreasing or increasing subsequence of length at least $n^{1/2}$ [17]. Further research culminated in 1999, when Baik et al. proved that the LIS expected length is $2n^{1/2} - \Theta(n^{1/6})$ and a standard deviation is $\Theta(n^{1/6})$ [4]. Intermediary results are summarised in a survey paper [2]. The fastest algorithms solving the LIS problem, for a comparison model of computation need $O(n \log \ell)$ time [27, 21], where ℓ is the result length, which is known to be a lower bound of time complexity in this model. $O(n \log\log n)$ time complexity is possible [22, 6] if the input sequence is a permutation of integers from range $[1, n]$ and a more permissive WordRAM model is allowed.

Recently, a number of variants of this problem, motivated by various reasons, were formulated. In a minimal/maximal weight/height/sum LIS problem [29, 10], some extremal LISs are looked for. In a slope-constrained LIS problem [30, 11], the symbols of the resulting subsequence must grow rapidly. Conversely, in a longest almost-increasing subsequence problem [16], some symbols in the output sequence may be smaller than the largest of previous symbols by some (usually small) constant. In a longest increasing cyclic subsequence problem (LICS), the result is a longest LIS among all cyclic shifts of the input sequence [1, 9, 11].

Finally, in a longest increasing subsequence in a sliding window (LISW) problem, the problem the paper is about, an LIS in any window of fixed size in an input sequence is considered [3]. This problem can be defined in four ways:

- find the length of a longest LIS among all windows of size u,
- find the above-specified subsequence,
- for each window of size u, find the length of an LIS in it,
- for each window of size u, find an LIS in it.

In this paper, we consider the first two variants. The LISW problem is a generalisation of the LICS problem, since to compute an LICS for a sequence A of length n it suffices to solve the LISW problem for sequence AA (i.e., a concatenation of two A sequences) and window size n.

For the LISW problem, Albert et al. proposed an algorithm based on Young tableau representation of the sequence [31] working in $O(n \log\log n + n\ell)$ time in the worst case [3], where ℓ is the result length. This time complexity was then improved to $O(n\ell)$ [7]. Recently, Tiskin presented an algorithm for some related problem [28]. When adapted to the LISW problem, it is of the worst-case time complexity $O(n \log^2 n)$.

We propose an algorithm based on cover representation of some windows of the input sequence. (A cover is a list of decreasing lists containing all symbols from a sequence, where each sequence symbol belongs to exactly of the decreasing list.) The covers are then merged to obtain covers for some windows, which we prove to be the only candidates to contain the result. In our algorithm, various data structures to represent the covers can be used. We examine three possibilities and show that the worst-case time complexity for our best variant is $O(n \log\log n + \min(n\ell, n\lceil \ell^3/u\rceil) \times \log\lceil u/\ell^2 + 1\rceil)$, which is better than Tiskin's result if $\ell = o\left(u^{1/3}\frac{\log^{2/3} n}{\log^{1/3} u}\right)$.

The idea of using a cover-based representation of a sequence to solve some LIS-related problems was formulated in our recent paper [9]. We considered in it the LICS problem in which a working cover all the time represents almost the whole input sequence (in various rotations). Here, for the LISW problem, it is necessary to represent as a working cover only a small part of the input sequence at a time and to insert symbols to this cover in a different way than for the LICS problem. The choice of the parameters of the data structures used in the examined cover representations is also different.

Algorithms solving the LISW problem could be applied to a system for preparation of probes for virus identification [25]. The most time-consuming part of this system is verifying how probe candidates (DNA sequences of length of tens nucleotides) are similar to the sequences from a database. In this verification, the probe candidates are obtained from some sequence by sliding a fixed-size window over it. Then, for each candidate probe and each sequence from a database an integer sequence is constructed and an LIS in this sequence is computed. Since the window is slided by a few nucleotides a time, the probe candidates overlap and a lot of work is repeated. This problem can be partially solved by computing an integer sequence for the 'probe' sequence and the sequence from the database and then using an LISW-computing algorithm. The LICS problem, a special case of the LISW problem, also appears in bioinformatics when working on circular genomes [8].

The paper is organised as follows. Section 2 contains the necessary definitions. In Section 3, an LIS-computation algorithm based on cover representation is presented. Section 4 contains description of the main concepts of the proposed algorithm for the LISW problem. In Section 5, implementation details and complexity analysis are presented. Last section concludes the paper.

2 DEFINITIONS

Let $A = a_1a_2 \ldots a_n$ be a sequence composed of unique symbols over an integer alphabet $\Sigma \subset \mathbb{Z}$. A sequence A' is a subsequence of A if it can be obtain from A by removing zero or more symbols from any positions, i.e., $A' = a_{i_1}a_{i_2} \ldots a_{i_k}$, where $1 \le i_1 < i_2 < \cdots < i_k \le n$. A continuous subsequence of A is a subsequence that can be obtained from A by removing zero or more symbols from the beginning and/or from the end, i.e., $A' = a_ia_{i+1} \ldots a_{i+k}$, where $1 \le i \le n - k$. The *longest increasing subsequence* (LIS) problem for A is to find a longest subsequence $A' = a_{i_1}a_{i_2} \ldots a_{i_\ell}$

such that $1 \leq i_1 < i_2 < \cdots < i_\ell \leq n$ and $a_{i_1} < a_{i_2} < \cdots < a_{i_\ell}$. The notation A_i^j means $a_i a_{i+1} \ldots a_j$ if $i \leq j$ and an empty sequence otherwise. A window of size u is a continuous subsequence composed of u symbols. For a given window size u and a sequence A, the *longest increasing subsequence in a sliding window* (LISW) problem is to find a longest LIS among all A_i^{i+u-1}, where $1 \leq i \leq n - u + 1$. The length of the result will be denoted by ℓ.

3 LONGEST INCREASING SUBSEQUENCE COMPUTATION

One of the popular ways of computing an LIS for a sequence is to construct for it a greedy cover [21] and read the result from it. (Since the algorithm proposed in this paper for the LISW problem relates on the cover representation of a sequence, we firstly discuss the cover-based solving of the LIS problem.) Let us now start from some definitions.

Definition 1. A *cover* of a sequence A is an ordered set of lists containing decreasing subsequences of A. Each symbol of A belongs to exactly one list of the cover.

Definition 2. A cover *size* is a number of lists the cover is composed of.

A cover of sequence A is denoted by $C(A)$. Cover lists are numbered from 1, and $C(A)[k]$ is k^{th} list of the cover for A.

Definition 3. A cover $C(A)$ is called *greedy* iff for each sequence symbol a_i, where $1 \leq i \leq n$ holds: a_i belongs to $C(A)[k_i]$, where k_i is the LIS length for $a_1 a_2 \ldots a_i$.

We deal only with greedy covers, so for brevity of presentation we always write ‘cover’ instead of ‘greedy cover’ in the rest of the paper. A cover is constructed by *Cover-Make* algorithm (Figure 1) [21]. The algorithm processes successive symbols and for each one finds in the current cover the leftmost list that can be extended by the current symbol. If there is no such list, the current symbol starts a new list in the cover. A proof is made as follows:

1. a greedy cover is obtained,
2. its size is the LIS length,
3. a greedy cover is unique,
4. a greedy cover is a cover of minimal size [21].

An example of the algorithm in work is given in Figure 2.

It is easy to obtain an LIS from a cover representation of a sequence. It suffices to start from any symbol of the last list of the cover and in a loop jump to the largest of smaller symbols than the current one in the left neighbour list. The visited symbols form an LIS read from the end. A proof that *LIS-Read* algorithm (Figure 3) returns LIS can be found e.g. in [21].

```
Cover-Make(A)
C ← empty list containing decreasing lists
for i ← 1 to n do
    k ← smallest list index of C, which tail symbol is larger than a_i
    if k exists then
        Append a_i to the end of list C[k]
    else
        Create new list containing a_i and append it to the end of C
return C
```

Fig. 1. Greedy cover making algorithm of the sequence

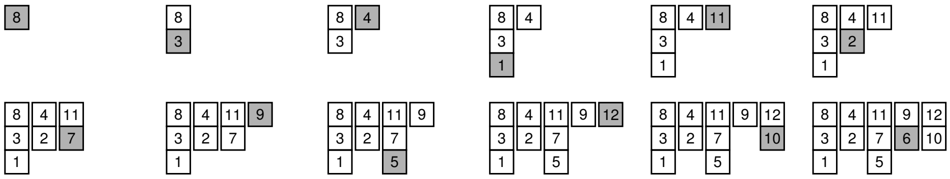


Fig. 2. Example of *Cover-Make* algorithm finding cover of $A = 8\,3\,4\,1\,11\,2\,7\,9\,5\,12\,10\,6$. The just placed elements are gray. The lists are vertical

Time complexity of *Cover-Make* algorithm depends on the way the list to extend is looked for. It is easy to see that the trailing symbols of the lists are ordered increasingly. Thus, they can be stored in an ordered array and a binary search over this array allows to find the list to extend in $O(\log \ell)$ time. The same complexity can be obtained if the trailing symbols of the lists are stored in a balanced binary search tree, e.g., red-black tree. In these cases, the time complexity of *Cover-Make* algorithm is $O(n \log \ell)$. If the input sequence is a permutation of unique integers from the range $[1, n]$, then van Emde Boas trees [14, 15] can be applied to determine the list to extend and the time complexity of the algorithm becomes $O(n \log \log n)$. In all cases, the space complexity is $\Theta(n)$. The time complexity of *LIS-Read* algorithm is $O(n)$.

```
LIS-Read(C)
  {C is a greedy cover of sequence A}
ℓ ← |C|
s_ℓ ← any symbol of C[ℓ]
for i ← |C| − 1 downto 1 do
    s_i ← largest symbol of C[i] smaller than s_{i+1}
return s_1 s_2 ... s_ℓ
```

Fig. 3. An algorithm reading LIS from the cover produced by *Cover-Make* algorithm

4 MAIN CONCEPTS OF THE PROPOSED ALGORITHMS

In [9], we proposed *Cover-Merge* algorithm (Figure 4)[2] as a fast way of obtaining a cover for a continuous subsequence if two covers for subsequences that concatenated give this continuous subsequence are known. This utility is useful in the algorithm proposed below. Let us now remember two definitions and a lemma from [9].

```
Cover-Merge($C'$, $C''$)
1  for $i \leftarrow 1$ to $|C''|$ do
2      if starting element of $C''[i]$ is larger than ending element of $C'[|C'|]$ then
3          Append to $C'$ an empty list
4      $j \leftarrow |C'|$
5      while $C''[i]$ is not empty and $j > 1$ do
6          Find largest symbol $s$ in $C''[i]$ smaller than symbol ending $C'[j-1]$
7          Move symbols larger than $s$ from $C''[i]$ to $C'[j]$
8          $j \leftarrow j - 1$
9      Append the remaining part of list $C''[i]$ (if it is not empty) to $C'[1]$
10 return $C'$
```

Fig. 4. A general scheme of the cover merging algorithm. (If there is no such a symbol s in line 6, all symbols are moved in line 7.) [9]

Definition 4. A *cover read of a sequence* for a sequence A, denoted by $R(A)$, is a concatenation of successive decreasing lists forming cover $C(A)$.

For example, in Figure 2, a cover for sequence $A = 8\,3\,4\,1\,11\,2\,7\,9\,5\,12\,10\,6$ is shown. The cover read for it is $R(A) = 8\,3\,1\,4\,2\,11\,7\,5\,9\,6\,12\,10$.

Definition 5. The symbols heading lists of cover $C(A)$ for sequence A are called *stop points for A*.

Lemma 1. (Lemma 1 in [9]) Let sequence A be concatenation of A' and A'', i.e., $A = A'A''$. The cover of $A'A''$ is identical to the covers of $R(A')R(A'')$ and of $A'R(A'')$.

Proof. See [9]. □

For easier understanding of the proposed algorithm for the LISW problem, it is convenient to assume that in the first stage covers are computed for the following continuous subsequences: A_1^u, A_{u+1}^{u+u}, ..., A_{iu+1}^{iu+u}, ..., $A_{\lceil n/u-1\rceil u+1}^{\lceil n/u-1\rceil u+u}$ (for simplicity, we

[2] There is a small mistake in the caption of the pseudocode in [9]. The claim 'If there is no such a symbol p in line 06, no symbols are moved in line 07.)' should be 'If there is no such a symbol p in line 06, all symbols are moved in line 07.)', which is corrected in this paper.

assume that n is an integer multiplicity of u, but a generalisation is easy). The LIS length for A_{iu+1}^{iu+u} will be denoted as ℓ_i. Now we need some lemmas.

Lemma 2. An LIS length in any window of size u in A_{iu+1}^{iu+2u} is not larger than $\ell_i + \ell_{i+1}$.

Proof. The proof is immediate. It suffices to note that the LIS length in A_{iu+1}^{iu+2u} must be not larger than $\ell_i + \ell_{i+1}$, so the LIS length in any continuous subsequence of that sequence cannot be larger. □

Lemma 3. Let us consider a continuous subsequence of double-window size, A_{iu+1}^{iu+2u}. For each window A_{iu+1+k}^{iu+u+k} of size u in it ($0 \leq k \leq u$), a symbol heading the last list of the cover for this window is either a stop point of the rightmost window of A_{iu+1}^{iu+2u}, i.e., $A_{(i+1)u+1}^{(i+1)u+u}$, or a symbol from the leftmost window, A_{iu+1}^{iu+u}.

Proof. From Lemma 1 we know that

$$C(A_{iu+1+k}^{iu+u+k}) = C(A_{iu+1+k}^{iu+u} R(A_{iu+u+1}^{iu+u+k})).$$

The *Cover-Make* algorithm constructing a cover for A_{iu+1+k}^{iu+u+k} at the beginning computes cover C for A_{iu+1+k}^{iu+u}. Then, it can extend C by successive symbols of lists of $C(A_{iu+u+1}^{iu+u+k})$. Each symbol heading any of these lists is a stop point for $A_{(i+1)u+1}^{(i+1)u+u}$ and may be appended to some list in C or may start a new list in C. Because the symbols in lists of the appended cover are decreasing, only a stop point may start a new list in C. From this, we know that the last list of the final cover is headed by a stop point of $A_{(i+1)u+1}^{(i+1)u+u}$ or a symbol from A_{iu+1+k}^{iu+u}. □

Lemma 4. The LISW length for sequence A_{iu+1}^{iu+2u} and window size u is the largest value of: the LIS length for every window A_{iu+1+k}^{iu+u+k}, where $1 \leq k \leq u$, ended at a stop point of $A_{(i+1)u+1}^{(i+1)u+u}$ or the LIS length for A_{iu+1}^{iu+u}.

Proof. From Lemma 3 we know that the head of the last list of a cover for any window of size u in A_{iu+1}^{iu+2u} is a stop point for $A_{(i+1)u+1}^{(i+1)u+u}$ or a symbol from the leftmost window, A_{iu+1}^{iu+u}. If it is a symbol of A_{iu+1}^{iu+u}, then an LIS in this window ending at that symbol consists only of symbols from A_{iu+1}^{iu+u} and a longest LIS for such symbols can be found in A_{iu+1}^{iu+u}.

If the symbol heading the last list of the cover for the considered window is a stop point for $A_{(i+1)u+1}^{(i+1)u+u}$, then it is the last symbol of an LIS in that window. For any such a stop point a_j, a longest increasing subsequence ending at a_j for all possible windows of size u can be found in A_{j-u+1}^{j}. □

According to the above lemmas we can construct *LISW-Range* algorithm (Figure 5) finding an LISW in a continuous subsequence A_{iu+1}^{iu+2u} for window size u. The algorithm starts from cover $C'' = C(A_{(i+1)u+1}^{(i+1)u+u})$ and removes from it the trailing symbols of $A_{(i+1)u+1}^{(i+1)u+u}$ until some stop point is found. Then, it constructs cover C'

and merges covers C' and C'' using *Cover-Merge* algorithm. In this way, we obtain a cover for the window ending at a stop point. In this cover, the LIS length is determined and it is verified whether we have a new best LISW length. Then, the procedure is repeated until next stop point is visited. Finally, it is verified whether the LIS length for the leftmost window of the current range is better or not. Relating on Lemma 4 we can conclude:

Corollary 1. Algorithm *LISW-Range* correctly computes the LISW length and the index of the last symbol of LISW in a continuous subsequence of double-window size.

An example of algorithm *LISW-Range* in work is shown in Figure 6.

```
LISW-Range($A$, $i$, $C^{\text{left}}$, $C^{\text{right}}$)
 1  $C'' \leftarrow C^{\text{right}}$; $j \leftarrow (i+1)u + u$
 2  if $a_j$ is a stop point in $C^{\text{right}}$ then $\ell \leftarrow |C''|$; $s \leftarrow j$ else $\ell \leftarrow 0$; $s \leftarrow 0$
 3  Remove $a_j$ from $C''$
 4  for $k \leftarrow (i+1)u + u - 1$ downto $(i+1)u + 1$ do
 5      if $a_k$ is a stop point in $C^{\text{right}}$ then
 6          $C' \leftarrow$ Cover-Make($A_{k+1-u}^{j-u}$)
 7          $C'' \leftarrow$ Cover-Merge($C', C''$)    {Computes $C(A_{k+1-u}^{k})$ by merging $C', C''$}
 8          if $\ell < |C''|$ then $\ell \leftarrow |C''|$; $s \leftarrow k$
 9          $j \leftarrow k$
10      Remove $a_k$ from $C''$
11  if $\ell < |C^{\text{left}}|$ then $\ell \leftarrow |C^{\text{left}}|$; $s \leftarrow (i+1)u$
12  return $\ell$, $s$
```

Fig. 5. A general scheme of the algorithm computing LISW in a range of width twice as large as window size

The LISW length and an index of LISW last symbol in the whole sequence can be obtained by running *LISW-Range* algorithm for each $i = 0, 1, \ldots, \lceil n/u \rceil - 2$ (Figure 7). Let us note that if we know the index of the last symbol of LISW, we can easily construct the cover for window ending at that symbol and read an LISW. This needs $O(u \log\log n)$ time, which is negligible.

Theorem 1. *LISW* algorithm (Figure 7) computes the LISW length and the index of LISW last symbol in input sequence $A = a_1 a_2 \ldots a_n$ for window size u.

Proof. *LISW-Range* algorithm computes the LISW length and the index of its last symbol for sequence A_{iu+1}^{iu+2u}. This algorithm is executed for each $0 \le i \le \lceil n/u \rceil - 2$, so each window of size u of A_1^n is considered in some execution of *LISW-Range*. From the obtained results the longest one is taken, so it must be the LISW length. Moreover, also the index of the last symbol of an LISW is found. □

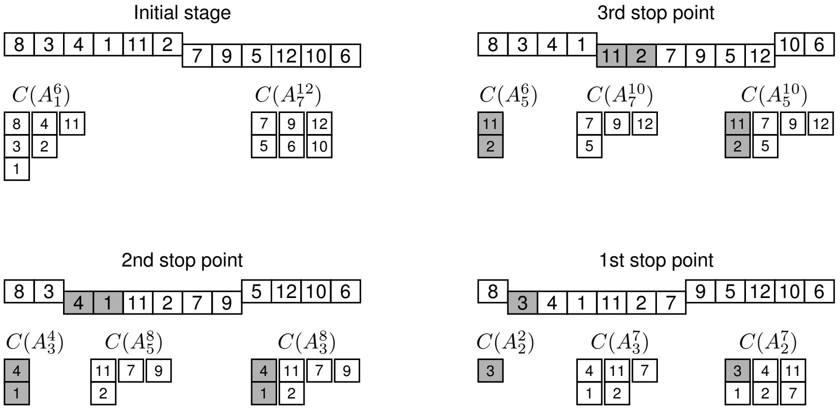


Fig. 6. Example of *LISW-Range* algorithm finding an LISW for $A = 8\,3\,4\,1\,11\,2\,7\,9\,5\,12\,10\,6$ and $u = 6$. There are a few stages: initial (at the beginning of the algorithm) and after visiting each stop point. Some symbols of the sequence are lowered to show the current window position. In the initial stage, both input covers C^{left} and C^{right} are shown. The gray cells show the symbols that just came to the window since last stop point: for them cover C' is constructed (cover at left side). The middle covers are C'' before merging and the right covers – C'' after merging

LISW(A)

```
1  $\ell \leftarrow 0$; $s \leftarrow 0$
2  $C^{\text{right}} \leftarrow$ Cover-Make($A_{n-u+1}^{n}$)
3  for $i \leftarrow \lceil n/u \rceil - 2$ downto $0$ do
4      $C^{\text{left}} \leftarrow$ Cover-Make($A_{iu+1}^{iu+u}$)
5      $\ell', s' \leftarrow$ LISW-Range($A, i, C^{\text{left}}, C^{\text{right}}$)
6      if $\ell' > \ell$ then $\ell \leftarrow \ell'$; $s \leftarrow s'$
7      $C^{\text{right}} \leftarrow C^{\text{left}}$
8  return $\ell$, $s$
```

Fig. 7. A general scheme of the algorithm computing LISW length

5 IMPLEMENTATION DETAILS AND COMPLEXITY ANALYSIS

According to Definition 1, a cover is composed of decreasing lists; nevertheless, this is only a concept. An internal organisation of a cover may be various, e.g., for the *Cover-Make* algorithm it does not matter how the cover is internally stored and it is possible to store it as an array without affecting the time complexities. For *Cover-Merge* algorithm, the internal representation is important since the algorithm splits and joins the 'lists'.

In [9] and [10], three internal representations were proposed. Below we examine each of them for the current problem, but firstly let us specify the time complexity formula of *LISW-Range* algorithm relating to the formulas introduced for the LICS problem [11]:

$$\tau_i = u\tau^{\text{del}} + \sum_{j=1}^{n_i^{\text{stop}}} \left(\tau^{\text{build}}(n_j) + \tau^{\text{find}}(n_j) + O(\min(n_j, \ell)\ell)\left(\tau^{\text{split}} + \tau^{\text{join}}\right)\right), \qquad (1)$$

where:

- n_i^{stop} – number of stop points for $A_{(i+1)u+1}^{(i+1)u+u}$, upper bounded by ℓ_{i+1},
- n_j – number of symbols between jth and $(j+1)$th stop point for $A_{(i+1)u+1}^{(i+1)u+u}$; by definition $(n_i^{\text{stop}}+1)$th stop point is the last symbol of $A_{(i+1)u+1}^{(i+1)u+u}$,
- τ^{del} – time complexity of removal of a single symbol from cover,
- $\tau^{\text{build}}(n_j)$ – time complexity of cover construction for n_j symbols,
- $\tau^{\text{find}}(n_j)$ – total time complexity of finding split points in cover lists in a single merging,
- τ^{split} – time complexity of splitting a single cover list,
- τ^{join} – time complexity of joining two cover lists.

A justification of formula (1) is as follows. Exactly u times we need to remove a single symbol from the cover (time τ^{del} per symbol). *LISW-Range* algorithm calls *Cover-Merge* procedure $\Theta(n_i^{\text{stop}})$ times. Before each merging, cover C' is constructed from n_j symbols. Before the lists are split it is necessary to find the split points (total time τ^{find} per single execution of *Cover-Merge* algorithm). Then, the lists are split and merged, and the total number of times these operations are performed is a multiple of the sizes of the merged covers, $|C'| \times |C''|$, [9, Lemma 5].

Considering the time complexity of *LISW* algorithm we should also take into account that we must construct a cover for each A_{iu+1}^{iu+u}, where $0 \le i < \lceil n/u \rceil$. The time complexity of computing such a single cover is denoted by τ^{init}, so the total time complexity of *LISW* algorithm can be expressed as:

$$\sum_{i=0}^{\lceil n/u \rceil - 1} \left(\tau^{\text{init}} + \tau_i\right). \qquad (2)$$

The actual values of the above terms depend on the internal cover representation and are determined below.

5.1 List-Based Cover Representation

In the list-based cover representation, time complexity of removal of a single symbol from a cover is $\tau^{\text{del}} = O(1)$ due to an auxiliary array of pointers to the symbols on the lists (we can rapidly localise the list the symbol is part of). The cost of maintaining this array is constant per element and has no influence on the total time complexity of the algorithm. (The same holds for the remaining ways of cover representation.) The time complexity of construction of small covers, $\tau^{\text{build}}(n_j) = O(n_j \log \ell)$. The time complexity of finding split points is $\tau^{\text{find}}(n_j) = O(\min(n_j, \ell)\ell + u)$ (by the same arguments as in [9]). Obviously, $\sum_{j=1}^{n_i^{\text{stop}}} n_j \leq u$. Moreover, $\tau^{\text{split}} = \tau^{\text{join}} = O(1)$ and $\tau^{\text{init}} = O(u \log \ell)$. Putting this values to (1) we obtain:

$$\begin{aligned}
\tau_i &= uO(1) \\
&\quad + \sum_{j=1}^{n_i^{\text{stop}}} (O(n_j \log \ell) + O(\min(n_j, \ell)\ell + u) + O(\min(n_j, \ell)\ell)(O(1) + O(1))) \\
&= O(u) + O(u \log \ell) + O(\min(u\ell, \ell^3)) + O(u\ell) + O(\min(u\ell, \ell^3)) = O(u\ell). \quad (3)
\end{aligned}$$

From (2) we have:

Corollary 2. Time complexity of *LISW* algorithm for list-based cover representation is:

$$\sum_{i=0}^{\lceil n/u \rceil - 1} (O(u \log \ell) + O(u\ell)) = O(n\ell). \quad (4)$$

5.2 Balanced-Tree-Based Cover Representation

In the variant with representation of cover as a list of balanced search trees, time complexities of components of the algorithm are:

$$\begin{aligned}
\tau^{\text{del}} &= \tau^{\text{split}} = \tau^{\text{join}} = O(\log u), \\
\tau^{\text{build}}(n_j) &= O(n_j \log \ell), \\
\tau^{\text{find}}(n_j) &= O(\min(n_j, \ell)\ell \log u), \\
\tau^{\text{init}} &= O(u \log \ell).
\end{aligned}$$

So, according to (1) we have

$$\begin{aligned}
\tau_i &= uO(\log u) + \sum_{j=1}^{n_i^{\text{stop}}} (O(n_j \log \ell) + O(\min(n_j, \ell)\ell \log u) \\
&\quad + O(\min(n_j, \ell)\ell)(O(\log u) + O(\log u))) \\
&= O(u \log u) + O(u \log \ell) + O(\min(u\ell, \ell^3) \log u) + O(\min(u\ell, \ell^3) \log u) \\
&= O(u \log u + \min(u\ell, \ell^3) \log u) = O(\min(u\ell, u + \ell^3) \log u). \quad (5)
\end{aligned}$$

From (2) we conclude:

Corollary 3. Time complexity of *LISW* algorithm for balanced-tree-based representation of the cover is:

$$\sum_{i=0}^{\lceil n/u\rceil-1}\big(O(u\log\ell)+O(\min(u\ell,u+\ell^3)\log u)\big)=O\left(\min\left(n\ell,n\left\lceil\frac{\ell^3}{u}\right\rceil\right)\log u\right). \quad (6)$$

5.3 List-of-Balanced-Tree-Based Cover Representation

Which of the two above-described cover representations is better depends on the LISW length. Since it is rather easy to bound it for each double-window-size range on the LIS lengths of leftmost and rightmost window (Lemma 2) we can always choose the right representation for the range. It is possible, however, to propose one more cover representation, which shares the assets of the two above-described ones in one data structure.

The idea is to store each cover list as an ordered list of balanced search trees (e.g., red-black trees) of size $\Theta(e)$ [11], which will be determined later. Moreover, each tree is accompanied with a single variable storing its minimal value, so it is accessible in a constant time.

For this representation, time complexities of algorithm components are:

$$\begin{aligned}
\tau^{\text{del}} &= \tau^{\text{split}}=\tau^{\text{join}}=O(\log(e+1)),\\
\tau^{\text{build}}(n_j) &= O(n_j\log\log n),\\
\tau^{\text{find}}(n_j) &= O(\min(n_j,\ell)\ell\log(e+1)+u/e+\ell),\\
\tau^{\text{init}} &= O(u\log\log n).
\end{aligned}$$

When we put these values to (1) and (2) we obtain:

$$\begin{aligned}
\tau_i &= uO(\log(e+1))+\sum_{j=1}^{n_i^{\text{stop}}}(O(n_j\log\log n)+O\left(\min(n_j,\ell)\ell\log(e+1)+\frac{u}{e}+\ell\right)\\
&\quad+O\left(\min(n_j,\ell)\ell)(O(\log(e+1))+O(\log(e+1))\right))\\
&= O(u\log(e+1))+O(u\log\log n)+O\left(\min(u\ell,\ell^3)\log(e+1)+\frac{u\ell}{e}+\ell^2\right)\\
&\quad+O(\min(u\ell,\ell^3)\log(e+1))\\
&= O\left(u\log\log n+\min(u\ell,u+\ell^3)\log(e+1)+\frac{u\ell}{e}\right) \qquad (7)
\end{aligned}$$

and the total time complexity of *LISW* algorithm:

$$\begin{aligned}
&\textstyle\sum_{i=0}^{\lceil n/u\rceil-1}\left(O(u\log\log n)+O\left(u\log\log n+\min(u\ell,u+\ell^3)\log(e+1)+\frac{u\ell}{e}\right)\right)\\
&\qquad=O\left(n\log\log n+\min\left(n\ell,n\left\lceil\tfrac{\ell^3}{u}\right\rceil\right)\log(e+1)+\tfrac{n\ell}{e}\right). \qquad (8)
\end{aligned}$$

Calculating a derivative of the above complexity and equalling it to 0, it can be found that this value is minimal for $e = \Theta(\lceil u/\ell^2 \rceil)$. Therefore, the time complexity of this variant is

$$O\left(n \log\log n + \min\left(n\ell, n\left\lceil \frac{\ell^3}{u} \right\rceil\right) \log\left\lceil \left(\frac{u}{\ell^2} + 1\right) \right\rceil\right). \tag{9}$$

The value of e may be determined up to the factor 4 relating on Lemma 2 if at the initial stage of the algorithm the LIS length will be determined for every A_{iu+1}^{iu+u}, where $0 \le i \le \lceil n/u \rceil - 1$. This does not affect the total time complexity of the algorithm.

The last question is how to update the trees in the lists without influencing the time complexity of the algorithm. Particularly, we have to be sure that after each operation on the list of balanced search trees, i.e., splitting, removing a symbol, joining, each tree is of size $\Theta(e)$. (The only exception is for a cover list containing less than $\Theta(e)$ symbols, for which the cover list is composed of a single tree.) To assure that, we require that each tree contains always $[e, 2e]$ symbols. After each removal of a symbol from a tree (in $O(\log(e+1))$ time), it is verified whether the tree size is less than e. If such a case occurred, this tree is joined with one of the neighbour trees in the list (in $O(\log(e+1))$ time) and we obtain the tree of size in the range $[e, 3e]$. Then, if necessary, i.e., if the tree size is larger than $2e$, the tree is split into two roughly equal-size trees in order to the size of each of the resulting trees is in the range $[e, 2e]$. For this task, it may be necessary to find a median of the elements in a tree. To find a median rapidly, each node of the tree must be augmented by an additional field containing number of elements in the subtree rooted at the that node. The data in these additional fields can be updated without influence on the time complexity of other operations on the tree [23, 5, 26]. Time complexity of the above-described tree-size normalisation is $O(\log(e+1))$, which is exactly the same as the time of removal of a single element of the tree, so it does not matter in time complexity analysis.

Similarly, the trees are normalised after split and join operations. After each split we obtain two trees. One of them is moved to the other cover list. In both cover lists, it is checked whether the obtained tree is of proper size. If not, it is joined with the neighbour tree and eventually, the joined tree is split. The time complexity of this normalisation is $O(\log(e+1))$, so it does not influence the time complexity of raw split and join operations. From the above, we can conclude that it is possible to assure that each tree is of size in the range $[e, 2e]$ without influencing the time complexity of the algorithm. Therefore:

Corollary 4. Time complexity of *LISW* algorithm for the list-of-balanced-trees cover representation is

$$O\left(n \log\log n + \min\left(n\ell, n\left\lceil \frac{\ell^3}{u} \right\rceil\right) \log\left\lceil \left(\frac{u}{\ell^2} + 1\right) \right\rceil\right). \tag{10}$$

It can be noticed that none of the previous variants, i.e., list-based and balanced-trees-based, is faster, in terms of the worst-case time complexity, than the list-of-balanced-trees-based variant. Previous variants are easier to implement, so the result length can be estimated. To this end, it is necessary to compute LIS for all windows A_{iu+1}^{iu+u} for all valid i in total time $O(n \log \ell)$ and then to estimate ℓ according to Lemma 2 up to factor 2, so the faster of the list-based and balanced-trees-based algorithm can be chosen. In all cases, it will be at most as fast as the list-of-balanced-trees-based algorithm.

6 CONCLUSIONS

We presented three variants of the algorithm finding the LISW length and LISW position in the input sequence. The variants differ in employed data structures, which is reflected in the obtained time complexity. The time complexities of the literature algorithms for the LISW problem are $O(n \log\log n + n\ell)$ [3], $O(n\ell)$ [7], and $O(n \log^2 n)$ [28].

The time complexity of the list-based variant is the same as of the second mentioned algorithm and better than the first one. Our best variant (list-of-balanced-tree-based cover representation) is faster if

$$\ell = o\left(u^{1/2} / \log^{1/2} u\right)$$

and not worse if

$$\ell = O\left(u^{1/2}\right).$$

Comparing our best variant of the worst-case time complexity $O(n \log\log n + \min(n\ell, n\lceil \ell^3/u \rceil) \times \log\lceil u/\ell^2 + 1 \rceil)$ to the most recent algorithm by Tiskin [28] we are better if

$$\ell = o\left(u^{1/3} \frac{\log^{2/3} n}{\log^{1/3} u}\right). \tag{11}$$

It is, however, difficult to say which algorithm will be faster in practice, since the constant hidden in the big-O notation could be large in all the existing algorithms. The worst-case time complexities suggest that our algorithm could win if the window size is much smaller than the input sequence and the resulting subsequence will not be to long.

It is an open question what is the lower bound of the time complexity for this problem. We only know that it cannot be better than lower bound for the LIS problem, because in such a case we could break the bound for the LIS problem by setting $u = n$ and using LISW-computing algorithm.

Acknowledgement

The research of this project was partially supported by the Minister of Science and Higher Education grants 3177/B/T02/2008/35 and N N516 441938. The author thanks the anonymous reviewer for the notification of a small mistake in the caption of the author's pseudocode (used also in this paper) from [9].

REFERENCES

[1] Albert, M. H.—Atkinson, M. D.—Nussbaum, D.—Sack, J.-R.—Santoro, N.: On the Longest Increasing Subsequence of a Circular List. Information Processing Letters, Vol. 101, 2007, pp. 55–59.

[2] Aldous, D.—Diaconis, P.: Longest Increasing Subsequences: From Patience Sorting to the Baik-Deift-Johansson Theorem. Bulletin of the AMS, Vol. 36, 1999, pp. 413–432.

[3] Albert, M. H.—Golynski, A.—Hamel, A. M.—López-Ortiz, A.—Rao, S. S.—Safari, M. A.: Longest Increasing Subsequences in Sliding Windows. Theoretical Computer Science, Vol. 321, 2004, pp. 405–414.

[4] Baik, J.—Deift, P.—Johansson, K.: On the Distribution of the Length of the Longest Increasing Subsequence in Random Permutations. Journal of American Mathematical Society, Vol. 12, 1999, pp. 1119–1178.

[5] Brass, P.: Advanced Data Structures. Cambridge University Press 2008, 472 pp.

[6] Bespamyatnikh, S.—Segal, M.: Enumerating Longest Increasing Subsequences and Patience Sorting. Information Processing Letters, Vol. 76, 2000, No. 1-2, pp. 7–11.

[7] Chen, E.—Yang, L.—Yuan, H.: Longest Increasing Subsequences in Windows Based on Canonical Antichain Partition. Theoretical Computer Science, Vol. 378, 2007, No. 3, pp. 223–236.

[8] Dalevi, D.—Eriksen, N.: Expected Gene-Order Distances and Model Selection in Bacteria. Bioinformatics, Vol. 24, 2008, No. 11, pp. 1332–1338.

[9] Deorowicz, S.: An Algorithm for Solving the Longest Increasing Circular Subsequence Problem. Information Processing Letters, Vol. 109, 2009, No. 12, pp. 630–634.

[10] Deorowicz, S.: On Some Variants of the Longest Increasing Subsequence Problem. Theoretical and Applied Informatics, Vol. 21, 2009, No. 3–4, pp. 135–148.

[11] Deorowicz, S.—Grabowski, Sz.: On Two Variants of the Longest Increasing Subsequence Problem. In: Man-Machine Interactions, Cyran, K. A., Kozielski, S., Peters, J. F., Staczyk, U. and Wakulicz-Deja, A. (Eds.), Series Advances in Intelligent and Soft Computing. Springer-Verlag Berlin Heidelberg 2009, pp. 541–549.

[12] Delcher, A. L.—Kasif, S.—Fleischmann, R. D.—Peterson, J.—White, O.—Salzberg, S. L.: Alignment of While Genomes. Nucleic Acids Research, Vol. 27, 1999, No. 11, pp. 2369–2376.

[13] Deogun, J. S.—Yang, J.—Ma, F.: Emagen: An Efficient Approach to Multiple Whole Genome Alignment. In: Proceedings of the Second Asia-Pacific Bioinformatics Conference, 2004, pp. 113–122.

[14] van Emde Boas, P.—Kaas, R.—Zijlstra, E.: Design and Implementation of an Efficient Priority Queue. Mathematical Systems Theory, Vol. 10, 1977, pp. 99–127.

[15] van Emde Boas, P.—Kaas, R.—Zijlstra, E.: Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. Information Processing Letters, Vol. 6, 1977, No. 3, pp. 80–82.

[16] Elmasry, A.: The Longest Almost-Increasing Subsequence. Information Processing Letters, Vol. 110, 2010, No. 16, pp. 655–658.

[17] Erdös, P.—Szekeres, G.: A Combinatorial Problem in Geometry. Compositio Mathematica, Vol. 2, 1935, pp. 463–470.

[18] Faraut, T.—de Givry, S.—Chabrier, P.—Derrien, T.—Galibert, F.—Hitte, C.—Schiex, T.: A Comparative Genome Approach to Marker Ordering. Bioinformatics, Vol. 23, 2007, No. 2, pp. e50–e56.

[19] Golab, L.—Karloff, H. J.—Korn, F.—Saha, A.—Srivastava, D.: Sequential Dependencies. In: Proceedings of the Very Large Database Conference, Vol. 2, 2009, pp. 574–585.

[20] Golumbic, M. C.: Algorithmic Graph Theory and Perfect Graphs. 2nd ed. Academic Press 2004, 340 pp.

[21] Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press 1997, 554 pp.

[22] Hunt, J. W.—Szymanski, T. G.: A Fast Algorithm for Computing Longest Common Subsequences. Communications of the ACM, Vol. 20, 1977, No. 5, pp. 350–353.

[23] Knuth, D. E.: The Art of Computer Programming. Vol. 3: Sorting and Searching. 2nd ed. Addison Wesley Longman 1998, 800 pp.

[24] Kurtz, S.—Phillippy, A.—Delcher, A. L.—Smoot, M.—Shumway, M.—Antonescu, C.—Salzberg, S. L.: Versatile and Open Software for Comparing Large Genomes. Genome Biology, Vol. 5, 2004, No. 2, pp. R12.1–R12.9.

[25] Lin, F.-M.—Huang, H.-D.—Chang, Y.-C.—Tsou, A.-P.—Chan, P.-L.—Wu, L.-C.—Tsai, M.-F.—Horng, J.-T.: Database to Dynamically Aid Probe Design for Virus Identification. IEEE Transactions on Information Technology in Biomedicine, Vol. 10, 2006, No. 4, pp. 705–713.

[26] Mehlhorn, K.—Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer-Verlag, Berlin Heidelberg 2008, 312 pp.

[27] Schensted, C.: Longest Increasing and Decreasing Subsequences. Canadian Journal of Mathematics, Vol. 13, 1961, pp. 179–191.

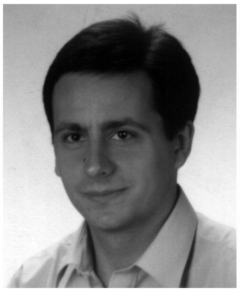
[28] Tiskin, A.: Semi-Local String Comparison: Algorithmic Techniques and Applications. arXiv.org, 2010, pp. 1–115 (arXiv:0707.3619v15).

[29] Tseng, C.-T.—Yang, C.-B.—Ann, H.-Y.: Minimal Height and Sequence Constrained Longest Increasing Subsequence. Journal of Internet Technology, Vol. 10, 2009, No. 2, pp. 173–178.

[30] Yang, I.-H.—Chen, Y.-C.: Fast Algorithms for the Constrained Longest Increasing Subsequence Problems. In: Proceedings of the 25th Workshop on Combinatorial Mathematics and Computation Theory, 2008, pp. 226–231.

[31] Young, A.: On Quantitative Substitutional Analysis. Proceedings of the London Mathematical Society, Vol. s2–28, 1928, No. 1, pp. 255–292.

[32] Zhang, H.: Alignment of BLAST High-Scoring Segment Pairs Based on the Longest Increasing Subsequence Algorithm. Bioinformatics, Vol. 19, 2003, No. 11, pp. 1391–1396.



Sebastian Deorowicz received his M. Sc. degree from Silesian University of Technology in 1998 and Ph. D. degree in the same university in 2003, both in computer science. His research interest include string matching, sequence alignment, data compression, and combinatorial optimization. He has published about 30 journal and conference papers. He is currently an Assistant Professor at Silesian University of Technology in Gliwice.