

Longest Increasing Subsequences in Windows based on Canonical Antichain Partition^{*}

Erdong Chen, Linji Yang^{*}, Hao Yuan

*Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, PRC*

Abstract

Given a sequence $\pi_1\pi_2\dots\pi_n$, a longest increasing subsequence (LIS) in a window $\pi\langle l, r \rangle = \pi_l\pi_{l+1}\dots\pi_r$ is a longest subsequence $\sigma = \pi_{i_1}\pi_{i_2}\dots\pi_{i_T}$ such that $l \leq i_1 < i_2 < \dots < i_T \leq r$ and $\pi_{i_1} < \pi_{i_2} < \dots < \pi_{i_T}$. We consider the LISW problem, which is to find the longest increasing subsequences in a sliding window of fixed-size w over a sequence. Formally, it is to find a LIS for every window in a set $S_{\text{FIX}} = \{\pi\langle i+1, i+w \rangle \mid 0 \leq i \leq n-w\} \cup \{\pi\langle 1, i \rangle, \pi\langle n-i, n \rangle \mid i < w\}$. By maintaining a *canonical antichain partition* in windows, we present an optimal *output-sensitive* algorithm to solve this problem in $O(\text{OUTPUT})$ time, where OUTPUT is the sum of the lengths of the $n+w-1$ LISs in those windows of S_{FIX} . In addition, we propose a more generalized problem called LISSET problem, which is to find a LIS for every window in a set S_{VAR} containing *variable-size* windows. By applying our algorithm, we provide an efficient solution for the LISSET problem to output a LIS (or all the LISs) in every window which is better than the straightforward generalization of classical LIS algorithms. An upper bound of our algorithm on the LISSET problem is discussed.

Key words: Longest Increasing Subsequences, Canonical Antichain Partition, Data Streaming Model

^{*} A preliminary version of this article appears in Proceedings of the 16th Annual International Symposium on Algorithms and Computation, Springer LNCS, Vol. 3827, pp. 1153 - 1162, Hainan, China, December 19-21, 2005.

^{*} Corresponding author.

Email addresses: edchen@cs.sjtu.edu.cn (Erdong Chen),
ljiang@cs.sjtu.edu.cn (Linji Yang), hyuan@cs.sjtu.edu.cn (Hao Yuan).

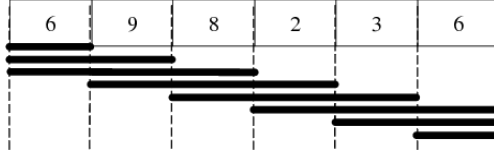


Fig. 1. Sliding windows of size 3 over the sequence 6, 9, 8, 2, 3, 6 (LISW problem)

1 Introduction

Given a sequence $\pi = \pi_1\pi_2 \dots \pi_n$ of n elements, the longest increasing subsequence (LIS) problem is to find a longest subsequence $\lambda = \pi_{i_1}\pi_{i_2} \dots \pi_{i_U}$ such that $1 \leq i_1 < i_2 < \dots < i_U \leq n$ and $\pi_{i_1} < \pi_{i_2} < \dots < \pi_{i_U}$. A longest increasing subsequence in a window $\pi\langle l, r \rangle = \pi_l\pi_{l+1} \dots \pi_r$ is defined to be a longest subsequence $\sigma = \pi_{i_1}\pi_{i_2} \dots \pi_{i_T}$ such that $l \leq i_1 < i_2 < \dots < i_T \leq r$ and $\pi_{i_1} < \pi_{i_2} < \dots < \pi_{i_T}$. The LISSET problem proposed in this paper, is to find a longest increasing subsequence (or all the longest increasing subsequences when needed¹) for every window in a set of variable-size windows. In other words, LISSET problem is to solve the LIS problem in a subsequence $\pi\langle l, r \rangle = \pi_l\pi_{l+1} \dots \pi_r$ for different pairs of indices l and r .

In [AGH⁺04], Albert et al. defined the LISW problem, which is to find the longest increasing subsequences in sliding windows over a sequence of n elements. A w -size window is a subsequence $\pi\langle i+1, i+w \rangle$ for some $0 \leq i \leq n-w$. Additionally, all the truncated windows $\pi\langle 1, j \rangle$ for $j < w$ and $\pi\langle j, n \rangle$ for $j > n-w+1$ are also regarded as w -size windows (see Fig. 1 for an example). Albert et al. proposed an algorithm to solve the LISW problem in $O(n \log \log n + \text{OUTPUT})$ time, where OUTPUT is the total size of the output. In this paper, we will give a faster algorithm for this problem, which runs in $O(\text{OUTPUT})$ time and uses $O(w)$ space. Our algorithm solves the LISW problem in optimal time, which is linear in the size of the output.

The remainder of this paper is organized as follows. In Section 2, some problems and techniques related to the LISSET problem are reviewed. In Section 3, the LISSET problem is defined, which takes the LISW problem as a subcase. In Section 4, the Canonical Antichain Partition is discussed, which is the basic data structure of our algorithm. In Section 5, we design and analyze a sweep algorithm for the LISSET problem and the LISW problem, and summarize our contributions. Finally, conclusions and future work are discussed in Section 6.

¹ We would like to thank the reviewer who suggested us to discuss how to output all the LISs in every window, which is actually an improvement of this article based on the conference version [CYY05].

2 Related Work

The longest increasing subsequences problem is a fundamental combinatorial problem which has been widely studied [Sch61,BS00,AAH93]. Knuth proposed an $O(n \log n)$ algorithm whose mechanism is to maintain the first row of Young tableau [Knu98]. Fredman proved an $\Omega(n \log n)$ lower bound under the decision tree model [Fre75]. However, an $O(n \log \log n)$ algorithm is possible by using a van Emde Boas tree [vEB77] on a permutation.

Liben-Nowell et al. [LNVZ03] proposed two algorithms for LIS problem in data streaming model [HRR98]. One is to decide whether the LIS of a given stream drawn from $\{1 \dots M\}$ has length at least k using $O(k \log M)$ space and update time $O(\min\{\log k, \log \log M\})$, and the other is a multi-pass data streaming algorithm to return the actual LIS itself using space $O(k^{1+\epsilon} \log M)$. If $n \gg w$ (average window size), then our algorithm is a data streaming algorithm which makes only a single pass over the input sequence with $O(w)$ space.

Longest Increasing Subsequence has been widely used in bioinformatics [SW81]. On the topic of sequence alignment, Zhang proposed a BLAST+LIS strategy to find the correct longest consecutive list of high scoring segment pairs (HSPs) in the BLAST output, if the BLAST output contains multiple HSPs for a pair of sequences, and hence reduced the redundant HSPs in each hit and filtered out the redundant genomic hits [Zha03].

3 Problem Definition

The longest increasing subsequence (LIS) in a window $W = \pi\langle l, r \rangle$ is a longest subsequence $\sigma = \pi_{i_1}\pi_{i_2}\dots\pi_{i_t}$ such that $l \leq i_1 < i_2 < \dots < i_t \leq r$ and $\pi_{i_1} < \pi_{i_2} < \dots < \pi_{i_t}$. Given a window W , let $\omega(W) = |\sigma|$ denote the length of the LIS in window W .

Let $S = \{W_i \mid W_i = \pi\langle l_i, r_i \rangle\}$ be a set of m variable-size windows. The LIS-SET problem is to calculate $\omega(W_i)$ and find a corresponding LIS in W_i for each i ($1 \leq i \leq m$).

4 Canonical Antichain Partition

Given a sequence $\pi = \pi_1\pi_2\dots\pi_n$, each element π_i can be represented by a point (i, π_i) in the plane. For example, the sequence 6, 7, 7, 2, 3, 6, 5, 4, 9 is represented by $p_1p_2\dots p_9$ (See Fig. 2). Let P be the planar point set of n

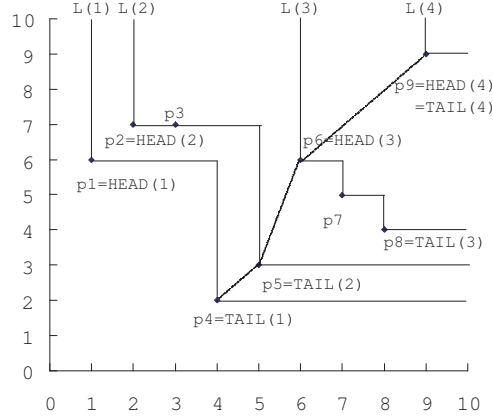


Fig. 2. The canonical antichain partition of 6, 7, 7, 2, 3, 6, 5, 4, 9, and a longest chain: $\langle (4, 2), (5, 3), (6, 6), (9, 9) \rangle$.

elements in the form of (i, π_i) . For any point $p \in P$, let p_x and p_y denote its x - and y -coordinate respectively. Following Felsner and Wernisch [FW98], for two points $p, q \in P$, the *dominance order* is given by the relation $p \prec q$ (say q *dominates* p) if $p_x < q_x$ and $p_y < q_y$. The *shadow* [FW98] of a point p is defined as the area of all points (u, v) dominating p , i.e. with $u > p_x$ and $v > p_y$. A chain $C \subseteq P$ is an ordered points list of the dominance order on P , i.e. $C = \langle p_1, p_2, \dots, p_t \rangle$ is a chain if and only if $p_j \prec p_{j+1}$ for $1 \leq j < t$. For instance, the chain $\langle p_4, p_5, p_6, p_9 \rangle$ is a *longest chain* in Fig. 2.

The *height* of a point $p \in P$, denoted by $h(p)$, is the size of a longest chain with p as the ending point, more specifically, $h(p) = \max\{|C| \mid C = \langle p_1, \dots, p \rangle\}$. Given a point $p = (i, \pi_i)$, it is clear that the height $h(p)$ is also the length of the longest increasing subsequence with π_i as the ending element. In addition, the points with the same height are not comparable by dominance order. Thus, all points with the same height in the same set yield a partition of P into *antichains*, the *canonical antichain partition*. All points with the same height h in P form an antichain L_h .

In our algorithm and the rest part of this paper, points in an antichain L_h are sorted in increasing order by x -coordinate and in non-increasing order by y -coordinate. Since all the points with height h are stored in linked list L_h , we define $\text{HEAD}(h)$ and $\text{TAIL}(h)$ to be the first and last point in L_h . Obviously, $\text{HEAD}(h)$ is the point with the smallest x -coordinate in L_h , and $\text{TAIL}(h)$ is the point with the largest x -coordinate in L_h . Given a point $p \in L_h$, the point following p in L_h is denoted by $\text{NEXT}(p)$. By definition, $\text{NEXT}(\text{TAIL}(h)) = \emptyset$ for any h . A point q is said to be a *preceding point* of p in L_i , if $h(q) = i$ and $q \prec p$.

Suppose the point p is in L_i , in order to return a chain with p as the ending point, the rightmost preceding point of p in L_j for each $1 \leq j < i$, denoted

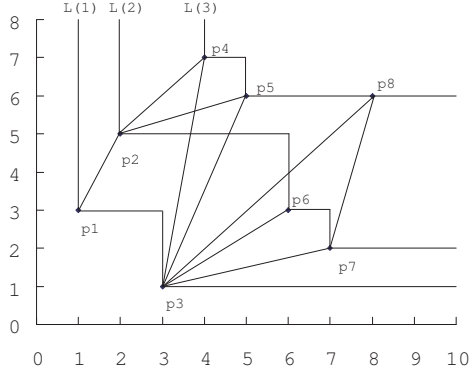


Fig. 3. $\text{PRED}(p_i, j)$ of each point p_i , pointed by solid lines. For instance $\text{PRED}(p_4, 2) = p_2$, $\text{PRED}(p_4, 1) = p_3$ and $\text{PRED}(p_2, 1) = p_1$.

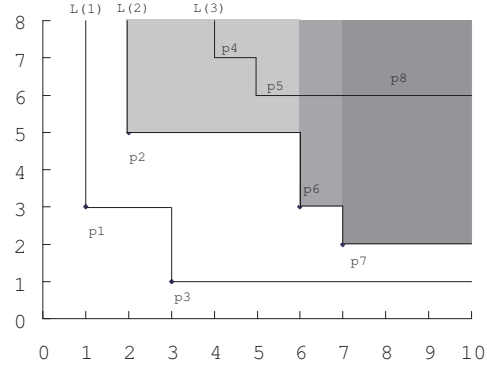


Fig. 4. The points covered by areas of different darkness appear in $\text{SUCC}(p_2)$, $\text{SUCC}(p_6)$ and $\text{SUCC}(p_7)$ respectively.

by $\text{PRED}(p, j)$, is defined as $\arg \max_q \{q_x | q \in L_j \text{ and } q \prec p\}$. By definition, we have for all $j \geq h(p)$, $\text{PRED}(p, j) = \emptyset$. We also define $\text{SUCC}(q)$ as a set of points: $p \in \text{SUCC}(q)$ if and only if $\text{PRED}(p, h(q)) = q$. The points in $\text{SUCC}(q)$ for each q are sorted according to p_x in strictly increasing order. Both PRED and SUCC are typically used to print out the LIS solution in each window efficiently, and the implementation will be discussed later. In Fig. 3 and Fig. 4, we give an example of PRED and SUCC . For the PRED we have $\text{PRED}(p_8, 2) = p_7$, $\text{PRED}(p_8, 1) = p_3$, $\text{PRED}(p_5, 2) = p_2$, $\text{PRED}(p_5, 1) = p_3$, and so on. For the SUCC , the lightest area in Fig. 4 covers points appearing in the set $\text{SUCC}(p_2)$, and hence $\text{SUCC}(p_2) = \{p_4, p_5\}$. The set $\text{SUCC}(p_3)$, whose illustration is not shown in Fig. 4 for simplicity, is equal to $\{p_4, p_5, p_6, p_7, p_8\}$.

In the implementation, the set $\text{PRED}(p, \cdot)$ for each p is organized as a doubly linked list, and the set $\text{SUCC}(q)$ for each q is organized as a linked list with an additional pointer field in each node. Suppose the point stored in some node is r , the pointer field of this node is pointing to the memory address of $\text{PRED}(r, h(q))$. In Fig. 5, we give an example of the data structure which is accord with the example shown in Fig. 3 and Fig. 4.

5 Sweep Algorithm

In order to output the longest increasing subsequence in a window, we have to design a data structure to maintain the structural information about all the LISs in a window. Our data structure needs to support the following operations: remove the first element of the sequence, insert an element to the end of the sequence, and output a longest increasing subsequence with constraints. Formally, if $\mathcal{W} = \pi\langle l, r \rangle$ is a subsequence, our data structure

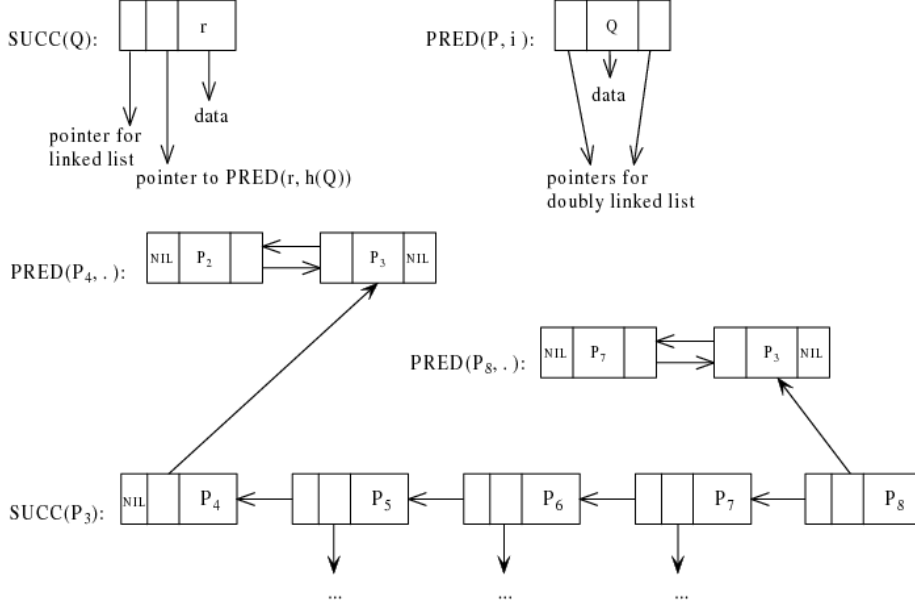


Fig. 5. Example of the data structure for PRED and SUCC

needs to support:

- (1) Insert π_{r+1} to \mathcal{W} ;
- (2) Remove π_l from \mathcal{W} ;
- (3) Output a longest increasing subsequence $\sigma = \pi_{i_1} \pi_{i_2} \dots \pi_{i_T}$ in \mathcal{W} satisfying $i_T \leq X_{\text{QRY}}$ for a parameter X_{QRY} , where $l \leq X_{\text{QRY}} \leq r$.

Instead of dealing with subsequences directly, our data structure maintains a canonical antichain partition in a window, for there is a mapping between a sequence and a point set in the plane. Let $P = \langle p_0, p_1, \dots, p_{r-l+1} \rangle$ be the ordered points set that represents \mathcal{W} , i.e. $p_i = (l + i, \pi_{l+i})$, for $0 \leq i \leq r - l$, and $\omega(P) = \omega(\mathcal{W})$ is the number of antichains in the canonical antichain partition. We design three operations on the point set P corresponding to the three operations on the subsequence \mathcal{W} :

- (1) Insert: insert into P a new point p_{INS} with a larger x -coordinate than that of any point in P
- (2) Delete: remove from P a point p_{DEL} with the smallest x -coordinate
- (3) Query(X_{QRY}): output a longest chain σ satisfying that the largest x -coordinate of points in σ is equal to or less than X_{QRY}

For any h , let L'_h be the new antichain with height h after an operation, and for any $p \in P$, the point $\text{PRED}'(p, j)$ is the new rightmost preceding point of p in L'_j after an operation. In the following, we will provide the details and complexity analysis for each operation.

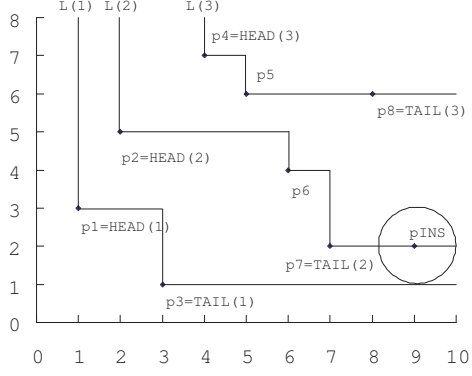


Fig. 6. Insert operation (Case 1)

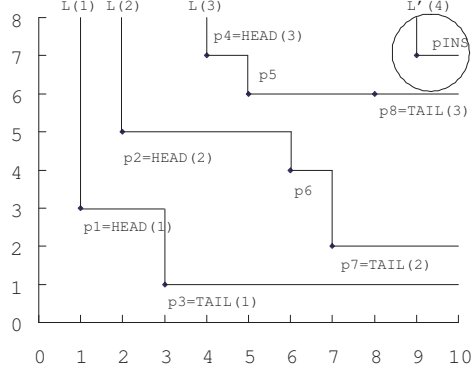


Fig. 7. Insert operation (Case 2)

5.0.1 Insert operation

The Insert operation is to insert a new point p_{INS} with a larger x -coordinate than that of any point in P . The main problem of the Insert operation is to find an antichain which p_{INS} belongs to. There are two cases (Fig. 6 and Fig. 7 show these two cases).

- (1) if $h(p_{\text{INS}}) = i$ ($1 \leq i \leq \omega(P)$), then insert p_{INS} to the end of L_i , i.e. $L'_i = L_i \cup \{p_{\text{INS}}\}$.
- (2) if $h(p_{\text{INS}}) = \omega(P) + 1$, then create a new antichain $L'_{\omega(P)+1} = \{p_{\text{INS}}\}$.

If $\langle a_1, a_2, \dots, a_t, p_{\text{INS}} \rangle$ is a longest chain, there is always another longest chain by replacing a_t by $\text{TAIL}(h(a_t))$. As in Fig. 8, we can always replace the solid line $\langle p_1, p_2, p_5, p_{\text{INS}} \rangle$ by the dashed line $\langle p_1, p_2, p_8, p_{\text{INS}} \rangle$. Therefore, to compute $h(p_{\text{INS}})$ is to check $\text{TAIL}(i-1)$ for $i = 2, \dots, \omega(P) + 1$, until we find the highest $\text{TAIL}(i-1)$ that p_{INS} dominates. More specifically, we do the following steps when inserting p_{INS} : (1) Construct the new doubly linked list $\text{PRED}'(p_{\text{INS}}, \cdot)$ in sequential order and set $\text{PRED}'(p_{\text{INS}}, j) = \text{TAIL}(j)$ for each $j < i$; (2) Insert a node storing p_{INS} at the end of the linked list $\text{SUCC}(\text{TAIL}(j))$ for each $j < i$; (3) If $i = \omega(P) + 1$, we create a new antichain $L'_{\omega(P)+1}$ with only a point p_{INS} , otherwise we insert the point p_{INS} at the end of L_i . Our Insert operation adopts linear search algorithm, so that we can insert each node storing $\text{TAIL}(j)$ into the doubly linked list $\text{PRED}'(p_{\text{INS}}, \cdot)$ one by one, and find the antichain which p_{INS} belongs in sequential order. The total cost of the Insert operation is $O(h(p_{\text{INS}}))$, which equals to the total cost of the Delete operation according to the analysis in Section 5.1.

Theorem 1 *The cost of Insert operation equals $O(h(p_{\text{INS}}))$.*

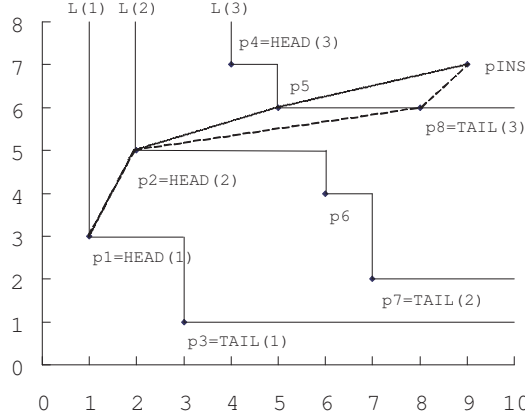


Fig. 8. Insert operation

5.0.2 Delete operation

The Delete operation is to delete the point p_{DEL} with the smallest x -coordinate in P . After deleting p , the heights of some points may decrease. For any $p \in P$, the new height $h'(p)$ is defined to be the height of p after deleting p_{DEL} . Let L'_i be the new antichain with height i after the operation, i.e., $L'_i = \{p \mid h'(p) = i\}$. Let $D_i = \{p \mid h(p) = i \text{ and } h'(p) = h(p) - 1\}$, so we have $D_1 = \{p_{\text{DEL}}\}$ and $D_{\omega(P)+1} = \emptyset$. Since D_i is a set of consecutive points including the point with the smallest x -coordinate in L_i , we have $D_i = \{\text{HEAD}(i), \text{NEXT}(\text{HEAD}(i)), \dots, \text{PMAX}(i)\}$, where $\text{PMAX}(i)$ is defined to be the point with the largest x -coordinate in D_i (if $D_i = \emptyset$, then $\text{PMAX}(i) = \emptyset$).

We will illustrate our method based on the example in Fig. 9. By definition, we have $D_1 = \{p_1\}$, $\text{PMAX}(1) = p_1$, and $L_1 \setminus D_1 = \{p_3\}$. The points in the shadow area of Fig. 11 dominate p_1 , while the points in the shadow area of Fig. 12 dominate p_3 . In Fig. 13, the points in the deep shadow area dominate p_1 but not p_3 . Therefore, the height of p_2 should decrease by one, while the heights of the points in L_2 in the light shadow area do not change, i.e., $D_2 = \{p_2\}$ and $L_2 \setminus D_2 = \{p_6, p_7\}$. Also, the point p_2 is the point with the largest x -coordinate that does not dominate $\text{NEXT}(\text{PMAX}(1)) = p_3$. In Fig. 14, the points in the deep shadow area dominate the points in D_2 but not any point in $L_2 \setminus D_2$. Therefore, the heights of p_4 and p_5 should decrease by one, while the heights of the points in L_3 in the light shadow area do not change, i.e., $D_3 = \{p_4, p_5\}$ and $L_3 \setminus D_3 = \{p_8\}$. Also, the point p_5 is the point with the largest x -coordinate that does not dominate $\text{NEXT}(\text{PMAX}(2)) = p_6$, so $\text{PMAX}(3) = p_5$. Finally, we have $L'_1 = \{p_2, p_3\}$, $L'_2 = \{p_4, p_5, p_6, p_7\}$, and $L'_3 = \{p_8\}$. In Fig. 10, the dashed line represents removing the D_i , and the broad-brush line represents the concatenation of D_{i+1} and $L_i \setminus D_i$, which forms L'_i .

The following observations are important for the correctness of the algorithm, and the proofs of these lemmas can be found in the appendix.

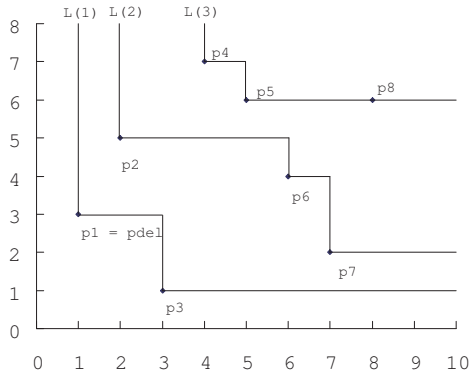


Fig. 9. Before the Delete operation

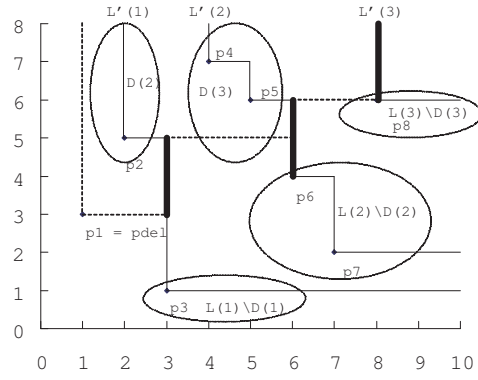


Fig. 10. After the Delete operation

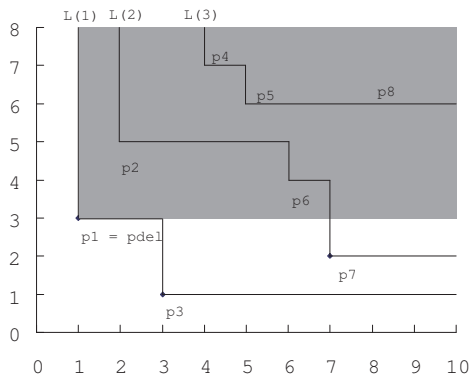


Fig. 11. The points in the shadow area dominate p_1

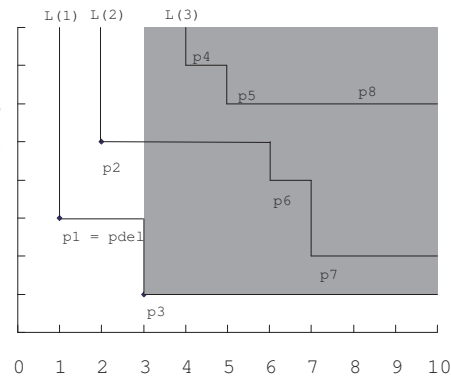


Fig. 12. The points in the shadow area dominate p_3

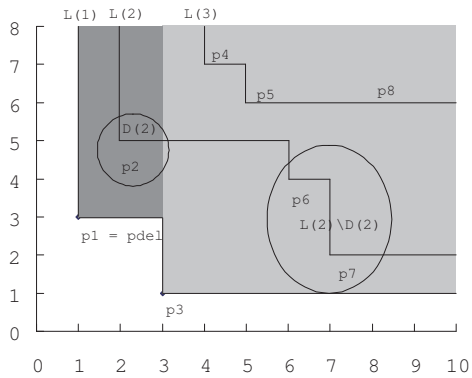


Fig. 13. For those points in L_2 , the heights of the points in darker shadow decrease by one, and the heights of the points in lighter shadow do not change

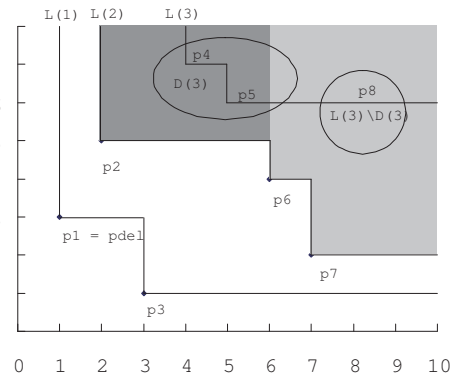


Fig. 14. For those points in L_3 , the heights of the points in darker shadow decrease by one, and the heights of the points in lighter shadow do not change

Lemma 2 *The height of each point may decrease at most by one.*

Lemma 3 *For any $i > 1$, the height of a point in L_i decreases if and only if it does not dominate any point in $L_{i-1} \setminus D_{i-1}$ or $L_{i-1} \setminus D_{i-1}$ is empty.*

Lemma 4 *If $D_i = \emptyset$, then $D_j = \emptyset$ for all $j \geq i$; if $L_i \setminus D_i = \emptyset$, then $L_j \setminus D_j = \emptyset$ for all $j \geq i$.*

Lemma 5 *If $D_i \neq \emptyset$, the point set D_i is a set of consecutive points including the point with the smallest x -coordinate in L_i .*

Lemma 6 *If $D_{i+1} \neq \emptyset$ and $L_i \setminus D_i \neq \emptyset$, the x -coordinate of any point in D_{i+1} is smaller than that of any point in $L_i \setminus D_i$.*

By Lemma 5, the set D_i can be calculated by computing the point with the largest x -coordinate in D_i , i.e. $\text{PMAX}(i)$. By definition, $\text{PMAX}(1) = \{p_{\text{DEL}}\}$. For $i > 1$, if $\text{PMAX}(i-1) = \text{TAIL}(i-1)$, then $\text{PMAX}(i) = \text{TAIL}(i)$. Otherwise, by Lemma 3, to compute $\text{PMAX}(i)$, our method scans the antichain L_i from $\text{HEAD}(i)$ to the last point $q \in L_i$ that does not dominate the point $\text{NEXT}(\text{PMAX}(i-1))$. The next step is to compute the new antichain partition L'_i after the Delete operation. By Lemma 6, for a fixed i , our method moves D_{i+1} to the head part of $L_i \setminus D_i$ in $O(1)$ time without destroying the relative orders of the points in D_{i+1} and $L_i \setminus D_i$ (Fig. 10 shows the concatenation of D_{i+1} and $L_i \setminus D_i$ for $i = 1, 2, 3$). Let \mathcal{D} be the union of D_i , i.e. $\mathcal{D} = \bigcup_{i=1}^{\omega(P)} D_i$. Thus, the time complexity of maintaining the data structure of the antichains after deleting p_{DEL} is $O(|\mathcal{D}|)$.

In order to efficiently print out the solution, we need the following lemmas to analyze the maintenance of the PRED and SUCC , so that during one Delete operation, the time complexity of maintaining the PRED and SUCC is $O(|\mathcal{D}|)$.

Lemma 7 *Suppose $i \neq 1$ and $p \in L_i$, then $p \in D_i$ if and only if there exists j ($1 \leq j < i$), s.t. for all k with $j \leq k < h(p)$, $\text{PRED}'(p, k) = \text{PRED}(p, k+1)$, and for all k with $1 \leq k < j$, $\text{PRED}'(p, k) = \text{PRED}(p, k)$.*

By Lemma 7 and the definition of $\text{SUCC}(q)$, if any point p is deleted from $\text{SUCC}(q)$ as a result of the Delete operation, then q equals $\text{PRED}(p, h(q))$, and q becomes no longer the rightmost preceding point of p in $L'_{h'(q)}$, where $h'(q) = h(q) - 1$.

Lemma 8 *Suppose $q \in D_i$, if the point a ($\in \text{SUCC}(q)$) should be deleted from $\text{SUCC}(q)$, then for every point b with $b_x > a_x$, b should be deleted from $\text{SUCC}(q)$.*

During the Delete operation, when the height of point q decreases, we scan $\text{SUCC}(q)$ from the end of the linked list to the head of it, and check whether

the point p stored in the last node of linked list has the property: p dominates $\text{PRED}(p, h(q) - 1)$ and the height of $\text{PRED}(p, h(q) - 1)$ does not decrease. This can be done in $O(1)$ time because this node has the pointer to the node storing $\text{PRED}(p, h(q))$. If so, we delete this node from the end of $\text{SUCC}(q)$, link the node storing $\text{PRED}(p, h(q) + 1)$ directly to the node storing $\text{PRED}(p, h(q) - 1)$, and continue to check the last node of $\text{SUCC}(q)$; otherwise, we stop checking $\text{SUCC}(q)$ according to Lemma 8. So for each p whose height remains unchanged, the $\text{PRED}(p, \cdot)$ list will not change during this Delete operation by Lemma 7; also by Lemma 7, for each p whose height decreases by one, there is only one related point q whose height also decreases and causes p deleted from $\text{SUCC}(q)$. So the time complexity of total operations on SUCC and PRED during this Delete operation is $O(|\mathcal{D}|)$. Actually, by Lemma 6, we can further conclude that $\text{SUCC}(q)$ shortens only if $q = \text{PMAX}(i)$ for some i . This procedure is not shown in the following pseudo-code for simplicity.

Theorem 9 *The cost of one Delete operation equals the total number of points whose height decreases, i.e. $O(|\mathcal{D}|)$.*

Algorithm 1 Algorithm for Delete operation

```

initialize  $\text{PMAX}(1) \leftarrow p_{\text{DEL}}, D_1 \leftarrow \{p_{\text{DEL}}\}, D_i \leftarrow \emptyset$  for  $i > 1$ , and  $L'_i \leftarrow L_i$ 
for  $i > 1$ 
  for  $i = 2$  to  $\omega(P)$  do
    if  $\text{PMAX}(i - 1) = \text{TAIL}(i - 1)$  then
       $\text{PMAX}(i) \leftarrow \text{TAIL}(i)$ 
    else
       $\text{PMAX}(i) \leftarrow \arg \max_p \{p_x \mid p \in L_i \text{ and } \neg(\text{NEXT}(\text{PMAX}(i - 1)) \prec p)\}$ 
    end if
    if  $\text{PMAX}(i) = \emptyset$  then
      EXIT LOOP
    end if
     $D_i = \{p \mid p \in L_i \text{ and } p_x \leq \text{PMAX}(i)_x\}$ 
  end for
  for  $i = 1$  to  $\omega(P)$  do
    if  $\text{PMAX}(i) = \emptyset$  then
      EXIT LOOP
    end if
     $L'_i \leftarrow D_{i+1} + (L_i \setminus D_i)$ 
  end for

```

5.0.3 Query operation

Suppose $\sigma = \langle p_1, p_2, \dots, p_t \rangle$ is a longest chain in P that the x -coordinate of p_t is equal to or less than X_{QRY} . The reason why our method adopts the parameter X_{QRY} is to handle the *contain* situation of our algorithm in Section 5.1. Let P' be the point set in P whose x -coordinate is equal to or less than

X_{QRY} , i.e. $P' = \{p \mid p \in P \text{ and } p_x \leq X_{\text{QRY}}\}$. Let H be the length of a longest chain in P' , which is the size of $\text{Query}(X_{\text{QRY}})$'s output.

We will illustrate our method based on the example in Fig. 15. The sequence π is 3, 5, 1, 7, 6, 4, 2, 6, and the current sliding window \mathscr{W} is $\pi\langle 1, 8 \rangle$ (the current point set $P = \{p_1, p_2, \dots, p_8\}$). Our algorithm needs to output the longest increasing subsequence in the window $\pi\langle 1, 6 \rangle$, even though the current sliding window is $\pi\langle 1, 8 \rangle$. The Query operation is required to output the longest chain $\sigma = \langle p_1, p_2, \dots, p_u \rangle$ satisfying that the x -coordinate of p_u is equal to or less than 6, which is also to find the longest chain in the points set $P' = \{p_1, p_2, \dots, p_6\}$. There are two longest chains in the point set P' , $\langle p_1, p_2, p_4 \rangle$ and $\langle p_1, p_2, p_5 \rangle$.

Since the x -coordinate of $\text{HEAD}(H)$ is the smallest among that of all the points in L_H , if there is a longest chain in P' with any point $p \in L_H$ as the ending point, there will always be another longest chain in P' with $\text{HEAD}(H)$ as the ending point. As in Fig. 15, our method can always replace the solid line $\langle p_1, p_2, p_5 \rangle$ by the dashed line $\langle p_1, p_2, p_4 \rangle$.

Therefore, to compute H is to check $\text{HEAD}(i)$ for $i = 1, 2, \dots, \omega(P)$, until we find the highest $\text{HEAD}(i)$, the x -coordinate of which is equal to or less than X_{QRY} , i.e. $H = \max\{i \mid \text{HEAD}(i)_x \leq X_{\text{QRY}}\}$. We can find a longest chain C in P' satisfying that $C = \langle c_1, c_2, \dots, c_H \rangle$, $c_H = \text{HEAD}(H)$ and $c_i = \text{PRED}(c_{i+1}, i)$ for $i = 1, 2, \dots, H - 1$. By careful analysis, computing H requires $H + 1$ steps of searching, and outputting the sequence requires H steps. In short, the total time complexity of $\text{Query}(X_{\text{QRY}})$ is $O(H)$. Furthermore, by enumerating all points s in the highest antichain L_h to the left of X_{QRY} , all the feasible points q in L_{h-1} satisfying both $q_x \leq \text{PRED}(s, h-1)_x$ and $q_y < s_y$, setting $s = q$ and doing this recursively from L_h down to L_1 , we can print out all feasible LIS solutions within the current window \mathscr{W} .

Theorem 10 *The cost of outputting a longest chain σ satisfying that the largest x -coordinate of points in σ is equal to or less than X_{QRY} , is the length of σ i.e. $O(|\sigma|)$.*

5.1 Algorithm and complexity analysis

Our algorithm is based on the data structures proposed above. Firstly, windows are sorted in increasing order, and then we slide the window \mathscr{W} from left to right to output the LIS in each window. In order to make sure that points are inserted into \mathscr{W} only once, the windows are ordered by their left endpoints (if two windows share the same left endpoint, the longer window comes first), i.e. for two windows W_i and W_j , $W_i < W_j$ if and only if $l_i <$

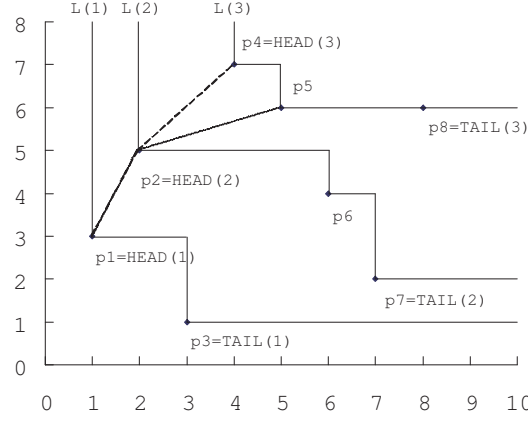


Fig. 15. Query operation

l_j or $(l_i = l_j \text{ and } r_i > r_j)$. Since the endpoints are drawn from $\{1, 2, \dots, n\}$, the preprocessing sorting can be completed in $O(n + m)$ time by radix sort.

Two distinct windows W_a and W_b *intersect* if there exists a number c such that $l_a \leq c \leq r_a$ and $l_b \leq c \leq r_b$; W_a and W_b are *disjointed* if they do not *intersect*; W_a *contains* W_b if $l_a \leq l_b \leq r_b \leq r_a$; W_a and W_b *overlap*, if W_a *intersects* W_b but neither one of them *contains* the other.

Before analyzing the complexity, depth_i is defined to be the largest height that π_i achieved in the m windows. Among all increasing subsequences in the m windows, depth_i is the length of the longest one with π_i as the ending element. For further illustrating the algorithm, let's consider a simple example: there are three windows W_1 , W_2 and W_3 . The relations among them are W_1 contains W_2 and W_3 ; W_2 and W_3 are disjointed; W_2 precedes W_3 . They are sorted as W_1, W_2, W_3 . At first, the algorithm inserts all the points in W_1 , and does a $\text{Query}(r_1)$; secondly, the algorithm deletes points between l_1 and $l_2 - 1$, and does a $\text{Query}(r_2)$; finally, the algorithm deletes points between l_2 and $l_3 - 1$, and does a $\text{Query}(r_3)$. Suppose there is another window W_4 with $r_3 < l_4 < r_1$ and $r_4 > r_1$, the algorithm will insert points between r_1 and r_4 , delete points between l_3 and $l_4 - 1$, and do a $\text{Query}(r_4)$. The positions of pointers h and e in algorithm 2 are monotonically increasing, so no point will be re-inserted after it is deleted.

5.1.1 Complexity Analysis for the LISSET Problem

Theorem 11 (LISSET Problem) *The algorithm described above computes the m longest increasing subsequences, one for each window, in total time $O(n + \text{OUTPUT} + \sum_{i=1}^n \text{depth}_i)$ and total space $O(\sum_{i=1}^n \text{depth}_i)$.*

Algorithm 2 Algorithm for the LISSET Problem

```
initialize  $\mathcal{W} \leftarrow \emptyset$ 
Insert into  $\mathcal{W}$  the elements in  $\pi\langle l_1, r_1 \rangle$  separately
SET the left end of  $\mathcal{W}$ :  $h = l_1$ , the right end of  $\mathcal{W}$ :  $e = r_1$ 
Query( $r_1$ ) to output the LIS in  $\mathcal{W}$ 
for  $j = 2$  to  $m$  do
  if ( $l_j > e$ ) then {DISJOINT}
    reset  $\mathcal{W} \leftarrow \emptyset$ 
    Insert into  $\mathcal{W}$  the elements in  $\pi\langle l_j, r_j \rangle$  separately
    SET  $h = l_j$ ,  $e = r_j$ 
  else if ( $l_j \leq e$  and  $r_j > e$ ) then {OVERLAP}
    Delete from  $\mathcal{W}$  the elements in  $\pi\langle h, l_j - 1 \rangle$  separately
    Insert into  $\mathcal{W}$  the elements in  $\pi\langle e + 1, r_j \rangle$  separately
    SET  $h = l_j$ ,  $e = r_j$ 
  else if ( $h \leq l_j$  and  $r_j \leq e$ ) then {CONTAIN}
    if ( $l_j = h$ ) then {SAME LEFT ENDPOINT}
      NO OPERATION
    else if  $l_j > h$  then {DIFFERENT LEFT ENDPOINTS}
      Delete from  $\mathcal{W}$  the elements in  $\pi\langle h, l_j - 1 \rangle$  separately
      SET  $h = l_j$ 
    end if
  end if
  Query( $r_j$ ) to output a LIS  $\sigma = \pi_{i_1}\pi_{i_2}\dots\pi_{i_T}$  in  $\mathcal{W}$  satisfying  $i_T \leq r_j$ 
end for
```

PROOF. Most of the time required comes from three operations: Insert, Delete, Query. By Theorem 10, the cost of the Query operation equals the total length of the LISs in all m windows, so $T_{Query} = O(\text{OUTPUT})$. By Theorem 1, the cost of inserting π_i is equal to the length of the longest LIS with π_i as the ending element in the m windows, i.e. $T_{INS} = O(\sum_{i=1}^n \text{depth}_i)$. It is difficult to analyze the cost of each Delete operation, but we can calculate the cumulative cost of all the Delete operations. By Theorem 9, the cost is equal to the sum of the number of points which decrease after each operation, and a point $p = (i, \pi_i)$ may decrease at most depth_i times. Therefore, $T_{DEL} = O(\sum_{i=1}^n \text{depth}_i)$. Thus, $T = T_{Query} + T_{INS} + T_{DEL} = O(n + \text{OUTPUT} + \sum_{i=1}^n \text{depth}_i)$. For the worst case space complexity, the total number of space PRED uses is equal to that SUCC uses, and equals to $\sum_{i=1}^n \text{depth}_i$. Hence, the total space used is $O(\sum_{i=1}^n \text{depth}_i)$. \square

The LISSET problem has a straightforward approach, which is to find all LISs for each window separately. In the worst case that all m windows are disjointed, our algorithm does not give any asymptotic improvement over the straightforward method. However, similar to the analysis of Albert et al., our algorithm gives a better performance in average cases. Albert et al. proved that

the expected length of LIS in a window of size w is asymptotically $2\sqrt{w}$, therefore, if average window size is $O(w)$, our time complexity is $O(n + \text{OUTPUT} + \sum_{i=1}^n \text{depth}_i) = O(n + \text{OUTPUT} + n\sqrt{w}) = O((n+m)\sqrt{w})$. If $m = O(n)$, then our algorithm would certainly perform better than the straightforward approach whose complexity is $O(\min\{mw \log w, mw \log \log n\})$.

5.1.2 Complexity Analysis for the LISW Problem

The algorithm for the LISSET problem can be effectively used to solve the LISW problem directly, as theorem 12 below states.

Theorem 12 (LISW Problem) *Our algorithm finds the longest increasing subsequence in a sliding window over a sequence of n elements in $O(\text{OUTPUT})$ time and $O(\sum_{j=1}^n \text{depth}_j)$ space.*

PROOF. By Theorem 11, the time complexity is $O(n + \text{OUTPUT} + \sum_{j=1}^n \text{depth}_j)$. By definition, for $1 \leq i < w$, depth_i is equal to or less than the length of the LIS in the window $\pi\langle 1, i \rangle$. For $w \leq i \leq n$, depth_i is equal to or less than the length of the LIS in the window $\pi\langle i - w + 1, i \rangle$. Therefore, $\sum_{i=1}^n \text{depth}_i = O(\sum_{i=1}^m \omega(W_i)) = O(\text{OUTPUT})$. In short, our time complexity is $O(n + \text{OUTPUT} + \sum_{j=1}^n \text{depth}_j) = O(n + \text{OUTPUT}) = O(\text{OUTPUT})$, and the space complexity is $O(\sum_{j=1}^n \text{depth}_j)$ according to Theorem 11. \square

However, regarding the space complexity, some data structures in the algorithm for the LISSET problem is a waste of memory space for the LISW problem, mainly because the PRED and SUCC used in that algorithm to maintain the dependencies among points and effectively print out the solution is unnecessary for the LISW problem. The main difference between the output procedures of the LISSET problem and the LISW problem is, we only need to maintain one longest chain in current window for the LISW problem while we need to maintain almost every longest chain in current window for the LISSET problem, because the windows in the LISSET problem might *contain* other windows.

Therefore, for the LISW problem, we redefine PRED(p) to be the right most preceding point of p in $L_{h(p)-1}$. Now based on algorithm 1 and 2, we present an algorithm to dynamically maintain only one longest chain while window sliding by using $O(w)$ space in total. Initially, for $1 \leq i < w$, the windows are in the form $\pi\langle 1, i \rangle$, and points are inserted into current window one by one. So PRED(p) can be calculated accordingly, and it is easy to find a longest chain in current window from HEAD(H) by referring to PRED recursively. For $w \leq i \leq n$, the windows are in the form $\pi\langle i - w + 1, i \rangle$, and from now on, we do not use PRED any more. While current window slides, some point will

be deleted and some point will be inserted, then how to effectively maintain a longest chain C within current window? The following simple lemma helps.

Lemma 13 *For a longest chain $C = \langle p_1, p_2, \dots, p_t \rangle$, after the Delete operation, if the height of one point p_i decreases, the heights of all the points p_j ($j \leq i$) decrease; hence if the height of one point p_i remains unchanged, the heights of all the points p_j ($j \geq i$) remain unchanged.*

Suppose for a longest chain $C = \langle p_1, p_2, \dots, p_t \rangle$, after the Delete operation, the height of p_i is not changed, it is easy to verify that there is a new longest chain $C' = \langle p'_1, p'_2, \dots, p'_t \rangle$ satisfying: for all j with $j \geq i$, $p'_j = p_j$ and for all j with $j < i$, p'_j in L'_j after the Delete is a point in L_j to the right of p_j before the Delete. Note that, j is the height of p_j before the Delete. If there is no such i , the height of p_t decreases, and there are two cases to analyze. (1) There are still some point(s) in L'_t . Suppose the left most one is q_t , it is also easy to verify that there is a new longest chain $C' = \langle p'_1, p'_2, \dots, p'_t \rangle$ satisfying: $p'_t = q_t$ and for all $j < t$, p'_j in L'_j after the Delete is a point in L_j to the right of p_j before the Delete. For example in Fig. 10, suppose before the Delete p_1 , one longest chain is $C = \langle p_1, p_2, p_4 \rangle$, then after the Delete p_1 , it becomes $C' = \langle p_3, p_6, p_8 \rangle$. (2) There is no point in L'_t , then the new longest chain is $C' = \langle p'_1, p'_2, \dots, p'_{t-1} \rangle = \langle p_2, p_3, \dots, p_t \rangle$. In the Insert operation, the situation is nearly the same as case (1).

The total cost of finding i is $O(\text{OUTPUT})$ obviously. If there is some p_i , whose height is not changed, the searching strategy for C' (when needed) is: (1) set $t = p_i, j = i - 1$; (2) start at p_j in L_j , seek a point q satisfying both $q \prec t$ and $q \in L'_j$, set $p'_j = t = q, j = j - 1$, and doing (2) repeatedly until $j = 0$. The searching strategy for finding C' in the case (1) on the condition that there is no such i and for finding C' in the Insert operation is the same as the discussed one.

By the analysis above, we can search the new longest chain from L_i down to L_1 , and the cost for this searching is the summation over the number of points between p_j and p'_j in L_j for all $j < i$. It is not hard to see, by Lemma 5 and Lemma 13, for any point between p_j and p'_j in one Delete or Insert operation, it has no second chance to be scanned by this search algorithm during the remaining Delete and Insert operations. Therefore, the time complexity is $O(n + \text{OUTPUT}) = O(\text{OUTPUT})$, and the space complexity is reduced from $O(\sum_{i=1}^n \text{depth}_i)$ to $O(w)$ in the worst case.

Theorem 14 (LISW Problem) *The algorithm finds the longest increasing subsequence in a sliding window over a sequence of n elements in $O(\text{OUTPUT})$ time, and $O(w)$ space.*

6 Concluding Remarks

We investigate the problem of finding the longest increasing subsequences in a set of variable-size sliding windows over a given sequence. By maintaining a canonical antichain partition, we propose an approach that solve the problems in time $O(n + \text{OUTPUT} + \sum_{i=1}^n \text{depth}_i)$ for m windows over a sequence of n elements. This algorithm is able to solve the problem significantly better than straightforward methods. Since the LISW problem is a subcase of the LISSET problem, our algorithm solves LISW problem in time $O(\text{OUTPUT})$ and uses $O(w)$ space in total, while the time complexity of the best solution for the LISW problem previously achieved is $O(\text{OUTPUT} + n \log \log n)$.

Some problems related to the LISSET problem are still open problems. Firstly, the time complexity of finding *global maximal* LIS among all LISs in all windows is particularly interesting. Because our algorithm only finds the global maximal LIS after computing all LISs in all m window individually, it would be useful to design an algorithm to find global maximal LIS directly in the future. The second problem is to design an efficient online algorithm to output the LIS in a given window with size w in linear time $O(w)$. Our algorithm for the LISSET problem requires $O(n^2)$ time and $O(n^2)$ space for preprocessing to solve this problem. Because we can solve the LISSET problem on the set $S = \{W_i \mid W_i = \pi\langle i, n \rangle\}$, and store the state of the data structure for each window W_i when deleting the point p_{i-1} from the sliding window. If the given query window is $\pi\langle i, j \rangle$, we simply refer to the state of the data structure for the window W_i , do a $\text{Query}(j)$ and by referring to PRED recursively print out a solution in linear time. Note that, when storing the state of the data structure for the window W_i , we need not to keep the whole doubly linked list $\text{PRED}(p_k, \cdot)$ for each $k \geq i$, because the information in $\text{PRED}(p_k, h(p_k) - 1)$ is enough for printing the solution.

Acknowledgements

Thanks to Hong Zhu, Binhai Zhu, Qiqi Yan, and the reviewers for their helpful comments and discussions. Thanks to Prof. Yong Yu¹ for his support on this work. Thanks to Yuan Hang Li and Lee-Ping Wang for polishing the English in this article.

¹ The advisor of the undergraduate program in Department of Computer Science and Engineering, Shanghai Jiao Tong University.

References

- [AAH93] Alberto Apostolico, Mikhail J. Atallah, and Susanne E. Hambruch. New clique and independent set algorithms for circle graphs (discrete applied mathematics 36 (1992) 1-24). *Discrete Applied Mathematics*, 41(2):179–180, 1993.
- [AGH⁺04] Michael H. Albert, Alexander Golynski, Angèle M. Hamel, Alejandro López-Ortiz, S. Srinivasa Rao, and Mohammad Ali Safari. Longest increasing subsequences in sliding windows. *Theor. Comput. Sci.*, 321(2-3):405–414, 2004.
- [BS00] Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Inf. Process. Lett.*, 76(1-2):7–11, 2000.
- [CYY05] Erdong Chen, Hao Yuan, and Linji Yang. Longest increasing subsequences in windows based on canonical antichain partition. In *ISAAC*, pages 1153–1162, 2005.
- [Fre75] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.
- [FW98] Stefan Felsner and Lorenz Wernisch. Maximum k-chains in planar point sets: Combinatorial structure and algorithms. *SIAM J. Comput.*, 28(1):192–209, 1998.
- [HRR98] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report 1998-011, Digital Equipment Corporation, Systems Research Center, May 1998.
- [Knu98] D. E. Knuth. *The Art of Computer Programming. Vol 3, Sorting and Searching. Second Edition*. Addison-Wesley, 1998.
- [LNVZ03] David Liben-Nowell, Erik Vee, and An Zhu. Finding longest increasing and common subsequences in streaming data. Technical Report MIT-LCS-931, Cambridge, MA 02139, November 2003.
- [Sch61] C. Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 1961.
- [SW81] T.F. Smith and M.S. Waterman. Comparison of biosequences. *Adv. in Appl. Math.*, 2:482–489, 1981.
- [vEB77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.
- [Zha03] Hongyu Zhang. Alignment of blast high-scoring segment pairs based on the longest increasing subsequence algorithm. *Bioinformatics*, 19(11):1391–1396, 2003.

A Proofs

A.1 Proof of Lemma 5

PROOF. For the case $D_i \neq \emptyset$, we prove this statement by induction on the index i . At first, for $i = 1$, p_{DEL} is the first element of L_1 and other points in L_1 are not comparable with each other, so their heights will not decrease. Therefore, D_1 satisfies the statement. Then, suppose the statement is true when $i = k$, now we prove that it is also true when $i = k + 1$. Suppose that D_{k+1} does not satisfy the statement, then $\exists p \in L_{k+1}$, $h'(p) = h(p)$ and $\exists q \in L_{k+1}$, $h'(q) < h(q)$ with $p_x < q_x$. In another words, $\exists p' \in L_k$, $h'(p') = h(p')$ with $p'_x < p_x$, and there is no point q' in L_k satisfying both $q'_x < q_x$ and $h'(q') = h(q')$. Based on the definition of antichain, because of $q \in L_{k+1}$, there exists a point μ in L_k dominated by q . We prove there is no such a point μ supporting q by apagoge: If $\mu_x \leq p'_x$, then $q_y > \mu_y > p'_y$, which implies $h'(q) = h(q)$; Otherwise, because D_k is consecutive, the equality $h'(\mu) = h(\mu)$ holds, which also implies $h'(q) = h(q)$. This is a contradiction. \square

A.2 Proof of Lemma 6

PROOF. Suppose that there exist two points p, q satisfying $p \in D_{i+1}$, $q \in L_i \setminus D_i$ and $p_x > q_x$ (no two points share the same x -coordinate). Because the height of p decreases, p does not dominate q , i.e. $p_y \leq q_y$. Before deleting p_{DEL} , there exists a point v in D_i s.t. $v \prec p$. Based on Lemma 5, we have $v_x < q_x$ and $v_y \geq q_y$. Therefore, $p_y > v_y \geq q_y$ which contradicts with $p_y \leq q_y$. Thus, no such p exists. \square

A.3 Proof of Lemma 7

PROOF. Before starting to prove the lemma, we first state a *continuous property* for this lemma: If there exists some k , s.t. $h(\text{PRED}(p, k))$ decreases, according to Lemma 5, for every l with $l \geq k$, $h(\text{PRED}(p, l))$ decreases. Note that, If the heights of both $\text{PRED}(p, k)$ and $\text{PRED}(p, k - 1)$ decrease, then $\text{PRED}(p, k)$ is still the rightmost preceding point of p in L'_{k-1} and hence it becomes $\text{PRED}'(p, k - 1)$. If the height of $\text{PRED}(p, k)$ does not decrease, then $\text{PRED}(p, k)$ is still the rightmost preceding point of p in L'_k and hence it equals to $\text{PRED}'(p, k)$.

‘ \Leftarrow ’: Suppose $p \in L_i \setminus D_i$, then for every j with $1 \leq j < i$, the height of the rightmost preceding point of p in L_j (i.e. $h(\text{PRED}(p, j))$) will not decrease.

Otherwise, by the *continuous property*, $p \in D_i$. This is a contradiction.

‘ \implies ’: If $p \in D_i$, then $h(\text{PRED}(p, h(p) - 1))$ will decrease, and there are two cases to analyze. (1) If there exists $j > 1$ s.t. $h(\text{PRED}(p, j))$ decreases and $h(\text{PRED}(p, j-1))$ does not decrease, then $\text{PRED}(p, j)$ is no longer the rightmost preceding point of p in L'_{j-1} after the height decreases, and $\text{PRED}(p, j-1)$ will be that rightmost point. Also by the *continuous property*, for all k with $k \leq j-1$, $h(\text{PRED}(p, k))$ will not decrease. In conclusion, only for each k with $k \geq j$, $h(\text{PRED}(p, k))$ decreases, and only point $\text{PRED}(p, j)$ becomes no longer the rightmost preceding point of p in L_{j-1} . (2) If for each k with $k \geq 1$, $h(\text{PRED}(p, k))$ decreases, then it is obvious that the point $\text{PRED}(p, 1)$ is p_{DEL} in this Delete operation, and therefore $j = 1$. \square

A.4 Proof of Lemma 8

PROOF. The deletion of a from $\text{SUCC}(q)$ means that the decrease of $h(q)$ causes the point q no longer the rightmost preceding point of a in $L'_{h(q)-1}$, and the point instead of q is another one denoted v at the same antichain. For each point b satisfying $b_x > a_x > v_x > q_x$, we have $v \prec b$. Therefore for these points, the rightmost point they dominate respectively in $L'_{h(q)-1}$ is not q but some point to the right of v or v itself. \square