Computing Longest Increasing Subsequences over Sequential Data Streams

Youhuan Li
Peking University
Beijing, China
liyouhuan@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@pku.edu.cn

Huaming Zhang
University of Alabama in Huntsville,USA
hzhang@cs.uah.edu

Dongyan Zhao
Peking University
Beijing, China
zhaody@pku.edu.cn

ABSTRACT

In this paper, we propose a data structure, a quadruple neighbor list (QN-list, for short), to support real time queries of all longest increasing subsequence (LIS) and LIS with constraints over sequential data streams. The QN-List built by our algorithm requires $O(w)$ space, where w is the time window size. The running time for building the initial QN-List takes $O(w \log w)$ time. Applying the QN-List, insertion of the new item takes $O(\log w)$ time and deletion of the first item takes $O(w)$ time. To the best of our knowledge, this is the first work to support both LIS enumeration and LIS with constraints computation by using a single uniform data structure for real time sequential data streams. Our method outperforms the state-of-the-art methods in both time and space cost, not only theoretically, but also empirically.

1. INTRODUCTION

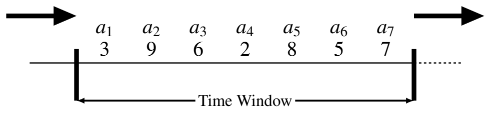
Sequential data is a time series consisting of a sequence of data points, which are obtained by successive measurements made over a period of time. Lots of technical issues have been studied over sequential data, such as (approximate) pattern-matching query [9, 17], clustering [18]. Among these, computing the Longest Increasing Subsequence (LIS) over sequential data is a classical problem. Let's see the definition of LIS as follows.

Definition 1. ***(Longest Increasing Subsequence).*** *Let $\alpha = \{a_1, a_2, \cdots, a_n\}$ be a sequence, an increasing*[1] *subsequence s of α is a subsequence of α whose elements are sorted in order from the smallest to the biggest. An increasing subsequence s of α is called a Longest Increasing Subsequence (LIS) if there is no other increasing subsequence s' with $|s| < |s'|$. A sequence α may contain multiple LIS, all of which have the same length. We denote the set of LIS of α by $LIS(\alpha)$.*

[1]Increasing subsequence in this paper is not required to be strictly monotone increasing and all items in α can also be arbitrary numerical value.

Proceedings of the VLDB Endowment, Vol. 10, No. 3
Copyright 2016 VLDB Endowment 2150-8097/16/11.

Besides the static model (i.e., computing LIS over a given sequence α), computing LIS has been considered in the streaming model [3, 6]. Given an infinite time-evolving sequence $\alpha_\infty = \{a_1, ..., a_\infty\}$ ($a_i \in \mathbb{R}$), we continuously compute LIS over the subsequence induced by the time window $\{a_{i-(w-1)}, a_{i-(w-2)}, ..., a_i\}$. The size of the time window is the number of the items it spans in the data stream. Consider the sequence $\alpha = \{3, 9, 6, 2, 8, 5, 7\}$ under window W in Figure 1. There are four LIS in α: $\{3, 6, 7\}, \{3, 6, 8\}$, $\{2,5,7\}$ and $\{3, 5, 7\}$. Besides LIS enumeration, we introduce two important features of LIS, i.e., *gap* and *weight* and compute LIS with various constraints, where "gap" measures the value difference between the tail and the head item of LIS and "weight" measures the sum of all items in LIS (formally defined in Definitions 3-4). Figure 1 shows LIS with various specified constraints. In the following, we demonstrate the usefulness of LIS in different applications.



LIS with Maximum Weight $\{a_1 = 3, a_3 = 6, a_5 = 8\}$
LIS with Minimum Weight $\{a_4 = 2, a_6 = 5, a_7 = 7\}$
LIS with Maximum Gap $\{a_1 = 3, a_3 = 6, a_5 = 8\}, \{a_4 = 2, a_6 = 5, a_7 = 7\}$
LIS with Minimum Gap $\{a_1 = 3, a_6 = 5, a_7 = 7\}, \{a_1 = 3, a_3 = 6, a_7 = 7\}$

Figure 1: Computing LIS with constraints in data stream model

Example 1: Realtime Stock Price Trend Detection. LIS is a classical measure for sortedness and trend analysis [11]. As we know, a company's stock price forms a time-evolving sequence and the real-time measuring the stock trend is significant to the stock analysis. Given a sequence α of the stock prices within a period, an LIS of α measures an uptrend of the prices. We can see that price sequence with a long LIS always shows obvious upward tendency for the stock price even if there are some price fluctuations.

Although the LIS length can be used to measure the uptrend stability, LIS with different gaps indicate different growth intensity. For example, Figure 2 presents the stock prices sequences of two company: A and B. Although both sequences of A and B have the same LIS length (5), growth intensity of A's stock obvious dominates that of B, which is easily observed from the different gaps in LIS in A and B. Therefore, besides LIS length, *gap* is another feature of LIS that weights the growth intensity. We consider that the computation of LIS with extreme gap that is more likely chosen as measurement of growth intensity than a random LIS. Furthermore, this paper also considers other constraints for LIS, such as *weight* (see Definition 3) and study how to compute LIS with constraints directly rather than using post-processing technique.

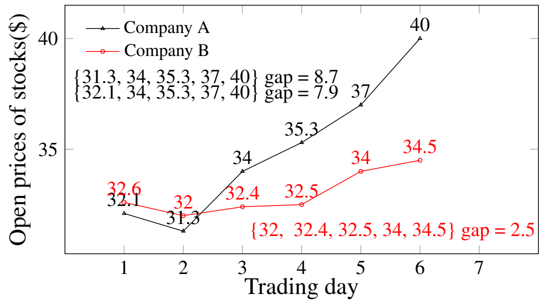


Figure 2: LIS with different gaps of stock price sequence

Example 2: Biological Sequence Query. LIS is also used in biological sequence matching [3, 24]. For example, Zhang [24] designed a two-step algorithm (BLAST+LIS) to locate a transcript or protein sequence in the human genome map. The BLAST (Basic Local Alignment Search Tool) [4] algorithm is to identify high-scoring segment pairs (HSPs) between query transcript sequence Q and a long genomic sequence L. Figure 3 visualizes the outputs of BLAST. The segments with the same color (number) denote the HSPs. For example, segment 2 (the red one) has two matches in the genomic sequence L, denoted as 2_1 and 2_2. To obtain a global alignment, the matches of segments 1, 2, 3 in the genomic sequence L should coincide with the segment order in query sequence Q, which constitutes exactly the LIS (in L) that are listed in Figure 3. For example, LIS $\{1, 2_1, 3_1\}$ represents a global alignment of Q over sequence L. Actually, there are three different LIS in L as shown in Figure 3, which correspond to three different alignments between query transcript/protein Q and genomic sequence L. Obviously, outputting only a single LIS may miss some important findings. Therefore, we should study LIS enumeration problem.

We extend the above LIS enumeration application into the sliding window model [13]. In fact, the range of the whole alignment result of Q over L should not be too long. Thus, we can introduce a threshold length $|w|$ to discover all LIS that span no more than $|w|$ items, i.e, all LIS in each time window with size $|w|$. This is analogous to our problem definition in this paper.

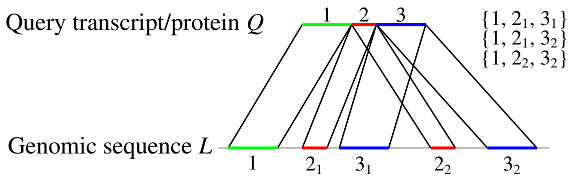


Figure 3: Biological Sequence Alignment

Although LIS has received considerable attention from the theoretical computer science community [6, 7, 16, 22], none of the existing approaches support both LIS enumeration and constrained LIS enumeration simultaneously. The method presented in [6] supports LIS enumeration, but fails to compute constrained LIS. In [7] and [22], the method can be used to compute constrained LIS, but not to enumerate all LIS. More importantly, many works are based on *static sequences* and techniques developed in these works cannot handle updates which are essential in the context of data streams. To the best of our knowledge, there are only three research articles that addressed the problem of computing LIS over data stream model [3, 6, 8]. None of them computes constrained LIS. Literature review and the comparative studies of our method against other related work are given in Section 2 and Section 7, respectively.

1.1 Our Contributions

Observed from the above examples, we propose a novel solution in this paper that studies both LIS enumeration and computing LIS with constraints with *a uniform method under the data stream model*. We propose a novel data structure to efficiently support both LIS enumeration and LIS with constraints. Furthermore, we design an efficient update algorithm for the maintenance of our data structure so that our approach can be applied to the data stream model. Theoretical analysis of our algorithm proves that our method outperforms the state-of-the-arts work (see Section 7.1 for details). We prove that the space complexity of our data structure is $O(w)$, while the algorithm proposed in [6] needs a space of size $O(w^2)$. Time complexities of our data structure construction and update algorithms are also better than [6]. For example, [6] needs $O(w^2)$ time for the data structure construction, while our method needs $O(w \log w)$ time. Besides, we prove that both our LIS enumeration and LIS with constraints query algorithms are *optimal output-sensitive* algorithms[2]. Comprehensive comparative study of our results against previous results is given in Section 7. We use real and synthetic datasets to experimentally evaluate our approach against the state-of-the-arts work. Experimental results also confirm that our algorithms outperform existing algorithms. Experimental codes and datasets are available at Github [1].

We summarize our major contributions in the following:

1. We are the first to consider the computation of both LIS with constraints and LIS enumeration in the data stream model.
2. We introduce a novel data structure to handle both LIS enumeration and computation of LIS with constraints uniformly.
3. Our data structure is scalable under data stream model because of the linear update algorithm and linear space cost.
4. Extensive experiments confirm the superiority of our method.

2. RELATED WORK

LIS-related problems have received considerable attention in the literature. We give a briefly review of the related work from the perspectives of the *solution* and *problem definition*, respectively.

2.1 Solution Perspective

Generally, existing LIS computation approaches can be divided into following three categories:

1. Dynamic Programming-based. Dynamic programming is a classical method to compute the length of LIS. Given a sequence α, assuming that α_i denotes the prefix sequence consisting of the first i items of α, then the dynamic programming-based method is to compute the LIS of α_{i+1} after computing the LIS of α_i. However, dynamic programming-based method costs $O(w^2)$ time where n denotes the length of the sequence α. Dynamic programming-based method can be easily extended to enumerate all LIS in a sequence which costs $O(w^2)$ space.

2. Young's tableau-based. [20] proposes a Young's tableau-based solution to compute LIS in $O(w \log w)$ time. The width of the first row of Young's tableau built over a sequence α is exactly the length of LIS in α. Albert et al.[3] followed the Young's tableau-based work to compute the LIS length in sliding window. They maintained the first row of Young's tableau, called principle row, when window slides. For a sequence α in a window, there are $n = |\alpha|$ suffix subsequences and the prime idea in [3] is to compress all principle rows of these suffix subsequence into an array, which can be updated in $O(w)$ time when update happens. Besides, they can output an LIS with a tree data structure which costs $O(w^2)$ space.

3. Partition-based. There are also some work computing LIS by partitioning items in the sequence [6, 7, 8, 22]. They classify items

[2]The algorithm time complexity is linear to the corresponding output size.

into l partitions: P_1,P_2...,P_l, where l is the length of LIS of the sequence. For each item a in P_k ($k = 1, ..., l$), the maximum length of the increasing subsequence ending with a is exactly k. Thus, when partition is built, we can start from items in P_l and then scan items in P_{l-k} ($1 \leq k < l$) to construct an LIS. The partition is called different names in different approaches, such as *greedy-cover* in [7, 8], *antichain* in [6]. Note that [7] and [22] conduct the partition over a static sequence to efficiently compute LIS with constraints. [8] use partition-based method as subprogram to find out the largest LIS length among $n - w$ windows where w is the size of the sliding window over a sequence α of size n. Their core idea is to avoid constructing partition on the windows whose LIS length is less than those previously found. In fact, they re-compute the greedy-cover in each of the windows that are not filtered from scratch. None of the partition-based solutions address the data structure maintenance issues expect for [6]. [6] is the only one to study the LIS enumeration in streaming model. Both of their insertion and deletion algorithms cost $O(w)$ time [6]. Besides, they assign each item with $O(w)$ pointers and thus their method costs $O(w^2)$ space.

Our approach belongs to the partition-based solution, where each horizontal list(see Definition 10) is a *partition*. While, with up/down neighbors in QN-list (see Definition 9 and 11), our data structure costs only $O(w)$ space. Besides, the insertion and deletion time of our method is $O(\log w)$ and $O(w)$, respectively, which makes it suitable in the streaming context. Furthermore, our data structure supports both LIS enumeration and LIS with various constraints.

2.2 Problem Perspective

We briefly position our problem in existing work on LIS computation in *computing task* and *computing model*. Note that LIS can also be used to compute LCS (longest common subsequence) between two sequences [12], but that is not our focus in this paper. First, there are three categories of LIS computing tasks. The first is to compute the length of LIS and output a single LIS (not enumerate all) in sequence α [3, 8, 10, 19, 20]. The second is LIS enumeration, which finds all LIS in a sequence α [5, 6]. [5] computes LIS enumeration only on the sequence that is required to be a permutation of $\{1,2,...,n\}$ rather than a general sequence (such as $\{3, 9, 6, 2, 8, 5, 7\}$ in the running example). The last computing task studies LIS with constraints, such as gap and weight [7, 22]. On the other hand, there are two computing models for LIS. One is the static model assuming that the sequence α is given without changes. For example, [7, 20, 21, 22] are based on the static model. These methods cannot be applied to the streaming context directly except re-computing LIS from scratch in each time window. The other model is the data stream model, which has been considered in some recent work[3, 6].

Table 1 illustrates the existing works from two perspectives: *computing task* and *computing model*. There are two observations from the table. First, there is no existing uniform solution for all LIS-related problems, such as LIS length, LIS enumeration and LIS with constraints. Note that any algorithm for computing LIS enumeration and LIS with constraints can be applied to computing LIS length directly. Thus, we only consider LIS enumeration and LIS with constraints in the later discussion. Second, no algorithm supports computing LIS with constraints in the streaming context. Therefore, the major contribution of our work lies in that we propose a uniform solution (the same data structure and computing framework) for all LIS-related issues in the streaming context. Table 1 properly positions our method with regard to existing works.

None of the existing work can be easily extended to support all LIS-related problems in the data steam model except for LISSET [6], which is originally proposed to address LIS enumeration in the sliding window model. Also, LISSET can compute LIS with constraints using post-process technique (denoted as LISSET-post in Figure 11). So, we compare our method with LISSET not only theoretically, but also empirically in Section 7. LISSET requires $O(w^2)$ space while our method only uses $O(w)$ space, where w is the size of the input sequence. Experiments show that our method outperforms LISSET significantly, especially computing LIS with constraints (see Figures 11f-11i).

Computing Task	Static only	Stream
LIS length(outputting a single LIS)	[5][7][20][21][22]	[3][6][8], Our Method
LIS Enumeration	[5][3]	[6], Our Method
LIS with constraints	[7][22]	Our Method

Table 1: Our Method VS. Existing Works on Computing LIS(s)

3. PROBLEM FORMULATION

Given a sequence $\alpha = \{a_1, a_2, \cdots, a_n\}$, the set of increasing subsequences of α is denoted as $IS(\alpha)$. For a sequence s, the head and tail item of s is denoted as s^h and s^t, respectively. We use $|s|$ to denote the length of s.

Consider an infinite time-evolving sequence $\alpha_\infty = \{a_1, ..., a_\infty\}$ ($a_i \in \mathbb{R}$). In the sequence α_∞, each a_i has a unique position i and a_i occurs at a corresponding time point t_i, where $t_i < t_j$ when $0 < i < j$. We exploit the *tuple-basis* sliding window model [13] in this work. There is an internal *position* to tuples based on their arrival order to the system, ensuring that an input tuple is processed as far as possible before another input tuple with a higher *position*. A sliding window W contains a consecutive block of items in $\{a_1, \cdots, a_\infty\}$, and W slides a single unit of position per move towards a_∞ continually. We denote the size of the window W by w, which is the number of items within the window. During the time $[t_i, t_{i+1})$, items of α within the sliding time window W induce the sequence $\{a_{i-(w-1)}, a_{i-(w-2)}, ..., a_i\}$, which will be denoted by $\alpha(W, i)$. Note that, in the sliding window model, as the time window continually shifts towards a_∞, at a pace of one unit per move, the sequence formed and the corresponding set of all its LIS will also change accordingly. In the remainder of the paper, all LIS-related problems considered are in the data stream model with sliding windows.

Definition 2. ***(LIS-enumeration)****. Given a time-evolving sequence $\alpha_\infty = \{a_1, ..., a_\infty\}$ and a sliding time window W of size w,* LIS-enumeration *is to report $LIS(\alpha(W, i))$ (i.e., all LIS within the sliding time W) continually as the window W slides. All LIS in the same time window have the same length.*

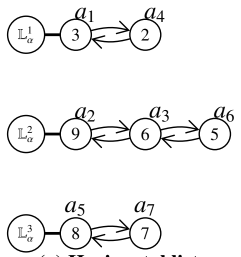
As mentioned in Introduction, some applications are interested in computing LIS with constraints instead of simply enumerating all of them. Hence, we study the following constraints over the LIS's *weight* (Definition 3) and *gap* (Definition 4), after which we define several problems computing LIS with various constraints (Definition 5) [4] .

Definition 3. ***(Weight)****. Let α be a sequence, s be an LIS in $LIS(\alpha)$. The* weight *of s is defined as $\sum_{a_i \in s} a_i$, i.e., the sum of all the items in s, we denote it by weight(s).*

Definition 4. ***(Gap)****. Let α be a sequence, s be an LIS in $LIS(\alpha)$. The* gap *of s is defined as $gap(s) = s^t - s^h$, i.e., the difference between the tail s^t and the head s^h of s.*

[3][5] computes LIS enumeration only on the sequence that is required to be a permutation of $\{1,2,...,n\}$.

[4]So far, eight kinds of constraints for LIS were proposed in the literature [7, 22, 23]. Due to the space limit, we only study four of them (i.e., max/min weight/gap) in this paper. However, our method can also easily support the other four constraints, which are provided in Appendix H of the full version of this paper [2] .


(a) Horizontal lists

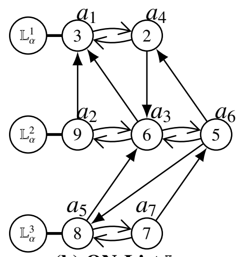

(b) QN-List $\mathbb{L}_\alpha$

Figure 4: Horizontal lists and QN-List

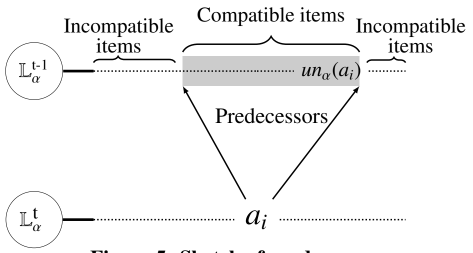


Figure 5: Sketch of predecessors

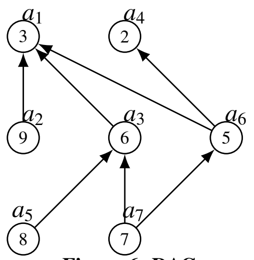


Figure 6: DAG

Definition 5. ***(Computing LIS with Constraint)**. Given a time-evolving sequence $\alpha_\infty = \{a_1, ..., a_\infty\}$ and a sliding window W, each of the following problems is to report all the LIS subject to its own specified constraint within a time window continually as the window slides. For $s \in LIS(\alpha(W, t_i))$:*

*s is an **LIS with Maximum Weight** if*

$$\forall s' \in LIS(\alpha(W, t_i)), weight(s) \geq weight(s')$$

*s is an **LIS with Minimum Weight** if*

$$\forall s' \in LIS(\alpha(W, t_i)), weight(s) \leq weight(s')$$

*s is an **LIS with Maximum Gap** if*

$$\forall s' \in LIS(\alpha(W, t_i)), gap(s) \geq gap(s')$$

*s is an **LIS with Minimum Gap** if*

$$\forall s' \in LIS(\alpha(W, t_i)), gap(s) \leq gap(s')$$

A running example that is used throughout the paper is given in Figure 1, which shows a time-evolving sequence α_∞ and its first time window W.

4. QUADRUPLE NEIGHBOR LIST $\mathbb{L}_\alpha$

In this section, we propose a data structure, a *q*uadruple *n*eighbor list (QN-list for short), denoted as $\mathbb{L}_\alpha$, for a sequence $\alpha = \{a_1, a_2, ..., a_w\}$, which is induced from α_∞ by a time window W of size w. Some important properties and the construction of $\mathbb{L}_\alpha$ are discussed in Section 4.2 and Section 4.3, respectively. In Section 4.4, we present an efficient algorithm over $\mathbb{L}_\alpha$ to enumerate all LIS in α. In the following two sections, we will discuss how to update the QN-List efficiently in data stream scenario (Section 5) and compute LIS with constraints (Section 6).

4.1 $\mathbb{L}_\alpha$—Background and Definition

For the easy of the presentation, we introduce some concepts of LIS before we formally define the quadruple neighbor list (QN-List, for short). Note that two concepts (*rising length* and *horizontal list*) are analogous to the counterpart in the existing work. We explicitly state the connection between them as follows.

Definition 6. ***(Compatible pair)** Let $\alpha = \{a_1, a_2, ..., a_w\}$ be a sequence. a_i is* compatible *with a_j if $i < j$ and $a_i \leq a_j$ in α. We denote it by $a_i \overset{\alpha}{\preccurlyeq} a_j$.*

Definition 7. ***(Rising Length)** [6] [5] Given a sequence $\alpha = \{a_1, a_2, ..., a_w\}$ and $a_i \in \alpha$, we use $IS_\alpha(a_i)$ to denote the set of all* increasing subsequences *of α that ends with a_i.*

The rising length *$RL_\alpha(a_i)$ of a_i is defined as the maximum length of subsequences in $IS_\alpha(a_i)$, namely,*

$$RL_\alpha(a_i) = \max\{\, |s| \mid s \in IS_\alpha(a_i)\}$$

[5]*Rising length* in this paper is the same as *height* defined in [6]. We don't use *height* here to avoid confusion because *height* is also defined as the difference between the head item and tail item of an LIS in [22].

For example, consider the sequence $\alpha = \{a_1 = 3, a_2 = 9, a_3 = 6, a_4 = 2, a_5 = 8, a_6 = 5, a_7 = 7\}$ in Figure 1. Consider $a_5 = 8$. There are four increasing subsequences$\{a_1 = 3, a_5 = 8\}$, $\{a_3 = 6, a_5 = 8\}$, $\{a_4 = 2, a_5 = 8\}$, $\{a_1 = 3, a_3 = 6, a_5 = 8\}$ that end with a_5[6]. The maximum length of these increasing subsequences is 3. Hence, $RL_\alpha(a_5) = 3$.

Definition 8. ***(Predecessor)**. Given a sequence α and $a_i \in \alpha$, for some item a_j, a_j is a* predecessor *of a_i if*

$$a_j \overset{\alpha}{\preccurlyeq} a_i \ AND\ RL_\alpha(a_j) = RL_\alpha(a_i) - 1$$

and the set of predecessors of a_i is denoted as $Pred_\alpha(a_i)$.

In the running example in Figure 1, a_3 is a predecessor of a_5 since $a_3 \overset{\alpha}{\preccurlyeq} a_5$ and $RL_\alpha(a_3)(= 2) = RL_\alpha(a_5)(= 3) - 1$. Analogously, a_1 is also a predecessor of a_3.

With the above definitions, we introduce four neighbours for each item a_i as follows:

Definition 9. ***(Neighbors of an item)**. Given a sequence α and $a_i \in \alpha$, a_i has up to four neighbors.*

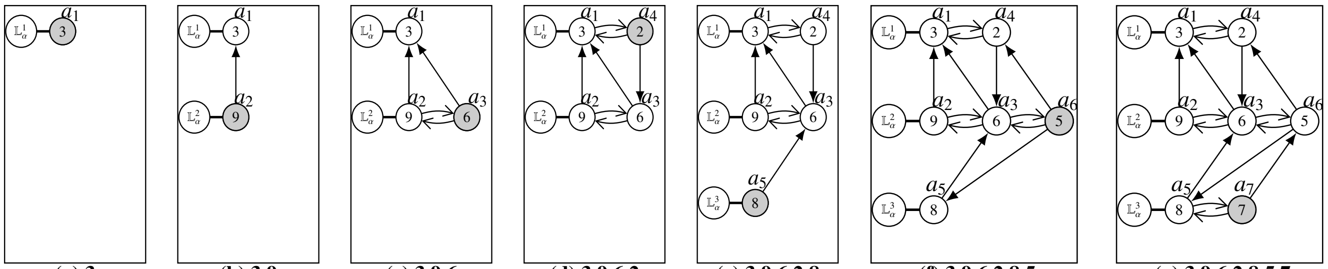
1. ***left neighbor** $ln_\alpha(a_i)$: $ln_\alpha(a_i) = a_j$ if a_j is the* nearest *item* before *a_i such that $RL_\alpha(a_i) = RL_\alpha(a_j)$.*
2. ***right neighbor** $rn_\alpha(a_i)$: $rn_\alpha(a_i) = a_j$ if a_j is the* nearest *item* after *a_i such that $RL_\alpha(a_i) = RL_\alpha(a_j)$.*
3. ***up neighbor** $un_\alpha(a_i)$: $un_\alpha(a_i) = a_j$ if a_j is the* nearest *item* before *a_i such that $RL_\alpha(a_j) = RL_\alpha(a_i) - 1$.*
4. ***down neighbor** $dn_\alpha(a_i)$: $dn_\alpha(a_i) = a_j$ if a_j is the* nearest *item* before *a_i such that $RL_\alpha(a_j) = RL_\alpha(a_i) + 1$.*

Apparently, if $a_i = ln_\alpha(a_j)$ then $a_j = rn_\alpha(a_i)$. Besides, we know that left neighbor(Also right neighbor) of item a_i has the same rising length as a_i and naturally, items linked according to their left and right neighbor relationship forms a *horizontal list*, which is formally defined in Definition 10. The horizontal lists of α is presented in Figure 4a.

Definition 10. ***(Horizontal list)**. Given a sequence α, consider the subsequence consisting of all items whose rising lengths are k: $s_k = \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$, $i_1 < i_2, ..., < i_k$. We know that for $1 \leq k' < k$, $a_{i_{k'}} = ln_\alpha(a_{i_{k'+1}})$ and $a_{i_{k'+1}} = rn_\alpha(a_{i_{k'}})$. We define the list formed by linking items in s_k together with left and right neighbor relationships as a horizontal list, denoted as $\mathbb{L}_\alpha^k$.*

Recall the *partition-based solutions* mentioned in Section 2. Each horizontal list is essentially a *partition*, which is the same as a *greedy-cover* in [8] and *antichain* in [6]. Based on the horizontal list, we define our data structure QN-list (Definition 11) as follows.

[6]Strictly speaking, $\{a_5\}$ is also an increasing subsequence with length 1.



(a) $\underline{3}$ **(b) 3,$\underline{9}$** **(c) 3,9,$\underline{6}$** **(d) 3,9,6,$\underline{2}$** **(e) 3,9,6,2,$\underline{8}$** **(f) 3,9,6,2,8,$\underline{5}$** **(g) 3,9,6,2,8,5,$\underline{7}$**

Figure 7: Example of the quadruple neighbor list construction on sequence: $\{a_1 = 3, a_2 = 9, a_3 = 6, a_4 = 2, a_5 = 8, a_6 = 5, a_7 = 7\}$.

Definition 11. ***(Quadruple Neighbor List (QN-List)).*** *Given a sequence $\alpha = \{a_1, ..., a_w\}$, the quadruple neighbor list over α (denoted as $\mathbb{L}_\alpha$) is a data structure containing all horizontal lists (See Definition 10) of α and each item a_i in $\mathbb{L}_\alpha$ is also linked directly to its up neighbor and down neighbor. In essence, $\mathbb{L}_\alpha$ is constructed by linking all items in α with their four kinds of neighbor relationship. Specifically, $|\mathbb{L}_\alpha|$ denotes the number of horizontal lists in $\mathbb{L}_\alpha$.*

Figure 4b presents the QN-List $\mathbb{L}_\alpha$ of running example sequence α (in Figure 1) and the horizontal curve arrows indicate the left and right neighbor relationship while the vertical straight arrows indicate the up and down neighbor relationship.

Theorem 1. *Given a sequence $\alpha = \{a_1, ..., a_w\}$, the data structure $\mathbb{L}_\alpha$ defined in Definition 11 uses $O(w)$ space* [7].

4.2 $\mathbb{L}_\alpha$—Properties

Next, we discuss some properties of the QN-List $\mathbb{L}_\alpha$. These properties will be used in the maintenance algorithm in Section 5 and various $\mathbb{L}_\alpha$-based algorithms in Section 6.

LEMMA 1. *Let $\alpha = \{a_1, a_2, ..., a_w\}$ be a sequence. Consider two items a_i and a_j in a horizontal list $\mathbb{L}^t_\alpha$ (see Definition 10).*

1. *If $t = 1$, a_i has no predecessor. If $t > 1$ then a_i has at least one predecessor and all predecessors of a_i are located in $\mathbb{L}^{t-1}_\alpha$.*
2. *If $rn_\alpha(a_j) = a_i$, then $i > j$ and $a_i < a_j$. If $ln_\alpha(a_j) = a_i$, then $i < j$ and $a_i > a_j$. Items in a horizontal list $\mathbb{L}^t_\alpha$ ($t = 1, \cdots, m$) are monotonically decreasing while their subscripts (i.e., their original position in α) are monotonically increasing from the left to the right. And no item is compatible with any other item in the same list.*
3. *$\forall a_i \in \alpha$, all predecessors of a_i form a nonempty consecutive block in $\mathbb{L}^{t-1}_\alpha$ ($t > 1$).*
4. *$un_\alpha(a_i)$ is the rightmost predecessor of a_i in $\mathbb{L}^{t-1}_\alpha$ ($t > 1$).*

Figure 5 shows that all predecessors of $a_i \in \mathbb{L}^t_\alpha$ form a consecutive block from $un_\alpha(a_i)$ to the left in $\mathbb{L}^{t-1}_\alpha$, i.e., Lemma 1(3).

LEMMA 2. *Given sequence α and its $\mathbb{L}_\alpha$, $\forall\ a_i \in \mathbb{L}^t_\alpha$ ($1 \le t \le m$).*

1. *$RL_\alpha(a_i) = t$ if and only if $a_i \in \mathbb{L}^t_\alpha$. In addition, the length of LIS in α is exactly the number of horizontal lists in $\mathbb{L}_\alpha$.*
2. *$un_\alpha(a_i)$(if exists) is the rightmost item in $\mathbb{L}^{t-1}_\alpha$ which is before a_i in sequence α.*
3. *$dn_\alpha(a_i)$(if exists) is the rightmost item in $\mathbb{L}^{t+1}_\alpha$ which is before a_i in sequence α. Besides, $dn_\alpha(a_i) > a_i$.*

LEMMA 3. *Given sequence α and its $\mathbb{L}_\alpha$, for $1 \le i, j \le |\mathbb{L}_\alpha|$*

$$Tail(\mathbb{L}^i_\alpha) \le Tail(\mathbb{L}^j_\alpha) \leftrightarrow i \le j$$

where $Tail(\mathbb{L}^i_\alpha)$ denotes the last item in list $\mathbb{L}^i_\alpha$.

[7]Due to space limits, all proofs for theorems and lemmas are given in Appendix B of the full version of this paper [2] .

4.3 $\mathbb{L}_\alpha$—Construction

The construction of $\mathbb{L}_\alpha$ over sequence α lies in the determination of the four neighbors of each item in α. We discuss the construction of $\mathbb{L}_\alpha$ as follows. Figure 7 visualizes the steps of constructing $\mathbb{L}_\alpha$ for a given sequence α.

Building QN-List $\mathbb{L}_\alpha$.

1. Initially, four neighbours of each item a_i are set NULL;
2. At step 1, $\mathbb{L}^1_\alpha$ is created in $\mathbb{L}_\alpha$ and a_1 is added into $\mathbb{L}^1_\alpha$ [8];
3. At step 2, if $a_2 < a_1$, it means $RL_\alpha(a_2) = RL_\alpha(a_1) = 1$. Thus, we append a_2 to $\mathbb{L}^1_\alpha$. Since a_2 comes after a_1 in sequence α, we set $rn_\alpha(a_1) = a_2$ and $ln_\alpha(a_2) = a_1$ respectively.

   If $a_2 \ge a_1$, we can find an increasing subsequence $\{a_1, a_2\}$, i.e, $RL_\alpha(a_2) = 2$. Thus, we create the second horizontal list $\mathbb{L}^2_\alpha$ and add a_2 to $\mathbb{L}^2_\alpha$. Furthermore, it is straightforward to know a_1 is the nearest predecessor of a_2; So, we set $un_\alpha(a_2) = a_1$;
4. (By the induction method) At step i, assume that the first $i-1$ items have been correctly added into the QN-List (in essence, the QN-List over the subsequence of the first $(i-1)$ items of α is built), let's consider how to add the i-th item a_i into the data structure. Let m denote the number of horizontal lists in the current $\mathbb{L}_\alpha$. Before adding a_i into $\mathbb{L}_\alpha$, let's first figure out the rising length of a_i. Consider a horizontal list $\mathbb{L}^t_\alpha$, we have the following two conclusions[9]:

   (a) If $Tail(\mathbb{L}^t_\alpha) > a_i$, then $RL_\alpha(a_i) \le t$. Assume that $RL_\alpha(a_i) > t$. It means that there exits at least one item a_j ($\in \mathbb{L}^t_\alpha$) such that $a_j \overset{\alpha}{\preccurlyeq} a_i$, i.e., a_j is a predecessor (or recursive predecessor) of a_i. As we know $Tail(\mathbb{L}^t_\alpha)$ is the minimum item in $\mathbb{L}^t_\alpha$ (see Lemma 2). $Tail(\mathbb{L}^t_\alpha) > a_i$ means that all items in $\mathbb{L}^t_\alpha$ are larger than a_i. That is contradicted to $a_j \overset{\alpha}{\preccurlyeq} a_i \wedge a_j \in \mathbb{L}^t_\alpha$. Thus, $RL_\alpha(a_i) \le t$.

   (b) If $Tail(\mathbb{L}^t_\alpha) \le a_i$, then $RL_\alpha(a_i) > t$. Since $Tail(\mathbb{L}^t_\alpha)$ is before a_i in α and $Tail(\mathbb{L}^t_\alpha) \le a_i$, $Tail(\mathbb{L}^t_\alpha)$ is compatible a_i. Let us consider an increasing subseqeunce s ending with $Tail(\mathbb{L}^t_\alpha)$, whose length is t since $Tail(\mathbb{L}^t_\alpha)$'s rising length is t. Obviously, $s' = s \oplus a_i$ is a length-(t+1) increasing subsequence ending with a_i. In other words, the rising length of a_i is at least $t+1$, i.e, $RL_\alpha(a_i) > t$.

   Besides, we know that $Tail(\mathbb{L}^t_\alpha) \ge Tail(\mathbb{L}^{t'}_\alpha)$ if $t \ge t'$(see Lemma 3). Thus, we need to find the first list $\mathbb{L}^t_\alpha$ whose tail $Tail(\mathbb{L}^t_\alpha)$ is larger than a_i. Then, we append a_i to the list. Since all tail items are increasing, we can perform the binary search (Lines 4-14 in Algorithm 1) that needs $O(\log m)$ time. If there is no such list, i.e., $Tail(\mathbb{L}^m_\alpha) \le a_i$, we create a new empty list $Tail(\mathbb{L}^{m+1}_\alpha)$ and insert a_i into $Tail(\mathbb{L}^{m+1}_\alpha)$.

[8]We also record the position i of each item a_i in $\mathbb{L}_\alpha$ besides the item value.
[9]Readers can skip the following paragraphs (a) and (b) if they only care about the construction steps.

According to Lemma 1, it is easy to know a_i can only be appended to the end of $\mathbb{L}_\alpha^t$, i.e., $rn_\alpha(Tail(\mathbb{L}_\alpha^t)) = a_i$ and $ln_\alpha(a_i) = Tail(\mathbb{L}_\alpha^t)$. Besides, according to Lemma 2(2), we know that $un_\alpha(a_i)$ is the rightmost item in $\mathbb{L}_\alpha^{t-1}$ which is before a_i in α, then we set $un_\alpha(a_i) = Tail(\mathbb{L}_\alpha^{t-1})$ (if exists). Analogously, we set $dn_\alpha(a_i) = Tail(\mathbb{L}_\alpha^{t+1})$ (if exists).

So far, we correctly determine the four neighbors of a_i. We can repeat the above steps until all items are inserted to $\mathbb{L}_\alpha$.

We divide the above building process into two pieces of pseudo codes. Algorithm 1 presents pseudo codes for inserting one element into the current QN-List $\mathbb{L}_\alpha$, while Algorithm 2 loops on Algorithm 1 to insert all items in α one by one to build the QN-List $\mathbb{L}_\alpha$. Initially, $\mathbb{L}_\alpha = \emptyset$. The QN-List $\mathbb{L}_\alpha$ obtained in Algorithm 2 will be called the *corresponding data structure* of α.

Algorithm 1: Insert an element into $\mathbb{L}_\alpha$

```
Input: $a_i$, an element to be inserted
Output: the updated QN-List $\mathbb{L}_\alpha$
1  Let $m = |\mathbb{L}_\alpha|$
2  Since the sequence $\{Tail(\mathbb{L}_\alpha^1), Tail(\mathbb{L}_\alpha^2),...,Tail(\mathbb{L}_\alpha^m)\}$ is increasing
     (Lemma 3), we can conduct a binary search to determine minimum $k$
     where $Tail(\mathbb{L}_\alpha^k) > a_i$.
3  if ($\mathbb{L}_\alpha^k$ exists) then
4      $a^* = Tail(\mathbb{L}_\alpha^k)$;
5      /*append $a_i$ to the list $\mathbb{L}_\alpha^k$*/
6      $rn_\alpha(a^*) = a_i$; $ln_\alpha(a_i) = a^*$;
7      If $k > 1$, then let $un_\alpha(a^*) = Tail(\mathbb{L}_\alpha^{k-1})$;
8      If $k<|\mathbb{L}_\alpha|$, then let $dn_\alpha(a^*) = Tail(\mathbb{L}_\alpha^{k+1})$;
9  else
10     Create list $\mathbb{L}_\alpha^{m+1}$ in $\mathbb{L}_\alpha$ and add $a_i$ into $\mathbb{L}_\alpha^{m+1}$
11 RETURN $\mathbb{L}_\alpha$
```

Algorithm 2: Building $\mathbb{L}_\alpha$ for a sequence $\alpha = \{a_1, ..., a_w\}$

```
Input: a sequence $\alpha = \{a_1, ..., a_w\}$
Output: the corresponding data structure $\mathbb{L}_\alpha$ of $\alpha$
1  for each item $a_i$ in $\alpha$ do
2      Call Algorithm 1 to insert $a_i$ into $\mathbb{L}_\alpha$.
3  RETURN $\mathbb{L}_\alpha$;
```

Theorem 2. *Let $\alpha = \{a_1, a_2, ..., a_w\}$ be a sequence with w items. Then we have the following:*

1. *The time complexity of Algorithm 1 is $O(\log w)$.*
2. *The time complexity of Algorithm 2 is $O(w \log w)$.*

4.4 LIS Enumeration

Let's discuss how to enumerate all LIS of sequence α based on the QN-List $\mathbb{L}_\alpha$. Consider an LIS of α : $s = \{a_{i_1}, a_{i_2},...,a_{i_m}\}$. According to Lemma 2(1), $a_{i_m} \in \mathbb{L}_\alpha^m$. In fact, the last item of each LIS must be located at the last horizontal list of $\mathbb{L}_\alpha$ and we can enumerate all LIS of α by enumerating all $|\mathbb{L}_\alpha|$ long increasing subsequence ending with items in $\mathbb{L}_\alpha^{|\mathbb{L}_\alpha|}$. For convenience, we use $MIS_\alpha(a_i)$ to denote the set of all $RL_\alpha(a_i)$ long increasing subsequences ending with a_i. Formally, $MIS_\alpha(a_i)$ is defined as follows:

$$MIS_\alpha(a_i) = \{s \mid s \in IS_\alpha(a_i) \wedge |s| = RL_\alpha(a_i)\}$$

Consider each item a_i in the last list $\mathbb{L}_\alpha^{|\mathbb{L}_\alpha|}$. We can compute all LIS of α ending with a_i by iteratively searching for predecessors of a_i in the above list from the bottom to up until reaching the first list $\mathbb{L}_\alpha^1$. This is the basic idea of our LIS enumeration algorithm.

For brevity, we virtually create a *directed acyclic graph* (DAG) to more intuitively discuss the LIS enumeration on $\mathbb{L}_\alpha$. The DAG is defined based on the predecessor relationships between items in α. Each vertex in the DAG corresponds to an item in α. A directed edge is inserted from a_i to a_j if a_j is a predecessor of a_i (a_i and a_j is also called parent and child respectively).

Definition 12. *(DAG $G(\alpha)$). Given a sequence α, the directed graph G is denoted as $G(\alpha) = (V, E)$, where the vertex set V and the edge set E are defined as follows:*

$$V = \{a_i | a_i \in \alpha\}; \quad E = \{(a_i, a_j) | a_j \text{ is a predecessor of } a_i\}$$

The $G(\alpha)$ over the sequence $\alpha = \{3, 9, 6, 2, 8, 5, 7\}$ is presented in Figure 6. We can see that each path with length $|\mathbb{L}_\alpha|$ in $G(\alpha)$ corresponds to an LIS. For example, we can find a path $a_5 = 8 \rightarrow a_3 = 6 \rightarrow a_1 = 3$, which is the reverse order of LIS $\{3,6,8\}$. Thus, we can easily design a DFS-like traverse starting from items in $\mathbb{L}_\alpha^{|\mathbb{L}_\alpha|}$ to output all path with length $|\mathbb{L}_\alpha|$ in $G(\alpha)$.

Note that we do not actually need to build the DAG in our algorithm since we can equivalently conduct the DFS-like traverse on $\mathbb{L}_\alpha$. Firstly, we can easily access all items in $\mathbb{L}_\alpha$ which are the starting vertexes of the traverse. Secondly, the key operation in the DFS-like traverse is to get all predecessors of a vertex. In fact, according to Lemma 1 which is demonstrated in Figure 5, we can find all predecessors of a_i by searching $\mathbb{L}_\alpha^{t-1}$ from $un_\alpha(a_i)$ to the left until meeting an item a^* that is not compatible with a_i. All touched items (a^* excluded) during the search are predecessors of a_i.

We construct LIS s from each item a_{i_m} in $\mathbb{L}_\alpha^m$ (i.e., the last list) as follows. a_{i_m} is first pushed into the bottom of an initially empty stack. At each iteration, the up neighbor of the top item is pushed into the stack. The algorithm continues until it pushes an item in $\mathbb{L}_\alpha^1$ into the stack and output items in the stack since this is when the stack holds an LIS. Then the algorithm starts to pop top item from the stack and push another predecessor of the current top item into stack. It is easy to see that this algorithm is very similar to depth-first search (DFS) (where the function call stack is implicitly used as the stack) and more specifically, this algorithm outputs all LIS as follows: (1) every item in $\mathbb{L}_\alpha^m$ is pushed into stack; (2) at each iteration, every predecessor (which can be scanned on a horizontal list from the up neighbor to left until discovering an incompatible item) of the current topmost item in the stack is pushed in the stack; (3) the stack content is printed when it is full (i.e., an LIS is in it).

Theorem 3. *The time complexity of our LIS enumeration algorithm is $O(OUTPUT)$, where* OUTPUT *is the total size of all LIS.*

Pseudo code for LIS enumeration is presented in Appendix C of the full version of this paper [2] .

5. MAINTENANCE

When time window slides, a_1 is deleted and a new item a_{w+1} is appended to the end of α. It is easy to see that the quadruple neighbor list maintenance consists of two operations: deletion of the first item a_1 and insertion of a_{w+1} to the end. Algorithm 1 in Section 4.3 takes care of the insertion already. Thus we only consider "deletion" in this section. The sequence $\{a_2, \cdots, a_w\}$ formed by deleting a_1 from α is denoted as α^-. We divide the discussion of the quadruple neighbor list maintenance into two parts: the horizontal update for updating left and right neighbors and the vertical update for up and down neighbors.

5.1 Horizontal Update

This section studies the horizontal update. We first introduce "k-hop up neighbor" that will be used in latter discussions.

Definition 13. *(k-Hop Up Neighbor). Let $\alpha = \{a_1, a_2, ..., a_w\}$ be a sequence and $\mathbb{L}_\alpha$ be its corresponding quadruple neighbor list. For $\forall a_i \in \alpha$, the* k-hop up neighbor *$un_\alpha^k(a_i)$ is defined as follows:*

$$un_\alpha^k(a_i) = \begin{cases} a_i & k = 0 \\ un_\alpha(un_\alpha^{k-1}(a_i)) & k \geq 1 \end{cases}$$

To better understand our method, we first illustrate the main idea and the algorithm's sketch using a running example. More analysis and algorithm details are given afterward.

Running example and intuition. Figure 8(a) shows the corresponding QN-list $\mathbb{L}_\alpha$ for the sequence α in the running example. After deleting a_1, some items in $\mathbb{L}_\alpha^t$ ($1 \leq t \leq m$) should be promoted to the above list $\mathbb{L}_\alpha^{t-1}$ and the others are still in $\mathbb{L}_\alpha^t$. The following Theorem 4 tells us how to distinguish them. In a nutshell, given an item $a \in \mathbb{L}_\alpha^t$ ($1 < t \leq m$), if its $(t-1)$-hop up neighbor is a_1 (the item to be deleted), a should be promoted to the above list; otherwise, a is still in the same list.

For example, Figure 8(a) and 8(b) show the QN-lists before and after deleting a_1. $\{a_2, a_3\}$ are in $\mathbb{L}_\alpha^2$ and their 1-hop up neighbors are a_1 (the item to be deleted), thus, they are promoted to the first list of $\mathbb{L}_{\alpha^-}$. Also, $\{a_4\}$ is in $\mathbb{L}_\alpha^3$, whose 2-hop up neighbor is also a_1. It is also promoted to $\mathbb{L}_{\alpha^-}^2$. More interesting, for each horizontal list $\mathbb{L}_\alpha^t$ ($1 \leq t \leq m$), the items that need to be promoted are on the left part of $\mathbb{L}_\alpha^t$, denoted as $Left(\mathbb{L}_\alpha^t)$, which are the shaded ones in Figure 8(a). Note that $Left(\mathbb{L}_\alpha^1) = \{a_1\}$. The right(remaining) part of $\mathbb{L}_\alpha^t$ is denoted as $Right(\mathbb{L}_\alpha^t)$. The horizontal update is to couple $Left(\mathbb{L}_\alpha^{t+1})$ with $Right(\mathbb{L}_\alpha^t)$ into a new horizontal list $\mathbb{L}_{\alpha^-}^t$. For example, $Left(\mathbb{L}_\alpha^2) = \{a_2, a_3\}$ plus $Right(\mathbb{L}_\alpha^1) = \{a_4\}$ to form $\mathbb{L}_{\alpha^-}^1 = \{a_2, a_3, a_4\}$, as shown in Figure 8(b). Furthermore, the red bold line in Figure 8(a) denotes the *separatrix* between the left and the right part, which starts from a_1. Algorithm 3 studies how to find the separatrix to divide each horizontal list $\mathbb{L}_{\alpha^-}^t$ into two parts efficiently.

Analysis and Algorithm. Lemma 4 tells us that the up neighbour relations of the two items in the same list do not cross, which is used in the proof of Theorem 4.

LEMMA 4. *Let $\alpha = \{a_1, ..., a_w\}$ be a sequence and $\mathbb{L}_\alpha$ be its corresponding quadruple neighbor list. Let m be the number of horizontal lists in $\mathbb{L}_\alpha$. Let a_i and a_j be two items in $\mathbb{L}_\alpha^t, t \geq 1$. If a_i is on the left of a_j, $un_\alpha^k(a_i) = un_\alpha^k(a_j)$ or $un_\alpha^k(a_i)$ is on the left of $un_\alpha^k(a_j)$, for every $0 \leq k < t$.*

Theorem 4. *Given a sequence $\alpha = \{a_1, a_2, \cdots, a_w\}$ and $\mathbb{L}_\alpha$. Let $m = |\mathbb{L}_\alpha|$. Let $\alpha^- = \{a_2, \cdots, a_w\}$ be obtained from α by deleting a_1. Then for any $a_i, 2 \leq i \leq m \in \mathbb{L}_\alpha^t, 1 \leq t \leq m$, we have the following:*

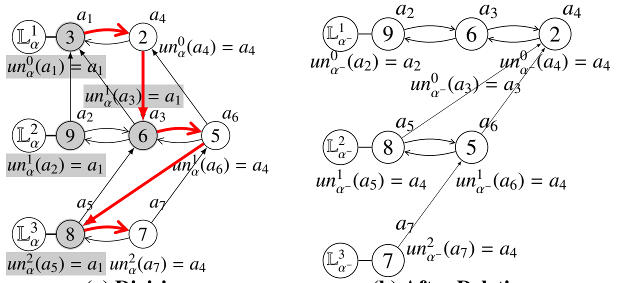
1. *If $un_\alpha^{t-1}(a_i)$ is a_1, then $RL_{\alpha^-}(a_i) = RL_\alpha(a_i) - 1$.*
2. *If $un_\alpha^{t-1}(a_i)$ is not a_1, then $RL_{\alpha^-}(a_i) = RL_\alpha(a_i)$.*

Naive method. With Theorem 4, the straightforward method to update horizontal lists is to compute $un_\alpha^{t-1}(a_i)$ for each a_i in $\mathbb{L}_\alpha^t$. If $un_\alpha^{t-1}(a_i)$ is a_1, promote a_i into $\mathbb{L}_\alpha^{t-1}$. After grouping items into the correct horizontal lists, we sort the items of each horizontal list in the decreasing order of their values. According to Theorem 4 and Lemma 1(2) (which states that the horizontal list is in decreasing order), we can easily know that the horizontal lists obtained by the above process is the same as re-building $\mathbb{L}_{\alpha^-}$ for sequence α^- (i.e., the sequence after deleting a_1).

Optimized method. For each item a_i in $\mathbb{L}_\alpha^t$ ($1 \leq t \leq m$) in the running example, we report its $(t-1)$-hop up neighbor in Figure 8a. The shaded vertices denote the items whose $(t-1)$-hop up neighbors are a_1 in $\mathbb{L}_\alpha^1$; and the others are in the white vertices. Interestingly, the two categories of items of a list form two consecutive blocks. The shaded one is on the left and the other on the right.

Let us recall Lemma 4, which says that the up neighbour relations of the two items in the same list do not cross. In fact, after deleting a_1, for each $a_i \in \mathbb{L}_\alpha^t$, if $un_\alpha^{t-1}(a_i)$ is a_1, then for any item a_j at the left side of a_i in $\mathbb{L}_\alpha^t$, $un_\alpha^{t-1}(a_i)$ is also a_1. While, if $un_\alpha^{t-1}(a_i)$ is not a_1, then for any item a_k at the right side of a_i in $\mathbb{L}_\alpha^t$, $un_\alpha^{t-1}(a_i)$ is not a_1. The two claims can be proven by Lemma 4. This is the reason why two categories of items form two consecutive blocks, as shown in Figure 8a.

After deleting a_1, we can divide each list $\mathbb{L}_\alpha^t$ into two sublists: $Left(\mathbb{L}_\alpha^t)$ and $Right(\mathbb{L}_\alpha^t)$. For any item $a_j \in Left(\mathbb{L}_\alpha^t)$, $un_\alpha^{t-1}(a_j)$ is a_1 while for any item $a_k \in Right(\mathbb{L}_\alpha^t)$, $un_\alpha^{t-1}(a_k)$ is not a_1. Instead of computing the $(t-1)$-hop up neighbor of each item, we propose an efficient algorithm (Algorithm 3) to divide each horizontal list $\mathbb{L}_\alpha^t$ into two sublists: $Left(\mathbb{L}_\alpha^t)$ and $Right(\mathbb{L}_\alpha^t)$.



(a) Division. **(b) After Deletion.**

Figure 8: Maintenance

Let's consider the division of each horizontal list of $\mathbb{L}_\alpha$. In fact, in our division algorithm, the division of $\mathbb{L}_\alpha^t$ depends on that of $\mathbb{L}_\alpha^{t-1}$. We first divide $\mathbb{L}_\alpha^1$. Apparently, $Left(\mathbb{L}_\alpha^1) = \{a_1\}$ and $Right(\mathbb{L}_\alpha^1) = \{\mathbb{L}_\alpha^1\} - \{a_1\}$. Recursively, assuming that we have finished the division of $\mathbb{L}_\alpha^t$, $1 \leq t < m$, there are three cases to divide $\mathbb{L}_\alpha^{t+1}$. Note that for each item $a_i \in Left(\mathbb{L}_\alpha^t)$, $un_\alpha^{t-1}(a_i) = a_1$; while for each item $a_i \in Right(\mathbb{L}_\alpha^t)$, $un_\alpha^{t-1}(a_i) \neq a_1$.

1. If $Right(\mathbb{L}_\alpha^t) = NULL$, for any item $a_j \in \mathbb{L}_\alpha^{t+1}$, we have $un_\alpha(a_j) \in Left(\mathbb{L}_\alpha^t)$, thus, $un_\alpha^t(a_j)$ is exactly a_1. Thus, all below lists are set to be the left part. Specifically, for any $t' > t$, we set $Left(\mathbb{L}_\alpha^{t'}) = \mathbb{L}_\alpha^{t'}$ and $Right(\mathbb{L}_\alpha^{t'}) = NULL$.
2. If $Right(\mathbb{L}_\alpha^t) \neq NULL$ and the head item of $Right(\mathbb{L}_\alpha^t)$ is a_k:
   (a) if $dn_\alpha(a_k)$ does not exist, namely, $\mathbb{L}_\alpha^{t+1}$ is empty at the time when a_k is inserted into $\mathbb{L}_\alpha^t$, then all items in $\mathbb{L}_\alpha^{t+1}$ come after a_k and their up neighbors are either a_k or item at the right side of a_k, thus, the t-hop up neighbor of each item in $\mathbb{L}_\alpha^{t+1}$ cannot be a_1. Actually, all below lists are set to be the right part. Specifically, for any $t' > t$, we set $Left(\mathbb{L}_\alpha^{t'}) = NULL$ and $Right(\mathbb{L}_\alpha^{t'}) = \mathbb{L}_\alpha^{t'}$.
   (b) if $dn_\alpha(a_k)$ exists, then $dn_\alpha(a_k)$ and items at its left side come before a_k and their up neighbors can only be at the left side of a_k (i.e., $Left(\mathbb{L}_\alpha^t)$), thus, the t-hop up neighbor of $dn_\alpha(a_k)$ or items on the left of $dn_\alpha(a_k)$ must be a_1. Besides, items at the right side of $dn_\alpha(a_k)$ come after a_k, and their up neighbors is either a_k or item at the right side of a_k, thus, the t-hop up neighbor of each item on the right of dn_α cannot be a_1. Generally, we set $Left(\mathbb{L}_\alpha^{t+1})$ as the induced sublist from the head of $\mathbb{L}_\alpha^{t+1}$ to $dn_\alpha(a_k)$(included) and set $Right(\mathbb{L}_\alpha^{t+1})$ as the remainder, namely, $Right(\mathbb{L}_\alpha^{t+1}) = \mathbb{L}_\alpha^{t+1} - Left(\mathbb{L}_\alpha^{t+1})$. We iterate the above process for the remaining lists.

Finally, for any $1 \leq t \leq m$, the left sublist $Left(\mathbb{L}_\alpha^t)$ should be promoted to the above list; and $Right(\mathbb{L}_\alpha^t)$ is still in the t-th list. Specifically, $\mathbb{L}_{\alpha^-}^t = Left(\mathbb{L}_{\alpha^-}^{t+1}) + Right(\mathbb{L}_{\alpha^-}^t)$, i.e., appending $Right(\mathbb{L}_{\alpha^-}^t)$ to $Left(\mathbb{L}_{\alpha^-}^{t+1})$ to form $\mathbb{L}_{\alpha^-}^t$. In the running example, we append $Right(\mathbb{L}_\alpha^1) = \{a_2, a_3\}$ to $Left(\mathbb{L}_\alpha^2) = \{a_4\}$ to form $\mathbb{L}_{\alpha^-}^1 = \{a_2, a_3, a_4\}$, as shown in Figure 8b.

Theorem 5. *The list that is formed by appending $Right(\mathbb{L}_\alpha^t)$ to $Left(\mathbb{L}_\alpha^{t+1})$ is monotonic decreasing from the left to the right.*

According to Theorem 4 and Lemma 2(1), we can prove that the list formed by appending $Right(\mathbb{L}_\alpha^t)$ to $Left(\mathbb{L}_\alpha^{t+1})$, denoted as L, contains the same set of items as $\mathbb{L}_{\alpha^-}^t$ does. Besides, according to Lemma 1(2) and Theorem 5, both L and $\mathbb{L}_{\alpha^-}^t$ are monotonic decreasing, thus, we can know that L is equivalent to $\mathbb{L}_{\alpha^-}^t$ and we can derive that the horizontal list adjustment method is correct.

Algorithm 3: Divide each horizontal list after deletion

```
Input: $\mathbb{L}_\alpha$: the quadruple neighbor list for $\alpha$.
Input: $a_1$: the item to be deleted.
Output: $Left(\mathbb{L}^t_\alpha)$ and $Right(\mathbb{L}^t_\alpha)$ for each $\mathbb{L}^t_\alpha$, $1 \le t \le m$.
$m = |\mathbb{L}_\alpha|$
$Left(\mathbb{L}^1_\alpha) = \{a_1\}$
$Right(\mathbb{L}^1_\alpha) = \mathbb{L}^1_\alpha / \{a_1\}$
for $t \leftarrow 1$ to $m - 1$ do
    if $Right(\mathbb{L}^t_\alpha) = NULL$ then
        for $t' \leftarrow (t + 1)$ to $m$ do
            $Left(\mathbb{L}^{t'}_\alpha) = \mathbb{L}^{t'}_\alpha$
            $Right(\mathbb{L}^{t'}_\alpha) = NULL$
        RETURN
    $a_k = Head(Right(\mathbb{L}^t_\alpha))$
    if $dn_\alpha(a_k) = NULL$ then
        for $t' \leftarrow (t + 1)$ to $m$ do
            $Left(\mathbb{L}^{t'}_\alpha) = NULL$
            $Right(\mathbb{L}^{t'}_\alpha) = \mathbb{L}^{t'}_\alpha$
        RETURN
    Set $Left(\mathbb{L}^{t+1}_\alpha)$ to be the part at the left side of $dn_\alpha(a_k)$ in $\mathbb{L}^{t+1}_\alpha$,
      including $dn_\alpha(a_k)$ itself.
    $Right(\mathbb{L}^{t+1}_\alpha) = \mathbb{L}^{t+1}_\alpha - Left(\mathbb{L}^{t+1}_\alpha)$
RETURN
```

5.2 Vertical Update

Besides adjusting the horizontal lists, we also need to update the vertical neighbor relationship in the quadruple neighbor list to finish the transformation from $\mathbb{L}_\alpha$ to $\mathbb{L}_{\alpha^-}$. Before presenting our method, we recall Lemma 2(2), which says, for item $a_i \in \mathbb{L}^t_\alpha$, $un_\alpha(a_i)$(if exists) is the rightmost item in $\mathbb{L}^{t-1}_\alpha$ who is before a_i in sequence α; while, $dn_\alpha(a_i)$(if exists) is the rightmost item in $\mathbb{L}^{t+1}_\alpha$ who is before a_i in sequence α.

Running example and intuition. Let us recall Figure 8. After adjusting the horizontal lists, we need to handle updates of vertical neighbors. The following Lemma 6 tells us which vertical relations will remain when transforming $\mathbb{L}_\alpha$ into $\mathbb{L}_{\alpha^-}$. Generally, when we promote $Left(\mathbb{L}^t_\alpha)$ to the above level, we need to change their up neighbors but not down neighbors. While, $Right(\mathbb{L}^t_\alpha)$ is still in the same level after the horizontal update. We need to change their down neighbors but not up neighbors.

For example, $Left(\mathbb{L}^3_\alpha) = \{a_5\}$ is promoted to the $\mathbb{L}^2_{\alpha^-}$. In $\mathbb{L}_\alpha$, $un_\alpha(a_5)$ is a_3, but we change it to $un_{\alpha^-}(a_5) = a_4$, i.e., the rightmost item in $\mathbb{L}^1_{\alpha^-}$ who is before a_5 in sequence α^-. Analogously, $Right(\mathbb{L}^2_\alpha) = \{a_6\}$ is still at the second level of $\mathbb{L}_{\alpha^-}$. $dn_\alpha(a_6)$ is a_5, but we change it to null (i.e., $dn_{\alpha^-}(a_6) = null$), since there is no item in $\mathbb{L}^3_{\alpha^-}$ who is before a_6. We give the formal analysis and algorithm description of the vertical update as follows.

Analysis and Algorithm.

LEMMA 5. *Given a sequence α and $\mathbb{L}_\alpha$, for any $1 \le t \le m$:*

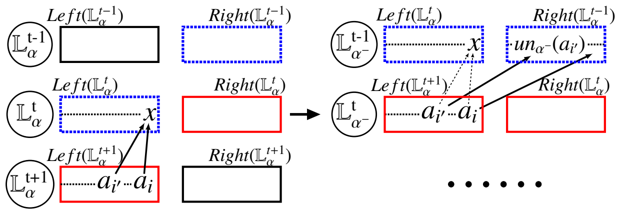
1. *$\forall a_i \in Left(\mathbb{L}^t_\alpha)$, $dn_\alpha(a_i)$ (if exists) $\in Left(\mathbb{L}^{t+1}_\alpha)$.*
2. *$\forall a_i \in Right(\mathbb{L}^{t+1}_\alpha)$, $un_\alpha(a_i)$ (if exists) $\in Right(\mathbb{L}^t_\alpha)$.*

LEMMA 6. *Let $\alpha = \{a_1, a_2, \cdots, a_w\}$ be a sequence. Let $\mathbb{L}_\alpha$ be its corresponding quadruple neighbor list and m be the total number of horizontal lists in $\mathbb{L}_\alpha$. Let $\alpha^- = \{a_2, \cdots, a_w\}$ be obtained from α by deleting a_1. Consider an item $a_i \in \mathbb{L}^t_{\alpha^-}$, where $1 \le t \le m$. According to the horizontal list adjustment, there are two cases for a_i: a_i is from $Left(\mathbb{L}^{t+1}_\alpha)$ or a_i is from $Right(\mathbb{L}^t_\alpha)$. Then, the following claims hold:*

1. *Assuming a_i is from $Left(\mathbb{L}^{t+1}_\alpha)$*
   (a) $dn_{\alpha^-}(a_i) = dn_\alpha(a_i)$ (i.e., the down neighbor remains).
   (b) Let x be the rightmost item of $Left(\mathbb{L}^t_\alpha)$. If $un_\alpha(a_i) \ne x$, then $un_{\alpha^-}(a_i) = un_\alpha(a_i)$ (i.e., the up neighbor remains).
2. *Assuming a_i is from $Right(\mathbb{L}^t_\alpha)$*
   (a) $un_{\alpha^-}(a_i) = un_\alpha(a_i)$ (i.e., the up neighbor do not change).
   (b) Let y be the rightmost item of $Left(\mathbb{L}^{t+1}_\alpha)$. If $dn_\alpha(a_i) \ne y$, $dn_{\alpha^-}(a_i) = dn_\alpha(a_i)$ (i.e., the down neighbor remains)

With Lemma 6, for an item $a_i \in \mathbb{L}^t_{\alpha^-}$, there are only two cases that we need to update the vertical neighbor relations of a_i.

1. Case 1: a_i is from $Left(\mathbb{L}^{t+1}_\alpha)$. Let x be the rightmost item of $Left(\mathbb{L}^t_\alpha)$. We need to update the *up neighbor* of a_i in $\mathbb{L}_{\alpha^-}$ if $un_\alpha(a_i) = x$. Figure 9 demonstrates this case.
2. Case 2: a_i is from $Right(\mathbb{L}^t_\alpha)$. Let y be the rightmost item of $Left(\mathbb{L}^{t+1}_\alpha)$. We need to update the *down neighbor* of a_i in $\mathbb{L}_{\alpha^-}$ if $dn_\alpha(a_i) = y$. Figure 10 demonstrates this case.



$Left(\mathbb{L}^{t-1}_\alpha)$
$Right(\mathbb{L}^{t-1}_\alpha)$
$\mathbb{L}^{t-1}_\alpha$
$Left(\mathbb{L}^t_\alpha)$
$Right(\mathbb{L}^t_\alpha)$
$\mathbb{L}^t_\alpha$
x
$Left(\mathbb{L}^{t+1}_\alpha)$
$Right(\mathbb{L}^{t+1}_\alpha)$
$\mathbb{L}^{t+1}_\alpha$
$a_{i'}$
a_i
$Left(\mathbb{L}^t_\alpha)$
$Right(\mathbb{L}^{t-1}_\alpha)$
$\mathbb{L}^{t-1}_{\alpha^-}$
x
$un_{\alpha^-}(a_{i'})$
$Left(\mathbb{L}^{t+1}_\alpha)$
$Right(\mathbb{L}^t_\alpha)$
$\mathbb{L}^t_{\alpha^-}$
$a_{i'}$
a_i

(a) Before Deletion **(b) After Deletion**

Figure 9: Case 1: updating up neighbors

Case 1: Consider all items in $Left(\mathbb{L}^{t+1}_\alpha)$. According to the horizontal adjustment, $Left(\mathbb{L}^{t+1}_\alpha)$ will be promoted into the list $\mathbb{L}^t_{\alpha^-}$.

Let a_i be the rightmost item of $Left(\mathbb{L}^{t+1}_\alpha)$ and $x = Tail(Left(\mathbb{L}^t_\alpha))$, namely, x is the rightmost item in $Left(\mathbb{L}^t_\alpha)$. According to Lemma 6(1.b), if $un_\alpha(a_i) \ne x$, then $un_\alpha(a_i) = un_{\alpha^-}(a_i)$. It is easy to prove that: If $un_\alpha(a_i) \ne x$ then $un_\alpha(a_j) \ne x$, where a_j is on the left of a_i in $Left(\mathbb{L}^{t+1}_\alpha)$. In other words, all items in $Left(\mathbb{L}^{t+1}_\alpha)$ do not change the vertical relations (see Lines 6-8 in Algorithm 4).

Now, we consider the case that $un_\alpha(a_i) = x$ (Lines 8-8 in Algorithm 4). Then can scan $Left(\mathbb{L}^{t+1}_\alpha)$ from a_i to the left until finding the leftmost item $a_{i'}$, where $un_\alpha(a_{i'})$ is also x. The up neighbors of the items in the consecutive block from $a_{i'}$ to a_i (included both) are all x in $\mathbb{L}_\alpha$ (note that x is the rightmost item in $Left(\mathbb{L}^t_\alpha)$), as shown in Figure 9(a). These items' up neighbors need to be adjusted in $\mathbb{L}_{\alpha^-}$. We work as follows: First, we adjust the up neighbor of $a_{i'}$ in $\mathbb{L}_{\alpha^-}$. Initially, we set $a^* = un_\alpha(a_{i'}) = x$. Then, we move a^* to the right step by step in $\mathbb{L}^{t-1}_{\alpha^-}$ until finding the rightmost item whose position is before $a_{i'}$ in sequence α^-. Finally, we set $un_{\alpha^-}(a_{i'}) = a^*$ (see Lines 14 in Algorithm 4).

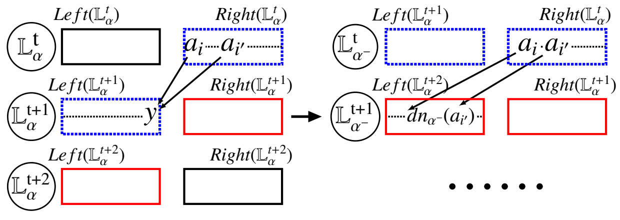
In the running example, when deleting a_1 in Figure 8a, $Left(\mathbb{L}^3_\alpha) = \{a_5\}$, and $un_\alpha(a_5)$ is exactly the tail item a_3 of $Left(\mathbb{L}^2_\alpha)$, since $\mathbb{L}^1_{\alpha^-}$ is $\{a_2 = 9, a_3 = 6, a_4 = 2\}$, formed by appending $Right(\mathbb{L}^1_\alpha)(\{a_2 = 9, a_3 = 6\})$ to $Left(\mathbb{L}^2_\alpha)$ ($\{a_4 = 2\}$), and a_4 is the rightmost item in $\mathbb{L}^1_{\alpha^-}$ who is before a_5 in α^-, then we set $un_{\alpha^-}(a_5)$ as $a_4 = 2$, as shown in Figure 8b.

Iteratively, we consider the items on the right of $a_{i'}$. Actually, the adjustment of the next item's up neighbor can begin from the current position of a^* (Line 13). It is straightforward to know the time complexity of Algorithm 4 is $O(|\mathbb{L}^{t-1}_{\alpha^-}|)$, since each item in $\mathbb{L}^{t-1}_{\alpha^-}$ is scanned at most one time.

Case 2: Consider all items in $Right(\mathbb{L}^t_\alpha)$. According to the horizontal adjustment, the down neighbors of items in $Right(\mathbb{L}^t_\alpha)$ are the tail item (i.e., the rightmost item) of $Left(\mathbb{L}^{t+1}_\alpha)$ or items in $Right(\mathbb{L}^{t+1}_\alpha)$.

Actually, Case 2 is symmetric to Case 1. We highlight some important steps as follows. Let a_i be the leftmost item in $Right(\mathbb{L}^t_\alpha)$ and let y be $Tail(Left(\mathbb{L}^{t+1}_\alpha))$, namely, y is the rightmost item in

$Left(\mathbb{L}_\alpha^{t+1})$. Obviously, $dn_\alpha(a_i) = y$ (Algorithm 3). Then we scan $Right(\mathbb{L}_\alpha^t)$ from a_i to the rightmost item $a_{i'}$ where $dn_\alpha(a_{i'})$ is y. The up neighbors of the items in the consecutive block from a_i to $a_{i'}$ (included both) are all y (see Figure 10(a)). Items on the right of $a_{i'}$ need no changes in their down neighbors, since their down neighbors in $\mathbb{L}_\alpha$ are not y (see Lemma 6(2.b)).


$Left(\mathbb{L}_\alpha^t)$
$Right(\mathbb{L}_\alpha^t)$
$\mathbb{L}_\alpha^t$
$a_i \cdots a_{i'}$
$Left(\mathbb{L}_\alpha^{t+1})$
$Right(\mathbb{L}_\alpha^{t+1})$
$\mathbb{L}_\alpha^{t+1}$
y
$Left(\mathbb{L}_\alpha^{t+2})$
$Right(\mathbb{L}_\alpha^{t+2})$
$\mathbb{L}_\alpha^{t+2}$
$Left(\mathbb{L}_\alpha^{t+1})$
$Right(\mathbb{L}_\alpha^t)$
$\mathbb{L}_{\alpha^-}^t$
$a_i \cdot a_{i'}$
$Left(\mathbb{L}_\alpha^{t+2})$
$Right(\mathbb{L}_\alpha^{t+1})$
$\mathbb{L}_{\alpha^-}^{t+1}$
$dn_{\alpha^-}(a_{i'})$
$\cdots\cdots$

(a) Before Deletion **(b) After Deletion**

Figure 10: Case 2: updating down neighbors

Algorithm 4: Update up neighbors of items in $Left(\mathbb{L}_\alpha^{t+1})$

```
   Input: $Left(\mathbb{L}_\alpha^{t+1})$)
   Output: the updated $Left(\mathbb{L}_\alpha^{t+1})$
1  if $Left(\mathbb{L}_\alpha^{t+1}) = NULL$ then
2      RETURN
3  if $t = 1$ then
4      Set $un_{\alpha^-}(a_i) = NULL$ for each $a_i \in Left(\mathbb{L}_\alpha^{t+1})$
5      RETURN
6  Let $a_i = Tail(Left(\mathbb{L}_\alpha^{t+1}))$ and $x = Tail(Left(\mathbb{L}_\alpha^t))$
7  if $un_\alpha(a_i)$ is not $x$ then
8      RETURN
9  Scan $Left(\mathbb{L}_\alpha^{t+1})$ from right to left and find the leftmost item whose up
     neighbor is $x$, denoted as $a_{i'}$
10 $a^* = x$
11 while $a_{i'} \geq a_i$ do
12     while $rn_{\alpha^-}(a^*)$ is before $a_{i'}$ do
13         $a^* = rn_{\alpha^-}(a^*)$
14     $un_{\alpha^-}(a_{i'}) = a^*$
15     $a_{i'} = rn_{\alpha^-}(a_{i'})$
16 RETURN
```

Algorithm 5: Update down neighbors of items in $Right(\mathbb{L}_\alpha^t)$

```
   Input: $Right(\mathbb{L}_\alpha^t)$)
   Output: the updated $Right(\mathbb{L}_\alpha^t)$
1  if $t \geq m - 1$ OR $Right(\mathbb{L}_\alpha^t) = NULL$ then
2      RETURN
3  Let $a_i = Head(Right(\mathbb{L}_\alpha^t))$ and $y = dn_\alpha(a_i)$
4  Scan $Right(\mathbb{L}_\alpha^t)$ from left to right and find the rightmost item whose
     down neighbor is $y$, denoted as $a_{i'}$
5  $a^* = Tail(Left(\mathbb{L}_\alpha^{t+1}))$
6  while $a_{i'} \leq a_i$ do
7      if $a^* = NULL$ OR $a^*$ is before $a_{i'}$ then
8          $dn_{\alpha^-}(a_{i'}) = a^*$
9          $a_{i'} = ln_{\alpha^-}(a_{i'})$
10     else
11         $a^* = ln_{\alpha^-}(a^*)$
12 RETURN
```

We only consider the consecutive block from a_i to $a_{i'}$ (see Figure 10) as follows. First, we adjust the down neighbor of $a_{i'}$ in $\mathbb{L}_{\alpha^-}$. Initially, we set $a^* = Tail(Left(\mathbb{L}_\alpha^{t+1}))$, i.e., the rightmost item of $Left(\mathbb{L}_\alpha^{t+1})$. Then, we move a^* to the left step by step in $\mathbb{L}_{\alpha^-}^{t+1}$ until finding the rightmost item whose position is before $a_{i'}$. Finally, we set $dn_{\alpha^-}(a_{i'}) = a^*$ (see Lines 8 in Algorithm 5).

In the running example, when deleting $a_1 = 3$, $Right(\mathbb{L}_\alpha^1)$ is $\{a_4 = 2\}$ whose head item is a_4. And $dn_\alpha(a_4)$ is $a_3 = 6$ that is the tail item of $Left(\mathbb{L}_\alpha^2)$. Then, initially, we set $dn_{\alpha^-}(a_4)$ as the tail item of $Left(\mathbb{L}_\alpha^3)$, namely, $dn_{\alpha^-}(a_4) = a_5$ and scan $\mathbb{L}_{\alpha^-}^2$ from the right to the left until finding a rightmost item who is before a_4 in α^-. Since there is no such item in $\mathbb{L}_{\alpha^-}^2$, we set $dn_{\alpha^-}(a_4)$ as $NULL$.

Iteratively, we consider the items on the left of $a_{i'}$. Actually, the adjustment of the down neighbor can begin from the current position of a^* (Line 11 in Algorithm 5). Thus, the time complexity of Algorithm 5 is $O(|\mathbb{L}_{\alpha^-}^{t+1}|)$, since $\mathbb{L}_{\alpha^-}^{t+1}$ is scanned at most twice.

5.3 Putting It All Together

Finally, we can see that solution to handle the deletion of the head item a_1 in sequence α consists two main phrase. The first phrase is to divides each list $\mathbb{L}_\alpha^t$ $(1 \leq t \leq m)$ using Algorithm 3 and then finishes the horizontal update by appending $Right(\mathbb{L}_\alpha^t)$ to $Left(\mathbb{L}_\alpha^{t+1})$. In the second phrase, we can call Algorithms 4 and 5 for vertical update. Pseudo codes of algorithm handling deletion are presented in Appendix D of the full version of this paper [2] .

Theorem 6. *The time complexity of our deletion algorithm is $O(w)$, where w denotes the time window size.*

6. COMPUTING LIS WITH CONSTRAINTS

As noted earlier in Section 1, some applications are more interested in computing LIS with certain constraints. In this section, we consider four kinds of constraints (maximum/minimum weight/gap) that are defined in Section 3.

In Section 4.4, we define the DAG (Definition 12) based on the *predecessor* (Definition 8). Each length-m path in DAG denotes a LIS. Considering the equivalence between DAG and $\mathbb{L}_\alpha$, we illustrate our algorithm using DAG for the ease of the presentation. These algorithm steps can be easily mapped to those in $\mathbb{L}_\alpha$. According to Lemma 1(2), items in $\mathbb{L}_\alpha^t$ $(1 \leq t \leq m)$ decrease from the left to the right. Thus, the leftmost length-m path in DAG denotes the LIS with the maximum weight; while, the rightmost length-m path denotes the LIS with the minimum weight. Formally, we define the *leftmost child* as follows.

Definition 14. ***(Leftmost child)**. Given an item $a_i \in \mathbb{L}_\alpha^t$ $(1 \leq t \leq m)$, the* leftmost child *of a_i, denoted as $lm_\alpha(a_i)$, is the* leftmost *predecessor (see Definition 8) of a_i in $\mathbb{L}_\alpha^{t-1}$.*

Recall Figure 6. a_3 is the leftmost child of a_7, denoted as $a_3 = lm_\alpha(a_7)$. Similar to the recursive definition of k-hop up neighbor $un_\alpha^k(a_i)$ for a_i, we recursively define $lm_\alpha^k(a_i) = lm_\alpha(lm_\alpha^{k-1}(a_i))$ $(k \geq 1)$ for any $k < t$, where $lm_\alpha^0(a_i) = a_i$. Obviously, given an item $a_i \in \mathbb{L}_\alpha^m$ (i.e., the last list), $(lm_\alpha^{m-1}(a_i), lm_\alpha^{m-2}(a_i), ..., lm_\alpha^0(a_i) = a_i)$ forms the leftmost path ending with a_i in the DAG. It is easy to know that the leftmost path $(lm_\alpha^{m-1}(a_i), lm_\alpha^{m-2}(a_i), ..., lm_\alpha^0(a_i) = a_i)$ in the DAG is the LIS ending with a_i with maximum weight and minimum gap; while the rightmost path $(un_\alpha^{m-1}(a_i), un_\alpha^{m-2}(a_i), ..., un_\alpha^0(a_i) = a_i)$ in the DAG is the LIS ending with a_i with minimum weight and maximum gap. Formally, we have the following theorem that is the central to our constraint-based LIS computation.

Theorem 7. *Given a sequence $\alpha = \{a_1, ..., a_w\}$ and $\mathbb{L}_\alpha$. Let $m = |\mathbb{L}_\alpha|$ and DAG G_α be the corresponding DAG created from $\mathbb{L}_\alpha$.*

1. *Given a_i, $a_j \in \mathbb{L}_\alpha^t$ where $a_i < a_j$. then for every $1 \leq k < t$, $lm_\alpha^k(a_i)$, $lm_\alpha^k(a_j)$, $un_\alpha^k(a_i)$ and $un_\alpha^k(a_j)$ are all in $\mathbb{L}_\alpha^{t-k}$, and $lm_\alpha^k(a_i) \leq lm_\alpha^k(a_j)$, $un_\alpha^k(a_i) \leq un_\alpha^k(a_j)$.*
2. *Given $a_i \in \mathbb{L}_\alpha^t$. Consider an LIS β ending with a_i: $\beta = \{a_{i_{t-1}}, \cdots, a_{i_k}, \cdots, a_{i_0} = a_i\}$. Then $\forall\ k \in [0, t-1]$, $lm_\alpha^k(a_i)$, a_{i_k}, and $un_\alpha^k(a_i)$ are all in $\mathbb{L}_\alpha^{t-k}$. Also $lm_\alpha^k(a_i) \geq a_{i_k} \geq un_\alpha^k(a_i)$.*
3. *Given $a_i \in \mathbb{L}_\alpha^m$ (the last list). Among all LIS ending with a_i, $\{lm_\alpha^{m-1}(a_i), \cdots, lm_\alpha^0(a_i)\}$ has maximum weight and minimum gap, while $\{un_\alpha^{m-1}(a_i), \cdots, un_\alpha^0(a_i)\}$ has the minimum weight and maximum gap.*
4. *Let a_h^m and a_t^m be the head and tail of $\mathbb{L}_\alpha^m$ respectively. Then the LIS $\{lm_\alpha^{m-1}(a_h^m), \cdots, lm_\alpha^0(a_h^m)\}$ has the maximum weight. The LIS $\{un_\alpha^{m-1}(a_t^m), \cdots, un_\alpha^0(a_t^m)\}$ has the minimum weight.*

LIS with maximum/minimum weight. Based on Theorem 7(4), we can design algorithms to compute the unique LIS with maximum weight and the unique LIS with minimum weight, respectively (Pseudo codes are in Appendix F of the full version [2]) . Generally speaking, it searches for the leftmost path and the rightmost path in the DAG and both algorithms cost $O(w)$ time.

LIS with maximum/minimum gap. For the maximum gap (minimum gap, respectively) problem, there may be numerous LIS with maximum gap (minimum gap, respectively). In the running example, LIS with the maximum gap are $\{a_1 = 3, a_3 = 6, a_5 = 8\}$ and $\{a_4 = 2, a_6 = 5, a_7 = 7\}$; while LIS with the minimum gap are $\{a_1 = 3, a_6 = 5, a_7 = 7\}$ and $\{a_1 = 3, a_3 = 6, a_7 = 7\}$, as shown in Figure 1. According to Theorem 7, it is obviously that if $\beta = \{a_{i_{m-1}}, a_{i_{m-2}}, ..., a_{i_0}\}$ is a LIS with the maximum gap, $a_{i_{m-1}} = un_\alpha^{m-1}(a_{i_0})$; while, if β is a LIS with the minimum gap, $a_{i_{m-1}} = lm_\alpha^{m-1}(a_{i_0})$. Note that, there may be multiple LIS sharing the same head and tail items. For example, $\{a_1 = 3, a_6 = 5, a_7 = 7\}$ and $\{a_1 = 3, a_3 = 6, a_7 = 7\}$ are both LIS with minimum gap, but they share the same head and tail items. Since computing LIS with the minimum gap is analogous to LIS with the maximum gap, we only consider LIS with the maximum gap as follows.

For the maximum gap problem, one could compute $\{un_\alpha^{m-1}(a_i), \cdots, un_\alpha^0(a_i)\}$ for all $a_i \in \mathbb{L}_\alpha^m$ first, and then figure out the maximum gap θ_{max} . We design a sweeping algorithm with $O(w)$ time to compute $un_\alpha^{t-1}(a_i)$ for each item $a_i \in \mathbb{L}_\alpha^t$, $1 \le t \le m$. We will return back to the sweeping algorithm at the end of this subsection. Here, we assume that $un_\alpha^{m-1}(a_i)$ has been computed for each item $a_i \in \mathbb{L}_\alpha$.

Assume that $a_i - un_\alpha^{m-1}(a_i) = \theta_{max}$ for some $a_i \in \mathbb{L}_\alpha^m$. To enumerate LIS starting with $un^{m-1}(a_i)$ ending with a_i, we only need to slightly modify the LIS enumeration algorithm : Initially, we push a_i into the stack. If $a_j \in \mathbb{L}_\alpha^t$ is a predecessor of top element in the stack, we push a_j into the stack *if and only if* $un_\alpha^{t-1}(a_j) = un_\alpha^{m-1}(a_i)$.

To compute $un_\alpha^{t-1}(a_i)$ for each $a_i \in \mathbb{L}_\alpha^t$, $1 \le t \le m$, we design a sweeping algorithm from $\mathbb{L}_\alpha^2$ to $\mathbb{L}_\alpha^m$ and once we figure out $un_\alpha^{t-1}(a_j)$ for each a_j in $\mathbb{L}_\alpha^t$, then for any $a_i \in \mathbb{L}_\alpha^{t+1}$, $un_\alpha^t(a_i)$ is $un_\alpha^{t-1}(un_\alpha(a_i))$. Apparently, this sweeping algorithm takes $\Theta(w)$ time.

Theorem 8. *The time complexity of our algorithm for LIS with maximum gap is* $O(w+\text{OUTPUT})$*, where w denotes the window size and* OUTPUT *is the total length of all LIS with maximal gap.*

Computing LIS with minimum gap is analogous to that of LIS with maximum gap by computing $lm_\alpha^{t-1}(a_i)$ instead of $un_\alpha^{t-1}(a_i)$. We also design a sweeping algorithm from $\mathbb{L}_\alpha^2$ to $\mathbb{L}_\alpha^m$ to figure out $lm_\alpha^{t-1}(a_i)$ for each $a_i \in \mathbb{L}_\alpha^t$, $1 \le t \le m$ (Pseudo codes are presented in the Appendix E of the full version of this paper [2]) .

7. COMPARATIVE STUDY

To the best of our knowledge, there is no existing work that studies both LIS enumeration and LIS with constraints in the data stream model. We compare our method with five related algorithms, four of which are state-of-the-art LIS algorithms, i.e., LISSET [6], MHLIS [22], VARIANT [7] and LISone [3], and the last one is the classical dynamic program (DP) algorithm. None of them covers either the same computing model or the same computing task with our approach. Table 2 summarizes the differences between our approach with other comparative ones.

LISSET [6] is the only one which proposed LIS enumeration in the context of "stream model". It enumerates all LIS in each sliding window but it fails to compute LIS with different constraints, such as LIS with extreme gaps and LIS with extreme weights. To enable the comparison in constraint-based LIS, we first compute all LIS followed by filtering using constraints to figure out constraint-based LIS, which is denoted as "LISSET-Post" in our experiments.

MHLIS [22] is to find LIS with the minimum gap but it does not work in the context of data stream model. The data structure in MHLIS does not consider the maintenance issue. To enable the comparison, we implement two streaming version of MHLIS: **MHLIS+Rebuild** and **MHLIS+Ins/Del** where MHLIS+Rebuild is to re-compute LIS from scratch in each time window and MHLIS+Ins/Del is to apply our update method in MHLIS.

A family of static algorithms was proposed in [7] including LIS of minimal/maximal weight/gap (denoted as **VARIANT**). For the comparison, we implement two stream version of VARIANT: **VARIANT+Rebuild** and **VARIANT+Ins/Del** where VARIANT+Rebuild is to re-compute LIS from scratch in each time window and VARIANT+Ins/Del is to apply our update method in VARIANT.

We include the classical dynamic programming (denoted as **DP**) algorithm in the comparative study. The standard DP algorithm only computes the LIS length and a single LIS. To enumerate all LIS, we save all predecessors of each item when determining the maximum length of the increasing subsequence ending with it.

LISone[3] computed LIS length and output an LIS in the sliding model. They maintained the first row of Young's Tableaux when update happened. The length of the first row is exactly the LIS length of the sequence in the window.

Methods	Stream Model	LIS Enumeration	LIS with extreme weight	LIS with extreme gap	LIS length
Our Method	✓	✓	✓	✓	✓
LISSET	✓	✓	✗	✗	✓
MHLIS	✗	✗	✗	✓	✓
VARIANT	✗	✗	✓	✓	✓
DP	✗	✓	✗	✗	✓
LISone	✓	✗	✗	✗	✓

Table 2: Compaison on supported computing task

7.1 Theoretical Analysis

Data Structure Comparison. We compare the space, construction time and update time of our data structure against those of other works (Table 4[10]) . Note that the data structures in the comparative approaches cannot support all LIS-related problems, while our data structure can support both LIS enumeration and constraint-based LIS problems (Table 2) in a uniform manner.

Since MHLIS, VARIANT or DP does not address data structure maintenance issue, we re-construct the data structure in each time window and the time complexity of the maintenance in MHLIS, VARIANT and DP are the same with their construction time.

Table 4 shows that our approach is better or not worse than any comparative work on any metric. Our data structure is better than LISSET on both space and the construction time complexity. Furthermore, the insertion time $O(\log w)$ in our method is also better than the time complexity $O(w)$ in LISSET. Also, none of MHLIS, VARIANT or DP addresses the data structure update issue. Thus, they need $O(w \log w)$ ($O(w^2)$ for DP) time to re-build data structure in each time window. Obviously, ours is better than theirs.

Online Query Algorithm Comparison. Table 3 shows online query time complexities of different approaches. The online query response time in the data stream model consists of online query time and the update time. Since the data structure maintenance time has been presented in Table 4, we only show the online query algorithm' time complexities in Table 3. We can see that, our online query time complexities are the same with the comparative ones. However, the data structure update time complexity in our method is better than others. Therefore, our overall query response time is better than the comparative ones from the theoretical perspective.

[10]The time complexities in Table 4 are based on the worst case analysis. We also studies the time complexity of our method over sorted sequence in Appendix G of the full version of this paper [2] .

Method	LIS Enumeration	LIS with max Weight	LIS with min Weight	LIS with max Gap	LIS with min Gap	LIS Length
Our Method	$O(OUTPUT)$	$O(OUTPUT)$	$O(OUTPUT)$	$O(w+OUTPUT)$	$O(w+OUTPUT)$	$O(OUTPUT)$
LISSET	$O(OUTPUT)$	–	–	–	–	$O(OUTPUT)$
MHLIS	–	–	–	–	$O(w+OUTPUT)$	$O(OUTPUT)$
VARIANT	–	$O(OUTPUT)$	$O(OUTPUT)$	$O(w+OUTPUT)$	$O(w+OUTPUT)$	$O(OUTPUT)$
DP	$O(OUTPUT)$	–	–	–	–	$O(OUTPUT)$
LISone	–	–	–	–	–	$O(OUTPUT)$

Table 3: Theoretical Comparison on Online Query

Methods	Space Complexity	Time Complexity		
		Construction	Insert	Delete
Our Method	$O(w)$	$O(w\log w)$	$O(\log w)$	$O(w)$
LISSET	$O(w^2)$	$O(w^2)$	$O(w)$	$O(w)$
MHLIS	$O(w)$	$O(w\log w)$	–	–
VARIANT	$O(w)$	$O(w\log w)$	–	–
DP	$O(w^2)$	$O(w^2)$	–	–
LISone	$O(w^2)$	$O(w^2)$	$O(w)$	$O(w)$

Table 4: Data Structure Comparison

7.2 Experimental Evaluation

We evaluate our solution against the comparative approaches. All methods, including comparative methods, are implemented by C++ and compiled by g++(5.2.0) under default settings. Each comparative method are implemented according the corresponding paper with our best effort. The experiments are conducted in Window 8.1 (Intel(R) i7-4790 3.6GHz, 8G). All codes, including those for comparative methods are provided in Github [1].

Dataset. We use four datasets in our experiments: real-world stock data, gene sequence datasets, power usage data and synthetic data. The stock data is about the historical open prices of Microsoft Cooperation in the past two decades[11], up to 7400 days. The gene datasets is a sequence of 4,525 matching positions, which are computed over the BLAST output of mRNA sequences[12] against a gene dataset[13] according to the process in [24]. The power usage dataset[14] is a public power demand dataset used in [14]. It contains 35,040 power usage value. The synthetic dataset [15] is a time series benchmark [15] that contains one million data points(See [15] for the details of data generation). Due to the space limits, we only present the experimental results over stock dataset in this section and the counterparts over the other three datasets are available in Appendix A of the full version of this paper [2] .

Data Structure Comparison. Evaluation of the data structures focuses on three aspects: space, construction time and update time.

The **space cost** of each method is presented in Figure 11a. Our method costs much less memory than LISSET, DP and LISone while slightly more than that of MHLIS and VARIANT, which results from the extra cost in our QN-List to support efficient maintenance and computing LIS with constraints. Note that none of the comparative methods can support both LIS enumeration and LIS with constraints; but our QN-List can support all these LIS-related problems in a uniform manner (see Table 2).

We **construct** each data structure five times and present their average constuction time in Figure 11b. Similarly, our method runs much faster than that of LISSET, DP and LISone, since our construction time is linear but LISSET , DP and LISone have the square time complexity (see Table 4). Our construction time is slightly slower than VARIANT and faster than MHLIS, since they have the same construction time complexity (Table 4).

None of MHLIS, VARIANT or DP addresses **maintenance** issue. To enable comparison, we implement two stream versions of MHLIS and VARIANT. The first is to rebuild the data structure in each time window(MHLIS+Rebuild, VARIANT+Rebuild). The second is to apply our update idea into MHLIS and VARIANT

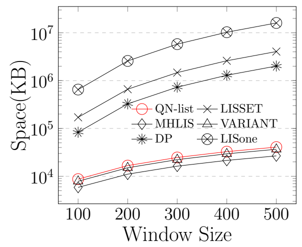
[11]http://finance.yahoo.com/q/hp?s=MSFT&d=8&e=13&f=2015&g=d&a=2&b=13&c=1986&z=66&y=66

[12]ftp://ftp.ncbi.nih.gov/refseq/B_taurus/mRNA_Prot/

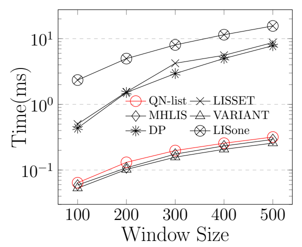
[13]ftp://ftp.ncbi.nlm.nih.gov/genbank/

[14]http://www.cs.ucr.edu/~eamonn/discords/power_data.txt

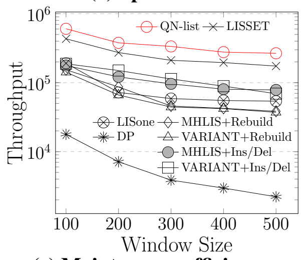
[15]https://archive.ics.uci.edu/ml/datasets/Pseudo+Periodic+Synthetic+Time+Series



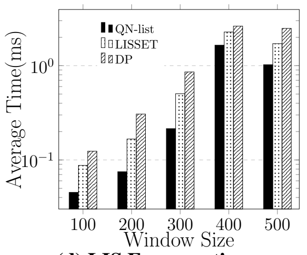
(a) Space cost



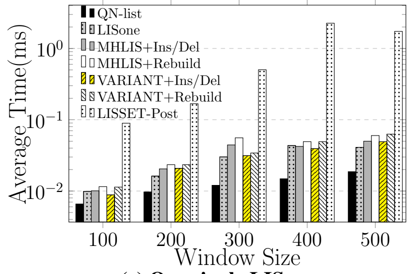
(b) Construction time cost



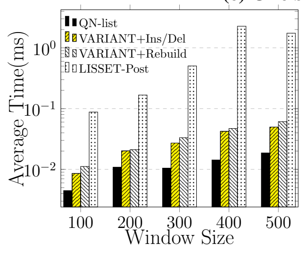
(c) Maintenance efficiency



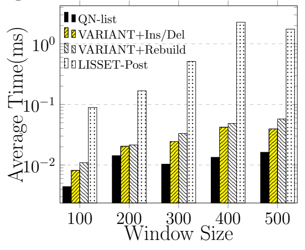
(d) LIS Enumeration



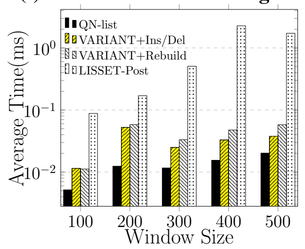
(e) One single LIS



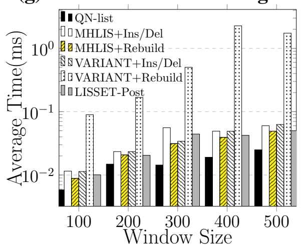
(f) LIS with maximum weight



(g) LIS with minimum weight



(h) LIS with maximum gap



(i) LIS with minimum gap

Figure 11: Evaluation on stock data

(MHLIS+Ins/Del, VARIANT+ Ins/Del). The update efficiency is measured by the throughput, i.e., the number of items handled per second without answering any query. Figure 11c shows that our method is obviously faster than comparative approaches on data structure update performance.

LIS Enumeration. We compare our method on LIS Enumeration with LISSET and DP, where LISSET is the only previous work that can be used to enumerate LIS under the sliding window model. We report the average query response time in Figure 11d. In the context of data stream, the overall query response time includes two parts, i.e., the data structure update time and online query time. Our method is faster than both LISSET and DP, and with the increasing of time window size, the performance advantage is more obvious.

LIS with Max/Min Weight. VARIANT [7] is the only previous work on LIS with maximum/minimum weight. Figures 11g and 11f confirms the superiority of our method with regard to VARIANT(VARIANT+Rebuild and VARIANT+Ins/Del).

LIS with Max/Min Gap. VARIANT [7] computes the LIS with maximum and minimum gap while MHLIS [22] only computes LIS with the minimum gap. The average running time in each window of different methods are in Figures 11h and 11i. We can see that our method outperforms other methods significantly.

LIS length (Output a single LIS). We compare our method with LISone [3] on outputting an LIS (The length comes out directly). We also add other comparative works into comparison they can easily support outputting an LIS. Figure 11e shows that our method is much more efficient than comparative methods on computing LIS length and output a single LIS.

8. CONCLUSIONS

In this paper, we propose a uniform data structure to support enumerating all LIS and LIS with specific constraints over sequential data stream. The data structure built by our algorithm only takes linear space and can be updated only in linear time, which make our approach practical in handling high-speed sequential data streams. To the best of our knowledge, our work is the first to proposes a uniform solution (the same data structure and computing framework) to address all LIS-related issues in the data stream scenario. Our method outperforms the state-of-the-art work not only theoretically, but also empirically in both time and space cost.

Acknowledgment

This work was supported by The National Key Research and Development Program of China under grant 2016YFB1000603 and NSFC under grant 61622201, 61532010 and 61370055. Lei Zou is the corresponding author of this paper.

9. REFERENCES

[1] `github.com/vitoFantasy/lis_stream/`.

[2] `http://arxiv.org/abs/1604.02552`.

[3] M. H. Albert, A. Golynski, A. M. Hamel, A. López-Ortiz, S. Rao, and M. A. Safari. Longest increasing subsequences in sliding windows. *Theoretical Computer Science*, 321(2-3):405–414, Aug. 2004.

[4] W. M. W. M. E. Altschul, Stephen; Gish and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[5] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1-2):7–11, 2000.

[6] E. Chen, L. Yang, and H. Yuan. Longest increasing subsequences in windows based on canonical antichain partition. *Theoretical Computer Science*, 378(3):223–236, June 2007.

[7] S. Deorowicz. On Some Variants of the Longest Increasing Subsequence Problem. *Theoretical and Applied Informatics*, 21(3):135–148, 2009.

[8] S. Deorowicz. A cover-merging-based algorithm for the longest increasing subsequence in a sliding window problem. *Computing and Informatics*, 31(6):1217–1233, 2013.

[9] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD.*, pages 419–429, 1994.

[10] M. L. Fredman. On Computation of the Length of the Longest Increasing Subsequences. *Discrete Mathematics*, 1975.

[11] P. Gopalan, T. Jayram, R. Krauthgamer, and R. Kumar. Estimating the sortedness of a data stream. *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, page 327, 2007.

[12] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[13] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Cetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming sql standard. *Proceedings of the Vldb Endowment*, 1(2):1379–1390, 2008.

[14] E. Keogh, J. Lin, S.-H. Lee, and H. Van Herle. Finding the most unusual time series subsequence: algorithms and applications. *Knowledge and Information Systems*, 11(1):1–27, 2007.

[15] E. J. Keogh and M. J. Pazzani. An indexing scheme for fast similarity search in large time series databases. In *Scientific and Statistical Database Management, 1999. Eleventh International Conference on*, pages 56–67. IEEE, 1999.

[16] H. Kim. Finding a maximum independent set in a permutation graph. *Information Processing Letters*, 36(1):19–23, 1990.

[17] X. Lian, L. Chen, and J. X. Yu. Pattern matching over cloaked time series. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 1462–1464, 2008.

[18] T. W. Liao. Clustering of time series data - a survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[19] D. Liben-Nowell, E. Vee, and A. Zhu. Finding longest increasing and common subsequences in streaming data. In *Journal of Combinatorial Optimization*, volume 11, pages 155–175, 2006.

[20] G. Rabson, T. Curtz, I. Schensted, E. Graves, and P. Brock. Longest increasing and decreasing subsequences. *Canad. J. Math*, 13:179–191, 1961.

[21] G. d. B. Robinson. On the representations of the symmetric group. *American Journal of Mathematics*, pages 745–760, 1938.

[22] C.-t. Tseng, C.-b. Yang, and H.-y. Ann. Minimum Height and Sequence Constrained Longest Increasing Subsequence. *Journal of internet Technology*, 10:173–178, 2009.

[23] I.-H. Yang and Y.-C. Chen. Fast algorithms for the constrained longest increasing subsequence problems. In *Proceedings of the 25th Workshop on Combinatorial Mathematics and Computing Theory*, pages 226–231, 2008.

[24] H. Zhang. Alignment of blast high-scoring segment pairs based on the longest increasing subsequence algorithm. *Bioinformatics*, 19(11):1391–1396, 2003.