Definitions and Algorithms in SEGID

1 Mathematical definitions

Given a (multiple) sequence alignment, SEGID first converts it into a sequence of numbers, where each number is the alignment score of a column. (SEGID also directly accepts a sequence of numbers as input.) Then it provides three algorithms to identify conserved segments (high score substrings):

1. **Longest segment (with average value lower bound):** given a string of numbers and a number A, find a substring (consecutive numbers in the string) with maximum length such that the average value of those numbers in the substring is at least A.

2. **All maximal length segments (with average value lower bound and length lower bound):** given a string of real numbers S, a number A, and an integer L (the lower bound of substring length), find all substrings T's satisfying the following properties:

(a) the average of any prefix/suffix of T is at least A;

(b) no proper superstring of T satisfies (a);

(c) the length of T is at least L.

Intuitively, all maximal length segments are some substrings of numbers with average value at least A. Property (a) implies that T does not include bad columns at its two ends; (b) guarantees that T is "maximal" in that it cannot be extended any more; and (c) allow users to set a length threshold to exclude extremely short segments. The set of all maximal length segments thus defined are disjoint, so it gives an intuitive view of clusters of good columns and is particularly suitable for coloring.

3. **N-Maximum scores segments:** given a string of real numbers S and two integers U (the upper bound of substring length) and N (number of segments), find N (disjoint) substrings with length at most U such that the k^th substring ($k \leq N$) has the highest score after the first $(k-1)$ (disjoint) substrings are selected.

We give below in detail the algorithms for the first two problems, which are both of time complexity O(n). The notion of *maximum score segment* (*N- maximum scores segments* when $N = 1$) is raised in *efficient algorithms for locating the length-constrained heaviest segments, with applications to biomolecular sequence analysis* (Y. L. Lin, T. Jiang, K. M. Chao, Journal of Computer and System Science, 2002), which also gives a linear algorithm to solve the problem. We repeat the

algorithm N times to solve the N- maximum scores segments problem. Since the time to find each segment is no more than $O(n)$, the total time complexity is $O(N * n)$.

2 Algorithm for computing the Longest Segment

First, we obtain a new string of numbers $S = (S_1, S_2, \ldots, S_n)$ by subtracting A from each of the numbers in the given string. Let (i, j) denote the substring $S_i, S_{i+1}, \ldots, S_j$. $sum(i, j)$ is the sum of all the numbers in (i, j). A substring (i, j) is *non-negative* if $sum(i, j) \geq 0$. Then the problem is to find a non-negative substring with maximal length. Calculate all $sum(1, i)$ in advance (in linear time), and then any $sum(i, j)$ can be retrieved in constant time.

Description of Algorithm

1. *preprocess.* For each $i = 1, 2, \ldots, n$, define $ms[i] = max_{j=i,i+1,\ldots,n} sum(i, j)$, and $msp[i] = j$, where $sum(i, j) = ms(i)$. To compute the vector of $ms[i]$ and $msp[i]$, we consider another vector $max[i] = max_{j=i,i+1,\ldots,n} sum(1, j)$. By definition, $max[n] = sum(1, n)$. $max[i]$ can be computed as $max[i] = max\{max[i + 1], sum(1, i)\}$ from right to left. Then, $ms[i]$ can be obtained as $ms[i] = max[i] - sum(1, i)$.

2. *main procedure.* For a fixed i, we want to compute the longest non-negative substring starting at position i. The main observation is that if (i, j) is non-negative, then either $(i, msp[j+1])$ is non-negative or (i, j) is the longest non-negative substring starting at position i. So, if $(i, msp[j + 1])$ is still non-negative, we get a longer substring (i, j'). We repeat the process until $sum(i, msp[j' + 1])$ is negative. In this way, we find $(i, l(i))$, the longest substring starting at position i that is non-negative. Another trick is that we do not have to compute $l(i)$ for all i's. Assume that we have computed $l(i_1)$. If $sum(i_1, i_2 - 1) \geq 0$ for some $i_2 > i_1$, then $l(i_2) \leq l(i_1)$ since we can always pad $(i_1, i_2 - 1)$ before $(i_2, l(i_2))$ to get a longer non-negative substring. Therefore, we can move forward i from i_1 until $sum(i_1, i - 1) < 0$. Now move forward j from $minl(i_1) + 1, i$, since $(i, l(i_1))$ is certainly non-negative when $l(i_1) > i$. The procedure terminates when i or j reaches the end of the string.

Theorem 1 *There is a linear time algorithm for the longest segment with average value lower bound problem.*

Proof. It is clear that the above algorithm correctly computes the longest segment subjected to average lower bound. The preprocessing stage takes linear time since it only need to scan through S once from right to left. In the main procedure, either the value of j or i increases by at least 1 in each iteration, and they never decrease. Thus, the total time required is at most $O(n)$. Therefore, the longest segment with average value lower bound problem can be solved in linear time. □

3 Algorithm for computing All Maximal Length Segments

For the all maximal length segments problem, we also subtract A from each number in the given string. Thus, property (a) becomes (a') "any prefix/suffix of T is non-negative".

Description of Algorithm

We process the resulting string S from left to right. Record the first number $S_i \geq 0$, and extend until some negative number S_j is met. Thus we detect that substring $(i, j-1)$ is a segment satisfying property (a'), and put it into a list. Then we go on to scan the text for next segment starting from S_{j+1} and satisfying (a'). Each time a new segment is detected, we examine whether it can be merged with other segments in the list to form a longer segment. The merging examination is based on the following property: Define the left partner $lp(i, j)$ of segment (i, j) to be the rightmost segment (k, l) in the list such that $sum(k, i-1) \geq 0$. If $sum(l+1, j) \geq 0$, then (k, j) satisfies (a'). In this case, (k, l) is inserted into the list, and correspondingly, segments between (k, l) and (i, j) are deleted. Otherwise, (i, j) cannot be merged, and is simply added into the list. In a word, while scanning through S, we detect new segments, examine if it can be merged, and merge or add them into the list. In this way, we maintain a segment-list SL where segments are ordered by position. Finally, when the end of the string is reached, those segments in SL whose lengths are less than L, the given length lower bound, are excluded.

Theorem 2 *There is a linear time algorithm for the all maximal length segments problem.*

To prove Theorem2, we need the following lemmas.

Lemma 3 *If $lp(i, j) = (k, l)$ and $sum(l+1, j) \geq 0$, then for any segment (p, q) in the list between (k, l) and (i, j), the following inequalities hold: (1) $sum(k, p-1) \geq 0$, and (2) $sum(q+1, j) \geq 0$.*

Proof. Note that (k, l) is the rightmost segment such that $sum(k, i-1) \geq 0$, and (p, q) is to the right of (k, l), Thus, we have $sum(p, i-1) < 0$. Since $i > p$, we have $sum(k, p-1) = sum(k, i-1) - sum(p, i-1)$. From the facts that $sum(k, i-1) \geq 0$ and $sum(p, i-1) < 0$, we have $sum(k, p-1) \geq 0$.

Now we prove inequality (2). If (2) does not hold, we have $sum(l+1, q) = sum(l+1, j) - sum(q+1, j) \geq 0$. If $(k, l) = lp(p, q)$, then from the condition $sum(l+1, q) \geq 0$, (k, l) and (p, q) should have been merged before segment (i, j) is created. So (k, l) cannot be the left partner of (p, q). Suppose $lp(p, q) = (p_1, q_1)$, where (p_1, q_1) is also a segment between (k, l) and (i, j). According to inequality (1), we have $sum(k, p_1 - 1) \geq 0$. Similarly, we can show $lp(p_1, q_1) = (p_2, q_2)$ is a segment between (k, l) and (p_1, q_1). This procedure can be repeated infinitely. However, the number of segments between (k, l) and (i, j) is finite, which leads to a contradictory. Therefore, inequality (2) must hold.□

Lemma 4 *A segment (i, j) satisfying property (a') can be merged into a longer segment (i', j) which also satisfies (a') if and only if (i, j)'s left partner (k, l) satisfies $sum(l + 1, j) \geq 0$. Moreover, if (i, j)'s left partner (k, l) satisfies $sum(l + 1, j) \geq 0$ then (k, j) satisfies (a').*

Proof. First, we prove that if $sum(l, j - 1) \geq 0$, then (k, j) satisfies property (a'), i.e. $\forall x \in [k, j]$, $sum(k, x) \geq 0$, and $sum(x, j) \geq 0$.

Case 1: x is in some segment (i_1, j_1).

From inequality (1) in Lemma 3 and the fact that (i_1, j_1) satisfies property (a'), we have $sum(k, x) = sum(k, i_1 - 1) + sum(i_1, x) \geq 0$. From inequality (2) in Lemma 3 and the fact that (i_1, j_1) satisfies (a'), we have $sum(x, j) = sum(x, j_1) + sum(j_1 + 1, j) \geq 0$.

Case2: x is not in any segment in the list, i.e., x is between segment (i_1, j_1) and (i_2, j_2).

Then the numbers between $j_1 + 1$ and x are all negative. (Otherwise, they would have formed another segment.) Thus, $sum(x, j) = sum(j_1 + 1, j) - sum(j_1 + 1, x - 1) \geq sum(j_1 + 1, j) \geq 0$. Similarly, the numbers between x and $i_2 - 1$ are all negative and we can conclude that $sum(k, x) \geq sum(k, i_2 - 1) \geq 0$.

Now, we prove that if there exists (i', j) satisfying (a') and $i' < i$, then (i, j)'s left partner (k, l) satisfies $sum(l + 1, j) \geq 0$.

i' must be in some segment in the list. (Otherwise $S_{i'}$ is negative, and prefix (i', i') breaks property (a').) We can assume that i' is the left point of the segment (i', j') in the list. (If i' is in the middle of segment (i_1, j_1), we can extend (i', j) to (i_1, j), which also satisfies (a').)

Since $(i', i - 1)$ is a prefix of (i', j), $sum(i', i - 1) \geq 0$. Thus, $(k, l) = lp(i, j)$ must be to the right of (i', j'). In that case, $(l + 1, j)$ is a suffix of (i', j). Thus, $sum(l + 1, j) \geq 0$.□

Lemma 5 *All left partners can be computed in time $O(n)$.*

Proof. We use the following procedure to compute the left partner of a new segment (i, j).

set (k, l) to be the last segment in the list;

while $sum(k, i - 1) < 0$ do $(k, l) = lp(k, l)$;

The above procedure correctly computes $(k, l) = lp(i, j)$, because $sum(k, i - 1) \geq 0$ when the while loop stops, and no segment (i', j') to the right of (k, l) can satisfy $sum(i', i - 1) \geq 0$. (Otherwise, (i', j') must have not been examined in while loop. Suppose (i', j') is between two examined segments (i_1, j_1) and (i_2, j_2). According to the algorithm, we have $(i_1, j_1) = lp(i_2, j_2)$. However, $sum(i', i - 1) \geq 0$ and $sum(i_2, i - 1) < 0$ results in $sum(i', i_2 - 1) \geq 0$, and (i', j') is to the right of (i_1, j_1), so (i_1, j_1) cannot be left partner of (i_2, j_2), which leads to a contradictory.)

If a segment is examined in while loop and does not halt the loop (we call this a non-halting step), then the segment is "bypassed" by the 'lp' link list and never examined again later. Therefore, the total number of non-halting steps is no more than the number of segments that have appeared

in the list, which is bounded by $O(n)$. On the other hand, the number of halting steps equals to the number of segments. Thus, the total time required to compute all left partners is $O(n)$.□

Now we are ready to prove Theorem 2.

Proof. First, the algorithm correctly computes the set of all maximal length segments.

we claim that after detection of each segment, the segment-list SL is the set of all segments satisfying property (a') and (b) with respect to substring $(1, i)$, where S_i is the last position scanned. The "if" part of Lemma 4 guarantees all segments in SL satisfy property (a'). And for any segment, there is no proper superstring in $(1, i)$ satisfying (a'), because otherwise the condition in Lemma 4 must be satisfied, and the segment should have been merged. Thus, when we reach the end of S, SL is just the set of all segments satisfying property (a') and (b). The final deletion step checks property (c), and returns the set of all maximal length segments.

Then, we show the time complexity of above algorithm is $O(n)$.

Obviously, the total time of detecting new segments is linear. According to Lemma 5, the total time to compute left partners for all these detected segments is $O(n)$, and so is the time for merging examination. If the new segment is simply added to SL, it takes time $O(1)$; otherwise, some segments need to be deleted from SL, but the time of deleting a segment can be charged to that of adding a segment, and the number of segments added to SL is at most n. In all, the time required by above algorithm is linear.

Therefore, the algorithm given at the beginning of this section is a linear algorithm for the all maximal length segments problem.□