IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

Longest Increasing Subsequence  
Computation over Streaming Sequences

YOuhuan Li[[1]](#footnote-0), Lei Zou1, Huaming Zhang2, Dongyan Zhao1

1 Peking University, China;

2 University of Alabama in Huntsville,USA

1 (liyouhuan,zoulei,zhaody}@pku.edu.cn, [2](mailto:2hzhang@cs.uah.edu)[hzhang@cs.uah.edu](mailto:2hzhang@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 increasing1 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 a), computing LIS has been considered in the streaming model [4], [5]. Given an infinite time­evolving sequence =修1,..., *a^} (a* e R), we con­tinuously compute LIS over the subsequence induced by the time window {*a;*-\_(*w*-1), *aj*\_(*w*\_2),…，*ai*). The size of the time window is the number of items that spans in the data stream. Consider a sequence *a* = {3,9,6,2,8,5,7} under window *W* in Figure 3 (in Section 2). There are four LIS in a: {3, 6, 7}, {3, 6, 8}, {2,5,7} and {3,5,7}. Besides LIS enumeration, we introduce some important features of LIS,

1. 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.

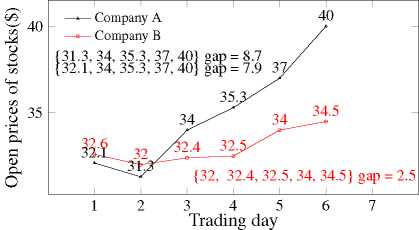


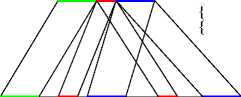
Fig. 1: LIS with different gaps of stock price sequence

**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 a of the stock prices within a period, an LIS of a 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

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING



Genomic sequence *L -*

{1, 21, 31}

{1, 21, 32}

{1, 22, 32}

1 2 3

1 21 31

22 32

Fig. 2: Biological Sequence Alignment

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.

Query transcript/protein *Q*

**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 21 and 22. 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 (1,21,31} 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. 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* algorithms2. 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.
5. Problem Formulation

Given sequence *a* =(们, *叽 ---*,*an},* an increasing subsequence *s* of *a* is a subsequence whose elements are sorted in order from the smallest to the biggest. An increasing subsequence *s* of *a* is called a Longest Increasing Subsequence (LIS) if there is no other increasing subse­quence *s'* with |*s*| < |*s*z|. A sequence *a* may contain multiple LIS, all of which have the same length. We denote the set

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

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

of LIS of *a* by *LIS* (a). For a sequence *s* the head and tail item of *s* are denoted as *S* and *S*, respectively. We use |*s*| to denote the length of *s*

*ai a2 a a.4 a a*

[3 9 6 2 8 5 7]

**I\*** Time Window **J**

Slope-constrainedLIS(*p* = 1.5): *{a4 = 2,a^ =* 5, *a =* 7} Range-constrainedLIS( *L = 2,U = 3,Lv = 1,Uv = 3* ): {*a*】=3死=6, *=* 8}

LIS with Maximum Gap: {*a*】=3,*a*3 = 6, *a$ =* 8},{*a*4 = *2a =* 5, *aq =* 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 aTO = {*a*i,…,*a^} (ai* e R). In the sequence aTO, each *a* has a unique position *i* and *ai* occurs at a corresponding time point *ti*, where *" < tj* 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*i, - - - , *a*TO}, and *W* slides a single unit of position per move towards *a*TO continually. We denote the size of the window *W* by *w*, which is the number of items within the window. During the time *[ti, ti*+1), items of *a* within the sliding time window *W* induce the sequence {*ai*\_(*w*\_i),*ai*\_(*w*\_2),...,*ai*}, which will be denoted by *a(W, i*). Note that, in the sliding window model, as the time window continually shifts towards *a*TO, 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* aTO = {*a*i,…，*a*TO} *and a time window W of size w,* LIS-enumeration *is to report LIS (a(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 a be a sequence, s be an LIS in LIS (a). The* weight *of s is defined a^^a.es ai, i.e., the sum of all the items in s, we denote it by weight(s).*

**Definition 3. *(Gap)****. Let a be a sequence, s be an LIS in LIS (a). The* gap *of s is defined as gap(s) = s* 一 *s”, i.e., the difference between the tail st and the head sh of s.*

**Definition 4. *(Width)****[12]. Let a be a sequence, s be an LIS in LIS (a) where s = 岡,a’：，…,aij (k =* |*s*|*). The* width *of s is defined as width(s) = ik* 一 *i*i*, i.e., the positional distance between the tail itemS) and the head item/Q of s.*

**Definition 5. *(Computing LIS with Constraint)****. Given a time-evolving sequence* aTO = {*a*i,…，*a*TO} *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* a(*W*, *ti*)*: s = {aii ,ai：，…,aim* }*.*

* *s is an* ***LIS with Maximum/Minimum Weight*** *if s has maximum/minimum weight among all LIS in LIS* (a(*W*, *ti*))*.*
* *s is an* ***LIS with Maximum/Minimum Gap*** *if s has maximum/minimum gap among all LIS in LIS* (a(*W*, *ti*))*.*
* *s is an* ***LIS with Maximum/Minimum Width*** *if s has maximum/minimum width among all LIS in LIS* (a(*W*, *ti*))*.*
* *s is a* ***Slope-constrained LIS(SLIS)*** *if for a nonnega­tive slope boundary p and all i < k < m, the slope between two consecutive items in s is not less than p,i.e, ~~D：~~ > p.*
* *s is a* ***Range-constrained LIS(RLIS)*** *if for two ranges [Li, Ui*] *and [Lv, Uv*] *where* 0 < *Li* < *Ui < n,* 0 < *Lv* < *Uv, Li* < *ik+i \_ ik* < *Ui and Ly* < *aik*+i 一 *aik* < *Uv (i* < *k < m).*

A running example that is used throughout the paper is given in Figure 3, which shows a time-evolving sequence aTO and its first time window *W*. The induced sequence within the time window is a = {*a*i = 3, *a： =* 9, *a^ = 6, a*4 = 2, *a*s = 8, *a(s =* 5, *a*? = 7}. There are four LIS in a: {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 =* i.5 (in Figure 3), then {3, 6, 8} does not satisfy the slope constraint since the slope between *a^ =* 6 and *as =* 8 is (8 一 6) /(5 一 3) =i < i.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 [*LI =* 2, *Ui =* 3] and [*Ly* = i, *Uy =* 3], {2,5,7} and {3,5,7} do not satisfy the range constraints since the positional distance between *a^ =* 5 and *a^ =* 7 is i < *Li =* 2. Similarly, *a*3 *=* 6 and *a*7 *=* 7 cannot constitute a range-constrained LIS either since 7 一 3 = 4 > *Ui =* 3. {3,6,8} is the only range-constrained LIS.

1. Quadruple Neighbor List La
   1. La—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 a *=* {*a*i , *a*2,..., *aw*} and *ai, aj* e a. *ai* is said to be *compatible a* with *aj* if *i* < *j* and *ai* < *aj.* We denote it by *ai W aj.* Also, we use *IS a(ai)* to denote the set of all *increasing subsequences* of a that ends with *ai* and we define *rising length* [5]3 of *ai,* denoted as *RLa(ai),* as the maximum length of subsequences in *iS* a(*ai*). For example, consider the sequence a *=* {*a*i *=* 3, *a*2 *=* 9, *a*3 *=* 6, *a*4 *=* 2, *a*5 *=*

1. *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].

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

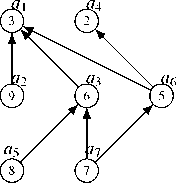
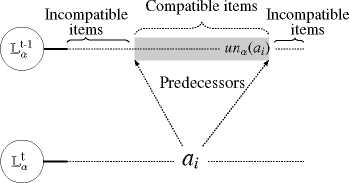
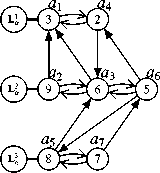
Fig. 4: Horizontal lists and QN-List

(a) Horizontal lists

(b) QN-List La

Fig. 5: Sketch of predecessors

Fig. 6: DAG



8, *a* = 5, *a* = 7} in Figure 3. Consider *a* = 8. There are five increasing subsequences{*a*5 = 8), *{ai =* 3, *a =* 8), {*a*3 = 6, *a5* = 8), {*a*4 = 2, *a*5 = 8), {*a*1 = 3, *a*3 = 6, *a*5 = 8) that end with *a、.* The maximum length of these increasing subsequences is 3. Hence, *RLa(a5)=* 3.

**Definition 6.** *(****Predecessor****). Given a sequence a and ai* e *a, for some item aj, aj is a* predecessor *of ai if*

*a*

*aj < ai AND RLa(aj) = RLa(ai) -* 1

*and the set of predecessors of ai is denoted as Preda(ai).*

In the running example in Figure 3, *a^* is a predecessor *a*

of *a5* since *a3 < a*、and *RLa(a3)(= 2) = RLa(a5)(=* 3) - 1. Analogously, *a*i is also a predecessor of *a^.*

With the above concepts, we introduce four neighbours for each item *a、*as follows:

**Definition 7.** *(****Neighbors of an item****). Given a sequence a and ai e a, a’ has up to four neighbors.*

1. ***left neighbor*** *lna(ai): l^a(ai) = aj if aj is the* nearest *item* before *a’ such that RLa(ai) = RLa(aj).*
2. ***right neighbor*** *rna(ai): rna(ai*) = *aj if aj is the* nearest *item* after *ai such that RLa(ai) = RLa(aj).*
3. ***up neighbor*** *una(ai): una(ai) = aj if aj is the* nearest *item* before *ai such that RLa(a j)* = *RLa(ai)* - *1.*
4. ***down neighbor*** *dna(ai): dna(ai) = aj if aj is the* nearest *item* before *ai such that RLa(aj) = RLa(ai)* + 1*.*

Apparently, if *ai = lna(aj)* then *aj = rna(ai).* Besides, we know that left neighbor (also right neighbor) of item *ai* has the same rising length as *ai* 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 *a* is presented in Figure 4a. **Definition 8.** *(****Horizontal list****). Given a sequence a, con­sider the subsequence consisting of all items whose rising lengths are k: Sk =岡,ai》,...,aik}, i1 < i2ik. We know that for 1 < k < k, a^k> = 叽⑷八)and aig = rnag*)*. We define the list formed by linking items in Sk together with left and right neighbor relationships as a horizontal list, denoted as* L*ka.*

Apparently, for V *ai* e L%, predecessor of *ai* must be in L尸*((*> 1).

**Definition 9.** *(****Quadruple Neighbor List (QN-List)****). Given a sequence a =* (*a*1,..., *aw}, the quadruple neighbor list over a (denoted as* L*a) is a data structure containing all horizontal lists (Definition 8) of a and each item ai in* La *is also linked directly to its up neighbor and down neighbor. In essence,* L*a is constructed by linking all items in a with their four kinds of neighbor relationship. Specifically,* |La| *denotes the number of horizontal lists in* L*a.*

Figure 4b presents the QN-List La of running example sequence *a* (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 *a* is exactly the number of horizontal lists in L*a*. Besides, QN-list over sequence *a* of *w* items costs only *O(w)* space.

* 1. La—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 B4.

**Lemma 1.** *Let a = {a1* , *a*》，…，*aw} be a sequence. Consider two items ai and aj in a horizontal list* L*ta.*

1. *Items in each horizontal list* L*ta are monotonically decreasing while their subscripts (i.e., their original position in a) are monotonically increasing from the left to the right.*
2. V*ai* e *a, all predecessors of ai form a nonempty consecutive block in* LJ-1 *(t >* 1) *and una(ai) is a' 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 *ai* e L*ta* form a consecutive block from *una(ai)* to the left in LJ-1 (Lemma 1(2)). **Lemma 2.** *Given sequence a and* La*,* V *ai e La :*

1. *RLa(ai) = t if and only if ai* e L*ta.*
2. *una(ai)(if exists) is the rightmost item in* LJ-1 *which is before ai in sequence a.*
3. *dna(ai)(if exists) is the rightmost item in* L^1 *which is before ai in sequence a. Besides, dna(ai) > ai.*

For example, in Figure 4b, *a3* is the rightmost item that is before *a、*in L;a *(RLa(a5) =* 3) and *una(a5)= a^.* **Lemma 3.** *Given sequence a and its* La*, for 1* < *i, j* < |La|

*Tail(Lla)* < *Tail(L*) *一 i* < *j*

*where Tail(Lla) denotes the last item in list Lla.*

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

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

(c) 3,9,6

(d) 3,9也

(e) 3,9,6,2,8

3, *a2*

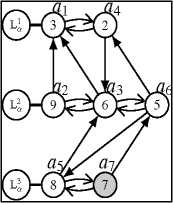
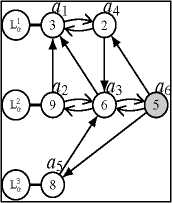
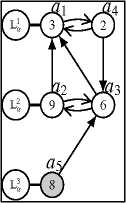
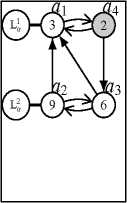
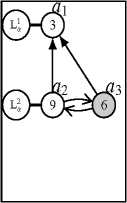
(f) 3,9,62,8,5

9, *a =* 6, *=* 2, *a.5*

(g) 3,9,62,8,5】

8, *a* = 5, *a* = 7}

**Fig. 7: Example of building QN-List for sequence: {***a*i



1)

2)

3)

4)

For example in Figure 4b, we can see that the sequence consisting of tail items: {*a*4 = 2, *a6 =* 5, *a^ =* 7} is in ascending order.

* 1. La—Construction

The construction of La over sequence a lies in the determination of the four neighbors of each item in a. We discuss the construction of La as follows. Figure 7 visualizes the steps of constructing La for a sequence a.

**Building QN-List** La**.**

Initially, four neighbours of each item *ai* are set NULL; Step 1: Lia is created in La and *a*i is added in Lia;

Step 2: if *a2 < a*i, it means *RLa(a2)= RLa(ai) =* 1. Thus, we append *a*2 to Lia. Since *a*2 comes after *a*i in sequence a, we set *rna(ai) = a?* and *lna(a2)= a*i.

If *a*2 > *a*i, we can find an increasing subsequence (*a*i,*a*2), i.e, *RLa(a2)=* 2. Thus, we create the second horizontal list L2a and add *a*2 to L2a. Furthermore, it is straightforward to know *a*i is the nearest predecessor of *a*?； So, we set *una(a2)= a*「

(By the induction method) Step *i*: assume that the first *i - i* items have been correctly added into the QN- List (in essence, the QN-List over the subsequence of the first *(i - i)* items of *a* is built), let's consider how to add the *i*-th item *ai* into the data structure. Let *m* denote the number of horizontal lists in the current La. Before adding *ai* into La, let's first figure out the rising length of *a、*Consider a horizontal list L；, we have the following two conclusions:

1. If *Tail*(La)> *ai,* then *RLa(ai) < t.* Assume that *RLa(ai) > t.* It means that there exits at least one

a

item *aj* (e La) such that *aj < ai,* i.e., *aj* is a predecessor (or recursive predecessor) of *ai*. As we know *T ail*(L*t*a) is the minimum item in L*t*a (see Lemma 2). *Tail(L*) > *ai* means that all items in a

La are larger than *a、*That is contradicted to *aj < ai* A *aj e* La. Thus, *RLa(a[)* < *t*.

1. If *T ail*(L*t*a) < *ai*, then *RL*a(*ai*) > *t*. Since *T ail*(L*t*a) is before *ai* in a and *T ail*(L*t*a) < *ai*, *T ail*(L*t*a) is compatible *ai*. Let us consider an increasing subseqeunce *s* ending with *Tail*(La), whose length is *t* since *T ail*(L*t*a)'s rising length is *t*. Obviously, *s' = s* ㊉ *ai* is a length-(t+1) increasing subsequence ending with *ai*. In other words, the rising length of *ai* is at least *t +* i, i.e, *RLa(ai) > t.*

Besides, we know that *Tail(L^) > Tail(L，)* if *t > t*(see Lemma 3). Thus, we need to find the first list L*t*a whose tail *T ail*(L*t*a) is larger than *ai*. Then, we append *ai* 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., *T ail*(La*m*) *< ai*, we create a new empty list *T ail*(La*m+*i) and insert *ai* into *T ail*(La*m+*i).

According to Lemma 1, it is easy to know *ai* can only be appended to the end of L*t*a, i.e., *rn*a(*T ail*(L*t*a)) *= ai* and *ln*a(*ai*) *= T ail*(L*t*a). Besides, according to Lemma 2(2), we know that *una(ai)* is the rightmost item in L}1 which is before *ai* in a, then we set *una(ai)= Tail*(LJ-i) . Analogously, we set *dna(ai) = TaiKL^1).* So far, we correctly determine the four neighbors of *ai*. We can repeat the above steps until all items are inserted to La. Pseudo codes for building the QN-List La are presented in Algorithm 1 in Appendix A.

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

3.4 LIS Enumeration

Let's discuss how to enumerate all LIS of sequence a based on the QN-List La. The last item of each LIS must be located at the last horizontal list of La and we can enumerate all LIS of *a* by enumerating all |La | length increasing subsequence ending with items in L?. For convenience, we use *MIS a(ai)* to denote the set of all *RL*a(*ai*) length increasing subsequences ending with *ai*. Consider each item *ai* in the last list Laa|. We can compute all LIS of a ending with *ai* by iteratively searching for predecessors of *ai* in the above list from the bottom to up until reaching the first list Lia. 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 La. The DAG is defined based on the predecessor relationships between items in a. Each vertex in the DAG corresponds to an item in a. A directed edge is inserted from *ai* to *aj* if *aj* is a predecessor of *ai* (*ai* and *aj* is also called parent and child respectively).

**Definition 10.** *(DAG G(a)). Given a sequence a, the directed graph G is denoted as G(a) = (V, E), where the vertex set V and the edge set E are defined as follows:*

*V = (ai\ai e* a); *E = ((ai, aj)\aj is a predecessor of ai}*

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

The *G(a)* over the sequence a = {3, 9, 6, 2, 8, 5, 7} is presented in Figure 6 where each path of length |La| corresponds to an LIS. For example, we can find a path *as =* 8 t *a*s = 6 *— ai =* 3, corresponding to LIS (3,6,8). Thus, we can easily design a DFS-like traverse starting from items in L^' to output all path with length |La| in *G*(a).

Note that we do not actually need to build the DAG in our algorithm since we can equivalently conduct the DFS-like traverse on La. Firstly, we can easily access all items in La 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 *ai* by searching LJ-1 from *una(ai*) to the left until meeting an item *a*\* that is not compatible with *ai.* All touched items (*a*\* excluded) during the search are predecessors of *ai*.

We construct LIS *s* from each item *aim* in L*\** (i.e., the last list) as follows. *aim* 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 L% 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 L*\** 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.*

1. Maintenance

When time window slides, *ai* is deleted and a new item *aw+i* is appended to the end of *a.* It is easy to see that the quadruple neighbor list maintenance consists of two operations: deletion of the first item *a*i and insertion of *aw*+i 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*?, - - - , *aw}* formed by deleting *a*i from a is denoted as a-. 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-Hoip Up Neighbor)****. Let a* = {*a*i ,*a*?,*aw} be a sequence and* La *be its corresponding quadruple neighbor list. For Vai* £ *a, the* k-hop up neighbor *un!a(ai*) *is defined as follows:*

*k ( ) =* **j** *ai k =* 0

*una(ai)* = *\una(unk-i(ai*)) *k > i*

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 La for the sequence a in the running example. After deleting *a*i, some items in La (1 < *t < m)* should be promoted to the above list LJ-1 and the others are still in L*t*a. Theorem 2 tells us how to distinguish them. In a nutshell, V *a* e La (1 < *t < m*), if its *(t-* i)-hop up neighbor is *a*i (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*i. *{a?, as}* are in L;a and their 1-hop up neighbors are *a*i (the item to be deleted), thus, they are promoted to the first list of La-. Also, {*a*s} is in L\* whose 2- hop up neighbor is also *a*i. It is also promoted to L\*\_. More interesting, for each horizontal list La (1 < *t < m*), the items that need to be promoted are on the left part of L*t*a, denoted as *Left(La),* which are the shaded ones in Figure 8(a). Note that *Left(L^) =* {*a*i}. The right(remaining) part of La is denoted as *Right(La).* The horizontal update is to couple *Left(Lt+i)* with *Right(La)* into a new horizontal list La-. For example, *Left(LO) = {a?, a*s} plus *Right(L\*) =* {*a*^ to form L\*\_ = {*a*?, *as, 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*i. Algorithm 3 in Appendix A studies how to find the separatrix to divide each horizontal list L*t*a- 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 a =* {*a*i*aw} be a sequence and* La *be its corresponding quadruple neighbor list. Let m be the number of horizontal lists in* La*. Let ai and aj be two items in* La, *t >* 1*. If ai is on the left of aj, un^ai) = un%(aj) or una(ai) is on the left of u^(aj), for every* 0 < *k < t.*

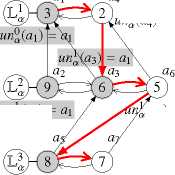
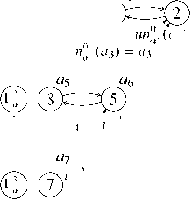
**Theorem 2.** *G^^en a sequence a = {a\, a?, - - -* , *aw} and* La*. Let m =* |La|*. Let* a- = *{a?, - - -* , *aw} be obtained from a by deleting ai. Then for any ai,* 2 < *i* < *m e* La, 1 < *t* < *m,*

1. *If untJi(ai) is ai, then RLa-(ai*) = *RLa(ai)* 一 1*.*
2. *If untJi(ai) is not ai, then RLa-(ai) = RLa(ai).*

Naive method. With Theorem 2, the straightforward method to update horizontal lists is to compute *u^~x(ai*) for each *ai* in L*t*a. If *unt*a-*i(ai)* is *ai*, promote *ai* into L*t*a-*i*. 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 La- for sequence a-.

Optimized method. For each item *ai* in L*t*a (*i* < *t* < *m*) in the running example, we report its *(t -* i)-hop up neighbor in Figure 8a. The shaded vertices denote the items whose *(t -* i)-hop up neighbors are *a*i in La; and the others are in the white vertices. Interestingly, the two categories of items

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING



皿「（*。*2）=*。*2

2

况-H 7

Lt-H 8

(*a*5) = *a*4

*wn*a\_(*a*6) = *a*4

*6*

*"*底（*。*5）= *a* 展（*。*7）= *«*4

(a) Division.

(*a*6)= *a4*

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 *ai,* for each *a* e L；, if is *a*i,

then for any item *aj* at the left side of *ai* in L；, *wn*j-1(*aj*) is also *a*i. While, if *unt-1(ai)* is not *a*i, then for any item *ak* at the right side of *ai* in L%, *unt~i(ai)* is not *a*i. 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*i, we can divide each list L； into two sublists: *Left(LO)* and *Right").* For any item *aj* e *Left(L^o), unt-i(aj)* is *a*i while for any item *ak e Right*(L；), *unt-i(ak)* is not *a*i. Instead of computing the *(t -* i)-hop up neighbor of each item, we propose an efficient algorithm (Algorithm 3 in Appendix A) to divide each horizontal list La into two sublists: *Left(La)* and *Right*(L；).

Let's consider the division of each horizontal list of La. In fact, in our division algorithm, the division of L； depends on that of L；-1. We first divide L；. Apparently, *Left(L'ix)* =(*a*i} and *Right(L；)* = (L；) 一 (*a*i}. Recursively, assuming that we have finished the division of L；, i < *t < m,* there are three cases to divide L；1. Note that for each item *ai e Left(L；), unt-i(ai) = a*i； while for each item *ai e Right(L；), unt<-i(ai), a*i.

1. If *Right(L；) = NULL,* for V *aj e* L； i, we have *un；(aj) e Left*(L；), thus, *un；(aj)* is exactly *a*i. Thus, all below lists are set to be the left part. Specifically, for any *tf > t,* we set *Left(L；*) = L； and *Right(L；*) = *NULL.*
2. If *Right(L；), NULL* and *HEAD(Right(L；))* is *ak：*
3. if *dn；(ak)* does not exist, namely, L；1 is empty at the time when *ak* is inserted into L*t*；, then all items in L*t*； i come after *ak* and their up neighbors are either *ak* or item at the right side of *ak*, thus, the *t*-hop up neighbor of each item in L*t*； i cannot be *a*i . Actually, all below lists are set to be the right part. Specifically, for any *t > t,* we set *Left(L；) =NULL* and *Right(L；*) = L；.
4. if *dn；(ak)* exists, then *dn；(ak*) and items at its left side come before *ak* and their up neighbors can only be at the left side of *ak* (i.e., *Le f t(*L*t*；*)*), thus, the *t*-hop up neighbor of *dn；(ak*) or items on the left of *dn*；*(ak)* must be *ai* . Besides, items at the right side of *dn；(ak*) come after *ak*, and their up neighbors is either *ak* or item at the right side of *a.,* thus, the *t*-hop up neighbor of each item on the right of *dn*； cannot be *ai* . Generally, we set *Le f t(*L*t*； *i)* as the induced sublist from the head of L*t*； *i* to *dn*；*(ak)*(included) and set *Right(*L*t*； *i)* as the remainder, namely, *Right(*L*t*； *i)* = L*t*； *i* -*Le f t(*L*t*； *i)*. We iterate the above process for the remaining lists.

Finally, for *i* < *t* < *m*, the left sublist *Le f t(*L*t*；*)* should be promoted to the above list； and *Right(*L*t*；*)* is still in the *t*-th list. Specifically, we append *Right(L；\_)* to *Left(L；-i)* to form L；-. In the running example, we append *Right(L；)=* (*a*2, *a*s) to *Left(L；*) = (*a*4)to form L；\_ = (*a*?, *a*s, *a*4), as shown in Figure 8b.

**Theorem 3.** *The list formed by appending Right(L；*) *to Left(L；*i) *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*(L*t*；) to *Le f t*(L*t*； i), denoted as *L*, contains the same set of items as L*t*；- does. Besides, according to Lemma 1(2) and Theorem 3, both *L* and L*t*；- are monotonic decreasing, thus, we can know that *L* is equivalent to L*t*；- and we can derive that the horizontal list adjustment method is correct.

*unl\_(a7) =。*4

**(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 L； to L；-. Before presenting our method, we recall Lemma 2(2), which says, for item *ai e* L；, *un；(ai)(if* exists) is the rightmost item in L；-1 who is before *ai* in sequence ；; while, *dn*；*(ai)*(if exists) is the rightmost item in L*t*； *i* who is before *ai* in sequence ；.

**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 L； into L；-. Generally, when we promote *Le f t(*L*t*；*)* to the above level, we need to change their up neighbors but not down neighbors. While, *Right(*L*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(^* = (*a*s) is promoted to the L；\_. In L；, *un；(as)* is *as,* but we change it to *un；-(as) = a*4, i.e., the rightmost item in L；\_ who is before *as* in sequence ；-. Analogously, *Right(L) = (a^)* is still at the second level of L；-. *dn*；(*a*6) is *a*s, but we change it to null, since there is no item in L；\_ who is before Formal analysis and algorithm description of the vertical update are as follows. **Analysis and Algorithm.**

**Lemma 5.** *Given a sequence ； and* L；*, for any 1* < *t* < *m:*

1. *Vai e Left(L；), dn；(ai) (if exists) e Left(L；i).*
2. *Vai e Right(L；i), un；(ai*) *(if exists) e Right(L；).*

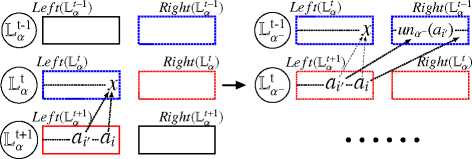
**Lemma 6.** *Let ； = (ai, a?, - - - , aw) be a sequence. Let* L； *be its corresponding quadruple neighbor list and m be the total number of horizontal lists in* L；*. Let* = *(a?, - - - , aw} be obtained from ； by deleting ai. Consider an item ai e* L*t*；-*, where i* < *t* < *m. According to the horizontal list adjustment, there are two cases for ai: ai is from Le f t(*L*t*； *i ) or ai is from Right(*L*t*；*). Then, the following claims hold:*

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

1. *Assuming ai is from*
2. *dna- (ai)* = *dna(ai) (the down neighbor remains).*
3. *Let x be the rightmost item of Left(L»). f una(ai*)*, x, una- (ai) = una(a.i) (the up neighbor remains).*
4. *Assuming ai is from Right(L%)*
5. *una-(ai) = una(a.i) (i.e., the up neighbor remains).*
6. *Let y be the rightmost item of Left(L^1). If dna(ai) , y, dna-(ai) = dna(ai) (i.e., the do^^^ neighbor remains)*

With Lemma 6, for an item *ai* e L%\_, there are two cases that we need to update the vertical neighbor relations of *a、*

1. Case 1: *ai* is from *Left(L0~1).* Let *x* be the rightmost item of *Left(L^).* We need to update the *up neighbor* of *ai* in La- if *una(ai) = x* as shown in Figure 9.
2. Case 2: *ai* is from *Right(L%).* Let *y* be the rightmost item of *Left(L0+1).* We need to update the *down neighbor* of *ai* in La- if *dna(ai) = 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(L:c+l).* According to Theo­rem 2, *Left(Lt+1)* will be promoted into the list L"

Let *ai* be the rightmost item of *Left(Lt0+i)* and *x* = *Tail(Left(Lta)).* According to Lemma 6(1.b), if *una(ai)*更 *x*, then *una(ai) = una-(ai).* It is easy to prove that: If *una(a.i)黄 x* then *una(aj)黄 x*, where *aj* is on the left of *ai* in *Left(L^1).* Thus, all items in *Left(L^1)* do not change the vertical relations (see Lines 6-8 in Algorithm 4).

Now, we consider the case that *una(ai) = x.* Let *a^* denote the leftmost item in *Left(Lt0+i)* where *una(ai>*) is *x*. The up neighbors of the items in the consecutive block from *a「*to *ai* (included both) are all *x* in La (note that *x* is the rightmost item in *Left(LO)* ), as shown in Figure 9(a). These items' up neighbors need to be adjusted in La-.

**Theorem 4.** *Given an sequence a and* La*, assume that una(ai) = x and a? denote the leftmost item in* 匚£烏°’ *where una(ai<) is x. If dna(x), NULL, then a^ is exactly rna(dna(x)); otherwise, av is HEAD(LRLa^ai')-*

With Theorem 4, we can easily find out *a^* in *O(1)* time. Then, we firstly adjust the up neighbor of *a^* in La-. Initially, we set *a*\* = *una(ai>*) = *x*. Then, we move *a*\* to the right step by step in L^1 until finding the rightmost item whose position is before *a?* in sequence a-. Finally, we set *una-(a/) = a*\* (see Line 14 in Algorithm 4 in Appendix A).

In the running example, when deleting *a1* in Figure 8a, *Left(LO) = {a5},* and *una(as)* is exactly the tail item *a^* of *Left(Ll),* since L\*\_ is *{a^* = 9, = 6, = 2), formed by

appending *Right(L0)({a2 = 9, a^ =* 6)) to *Left(L%) ({a4 =* 2)), and *a4* is the rightmost item in L\*\_ who is before *as* in *a-,* then we set *una-(as)* as *a4 = 2* (see Figure 8b).

Iteratively, we consider the items on the right of *a^*. 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*(|LU |), since each item in L'-1 is scanned at most once.

*a*

**Case 2:** Consider all items in *Right(L»).* According to the horizontal adjustment, the down neighbors of items in *Right(L^)* are the tail item (i.e., the rightmost item) of *Left(L^1)* or items in *Right(L^1).*

Actually, Case 2 is symmetric to Case 1. We highlight some important steps as follows. Let *ai* be the leftmost item in *Right(L)* and let *y* be *Tail(Left(L:c+1)),* namely, *y* is the rightmost item in *Left(Lt0+i).* Obviously, *dna(ai)= y* (Algorithm 3). Then we scan *Right(L*) from *ai* to the rightmost item *a^* where *dna(ai>*) is *y*. The up neighbors of the items in the consecutive block from *ai* to *a^* (included both) are all *y* (see Figure 10(a)). Items on the right of *a^* need no changes in their down neighbors, since their down neighbors in La are not *y* (see Lemma 6(2.b)).

We only consider the consecutive block from *ai* to *a^* (see Figure 10) as follows. First, we adjust the down neigh­bor of *a?* in La-. Initially, we set *a*\* = *TaiK^Left(U^}),* i.e., the rightmost item of *Left(Lt0+i).* Then, we move *a*\* to the left step by step in LR1 until finding the rightmost item whose position is before *a^*. Finally, we set *dna- (ai，*) = *a*\* (see Line 8 in Algorithm 5 in Appendix A).

In the running example, when deleting *a1 =* 3, *Right(L%*) is *{a4 = 2)* whose head item is *a*4. And *d^a(a4)* is *a^ = 6* that is the tail item of *Left(LO).* Then, initially, we set *dna- (a4)* as the tail item of *Left(LO),* namely, *dna- (a4) = as* and scan L\*\_ from the right to the left until finding a rightmost item who is before *a4* in a-. Since there is no such item in L\*\_, we set *dn- (a4)* as *NULL.*

*&*奶皿%)

I *件舟*i (^0^1 *s)&iaL—*

*— y* ―►(^0^5"纨-(£). I I

*Left*(L*t*+2) *Right*(L*t*+2)

……

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

**Fig. 10: Case 2: updating down neighbors**

Iteratively, we consider the items on the left of *a^*. 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*(|L0-1|), since L^1 is scanned at most twice.

Finally, we can see that solution to handle the deletion of the head item *a1* in sequence *a* consists two main phrase. The first phrase is to divides each list L； (1 < *t < m* and then finishes the horizontal update by appending *Right(L:a*) to *Left(U^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.*

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 *a* and La. 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.

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 *%,* the predecessors of *a* who satisfy the slope constraints are called *slope-proper* predecessors of *a、*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*(|a|) 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 *ai* that has only one slope-proper predecessor *aj,* if *aj* has no slope­proper predecessor, *ai* 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 L, are colored white. We iteratively precess other levels from L, to L^1. We color *ai* white if *ai* has at least one slope-proper predecessor that has previously been colored white. After collation, for any white item *ai*, there must exist an increasing subsequence *s* ending with *ai* where each item in *s* is white.

**Theorem 6.** *Given a sequence a and ai, aj* e L^1*. Assume that a’，, aj> e* L； *are the leftmost white slope-proper predecessors of ai and aj,respectively. If aj is at the right side of ai, then a^ is either a^ or at the right side of a^.*

We can know that finding a leftmost white slope-proper predecessor for *ai e* L；1 is enough to confirm that *ai* is white. With Theorem 6, after determining the leftmost white predecessor *a*- of *ai,* searching for the leftmost white predecessor of *rna(ai)* can be conducted from *a*- to the right of L；. Thus, after coloring items in L；, we can color items in L；1 by scanning L； and L； 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 *ai e* L*\* (m* = |L；|) and visit the leftmost white slope-proper predecessor of *ai*. 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*(|a|) time while outputting an SLIS costs *O(l)* time where *l* is the SLIS length. Thus, SLIS computation over L； costs *O(w)* time.

1. Range-constrained LIS(RLIS)

Consider a sequence ； and two ranges *[Lj*, *Ui*] and *[Lv*, *Uv*] where 0 < *Lj < Ui < n,* 0 < *Lv < Uv*. For *a^ e* L； and *ai e* L；1. We call *a?* as *range-proper* predecessor of *ai* if *ai - ai e [Lv*, *Uv*] and *i - i e [Lj*, *Ui*] (Note that since 0 < *Lj* and 0 < *Lv*, then we can easily know that

；

*ai，*< *ai* and *i < i,* namely, *a? W a»*

**Coloration** Similar to the solution of SLIS computation, we also assign color to each item. For each item *ai*, if *RL；(ai)=* 1, *ai* should be white; otherwise, *ai* will be white if and only if *ai* has at least one white range-proper predecessor. Non­white items are called black items. We design an efficient coloring algorithm which costs only linear time. Initially, items in L1； are colored white. We iteratively precess other levels from L； to L；；'. Assuming that we have finished coloring items in the first *t* horizontal lists (i.e.,L；,L；,...,L；), let's discuss how to color items in L； 1.

For an item *ai e* L；1 and *a^ e* L；, if *ai - a? e [Lv*, *Uv*] and *i-if e [Lj*, *Uj*], then equivalently, *ai-Uv < a^* < *a「Lv* and *i - Uj < i < i - Lj*. According to Lemma1(1), items in L*t*； are monotonically decreasing from the left to the right while their positions in ； are monotonically increasing. Thus, for *ai e* L*t*； 1, we can find a leftmost item *al* in L*t*； such that *ai < a「Lv* and *i - Uj < l.* Obviously, for any item *ai* at the right of *ai, ai < a^ - Lv* and *i - Uj < l.* Similarly, we use *ar* to denote the rightmost item such that *ai - Uv < ar* and *r < i - Lj*. Apparently, items from *ai* to *a『*in L； form a consecutive block which contains exactly all range-proper predecessors of *ai* (see Figure 23 in Appendix D). If *ar* is at the left of *al*, then there is no range-proper predecessor of *ai*; otherwise, *al* is the leftmost range-proper predecessor of *ai* while *ar* is the rightmost. However, *al* could be black item, which is useless to RLIS computation. Therefore, we focus on the leftmost white range-proper predecessor of *ai*. **Theorem 7.** *Given a sequence ； and ai e* L；1*, assume that ay is the leftmost white item in* L； *such that a^* < *ai - Lv and i - Uj* < *i. Then, we can conclude that either a^ is the leftmost white range-proper predecessor of ai or ai has no white range-proper predecessors (i.e.,ai should be black).* **Theorem 8.** *Given a sequence* ； *and* L；*. Consider ai,aj e* L；1*. Assume that a^, aj\* e* L； *are the leftmost white predecessors of ai and aj such that a^ < ai-Lv and i-Uj < if and aj> < aj - Lv and j - Uj < j. Then if aj is at the right side of ai, aj，is either a^ or at the right of a^.*

With Theorem 8 and Theorem 7, after determining the leftmost white item *a?* in L； such that *a? < ai - Lv* and *i - Uj < i,* if *ai* is exactly a range-proper predecessor of *ai,* we color *ai* white and set a pointer from *ai* to *a^* as the leftmost white range-proper predecessor; otherwise, if *a^* is not a range-proper predecessor of *ai*, then color *ai* black.

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

No matter what color *a* is, the coloration of *aj* = *rna(ai)* can be conducted by searching for leftmost white item *aj，* from *ai，*to the right such that *aj，< aj* 一 *Lv* and *j - Ui < 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 La cost 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

1. (Intel(R) i7-4790 3.6GHz, 8G). All codes, including those for comparative methods are provided in Github [16].
2. 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 decades5, up to 7400 days. The gene datasets is a sequence of 4,525 matching positions, which are computed over the BLAST output of mRNA sequences6 against a gene dataset7 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.

1. 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 MH- LIS: **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

1. <http://finance.yahoo.com/quote/MSFT/history?ltr=1>
2. ftp://ftp.ncbi.nih.gov/refseq/B\_taurus/mRNA\_Prot/
3. ftp://ftp.ncbi.nlm.nih.gov/genbank/

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.

mxv為dS

103

**102" (sm)UI[**

100

103

Fig. 11: Space

DP YangS

YangR LISone

**o Mu o**

\*DP 0YangS

YangR LISone

QN-list/QN-pre^ LISSET

— VARIANT &MHLIS

1

2 3 4

Window Size

(a) Stock

10°

*-*

QN-list/QN-pre^ LISSET — VARIANT V^MHLIS

12 3 4

Window Size

**(b) Gene**

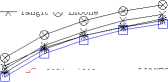
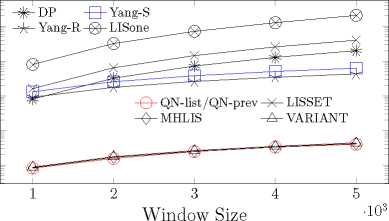
**Fig. 12: Construction**

-103

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



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).

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).

s

皀 100.5

s

日100

QN-list QN-prev

0 0 VARIANT+I/D ■ 0 VARIANT+Re

.目 10°

-503

1

2 3 4

Window Size

s

8 嶷

1

2 3 4

Window Size

(a) Max Weight on Stock

*0) bJ)* U

—

>

V10-1

QN-list QN-prev

& 0 VARIANT+I/D 目 0 VARIANT+Re

1 *W*indjw Sizt *-503*

**(b) Max Weight on Gene**

QN-list QN-prev

, S VARIANT+I/D ® 0 VARIANT+Re

|  |  |  |  |
| --- | --- | --- | --- |
|  | ' -Q-QN-list QN-prev |  | 'QN-list W- QN-prev |
| 105 | 角\_ \*LISSET€-LISone | 105  a | d K LISSET 忌-LISone |
| n  d  苛104 |  | M104 |  |
| p  会103 | 、  \* DP E- YangS \* YangR | |1O3 | 知7^ DP 每 Yan知 |
| -^VARIANT+R^pMnLlS+Re\* |  | \* \*■  VARIANT+R^ ^MHLIS+Re\* |
| 102 | 4 VARIANT+I/D OMHLIS+I/D | 102 | 4 VARIANT+I/D OMHLIS+I/D |
|  | 1 2 3 4 5 |  | 1 2 3 4 5 |
|  | Window Size ,103 |  | Window Size ,103 |
|  | **(a) Stock** |  | **(b) Gene** |

Fig. 13: Maintenance

•1Q5

-503

**(c) Min Weight on Stock**

**Fig. 16: LIS with**

2 . 3 4

Window Size

-503

**(d) Min Weight on Gene**

**Extreme Weight**

Window Size -1Q3

(a) Stock

•1Q5

1.4如 105

1.24.. 105

**6 5**

00

**4 3**

0-

lu-I">BS°ol'-y

lu-I">BS°ol'-y

4ndqMn°-H

1.53」I。5

1.29,

10

0.2273

0.2123

0.2023

唱4 0.1691

-Q-QN-list -G-QN-prev

**4ndqMnoJqH**

**6 5**

*a°*

**4 3**

Q Q

Window Size -1Q3

**(b) Gene**

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

020100

(巻)111EeAV

QN-list

SB QN-prev Dqdp

目目LISSET

102

101

100

11 QN-list SB QN-prev Dqdp

~~罪雷蚩毫書雷fialm,~~

1 2 3 4 5 s 1 2 3 4 5 s

Window Size ,10 Window Size ,10

**(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 win- dow(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

00T-2

loo (Sul)① UI①

QN-list HBQN-prev

0 0 VARIANT+I/D 眼 VARIANT+Re

Window Size

**(a) Max Gap on Stock**

101

s

£

尊10。

H

"T

>

QN-list QN-prev

MHLIS+I/D MHLIS+Re

VARIANT+I/D VARIANT+Re

S

皀

日100

中

bX)

2

中

Ao-\*

11 QN-list 目目 QN-prev

*S* 可 VARIANT+I/^0 VARIANT+Re

1

2 3 4

Window Size

-503

(b) Max Gap on Gene

11 QN-list 0B QN-prev

DeIMHLIS+I/D 恥 MHLIS+Re

如 VARIANT+I/D 目 S VARIANT+R

100

< 10-1

1 2 3 4 5.3 1 2 3 4 5.3

Window Size ,10 Window Size ,10

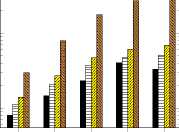
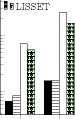
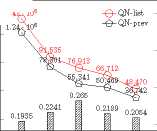
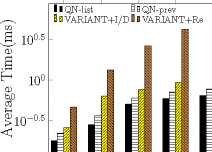
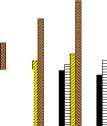
**(c) Min Gap on Stock (d) Min Gap on Gene**

**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 huge8 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



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

are better than the approaches in [14] and the solutions based on LIS enumeration.

Figures 17. We can see that our method outperforms other

methods significantly.

llQN-list 即 QN-prev

10 VARIANT+I/D VARIANT+Re

100.5

8

L

10

*<*

100

102

101

100

s

皀

日100

中

bX)

2

a

<10-1

11 QN-list 目 El QN-prev 0 □ LISone

血 MHLIS+I/D 口 G MHLIS+Re 如 VARIANT+I/D M VARIANT+Re 冃日 YangS & 国 YangR

Dddp

M1

*\M*

~ 2 3 4

Window Size

**(a) Max Width on Stock**

1-0.5

Ji

-5o3

2 3 4

Window Size

**(a) Stock**

(c) Min Width on Stock

Fig. 18: LIS with Extreme Width

(b) Max Width on Gene

1

WmdOv Sizt -503

(d) Min Width on Gene

s

邑

日100

H

中

bX)

冬

V10-1

*s*

g 102

W10

Il QN-list 目日 QN-prev。□ LISone 口口 MHLIS+I/D 口□ MHLIS+Re 如 VARIANT+I/D 目目 VARIANT+Re & I YangS 「

冃 0 YangO D DP n 「

2 3 4

Window Size

**(b) Gene**

**Fig. 21: LIS Length**

5 .IO3

**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

J[io0

8

1 *W*indoW Siz4 膈

(b) Gene

**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.

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(*R*l =

*{L =* 1,*U/* = 20, *Lv =* 0, *Uv =* 50}, *R2 = {L = 20,U =*

40, *Lv =* 50, *Uv =* 100}, *R*3 = *{L/ =* 40,*U/* = 60, *Lv =* 100, *Uv =* 150}) and three different slopes(*S* 1 = 0, *S* 2 = 0.5, *S* 3 = 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

7 Related Work

7.1 Solution Perspective

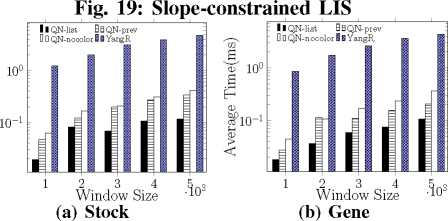
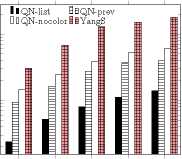
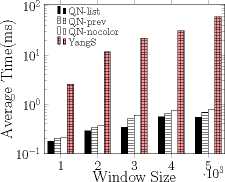
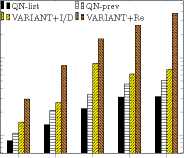
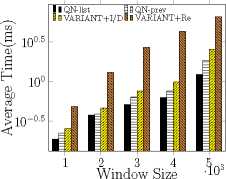
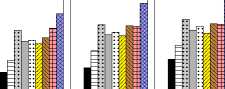
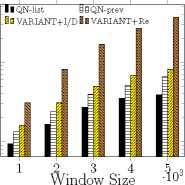
Generally, existing LIS computation approaches can be divided into following three categories:

1. *Dynamic Programming-based.* Dynamic programming is a classical method to compute the length of LIS. Given a sequence *a,* assuming that *ai* denotes the prefix sequence consisting of the first *i* items of a, then the dynamic programming-based method is to compute the LIS of a*i*+1 after computing the LIS of *a、*However, dynamic programming-based method costs *O*(*w*2) time where *w* denotes the length of the sequence a. In [14], the solution for computing range/slope-constrained is also based on dynamic programming, i.e., range/slope-constrained LIS of a*i*+1 is computed from that of *a、*Dynamic programming­based 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 a is exactly the length of LIS in a. 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 *a* in a window, there are *n =* |a| suffix subsequences and the prime idea in [4] is to compress all

1041-4347 (c) 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See <http://www.ieee.org/publications_standards/publications/rights/index.html> for

more

information.



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Method** | **LIS Enumeration** | **LIS with max Weight** | **LIS with min Weight** | **LIS with max Gap** | **LIS with min Gap** | **SLIS** | **RLIS** | **LIS**  **Length** |  | **Methods** | **Space**  **Complexity** | **Time Complexity** | | |
| **Construction** | **Insert** | **Delete** |
| QN-list/QN-prev | *O(OUTPUT*) | *O(OUTPUT*) | *O(OUTPUT*) | *O*(*w* + *OUTPUT*) | *O*(*w* + *OUTPUT*) | *O*(*w*) | *O*(*w*) | *O*(1) | QN-list/QN-prev | *O*(*w*) | *O(w* log *w*) | *O*(log *w*) | *O*(*w*) |
| LISSET [5] | *O(OUTPUT*) | - | - | - | - | - | - | *O*(1) | LISSET [5] | *O*(*w*2) | *O*(*w*2) | *O*(*w*) | *O*(*w*) |
| MHLIS [13] | - | - | - | - | *O*(*w* + *OUTPUT*) | - | - | *O*(1) | MHLIS [13] | *O*(*w*) | *O(w* log *w*) | *O*(log *w*) | - |
| VARIANT [12] | - | *O(OUTPUT*) | *O(OUTPUT*) | *O*(*w* + *OUTPUT*) | *O*(*w* + *OUTPUT*) | - | - | *O*(1) | VARIANT [12] | *O*(*w*) | *O(w* log *w*) | *O*(log *w*) | - |
| DP | *O(OUTPUT*) | - | - | - | - | - | - | *O*(1) | DP | *O*(*w*2) | *O*(*w*2) | *O*(*w*) | - |
| YangS [14] | - | - | - | - | - | *O*(*w*) | - | *O*(1) | YangS [14] | *O*(*w*) | *O(w* log *w*) | *O*(log *w*) | - |
| YangR [14] | - | - | - | - | - | - | *O*(*w*) | *O*(1) | YangR [14] | *O*(*w*) | *O(w* log *w*) | *O*(log *w*) | - |
| LISone [4] | - | - | - | - | - | - | - | *O*(1) | LISone [4] | *O*(*w*2) | *O*(*w*2) | *O*(*w*) | *O*(*w*) |

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

TABLE 2: Data Structure

**TABLE 1: Theoretical Comparison on Online Query**

principle rows of these suffix subsequence into an array, which can be updated in 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\,P2…,Pi,* where *l* is the length of LIS of the sequence. For each item *a* in *Pk (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 *Pl* and then scan items in *P—* (1 < *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 *a* 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.

1. 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 *a* [4], [15], [22], [23], [21]. The second is LIS enumeration, which finds all LIS in a sequence *a* [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 *a* 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 | **/** | **/** | **X** | **X** | **X** | **X** | **/** |
| MHLIS | **X** | **X** | **X** | **/** | **X** | **X** | **/** |
| VARIANT | **X** | **X** | **/** | **/** | **X** | **X** | **/** |
| DP | **X** | **/** | **X** | **X** | **X** | **X** | **/** |
| YangS | **X** | **X** | **X** | **X** | **/** | **X** | **/** |
| YangR | **X** | **X** | **X** | **X** | **X** | **/** | **/** |
| LISone | **/** | **X** | **X** | **X** | **X** | **X** | **/** |

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

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, Cancun, Mexico,* 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. Lopez-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. Balakr- ishnan, 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/3X_2009/>

2009\\_3\\_4\\_art01\\_deorowicz.pdf

1. 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.
2. 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.
3. 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.
4. github.com/vitoFantasy/lis\_stream/.
5. 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.
6. 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.
7. 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.
8. T. Shibuya and I. Kurochkin, “Match chaining algorithms for cdna mapping," in *International Workshop on Algorithms in Bioinformat­ics*. Springer, 2003, pp. 462-475.
9. 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.
10. M. L. Fredman, “On Computation of the Length of the Longest Increasing Subsequences," *Discrete Mathematics,* 1975.
11. 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.
12. 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
13. 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.

1. Increasing subsequence in this paper is not required to be strictly monotone increasing and all items in *a* can also be arbitrary numerical value. [↑](#footnote-ref-0)