Longest Increasing Subsequences in Windows based on Canonical Antichain Partition

Erdong Chen, Hao Yuan, and Linji Yang

Department of Computer Science and Engineering,
Shanghai Jiao Tong University,
200030 Shanghai, P.R.China
{edchen,hyuan,ljyang}@cs.sjtu.edu.cn

Abstract. We consider the LISW problem, which is to find the longest increasing subsequences (LIS) in a sliding window of fixed-size w over a sequence $\pi_1\pi_2\ldots\pi_n$. Formally, it is to find a LIS for every window in a set $S_{\text{FIX}} = \{\pi\langle i+1, i+w\rangle \mid 0 \le i \le n-w\} \cup \{\pi\langle 1, i\rangle, \pi\langle n-i, n\rangle \mid i < w\}$, where a window $\pi\langle l, r\rangle$ is a subsequence $\pi_l\pi_{l+1}\ldots\pi_r$. 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 length of the $n+w-1$ longest increasing subsequences in those windows of S_{FIX}. In addition, we propose a more generalized problem called LISSET , which is to find the LIS for every window in a set S_{VAR} containing *variable-size* windows. By applying our algorithm, we provide an efficient solution for LISSET problem which is better than the straight forward generalization of classical LIS algorithms. An upper bound of our algorithm on LISSET is discussed.

1 Introduction

Given a sequence $\pi = \pi_1\pi_2\ldots\pi_n$ of n elements, the longest increasing subsequences (LIS) problem is to find a longest subsequence $\sigma = \pi_{i_1}\pi_{i_2}\ldots\pi_{i_T}$ such that $1 \le i_1 < i_2 < \cdots < i_T \le n$ and $\pi_{i_1} < \pi_{i_2} < \cdots < \pi_{i_T}$. The LISSET problem proposed in this paper, is to find the longest increasing subsequences in a set of variable-size windows, which is to solve the LIS problem in a subsequence $\pi\langle l, r\rangle = \pi_l\pi_{l+1}\ldots\pi_r$ for different pairs of indices l and r.

In [AGH$^+$04], Albert et al. defined the problem LISW , 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 $0 \le i \le n-w$. 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 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. Our algorithm solves LISW problem in optimal time, linear on the size of the output.

The remainder of this paper is organized as follows. In Section 2, some problems and techniques related to LISSET are reviewed. In Section 3, the LISSET

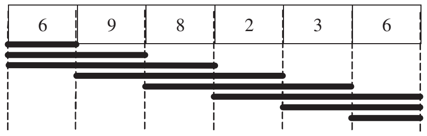

Fig. 1. Sliding windows of size 3 over sequence $6, 9, 8, 2, 3, 6$ (LISW problem)

problem is defined, which takes Lisw 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 give design and analysis of a sweep algorithm on Lisset problem and Lisw problem, and summarize our contributions. Finally, conclusions and future work are discussed in Section 6.

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 van Emde Boas tree [vEB77] on a permutation.

David et al.[DLNZ03] proposed two algorithms for LIS problem in data streaming model[MRHR98]. 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 $O(\min\{\log k, \log\log k\})$, and the other is a *multipass* data streaming algorithm to return the actual LIS itself using space $O(k^{1+\varepsilon} \log M)$. 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 alignment of sequences, 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; 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 $l \le i_1 < i_2 < \cdots < i_t \le r$ and $\pi_{i_1} < \pi_{i_2} < \cdots < \pi_{i_t}$. Given a window W, let $\omega(W) = |\sigma|$ denote the length of such 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 LISSET problem is to calculate $\omega(W_i)$ and find out a corresponding LIS in W_i for each $i(1 <= i <= m)$.

4 Canonical Antichain Partition

Given a sequence $\pi = \pi_1\pi_2 \dots \pi_n$, each element π_i may be represented by a point (i, π_i) in the plane. For example, the sequence $6, 9, 8, 2, 3, 5, 1, 4, 7,$ is represented by $p_1p_2 \dots p_9$(See Fig. 2). Let P be the planar point set of n 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-coordinates. 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.

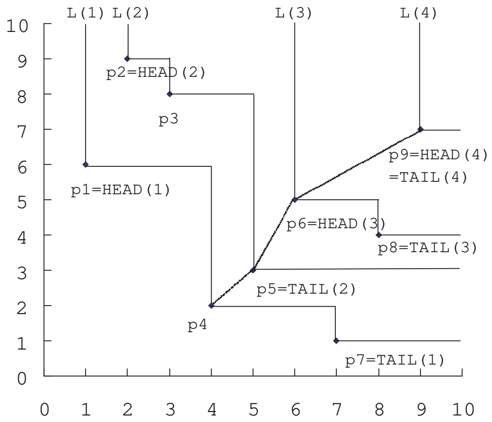

Fig. 2. The canonical antichain partition of $6, 9, 8, 2, 3, 5, 1, 4, 7,$ and a longest chain: $\langle(4, 2), (5, 3), (6, 5), (9, 7)\rangle$.

The *height* of a point $p \in P$, denoted $h(p)$, is the length of a longest chain with p as maximal point. Hence, given a point $p_i = (i, \pi_i)$, the height $h(p_i)$ is also the length of the longest increasing subsequence with π_i as the maximal element[FW98]. 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. Theorem 1 is a well-known result about antichains[FW98], which is a key issue in designing our algorithm.

Theorem 1. *All points with the same height h in P form an antichain L_h. 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 respectively. Obviously, $\text{HEAD}(h)$ is the point with the smallest x-coordinate in antichain L_h, and $\text{TAIL}(h)$ is the point with the largest x-coordinate in antichain 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, if $h(q) = h(p) - 1$ and $q \prec p$. In order to return a chain with p as the maximal point, the rightmost preceding point of p, denoted PRED(p), is defined to be the one with the largest x-coordinate among all p's preceding points. By definition, if $h(p) = 1$, then we have $\text{PRED}(p) = \emptyset$.

5 Sweep Algorithm

In order to output the longest increasing subsequence in a window, we have to design a data structure to maintain structural information about all LIS's 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, output a longest increasing subsequence with constraints. Formally, if $\mathscr{W} = \pi\langle l, r\rangle$ is a subsequence, our data structure needs to support:

1. remove π_l from $\mathscr{W}$;
2. insert π_{r+1} to $\mathscr{W}$;
3. output a longest increasing subsequence $\sigma = \pi_{i_1}\pi_{i_2}\dots\pi_{i_T}$ in $\mathscr{W}$ satisfying $i_T \leq X_{\text{QRY}}$ for a fixed X_{QRY} between l and r.

Instead of dealing 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 $P = [p_0, p_1, \dots, p_{r-l+1}]$ be the ordered points set that represents $\mathscr{W}$, i.e. $p_i = (l + i, \pi_{l+i})$, for $0 \leq i \leq r - l$, and $\omega(P) = \omega(\mathscr{W})$ 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 $\mathscr{W}$:

1. DELETE: remove from P a point p_{DEL} with the smallest x-coordinate
2. INSERT: insert into P a new point p_{INS} with a larger x-coordinate than that of any point in P
3. $\text{QUERY}(X_{\text{QRY}})$: 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 $\text{PRED}'(p)$ is the new rightmost preceding point of p after an operation. In the following, we will provide the details and complexity analysis for each operation.

DELETE operation The DELETE operation is to delete the point p_{DEL} with the smallest x-coordinate in P. After deleting p, the height 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 $D_i = \{p|h(p) = i \text{ and } h'(p) = h(p) - 1\}$, so we have $D_1 = \{p_{\text{DEL}}\}$ and $D_{\omega(\mathscr{W})+1} = \emptyset$. Since D_i is a set of consecutive points with lower x-coordinate in L_i, we have $D_i = \{\text{HEAD}(i), \text{NEXT}(\text{HEAD}(i)), \ldots, \text{PMAX}(i)\}$, (PMAX($i$) is defined to be the point with the largest x-coordinate in D_i). Since the height of each point may decrease at most by one, our method is simple: for each L_i, first delete D_i from L_i, then insert D_{i+1} to head part of L_i.

Algorithm 1 Algorithm for DELETE operation

initialize $\text{PMAX}(1) \leftarrow p_{\text{DEL}}, D_1 \leftarrow \{p_{\text{DEL}}\}$, and $D_i \leftarrow \emptyset$ for $i > 1$
for $i = 2$ to $\omega(P)$ **do**
  $\text{PMAX}(i) \leftarrow \max\{p_x|\neg(\text{NEXT}(\text{PMAX}(i-1)) \prec p)\}$
  if $\text{PMAX}(i) = \emptyset$ **then**
    EXIT LOOP
  end if
  $D_i = \{p|p \in L_i \text{ and } p_x \leq \text{PMAX}(i)_x\}$
end for
$\text{PRED}'(p) = \emptyset$ for $p \in D_2$
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

We will illustrate our method based on the example in Fig. 3 and 4. By definition, $\text{PMAX}(1) = p_1$ and D_1 =$\{p1\}$. 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. 3, the points in the *shadow* area *dominate* D_1 but not any point in $L_1 \setminus D_1$, so $D_2 = \{p_2\}$. The point p_5 is the point with the largest x-coordinate that does not *dominate* $\text{NEXT}(\text{PMAX}(2)) = p_6$, so $\text{PMAX}(3) = p_5$ and D_3 =$\{p_4, p_5\}$. Next step is computing L'_i. $L'_1 = D_2 + (L_1 \setminus D_1) = \{p_2, p_3\}$, $L'_2 = \{p_4, p_5, p_6, p_7\}$, and $L'_3 = \{p_8\}$. In Fig. 4, the *dashed line* represents removing the D_i, and the *broad-brush line* represents the *concatenation* of D_{i+1} and $L_i \setminus D_i$.

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

Lemma 2. *The x-coordinate of any point in D_{i+1} is smaller than that of any point in $L_i \setminus D_i$.*

Proof. Suppose $\exists p \in D_{i+1}$, $q \in L_i \setminus D_i$ for a particular i, and $p_x > q_x$. Because $h(p)$ decreases, the point p does not dominate q, that means $p_y \leq q_y$. Before delete p_{DEL}, there exits $v \in L_i$ that p dominates v, i.e. $p_x > v_x$ and $p_y > v_y$.

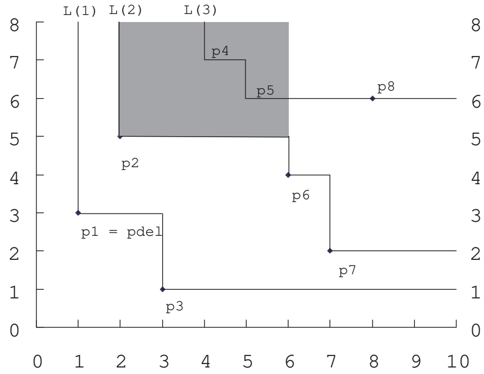

Fig. 3. Before DELETE operation

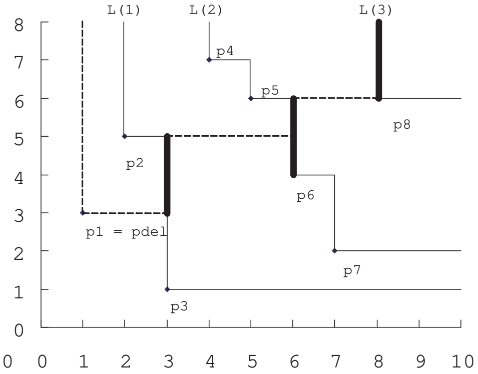

Fig. 4. After DELETE operation

Since v precedes q in L_i, we will have $v_x < q_x$, $v_y \geq q_y$ according to Theorem 1. Therefore, $p_y > v_y \geq q_y$ which contradicts with the inequity $p_y \leq q_y$. Thus, no such p, q exists. $\square$

By lemma 1, to compute PMAX$(i)(i > 1)$, we scan L_i from the head until the first point which is *dominating* NEXT(PMAX$(i-1)$). By lemma 2, for a fixed i, we can move 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 $L_i \setminus D_i$. Let $\mathscr{D}$ be the union of D_i, i.e. $\mathscr{D} = \bigcup_{i=1}^{\omega(\mathscr{W})} D_i$. Thus, the time complexity of maintaining the data structure after deleting p_{DEL} is $O(|\mathscr{D}|)$.

Theorem 2. *The cost of one DELETE operation equals the total number of points whose height decreases, i.e.* $O(|\mathscr{D}|)$.

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 INSERT operation is to find an *antichain* which p_{INS} belongs to. There are two cases:

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

By Theorem 1, if $\langle a_1, a_2, \ldots, a_t, p_{\text{INS}} \rangle$ is a longest chains, there always be another longest chain by replacing a_t by TAIL$(h(a_t))$. As in Fig. 5, 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 TAIL(k) for $k = 1, 2, \ldots, \omega(P)$, until we find the highest TAIL(k) that p_{INS} *dominates.* If $k = 0$, PRED$'(p_{\text{INS}}) = \emptyset$, otherwise PRED$'(p_{\text{INS}}) = $ TAIL(k). If $k = \omega(\mathscr{W})$, we create a new *antichain* $L'_{\omega(\mathscr{W})+1}$ with only a point p_{INS}. Since finding the antichain which p_{INS} belongs to requires $h(p_{\text{INS}})+1$ steps, the time complexity of INSERT opreation is $O(h(p_{\text{INS}}))$.

Theorem 3. *The cost of INSERT operation equals* $O(h(p_{INS}))$.

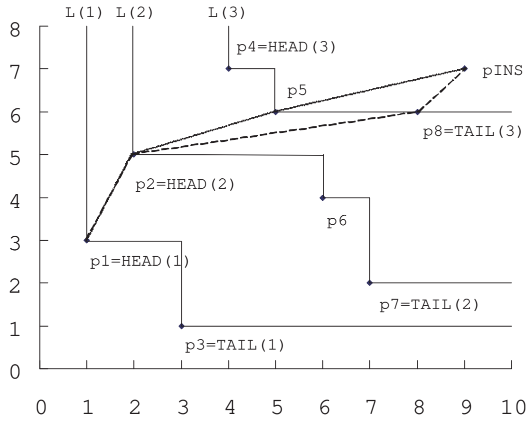

Fig. 5. INSERT operation

QUERY operation Suppose $\sigma = \langle p_1, p_2, \ldots, p_t \rangle$ is a longest chain in P that σ satisfies the x-coordinate of p_t is equal to or less than X_{QRY}. Let P' be the set of points in P whose x-coordinate is equal to or less than X_{QRY}, i.e. $P' = \{p \mid p \in P \text{ and } p_x \le X_{\text{QRY}}\}$. Let H be the length of a longest chain in P', which is the size of QUERY(X_{QRY})'s output.

By Theorem 1, since x-coordinate of HEAD(H) is the smallest among that of all points in L_H, if a longest chain in P' with any point $p \in L_H$ as the maximal point, there always be another longest chain in P' with HEAD(H) as the maximal point. Therefore, to compute H is to check HEAD(i) for $i = 1, 2, \ldots, \omega(P)$, until we find the highest HEAD(i) that the x-coordinate of HEAD(i) is equal to or less than X_{QRY}, i.e. $H = \max\{i \mid \text{HEAD}(i)_x \le X_{\text{QRY}}\}$. We can find a longest chain C in P' satisfying that $C = \langle c_1, c_2, \ldots, c_H \rangle$, $c_H = \text{HEAD}(H)$ and $c_i = \text{PRED}(c_{i+1})$ for $i = 1, 2, \ldots, H-1$. 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 QUERY(X_{QRY}) is $O(H)$.

Theorem 4. *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

Our algorithm is based on the data structure 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 in to $\mathscr{W}$ 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. $i < j$ if and only if $l_i < l_j$ or ($l_i = l_j$ and $r_i > r_j$). Since the endpoints are drawn from $\{1, 2, \ldots, n\}$, the preprocessing sorting can be completed in $O(n+m)$ time.

Two distinct windows W_a and W_b *intersect* if there exists a number c that $l(a) \le c \le r(a)$ and $l(b) \le c \le r(b)$; W_a and W_b are *disjointed* if they do not *intersect*; W_a *contains* W_b if $l(a) \le l(b) \le r(b) \le r(a)$; W_a and W_b *overlap*, if W_a *intersects* W_b but neither one of them *contains* the other.

Algorithm 2 Algorithm for LISSET Problem

```
initialize 𝒲 ← ∅
INSERT into 𝒲 the elements in π⟨l_1, r_1⟩ separately
QUERY(r_1) to output the LIS in 𝒲
for j = 1 to m − 1 do
  slide the window from W_j to W_{j+1}, let W_a = W_j, and W_b = W_{j+1}
  if (l_b > r_a) then {DISJOINT}
    reset 𝒲 ← ∅
    INSERT into 𝒲 the elements in π⟨l_b, r_b⟩ separately
  else if (l_b ≤ r_a and r_b > r_a) then {OVERLAP}
    DELETE from 𝒲 the elements in π⟨l_a, l_b − 1⟩ separately
    INSERT into 𝒲 the elements in π⟨r_a + 1, r_b⟩ separately
  else if (l_a ≤ l_b and r_b ≤ r_a) then {CONTAIN}
    if (l_b = l_a) then {SAME LEFT ENDPOINT}
      NO OPERATIONS
    else if l_b > l_a then {DIFFERENT LEFT ENDPOINTS}
      DELETE from 𝒲 the elements in π⟨l_a, l_b − 1⟩ separately
    end if
  end if
  QUERY(r_b) to output a LIS σ = π_{i_1} π_{i_2} ... π_{i_T} in 𝒲 satisfying i_T ≤ r_b
end for
```

Before analyzing the algorithm, depth_i is defined to be the *largest* height that π_i achieved in the m windows. In other words, among all increasing subsequences in m windows, depth_i is the length of the longest one with π_i as the maximal element.

Theorem 5 (LISSET Problem). *The algorithm described above computes the m longest increasing subsequences, one for each window, in total time $O(n + \textsc{output} + \sum_{i=1}^{n} depth_i)$.*

Proof. Most of the time required comes from three operations : INSERT, DELETE, QUERY. By Theorem 4, the cost of QUERY operation equals the total length of the LIS's in all m windows, so $T_{QUERY} = O(\textsc{output})$. By Theorem 3, the cost of inserting π_i is equal to the length of the longest LIS with π_i as the maximal element in the m windows, i.e. $T_{INS} = O(\sum_{i=1}^{n} \text{depth}_i)$. For DELETE operations, it is difficult to analyse the cost of each DELETE operation, but we can calculate the overall cost of DELETE operations. By Theorem 2, the cost equals 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, $T = T_{QUERY} + T_{INS} + T_{DEL} = O(n + \textsc{output} + \sum_{i=1}^{n} \text{depth}_i)$. □

Theorem 6 (LISW Problem). *Our algorithm finds the longest increasing subsequence in a sliding window over a sequence of n elements in $O(\text{OUTPUT})$ time.*

Proof. By Theorem 5, 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 $O(n + \text{OUTPUT} + \sum_{j=1}^{n} \text{depth}_j) = O(n + \text{OUTPUT}) = O(\text{OUTPUT})$. □

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

If average window size is $O(w)$, the optimal algorithm finds LIS in a single window in time $O(\min\{w \log w, w \log \log n\})$, so the time complex of finding LIS's in m windows is $O(\min\{mw \log w, mw \log \log n\})$. On the other hand, it was shown in[BS00] that average length of LIS of a permutation of length n is asymptotically $2\sqrt{n}$. Suppose the permutation π is randomly chosen from $n!$ different permutations, so the relative ordering of the elements in each window is also randomly chosen. Thus, the expected length of LIS in any window is asymptotically $2\sqrt{w}$. By Theorem 2, we can prove that $\text{depth}_i = O(\sqrt{w})$. Thus, 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 *straight forward* approach.

6 Conclusions 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 straight forward methods. In addition, we show that LISW problem is a subcase of LISSET problem. Our algorithm solve LISW in time $O(\text{OUTPUT})$, while the time complexity of the best solution for LISW previously achieved is $O(\text{OUTPUT} + n \log \log n)$.

Some problems related to LISSET problem are still open problems. Firstly, the time complexity of finding *global maximal* LIS among all LIS's in all windows is particularly interesting. Because our algorithm only finds the global maximal LIS after computing all LIS's in all m window individually, it is expected to design an algorithm to find global maximal LIS *directly* in the future. The second problem is to design a space-efficient *online* algorithm to output the LIS in a window with size w in *linear* time $O(w)$. Our algorithm requires $O(n^2)$ time and $O(n^2)$ space for preprocessing to solve this problem.

Acknowledgements

Thanks to Hong Zhu, Binhai Zhu, Yong Yu, Qiqi Yan, and the reviewers for their comments.

References

[AAH93] Alberto Apostolico, Mikhail J. Atallah, and Susanne E. Hambrusch. 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.

[DLNZ03] Erik Vee David Liben-Nowell and An Zhu. Finding longest increasing and common subsequences in streaming data. Technical Report MIT-LCS-931, Cambridge, MA 02139, November 2003.

[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.

[Gol80] M.C. Golumbic. *Algorithmic graph theory and perfect graphs*. Academic Press, New York, NY, 1980.

[Knu98] D. E. Knuth. *The Art of Computer Programming. Vol 3, Sorting and Searching. Second Edition*. Addison-Wesley, 1998.

[MNKF90] Sumio Masuda, Kazuo Nakajima, Toshinobu Kashiwabara, and Toshio Fujisawa. Efficient algorithms for finding maximum cliques of an overlap graph. *Networks*, 20:157–171, 1990.

[MRHR98] P. Raghavan M. R. Henzinger and S. Rajagopalon. Computing on data streams. Technical Report 1998-011, Digital Equipment Corporation,Systems Research Center, May 1998.

[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.