2007:03

RESEARCH REPORT

Computing Maximum-Scoring Segments Optimally

Fredrik Bengtsson
JingsenChen

Luleå University of Technology
Department of Computer Science and Electrical Engineering

2007:03 | ISSN: 1402-1528 | ISRN: LTU-FR -- 07/03 -- SE

Computing Maximum-Scoring Segments Optimally

Fredrik Bengtsson
Jingsen Chen

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

Phone: +46 920 492431
Fax: +46 920 493111
e-mail: bson@sm.luth.se

Abstract

Given a sequence of length n, the problem studied in this paper is to find a set of k disjoint subsequences of consecutive elements such that the total sum of all elements in the set is maximized. This problem arises in the analysis of DNA sequences. The previous best known algorithm requires $\Theta(n\alpha(n,n))$ time in the worst case, where $\alpha(n,n)$ is the inverse Ackermann function. We present a linear-time algorithm, which is optimal, for this problem.

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 [12, 13, 14]. 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 a 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 problem can be solved faster. Previous algorithms requires $\Theta(n \log n)$ time [12] and $\Theta(n\alpha(n, n))$ time [5, 6]. In this paper, we show how the algorithmic ideas presented in [5] leads to an optimal worst-case $O(n)$-time algorithm.

The problem studied can be viewed as a generalization of the classical *maximum sum subsequence problem* introduced by Bentley [8]. The latter is to find the continuous segment with largest sum of a given sequence and can be solved in linear time [9, 17]. 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, 7, 15]. Another generalization arising from bioinformatics is to look for an interesting segment (or segments) of constrained length [10, 11, 16].

The paper is organized as follows. In Section 2 the problem is defined. Section 3 gives an overview of the algorithmic ideas in [5]. The refined algorithm and its linear-time implementation are presented in Section 4. The analysis of our algorithm appears in Section 5.

2 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* (or *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.*

3 The Algorithm

By refining the algorithmic ideas proposed by Bengtsson et al. [5] we redesign that algorithm and achieve an optimal linear-time solution. We are able to avoid the union-find data structure that was employed in the previous algorithm [5]. This yields optimal performance.

The algorithm in [5] 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.

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 sequences needs no further computation.

Observation 1. *[5] 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 that follows.

The problem becomes trivial when $k \geq m$ (the number of positive runs of the input sequence). For the case when $k < m$, the following segmentation of the input is performed:

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.
3. Negative elements at both ends of Y (if any) are removed from Y.

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, if $k < m$. For details, se [6].

3.2 Partitioning and Concatenating

The other two phases presented in [5] for 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, a shorter a-sequence Y' from Y is first constructed. Such a construction is accomplished with concatenation operations run on an odd number of consecutive elements of Y [5]. By partitioning Y into smaller pieces and concatenating carefully chosen segments, the number of candidate segments for the optimal k-cover is decreased fast. If the length of Y' equals $2k-1$, one has the solution; otherwise, the construction is repeated starting from Y'. Such a procedure is 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 the 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 one has the solution.

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. Notice all the sequences Y_t $(t = 1, 2, \cdots)$ are not actually constructed; otherwise the time complexity would be $\Omega(m \log m)$. The idea is to operate directly on Y all the time, without actually constructing Y_t.

4 Linear-Time Algorithm

In this section we present our linear-time refinement of the algorithm in [5]. It is not obvious how to achieve linear time by using some simple data structures. The crucial point here is how to compute the block resulting from a series of concatenation of blocks (called merges) in the desired time bound.

The previous best algorithm [5] used a union-find data structure for finding the currently resulting block, for any given block in D or S. This was necessary in order to be able to compute the blocks resulting after a series of merges. Since there was a `find` operation performed for each block during computation, a performance of $O(n\alpha(n, n))$ was achieved.

In this paper we are able to bypass need for the union-find data structure by utilizing linear-time sorting on arrays containing *only* the blocks that participated in a concatenation step. We observe that, by sorting the elements involved in a concatenation step on their left (low) index, it is possible to extract the resulting blocks from a concatenation step faster than with find operations on the union-find data structure. The worst-case linear time stems from the fact that we sort the blocks using the indexes as keys and hence, the universe for the sorting procedure is linear in the number of elements sorted.

During the operation of our algorithm, we maintain the following data structures:

- A linked list L, containing the currently existing segments. This list will be shorter, as segments are merged with its neighbours. Initially, this list contains the same elements as Y.
- A linked list D_t, containing, before each merge step, the blocks to be merged. This list is empty when the merge step is completed.
- A linked list, M, containing one element for each merge performed by the merge step. Each element is a triple (v, i, j), where (i, j) is the start- and end-index of the resulting merged block, respectively, and v is the value of the block.

4.1 Partitioning and Concatenation

In the partitioning step, the data is partitioned into blocks that are to be merged and blocks that are not.

- Partitioning
  - Let ξ_t be the r^{th} largest element of S_t.
  - Let $D_t = \{x | x \in S_t \wedge x \leq \xi_t\}$. Since D_t is a linked list, preserve the ordering among elements from S_t.
- Concatenation
  - Merge all elements in D_t. During merge, save all modifications to all data structures.

The concatenation step merges all elements contain in the list D_t with its respective neighbouring elements in L. This is performed as follows. For each $(v, i, j) \in D_t$ (as long as D_t is not empty):

- Let $(v_{\text{left}}, i_{\text{left}}, j_{\text{left}})$ and $(v_{\text{right}}, i_{\text{right}}, j_{\text{right}})$ be the left and right neighbour of (v, i, j) in L, respectively.
- Let $(v', i', j') = (v_{\text{left}} + v + v_{\text{right}}, i_{\text{left}}, j_{\text{right}})$ be the new block created by the merge.
- In L, replace all three blocks $(v_{\text{left}}, i_{\text{left}}, j_{\text{left}})$, (v, i, j), and $(v_{\text{right}}, i_{\text{right}}, j_{\text{right}})$ with (v', i', j').
- In D_t, if the left neighbour of (v, i, j) is $(v_{\text{left}}, i_{\text{left}}, j_{\text{left}})$, then delete $(v_{\text{left}}, i_{\text{left}}, j_{\text{left}})$ from D_t.

- In D_t, if the right neighbour of (v, i, j) is $(v_{\text{right}}, i_{\text{right}}, j_{\text{right}})$, then delete $(v_{\text{right}}, i_{\text{right}}, j_{\text{right}})$ from D_t.
- In D_t, delete (v, i, j).

4.2 Iterating

Now, based upon the number of blocks resulting after the concatenation step, we either merge more blocks, or we cancel the concatenation step and merge fewer blocks instead. Let k' be the number of positive blocks in L after the merge step. We have two cases.

4.2.1 Too Few Blocks

If $k < k'$, then we have essentially performed too many merges in the latest merge step. In such case, we cancel the latest concatenation step altogether. This is possible if all updates performed during the latest concatenation step are saved. Thus, we save all updates performed on all data structures during the latest concatenation step. After the concatenation step is performed and there are not too many resulting blocks, we can throw away the data saved before. Hence, there is only need for saving the updates performed in a single concatenation step.

Now, let $S_{t+1} \leftarrow D_t$ and $\xi_t \leftarrow \xi_{t-1}$ (we cancel the effect of updating ξ). Let $t \leftarrow t + 1$ and perform the partition step (Section 4.1) again with D_t computed from this new S_{t+1}.

4.2.2 Too Many Blocks

If, on the other hand, the number of positive blocks (in L) after the concatenation step are still to many, we use the procedure of block filtering to filter out the blocks resulting from the previous concatenation step. This will be the new list S_{t+1}.

4.3 Block Filtering

In the following, we will present, the procedure to find the blocks after merging all the elements in D_t. The difficulty of this task lies in the requirement to find *only* the *resulting* blocks and not all blocks that have occurred during the merges. Notice that after a merge, it might be the case that the $v' \leq \xi_t$. In such a case the block (v', i', j') should still be in D_t and merged further. For instance, an element of D_t that is to be merged with its neighbouring elements in L results in one new element in L and one new element in D_t, if the resulting block is still smaller than ξ_t. Hence the intermediate block, (v', i', j'), is not interesting as a result of our block filtering procedure. We compute only the final blocks greater than ξ_t.

To resolve this issue, we use another linked list, M and, for each merge, insert a tuple (v, i, j), with the range of the resulting block $[i, j]$ with respect to Y, and the value of the block. For each tuple, we also maintain a reference to the corresponding block in L. Now, when all merges are done, we copy M to a sequence (an array) M'. After this, we sort the elements of M' with respect to index i by using counting sort, which takes linear time since the keys i used in sorting are integers from 1 to n. Call the sorted sequence M''. Observe that we can write M'' as follows.

$$\begin{aligned}M'' =&\langle (v_{(1,1)}, i_1, j_{(1,1)}), (v_{(1,2)}, i_1, j_{(1,2)}), \ldots, (v_{(1,p_1)}, i_1, j_{(1,p_1)}),\\ &(v_{(2,1)}, i_2, j_{(2,1)}), (v_{(2,2)}, i_2, j_{(2,2)}), \ldots, (v_{(2,p_2)}, i_2, j_{(2,p_2)}),\\ &\vdots\\ &(v_{(q,1)}, i_q, j_{(q,1)}), (v_{(q,2)}, i_q, j_{(q,2)}), \ldots, (v_{(q,p_q)}, i_q, j_{(q,p_q)})\rangle\end{aligned}$$

where $i_1 < i_2 < \cdots < i_q$. For each i_ℓ, $\ell \in [1, q]$, let $(v_\ell, , i_\ell, j_\ell) = \max_{1 \le j \le p_q}(v_{(\ell,j)}, i_\ell, j)$. That is, we find the largest j-index for each i_ℓ. Now, the tuple (v_ℓ, i_ℓ, j_ℓ) for $\ell \in [1, q]$ is the new blocks created by the merge procedure.

5 Analysis

The correctness of the above algorithm follows directly from the results in [5, 6]. Our refined method and its implementation require linear time in the worst case. In fact, 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 have been run on the working sequences S_t for $t = 1, 2, \cdots$, where $S_1 = Y$ (the segmented version of X). The time needed for the t^{th} iteration (in particular, the partitioning and concatenating procedure) is $O(\|S_t\|)$. Notice from the design of our algorithm that $\|S_{t+1}\| \le \frac{2}{3}\|S_t\|$ and $\|S_1\| = \|Y\| = m \le n$. Hence, the time complexity of our algorithm satisfies the recurrence $T(n) = T\left(\frac{2}{3}n\right) + O(n)$ and thus equals $O(n)$. To sum up,

Theorem 1. *Given a sequence X of n real numbers and an integer $1 \le k \le n$, the problem of computing an optimal k-cover of the sequence can be done in $O(n)$ time in the worst case, which is optimal.*

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. Computing maximum-scoring segments in almost linear time. In *Proceedings of the 12th Annual International Computing and Combinatorics Conference*, volume 4112 of *LNCS*, pages 255–264, 2006.

[6] F. Bengtsson and J. Chen. Computing maximum-scoring segments in almost linear time. Technical Report 2006:14, Department of Computer Science and Electrical Engineering, LuleåUniversity of Technology, Sweden, 2006. http://epubl.ltu.se/1402-1528/2006/14/.

[7] F. Bengtsson and J. Chen. Efficient algorithms for k maximum sums. *Algorithmica*, 46(1):27–41, 2006.

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

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

[10] A. Bergkvist and P. 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.

[11] K.M. Chung and H.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.

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

[13] 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.

[14] 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.

[15] 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.

[16] W. L. Ruzzo and M. 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.

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