This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

Longest Increasing Subsequence Computation over Streaming Sequences

Youhuan Li[1], Lei Zou[1], Huaming Zhang[2], Dongyan Zhao[1]

[1] *Peking University, China;*

[2] *University of Alabama in Huntsville,USA*

[1]{liyouhuan,zoulei,zhaody}@pku.edu.cn, [2]hzhang@cs.uah.edu

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.

Index Terms—Data Streams, Longest Increasing Subsequence, Enumeration, Constraints.

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 [1], [2], clustering [3]. Among these, computing the Longest Increasing Subsequence (LIS) over sequential data is a classical problem. An increasing[1] subsequence is a subsequence whose elements are sorted in order from the smallest to the biggest. Note that a sequence may contain multiple LIS, all of which have the same length.

Besides the static model (i.e., computing LIS over a given sequence α), computing LIS has been considered in the streaming model [4], [5]. 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 items that spans in the data stream. Consider a sequence $\alpha = \{3, 9, 6, 2, 8, 5, 7\}$ under window W in Figure 3 (in Section 2). There are four LIS in α: $\{3, 6, 7\}$,$\{3, 6, 8\}$, $\{2,5,7\}$ and $\{3,5,7\}$. Besides LIS enumeration, we introduce some important features of LIS, i.e., *gap*, *weight*, *slope*, *range* and compute LIS with various constraints, such as LIS with maximum gap, where "gap" measures the value difference between the tail and the head item of LIS. Among four LIS, {3,6,8} and {2,5,7} are both LIS with maximum gap. More constraints are formally defined in Section 2. We discuss two examples to demonstrate the usefulness of LIS in different applications.

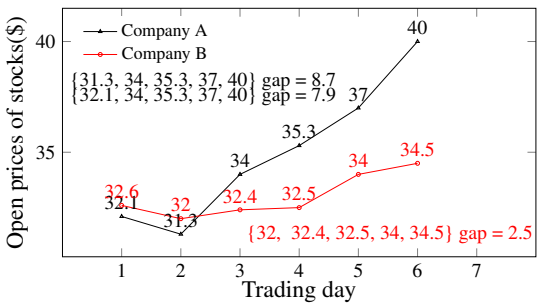


Fig. 1: LIS with different gaps of stock price sequence

1: Real-time Trend Detection.

LIS is a classical measure for sortedness and trend analysis [6], [7], [8]. The longer the LIS of a sequence is, the more sorted the sequence shows, which further indicates an uptrend of a sequence [6]. For example, LIS-based stock trend detection is studied in Jin et al. [7]. A company's stock price forms a time-evolving sequence and real-time measuring the stock trend is significant for 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 1 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

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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

intensity of A's stock obviously 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. Besides, slope between two items (see Definition 5) can also be used to describe the rising strength of stock prices.

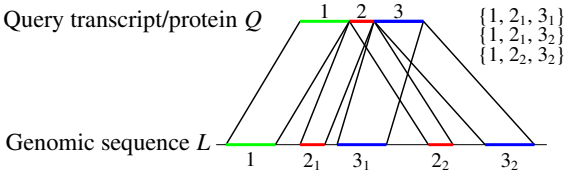


Fig. 2: Biological Sequence Alignment

2: Sequence Matching.

LIS is also used in sequence matching [4], [9], which is mainly used in biological sequence query. A typical example is a two-step algorithm (BLAST+LIS) proposed by Zhang [9] to locate a transcript or protein sequence in the human genome map. The BLAST (Basic Local Alignment Search Tool) [10] algorithm is to identify high-scoring segment pairs (HSPs) between query transcript sequence Q and a long genomic sequence L. Figure 2 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 2. 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 2, 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 [11]. 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. Also, some may limit the distance between two consecutive HSPs in a certain range which corresponds to the range-constrained LIS in Definition 5.

Therefore, LIS are useful and the applications require efficiently computing both LIS enumeration and constrained LIS enumeration simultaneously, but none of the existing approaches support that. For example, the method in [5] supports LIS enumeration, but fails to compute constrained LIS. In [12], [13] and [14], the method can be used to compute constrained LIS, but not to enumerate all LIS. A uniform method to support both LIS enumeration and constrained LIS enumeration is desirable.

Moreover, 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 [4], [5], [15]. None of them computes constrained LIS. Literature review and the comparative studies of our method against other related work are given in Section 6 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. We prove that the space complexity of our data structure is $O(w)$, while the algorithm proposed in [5] needs a space of size $O(w^2)$. Time complexities of our data structure construction and update algorithms are also better than [5]. For example, [5] 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 6. Experimental results on both real and synthetic datasets confirm that our algorithms outperform existing algorithms. Experimental codes and datasets are available at Github [16]. 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 all existing LIS with constraints uniformly.
3) Our data structure is scalable in stream model because of the linear update algorithm and linear space cost.
4) Extensive experiments confirm the superiority of our method.

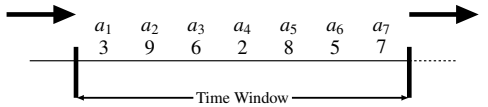
2 Problem Formulation

Given sequence $\alpha = \{a_1, a_2, \cdots, a_n\}$, an increasing subsequence s of α is a subsequence 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

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

of LIS of α by $LIS(\alpha)$. For a sequence s, the head and tail item of s are denoted as s^h and s^t, respectively. We use $|s|$ to denote the length of s.



Slope-constrained LIS(p = 1.5): $\{a_4 = 2, a_6 = 5, a_7 = 7\}$
Range-constrained LIS($L_I = 2, U_I = 3, L_V = 1, U_V = 3$): $\{a_1 = 3, a_3 = 6, a_5 = 8\}$
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: $\{a_1 = 3, a_6 = 5, a_7 = 7\}$,$\{a_1 = 3, a_3 = 6, a_7 = 7\}$

Fig. 3: LIS with constraints in data stream model

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 [11] 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 1. ***(LIS-enumeration)****. Given a time-evolving sequence $\alpha_\infty = \{a_1, ..., a_\infty\}$ and a time window W of size w,* LIS-enumeration *is to report $LIS(\alpha(W, i))$ (i.e., all LIS within 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. So far, eight kinds of constraints for LIS were proposed in the literature [12], [13], [14] and our method can easily support all of them. We study the following constraints over the LIS's *weight* (Definition 2), *gap* (Definition 3) and *width* (Definition 4), after which we define several problems computing LIS with various constraints (Definition 5).

Definition 2. ***(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 3. ***(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.*

Definition 4. ***(Width)****[12]. Let α be a sequence, s be an LIS in $LIS(\alpha)$ where $s = \{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$ ($k = |s|$). The* width *of s is defined as $width(s) = i_k - i_1$, i.e., the positional distance between the tail item(a_{i_k}) and the head item(a_{i_1}) of s.*

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 W continually as W slides. Consider an LIS of $\alpha(W, t_i)$: $s = \{a_{i_1}, a_{i_2}, ..., a_{i_m}\}$.*

- *s is an **LIS with Maximum/Minimum Weight** if s has maximum/minimum weight among all LIS in $LIS(\alpha(W, t_i))$.*
- *s is an **LIS with Maximum/Minimum Gap** if s has maximum/minimum gap among all LIS in $LIS(\alpha(W, t_i))$.*
- *s is an **LIS with Maximum/Minimum Width** if s has maximum/minimum width among all LIS in $LIS(\alpha(W, t_i))$.*
- *s is a **Slope-constrained LIS(SLIS)** if for a nonnegative slope boundary p and all $1 \le k < m$, the slope between two consecutive items in s is not less than p,i.e, $\frac{a_{i_{k+1}} - a_{i_k}}{i_{k+1} - i_k} \ge p$.*
- *s is a **Range-constrained LIS(RLIS)** if for two ranges $[L_I, U_I]$ and $[L_V, U_V]$ where $0 < L_I \le U_I < n$, $0 \le L_V \le U_V$, $L_I \le i_{k+1} - i_k \le U_I$ and $L_V \le a_{i_{k+1}} - a_{i_k} \le U_V$ $(1 \le k < m)$.*

A running example that is used throughout the paper is given in Figure 3, which shows a time-evolving sequence α_∞ and its first time window W. The induced sequence within the time window is $\alpha = \{a_1 = 3, a_2 = 9, a_3 = 6, a_4 = 2, a_5 = 8, a_6 = 5, a_7 = 7\}$. There are four LIS in α: $\{3, 6, 7\}$, $\{3, 6, 8\}$, $\{2, 5, 7\}$ and $\{3, 5, 7\}$. The gaps of these four LIS are 4, 5, 5, 4. Therefore, $\{3, 6, 8\}$ and $\{2, 5, 7\}$ have the maximum gap while $\{3, 6, 7\}$ and $\{3, 5, 7\}$ have the minimum gap. Also, when we set the slope $p = 1.5$ (in Figure 3), then $\{3, 6, 8\}$ does not satisfy the slope constraint since the slope between $a_3 = 6$ and $a_5 = 8$ is $(8-6)/(5-3) = 1 < 1.5$. Similarly, $\{3, 6, 7\}$ and $\{3, 5, 7\}$ do not satisfy the slope constraint and only $\{2, 5, 7\}$ is slope-constrained LIS. For the range-constrained LIS with ranges $[L_I = 2, U_I = 3]$ and $[L_V = 1, U_V = 3]$, $\{2, 5, 7\}$ and $\{3, 5, 7\}$ do not satisfy the range constraints since the positional distance between $a_6 = 5$ and $a_7 = 7$ is $1 < L_I = 2$. Similarly, $a_3 = 6$ and $a_7 = 7$ cannot constitute a range-constrained LIS either since $7 - 3 = 4 > U_I = 3$. $\{3, 6, 8\}$ is the only range-constrained LIS.

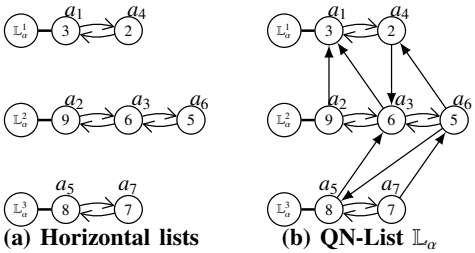
3 Quadruple Neighbor List $\mathbb{L}_\alpha$

3.1 $\mathbb{L}_\alpha$—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). Consider a sequence $\alpha = \{a_1, a_2, ..., a_w\}$ and a_i, $a_j \in \alpha$. a_i is said to be *compatible* with a_j if $i < j$ and $a_i \le a_j$. We denote it by $a_i \overset{\alpha}{\preccurlyeq} a_j$. Also, we use $IS_\alpha(a_i)$ to denote the set of all *increasing subsequences* of α that ends with a_i and we define *rising length* [5][3] of a_i, denoted as $RL_\alpha(a_i)$, as the maximum length of subsequences in $IS_\alpha(a_i)$. For example, consider the sequence $\alpha = \{a_1 = 3, a_2 = 9, a_3 = 6, a_4 = 2, a_5 =$

3. *Rising length* in this paper is the same as *height* defined in [5]. 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 [13].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering



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

Fig. 4: Horizontal lists and QN-List

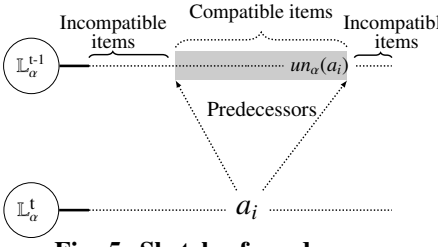


Fig. 5: Sketch of predecessors

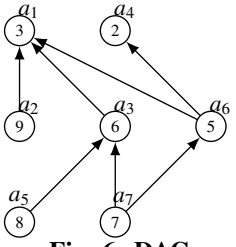


Fig. 6: DAG

$8, a_6 = 5, a_7 = 7\}$ in Figure 3. Consider $a_5 = 8$. There are five increasing subsequences$\{a_5 = 8\}$, $\{a_1 = 3, a_5 = 8\}$, $\{a_3 = 6, a_5 = 8\}$, $\{a_4 = 2, a_5 = 8\}$, $\{a_1 = 3, a_3 = 6, a_5 = 8\}$ that end with a_5. The maximum length of these increasing subsequences is 3. Hence, $RL_\alpha(a_5) = 3$.

Definition 6. ***(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 3, 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 concepts, we introduce four neighbours for each item a_i as follows:

Definition 7. ***(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 8. The horizontal lists of α is presented in Figure 4a.

Definition 8. ***(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 $1 \le k' < k$, $a_{i_{k'}} = ln_\alpha(a_{i_{k'+1}})$ 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 $\mathbb{L}^k_\alpha$.*

Apparently, for $\forall\ a_i \in \mathbb{L}^t_\alpha$, predecessor of a_i must be in $\mathbb{L}^{t-1}_\alpha$ $(t > 1)$.

Definition 9. ***(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 (Definition 8) 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 (Figure 3) and the curve arrows indicate the left and right neighbor relationship while the straight arrows indicate the up and down neighbor relationship. It is easy to understand that the length of LIS in α is exactly the number of horizontal lists in $\mathbb{L}_\alpha$. Besides, QN-list over sequence α of w items costs only $O(w)$ space.

3.2 $\mathbb{L}_\alpha$—Properties

We discuss some properties of the QN-List, which will be used in the maintenance algorithm in Section 4 and various QN-List-based algorithms in Section 5. Proofs for theorems and lemmas are given in Appendix B[4].

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 $\mathbb{L}^t_\alpha$.*

1) *Items in each horizontal list $\mathbb{L}^t_\alpha$ are monotonically decreasing while their subscripts (i.e., their original position in α) are monotonically increasing from the left to the right.*
2) *$\forall a_i \in \alpha$, all predecessors of a_i form a nonempty consecutive block in $\mathbb{L}^{t-1}_\alpha$ $(t > 1)$ and $un_\alpha(a_i)$ is a_i's the rightmost predecessor.*

In Figure 4a, we can see that items in each horizontal list are increasing from left to the right while their original positions are decreasing (Lemma 1(1)). Also, 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 $\mathbb{L}^{t-1}_\alpha$ (Lemma 1(2)).

Lemma 2. *Given sequence α and $\mathbb{L}_\alpha$, $\forall\ a_i \in \mathbb{L}^t_\alpha$:*

1) *$RL_\alpha(a_i) = t$ if and only if $a_i \in \mathbb{L}^t_\alpha$.*
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, $dn_\alpha(a_i) > a_i$.*

For example, in Figure 4b, a_3 is the rightmost item that is before a_5 in $\mathbb{L}^2_\alpha$ $(RL_\alpha(a_5) = 3)$ and $un_\alpha(a_5) = a_3$.

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 $\mathbb{L}^i_\alpha$.

4. All appendixes are given in the supplementary of the submission.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

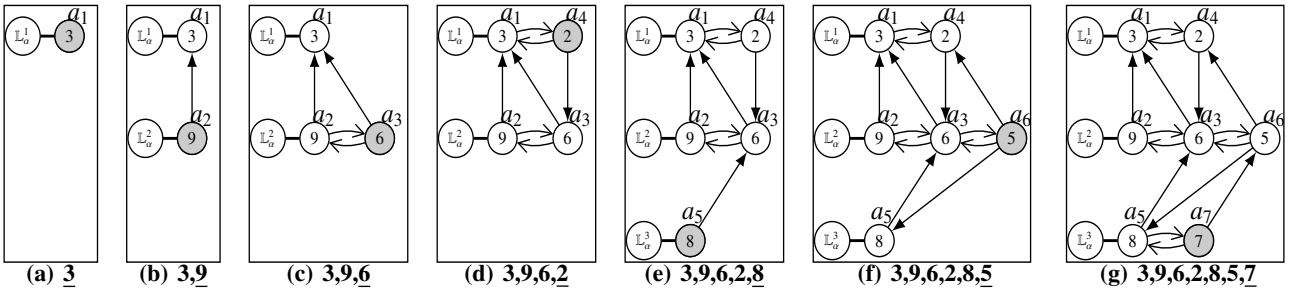


Fig. 7: Example of building QN-List for sequence: {$a_1 = 3$, $a_2 = 9$, $a_3 = 6$, $a_4 = 2$, $a_5 = 8$, $a_6 = 5$, $a_7 = 7$}

For example in Figure 4b, we can see that the sequence consisting of tail items: $\{a_4 = 2, a_6 = 5, a_7 = 7\}$ is in ascending order.

3.3 $\mathbb{L}_\alpha$—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 sequence α.

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

1) Initially, four neighbours of each item a_i are set NULL;
2) Step 1: $\mathbb{L}_\alpha^1$ is created in $\mathbb{L}_\alpha$ and a_1 is added in $\mathbb{L}_\alpha^1$;
3) Step 2: if $a_2 < a_1$, it means $RL_\alpha(a_2) = RL_\alpha(a_1) = 1$. Thus, we append a_2 to $\mathbb{L}_\alpha^1$. Since a_2 comes after a_1 in sequence α, we set $rn_\alpha(a_1) = a_2$ and $ln_\alpha(a_2) = a_1$.
   If $a_2 \geq a_1$, we can find an increasing subsequence $\{a_1, a_2\}$, i.e, $RL_\alpha(a_2) = 2$. Thus, we create the second horizontal list $\mathbb{L}_\alpha^2$ and add a_2 to $\mathbb{L}_\alpha^2$. 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) 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 $\mathbb{L}_\alpha^t$, we have the following two conclusions:
   a) If $Tail(\mathbb{L}_\alpha^t) > a_i$, then $RL_\alpha(a_i) \leq t$. Assume that $RL_\alpha(a_i) > t$. It means that there exits at least one item a_j ($\in \mathbb{L}_\alpha^t$) 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 $Tail(\mathbb{L}_\alpha^t)$ is the minimum item in $\mathbb{L}_\alpha^t$ (see Lemma 2). $Tail(\mathbb{L}_\alpha^t) > a_i$ means that all items in $\mathbb{L}_\alpha^t$ are larger than a_i. That is contradicted to $a_j \overset{\alpha}{\preccurlyeq} a_i \wedge a_j \in \mathbb{L}_\alpha^t$. Thus, $RL_\alpha(a_i) \leq t$.
   b) If $Tail(\mathbb{L}_\alpha^t) \leq a_i$, then $RL_\alpha(a_i) > t$. Since $Tail(\mathbb{L}_\alpha^t)$ is before a_i in α and $Tail(\mathbb{L}_\alpha^t) \leq a_i$, $Tail(\mathbb{L}_\alpha^t)$ is compatible a_i. Let us consider an increasing subseqeunce s ending with $Tail(\mathbb{L}_\alpha^t)$, whose length is t since $Tail(\mathbb{L}_\alpha^t)$'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 $Tail(\mathbb{L}_\alpha^t) \geq Tail(\mathbb{L}_\alpha^{t'})$ if $t \geq t'$(see Lemma 3). Thus, we need to find the first list $\mathbb{L}_\alpha^t$ whose tail $Tail(\mathbb{L}_\alpha^t)$ 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 that needs $O(\log m)$ time. If there is no such list, i.e., $Tail(\mathbb{L}_\alpha^m) \leq a_i$, we create a new empty list $Tail(\mathbb{L}_\alpha^{m+1})$ and insert a_i into $Tail(\mathbb{L}_\alpha^{m+1})$.
   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 $un_\alpha(a_i) = Tail(\mathbb{L}_\alpha^{t-1})$. Analogously, we set $dn_\alpha(a_i) = Tail(\mathbb{L}_\alpha^{t+1})$. So far, we correctly determine the four neighbors of a_i. We can repeat the above steps until all items are inserted to $\mathbb{L}_\alpha$. Pseudo codes for building the QN-List $\mathbb{L}_\alpha$ are presented in Algorithm 1 in Appendix A.

Apparently, the time complexity building QN-List over α is $O(w \log w)$.

3.4 LIS Enumeration

Let's discuss how to enumerate all LIS of sequence α based on the QN-List $\mathbb{L}_\alpha$. 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 $|\mathbb{L}_\alpha|$ length 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 $RL_\alpha(a_i)$ length increasing subsequences ending with 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 10. *(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:*

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

The $G(\alpha)$ over the sequence $\alpha = \{3, 9, 6, 2, 8, 5, 7\}$ is presented in Figure 6 where each path of length $|\mathbb{L}_\alpha|$ corresponds to an LIS. For example, we can find a path $a_5 = 8 \rightarrow a_3 = 6 \rightarrow a_1 = 3$, corresponding to 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) 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. Pseudo code for LIS enumeration is presented in Algorithm 2 in Appendix A.

Theorem 1. *The time complexity for LIS enumeration is* $O(OUTPUT)$*, where* OUTPUT *is the total size of all LIS.*

4 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. Insertion has been taken care of when we discuss how to construct QN-List in Section 3 (See Lines 2 to 10 of Algorithm 1 in Appendix A). 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 (Section 4.1) and the vertical update for up and down neighbors (Section 4.2).

4.1 Horizontal Update

Definition 11. ***(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:*

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

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 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 $\mathbb{L}_\alpha^t$. Theorem 2 tells us how to distinguish them. In a nutshell, $\forall\, a \in \mathbb{L}_\alpha^t$ $(1 < t \leq m)$, 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, $\{a_5\}$ 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 in Appendix A 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 2.

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 2. *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* $a_i, 2 \leq i \leq m \in \mathbb{L}_\alpha^t, 1 \leq t \leq m$*,*

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 2, 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 by decreasing order of their values. According to Theorem 2 and Lemma 1(1), we can know that the horizontal lists obtained by the above process is the same as re-building $\mathbb{L}_{\alpha^-}$ for sequence α^-.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

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 $\mathbb{L}_\alpha^t$, $un_\alpha^{t-1}(a_j)$ 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 in Appendix A) to divide each horizontal list $\mathbb{L}_\alpha^t$ into two sublists: $Left(\mathbb{L}_\alpha^t)$ and $Right(\mathbb{L}_\alpha^t)$.

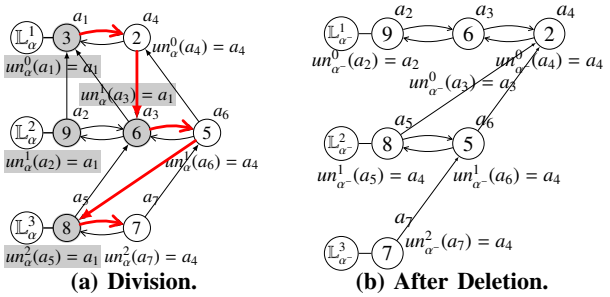
Let's consider the division of each horizontal list of $\mathbb{L}_\alpha$. In fact, in our division algorithm, the division of $\mathbb{L}_\alpha^t$ depends on that of $\mathbb{L}_\alpha^{t-1}$. We first divide $\mathbb{L}_\alpha^1$. Apparently, $Left(\mathbb{L}_\alpha^1) = \{a_1\}$ and $Right(\mathbb{L}_\alpha^1) = \{\mathbb{L}_\alpha^1\} - \{a_1\}$. Recursively, assuming that we have finished the division of $\mathbb{L}_\alpha^t$, $1 \le t < m$, 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 $\forall\, a_j \in \mathbb{L}_\alpha^{t+1}$, 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 $HEAD(Right(\mathbb{L}_\alpha^t))$ 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 $1 \le t \le m$, 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, we append $Right(\mathbb{L}_{\alpha^-}^t)$ 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 3. *The list formed by appending $Right(\mathbb{L}_\alpha^t)$ to $Left(\mathbb{L}_\alpha^{t+1})$ is strictly decreasing from the left to the right.*

According to Theorem 2 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 3, 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.



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

Fig. 8: Maintenance

4.2 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 $a_i \in \mathbb{L}_\alpha^t$, $un_\alpha(a_i)$(if exists) is the rightmost item in $\mathbb{L}_\alpha^{t-1}$ who is before a_i in sequence α; while, $dn_\alpha(a_i)$(if exists) is the rightmost item in $\mathbb{L}_\alpha^{t+1}$ 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 $Left(\mathbb{L}_\alpha^t)$ to the above level, we need to change their up neighbors but not down neighbors. While, $Right(\mathbb{L}_\alpha^t)$ is still in the same level after the horizontal update. We need to change their down neighbors but not up neighbors.

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

Analysis and Algorithm.

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

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

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 $a_i \in \mathbb{L}_{\alpha^-}^t$, where $1 \le t \le m$. According to the horizontal list adjustment, there are two cases for a_i: a_i is from $Left(\mathbb{L}_\alpha^{t+1})$ or a_i is from $Right(\mathbb{L}_\alpha^t)$. Then, the following claims hold:*

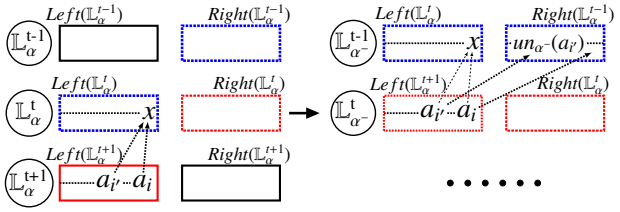
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

1) Assuming a_i is from $Left(\mathbb{L}_\alpha^{t+1})$
- *a) $dn_{\alpha^-}(a_i) = dn_\alpha(a_i)$ (the down neighbor remains).*
- *b) Let x be the rightmost item of $Left(\mathbb{L}_\alpha^t)$. If $un_\alpha(a_i) \neq x$, $un_{\alpha^-}(a_i) = un_\alpha(a_i)$ (the up neighbor remains).*

2) Assuming a_i is from $Right(\mathbb{L}_\alpha^t)$
- *a) $un_{\alpha^-}(a_i) = un_\alpha(a_i)$ (i.e., the up neighbor remains).*
- *b) Let y be the rightmost item of $Left(\mathbb{L}_\alpha^{t+1})$. If $dn_\alpha(a_i) \neq y$, $dn_{\alpha^-}(a_i) = dn_\alpha(a_i)$ (i.e., the down neighbor remains)*

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

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



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

Fig. 9: Case 1: updating up neighbors

Case 1: Consider items in $Left(\mathbb{L}_\alpha^{t+1})$. According to Theorem 2, $Left(\mathbb{L}_\alpha^{t+1})$ will be promoted into the list $\mathbb{L}_{\alpha^-}^t$.

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

Now, we consider the case that $un_\alpha(a_i) = x$. Let $a_{i'}$ denote the leftmost item in $Left(\mathbb{L}_\alpha^{t+1})$ where $un_\alpha(a_{i'})$ is 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 $Left(\mathbb{L}_\alpha^t)$), as shown in Figure 9(a). These items' up neighbors need to be adjusted in $\mathbb{L}_{\alpha^-}$.

Theorem 4. *Given an sequence α and $\mathbb{L}_\alpha$, assume that $un_\alpha(a_i) = x$ and $a_{i'}$ denote the leftmost item in $\mathbb{L}_\alpha^{RL_\alpha(a_i)}$ where $un_\alpha(a_{i'})$ is x. If $dn_\alpha(x) \neq NULL$, then $a_{i'}$ is exactly $rn_\alpha(dn_\alpha(x))$; otherwise, $a_{i'}$ is $HEAD(\mathbb{L}_\alpha^{RL_\alpha(a_i)})$.*

With Theorem 4, we can easily find out $a_{i'}$ in $O(1)$ time. Then, we firstly 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 $\mathbb{L}_{\alpha^-}^{t-1}$ until finding the rightmost item whose position is before $a_{i'}$ in sequence α^-. Finally, we set $un_{\alpha^-}(a_{i'}) = a^*$ (see Line 14 in Algorithm 4 in Appendix A).

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

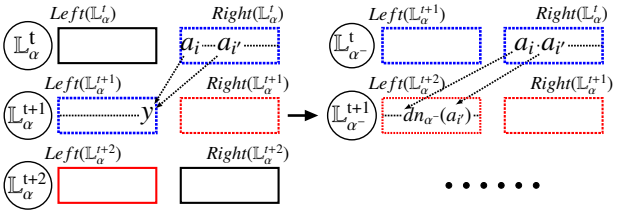
Iteratively, we consider the items on the right of $a_{i'}$. Actually, the adjustment of the next item's up neighbor can begin from the current position of a^*. It is straightforward to know the time complexity for updating up neighbors (Algorithm 4 in Appendix A) is $O(|\mathbb{L}_{\alpha^-}^{t-1}|)$,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

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

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

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 Line 8 in Algorithm 5 in Appendix A).

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 $NULL$.



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

Fig. 10: Case 2: updating down neighbors

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 in Appendix A). 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.

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 $\mathbb{L}_\alpha^t$ ($1 \leq t \leq m$ and then finishes the horizontal update by appending $Right(\mathbb{L}_\alpha^t)$ to $Left(\mathbb{L}_\alpha^{t+1})$. The second phrase is to conduct vertical update. Pseudo codes for handling deletion are presented in Algorithm 6 of Appendix A.

Theorem 5. *The time complexity of our deletion algorithm is $O(w)$, where w denotes the time window size.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

5 COMPUTING LIS WITH CONSTRAINTS

In this section, we consider all kinds of constraints that are defined in Section 2 and compute LIS with different constrains over a sequence α and $\mathbb{L}_\alpha$. Due to space limits, the computations for LIS with maximum/minimum weight/gap/width, which has been covered in our previous conference paper[17], are given in Appendix C and we focus on the computation for slope-constrained LIS and range-constrained LIS.

5.1 Slope-constrained LIS(SLIS)

According to the definition of SLIS (Definition 5), we can find that the slope only constrains each two consecutive items in an LIS. Thus, the slope is in essence constraints over the predecessors of an item in the sequence. For an item a_i, the predecessors of a_i who satisfy the slope constraints are called *slope-proper* predecessors of a_i. Thus, the naive solution to SLIS is to verify the slope constraints during the computation for LIS enumeration. However, repeatedly visiting items of no slope-proper predecessors may be wasteful. A possible optimization is to mark those items with no slope-proper predecessors and avoid visiting them during LIS enumeration. While, each item may have at most $O(|\alpha|)$ predecessors and the marking computation is costly. Also, an item with slope-proper predecessors may not be in an SLIS. For example, for an item a_i that has only one slope-proper predecessor a_j, if a_j has no slope-proper predecessor, a_i will never exist in an SLIS. Based on these observations, we propose a dynamic programming algorithm to color items (white or black) to determine who are to be ignored during SLIS computation. We will firstly introduce our coloration algorithm (coloring phrase) and then we will discuss how to efficiently enumerate all SLIS based on the coloring results (outputting phrase).

Coloration The coloration process begins at the first level of QN-list. Initially, all items in $\mathbb{L}_\alpha^1$ are colored white. We iteratively precess other levels from $\mathbb{L}_\alpha^2$ to $\mathbb{L}_\alpha^{|\mathbb{L}_\alpha|}$. We color a_i white if a_i has at least one slope-proper predecessor that has previously been colored white. After collation, for any white item a_i, there must exist an increasing subsequence s ending with a_i where each item in s is white.

Theorem 6. *Given a sequence α and a_i, $a_j \in \mathbb{L}_\alpha^{t+1}$. Assume that $a_{i'}$, $a_{j'} \in \mathbb{L}_\alpha^t$ are the leftmost white slope-proper predecessors of a_i and a_j,respectively. If a_j is at the right side of a_i, then $a_{j'}$ is either $a_{i'}$ or at the right side of $a_{i'}$.*

We can know that finding a leftmost white slope-proper predecessor for $a_i \in \mathbb{L}_\alpha^{t+1}$ is enough to confirm that a_i is white. With Theorem 6, after determining the leftmost white predecessor a'_i of a_i, searching for the leftmost white predecessor of $rn_\alpha(a_i)$ can be conducted from a'_i to the right of $\mathbb{L}_\alpha^t$. Thus, after coloring items in $\mathbb{L}_\alpha^t$, we can color items in $\mathbb{L}_\alpha^{t+1}$ by scanning $\mathbb{L}_\alpha^t$ and $\mathbb{L}_\alpha^{t+1}$ only once (Lines 3-12 in Algorithm 13 of Appendix A).

Outputting SLIS After the coloration, to output an SLIS, we can find a white item $a_i \in \mathbb{L}_\alpha^m$ ($m = |\mathbb{L}_\alpha|$) and visit the leftmost white slope-proper predecessor of a_i. Recursively, we can get an SLIS (Lines 13-21 in Algorithm 13). Our method can be easily extended to support outputting all SLIS. It is analogous to LIS enumeration approach but it only visits *white* item in the QN-list during the enumeration. SLIS computation over running example is presented in Appendix D. Coloration for SLIS costs $O(|\alpha|)$ time while outputting an SLIS costs $O(l)$ time where l is the SLIS length. Thus, SLIS computation over $\mathbb{L}_\alpha$ costs $O(w)$ time.

5.2 Range-constrained LIS(RLIS)

Consider a sequence α and two ranges $[L_I, U_I]$ and $[L_V, U_V]$ where $0 < L_I \le U_I < n$, $0 \le L_V \le U_V$. For $a_{i'} \in \mathbb{L}_\alpha^t$ and $a_i \in \mathbb{L}_\alpha^{t+1}$. We call $a_{i'}$ as *range-proper* predecessor of a_i if $a_i - a_{i'} \in [L_V, U_V]$ and $i - i' \in [L_I, U_I]$ (Note that since $0 < L_I$ and $0 < L_V$, then we can easily know that $a_{i'} \le a_i$ and $i' < i$, namely, $a_{i'} \overset{\alpha}{\preccurlyeq} a_i$).

Coloration Similar to the solution of SLIS computation, we also assign color to each item. For each item a_i, if $RL_\alpha(a_i) = 1$, a_i should be white; otherwise, a_i will be white if and only if a_i has at least one white range-proper predecessor. Non-white items are called black items. We design an efficient coloring algorithm which costs only linear time. Initially, items in $\mathbb{L}_\alpha^1$ are colored white. We iteratively precess other levels from $\mathbb{L}_\alpha^2$ to $\mathbb{L}_\alpha^{|\mathbb{L}_\alpha|}$. Assuming that we have finished coloring items in the first t horizontal lists (i.e.,$\mathbb{L}_\alpha^1$,$\mathbb{L}_\alpha^2$,...,$\mathbb{L}_\alpha^t$), let's discuss how to color items in $\mathbb{L}_\alpha^{t+1}$.

For an item $a_i \in \mathbb{L}_\alpha^{t+1}$ and $a_{i'} \in \mathbb{L}_\alpha^t$, if $a_i - a_{i'} \in [L_V, U_V]$ and $i - i' \in [L_I, U_I]$, then equivalently, $a_i - U_V \le a_{i'} \le a_i - L_V$ and $i - U_I \le i' \le i - L_I$. According to Lemma1(1), items in $\mathbb{L}_\alpha^t$ are monotonically decreasing from the left to the right while their positions in α are monotonically increasing. Thus, for $a_i \in \mathbb{L}_\alpha^{t+1}$, we can find a leftmost item a_l in $\mathbb{L}_\alpha^t$ such that $a_l \le a_i - L_V$ and $i - U_I \le l$. Obviously, for any item $a_{i'}$ at the right of a_l, $a_l \le a_{i'} - L_V$ and $i' - U_I \le l$. Similarly, we use a_r to denote the rightmost item such that $a_i - U_v \le a_r$ and $r \le i - L_I$. Apparently, items from a_l to a_r in $\mathbb{L}_\alpha^t$ form a consecutive block which contains exactly all range-proper predecessors of a_i (see Figure 23 in Appendix D). If a_r is at the left of a_l, then there is no range-proper predecessor of a_i; otherwise, a_l is the leftmost range-proper predecessor of a_i while a_r is the rightmost. However, a_l could be black item, which is useless to RLIS computation. Therefore, we focus on the leftmost white range-proper predecessor of a_i.

Theorem 7. *Given a sequence α and $a_i \in \mathbb{L}_\alpha^{t+1}$, assume that $a_{i'}$ is the leftmost white item in $\mathbb{L}_\alpha^t$ such that $a_{i'} \le a_i - L_V$ and $i - U_I \le i'$. Then, we can conclude that either $a_{i'}$ is the leftmost white range-proper predecessor of a_i or a_i has no white range-proper predecessors (i.e.,a_i should be black).*

Theorem 8. *Given a sequence α and $\mathbb{L}_\alpha$. Consider a_i,$a_j \in \mathbb{L}_\alpha^{t+1}$. Assume that $a_{i'}$, $a_{j'} \in \mathbb{L}_\alpha^t$ are the leftmost white predecessors of a_i and a_j such that $a_{i'} \le a_i - L_V$ and $i - U_I \le i'$ and $a_{j'} \le a_j - L_V$ and $j - U_I \le j'$. Then if a_j is at the right side of a_i, $a_{j'}$ is either $a_{i'}$ or at the right of $a_{i'}$.*

With Theorem 8 and Theorem 7, after determining the leftmost white item $a_{i'}$ in $\mathbb{L}_\alpha^t$ such that $a_{i'} \le a_i - L_V$ and $i - U_I \le i'$, if $a_{i'}$ is exactly a range-proper predecessor of a_i, we color a_i white and set a pointer from a_i to $a_{i'}$ as the leftmost white range-proper predecessor; otherwise, if $a_{i'}$ is not a range-proper predecessor of a_i, then color a_i black.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

No matter what color a_i is, the coloration of $a_j = rn_\alpha(a_i)$ can be conducted by searching for leftmost white item $a_{j'}$ from $a_{i'}$ to the right such that $a_{j'} \leq a_j - L_V$ and $j - U_I \leq j'$. It is quite similar to the process in the coloration for SLIS (Lines 3-15 in Algorithm 14 in Appendix A).

Outputting RLIS This phrase is just the same as that of outputting SLIS(See Lines 16-24 in Algorithm 14 in Appendix A). RLIS computation over running example is presented in Appendix D. Similarly, outputting an RLIS over $\mathbb{L}_\alpha$ cost $O(w)$ time.

6 Experiments

We experimentally 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 [16].

6.1 Dataset

We use four datasets in our experiments: real-world stock data, gene sequence data, power usage data and synthetic data. The stock data is the historical open prices of Microsoft Cooperation in the past two decades[5], 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[6] against a gene dataset[7] according to the process in [9]. The power usage dataset is a public power demand dataset used in [18]. It contains 35,040 power usage value. The synthetic dataset is a time series benchmark [19] that contains one million data points. Due to space limits, the experimental results over power and synthetic data are presented in Appendix F.

6.2 Comparative Methods

We compare our method (denoted as **QN-list**) with several comparative algorithms, including our previous method (denoted as **QN-prev**) in [17].

LISSET [5] 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.

MHLIS [13] is to find LIS with the minimum gap but it does not work under data stream model. To enable the comparison, we implement two streaming version of MHLIS: **MHLIS+Re** and **MHLIS+I/D** where MHLIS+Re is to re-compute LIS from scratch in each time window while MHLIS+I/D is to apply our update method in MHLIS.

A family of static algorithms was proposed in [12] including LIS of minimal/maximal weight/gap/width (denoted as **VARIANT**). For the comparison, we implement two stream version of VARIANT: **VARIANT+Re** and **VARIANT+I/D** where VARIANT+Re is to re-compute LIS from scratch in each time window and VARIANT+I/D 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.

Yang et al.[14] proposed two different approaches for slope-constrained LIS computation (denoted as **YangS**) and range-constrained LIS computation (denoted as **YangR**), respectively. They focused on static sequence and use range maximum/minimum query (RMQ) [20] to support efficient range constraints check. They only find an RLIS(SLIS) while our method can find out all LIS satisfying a given constraint. Thus, to enable comparison, the SLIS/RLIS computation in this section focus on finding only one eligible LIS. Also, to enable comparison, the stream version of YangS/YangR is implemented by re-computing SLIS/RLIS from scratch for each window.

LISone [4] 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.

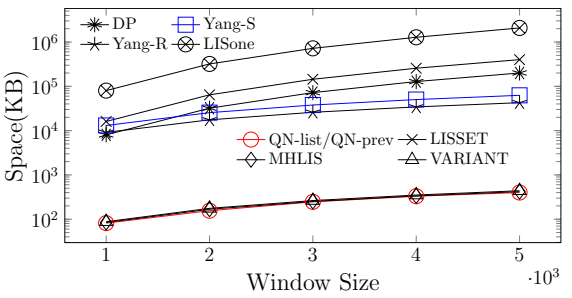
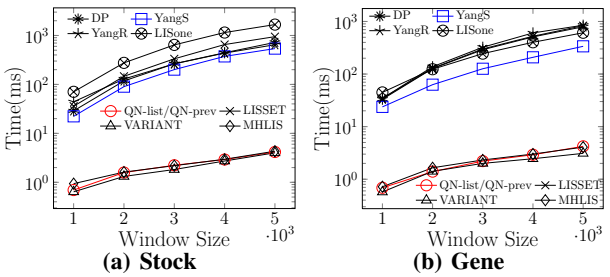


Fig. 11: Space



(a) Stock **(b) Gene**

Fig. 12: Construction

6.3 Experimental Evaluation

Data Structure Comparison. Evaluation of the data structures focuses on space, construction time and update time. Since the optimization of our method (QN-list) over our previous one (QN-prev) lies in maintenance, the space cost and construction time remain the same as that of previous version [17].

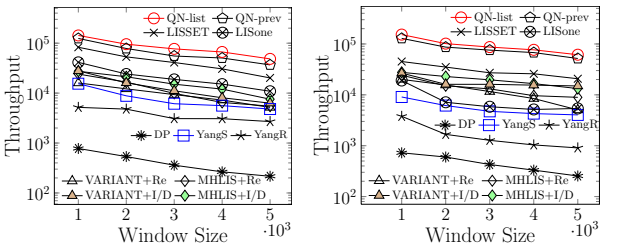
The **space cost** of each method is presented in Figure 11. Since space cost for the data structure of each method only depends on the size of sequence (window), the space

5. http://finance.yahoo.com/quote/MSFT/history?ltr=1
6. ftp://ftp.ncbi.nih.gov/refseq/B_taurus/mRNA_Prot/
7. ftp://ftp.ncbi.nlm.nih.gov/genbank/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

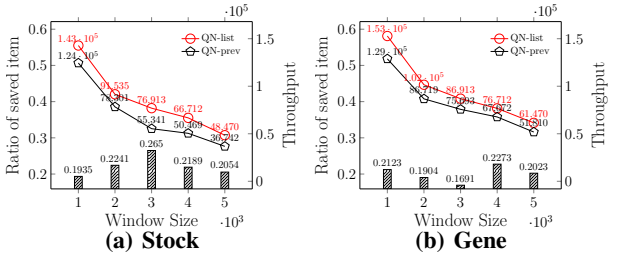
cost will be the same over different dataset and we only present the space cost over stock dataset. We can see that our method costs much less memory than LISSET, DP, YangS/YangR and LISone while slightly more than that of MHLIS and VARIANT, which results from the extra cost in our QN-List to support 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 (Table 3).

We **construct** each data structure five times and present their average constuction time in Figure 12. 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 2). Our construction time is slightly slower than VARIANT but faster than MHLIS, YangS and YangR, since they have the same construction time complexity (Table 2).



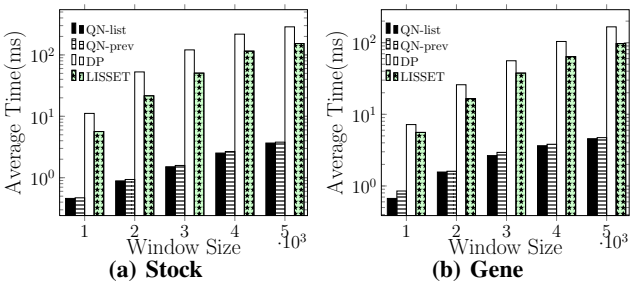
(a) Stock (b) Gene

Fig. 13: Maintenance



(a) Stock (b) Gene

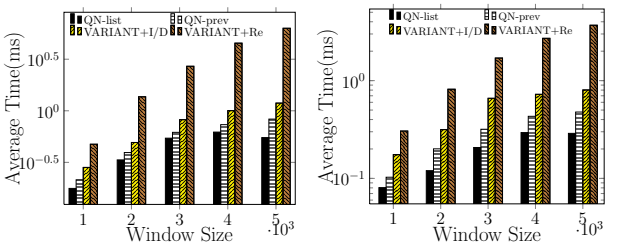
Fig. 14: Optimization of QN-list over QN-prev



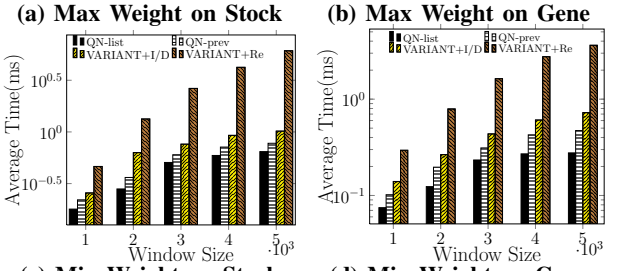
(a) Stock (b) Gene

Fig. 15: LIS Enumeration

None of MHLIS, VARIANT, DP or YangS/YangR 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+Re, VARIANT+Re). The second is to apply our update idea into MHLIS and VARIANT (MHLIS+I/D, VARIANT+I/D). The update efficiency is measured by the throughput, i.e., the number of items handled per second without answering any query. Figure 13 shows that our method is obviously faster than comparative approaches on data structure update performance. Besides, in Figure 14, we present the ratio of saved item of QN-list over QN-prev to explicitly measure the optimization (see Theorem 4).

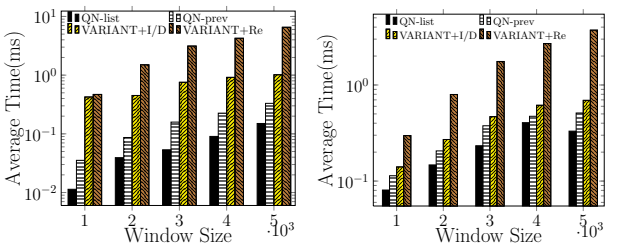


(a) Max Weight on Stock (b) Max Weight on Gene

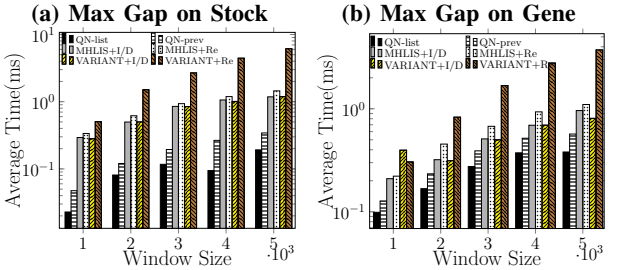


(c) Min Weight on Stock (d) Min Weight on Gene

Fig. 16: LIS with Extreme Weight



(a) Max Gap on Stock (b) Max Gap on Gene



(c) Min Gap on Stock (d) Min Gap on Gene

Fig. 17: LIS with Extreme Gap

LIS Enumeration. We compare our method on LIS Enumeration with LISSET and DP. LISSET is the only previous work that can be used to enumerate LIS under the sliding window model. The LIS number is quite huge[8] and each method only return not more than 10,000 LIS for each window. We report the average query response time in Figure 15. In data stream model, 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.

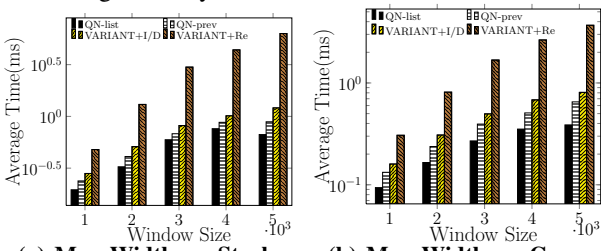
LIS with Max/Min Weight. VARIANT [12] is the only previous work on LIS with maximum/minimum weight. Figures 16 confirms the superiority of our method with regard to VARIANT(VARIANT+Re and VARIANT+I/D).

LIS with Max/Min Gap. VARIANT [12] computes the LIS with maximum and minimum gap while MHLIS [13]

8. We discuss this in Appendix E

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

only computes LIS with the minimum gap. The average running time in each window of different methods are in Figures 17. We can see that our method outperforms other methods significantly.

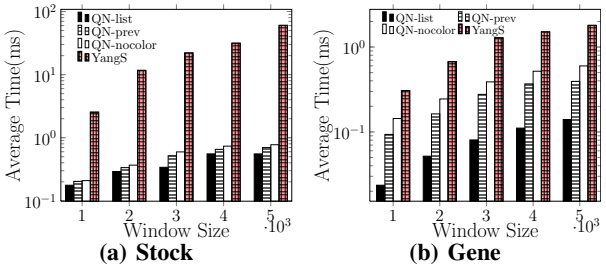


(a) Max Width on Stock **(b) Max Width on Gene**

(c) Min Width on Stock **(d) Min Width on Gene**

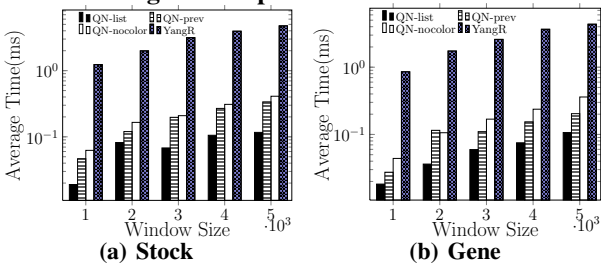
Fig. 18: LIS with Extreme Width

LIS with Max/Min Width. VARIANT [12] is the only previous work on LIS with maximum/minimum width. Figures 18 confirms the superiority of our method with regard to VARIANT(VARIANT+Re and VARIANT+I/D).



(a) Stock **(b) Gene**

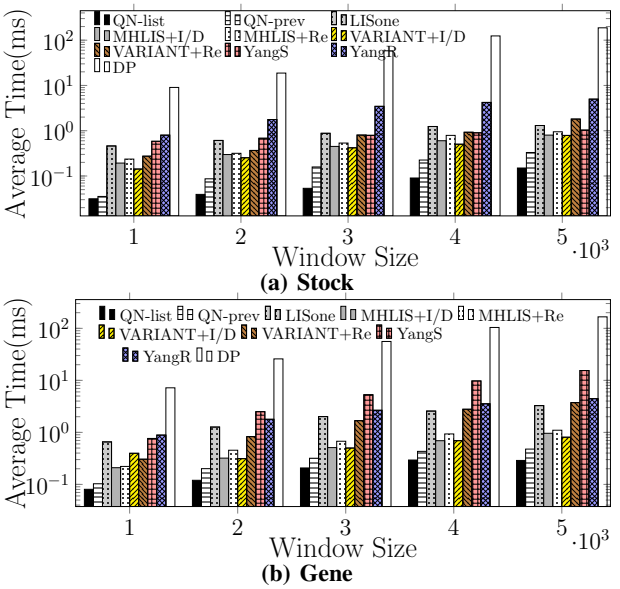
Fig. 19: Slope-constrained LIS



(a) Stock **(b) Gene**

Fig. 20: Range-constrained LIS

Slope/Range-constrained LIS [14] is the only previous work on slope-constrained LIS(SLIS) and range-constrained LIS(RLIS). We set three different ranges($R1$ = $\{L_I$ = 1,U_I = 20, L_v = 0, U_v = 50$\}$, $R2$ = $\{L_I$ = 20,U_I = 40, L_v = 50, U_v = 100$\}$, $R3$ = $\{L_I$ = 40,U_I = 60, L_v = 100, U_v = 150$\}$) and three different slopes($S1$ = 0, $S2$ = 0.5, $S3$ = 1.0) to evaluate the performance. We use the average running time under the three ranges/slopes for comparison. Also, since the coloration algorithm would introduce extra cost, we also compare our method with the solution verifying slope/range constraints during the LIS enumeration (denoted as QN-nocolor). We can see from Figures 19 and 20 that our method on this two problems are better than the approaches in [14] and the solutions based on LIS enumeration.



(a) Stock

(b) Gene

Fig. 21: LIS Length

LIS length (Output an LIS). We compare our method with LISone [4] on outputting an LIS (The length comes out directly). We also add other comparative works into comparison they can easily support outputting an LIS. Since there are user-defined parameters in RLIS and SLIS, we set the corresponding ranges large enough for RLIS and slope small enough for SLIS to guarantee the output of an LIS. Figure 21 shows that our method is much more efficient than comparative methods on computing LIS length and output a single LIS.

7 Related Work

7.1 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 w denotes the length of the sequence α. In [14], the solution for computing range/slope-constrained is also based on dynamic programming, i.e., range/slope-constrained LIS of α_{i+1} is computed from that of α_i. Dynamic programming-based method can also be easily extended to enumerate all LIS in a sequence which costs $O(w^2)$ space.

2. Young's tableau-based. [21] 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.[4] 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 [4] is to compress all

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

Method	LIS Enumeration	LIS with max Weight	LIS with min Weight	LIS with max Gap	LIS with min Gap	SLIS	RLIS	LIS Length
QN-list/QN-prev	$O(OUTPUT)$	$O(OUTPUT)$	$O(OUTPUT)$	$O(w+OUTPUT)$	$O(w+OUTPUT)$	$O(w)$	$O(w)$	$O(1)$
LISSET [5]	$O(OUTPUT)$	–	–	–	–	–	–	$O(1)$
MHLIS [13]	–	–	–	–	$O(w+OUTPUT)$	–	–	$O(1)$
VARIANT [12]	–	$O(OUTPUT)$	$O(OUTPUT)$	$O(w+OUTPUT)$	$O(w+OUTPUT)$	–	–	$O(1)$
DP	$O(OUTPUT)$	–	–	–	–	–	–	$O(1)$
YangS [14]	–	–	–	–	–	$O(w)$	–	$O(1)$
YangR [14]	–	–	–	–	–	–	$O(w)$	$O(1)$
LISone [4]	–	–	–	–	–	–	–	$O(1)$

TABLE 1: Theoretical Comparison on Online Query

Methods	Space Complexity	Time Complexity		
		Construction	Insert	Delete
QN-list/QN-prev	$O(w)$	$O(w \log w)$	$O(\log w)$	$O(w)$
LISSET [5]	$O(w^2)$	$O(w^2)$	$O(w)$	$O(w)$
MHLIS [13]	$O(w)$	$O(w \log w)$	$O(\log w)$	–
VARIANT [12]	$O(w)$	$O(w \log w)$	$O(\log w)$	–
DP	$O(w^2)$	$O(w^2)$	$O(w)$	–
YangS [14]	$O(w)$	$O(w \log w)$	$O(\log w)$	–
YangR [14]	$O(w)$	$O(w \log w)$	$O(\log w)$	–
LISone [4]	$O(w^2)$	$O(w^2)$	$O(w)$	$O(w)$

TABLE 2: Data Structure

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 [5], [12], [15], [13]. They classify items 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 [12], [15], *antichain* in [5]. Note that [12] and [13] conduct the partition over a static sequence to efficiently compute LIS with constraints. [15] 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 [5]. [5] is the only one to study the LIS enumeration in streaming model. Both of their insertion and deletion algorithms cost $O(w)$ time [5]. 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 (Definitions 8) is a *partition*. While, our data structure costs only $O(w)$ space and $O(w)$ time for each update and supports both LIS enumeration and LIS with various constraints. More theoretical analysis and feasibility comparison of existing work are discussed in the following Section 7.2.

7.2 Problem Perspective

We briefly position our problem in existing work on LIS computation in *computing task* and *computing model*. 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 [4], [15], [22], [23], [21]. The second is LIS enumeration, which finds all LIS in a sequence α [24], [5]. [24] 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, weight [12], [13]. Besides, there are two computing models for each of these LIS computing tasks. One is the static model assuming that the sequence α is given without changes [12], [21], [25], [13], [14]. The other model is the data stream model that has been considered in some recent work[4], [5].

Methods	Stream Model	LIS Enumeration	LIS with extreme weight	LIS with extreme gap	SLIS	RLIS	LIS length
QN-list/QN-prev	✓	✓	✓	✓	✓	✓	✓
LISSET	✓	✓	✗	✗	✗	✗	✓
MHLIS	✗	✗	✗	✓	✗	✗	✓
VARIANT	✗	✗	✓	✓	✗	✗	✓
DP	✗	✓	✗	✗	✗	✗	✓
YangS	✗	✗	✗	✗	✓	✗	✓
YangR	✗	✗	✗	✗	✗	✓	✓
LISone	✓	✗	✗	✗	✗	✗	✓

TABLE 3: Compaison on supported computing task

Table 3 illustrates the existing works considering both *computing task* and *computing model*. We can see that there is no existing uniform solution for all LIS-related problems, such as LIS length, LIS enumeration and LIS with constraints. Also, 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.

We also present theoretical comparison of existing work (More details of comparative works are available in Section 6.2) over these LIS computing tasks.

Data Structure Comparison. We compare the space, construction time and update time of our data structure against those of other works in Table 2 (The time complexities are based on the worst case analysis). We can see 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 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, DP or YangS/YangR 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 Comparison. Table 1 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. We can see that, our online query time complexities are the same with the comparative

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2761345, IEEE Transactions on Knowledge and Data Engineering

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.

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 we propose only takes linear space and can be updated in linear time, which makes 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.

References

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

[2] 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*, 2008, pp. 1462–1464.

[3] T. W. Liao, "Clustering of time series data - a survey," *Pattern Recognition*, vol. 38, no. 11, pp. 1857–1874, 2005.

[4] 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*, vol. 321, no. 2-3, pp. 405–414, Aug. 2004. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0304397504002142

[5] E. Chen, L. Yang, and H. Yuan, "Longest increasing subsequences in windows based on canonical antichain partition," *Theoretical Computer Science*, vol. 378, no. 3, pp. 223–236, Jun. 2007. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0304397507001235

[6] 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*, p. 327, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1283417

[7] R. Jin, S. McCallen, and E. Almaas, "Trend motif: A graph mining approach for analysis of dynamic complex networks," in *The 7th IEEE International Conference on Data Mining*. IEEE, 2007, pp. 541–546.

[8] L. Bonomi and L. Xiong, "On differentially private longest increasing subsequence computation in data stream." *Transactions on Data Privacy*, vol. 9, no. 1, pp. 73–100, 2016.

[9] H. Zhang, "Alignment of blast high-scoring segment pairs based on the longest increasing subsequence algorithm," *Bioinformatics*, vol. 19, no. 11, pp. 1391–1396, 2003.

[10] W. M. W. M. E. Altschul, Stephen; Gish and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[11] 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*, vol. 1, no. 2, pp. 1379–1390, 2008.

[12] S. Deorowicz, "On Some Variants of the Longest Increasing Subsequence Problem," *Theoretical and Applied Informatics*, vol. 21, no. 3, pp. 135–148, 2009. [Online]. Available: http://projekty.iitis.gliwice.pl/uploads/File/taai/3_2009/2009_3_4_art01_deorowicz.pdf

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

[14] 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*, 2008, pp. 226–231.

[15] S. Deorowicz, "A cover-merging-based algorithm for the longest increasing subsequence in a sliding window problem," *Computing and Informatics*, vol. 31, no. 6, pp. 1217–1233, 2013.

[16] github.com/vitoFantasy/lis_stream/.

[17] Y. Li, L. Zou, H. Zhang, and D. Zhao, "Computing longest increasing subsequences over sequential data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 181–192, 2016.

[18] 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*, vol. 11, no. 1, pp. 1–27, 2007.

[19] 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*. IEEE, 1999, pp. 56–67.

[20] T. Shibuya and I. Kurochkin, "Match chaining algorithms for cdna mapping," in *International Workshop on Algorithms in Bioinformatics*. Springer, 2003, pp. 462–475.

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

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

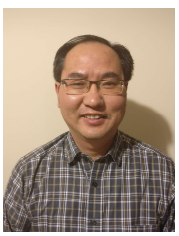
[23] D. Liben-Nowell, E. Vee, and A. Zhu, "Finding longest increasing and common subsequences in streaming data," in *Journal of Combinatorial Optimization*, vol. 11, no. 2, 2006, pp. 155–175.

[24] S. Bespamyatnikh and M. Segal, "Enumerating longest increasing subsequences and patience sorting," *Information Processing Letters*, vol. 76, no. 1-2, pp. 7–11, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019000001241

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

Youhuan Li received his B.S. degree in Electricity Engineering and Computer Science from Peking University in 2013. He is currently pursing the Ph.D. degree at Institute of Computer Science and Technology of Peking University, focusing on streaming data management.

Lei Zou received his B.S. degree and Ph.D. degree in Computer Science at Huazhong University of Science and Technology (HUST) in 2003 and 2009, respectively. Now, he is an associate professor in Institute of Computer Science and Technology of Peking University. His research interests include graph database and semantic data management.



Huaming Zhang received his B.S. degree in Mathematics at Anhui Normal University in 1992. He received his MA degree and Ph.D. degree in Computer Science and Engineering at State University of New York at Buffalo in 2002 and 2005, respectively. He is now an associate professor in the Department of Computer Science at University of Alabama in Huntsville. His research interests include algorithm design and graph theory.

Dongyan Zhao received the B.S. degree, M.S. degree and Ph.D. degree from Peking University in 1991, 1994 and 2000, respectively. Now, he is a professor in Institute of Computer Science and Technology of Peking University. His research interest is on information processing and knowledge management.