Information Processing Letters 109 (2009) 630–634

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

An algorithm for solving the longest increasing circular subsequence problem

Sebastian Deorowicz

Silesian University of Technology, Institute of Computer Science, 44-100 Gliwice, Akademicka 16, Poland

ARTICLE INFO

Article history:
Received 23 July 2007
Accepted 19 February 2009
Available online 27 February 2009
Communicated by S.E. Hambrusch

Keywords:
Algorithms
Longest increasing subsequence
Longest increasing circular subsequence
String matching

1. Introduction

The problem of finding a longest increasing circular subsequence (LICS) [1] in a sequence of integers is to find a longest subsequence of any rotation of a given sequence (a subsequence is obtained by removing zero or more elements) such that each integer of the subsequence is smaller than the integer that follows it. The LICS problem is a generalization of a classic longest increasing subsequence (LIS) problem, with applications in several areas, e.g., research on genomes [5] and as a tool for solving the widely known longest common subsequence problem [6]. The proved worst-case time lower bound for LIS is $\Omega(n \log n)$ in the comparison model [6]. When the sequence is a permutation of all integers from the range $[1, n]$, in the RAM model this result can be improved to $\Omega(n \log \log n)$ by using van Emde Boas trees [4].

In this paper, a hybrid algorithm solving the longest increasing circular subsequence problem is presented. It is based on a cover representation of the sequence and on two observations. First, it is sufficient to compute an LIS only for some rotations. Second, merging two precomputed covers of subsequences is faster than computing the cover from scratch. Two efficient cover merging techniques are introduced, which allow to construct the LICS computing algorithm of complexity competitive to the best known. The worst case time complexity of the presented algorithm is $O(\min(n\ell, n \log n + \ell^3 \log n))$, where n is the sequence length and ℓ is the LICS length.

The paper is organized as follows. Section 2 contains the definitions and the problem background. In Section 3, an algorithm solving the LIS problem based on a concept of cover of a sequence, important for further discussion, is described. The proposed algorithm solving the LICS problem is given in Section 4. The last section concludes the paper.

2. Definitions and background

Let $S = s_1 s_2 \ldots s_n$ be a sequence composed of unique symbols over an integer alphabet $\mathbb{Z}$. A sequence S' is a *subsequence* of S if it can be obtained from S by removing zero or more symbols. The *longest increasing subsequence* (LIS) problem for S is to find a longest possible subsequence of S such that its symbols increase monotonically, i.e., $s_{i_1} < s_{i_2} < \cdots < s_{i_k}$ for $i_1 < i_2 < \cdots < i_k$. S_i^j denotes $s_{f(i)} \ldots s_{f(j)}$, if $f(i) \leqslant f(j)$, and $s_{f(i)} \ldots s_n s_1 \ldots s_{f(j)}$ otherwise, where $f(x) = ((x-1) \bmod n) + 1$. In a *longest increasing circular subsequence* problem (LICS), the goal is to find an LIS among S_i^{i-1}, for each $1 \leqslant i \leqslant n$.

E-mail address: Sebastian.Deorowicz@polsl.pl.

0020-0190/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2009.02.019

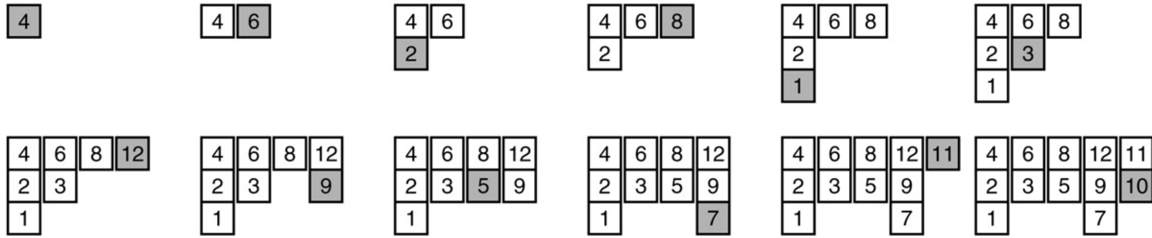

Fig. 1. Example of the algorithm finding the greedy cover of $S = 4, 6, 2, 8, 1, 3, 12, 9, 5, 7, 11, 10$. The just placed elements are gray. The lists are vertical. Note that the most recent (the lowest) cells of each list are increasing.

It is convenient to assume that sequence S is a permutation of all integers from the range $[1, n]$. The general problem can be transformed to fit this assumption by relabeling the elements of S in such a way that each element is replaced by its position in a sorted sequence of S. This relabeling can be done in $O(n \log n)$ time and is neglected in further calculations.

The simplest way of finding LICS is to execute an $O(n \log\log n)$-time LIS-computing algorithm n times, i.e., for S_1^n, S_2^1, ..., S_n^{n-1} and take a longest subsequence, which needs $O(n^2 \log\log n)$ time. Albert et al. [1] proposed two faster algorithms. The first one is of time complexity $O(n\ell \log n)$, where ℓ is the length of LICS (LLICS). The second one needs $O(n^{3/2} \log n)$ time, but can yield erroneous results with a tiny probability. Another possibility is to adapt the algorithms for related problems. The LISW problem [2] is to find the LIS among all fixed-size windows over the base sequence. If the base sequence is SS (two concatenated S sequences) and the window size is n, it is equivalent to the LICS problem. The LISW solving algorithms applied for the LICS problem work in time $O(n \log\log n + n\ell)$ [2] and $O(n\ell)$ [3]. Very recently, Tiskin proposed a set of new techniques to solve the longest common subsequence (LCS) and several related problems [7]. In particular, he managed to achieve $O(n^{3/2})$ time for the LICS problem. His technique is based on the alignment directed acyclic graphs and so-called highest-score matrices representing the matches between two sequences in the LCS problem.

3. The algorithm for finding a longest increasing subsequence

Gusfield [6, pp. 287–290] presents a simple algorithm computing an LIS. The algorithm splits the elements of sequence S into an ordered set of decreasing subsequences (lists) called a *cover* $C(S)$. There are many possible covers of the sequence, but the algorithm produces the one of the smallest *size*, when the *size* of the cover, $|C(S)|$, is the number of lists in it.

The cover-making (CM) algorithm for S works as follows. Each successive element of S (starting from s_1) extends the leftmost possible decreasing list of the cover or starts a new list if no list can be extended (Fig. 1). As list tails are increasing, the appropriate lists can be found with a binary search ($O(n \log \ell)$ time) or van Emde Boas trees [4] ($O(n \log\log n)$ time).

The obtained cover, called a *greedy cover*, has several important properties (all proved in [6]):

(1) the position of the list in the ordered set which each symbol belongs to is the length of an LIS ending at this symbol,
(2) it is unique, i.e., there is no other cover satisfying property (1),
(3) its size is the length of a longest increasing subsequence of S, LLIS(S).

Our algorithms for determining an LICS deal with greedy covers, but for brevity we will write "cover" instead of "greedy cover" in the remainder of this paper. The *cover read* of the sequence S, denoted by $R(S)$, is the concatenation of the successive decreasing sequences forming the cover of the sequence, e.g., the cover read of the sequence $S = 4, 6, 2, 8, 1, 3, 12, 9, 5, 7, 11, 10$ is $R(S) = 4, 2, 1, 6, 3, 8, 5, 12, 9, 7, 11, 10$ (cf. Fig. 1). $C(S)[i]$ denotes the ith decreasing list of cover $C(S)$, e.g., $C(S)[4]$ in Fig. 1 is $12, 9, 7$.

4. Algorithms for finding an LICS

4.1. The foundations of the proposed algorithms

The algorithms developed in this paper work on a cover representation of the sequence and its rotations, since such a representation immediately gives the length of an LIS of the sequence and is sufficient to compute LIS in linear time [6]. Now, we prove some necessary lemmas.

Lemma 1. *Let sequence S be a concatenation of S' and S'', i.e., $S = S'S''$. The cover of $S'S''$ is identical to the covers of $R(S')R(S'')$ and of $S'R(S'')$.*

Proof. The CM algorithm works on the successive symbols of sequence S, so it computes at first $C(S')$, which is obviously the same as $C(R(S'))$. Now, we proceed by recurrence on the length of sequence S'', denoted by m. The lemma is valid for $m = 1$ since $S'' = R(S'')$ then.

Let now $m > 1$ and the second sequence is $S''x$. If x is the last symbol of $R(S''x)$, the lemma is valid, so let us assume otherwise. We have:

$$R(S'') = C(S'')[1] \ldots C(S'')[p]\, C(S'')[p+1] \ldots C(S'')[k],$$

and

$R(S''x) = C(S'')[1]\ldots C(S'')[p]\,x\,C(S'')[p+1]\ldots C(S'')[k]$.

The tail of $C(S'')[p]$, denoted t for short, is larger than x and all the symbols from lists $C(S'')[p+1], \ldots, C(S'')[k]$ are larger than t. Therefore, when the CM algorithm processes $R(S')R(S''x)$ and arrives at x, it places x at the end of some list not to the right of the list t was placed. Then, the CM algorithm inserts the symbols from $C(S'')[p+1], \ldots, C(S'')[k]$ into the cover, but all they must be placed to the right of the list containing t, so their positions are unaffected by the prior insertion of x. Therefore, $C(R(S')R(S'')x)$ and $C(R(S')R(S''x))$ are the same, so processing S'' in a cover-read order does not change the resulting cover. □

In the LICS problem, all rotations are checked, so without loss of generality we assume in the remainder of the paper that s_1 is the largest symbol in S. (The pre-rotation cost is $O(n)$.) The LLIS of the pre-rotated sequence is denoted by ℓ' and it is known that $\ell' \leqslant \ell \leqslant 2\ell'$ [1].

Lemma 2. *Let the elements heading the lists of $C(S_1^n)$ be called stop points and s_1 be the largest symbol of S. If the sequence is rotated to the left by $1 \leqslant i < n$ symbols then the head of the last list in the corresponding cover $C(S_{n+1-i}^{n-i})$ is some stop point.*

Proof. From Lemma 1: $C(S_{n+1-i}^{n-i}) = C(S_{n+1-i}^{n} R(S_1^{n-i}))$. The CM algorithm builds the cover C from S_{n+1-i}^{n}. Then, it extends it with successive lists of $C(S_1^{n-i})$. A head of each of these lists is a stop point and it can extend some list of C or start a new one. The symbols in each list are monotonically decreasing, so only the head can start a new list in C. However, at least stop point s_1 (the maximal value in S) from $C(S_1^{n-i})$ must start a new list in C, so the last list of the final cover must be headed with a stop point. □

Lemma 3. *The* LLICS(S) *is equal to the maximal value of LLIS for rotations of S (s_1 is the largest symbol of S) ending at stop points.*

Proof. From Lemma 2, the head of the last list is some stop point s_k for any rotated sequence S_i^{i-1}, $1 \leqslant i \leqslant n$, which means that LLIS ending s_k in this rotation is equal to LLIS(S_i^{i-1}). For any stop point s_j, a longest LIS ending s_j among all the rotations can be found in S_{j+1}^{j}. Therefore, LLICS(S) is equal to the maximum of the LLIS of the rotations ending at stop points. □

Let $S = S'S''$, $C' = C(S')$ and $C'' = C(S'')$ are known, and $C = C(S)$ is to be computed. According to Lemma 1, the CM algorithm can add sequence S'' to cover C' in a cover-read order.

Theorem 4. *The cover merging algorithm presented in Fig. 2 computes the cover of $S'S''$, where $C' = C(S')$ and $C'' = C(S'')$ are given.*

Proof. The outer loop invariant of the algorithm is *C' stores cover for $S'C(S'')[1]\ldots C(S'')[i-1]$*. By convention $C(S'')[1]\ldots C(S'')[0]$ is an empty sequence.

```
01 for i ← 1 to |C″| do
02   if C″[i] head is larger than C′[|C′|] tail then
03     Add to C′ an empty list
04   j ← |C′|
05   while C″[i] is not empty and j > 1 do
06     Find the largest symbol p of C″[i] smaller than C′[j − 1] tail
07     Move the symbols larger than p from C″[i] to C′[j]
08     j ← j − 1
09   Append the rest (if any) of C″[i] to C′[1]
```

Fig. 2. A general scheme of the cover merging procedure. (If there is no such a symbol p in line 06, no symbols are moved in line 07.)

```
01 Rotate S as s1 is the maximal symbol of S
02 C″ ← Compute C(S_1^n); j ← n
03 if s_n is a stop point then ℓ ← |C″| else ℓ ← 0
04 Remove s_n from C″
05 for k ← n − 1 downto 1 do
06   if s_k is a stop point then
07     C′ ← Compute C(S_{k+1}^j)
08     C″ ← Compute C(S_{k+1}^k) by combining C′, C″
09     ℓ ← max(ℓ, |C″|)
10     j ← k
11   Remove s_k from C″
12 return ℓ
```

Fig. 3. A general scheme of the algorithm computing the LLICS.

Before the loop, C' stores the cover of S'. Let us now assume that for given i: C' stores cover for $S'C(S'')[1]\ldots C(S'')[i-1]$. List $C''[i]$ is decreasing and is processed as follows. The symbols of $C''[i]$ larger than the C' last list tail are moved to C' as a new list. For all $1 < j \leqslant |C'|$, the symbols larger than $C'[j-1]$ tail and smaller than $C'[j]$ tail are appended to $C'[j]$. The remaining symbols (if any) are smaller than $C'[1]$ tail and are appended to $C'[1]$. This procedure places the symbols exactly as the CM algorithm, so after incrementing i, C' stores cover of $S'C(S'')[1]\ldots C(S'')[i-1]$ and the outer loop invariant is true.

After completing the outer loop, C' stores the cover of $S'R(S'')$ and from Lemma 1 the theorem is proved. □

Lemma 5. *When merging covers C' of r lists and C'', the elements of each C'' list are compared to tails of at most $r+1$ last lists of C'.*

Proof. We follow by recurrence on i being the list number of C''. For $i = 1$ this is obvious. For $i > 1$, $C''[i]$ tail is placed at the end of some kth list of C' but not far than $r-1$ lists from the end. The last comparison was to $C'[k-1]$ tail (at most r lists from the end of C'). The $C''[i+1]$ tail is larger than $C''[i]$ tail, so the last comparison when processing $C''[i+1]$ can be at most with the $C'[k]$ tail. This is one list to the right than $C''[i]$ tail was compared to. While processing a single list of C'', the C' size increases by at most 1. Therefore, while processing $(i+1)$th list of C'' at most $r+1$ last lists of C' are visited. □

A general scheme of the proposed algorithm is presented in Fig. 3. Sequence S is rotated n times by one symbol and for some (possibly all) rotations the cover is computed. The algorithms return the LLICS, but knowing the largest cover it is also easy to compute an LICS.

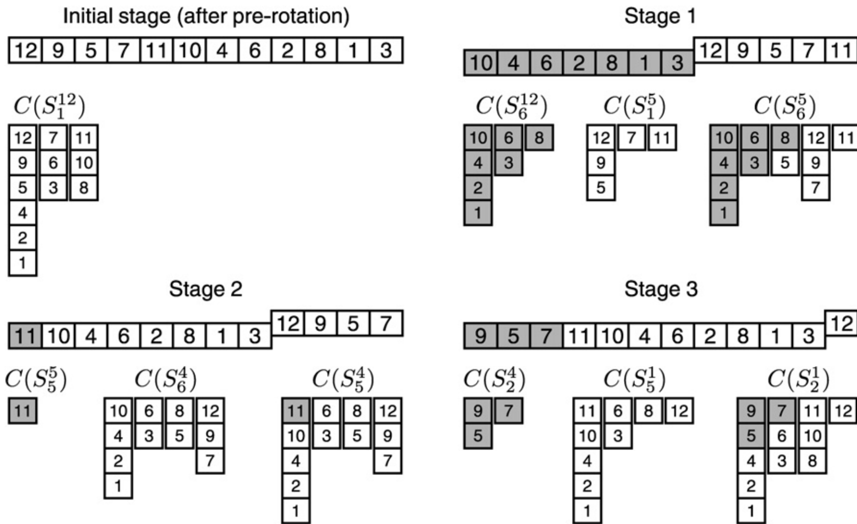

Fig. 4. Example of the list-based algorithm in work. The grayed cells denote the cells which are rotated in each stage. Some cells are pushed down to indicate the place where the original sequence S starts.

Theorem 6. *The algorithm presented in Fig.* 3 *computes the* LLICS(S).

Proof. Let the loop invariant be: *C'' stores the cover of S^k_{j+1} and ℓ stores the maximal LLIS for all stop points in range* $[k+1,n]$.

Before the loop, C'' stores the cover of $S^{n-1}_{n+1} = S^{n-1}_1$ and ℓ stores LLIS(S^n_1) if s_n is a stop point. Let us now assume the loop invariant is true for some k. Then, if s_k is not a stop point, it is removed from C'' (the ℓ value is unchanged), so decrementing k reestablishes the loop invariant for the next iteration. Let us now assume that s_k is a stop point. In line 07, cover C' of S^j_{k+1} is computed while C'' stores the cover of S^k_{j+1}. Then, the cover for S^k_{k+1} is computed (line 08). After that (line 09), ℓ is updated and stores the largest LLIS value for all stop points in range $[k,n]$. In line 10, j is substituted by k, so C'' stores cover of S^k_{j+1}. Then s_k is removed from C'' and decrementing k reestablishes the loop invariant for the next iteration. When the loop is completed, ℓ stores the maximal value of LLIS for all stop points. From Lemma 3, this is LLICS(S). □

4.2. List-based cover merging

The LLICS computing algorithm initially builds $C = C(S)$ and then cyclically, for all rotations ending at stop points, makes use of the cover merging technique as follows. Firstly, the symbols to the next stop point (excluding it) are removed from C obtaining C''. There is an auxiliary array storing n pointers to all symbols in the actual cover(s), so each symbol is located in constant time. (After initialization this array need not be maintained.) Between each $(i-1)$th and ith stop point (by convention, 0th stop point is s_n) m_i elements are removed in $O(m_i)$ time. Secondly, cover C' of the rotated symbols is built in $O(m_i \log \ell)$ time. Thirdly, C' and C'' are *merged* to obtain the cover for the actual rotation. For ith stop point, C' contains m_i elements, so it is composed of $O(m_i)$ lists. From Lemma 5, the number of constant time moves (Fig. 2, line 07) of symbols is $O(m_i\ell)$. Finding the symbol p (Fig. 2, line 06) costs $O(m_i\ell + n)$, since there are $O(n)$ symbols in C'' and only once each symbol can be decided to be larger than the tail of the processed list of C' and if it is smaller, searching is stopped.

There are ℓ' stop points, and for each of them the above procedure is repeated, so the total cost is $\sum_{i=1}^{\ell'}(m_i \log \ell + m_i\ell + n) = O(n\ell)$, since $m_1 + \cdots + m_{\ell'} = O(n)$. Adding the time of the initial cover computation, the time complexity $O(n \log \ell + n\ell) = O(n\ell)$ is obtained. Fig. 4 shows the algorithm in operation.

4.3. Red-black-tree-based cover merging

Now we show how to trade simplicity for effectiveness. An important part of the cover-merging algorithm complexity comes from: (i) locating the split element of list $C''[i]$ and (ii) moving the elements form $C''[i]$ to C'. Moving elements between lists takes $O(1)$ time, but looking for the split element is costlier. Use of a more sophisticated data structure like red-black tree instead of list changes the proportions between these operations, since both of them need $O(\log n)$ time. Therefore, the time complexity of a single cover merging is $O(m_i\ell \log n)$. The size of C' can be also bounded as $O(\ell)$, so the merging takes $O(\ell^2 \log n)$ time. Summing it over all stop points gives $O(\ell^3 \log n)$.

In this case, the initial cover computing takes $O(n \log n)$, so the total worst-case time complexity of the red-black-tree-based algorithm is $O((n + \ell^3) \log n)$.

Moreover, a hybrid of the two algorithms can be constructed. Since LLICS(S) $\leqslant$ 2 LLIS(S) [1], one can choose, according to LLIS(S) the cover merging procedure. This hybrid algorithm works in $O(\min(n\ell, n \log n + \ell^3 \log n))$ worst-case time: This is better than the complexity of the fastest known algorithms [3,7] if both $\ell = \omega(\log n)$ and $\ell = o(n^{1/2}/\log^{1/2} n)$, and not worse when $\ell = O(n^{1/2})$.

5. Conclusions

The question how far from the worst-case lower bound we actually are remains open. At present, the worst-case time complexities of the known algorithms are much worse than the worst-case complexity for the LIS problem. We point out also that no faster algorithm than $\Theta(n \log\log n)$ time can be constructed for the LICS problem. Otherwise, it could be used to solve the LIS problem breaking its proved lower bound. It would suffice to find the maximum x in S, extend S to $s_1 \ldots s_n (x+1) \ldots (x+n)$ and use the LICS algorithm.

Acknowledgements

The author thanks Zbigniew J. Czech and Szymon Grabowski for fruitful discussions on the problem and for comments on the paper, and the reviewers for their constructing remarks. The research of this project was supported by the Minister of Science and Higher Education grants 3177/B/T02/2008/35.

References

[1] M.H. Albert, M.D. Atkinson, D. Nussbaum, J.-R. Sack, N. Santoro, On the longest increasing subsequence of a circular list, Inform. Process. Lett. 101 (2007) 55–59.
[2] M.H. Albert, A. Golynski, A.M. Hamel, A. López-Ortiz, S.S. Rao, M.A. Safari, Longest increasing subsequences in sliding windows, Theoret. Comput. Sci. 321 (2004) 405–414.
[3] E. Chen, L. Yang, H. Yuan, Longest increasing subsequences in windows based on canonical antichain partition, Theoret. Comput. Sci. 378 (3) (2007) 223–236.
[4] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Systems Theory 10 (1977) 99–127.
[5] T. Faraut, S. de Givry, P. Chabrier, T. Derrien, F. Galibert, C. Hitte, T. Schiex, A comparative genome approach to marker ordering, Bioinformatics 23 (2) (2007) e50–e56.
[6] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, USA, 1999.
[7] A. Tiskin, Semi-local string comparison: Algorithmic techniques and applications, Math. in Comput. Sci. 1 (4) (2008) 571–603.