Fast Algorithms for the Constrained Longest Increasing Subsequence Problems

I-Hsuan Yang[1] and Yi-Ching Chen[2]
[1]Department of Computer Science
Stanford University, Stanford, CA 94305, USA
ihyang@cs.stanford.edu
[2]Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106
d94010@csie.ntu.edu.tw

Abstract

Let $\langle a_1, a_2, \ldots, a_n \rangle$ be a sequence of comparable elements. In this paper, we study two constrained versions of the longest increasing subsequence (LIS) problem. The first problem is the range-constrained longest increasing subsequence (RLIS) problem. Given $0 < L_I \leq U_I < n$ and $0 \leq L_V \leq U_V$, the objective of the RLIS problem is to deliver a maximum-length increasing subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$ satisfying $L_I \leq i_{k+1} - i_k \leq U_I$ and $L_V \leq a_{i_{k+1}} - a_{i_k} \leq U_V$ for all $1 \leq k < l$. We give an $O(n \log(U_I - L_I))$-time and $O(n)$-space algorithm for solving the RLIS problem. The second problem is the slope-constrained longest increasing subsequence (SLIS) problem. Given a nonnegative slope m, the objective of the SLIS problem is to obtain a maximum-length increasing subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$ satisfying $\frac{a_{i_{k+1}} - a_{i_k}}{i_{k+1} - i_k} \geq m$ for all $1 \leq k < l$. Our algorithm for the SLIS problem runs in $O(n \log r)$ time and $O(n)$ space, where r is the length of an SLIS.

1 Introduction

The longest increasing sequence (LIS) problem is a very classical problem in computer science [15]. Fredman [11] showed a lower bound $\Omega(n \log n)$ of the LIS problem in the decision tree model. Knuth [13] and Schensted [16] gave an optimal time algorithm for this problem where the input is an arbitrary sequence of n numbers. The expected length of an LIS of a random permutation has been shown to be $2\sqrt{n} - o(\sqrt{n})$ [3].

Recently, researchers have found several new applications related to the LIS problem in bioinformatics [8, 10, 18, 21]. For example, MUMmer [8, 9, 14], a large-scale DNA sequence alignment tool, extracts the longest possible set of matches found in the MUM (maximal unique match) alignment by solving an LIS problem. When extending the tool to align three or more sequences, Yang et al. [20] formulated the problem as the longest common increasing subsequence (LCIS) problem which combines the LIS problem with the longest common subsequence (LCS) problem. Further results in this line of investigation can be found in [5, 6, 12].

Albert et al. [2] defined a problem called LISW, which is to find the LIS in sliding windows over a sequence of n elements. Chen et al. [7] solved the LISW problem by maintaining a canonical antichain partition in windows. While dealing with the special case in which the input sequence is a permutation of $\langle 1, 2, \ldots, n \rangle$, Bespamyatnikh and Segal [4] gave an algorithm running in $O(n \log \log n)$ time by utilizing a data structure called van Emde Boas (vEB) tree [19].

In some situations, it might be desirable to impose some constraints between two consecutive elements in the resulting subsequence. For example, we might require two consecutive MUMs or HSPs (high-scoring segment pairs) not too close or too far away [8, 10]. Therefore, in this paper, we define two constrained versions of the LIS problem, namely, the range-constrained longest increasing subsequence (RLIS) problem and slope-constrained longest increasing subsequence (SLIS) problem, respectively. Let A be a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ whose elements are comparable, for example, a sequence of real numbers. An LIS of the sequence A is a subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$ of A with maximum length, where $i_1 < i_2 < \ldots <$

i_l and $a_{i_1} < a_{i_2} < \ldots < a_{i_l}$. The RLIS and SLIS problems are formally defined as follows.

The RLIS Problem:

Input: a sequence of comparable elements $\langle a_1, a_2, \ldots, a_n \rangle$ and $0 < L_I \leq U_I < n$, $0 \leq L_V \leq U_V$

Output: an RLIS $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$, which is a maximum-length increasing subsequence satisfying $L_I \leq i_{k+1} - i_k \leq U_I$ and $L_V \leq a_{i_{k+1}} - a_{i_k} \leq U_V$ for all $1 \leq k < l$

An element can be plotted on a 2-D plane using its index as the x-ordinate and value as the y-ordinate. Consequently, the sequence A would become n points on a 2-D plane and any pair of points makes a slope.

The SLIS Problem:

Input: a sequence of comparable elements $\langle a_1, a_2, \ldots, a_n \rangle$ and a nonnegative slope boundary m

Output: an SLIS $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$, which is a maximum-length increasing subsequence such that the slope between two consecutive points is no less than the input ratio, i.e., $\frac{a_{i_{k+1}} - a_{i_k}}{i_{k+1} - i_k} \geq m$ for all $1 \leq k < l$

Notice that the inputs of the RLIS and SLIS problems are constrained on $0 < L_I \leq U_I < n$, $0 \leq L_V \leq U_V$, and $m \geq 0$, respectively. If $L_V = 0$ or $m = 0$, the output subsequence would be a non-decreasing subsequence. Moreover, our algorithm for the RLIS problem still works if $L_V < 0$, and the output would be another related subsequence where any two consecutive elements may decrease by at most $-L_V$.

The rest of this paper is organized as follows. Section 2 gives an $O(n \log(U_I - L_I))$ time and $O(n)$ space algorithm for the RLIS problem. In Section 3, an $O(n \log r)$ time and $O(n)$ space algorithm is proposed for solving the SLIS problem, where r is the length of an SLIS. Concluding remarks are given in Section 4.

2 The RLIS Problem

The input of the RLIS problem contains an array $A = \langle a_1, a_2, \ldots, a_n \rangle$, $0 < L_I \leq U_I < n$, and $0 \leq L_V \leq U_V$. For any two elements a_i and a_j in A, a_i is said to dominate a_j, or said to be a dominating element of a_j, denoted by $a_i \prec a_j$, if and only if a_j *can* be the successive element of a_i in the subsequence. In this section, we define that $a_i \prec_R a_j$ if and only if $L_V \leq a_j - a_i \leq U_V$ and $L_I \leq j - i \leq U_I$.

2.1 The Algorithm

The *Rank* of element a_i, denoted by $Rank_i$, is defined as the length of an RLIS ending at a_i and computed by the following lemma.

Lemma 1: If $a_j \not\prec_R a_i$ for all j, let $Rank_i = 1$, and $\max_{1 \leq j < i}\{Rank_j | a_j \prec_R a_i\} + 1$ otherwise.

Our algorithm utilizes a data structure called the dynamic RMQ [17] described as follows. Let S be a set of items which is initially an empty set and maintained by an AVL tree [1]. Each item (v, r) of S includes two given values v and r, and can be inserted to or deleted from S by the key v at any time. For any pair (p, q), the maximum r among the items in S whose v value is within the range $[p, q]$ can be queried by this data structure. All operations run in $O(\log |S|)$ time, where $|S|$ is the cardinality of S at the operation time.

In the RLIS problem, let v and r be the value and *Rank* of a_i, respectively. In other words, each item in S has the form $(a_i, Rank_i)$. We define three operations specific to this problem, in which all of them run in $O(\log |S|)$ time by the dynamic RMQ datastructure.

1. *Insert*$(a_i, Rank_i)$ — inserts the item $(a_i, Rank_i)$ into S.
2. *Delete*(a_i) — deletes the item $(a_i, Rank_i)$ from S.
3. *Query*(p, q) — returns the item $(a_i, Rank_i)$ with maximum *Rank* among all elements in S, where $p \leq a_i \leq q$.

The strategy of this algorithm is using the dynamic programming to parse each element a_i from $i = 1$ to n. At the ith iteration, S maintains the set of items $(a_k, Rank_k)$ where the index k is in the range $[i - U_I, i - L_I]$, and $Rank_i$ is computed. Then, any element in S dominates a_i if and only if its value is in the range $[a_i - U_V, a_i - L_V]$. Querying in the dynamic RMQ datastructure with the range $[a_i - U_V, a_i - L_V]$ obtains the element which dominates a_i with maximum *Rank*.

The pseudo code for the RLIS problem is given in Figure 1. At the first step of **for** loop, in order to fit in with the constraint of L_I and U_I, it adjusts the set S by deleting a_{i-U_I-1} and inserting $(a_{i-L_I}, Rank_{i-L_I})$. Secondly, querying the

Algorithm RLIS
Input: $A = \langle a_1, a_2, \ldots, a_n \rangle$, U_I, L_I, U_V, L_V
Output: an RLIS
for $i \leftarrow 1$ **to** n **do**
  if $i - U_I - 1 \geq 1$ **then** $Delete(a_{i-U_I-1})$;
  if $i - L_I \geq 1$ **then**$Insert(a_{i-L_I}, Rank_{i-L_I})$;
  $(a_k, Rank_k) \leftarrow Query(a_i - U_V, a_i - L_V))$
  if $((a_k, Rank_k) = NULL$ **then** $Rank_i \leftarrow 1$;
  else $Rank_i \leftarrow Rank_k + 1$;
    Make backtracking link $(i \rightarrow k)$;
end for
Find j s.t. $Rank_j = \max_{1 \leq i \leq n} Rank_i$;
Output $Rank_j$ and the sequence in reverse order
    by backtracking from j;

Figure 1: An algorithm for the RLIS problem.

range $[a_i - U_V, a_i - L_V]$ returns the element which dominates element a_i with maximum $Rank$. If it returns $NULL$, a_i will be the starting sequence with $Rank = 1$. Otherwise, $Rank_i$ equals to the $Rank$ of returned element plus one, and we make a backtracking link from i to the index of returned element.

After the loop, the maximum value $Rank_j$ in $\{Rank_i | 1 \leq i \leq n\}$ will be the length of an RLIS. Tracing back through the backtracking links from j obtains the reverse order of the RLIS of the input sequence A.

2.2 Correctness and Efficiency

Lemma 2: $Rank_i$ computed by Lemma 1 is the length of an RLIS ending with a_i.

Proof: We prove it by induction on i. If $i = 1$ or there does not exist any element that dominates a_i, the subsequence only contains a_i, and this lemma is true. Assume that the lemma holds when $i = 2$ to k. Now we consider the condition $i = k + 1$. Let a_j have the maximum $Rank$ value and dominate a_{k+1}. If $Rank_{k+1} > Rank_j + 1$, the $Rank$ value of a_{k+1}'s preceding element in the subsequence will be equal to $Rank_{k+1} - 1 > Rank_j$, and it also dominates a_{k+1}, a contradiction. If $Rank_{k+1} < Rank_j + 1$, it is easy to see that $a_j \not\prec_R a_{k+1}$ because $Rank_j \geq Rank_{k+1}$, also a contradiction. Then we have $Rank_{k+1} = Rank_j + 1$. By the induction hypothesis, the lemma holds. □

By controlling the index in the set S and querying the value range in the dynamic RMQ data structure, the **for** loop in Algorithm RLIS maintains Lemma 2 to get the correct $Rank$ of all elements. The following theorem analyzes the time and space complexities of Algorithm RLIS.

Theorem 3: Algorithm RLIS takes $O(n \log(U_I - L_I))$ time and $O(n)$ space where n is the size of input and $(U_I - L_I) < n$.

Proof: In each round of the **for** loop, each operation of *Delete, Insert* and *Query* is performed once which runs in $O(log|S|)$ time. The cardinality of S is not bigger than $U_I - L_I$ because S only keeps the elements whose indices are in the range $[i - U_I, i - L_I]$. Thus, the total time complexity is $O(n \log(U_I - L_I))$. The space complexity used by the dynamic RMQ is $O(|S|)$. The total space complexity is $O(|S| + n) = O(n)$. □

3 The SLIS Problem

The input of this problem contains an array $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a nonnegative ratio m. For any two elements a_i and a_j in A, we define that $a_i \prec_S a_j$ if and only if $\frac{a_j - a_i}{j - i} \geq m$ and $j > i$ in this section. A main observation is that the property of transitivity holds for the relation $\prec_S$. That is, if $a \prec_S b$ and $b \prec_S c$, we have $a \prec_S c$. Note that this property does not hold for the relation $\prec_R$ in Section 2 because of two constraints L_I and L_V. The objective of the SLIS problem is to find a maximum-length increasing subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$ satisfying $\frac{a_{i_{k+1}} - a_{i_k}}{i_{k+1} - i_k} \geq m$ for all $1 \leq k < l$.

3.1 The Algorithm

In this problem, the $Rank$ of element a_i is defined as the length of an SLIS ending at a_i and computed by the following lemma.

Lemma 4: If $a_j \not\prec_S a_i$ for all j, let $Rank_i = 1$, and $\max_{1 \leq j < i}\{Rank_j | a_j \prec_S a_i\} + 1$ otherwise.

We partition the whole sequence into some groups $R_1, R_2, \ldots, R_k$ according to the value of $Rank_i$. In other words, $a_i \in R_k$ if and only if $Rank_i = k$. The pseudo code for the SLIS problem is given in Algorithm SLIS. The strategy of

Algorithm SLIS
Input: $A = \langle a_1, a_2, \ldots, a_n \rangle$, m
Output: an SLIS
for $i \leftarrow 1$ **to** n **do**
  Find $\max\{j | R_j \prec_S a_i\}$; // use binary search
  if $j = NULL$ **then** $R_1 \leftarrow a_i$;
  else $R_{j+1} \leftarrow a_i$;
    Make backtracking link ($i \leftarrow R_j$'s index);
end for
Find $\max\{j | R_j \neq NULL\}$;
Output j and the sequence in reverse order by
    backtracking from j;

Figure 2: An algorithm for the SLIS problem.

this algorithm is also using dynamic programming to parse each element a_i from $i = 1$ to n, and $Rank_i$ is computed at the ith iteration. At the ith iteration, let R_k be any partition of the element set $\{a_1, a_2, \ldots, a_{i-1}\}$. An element $c \in R_k$ is called a critical point in R_k if an element in R_k dominates a_i implies that c also dominates a_i.

When we want to find out if there exists any dominating element in a partition R_k, it is sufficient to check only the *critical point* in R_k. We shall prove that for any partition R_k, the last inserted element is the *critical point*. Thus, only the element with maximum index in each partition R_k should be recorded in our algorithm.

At the ith iteration of the algorithm, suppose that the *critical point* which dominates a_i with maximum $Rank$ is in the partition of R_k. We then have $Rank_i = k + 1$, thus a_i is inserted into R_{k+1}. Similarly, if there does not exist any dominating element of a_i, then $a_i \in R_1$ and $Rank_i = 1$.

We shall also show that for any element $a \in R_k$ at the ith iteration, if $a \prec_S a_i$, then we have $c' \prec_S a_i$ for all *critical points* c' in R_j with $j < k$. By this property, the algorithm uses binary search to find the dominating *critical point* with maximum $Rank$. The remaining steps of the algorithm such as making backtracking links and outputting length and sequence are the same with Section 2.

3.2 Correctness and Efficiency

Lemma 5: For any partition R_k, the last inserted element (with maximum index) is the critical point.

Proof: At the ith iteration, let c be the last inserted element in R_k and let $a \in R_k$ be an element that dominates element a_i. We show that c is the *critical point* if c also dominates a_i. Suppose each element is represented on the plane which has been described above. It forms a triangle with three vertices a, c, and a_i. Note that the index order will be a, c and a_i since c is the last element in R_k and a_i is the current element under processing.

Let the slopes of $\overline{aa_i}$, $\overline{ac}$ and $\overline{ca_i}$ be represented by $m_{\overline{aa_i}}$, $m_{\overline{ac}}$ and $m_{\overline{ca_i}}$, respectively. Since $a \prec_S a_i$, we have $m_{\overline{aa_i}} \geq m$. Assume that $c \not\prec_S a_i$. Then we have $m_{\overline{ca_i}} < m$. Because $m_{\overline{aa_i}}$ is the convex combination of $m_{\overline{ac}}$ and $m_{\overline{ca_i}}$, it follows that $m_{\overline{ac}}$ is greater than m. In other words, $a \prec_S c$ and c should not have been inserted into R_k. This is a contradiction. Therefore, we conclude that $c \prec_S a_i$. □

For any element $a \in R_k$, there must exist an element $a' \in R_{k-1}$ such that $a' \prec_S a$. At the ith iteration, $a \prec_S a_i$ implies that $a' \prec_S a_i$ by the property of transitivity. By Lemma 5, we know that the *critical point* c in R_{k-1} dominates a_i, i.e., $c \prec_S a_i$. Thus, the following lemma holds.

Lemma 6: For any element $a \in R_k$ at the ith iteration, if $a \prec_S a_i$, then the critical point of R_{k-1} also dominates a_i.

By the property of transitivity and Lemma 6, we have Corollary 7.

Corollary 7: For any element $a \in R_k$ at the ith iteration, if $a \prec_S a_i$, then for all *critical points* in R_j with $j < i$ dominates a_i.

By Lemma 5, it is obvious to know that Algorithm SLIS implements the Lemma 6. By the Corollary 7, performing a binary search takes $O(\log |R|)$ time for each element, where $|R|$ is the number of $Rank$ partitions and does not greater than the output size r. It takes $O(n)$ space to record all the *critical points* and backtracking links. Therefore, we have the following theorem.

Theorem 8: Algorithm SLIS delivers an SLIS of an input sequence in $O(n \log r)$ time and $O(n)$ space, where n is the input size and r is the output size.

4 Concluding Remarks

We investigate two constrained versions of the LIS problem. By using the dynamic RMQ data structure, we present an algorithm that solves the

RLIS problem in $O(n \log(U_I - L_I))$ time and $O(n)$ space, where U_I and L_I are the upper and lower bounds of differences on indices, respectively. Another algorithm presented in this paper utilizes the concept of "*critical point*" to solve the SLIS problem in $O(n \log r)$ time and $O(n)$ space, where r is the output length.

Acknowledgments. We thank our advisor Prof. Kun-Mao Chao for valuable comments. I-Hsuan Yang and Yi-Ching Chen were supported in part by NSC grants 95-2221-E-002078, 95-2221-E-002-126-MY3, and 96-2221-E-002-034 from the National Science Council, Taiwan.

References

[1] G.M. Adelson-Velskil and E.M. Landis. *Soviet Math. (Dokl.) 3*, 1259–1263, 1962.

[2] M.H. Albert, A. Golynski, A.M. Hamel, A. López-Ortiz, S.S. Rao, and M.A. Safari. Longest increasing subsequences in sliding windows. *Theor. Comput. Sci.*, 321:405–414, 2004.

[3] D. Aldous and P. Diaconis. Longest increasing subsequences: From patience sorting to the Baik-Deift-Johansson theorem. *Bulletin (New Series) of the American Mathematical Society*, 36(4):413–432, 1999.

[4] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Inf. Process. Lett.*, 76:7–11, 2000.

[5] G.S. Brodal, K. Kaligosi, I. Katriel, and M. Kutz. Faster algorithms for computing longest common increasing subsequences. *BRICS-RS-05-37*, December 2005.

[6] W.T. Chan, Y. Zang, S.P.Y Fung, D. Ye, and H. Zhu. Efficient algorithms for finding a longest common increasing subsequence. In *Proc. 16th Annual International Symposium on Algorithms and Computation (ISAAC 2005)*, 665–674, 2005.

[7] E. Chen, H. Yuan, and L. Yang. Longest increasing subsequences in windows based on canonical antichain partition. In *Proc. 16th Annual International Symposium on Algorithms and Computation (ISAAC 2005)*, 1153–1162, 2005.

[8] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. white, and S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Res.*, 27(11):2369–2376, 1999.

[9] A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res.*, 30(11):2478–2483, 2002.

[10] L. Florea, G. Hartzell, Z. Zhang, G.M. Rubin, and W. Miller. A computer program for aligning a cDNA sequence with a genomic DNA sequence. *Genome Res.*, 8(9), 967–974, 1998.

[11] M.L. Fredman. On computing the length of longest increasing subsequences. *Discrete Math.*, 11:29–35, 1975.

[12] I. Katriel and M. Kutz. A faster algorithm for computing a longest common increasing subsequence. *Research Report MPI-I-2005-1-007, Max-Planck-Institut für Informatik*, Stuhlsatzenhausweg 85, 66123 Saarbrücken, German, March 2005.

[13] D.E. Knuth. *The Art of Computer Programming. Vol 3, Sorting and searching.* Addison-Wesley, 1998.

[14] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology.*, 5:R12, 2004.

[15] U. Manber. *Introduction to algorithms – A creative approach.* Addison-Wesly, 1989.

[16] C. Schensted. Longest increasing and decreasing subsequences. *Canad. J. Math.*, 13:179–191, 1961.

[17] T. Shibuya and I. Kurochkin. Match chaining algorithms for cDNA mapping. In *Proc. 3rd International Workshop on Algorithms in Bioinformatics (WABI 2003)*, 2812:462–475, 2003.

[18] T.F. Smith and M.S. Waterman. Comparison of biosequences. *Adv. in Appl. Math.*, 2:482–489, 1981.

[19] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[20] I.H. Yang, C.P. Huang, and K.M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.*, 93:249–253, 2005.

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