Efficient Algorithms for Locating the Length-Constrained Heaviest Segments, with Applications to Biomolecular Sequence Analysis

Yaw-Ling Lin [*] Tao Jiang [†] Kun-Mao Chao [‡]

Abstract.

We study two fundamental problems concerning the search for interesting regions in sequences: (i) given a sequence of real numbers of length n and an upper bound U, find a consecutive subsequence of length at most U with the maximum sum and (ii) given a sequence of real numbers of length n and a lower bound L, find a consecutive subsequence of length at least L with the maximum average. We present an $O(n)$-time algorithm for the first problem and an $O(n \log L)$-time algorithm for the second. The algorithms have potential applications in several areas of biomolecular sequence analysis including locating GC-rich regions in a genomic DNA sequence, post-processing sequence alignments, annotating multiple sequence alignments, and computing length-constrained ungapped local alignment. Our preliminary tests on both simulated and real data demonstrate that the algorithms are very efficient and able to locate useful (such as GC-rich) regions.

Keywords: Algorithm, efficiency, maximum consecutive subsequence, length constraint, biomolecular sequence analysis, ungapped local alignment.

1 Introduction

With the rapid expansion in genomic data, the age of large-scale biomolecular sequence analysis has arrived. An important line of research in sequence analysis is to locate biologically meaningful segments, *e.g. conserved* segments and *GC-rich* regions, in DNA sequences. Conserved segments of a DNA sequence are

[*]Department of Comput. Sci. and Info. Management, Providence University, 200 Chung Chi Road, Shalu, Taichung County, Taiwan 433. e-mail: `yllin@pu.edu.tw`

[†]Department of Computer Science and Engineering, University of California Riverside, Riverside, CA 92521-0144, USA. e-mail: `jiang@cs.ucr.edu`

[‡]Department of Life Science, National Yang-Ming University, Taipei, Taiwan 112. e-mail: `kmchao@ym.edu.tw`

slow changing sequences that form strong candidates for functional elements both in protein coding and regulatory regions of genes [7, 12, 20]. Regions of a DNA sequence that are rich in nucleotides C and G are usually significant in gene recognition. In order to locate these interesting segments, many combinatorial and probabilistic techniques have been proposed. Perhaps the most popular ones are window-based. That is, a window of a fixed length is moved down the sequence/alignment and the content statistics are calculated at each position that the window is moved to [15, 17]. Since an optimal region could span several windows, the window-based approach might fail in finding the exact locations of some interesting regions.

In this paper, we study two fundamental problems concerning the search for the "heaviest" segment of a numerical sequence that naturally arises in the above applications. Our main results, as described below, are efficient algorithms for locating the length-constrained heaviest segments in a given sequence or alignment. The algorithms have potential applications in locating GC-rich regions in a genomic DNA sequence, post-processing sequence alignments, annotating multiple sequence alignments, and computing length-constrained ungapped local alignment.

Let $A = \langle a_1, a_2, \ldots, a_n \rangle$ be a sequence of real numbers and $U \leq n$ a positive integer, the objective of our first problem is to find a consecutive subsequence [1] of A of length *at most* U such that the sum of the numbers in the subsequence is maximized. By using a technique of partitioning each suffix of A into minimal *left-negative* (consecutive) subsequences, we propose an $O(n)$-time algorithm for finding the length-constrained maximum sum consecutive subsequence of A. The algorithm can be used to find GC-rich regions and efficiently construct ungapped local alignments with length constraints in $O(mn)$ time, where m, n are the lengths of the two input sequences being aligned, as explained in the next section. We note in passing that a linear-time algorithm for finding the maximum sum consecutive subsequence with length at least L can be easily ob-

[1] Note that a consecutive subsequence is often referred to as a *substring* in some areas of computer science.

tained [13] by extending the dynamically algorithm for the standard maximum sum consecutive subsequence problem in [6].

An alternative measure of the weight of the target segment that we consider is as follows. Given a sequence of real numbers, $A = \langle a_1, a_2, \ldots, a_n \rangle$, and a positive integer $L \leq n$, the goal is to find a consecutive subsequence of A of length *at least* L such that the average of the numbers in the subsequence is maximized. We propose a novel technique to partition each suffix of A into *right-skew* segments of strictly decreasing averages, and based on this partition, we devise an $O(n \log L)$-time algorithm for locating the maximum average consecutive subsequence of length at least L. [2] The algorithm is expected to have applications in finding GC-rich regions in a genomic DNA sequence, postprocessing sequence alignments, and annotating multiple sequence alignments.

Observe that both problems studied in this paper have straightforward dynamic programming algorithms with running time proportional to the product of the input sequence length n and the length constraint (*i.e.* U or L). Such algorithms are perhaps fast enough for sequences of small lengths, but can be too slow for instances in some biomolecular sequence analysis applications, such as finding GC-rich regions and post-processing sequence alignments, where long genomic sequences are involved. Our above algorithms would be able to handle genomic sequences of length up to millions of bases with satisfactory speeds, as demonstrated in the preliminary experiments.

The rest of the paper is organized as follows. Section 2 discusses the biological applications in more depth. We present the algorithm for the length-constrained maximum sum consecutive subsequence problem in Section 3 and the algorithm for the length-constrained maximum average consecutive subsequence problem in Section 4. Some preliminary experiments on the speed and performance of the algorithms are given in Section 5. Section 6 concludes the paper with a few remarks.

[2] Note that, when there is no length constraint, finding the maximum average consecutive subsequence is equivalent to finding the maximum element.

2 Applications to Biomolecular Sequence Analysis

Since the heaviest segment problems that we study here are mostly motivated by their applications in several areas of biomolecular sequence analysis, we first describe the applications in detail to put the problems and our results in proper perspective.

2.1 Locating GC-Rich Regions

In all organisms, the GC base composition of DNA varies between 25–75%, with the greatest variation in bacteria. Mammalian genomes typically have a GC content of 45-50%. Nekrutenko and Li [15] showed that the extent of the compositional heterogeneity in a genomic sequence strongly correlates with its GC content. Genes are found predominantly in the GC-richest isochore classes. Hence, finding GC-rich regions is an important problem in gene recognition and comparative genomics. As being mentioned in Section 1, previously devised window-based strategies [15, 17] might fail in finding the exact locations of some interesting regions.

Huang [13] used the expression $x - p \cdot l$ to measure the GC richness of a region, where x is the C+G count of the region, p is a positive constant ratio, and l is the length of the region. In other words, each of nucleotides C and G is given a reward of $1 - p$, and each of nucleotides A and T is penalized by p. Similar expression was used by Sellers [18] for recognizing patterns by mismatch density. A length cutoff L is given to avoid reporting extremely short optimal regions. Huang extended the well-known recurrence relation used by Bentley [6] for solving the maximum sum consecutive subsequence problem, and derived a linear-time algorithm for computing the optimal segments with lengths at least L.

Here we explain briefly Huang's idea for computing the maximum sum consecutive subsequence of length at least L. Let $A = \langle a_1, a_2, \ldots, a_n \rangle$ be a DNA sequence of length n. Use $w(X)$ to denote the *score* of nucleotide X, *i.e.*

$w(G)$=$w(C)$=$1-p$, and $w(A)$=$w(T)$=$-p$. Define $S(i)$ to be the maximum score of regions ending at position i of A, which include the empty region. The scores $S(i)$ can be computed by the following recurrence:

$$S(i) = \begin{cases} \max\{S(i-1) + w(a_i), 0\} & \text{if } i > 0 \\ 0 & \text{if } i = 0 \end{cases}$$

Now let us shift along the sequence with a window of size L. For each fixed window, we can compute its score, and then the maximum score of regions ending at the front of the window with the help of the vector S. This results in a linear-time method for computing the maximum sum consecutive subsequence of length at least L.

As noted by Huang, the lengths of the regions reported by the algorithm are usually much greater than the cutoff L. An immediate implication is that they might contain some very poor and irrelevant regions. It is therefore natural to consider bounding the target regions with additional upper bound. Our algorithm for the length-constrained maximum sum consecutive subsequence problem can be combined with Huang's algorithm to yield a linear-time algorithm for computing the maximum sum consecutive subsequence of length between lower bound L and upper bound U. The details will be given in Section 3.

Huang has also proposed an interesting alternative measure for finding GC-rich regions [13]. Namely, given a DNA sequence, one would now attempt to find segments of length at least L with the highest C+G ratio. He noted that such an optimal segment is of length at most $2L-1$ (see Lemma 7 for a proof). This observation yields an $O(nL)$-time algorithm for computing a segment of length at least L with the highest C+G ratio, where n is the length of the input sequence [13]. Our algorithm for the length-constrained maximum average consecutive subsequence problem would improve on this result and locate the regions of length at least L with the highest C+G ratio in $O(n \log L)$ time.

CpG islands are defined as regions of DNA of at least 200bp (*i.e.* base pairs) in length with G+C content above 50%, and a ratio of observed vs. expected CpGs (CG di-nucleotides) at least 0.6 [9, 14]. Most of the CpG islands are between 200 and 1400bp with a majority of them being 200–400bp. Based on

large genomic datasets, Hannenhalli and Levy [10] have recently showed that CpG islands play an important role in the prediction of promoter. We expect that some of the techniques used in our $O(n \log L)$-time algorithm, such as the concept of right-skew segments and the decreasingly right-skew partitions developed in this paper, would be useful in efficiently locating all CpG islands in a genomic sequence. For example, the method can be easily extended to computing the region of the most frequent GC di-nucleotides occurrences.

2.2 Post-Processing Sequence Alignments

A new popular approach to gene prediction in the human genome is based on comparative analysis of human and mouse DNA [4, 5, 16]. The rational behind this approach is that similarity between corresponding human and mouse exons is 85% on average, while similarity between introns is 35% on average [3]. Though the ingenious Smith-Waterman [19, 21] local alignment approach has been very successful in revealing highly conserved regions by discarding poorly conserved surrounding regions, a potential drawback of the method is that it may lead to the inclusion of arbitrarily poor internal regions (called the *mosaic effect*) [3, 23, 24].

In an attempt to fix the mosaic effect problem, Zhang *et al.* [23] developed some efficient heuristic algorithms for delivering alignments that contain no low scoring regions. This, however, does not take into account the length of the alignment. Alexandrov and Solovyev [1] proposed to normalize the alignment score by its length [3] and demonstrated that this new approach leads to better protein classification. Following this line of investigation, Arslan *et al.* [3] studied a variant of normalized score, which is simply called *length-adjusted normalized score* for the ease of presentation, [4] and gave an $O(mn \log n)$-time algorithm for reporting regions with the maximum length-adjusted normalized degree of similarity, where m, n are the lengths of the two sequences being

[3] For an alignment of length l with score s, its normalized score is defined as $\frac{s}{l}$ [1]. Of course, in order to avoid extremely short alignments, we need impose a constraint (lower bound) on the length of the target alignment.

[4] For an alignment of length l with score s, its length-adjusted normalized score is defined as $\frac{s}{l+L}$, where $L > 0$ is a predetermined constant used to avoid extremely short alignments.

compared.

An alternative approach is to first run Smith-Waterman type of alignment algorithms and then post-process the computed alignments. Zhang *et al.* [24] developed an elegant linear-time algorithm that decomposes a long alignment into sub-alignments to avoid the mosaic effect. Our method for computing the length-constrained maximum average consecutive subsequences can be used to locate within an alignment the region that is sufficiently long and has the maximum degree of normalized similarity. It is expected that this would turn out to be a useful technique for alignment decomposition.

2.3 Annotating Multiple Sequence Alignments

As mentioned above, conserved regions in biomolecular sequences are strong candidates for functional elements. The most popular methods to compute conserved regions all start with a given multiple sequence alignment. Stojanovic *et al.* [20] gave several methods for finding highly conserved regions within previously computed multiple alignments. Three of the methods are based on assigning a numerical score to each column of a multiple alignment and then looking for runs of columns with high cumulative scores. Since the assigned scores may be all positive (*e.g.* in the information content case), each examined column could increase the cumulative score. It follows that the entire alignment could be reported erroneously as a conserved region. Therefore, it is imperative that each column score is adjusted by subtracting a positive *anchor* value [20]. Determining such an anchor value appropriately for each dataset could make the use of a program based on the above approach very complicated.

A solution to the above problem is to look for runs of sufficiently many columns in the multiple alignment with the maximum *average* (or *normalized*) score instead. This can be efficiently computed by our $O(n \log L)$-time algorithm for the length-constrained maximum average consecutive subsequence problem.

2.4 Computing Ungapped Local Alignments with Length Constraints

Consider a two-dimensional matrix where each entry (i, j) is filled with the similarity score between the i-th element of one sequence and the j-th element of another sequence. Our algorithms can be used to compute the length-constrained segment of each diagonal in the matrix with the largest sum (or average) of scores. As a consequence, we have an $O(mn)$-time algorithm for constructing an ungapped local alignment of length at most U with the largest total score, where m, n are the length of the input sequences and U is the length upper bound. We also have an $O(mn \log L)$-time algorithm for constructing an ungapped local alignment of length at least L with the largest normalized (*i.e.* average) score, where the normalized score of an alignment is defined as its score divided by its length. Ungapped local alignment is an important and well studied variant of local sequence alignment and has applications in motif identification (see, *e.g.*, [1]).

It should be noticed that the best known algorithms for the general (*i.e.* gapped) length-constrained local alignment problems with the standard (sum) score and normalized score run in time $O(mnU)$ and $O(mnL)$, respectively. Arslan *et al.* have recently reported an $O(mn)$ time *approximation* algorithm for computing a length-constrained local alignment with the largest score [2] and an $O(mn \log n)$-time algorithm for computing a local alignment with the optimal length-adjusted normalized score [3], which is closely related to the problem of constructing a length-constrained local alignment with the largest normalized score.

3 Maximum Sum Consecutive Subsequence with Length Constraints

Given a sequence of real numbers, $A = \langle a_1, a_2, \ldots, a_n \rangle$, and a positive integer $U \leq n$, the goal is to find a consecutive subsequence of A of length at most U such that the sum of the numbers in the subsequence is maximized.

It is straightforward to design a dynamic programming algorithm for the problem with running time $O(nU)$. We also note in passing that since there is an $O(n \log^2 n)$-time algorithm for finding the maximum sum path on a tree with length at most U [22], the above problem can also be solved in $O(n \log^2 n)$ time. Here, we present an algorithm running in $O(n)$.

Let $A_1, A_2, \ldots, A_k$ be disjoint (consecutive) subsequences of A forming a *partition* of A, *i.e.* $A = A_1 A_2 \cdots A_k$. A_i is called the ith segment of the partition. Denote $w(A) = \sum_{a_i \in A} a_i$ as the sum of the sequence. The following definition is a key of our linear-time construction.

Definition 1 *A real sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ is* left-negative *if and only if the sum of each proper prefix $\langle a_1, a_2, \ldots, a_i \rangle$ is negative or zero for all $1 \leq i \leq n-1$; that is, $w(\langle a_1, a_2, \ldots, a_i \rangle) \leq 0$ for all $1 \leq i \leq n-1$. A partition of the sequence $A = A_1 A_2 \cdots A_k$ is* minimal left-negative *if each $A_i, 1 \leq i \leq k$, is left-negative, and, for each $1 \leq i \leq k-1$, the sum of A_i is positive, i.e. $w(A_i) > 0$.*

For example, the sequence $\langle -4, 1, -2, 3 \rangle$ is left-negative while the sequence $\langle$5, -3, 4, -1, 2, $-6\rangle$ is not. On the other hand, the partition $\langle 5 \rangle \langle -3, 4 \rangle \langle -1, 2 \rangle \langle -6 \rangle$ of the latter sequence is minimal left-negative. Note that any singleton sequence is trivially left-negative by definition. Furthermore, it can be shown that any sequence can be uniquely partitioned into minimal left-negative segments.

Lemma 1 *Every sequence of real numbers can be uniquely partitioned into minimal left-negative segments.*

Proof. Let $A = \langle a_1 a_2 \ldots a_n \rangle$. The statement obviously holds if $n = 1$. By induction, assume that a sequence B, where $|B| = n$, is uniquely partitioned into minimal left-negative segments as $B = A_1 A_2 \cdots A_k$. Now consider the sequence $A = \langle a, B \rangle$, where a is a real number.

The lemma is true if $a > 0$ since $\langle a \rangle A_1 A_2 \cdots A_k$ would form a minimal left-negative partition by induction. Otherwise, let i be the smallest index such that $(a + \sum_{j=1}^{i} \sum_{x \in A_j} x) > 0$. It is easily verified that the sequence $\langle a, A_1 \cdots A_i \rangle$

MLN-POINT(A)
Input: A real sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$.
Output: n left-negative pointers of A, encoded by array $p[\cdot]$.
1 **for** $i \leftarrow n$ **downto** 1 **do**
2 $\quad p[i] \leftarrow i; w[i] \leftarrow a_i;$ $\quad \triangleright$ Each $\langle a_i \rangle$ alone is left-negative.
3 $\quad$ **while** $(p[i] < n)$ and $w[i] \leq 0$ **do**
4 $\quad\quad w[i] \leftarrow w[i] + w[p[i] + 1]$
5 $\quad\quad p[i] \leftarrow p[p[i] + 1]$

Figure 1: Set up the left-negative pointers.

REPORT-MLN-PART(i)
Input: i denoting the suffix sequence $\langle a_i, a_{i+1}, \ldots, a_n \rangle$.
Output: the minimal left-negative partition of the suffix.
1 **while** $i \leq n$ **do** $\quad \triangleright$ Reports (i, j) as a left-negative segment $\langle a_i, \ldots, a_j \rangle$.
2 $\quad$ OUTPUT $(i, p[i]); i \leftarrow p[i] + 1$

Figure 2: Computing the minimal left-negative partition of a suffix sequence.

is left-negative. Thus, $\langle a, A_1 \cdots A_i \rangle A_{i+1} \cdots A_k$ forms a uniquely minimal left-negative partition of A. □

For any sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, each suffix sequence of A, $\langle a_i, \ldots, a_n \rangle$, defines a minimal left-negative partition, denoted as $A_1^{(i)} A_2^{(i)} \cdots A_k^{(i)}$, for some $k \geq 1$. Suppose that $A_1^{(i)} = \langle a_i, \ldots, a_{p[i]} \rangle$. Then, $p[i]$ is called the *left-negative pointer* of index i. Note that the left-negative pointers of A implicitly encode the minimal left-negative partition of each suffix $\langle a_i, \ldots, a_n \rangle$ of A. An efficient algorithm for computing the left-negative pointers as well as the minimal left-negative partition of each suffix of A is illustrated in Figures 1 and 2.

Lemma 2 *The algorithm* MLN-POINT *given in Figure 1 finds all left-negative pointers for a length n sequence in $O(n)$ time.*

Proof. Consider the algorithm MLN-POINT(A) shown in Figure 1. The variable i is the current working pointer scanning elements of A from right to left. The pair $(i, p[i])$ represents a consecutive subsequence (or segment) of A, $\langle a_i, \ldots, a_{p[i]} \rangle$, while variable $w[i]$ represents the sum of the segment $(i, p[i])$.

An example run of MLN-POINT(A) on a 15-element sequence is illustrated in

Figure 3. Note that the pair $(i, p[i])$ always represents a left-negative segment of A throughout the entire algorithm. This invariant is true because a left-negative segment with negative sum can be grouped with another adjacent left-negative segment into a longer left-negative segment. The grouping and checking of the involved condition are done by Steps 3 and 5 of the algorithm.

The correctness of the algorithm follows from the fact that, after the execution of Step 1 to Step 5, each pair $(i, p[i])$ represents a left-negative segment with a positive sum, except the last pair. Furthermore, by Lemma 1, the left-negative pointers found by the algorithm encode the unique minimal left-negative partition of each suffix of A.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
a_i	9	-3	1	7	-15	2	3	-4	2	-7	6	-2	8	4	-9
$p[i]$	1	4	3	4	15	6	7	13	9	13	11	13	13	14	15
$w[i]$	9	5	1	7	-12	2	3	3	2	5	6	6	8	4	-9

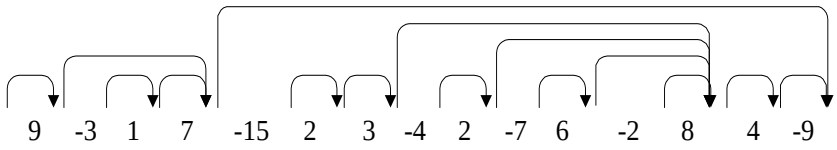

Figure 3: The left-negative pointers of a 15-element sequence.

The $O(n)$-time complexity of MLN-POINT(A) can be shown by a simple amortized analysis [8]. The total number of operations of the algorithm is clearly bounded by $O(n)$ except for the while-loop body of Step 3 to Step 5. In the following, we show that the amortized cost of the while-loop is a constant. Therefore, the overall time required by the loop is $O(n)$.

We define the *potential function* of A after the jth iteration of the for-loop (*i.e.* Steps 1 to 5) to be $\Phi(n - j + 1)$, *i.e.* the numbers of left-negative segments within the minimal left-negative partition of the suffix sequence $A_{n-j+1} = \langle a_{n-j+1}, \ldots, a_n \rangle$ considered at the jth iteration of the for-loop. Note that the loop variable i is just $n-j+1$ in the jth iteration. Let us compute the amortized cost of the operations done by Step 3 to Step 5 in this jth iteration. Suppose

that the pointer $p[\cdot]$ is advanced c_j times in this period. Then the actual cost of the operations is $c_j + 1$. Observing that $\Phi(n-j+1) = \Phi(n-j+2) - c_j + 1$, the change of the potential of A during the jth iteration is

$$\Phi(n-j+1) - \Phi(n-j+2) = \Phi(n-j+2) - c_j + 1 - \Phi(n-j+2) = 1 - c_j.$$

The amortized cost is therefore calculated as

$$\hat{c}_j = c_j + 1 + \Phi(n-j+1) - \Phi(n-j+2) = 2.$$

In other words, we deposit a credit (as a unit of the potential of A) whenever a correct value of the left-negative pointer $p[\cdot]$ is found. Later on, when the algorithm needs to advance the $p[\cdot]$ pointer in the while-loop, the cost can be charged to the pre-deposited credits. Since exactly n credits would be deposited in the entire process, the while-loop spends at most overall $O(n)$ time. □

We are ready to show that the length-constrained maximum sum consecutive subsequence problem can be solved in linear time.

Theorem 1 *Given a length n real sequence, finding the consecutive subsequence of length at most U with the maximum sum can be done in $O(n)$ time.*

Proof. We propose an $O(n)$ time algorithm, $\mathrm{MSLC}(A, U)$, as shown in Figure 4. In the algorithm, the variable i is the current working pointer scanning elements of A from left to right. The pair (i, j) represents a consecutive subsequence of A, $\langle a_i, \ldots, a_j \rangle$, currently being considered as a candidate maximum sum consecutive subsequence satisfying the length constraint. The algorithm essentially looks at every positive a_i and identifies its corresponding *good partner*, a_j, such that (i, j) constitutes a candidate solution.

Note that the sum of any proper prefix of a left-negative segment is negative by definition. The correctness of the algorithm then follows from the fact that a left-negative segment is *atomic* in the sense that when it is combined with preceding left-negative segments, it is always combined *as a whole*; for otherwise the addition of any proper prefix of the segment would only decrease the sum

MSLC(A, U)
Input: A real sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, and an upper bound U.
Output: The maximum consecutive subsequence of A with length at most U.
1 $i \leftarrow 1$
2 **while** $a_i \leq 0$ and $i \leq n$ **do** $i \leftarrow i + 1$
3 **if** $i = n$ **then** ▷ Elements $a_1, a_2, \ldots, a_{n-1}$ are all negative.
4 Find the maximum element in A and return.
5 MLN-POINT(A) ▷ Compute left-negative pointers. See Fig 1.
6 $j \leftarrow i$; $ms \leftarrow 0$ ▷ Initialization.
7 **while** $i \leq n$ **do**
8 **while** $a_i \leq 0$ and $i \leq n$ **do** $i \leftarrow i + 1$
9 $j \leftarrow \max(i, j)$
10 **while** $j < n$ and $p[j + 1] < i + U$ and $w[j + 1] > 0$ **do** $j \leftarrow p[j + 1]$
11 **if** SUM$(i, j) > ms$ **then** $mi \leftarrow i; mj \leftarrow j; ms \leftarrow$ SUM(i, j) ▷ Update max.
12 $i \leftarrow i + 1$
13 **return** (mi, mj, ms)

SUM(i, j)
Output: The sum of the subsequence $\langle a_i, a_{i+1}, \ldots, a_j \rangle$, $\sum_{x=i}^{j} a_x$, is just $s_j - s_{i-1}$.
The prefix sums, $s_k = \sum_{i=1}^{k} a_i$, $s_0 = 0$, can be pre-computed in $O(n)$ time.

Figure 4: Finding the maximum sum consecutive subsequence with length constraint.

of the combined segment. This observation justifies the condition checking and grouping in Step 10 of the algorithm.

The time complexity of the algorithm is $O(n)$ because the good-partner pointer j only advances forward as the scanning pointer i advances. It follows that the total work spent on Step 10 is bounded by $O(n)$. It is not hard to verify that the remaining part of the algorithm spends at most $O(n)$ time. □

The algorithm MSLC can be easily combined with Huang's technique [13] to yield a linear-time algorithm that is able to handle a length upper bound and a length lower bound simultaneously.

Corollary 2 *Given a length n real sequence and positive integers $L \leq U$, finding the consecutive subsequence of length between L and U with the maximum sum can be done in $O(n)$ time.*

Proof. We modify algorithm MSLC to obtain an algorithm MSLU that finds

MSLU(A, L, U)
Input: A real sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, a lower bound L, and an upper bound U.
Output: The maximum consecutive subsequence of A with length at least L and at most U.
1 MLN-POINT(A) ▷ Compute left-negative pointers. See Fig 1.
2 $i \leftarrow mi \leftarrow 1$; $j \leftarrow mj \leftarrow L$; $ms \leftarrow$ SUM$(1, L)$ ▷ Initialization.
3 **while** $i \leq n - L + 1$ **do**
4 $j \leftarrow \max(i + L - 1, j)$
5 **while** $j < n$ and $p[j + 1] < i + U$ and $w[j + 1] > 0$ **do** $j \leftarrow p[j + 1]$
6 **if** SUM$(i, j) > ms$ **then** $mi \leftarrow i; mj \leftarrow j; ms \leftarrow$ SUM(i, j) ▷ Update max.
7 $i \leftarrow i + 1$
8 **return** (mi, mj, ms)

Figure 5: Finding the maximum sum consecutive subsequence with length between given lower and upper bounds.

a consecutive subsequence of length between L and U with the maximum sum. The algorithm is shown in Figure 5.

The idea of the algorithm is similiar to MSLC(A, U) in Figure 4. Again, the variable i is the current working pointer scanning elements of A from left to right. The algorithm essentially looks at every positive a_i and identifies its corresponding *good partner*, a_j, such that (i, j) constitutes a candidate solution. The correctness of the algorithm then follows from the fact that a left-negative segment is *atomic*, similiar to the proof of Theorem 1. The time complexity of the algorithm is $O(n)$ because the good-partner pointer j only advances forward as the scanning pointer i advances. It follows that the total work spent on Step 5 is bounded by $O(n)$. □

4 Maximum Average Consecutive Subsequence with Length Constraints

Given a sequence of real numbers, $A = \langle a_1, a_2, \ldots, a_n \rangle$, and a positive integer L, $1 \leq L \leq n$, our goal is now to find a consecutive subsequence of A with length at least L such that the average value of the numbers in the subsequence is maximized.

Recall that $w(A) = \sum_{i=1}^{n} a_i$ is the sum of elements of A. Furthermore, let

$d(A) = |A| = n$, be the length of the sequence A. The *average* of A is defined as $\mu(A) = w(A)/d(A)$. The definition below is the key to our construction.

Definition 2 *A sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ is* right-skew *if and only if the average of any prefix $\langle a_1, a_2, \ldots, a_i \rangle$ is always less than or equal to the average of the remaining suffix subsequence $\langle a_{i+1}, a_{i+2}, \ldots, a_n \rangle$. A partition $A = A_1 A_2 \cdots A_k$ is* decreasingly right-skew *if each segment A_i of the partition is right-skew and $\mu(A_i) > \mu(A_j)$ for any $i < j$.*

The following are some useful properties of right-skew segments and their averages.

Lemma 3 (Combination) *Let A, B be two sequences with $\mu(A) < \mu(B)$. Then $\mu(A) < \mu(AB) < \mu(B)$.*

Proof. Let $\lambda = d(A)/d(AB)$. We have $\mu(AB) = \lambda\mu(A) + (1 - \lambda)\mu(B)$. The result is true because $0 < \lambda < 1$. □

Lemma 4 *Let A, B be two right-skew sequences with $\mu(A) \leq \mu(B)$. Then the sequence AB is also right-skew.*

Proof. Consider a prefix P of AB. Clearly, $\mu(P) \leq \mu(B)$ if $P = A$. If P is a proper prefix of A, *i.e.* $A = PY$ for some nonempty sequence Y, then we have $\mu(P) \leq \mu(A) \leq \mu(Y)$ by the last lemma. Hence, $\mu(P) \leq \mu(YB)$ since $\mu(P) \leq \mu(B)$.

On the other hand, if P contains a proper prefix of B, *i.e.* $B = CD$ and $P = AC$ for some nonempty sequences C and D, then $\mu(C) \leq \mu(B) \leq \mu(D)$. Hence, $\mu(P) = \mu(AC) \leq \mu(D)$ since $\mu(A) \leq \mu(B) \leq \mu(D)$. □

Lemma 5 *Every real sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ has a unique decreasingly right-skew partition.*

Proof. We prove the lemma by induction. The lemma obviously holds if $n = 1$. Assume that Q is a sequence of length n and $Q = A_1 A_2 \cdots A_k$ is the unique decreasingly right-skew partition of Q. Now consider sequence $A = \langle Q, a \rangle$.

REPORT-DRS-PART$(i, p[\cdot])$
Input: i denoting the suffix sequence $\langle a_i, a_{i+1}, \ldots, a_n\rangle$; $p[\cdot]$: right-skew pointers of A.
Output: The decreasingly right-skew partition of the suffix.
1 **while** $i \leq n$ **do** $\triangleright$ Reports $\langle a_i, \ldots, a_j\rangle$ as a right-skew segment.
2 OUTPUT $(i, p[i]); i \leftarrow p[i] + 1$

Figure 6: Reporting the decreasingly right-skew partition of a suffix sequence.

The Lemma is proven if $\mu(a) < \mu(A_k)$. Otherwise, we find the largest i such that $\mu(A_i A_{i+1} \cdots A_k a) < \mu(A_{i-1})$; let $i = 1$ if such i can not be found. It suffices to show that $\langle A_i A_{i+1} \cdots A_k a\rangle$ is right-skew, and this can be done by observing that the single segment $\langle a\rangle$ is right-skew and by applying Lemma 4 repeatedly to the segments $A_i, A_{i+1}, \ldots, A_k, \langle a\rangle$ from right to left. That is, $\langle A_k a\rangle$ is right-skew because $\mu(A_k) \leq \mu(a)$, $\langle A_{k-1} A_k a\rangle$ is right-skew because $\mu(A_{k-1}) \leq \mu(A_k a)$, *etc.* Clearly, the partition is unique because other choices of i would not result in a decreasingly right-skew partition of A. □

For a sequence $A = \langle a_1, a_2, \ldots, a_n\rangle$, each suffix of A, $\langle a_i, \ldots, a_n\rangle$, defines a decreasingly right-skew partition, denoted as $A_1^{(i)} A_2^{(i)} \cdots A_k^{(i)}$, for some $k \geq 1$. Suppose that $A_1^{(i)} = \langle a_i, \ldots, a_{p[i]}\rangle$, where $p[i]$ is called the *right-skew pointer* of index i. Note that the right-skew pointers of A implicitly encode the decreasingly right-skew partitions for each suffix $\langle a_i, \ldots, a_n\rangle$ of A. Given the right-skew pointers, one can easily report the decreasingly right-skew partitions of a suffix as illustrated in Figure 6. Interestingly, we can compute all right-skew pointers in linear time.

Lemma 6 *The algorithm* DRS-POINT *given in Figure 7 computes all right-skew pointers for a length n sequence in $O(n)$ time.*

Proof. Consider the algorithm DRS-POINT shown in Figure 7. The working pointer i scans the elements of A from right to left. By Lemma 4, two increasingly right-skew segment can be grouped into one right-skew segment and hence, the pair $(i, p[i])$ always represents a segment of A that is right-skew throughout the entire algorithm. The correctness of the algorithm follows from the fact that

```
DRS-Point(A)
Input: A sequence A = ⟨a_1, a_2, ..., a_n⟩.
Output: n right-skew pointers of A, encoded by array p[·].
1 for i ← n downto 1 do
2     p[i] ← i; w[i] ← w(a_i); d[i] ← d(a_i);        ▷ Each ⟨a_i⟩ alone is right-skew.
3     while (p[i] < n) and (w[i]/d[i] ≤ w[p[i] + 1]/d[p[i] + 1]) do
4         w[i] ← w[i] + w[p[i] + 1]
5         d[i] ← d[i] + d[p[i] + 1]
6         p[i] ← p[p[i] + 1]
```

Figure 7: Setting up the right-skew pointers in $O(n)$ time.

the right-skew pointers found by the algorithm encode a partition of each suffix of A with strictly decreasing averages.

We can analyze the time complexity of the algorithm by an amortized argument similar to that in Lemma 2. We conclude that the amortized cost of each iteration of the for-loop is just a constant.

In short, we deposit a credit whenever a correct value of the right-skew pointer $p[\cdot]$ is found. Later on, when the algorithm needs to advance the $p[\cdot]$ pointer in the while-loop, the skipping cost can be charged to the pre-deposited credits. Since exactly n credits are deposited in the entire process, the while-loop spends at most overall $O(n)$ time. □

The next lemma is first presented in [13]. We include it here for completeness.

Lemma 7 *Given a real sequence A, let B denote the shortest consecutive subsequence of A with length at least L such that the average is maximized. Then the length of B is at most $2L - 1$.*

Proof. Suppose that $|B| \geq 2L$. Then B can be bisected into two segments, C and D, such that $|C| \geq L$ and $|D| \geq L$. Without loss of generality, assume that $\mu(C) \geq \mu(D)$. However, by Lemma 3, $\mu(C) \geq \mu(CD) = \mu(B)$, a contradiction! □

In searching for the maximum average consecutive subsequence, our construction will need to locate, for each element a_i, its corresponding partner, a_j,

such that the segment $\langle a_i, \ldots, a_j \rangle$ constitutes a candidate solution. Suppose that segment $A = \langle a_i \ldots a_j \rangle$ is being currently considered a candidate solution, where $j - i + 1 \geq L$, and $B = \langle a_{j+1}, \ldots, a_{p[j+1]} \rangle$ is the first right-skew segment to the right of A. We consider if the segment A should be extended to include some prefix (or the whole) of the segment B. The following lemma shows that A should be combined with the segment B *as a whole* if and only if $\mu(A) < \mu(B)$. In other words, the segment $B = \langle a_{j+1}, \ldots, a_{p[j+1]} \rangle$ is *atomic* (for A).

Lemma 8 (Atomic) *Let A, B, C be three real sequences with $\mu(A) < \mu(B) < \mu(C)$. Then $\mu(AB) < \mu(ABC)$.*

Proof. By Lemma 3, we must have $\mu(A) < \mu(AB) < \mu(B)$. Furthermore, since $\mu(B) < \mu(C)$, we have $\mu(AB) < \mu(C)$. It follows that $\mu(AB) < \mu(ABC) < \mu(C)$, again by Lemma 3. □

The next lemma allows us to perform binary search in the decreasingly right-skew partition of a suffix sequence when trying to find the "optimal" extension from a candidate solution segment.

Lemma 9 (Bitonic) *Let P be a (prefix) real sequence, and $A_1 A_2 \cdots A_m$ the decreasingly right-skew partition of a sequence A. Suppose that $\mu(PA_1 \cdots A_k) = \max\{\mu(PA_1 \cdots A_i) |\ 0 \leq i \leq m\}$. Then $\mu(PA_1 \cdots A_i) > \mu(A_{i+1})$ if and only if $i \geq k$.*

Proof. First, we show that $\mu(PA_1 \cdots A_i) > \mu(A_{i+1})$ implies $i \geq k$. Assume that $\mu(PA_1 \cdots A_i) > \mu(A_{i+1})$ for some i. Since $A_1 A_2 \cdots A_m$ is the decreasingly right-skew partition, we have $\mu(A_1) > \mu(A_2) > \cdots > \mu(A_m)$. Thus, $\mu(PA_1 \cdots A_i) > \mu(A_{i+1}) > \cdots > \mu(A_m)$. By Lemma 3, $\mu(PA_1 \cdots A_i) > \mu(PA_1 \cdots A_i A_{i+1} \cdots A_j)$ for any $j > i$. Therefore, $k \leq i$.

In the second part, we show that $i \geq k$ implies $\mu(PA_1 \cdots A_i) > \mu(A_{i+1})$. Observe that $\mu(PA_1 \cdots A_k) > \mu(A_{k+1}) > \cdots > \mu(A_m)$. By repeatedly applying Lemma 3, we have $\mu(PA_1 \cdots A_i) > \mu(A_{i+1})$ for any $i \geq k$. □

Now we are ready to state the main result of this section.

MAXAVGSEQ(A, L)
Input: A real sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a lower bound L.
Output: The maximum average consecutive subsequence of A of length at least L.
1 DRS-POINT(A) ▷ Compute the right-skew pointers, see Fig 7.
2 **for** $i \leftarrow 1$ **to** $n - L + 1$ **do**
3 $\quad j \leftarrow i + L - 1$
4 $\quad$ **if** $\mu(i, j) < \mu(j + 1, p[j + 1])$ **then** $j \leftarrow$ LOCATE(i, j) ▷ Move j.
5 $\quad g[i] \leftarrow j$
6 **return** The maximum $\mu(i, g[i])$ pair.

$\mu(i, j) =$ SUM$(i, j)/(j - i + 1)$ is the average of segment $\langle a_i, \ldots, a_j \rangle$.
LOCATE(i, j): Binary search in the list: $\langle \mu(i, j^{(0)}), \ldots, \mu(i, j^{(L)}) \rangle$, where $j^{(k)}$ is defined recursively: $j^{(0)} = j$, $j^{(k)} = \min\{p[j^{(k-1)} + 1], n\}$.

Figure 8: Finding the maximum average consecutive subsequence with length constraint.

Theorem 3 *Given a length n real sequence, finding the consecutive subsequence of length at least L with the maximum average can be done in $O(n \log L)$ time.*

Proof. We propose an $O(n \log L)$ time algorithm, MAXAVGSEQ(A, L), as shown in Figure 8. The pointer i scans elements of A from left to right. The pair (i, j) represents a segment of A, $\langle a_i, \ldots, a_j \rangle$, currently being considered as the candidate solution. For each element a_i, the algorithm finds its corresponding *good partner*, a_j, such that (i, j) constitutes a candidate solution.

Observe that right-skew segments are *atomic* in the sense that it is always better to add a whole right-skew segment in an extension process than to add a proper prefix, as shown in Lemma 8. Thus the possible good partners will be the right endpoints of the right-skew segments in the decreasingly right-skew partition of the suffix sequence $\langle a_{j+1}, \ldots, a_n \rangle$.

Let $j^{(k)}$ denote the right endpoint of the kth right-skew segment in the suffix sequence $\langle a_{j+1}, \ldots, a_n \rangle$. Note that $j^{(k)}$ can be defined recursively using the formula: $j^{(0)} = j$ and $j^{(k)} = \min\{p[j^{(k-1)} + 1], n\}$. By Lemma 7, there exists a maximum average segment whose length is at most $2L - 1$. Thus, the correctness of algorithm MAXAVGSEQ(A, L) follows if LOCATE(i, j) correctly

LOCATE(i, j)
Input: A prefix subsequence $\langle a_i, \ldots, a_j \rangle$ of A.
Output: The maximum average subsequence with with prefix $\langle a_i, \ldots, a_j \rangle$ and length at most $2L - 1$.
1 **for** $k \leftarrow \lceil \log L \rceil$ **downto** 0 **do**
2 **if** $j \geq n$ or $\mu(i, j) \geq \mu(j + 1, p[j + 1])$ **then return** j
3 **if** $p^{(k)}[j + 1]) < n$ and $\mu(i, p^{(k)}[j + 1]) < \mu(p^{(k)}[j + 1] + 1, p[p^{(k)}[j + 1] + 1])$
then $j \leftarrow p^{(k)}[j + 1]$
4 **if** $j < n$ and $\mu(i, j) < \mu(j + 1, p[j + 1])$ **then** $j \leftarrow p[j + 1]$ ▷ Final step.
5 **return** $j^* = j$

Figure 9: Finding the maximum average consecutive subsequence with prefix $\langle a_i, \ldots, a_j \rangle$ and length at most $2L - 1$.

computes the optimal j^* such that $\mu(i, j^*) = \max\{\mu(i, j^{(k)}) | 0 \leq k \leq L\}$, where $\mu(i, j)$ denotes the average of segment $\langle a_i, \ldots, a_j \rangle$. This is explained along with the following time complexity analysis of algorithm LOCATE.

To prove that the algorithm MAXAVGSEQ runs in $O(n \log L)$ time, it suffices to prove that algorithm LOCATE finds the (restricted) good partner j^* of i in $O(\log L)$ time. The key idea used in the algorithm is as follows. Although exploring the entire list $\langle j^{(1)}, \ldots, j^{(L)} \rangle$ to find the (restricted) good partner requires $O(L)$ time, Lemma 9 suggests that we may be able to find j^* by a binary search *without* having to generate the entire list $\langle j^{(1)}, \ldots, j^{(L)} \rangle$. To do so, we need maintain $\lceil \log L \rceil$ *pointer-jumping tables* $p^{(k)}[1..n], 1 \leq k \leq \lceil \log L \rceil$. Let $p^{(0)}[i] = p[i]$ and $p^{(k+1)}[i] = \min\{p^{(k)}[p^{(k)}[i] + 1], n\}$ be defined recursively. Intuitively, one pointer jump from j to $p^{(k)}[j + 1]$ is equivalent to 2^k original pointer jumps from j to $j^{(2^k)}$. Note that, these $p^{(k)}[1..n]$ tables can be precomputed with an overall time complexity of $O(n \log L)$.

Now we explain how the binary search performed in Steps 1 through 3 of LOCATE(i, j) for finding j^* works. Let $j^* = j^{(\ell)}$ for some $0 \leq \ell \leq L$. Then the problem of finding j^* can be thought of identifying an unknown binary string (the binary encoding of ℓ) of at most $\lceil \log L \rceil$ bits. In the algorithm, we identify the bits one by one from the $(\lceil \log \rceil - 1)$th (the most significant bit) down to the 0th (the lowest) bit, and for each kth bit, we check if $\mu(i, p^{(k)}[j + 1]) < \mu(p^{(k)}[j + 1] + 1, p[p^{(k)}[j + 1] + 1])$ using the pointer-jumping tables. The

bitonicity property in Lemma 9 can be used to determine whether the current index $j^{(\ell)}$ under consideration has surpassed the desired j^*. Note that, Step 4 of LOCATE(i, j) makes a final check on the result since the value of index j at the moment can be one step short of the optimal index value $j^* = j^{(\ell)}$ for some even number ℓ.

Therefore, LOCATE(i, j) finds a (restricted) good partner of i in $O(\log L)$ time. It follows that the algorithm MAXAVGSEQ(A, L) runs in at most $O(n \log L)$ time since Step 4 of the algorithm takes $O(\log L)$ time, and the precomputation of the jumping tables also takes at most $O(n \log L)$ time. □

5 Implementation and Preliminary Experiments

We have implemented a family of programs for locating the length-constrained heaviest segments, based on the algorithms described in this paper. Specifically, five programs are discussed below:

- *mslc*: Given a real sequence of length n and an upper bound U, this program locates the maximum-sum subsequence of length at most U in $O(n)$-time.
- *mslc_slow*: A brute-force $O(nU)$-time version of *mslc*.
- *mavs*: Given a real sequence of length n and a lower bound L, this program locates the maximum-average subsequence of length at least L in $O(n \log L)$.
- *mavs_slow*: A brute-force $O(nL)$-time version of *mavs*.
- *mavs_linear*: Instead of finding a good partner by binary search, as done in *mavs*, this program linearly scan right-skew segments for the good partnership. In the worst case, the time complexity is $O(nL)$. However, our empirical tests showed that it ran faster than *mavs* in most cases.

Table 1: Comparative evaluation of the five methods on a random integer sequence ranged from -50 to 50 of length 1,000,000. The time unit is second.

		Maximum Sum		Maximum Average		
n	L, U	*mslc*	*mslc_slow*	*mavs*	*mavs_slow*	*mavs_linear*
1,000,000	100	1.14	12.67	8.55	46.72	3.15
1,000,000	500	1.12	57.36	9.63	232.17	3.29
1,000,000	1,000	1.15	122.97	9.11	471.64	3.06
1,000,000	5,000	1.08	578.45	10.92	2331.52	3.36
1,000,000	10,000	1.12	1270.11	11.92	4822.25	3.13

Table 1 summarizes the comparative evaluation of the five programs on a random integer sequence ranged from -50 to 50 of length 1,000,000. These experiments were carried out on a Sun Enterprise 3000 UltraSPARC based system. Several length lower and upper bounds were used to illustrate their performance. For example, with L=U=5,000, *mslc* ran in 1.08 seconds, while *mslc_slow* took 578.45 seconds. It is not surprising to see that the running time of *mslc* was independent of U, and the running time of *mavs* increased slightly for larger L, whereas *mslc_slow* and *mavs_slow* grew proportionally to U and L, respectively. It is worth mentioning that *mavs_linear*, which scans right-skew segments linearly, ran even faster than *mavs*, which performs binary search among right-skew segments. The main reason was that the length of the maximum average consecutive subsequence seems usually quite close to L. Thus, *mavs_linear* could quickly locate the good partners by a linear scan.

We have also used the programs to analyze the homo sapiens 4q sequence contig of size 459kb from position 114130kb to 114589kb (sequenced by YMGC and WUGSC, GenBank accession number NT_003253). For instance, we found that the regions with the highest C+G ratio of length at least 200, 5000, and 10000 are 390396–390604 (C+G ratio 0.866), 389382–394381 (C+G ratio 0.513), and 153519–163520 (C+G ratio 0.475), respectively. This might suggest further biological experiments to better understand these GC-rich regions.

Huang's *LCP* program [13] is very efficient in finding in a sequence all GC-rich regions of length at least L. These GC-rich regions can be refined by locating their subregions with the highest C+G ratio by using our programs

Table 2: Refining the regions found by program *LCP*.

LCP				*mavs*			
Start	End	Length	C+G Ratio	Start	End	Length	C+G Ratio
3372	3444	73	0.740	3395	3444	50	0.740
6355	6713	359	0.805	6619	6671	53	0.943
7830	7933	104	0.779	7861	7912	52	0.808
8029	8080	52	0.769	8029	8081	52	0.769
8483	8578	96	0.760	8483	8532	50	0.800
9557	10167	611	0.782	9644	9695	52	0.981

mavs or *mavs_linear*. To illustrate this approach, we studied the rabbit α-like globin gene cluster sequence of 10621bp, which is available from GenBank by accession number M35026 [11]. The length cutoff L considered was 50, and the minimum ratio p was chosen at 0.7. Table 2 summarizes the empirical results. *LCP* found six interesting GC- rich regions. Take the region starting from position 6355 and ending at position 6713 for example. The length of this region is 359bp, and its C+G ratio is 0.805. Using the program *mavs*, we were able to find a subregion (of length 53bp) with the highest C+G ratio, which starts from position 6619 and ends at position 6671 with C+G ratio 0.943. Table 2 presents more examples of such refinements.

6 Concluding Remarks

In this paper, two fundamental problems concerning the search for the heaviest segment of a sequence with length constraints are considered. The first problem is to find a consecutive subsequence of length at most U with the maximum sum and the second is to find a consecutive subsequence of length at least L with the maximum average. We presented a linear-time algorithm for the first and an $O(n \log L)$-time algorithm for the second. Our results also imply efficient solutions for finding a maximum sum consecutive subsequence of length within a certain range and length-constrained ungapped local alignment. The algorithms have applications to several important problems in biomolecular sequence analysis.

It would be interesting to know if there is a linear-time algorithm to find

a maximum average consecutive subsequence of length at least L. It also remains open to develop an efficient algorithm for locating the maximum average consecutive subsequence of length between bounds L and U.

Acknowledgements

We thank Xiaoqiu Huang for his freely available program *LCP*. We would also like to thank Wen-Lian Hsu, Ming-Yang Kao, Ming-Tat Ko, and Hsueh-I Lu for helpful conversations, and the anonymous referees for their helpful comments and corrections. Y.-L. Lin was supported in part by grant NSC 89-2218-E-126-006 from the National Science Council, Taiwan. T. Jiang was supported in part by a UCR startup grant and NSF Grants CCR-9988353 and ITR-0085910. K.-M. Chao was supported in part by grant NSC 90-2213-E-010-003 from the National Science Council, Taiwan, and by the Medical Research and Advancement Foundation in Memory of Dr. Chi-Shuen Tsou.

References

[1] N.N. Alexandrov and V.V. Solovyev. Statistical significance of ungapped alignments. *Pacific Symposium on Biocomputing (PSB-98)*, pages 463–472, 1998.

[2] A. Arslan and Ö Eğecioğlu. Algorithms for local alignments with constraints. *Manuscript*, 2001.

[3] A. Arslan, Ö Eğecioğlu, and P. Pevzner. A new approach to sequence comparison: Normalized sequence alignment. *Bioinformatics*, 17:327–337, 2001.

[4] V. Bafna and D.H. Huson. The conserved exon method for gene finding. *Proc. Int. Conf. Intell. Syst. Mol. Biol. (ISMB)*, 8:3–12, 2000.

[5] S. Batzoglou, L. Pachter, J. Mesirov, B. Berger, and E. Lander. Comparative analysis of mouse and human DNA and applications to exon prediction. *Proc. Int. Conf. Comp. Mol. Biol. (RECOMB)*, 4, 2000.

[6] J. Bentley. *Programming Pearls*. Addison-Wesley, Reading, Massachusetts, 1986.

[7] M.S. Boguski, R.C. Hardison, S. Schwartz, and W. Miller. Analysis of conserved domains and sequence motifs in cellular regulatory proteins and locus control regions using new software tools for multiple alignment and visualization. *New Biol.*, 4:247–260, 1992.

[8] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[9] M. Gardiner-Garden and M. Frommer. CpG islands in vertebrate genomes. *J. Mol. Biol.*, 196:261–282, 1987.

[10] S. Hannenhalli and S. Levy. Promoter prediction in the human genome. *Bioinformatics*, 17:S90–S96, 2001.

[11] R.C. Hardison, D. Krane, C. Vandenbergh, J.-F.F. Cheng, J. Mansberger, J. Taddie, S. Schwartz, X. Huang, and W. Miller. Sequence and comparative analysis of the rabbit alpha-like globin gene cluster reveals a rapid mode of evolution in a G+C rich region of mammalian genomes. *J. Mol. Biol.*, 222:233–249, 1991.

[12] R.C. Hardison, J.L. Slighton, D.L. Gumucio, M. Goodman, N. Stojanovic, and W. Miller. Locus control regions of mammalian beta-globin gene clusters: combining phylogenetic analyses and experimental results to gain functional insights. *Gene*, 205:73–94, 1997.

[13] X. Huang. An algorithm for identifying regions of a DNA sequence that satisfy a content requirement. *CABIOS*, 10:219–225, 1994.

[14] F. Larsen, R. Gundersen, R. Lopez, and H. Prydz. CpG islands as gene marker in the human genome. *Genomics*, 13:1095–1107, 1992.

[15] A. Nekrutenko and W.-H. Li. Assessment of compositional heterogeneity within and between eukaryotic genomes. *Genome Research*, 10:1986–1995, 2000.

[16] P.S. Novichkov, M.S. Gelfand, and A.A. Mironov. Prediction of the exon-intron structure by comparison sequences. *Mol. Biol.*, 34:200–206, 2000.

[17] P. Rice, I. Longden, and A. Bleasby. EMBOSS: the European molecular biology open software suite. *Trends Genet.*, 16:276–277, 2000.

[18] P.H. Sellers. Pattern recognition in genetic sequences by mismatch density. *Bull. Math. Biol.*, 46:501–514, 1984.

[19] T.F. Smith and M.S. Waterman. The identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[20] N. Stojanovic, L. Florea, C. Riemer, D. Gumucio, J. Slightom, M. Goodman, W. Miller, and R. Hardison. Comparison of five method for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acids Research*, 27:3899–3910, 1999.

[21] M.S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *J. Mol. Biol.*, 197:723–728, 1987.

[22] B.Y. Wu, K.-M. Chao, and C. Y. Tang. An efficient algorithm for the length-constrained heaviest path problem on a tree. *Infomation Processing Letters*, 69:63–67, 1999.

[23] Z. Zhang, P. Berman, and W. Miller. Alignments without low-scoring regions. *J. Comput. Biol.*, 5:197–200, 1998.

[24] Z. Zhang, P. Berman, T. Wiehe, and W. Miller. Post-processing long pairwise alignments. *Bioinformatics*, 15:1012–1019, 1999.