2006:14

RESEARCH REPORT

Computing Maximum-Scoring Segments in Almost Linear Time

Fredrik Bengtsson and Jingsen Chen

Luleå University of Technology
Department of Computer Science and Electrical Engineering

2006:14 • ISSN:1402-1528 • ISRN: LTU - FR -- 06/14 -- SE

Computing Maximum-Scoring Segments in Almost Linear Time

Fredrik Bengtsson and Jingsen Chen

Department of Computer Science and Electrical Engineering
Luleå University of Technology
S-971 87 Luleå
SWEDEN

August 10, 2006

Abstract

Given a sequence, the problem studied in this paper is to find a set of k disjoint continuous subsequences such that the total sum of all elements in the set is maximized. This problem arises naturally in the analysis of DNA sequences. The previous best known algorithm requires $\Theta(n \log n)$ time in the worst case. For a given sequence of length n, we present an almost linear-time algorithm for this problem. Our algorithm uses a disjoint-set data structure and requires $O(n\alpha(n, n))$ time in the worst case, where $\alpha(n, n)$ is the inverse Ackermann function.

1 Introduction

In the analysis of biomolecular sequences, one is often interested in finding biologically meaningful segments, e.g. GC-rich regions, non-coding RNA genes, transmembrane segments, and so on [11, 12, 13]. Fundamental algorithmic problems arising from such sequence analysis are to locate consecutive subsequences with high score (given a suitable scoring function on the subsequences). In this paper, we present an almost linear time algorithm that takes a sequence of length n together with an integer k and computes a set of k non-overlapping segments of the sequence that maximizes the total score. If a general segment scoring function (other than the sum of the segments as in this paper) is used, then the problem of finding a k-cover with maximum score can be solved in $O(n^2k)$ time [1, 4]. When the score of a segment is the sum of the elements in the segment, the previous best known algorithm runs in $\Theta(n \log n)$ time [11]. Our new algorithm requires $O(n\alpha(n, n))$ time in the worst case, where $\alpha(n, n)$ is the inverse Ackermann function.

The problem studied can be viewed as a generalization of the classical *maximum sum subsequence problem* introduced by Bentley [6]. The latter is to find the continuous segment with largest sum of a given sequence and can be solved in linear time [7, 16]. Several other generalizations of this classical problem have been investigated as well. For example when one is interested in not only the largest, but also k continuous largest subsequences (for some parameter k) [2, 3, 5, 14]. Other generalizations arising from bioinformatics is to look for an interesting segment (or segments) with constrained length [8, 9, 15].

The paper is organized as follows. In Section 2 the problem is defined and Csűrös work on this problem [11] is discussed. Some minor flaws of his algorithm are also investigated. Section 3 gives an overview of our algorithm while the details of the algorithm is presented in Section 4. Finally, the analysis of our algorithm appears in Section 5 and the paper is concluded with some open problems in Section 6.

2 Preliminaries

After introducing the problem to be studied and some notations, we will describe Csűrös' algorithm [11] for finding maximum-scoring segments and fix some minor flaws in his paper.

2.1 Problem and Notations

Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of real numbers, let $X_{i,j}$ denote the subsequence of consecutive elements of X starting at index i and ending at index j; i.e. the *segment* $X_{i,j} = \langle x_i, x_{i+1}, \ldots, x_j \rangle$. A segment $X_{i,j}$ is *positive* (*negative*) if its *score* (*sum* or *value*) $\sum_{\ell=i}^{j} x_\ell > 0$ (< 0). Call $X_{i,j}$ a *non-negative run* (*negative run*) of X if

- $x_\ell \geq 0$ (or < 0) for all $i \leq \ell \leq j$;
- $x_{i-1} < 0$ (or ≥ 0) if $i > 1$; and
- $x_{j+1} < 0$ (or ≥ 0) if $j < n$.

Given an integer $1 \leq k \leq n$, a *k-cover* for the sequence X is a set of k disjoint non-empty segments of X. The *score* (*sum* or *value*) of a k-cover is determined by adding up the sums of each of its segments.

Definition 1. *An* optimal k-cover *for a given sequence X is a k-cover of X whose score is the maximum over all possible k-covers of X.*

2.2 Csűrös Algorithm

By adapting Kadane's linear-time algorithm for the maximum-scoring segments problem (where $k = 1$) [7], Csűrös [11] presented an $O(nk)$-time algorithm for the general k. Moreover, he showed that an optimal k-cover can be created from an optimal $(k+1)$-cover by removing a segment with minimum absolute value or concatenating that segment with its neighbors. Therefore, after a segmentation of the input sequence, an optimal k-cover can be produced by a series of concatenations of neighboring segments with current minimum absolute values. More precisely, the following algorithm is proposed by Csűrös [11]:

Input: A sequence X of n real numbers and an integer $1 \leq k \leq m$. (Where m is the length of Y, as described shortly.)

1. Partition X into runs and let each run be replaced by its sum. Let the sequence obtained be $Y = \langle y_1, y_2, \ldots, y_m \rangle$, where $m \leq n$. That is, each run computed in X corresponds to an element of Y.
2. while $\|\{y \in Y : y > 0\}\| > k$ do
   - Select the element y_j with minimum absolute value of Y. During this selection, let $y_1 = y_m = \infty$, so that these two elements never get selected.
   - Replace elements y_{j-1}, y_j and y_{j+1} in Y with one single element having value $y_{j-1} + y_j + y_{j+1}$.
3. Return the segments of X corresponding to all the positive elements in Y.

The above algorithm runs in $\Theta(n \log n)$ in the worst case [11]. This can be done by storing the blocks at the leaves of a balanced binary search tree where each block has a pointer to its successor and predecessor. The correctness of this algorithm was shown by Csűrös [11]. However, some observations about this algorithm will be made here.

First, this algorithm does not consider the case (although trivial) when $k > m$, where m is the the number of positive runs in the input sequence. However, this can be solved by preprocessing the input data. We will show in the next section how this can easily be accomplished.

Next, this algorithm contains a minor flaw when smallest elements (with respect to their absolute values) are the first or the last element of the current sequence. It is claimed [11] that these elements should be avoided and never selected (by assigning ∞ to them). However, this can lead to an incorrect solution in some cases. Consider the sequence $X = \langle 2, -5, 8, -1, 4, -7, 6 \rangle$ and $k = 2$. Csűrös' algorithm will first replace $\langle 8, -1, 4 \rangle$ with $8 - 1 + 4 = 11$

(since the element (-1) has the minimum absolute value), obtaining a new sequence $\langle 2, -5, 11, -7, 6\rangle$. Then, the smallest absolute value in the new sequence is 2, but the algorithm avoids selecting this element, due to the fact that it lies at the first position of the sequence. Instead, the next smallest element (which is -5) is selected and the sequence $\langle 8, -7, 6\rangle$ is the outcome of this round (since $2 - 5 + 11 = 8$). Now, the number of positive elements in the sequence $\langle 8, -7, 6\rangle$ is 2, which equals k, and we are done. Thus, the score of an optimal 2-cover of X is claimed to be $8 + 6 = 14$. Observe that the following 2-cover, $\{\langle 8, -1, 4\rangle, \langle 6\rangle\}$, has a total sum of 17 which is larger than the previous of 14. Hence, it is not correct to just avoid selecting the first and last elements. Instead, these elements should be selected in the same way as for other elements.

For such special cases, say y_1 has the current smallest absolute value, we will replace $\langle y_1, y_2\rangle$ with $\langle y\rangle$, where $y = y_1 + y_2$. The soundness of this correction will be presented together with the analysis of our algorithm.

3 The Algorithm

By generalizing the recursive relation between maximum-scoring segments [11], we are able to design an algorithm that computes an optimal k-cover in almost linear time; the complexity of our algorithm is independent of the parameter k. The main idea is to transform the input sequence into a series of sequences of decreasing lengths under cover-preserving. We employ the union-find algorithmic technique to achieve the efficiency.

Our algorithm consists of three phases: Preprocessing, Partitioning, and Concatenating. The preprocessing phase deals with some trivial cases of the problem and simplifies the input for the rest of the algorithm. The latter two phases choose subsequences of which candidate segments to the optimal k-cover are examined and will be executed in an iterative fashion.

A special class of sequences plays an important role in our algorithm design, namely alternating sequences.

Definition 2. *A sequence $Y = \langle y_1, y_2, \ldots, y_m\rangle$ of real numbers is an* alternating sequence *if m is odd, $y_1, y_3, \ldots,$ and y_m are positive, and $y_2, y_4, \ldots,$ and y_{m-1} are negative. An alternating sequence is an* a-sequence *if its elements are mutually distinct.*

The following fact implies that finding an optimal k-cover in some alternating sequence needs only constant time.

Observation 1. *All the positive elements of an alternating sequence of length $2k - 1$ represent an optimal k-cover of the sequence.*

3.1 Preprocessing

This phase processes special cases of the problem and, if needed, finds the segmentation of the input into runs. The segmented input will be of alternating type and work as input for the two phases of our algorithm that follows. We treat the preprocessing step here separately from the other two steps of the algorithm, which simplifies the presentation of the latter. However, it does not make the problem to be solved simple.

The *special cases* of the problem for finding an optimal k-cover are investigated with respect to the number m_1 of the non-negative elements, the number m_2 of the non-negative runs, and the number m_3 of the negative runs in the input. For a given sequence X of length n and an integer k, $1 \leq k \leq n$, we proceed as follows.

1. Scan the sequence X and compute m_1, m_2, and m_3.
2. If $k \geq m_1$, then compute the $(k - m_1)$ largest elements among all the negative ones. Now, all the non-negative elements together with the negative elements just computed form an optimal k-cover, where each member of the cover is of length 1.
3. If $m_2 \leq k < m_1$, then all the non-negative runs represent an optimal solution by splitting the runs until the number of non-negative blocks equals k. Moreover, we can, in this case, delete any subsequence (prefix or suffix of X) containing only zeroes according to the value of k without affecting the optimal solution.
4. If $1 \leq k < m_2$ a segmentation of X will be performed.

The *segmentation* of the input produces a new sequence that serves as the input to the iteration (the partitioning and concatenating) phases of our algorithm. Segmenting a sequence X (where $k < m_2$) works as follows:

1. Find all the non-negative runs and negative runs of X.
2. Construct a new sequence Y from X by replacing every run with its score. Clearly, Y is a sequence of length $m_2 + m_3$.
3. Let m' be the number of positive elements of Y.
4. If $k < m'$, then delete all the elements equal to zero from Y.
5. If $k \geq m'$, then the optimal k-cover consists of all the positive runs of X (positive elements of Y) together with $(m' - k)$ of the zero elements of Y and we are done.
6. Negative elements at both ends of Y (if any) are removed from Y.

Moreover, by storing the indices of all the runs of X, one can easily refer each element of Y to its corresponding segment of X. Clearly, the whole preprocessing phase takes at most $O(n)$ time in the worst case. Furthermore, any optimal k-cover of Y corresponds to an optimal k-cover of X. In fact,

Proposition 1. *Given a sequence X of real numbers, if the number of non-negative runs of X is at least k, then there is an optimal k-cover of X such that any member of the optimal cover is either an entire non-negative run of X or a concatenation of neighboring runs of X.*

Proof. Consider any k-cover C of X. If the cover C did contain only a part of a non-negative run, we could just include the rest of that non-negative run; obtaining a new k-cover with a total score at least as good as the old one. Hence the cover C would not be optimal.

If the cover C did contain only a part of a negative run, we could just exclude this negative run in order to increase the value of the cover. Hence the cover C would not be optimal. □

To sum up, we have

Corollary 1. *Let $T(n)$ and $A(m)$ be the time needed to compute an optimal k-cover of a sequence X of length n and of an alternating sequence Y of length m, respectively. Then,*

- $T(n) = O(n + A(n))$.
- *If Y is the sequence resulted from preprocessing X, then the value of an optimal k-cover of X is the same as that of Y.*

If an optimal k-cover of X has not yet been computed during the preprocessing (Steps 1-3), we obtain an *alternating sequence* $Y = \langle y_1, y_2, \ldots, y_m \rangle$, where $m \leq n$. Moreover, we can define a new order ($\prec$) on the elements of Y such that $y_i \prec y_j \Leftrightarrow \{y_i < y_j \vee y_i = y_j \wedge i < j\}$. However, throughout the paper we will use the standard notation to simplify presentation. Therefore, after $O(n)$ preprocessing time, we consider only alternating sequences of mutually distinct real numbers (that is, *a-sequences*) in the rest of our algorithm. The following property of such sequences characterizes optimal covers of the sequences.

Observation 2. *Let Y be an alternating sequence of length m and $k \leq \lceil m/2 \rceil$. Then, each member of an optimal k-cover of Y will always be a segment of Y of odd length.*

Proof. Suppose there is a segment of an optimal k-cover that does have an even length. Then either a prefix or a suffix of this segment will be negative. We could thus remove that negative prefix or suffix; resulting in a new k-cover with a higher total score. Hence, the cover was not optimal. This is a contradiction. □

3.2 Partitioning and Concatenating

In this subsection, we give an overview of these two phases of our algorithm, whereafter a detailed presentation follows. Our approach in computing an optimal k-cover for a given a-sequence Y is to construct a series of a-sequences from Y while the lengths of the sequences are decreasing and the optimal k-cover remains the same.

More precisely, we first construct a shorter a-sequence Y' from Y. Such a construction can be accomplished with concatenation operations run on an odd number of consecutive elements of Y according to Observation 2. By partitioning Y into smaller pieces and concatenating carefully chosen segments, we can reduce the number of candidate segments for the optimal k-cover fast. If the length of Y' equals $2k-1$, we are done; otherwise, repeat the construction starting from Y' . Such a procedure will be run iteratively.

For a given a-sequence Y of length m and an integer k, $1 \leq k \leq m$, we consider only the case when $k < \lceil m/2 \rceil$. Otherwise, the problem is taken care of by our preprocessing phase. Let Y_t $(t = 1, 2, \cdots)$ be an a-sequence of length m_t associated with a working sequence S_t of length n_t, where $Y_1 = Y$ and $S_1 = Y$. The working sequence contains elements of Y that are currently interesting for the algorithm. In the following, each element of Y_t and S_t refers to the block of concatenated elements that the element currently corresponds to. The t^{th} iteration of our algorithm (in particular, the partitioning and concatenating procedure) is as follows. Let ξ_0 be the largest absolute value of the elements in Y and let $r_1 = \lceil \|S_1\|/2 \rceil$.

Input: Y_t, S_t, a threshold r_t, and a pivot ξ_{t-1}, where $k < \lceil m_t/2 \rceil$
Output: Y_{t+1}, S_{t+1}, r_{t+1}, and ξ_t, where $m_{t+1} \leq m_t$ and $n_{t+1} < n_t$.

1. Partition

   (a) Compute the $(r_t)^{th}$ largest absolute value ξ_t of all the elements in S_t.

   (b) Let D_t be the sequence containing all the elements of S_t whose absolute value is less than or equal to ξ_t. Preserve the ordering among the elements from Y_t; the indices of the elements are in increasing order.

2. Concatenation

   (a) Let Y_t' be the sequence resulting from, for each element y in D_t, repeatedly replacing some blocks in Y_t of odd lengths around y with their score until every element has an absolute value not less than ξ_t. Let k' be the number of positive elements in Y_t'.

   (b) If $k < k'$ (that is, we merged too few blocks in the previous step), then

- $S_{t+1} \leftarrow \langle$All the elements now in S_t whose absolute value lies between ξ_t and ξ_{t-1}; if some elements now belong to the same block, then just insert one of them into $S_{t+1}\rangle$
- $r_{t+1} \leftarrow \lceil \|S_{t+1}\| / 2 \rceil$
- $Y_{t+1} \leftarrow Y'_t$
- $\xi_t \leftarrow \xi_{t-1}$.

(c) If $k > k'$ (that is, we merged too many blocks in the previous step), then $S_{t+1} \leftarrow D_t$, $r_{t+1} \leftarrow \lceil \|S_{t+1}\|/2 \rceil$, and $Y_{t+1} \leftarrow Y_t$.

(d) If $k = k'$, then we are done.

The goal is to eventually construct an a-sequence of length $2k - 1$ and hence the optimal k-cover is found due to Observation 1. With a careful implementation, the lengths of the a-sequences constructed will gradually decrease. In accomplishing our task within the desired time bound, we cannot afford to actually construct all the sequences Y_t $(t = 1, 2, \cdots)$. In such case, we may end up with an algorithm that takes $\Omega(m \log m)$ time in the worst case.

In the following, we show how to implement the algorithm efficiently. Actually, we never construct Y_t, but operate directly on Y all the time.

4 Algorithmic Details

Recall that the input now is an a-sequence $Y = \langle y_1, y_2, \ldots, y_m \rangle$ and an integer k, where $k < \lceil m/2 \rceil$. To implement the above algorithm efficiently, we will employ a disjoint-set data structure [10] augmented with extra information such as indices and scores of blocks. In addition to the standard disjoint-set data structure, we store the following extra fields at the leader node of each set:

- The index, in Y, of the leader
- The range, in Y, of the largest block created to which the leader belongs
- The score of the block

4.1 Union-Find

Initially, for $i = 1, 2, \ldots, m$, we perform `MakeSet`(y_i) with $\{i, (i, i), y_i\}$ as extra information. Also, let `FindSet`(y_i) return this extra information. For any two elements x and y in Y, the operation `Union`(x, y) is performed as follows.

`Union(x,y)`

1. Let $(i, (i_1, i_2), s_x) = \texttt{FindSet}(x)$.

2. Let $(j, (j_1, j_2), s_y) = \texttt{FindSet}(y)$.

3. If $i = j$, then return; else let ℓ be the new leader index decided by the disjoint-set data structure (i.e., ℓ is either i or j).

4. The new extra information will be $(\ell, (\min\{i_1, j_1\}, \max\{i_2, j_2\}), s_x + s_y)$.

In the above procedure, if $j_1 - i_2 = 1$ (that is, the blocks joined are adjacent), the extra information maintained for each block will represent the intended extra information in the previous subsection. This is the case that is needed by our algorithm later on.

For the simplicity, for any y in Y, denote by $b(y)$ and $v(y)$ the index, in Y, of its leader and the score of its block, respectively; i.e., $(b(y), (j_1, j_2), v(y)) = \texttt{FindSet}(y)$. It is important to emphasize that any block created with `Union`s can be represented by an arbitrary element of Y from within the block. Any intermediate sequence used during the process contains only the original element from Y. Consider now the t^{th} iteration of the phases for $t = 1, 2, \cdots$.

4.2 Partition

In this step, we partition the current input S_t according to the given threshold r_t. Notice that each element in S_t corresponds to some block of Y. Therefore, it is necessary to map from the elements stored in S_t to the blocks created by the previous concatenations. Thus, one `FindSet` is done on each element of S_t.

Now, we can do the desired selection on all the absolute values obtained using a worst-case linear-time selection algorithm [10]. After that, a partition is performed around the pivot, ξ_t, and all the elements of S_t whose absolute value is less than or equal to ξ_t are included in a sequence D_t. Observe that for each comparison done on a pair of elements, one must do `FindSet` operations first. The output of the partition step is (D_t, ξ_t). Thereafter, a series of replacements is applied to D_t in order to create a new a-sequence of smaller length.

4.3 Concatenation

In this subsection, we focus on the problem of constructing shorter a-sequences from a given a-sequence. One approach is to repeatedly concatenate blocks and replace them with their score. However, one cannot just choose an arbitrary block and then do the replacement, because this could potentially yield incorrect results. What we want from the replacement is that the operation should result in both a shorter a-sequence and

that the optimal k-cover of this sequence remains the same as the one before the operation.

Recall that for a given input sequence, D_t and ξ_t, to the concatenation step, we aim at editing the sequence Y so that there is no element y in Y with $|v(y)| \leq \xi_t$. By this we mean that all the elements in D_t and all the blocks created within this step with smaller absolute values will be involved in some replacement. From Fact 2, we know that blocks of odd lengths may be a good choice for preserving optimal k-covers. Among all possible replacements of blocks, one special kind of the replacements, the *merge*, will do the job.

4.3.1 The Merge Operation

A *merge* operation only applies to segments of length three. In particular, a *merge* can apply to a segment $\langle y_{i-1}, y_i, y_{i+1}\rangle$ only if $|y_i| < \min\{|y_{i-1}|, |y_{i+1}|\}$. The result of a merge on the segment $\langle y_{i-1}, y_i, y_{i+1}\rangle$, denoted by *merge*$(y_{i-1}, y_i, y_{i+1})$, is a new block with a value equal to $y_{i-1}+y_i+y_{i+1}$; realizable with $\texttt{Union}(\texttt{Union}(y_{i-1}, y_i), y_{i+1})$. Call such an operation a *merge around* y_i. Specially, a *merge around* y_1 implies $\texttt{Union}(y_1, y_2)$ if $|y_1| < |y_2|$ and a *merge around* y_m implies $\texttt{Union}(y_{m-1}, y_m)$ if $|y_{m-1}| < |y_m|$. Hence, the elements y_1 and y_m can be treated in the same way as other elements. In general, for any y in Y, let $(b(y), (i, j), s) = \texttt{FindSet}(y)$. A merge around y is then the merge operation on the segment $\langle y_{i-1}, y_{b(y)}, y_{j+1}\rangle$ (i.e., $\texttt{Union}(\texttt{Union}(y_{i-1}, y_{b(y)}), y_{j+1})$) if applicable. In this case, such a merge is also called a *merge around* y' for any y' in the block $Y_{i,j}$. Hence, we use the term element y to mean both the original element in Y and interchangeably the longest block created containing y.

The merge operation designed above has some nice properties which ensures the correctness of our algorithm. First, each merge operation will result in a new block with larger absolute score. Namely,

Proposition 2. *Let y be an element in the block resulting from* merge$(y_{i_1}, y_{i_2}, y_{i_3})$. *Then,* $|v(y)| \geq \max\{|v(y_{i_1})|, |v(y_{i_2})|, |v(y_{i_3})|\}$.

This is because a merge operation can apply to $y_{i_1}, y_{i_2}, y_{i_3}$ only if $|v(y_{i_2})| \leq \min\{|v(y_{i_1})|, |v(y_{i_3})|\}$. Thus, $|v(y_{i_1}) + v(y_{i_2}) + v(y_{i_3})| = ||v(y_{i_1}) + v(y_{i_3})| - |v(y_{i_2})|| = ||v(y_{i_1})| + |v(y_{i_3})| - |v(y_{i_2})|| \geq \max\{|v(y_{i_1})|, |v(y_{i_3})|\}$.

Next, alternating sequences are invariant under merge operations. More precisely,

Proposition 3. *Let M be the set of merge operations performed on a given alternating sequence $Y = \langle y_1, y_2, \ldots, y_m\rangle$ and Y_M the sequence (called the* compact version *of Y under M) constructed from Y by replacing each merged block with a singleton element. If no merge is done around neither y_1 nor y_m, then*

1. $\|Y_M\| = \|Y\| - 2\,\|M\|$.

2. *Y_M is also an alternating sequence.*

Proof. We prove by induction on the number $\|M\|$ of merge operations performed. For the case of $\|M\| = 1$, say $merge(z_{i-1}, z_i, z_{i+1})$ results in a sequence Z' from an alternating sequence $Z = \langle z_1, z_2, \ldots, z_\ell \rangle$. Assume $z_i > 0$, then $z_{i-2} > 0$ and $z_{i+2} > 0$. Since $z_i = |z_i| < \min\{|z_{i-1}|, |z_{i+1}|\}$ we know that $z_{i-1} + z_i + z_{i+1} = -|z_{i-1}| + z_i - |z_{i+1}| < -|z_{i+1}| < 0$. Hence, Z' is an alternating sequence containing one positive element less than that in Z. Thus, $\|Z'\| = \ell - 2$. The case of $z_i < 0$ is similar. Now, let Z' be the new input and the result easily follows from the induction. □

Moreover, if Y is an a-sequence, so is Y_M. Hence, by repeatedly merging blocks one can obtain some compact version of Y, particularly a version with no smaller elements (that is, all its elements have absolute values greater than ξ_t).

4.3.2 Repeated Merges on D_t

Obviously, in order to ensure that there is no element y in Y with $|v(y)| \leq \xi_t$, at least all the elements in D_t must be involved in some merges. For each element in D_t and all the newly formed blocks (regarded as new elements of Y_t for some t), we need to decide whether a merge operation *will* be performed around it. For y in Y, define $Test(y) = true$ if a merge *can* be done around y according to the definition of the merge operation and $|v(y)| \leq \xi_t$; otherwise, $Test(y) = false$. Let $D_t = \langle d_1, d_2, \ldots, d_{n_t} \rangle$. Basically, we traverse D_t from d_1 to d_{n_t} and, for each element, determines if it should be merged. The current element is merged repeatedly until it is larger than the pivot. When it is, then its left neighbour is checked again to see if it should be merged again. More precisely:

1. For $j = 1, \ldots, n_t$ let $i_j = b(d_j)$ (i_j is the index, in Y, to the set leader for d_j).

2. Set up a double-linked list L_t with its j^{th} node containing a *prev* pointer, a *next* pointer, and a numeric field *num* storing i_j, $j = 1, \ldots, n_t$.

3. Let p point to the first element of L_t.

4. while $L_t \neq \emptyset$ do

   - $\ell \leftarrow p.num$
   - If $Test(y_\ell) = false$, then $p \leftarrow p.next$
   - If $Test(y_\ell) = true$, then

– Do a merge around y_ℓ. If the element corresponding to the node $p.prev$ is involved in this merge as well, then delete the node $p.prev$. Perform similar for the node $p.next$.
– If $|v(y_\ell)| > \xi_t$ (the block after merge is larger than th threshols), then delete the node p (we should not perform any more merges on block p) and let $p \leftarrow p.prev$ (we check if the previous block should be merged again).

As mentioned before, each merge operation creates a compact version of Y with the length decreased by 2. Therefore, the number of positive elements in the current version, Y_t of Y, (after all the merges done) can easily be counted when doing merges. This means that we do not need to actually construct the compact versions Y_{t+1} of Y at the moment. Only when we finally find an a-sequence of length $2k-1$, that compact version of Y is then computed; which costs in the worst case $O(m)$.

Furthermore, if a merge around y_1 (or y_m) was performed during the process, then a new block with negative score (which is either the prefix or suffix of Y) appears. The reason for the block being negative is that $y_1 < y_2$, otherwise the merge would not have occurred. Such a block can immediately be removed, because it is always unnecessary to include it in the solution to the current compact version Y_{t+1} of Y. Thus, if the block $\langle y_1 y_2 \rangle$ is selected for merge, we can effectively remove this block without doing any merge. The merge around y_m is analogous. Hence, Y_{t+1} is an a-sequence as well.

4.3.3 t^{th} Iteration

The goal of the t^{th} iteration is to construct implicitly a new a-sequence Y_{t+1} of length m_{t+1} from an a-sequence Y_t of length m_t. From the construction, we know that $m_{t+1} \leq m_t$. The equality holds when there are too few positive elements in the compact version of Y after the merges. In this case, we cancel all the merges performed in this iteration. In order to be able to cancel the merge operations performed earlier, we record and store all changes made to the disjoint-set data structure. We need only store all the changes made in the current iteration. This is because that if there are not too few positive elements in the resulting sequence, we will never need to cancel the previous merge operations.

Observe that the iteration works on the working sequence S_t associated with Y_t, decides whether a merge should be performed around every element, and produces a new working sequence S_{t+1} associated with Y_{t+1}. Fortunately, the working sequences get shorter after every iteration. In fact, the lengths of such sequences decrease very fast, which implies that the number of iterations performed in our algorithm is not too many.

From Propositions 2 and 3, the lengths of a-sequences (i.e., the compact versions of Y) will eventually decrease to $2k-1$; say the last one is $Y_{t'}$. We

will show in the next section that the optimal k-cover of Y_{t+1} corresponds to an optimal k-cover of Y_t, and thus is represented by all the positive elements in $Y_{t'}$.

5 Correctness

After preprocessing, our algorithm takes an alternating sequence $Y = \langle y_1, y_2, \ldots, y_m \rangle$ and an integer k, $1 \leq k < \lceil m/2 \rceil$, and computes an optimal k-cover of the sequence. The approach is to construct a set of alternating sequences $\{Y_t\colon\ t = 1, 2, \cdots, K\}$ of length m_t, for some K, where $Y_1 = Y$, such that

- $k < \lceil m_t/2 \rceil$ and $m_{t+1} \leq m_t$ for $t = 1, 2, \cdots, K-1$
- $k = \lceil m_K/2 \rceil$

For simplicity, call the number of positive elements in an alternating sequence its *p-length*. Thus, the optimal ℓ-cover of an alternating sequence of p-length ℓ consists of all the positive elements in the sequence only. By definition, the p-length of Y is $\lceil m/2 \rceil$. The following property is straightforward from the discussion in the previous section.

Observation 3. *For a given threshold $\xi > 0$, if the value ξ lies between the smallest and the largest absolute value in Y, then the sequence $Z = Concatenate(Y, \xi)$ is always an alternating sequence of length at most m.*

Moreover, our algorithm also have the following nice recursive property:

Proposition 4. *Given a threshold $\xi > 0$ and an alternating sequence Y of length m, let ℓ be the p-length of $Z = Concatenate(Y, \xi)$. If $\ell < \lceil m/2 \rceil$ and $0 < \xi' < \xi$, then by ignoring all the merge operations around elements in $(\xi', \xi]$ ever done, the algorithm that produced Z will now output a sequence equivalent to $Concatenate(Y, \xi')$.*

Proof. Denote by $Algo(\xi)$ the algorithm that produced $Z = Concatenate(Y, \xi)$. Consider any operation $merge(y', y, y'')$ done by the algorithm $Algo(\xi)$ for any $y \in (\xi', \xi]$. On executing this merge by $Algo(\xi)$, one must have that $|v(y)| < |v(y')|$ and $|v(y)| < |v(y'')|$ before the merge. Hence, ignoring this merge would not change the fact that both the elements y' and y'' were above ξ'. For all the merge operations done by $Algo(\xi)$ after $merge(y', y, y'')$, notice that a merge can only increase the absolute value of the block. Hence, if y'' was involved in any merge $merge(y'', *, *)$ later on, then the element corresponding to $merge(y'', *, *)$ would lie above ξ' with or without the execution of $merge(y', y, y'')$. Therefore, the algorithm $Algo(\xi)$ without all its merge operations around elements in $(\xi', \xi]$ would behave exactly the same as some algorithm that produces $Concatenate(Y, \xi')$. □

Furthermore, the number of merges around elements in $(\xi', \xi]$ as above done by our algorithm are bounded. Namely,

Proposition 5. *Given two thresholds $0 < \xi' < \xi$, let $\ell(\xi)$ and $\ell(\xi')$ be the p-lengths of the sequences $Z = Concatenate(Y, \xi)$ and $Z' = Concatenate(Y, \xi')$, respectively. If $1 < \ell(\xi) < \ell(\xi') < \lceil m/2 \rceil$, then the number of merge operations around elements in $(\xi', \xi]$ done by the algorithm that produced the sequence Z equals $\ell(\xi') - \ell(\xi)$.*

Proof. Induction on the difference between $\ell(\xi')$ and $\ell(\xi)$.

Denote by $Algo(\xi)$ and $Algo(\xi')$ the algorithms that produce the sequences Z and Z', respectively. Observe from Proposition 3 that each merge operation decreases the p-length of the sequence by exactly 1. Thus, the numbers of merge operations performed by $Algo(\xi)$ and $Algo(\xi')$ are $\lceil m/2 \rceil - \ell(\xi)$ and $\lceil m/2 \rceil - \ell(\xi')$, respectively.

Consider first the case when $\ell(\xi') - \ell(\xi) = 1$. In this case, we claim that $Algo(\xi)$ has performed *exactly* one merge around some element $y \in (\xi', \xi]$.

It is clear that $Algo(\xi)$ must have done at least one such merge operation. Otherwise, the algorithm $Algo(\xi)$ would behave exactly the same as if the input threshold ξ were replaced with a new threshold ξ'; resulting in an alternating sequence Z'' of p-length $\ell(\xi')$ equivalent to Z'. However, the number of merge operations performed by $Algo(\xi)$ is $\lceil m/2 \rceil - \ell(\xi)$ and hence *any* sequence produced by $Algo(\xi)$ would have a p-length of $\lceil m/2 \rceil - (\lceil m/2 \rceil - \ell(\xi)) = \ell(\xi)$; contradicting the fact that $\ell(\xi') = \ell(\xi) + 1$.

On the other hand, assume that the algorithm $Algo(\xi)$ has performed two or more merges around elements in $(\xi', \xi]$. From Proposition 4 we know that $Algo(\xi)$ will produce an alternating sequence equivalent to Z' by ignoring all the merges done around elements in $(\xi', \xi]$. This is impossible since one cannot produce an alternating sequence of p-length $\ell(\xi')$ using at most $(\lceil m/2 \rceil - \ell(\xi)) - 2 = (\lceil m/2 \rceil - \ell(\xi') + 1) - 2 = \lceil m/2 \rceil - \ell(\xi') - 1$ merge operations from Y. Hence, the claim is true.

Assume that the proposition is true for any number of merge operations ℓ, $\ell(\xi') - \ell(\xi) < \ell$. Now, consider the case when there is some threshold ξ'', $0 < \xi'' < \xi$, such that the sequence resulting from $Concatenate(Y, \xi'')$ is of p-length $\ell(\xi'') = \ell - \ell(\xi)$.

Notice that if $\ell(\xi') - \ell(\xi) = \ell - 1$, then $Algo(\xi')$ will, from the inductive assumption, perform $\ell(\xi') - \ell(\xi)$ merge operations around elements in $(\xi', \xi]$. At the same time, since $\ell(\xi'') - \ell(\xi') = (\ell - \ell(\xi)) - (\ell - \ell(\xi) - 1) = 1$, the algorithm $Algo(\xi')$ will also execute exactly one merge around some element in $(\xi'', \xi']$. Hence, the number of merge operations around elements in $(\xi'', \xi]$ by both $Algo(\xi)$ and $Algo(\xi')$ is $\ell(\xi') - \ell(\xi) + 1 = \ell(\xi'') - \ell(\xi)$. Observe again that the above reasoning works for any algorithms that produce alternating sequences of p-lengths $\ell(\xi)$, $\ell(\xi')$, and $\ell(\xi'')$ with respect to the given thresholds ξ, ξ', and ξ''. Therefore, there is an algorithm of comput-

ing *Concatenate*(Y, ξ) that performs $\ell(\xi'') - \ell(\xi)$ merges around elements in $(\xi'', \xi]$. By induction, the result follows. □

In order to establish the correctness of our algorithm, we will first investigate the recursive behavior of the optimal k-cover for alternating sequences. An element (or a block) y in the input sequence Y is *included* in an optimal k-cover $\mathcal{C}$ for the input if y is either a member of $\mathcal{C}$ or a segment of some member of $\mathcal{C}$. The following fact can be obtained directly from a nice relationship between optimal covers presented by Csűrös [11].

Lemma 1. *If $k < \lceil m/2 \rceil$ and y_i is an element with the smallest absolute value in Y, then either the entire segment $\langle y_{i-1}, y_i, y_{i+1} \rangle$ is included in an optimal k-cover of Y or none of y_{i-1}, y_i, or y_{i+1} is.*

Since y_i has the minimum absolute value, one can always perform a merge operation around it; call such a merge a *min-merge* operation. The $\Theta(m \log m)$-time algorithm for finding an optimal k-cover proposed by Csűrös [11] performs only min-merge operations. Our algorithm goes further by using general merge operations, which leads to an almost-linear-time solution. Actually, the elements included in the optimal k-cover computed by our algorithm are exactly the same as those selected by Csűrös' algorithm for being included in some optimal k-cover.

For a given alternating sequence Y of length m and an integer k, $k < \lceil m/2 \rceil$, the procedure to construct an optimal k-cover of Y by repeated min-merge operations [11] actually implies the following recursive constructions. For any integer q, $k \le q < \lceil m/2 \rceil$, an optimal q-cover of Y can be computed as follows: While the number of positive elements in the sequence is greater than q repeatedly join segments each of length 3 by performing min-merges. Call the sequence obtained Z_q and let M_q be the set of min-merge operations performed (ordered in the time at which the operation is executed). Then, an optimal k-cover of Y can be found with further min-merge operations starting from Z_q. We will show that the optimal q-cover computed by such a procedure is exact the same as the one resulted from our algorithm. More precisely,

Lemma 2. *Given an alternating sequence, Y, and two elements $y_i \in Y$ and $y_j \in Y$, the merges $\langle y_{i-1}, y_i, y_{i+1} \rangle$ and $\langle y_{j-1}, y_j, y_{j+1} \rangle$ can be performed in any order, if $j - i \ge 2$.*

Proof. We distinguish two cases: $j - i \ge 3$ and $j - i = 2$.

- $j - i \ge 3$: In this case, since $i + 1 < j - 1$, the two merges does not affect each other. The merge around y_i yields a new element $y'_i = y_{i-1} + y_i + y_{i+1}$ and the merge around y_j yields a new element $y'_j = y_{j-1} + y_j + y_{j+1}$ independently of each other.

- $j - i = 2$: In this case, $y_{i+1} = y_{j-1}$, so the merges are not completely independent of each other.

  First, consider the case when we first merge around y_i and then around y_j. The first merge (around y_i) will yield a new element $y'_i = y_{i-1}+y_i+y_{i+1}$. Now consider the merge around y_j. We notice that $y_{j-1} = y'_i$. Hence, the merge around y_j will yield a new element $y'_j = y'_i + y_j + y_{j+1} = y_{i-1} + y_i + y_{i+1} + y_j + y_{j+1}$.

  Now, consider the case when we first merge around y_j and then around y_i. The merge around y_j will yield a new element $y''_j = y_{j-1}+y_j+y_{j+1}$. Notice that $y_{i+1} = y''_j$. Hence, the merge around y_i will yield a new element $y''_i = y_{i-1} + y_i + y'_j = y_{i-1} + y_i + y_{j-1} + y_j + y_{j+1}$.

  Since $y_{i+1} = y_{j-1}$, we have that $y'_j = y''_i$, which proves the lemma.

□

Notice that, for the case when $j = i + 1$, in the above lemma, the order between merges *are* significant (the algorithm takes care of this case).

Lemma 3. *Let ℓ be the p-length of $Z = Concatenate(Y, \xi)$, where $\xi > 0$ is a given threshold. If $\ell < \lceil m/2 \rceil$ and an optimal ℓ-cover of Y is constructed from Y by repeated min-merge operations, then the sequence $Z_\ell = Concatenate(Y, M_\ell)$ is the same as the sequence $Z = Concatenate(Y, \xi)$.*

Proof. Notice that we will compare operands of the operations in M_ℓ and those used for achieving $Z = Concatenate(Y, \xi)$ by assuming that these two algorithms are virtually run, respectively, on its own copy of the input Y. In what follows, call $Concatenate(Y, \xi)$ *our algorithm*.

Let D_ξ be the partition set in computing $Z = Concatenate(Y, \xi)$; i.e., $D_\xi = \{y \in Y : |v(y)| \leq \xi\}$. The set D_ξ is dynamic under deletions. Observe that we implement such a set as a sequence and employ a double-linked list to run operations on it. Here, we ignore such details.

From Proposition 3, each merge (and thus min-merge) operation decreases the number of positive elements in the sequence by 1. Thus, the number $\|M_\ell\|$ of min-merge operations performed equals $\lceil m/2 \rceil - \ell$. That is, $\ell = \lceil m/2 \rceil - \|M_\ell\|$.

The proof of the lemma is done by an induction on $\|M_\ell\|$. We prove the result for general merges and min-merges. The case when the operation is 2-merge can be solved similarly.

1) Consider the case when $\|M_\ell\| = 1$. In this case, the only min-merge operation is around the element y which has the minimum absolute value in Y.

Since $\ell < \lceil m/2 \rceil$, we know that at least one merge have been executed by our algorithm in order to arrive at Z from Y. Such a merge must be done

around y; otherwise, say it is around $y' \neq y$. Hence, $y' \in D_\xi$ due to the fact that our algorithm performs a merge around y' only if $|v(y')| \leq \xi$. At the same time, we know that $y \in D_\xi$ since $|v(y)| < |v(y')|$. Therefore, our algorithm must perform a merge around y as well. Then, our algorithm would have performed at least two merges, which will results in an alternating sequence of p-length at most $\lceil m/2 \rceil - 2$ from Proposition 3. This contradicts the fact that $\ell = \lceil m/2 \rceil - \|M_\ell\| = \lceil m/2 \rceil - 1$. Hence, our algorithm do exactly the same operation as in M_ℓ and thus $Z_\ell = Z$.

2) Assume that the claim of the above lemma is true for $\|M_\ell\| < m'$. Now, consider the case when $\|M_\ell\| = m'$.

In the rest of this proof, we use the following expression for convenience: For a given pivot $\xi > 0$ and an alternating sequence Y of length m,

$Z(\xi, Y) = Concatenate(Y, \xi)$.

$\ell(\xi, Y)$: The p-length of $Z(\xi, Y)$; always assuming that $\ell(\xi, Y) < \lceil m/2 \rceil$.

$M(\xi, Y)$: The set of min-merge operations done during the procedure, of constructing an optimal $\ell(\xi, Y)$-cover of Y by performing repeatedly min-merge operations on Y, sorted in the increasing order of the time it is executed.

$Z(M(\xi, Y))$: The sequence obtained from Y by operating $M(\xi, Y)$ on Y.

Hence, the inductive assumption can also be stated as follows:

$$\text{For any threshold } \xi' > 0, \text{ if } \lceil m/2 \rceil - \ell(\xi', Y) < m', \text{ then}$$
$$Z(\xi', Y) \equiv Z(M(\xi', Y)).$$

What we want to prove in the inductive step is thus the following:

$$\text{For a given threshold } \xi > 0, \text{ if } \lceil m/2 \rceil - \ell(\xi, Y) = m', \text{ then}$$
$$Z(\xi, Y) \equiv Z(M(\xi, Y)).$$

Let ξ be the pivot such that our algorithm $Algo(Z(\xi, Y))$ produces the sequence $Z(\xi, Y)$ with $\lceil m/2 \rceil - \ell(\xi, Y) = m'$. For any real number ξ' $(< \xi)$ that would cause $\lceil m/2 \rceil - \ell(\xi', Y) = m' - 1$, we have $\ell(\xi', Y) = \ell(\xi, Y) + 1$.

From Proposition 4 we know that, by ignoring the only merge operation $merge(y', y, y'')$ ever done around some element y in $(\xi', \xi]$, the algorithm $Algo(Z(\xi, Y))$ will produce an alternating sequence Z' equivalent to the one constructed by $Algo(Z(\xi', Y))$. However, by the inductive assumption, $Algo(Z(\xi', Y))$ produced an alternating sequence equivalent to the sequence $Z(M(\xi', Y))$. Hence, Z' is the same sequence as $Z(M(\xi', Y))$. At the same time, the element y is now the one with the smallest absolute value in the sequence $Z(M(\xi', Y))$. Therefore, the operation $merge(y', y, y'')$ performed by the algorithm $Algo(Z(\xi, Y))$ can now be viewed as a min-merge operation run on Z'. Also, according to Lemma 2, the order in which $merge(y', y, y'')$

and some other merge, $merge(x', x, x'')$, is performed by $Algo(Z(\xi', Y))$ is insignificant as long as there is at least one element between y and x. This is ensured by the fact that $|y'| > |y|$ and $|y''| > |y|$, because if $|y| > \xi'$, then $|y'| > \xi'$ and $|y''| > \xi'$. Now, ξ' is chosen so that y is the only element larger than ξ' but smaller than ξ which implies that both y' and y'' is, in fact, larger than ξ and would be untouched by $Algo(Z(\xi', Y))$. Hence, we know that there is at least one element between y and any other element merged (namely y' or y''). This means that $Algo(Z(\xi, Y))$ will produce the same sequence as $Z(M(\xi, Y))$. The result follows. □

Notice that the optimal ℓ-cover of an alternating sequence of p-length ℓ is composed of all its positive elements. Then, we have that

Corollary 2. *Let ℓ be the p-length of $Z = Concatenate(Y, \xi)$, where $\xi > 0$ is a given real number. If $\ell < \lceil m/2 \rceil$, then the optimal ℓ-cover of Z corresponds to an optimal ℓ-cover of Y.*

Observe that our algorithm try to compute the p-length ℓ of the sequence resulted from the concatenation step for a given threshold ξ. We want to ensure that $k \leq \ell < \lceil m/2 \rceil$ and try to decrease the value of ℓ by a recursive computation of the threshold value. What we have done can be regarded as a binary search on the interval $[k, m/2]$. The correctness of our algorithm thus follows.

Theorem 1. *Let ℓ be the p-length of $Z = Concatenate(Y, \xi)$, where $\xi > 0$ is a given threshold and $\ell < \lceil m/2 \rceil$. If $k \leq \ell$, then there is an optimal k-cover of Z that corresponds to an optimal k-cover of Y.*

5.1 Complexity

Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of real numbers, the preprocessing step of our algorithm takes $O(n)$ time. After that, the iterations is run on the working sequences S_t for $t = 1, 2, \cdots$, where $S_1 = Y$ (the segmented version of X). Each iteration will decrease the length of the working sequence by a constant factor. In fact,

Lemma 4. $\|S_{t+1}\| \leq \frac{2}{3} \|S_t\|$ *for* $t = 1, 2, \cdots$.

Proof. Consider the i^{th} iteration. If we have merged too few segments during this iteration (i.e., the case of Step 2b), then our algorithm creates S_{t+1} from S_t by deleting all the elements $y \in S_t$ with $|v(y)| > \xi_{t-1}$ (the absolute value here corresponds to the new block after the merges been done; call this value $|v(y)|_{new}$). Observe that with the merges done in Step 2a, all the elements in S_t lie above ξ_t. In other words, $S_{t+1} = \{y \in S_t : \xi_t < |v(y)|_{new} \leq \xi_{t-1}\}$.

We claim that $\|S_{t+1}\| \leq \frac{2}{3} \|S_t\|$. Denote by $|v(y)|_{old}$ the absolute value of the element y in S_t before entering the concatenation step (i.e., before

performing merges in Step 2a). Observe that for any element y in S_t , we have $|v(y)|_{old} \leq \xi_{t-1}$. Let $U_t = \{y \in S_t : |v(y)|_{old} > \xi_t\}$. That is, $S_t = D_t \cup U_t$ when regarding them as sets.

Recall that the threshold ξ_{t-1} is the latest value corresponding to the case that we had merged too many segments. With respect to the current threshold ξ_t, we have in this iteration merged too few segments. Hence, $\|S_{t+1}\| > 0$ since otherwise we would have also done too much even during this iteration.

Before the concatenation starts, all the elements in $S_t = D_t \cup U_t$ are the candidates to be included in S_{t+1}. Hence, the number n_{t+1} of the candidates equals $\|S_t\|$, initially. Consider any element y in D_t and a merge *merge* (y', y, y'') (if exists) around y, let $z =$*merge*(y', y, y'').

- If $|v(z)|_{new} > \xi_{t-1}$, then $y \notin S_{t+1}$ and the number n_{t+1} is decreased by 1 for every element in S_t involved in the merge.
- If $|v(z)|_{new} \leq \xi_{t-1}$, then we know from Proposition 2 that $|v(y')|_{old} \leq \xi_{t-1}$ and $|v(y'')|_{old} \leq \xi_{t-1}$. That is, $y' \in S_t$ and $y'' \in S_t$. In this case, the number n_{t+1} is decreased by 2 depending for every three elements in S_t involved in the merge. For the case of 2-merges, the number n_{t+1} is decreased by 1 for every element in S_t involved in the merge (since such an element is no longer included in the optimal cover).

Therefore, the number of candidates to S_{t+1} is decreased by at least 2 for every three elements in D_t. Observe that the number of merges around elements D_t is bounded below by $\frac{1}{3}\|D_t\|$. We thus have that $\|S_{t+1}\| \leq \|S_t\| - \frac{2}{3}\|D_t\| = \|S_t\| - \frac{2}{3} \cdot \frac{1}{2}\|S_t\| = \frac{2}{3}\|S_t\|$.

On the other hand, if we have merged too many segments (i.e., the case of Step 2c), then our algorithm will undo all the merges done in Step 2a by assigning S_{t+1} to be the sequence D_t. Notice that $\|D_t\| = \frac{1}{2}\|S_t\|$. Hence, $\|S_{t+1}\| = \frac{1}{2}\|S_t\| < \frac{2}{3}\|S_t\|$. □

The time (except for that consumed by the disjoint-set data structure) needed for the t^{th} iteration of our algorithm is $O(\|S_t\|)$. Hence, the time complexity of our algorithm (excluding cost for union-finds) satisfies the recurrence $T(n) = T(\frac{2}{3}n) + O(n)$ and thus equals $O(n)$.

Moreover, the number of union-find operations performed during the t^{th} iteration is also $O(\|S_t\|)$. This implies that the total number of disjoint-set operations executed by our algorithm is $\sum_{t\geq 1} O(\|S_t\|) = O(\sum_{t\geq 1}\|S_t\|) = O(n)$. All these operations cost thus $O(n \cdot \alpha(n, n))$ in the worst case, where $\alpha(n, n)$ is the inverse Ackerman function. To sum up,

Theorem 2. *Given a sequence X of n real numbers and an integer $1 \leq k \leq n$, the problem of computing an optimal k-cover of the sequence can be done in $O(n \cdot \alpha(n, n))$ time in the worst case, where $\alpha(n, n)$ is the inverse Ackerman function.*

6 Conclusions

The problem of computing the maximum-scoring segments of a given sequence has been studied. We show how to solve the problem in $O(n\alpha(n, n))$ time in the worst case. Of course, a linear-time algorithm for this problem is desirable. Many algorithmic problems arising in the analysis of DNA sequences are of this flavor. Namely, one is interested in finding segments with constraints on the length of the segments and/or with different scoring functions. Both theoretical and practical efficient algorithms for these problems are interesting.

References

[1] I. E. Auger and C. E. Lawrence. Algorithms for the optimal identification of segment neighbourhoods. *Bulletin of Mathematical Biology*, 51(1):39–54, 1989.

[2] S. E. Bae and T. Takaoka. Algorithms for the problem of k maximum sums and a VLSI algorithm for the k maximum subarrays problem. In *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 247–253, 2004.

[3] S. E. Bae and T. Takaoka. Improved algorithms for the k-maximum subarray problem for small k. In *Proceedings of the 11th Annual International Conference on Computing and Combinatorics*, volume 3595 of *LNCS*, pages 621–631, 2005.

[4] T. R. Bement and M. S. Waterman. Locating maximum variance segments in sequential data. *Mathematical Geology*, 9(1):55–61, 1977.

[5] F. Bengtsson and J. Chen. Efficient algorithms for k maximum sums. In *Proceedings of the 15th Annual International Symposium Algorithms and Computation*, volume 3341 of *LNCS*, pages 137–148, 2004. Revised version to appear in Algorithmica.

[6] J. L. Bentley. Programming pearls: Algorithm design techniques. *Communications of the ACM*, 27:865–871, 1984.

[7] J. L. Bentley. Programming pearls: Perspective on performance. *Communications of the ACM*, 27:1087–1092, 1984.

[8] Anders Bergkvist and Peter Damaschke. Fast algorithms for finding disjoint subsequences with extremal densities. In *Proceedings of the 16th Annual International Symposium on Algorithms and Computation*, volume 3827 of *LNCS*, pages 714–723, 2005.

[9] Kai-Min Chung and Hsueh-I Lu. An optimal algorithm for the maximum-density segment problem. In *Proceedings of 11th Annual European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 136–147, 2003.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[11] Miklós Csűrös. Maximum-scoring segment sets. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(4):139–150, 2004.

[12] P. Fariselli, M. Finelli, D. Marchignoli, P.L. Martelli, I. Rossi, and R. Casadio. Maxsubseq: An algorithm for segment-length optimization. The case study of the transmembrane spanning segments. *Bioinformatics*, 19:500–505, 2003.

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

[14] T. C. Lin and D. T. Lee. Randomized algorithm for the sum selection problem. In *In Proceedings of the 16th Annual Internatinal Symposium on Algorithms and Computation*, volume 3827 of *LNCS*, pages 515–523, 2005.

[15] Walter L. Ruzzo and Martin Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Proceedings of the 7th Annual International Conference on Intelligent Systems for Molecular Biology*, pages 234–241, 1999.

[16] D.R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8:213–229, 1987.