Data Mining and Knowledge Discovery manuscript No.
(will be inserted by the editor)

Fast Sequence Segmentation using Log-Linear Models

Nikolaj Tatti

the date of receipt and acceptance should be inserted later

Abstract Sequence segmentation is a well-studied problem, where given a sequence of elements, an integer K, and some measure of homogeneity, the task is to split the sequence into K contiguous segments that are maximally homogeneous. A classic approach to find the optimal solution is by using a dynamic program. Unfortunately, the execution time of this program is quadratic with respect to the length of the input sequence. This makes the algorithm slow for a sequence of non-trivial length. In this paper we study segmentations whose measure of goodness is based on log-linear models, a rich family that contains many of the standard distributions. We present a theoretical result allowing us to prune many suboptimal segmentations. Using this result, we modify the standard dynamic program for one-dimensional log-linear models, and by doing so reduce the computational time. We demonstrate empirically, that this approach can significantly reduce the computational burden of finding the optimal segmentation.

Keywords segmentation, pruning, change-point detection, dynamic program

1 Introduction

Sequence segmentation is a well-studied problem, where given a sequence of elements, an integer K, and some measure of homogeneity, the task is to split the sequence into K contiguous segments that are maximally homogeneous.

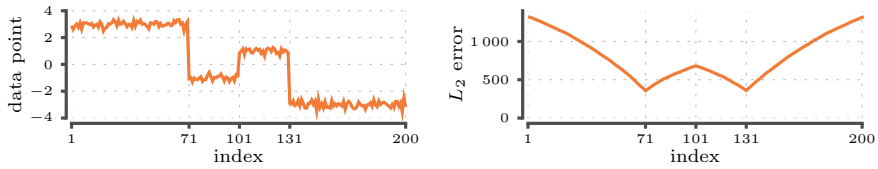
An exact solution for segmentation with K segments can be obtained by a classic dynamic program in $O(L^2K)$ time, where L is the length of the sequence (Bellman, 1961). Due to the quadratic complexity, we cannot apply

Nikolaj Tatti
Department of Mathematics and Computer Science, University of Antwerp, Antwerp,
Department of Computer Science, Katholieke Universiteit Leuven, Leuven,
Belgium
E-mail: nikolaj.tatti@gmail.com

segmentation for sequences of non-trivial length. In this paper we introduce a speedup to the dynamic program used for solving the exact solution. Our key result, given in Theorem 1, states that when certain conditions are met, we can discard the candidate for a segment border, thus speeding up the inner loop of the dynamic program.

We consider segmentation using the log-likelihood of a log-linear model to score the goodness of individual segments. Many standard distributions can be described as log-linear models, including Bernoulli, Gamma, Poisson, and Gaussian distributions. Moreover, when using a Gaussian distribution, optimizing the log-likelihood is equal to the minimizing the L_2 error (see Example 1).

The conditions given in Theorem 1 are hard to verify, however, we demonstrate that this can be done with relative ease for one-dimensional models. The key idea is as follows: Consider segmenting the sequence given in Figure 1(a) into 2 segments using the L_2 error. Assume a segmentation $[1, 100], [101, 200]$. Figure 1(b) tells us that this segmentation is not optimal. In fact, the optimal segmentation with 2 segments for this data is $[1, 70], [71, 200]$.



(a) Sequence (b) Cost of segmentation

Fig. 1 Toy sequence and the L_2 cost of a segmentation $[1, k-1]$, $[k, 200]$ as a function of k. In this paper we propose a necessary condition for a segmentation to be optimal. This condition allows us to prune suboptimal segmentations, such as $[1, 100], [101, 200]$

Sequence values around 101 have a particular characteristic which we can exploit to speedup the optimization. In order to demonstrate this, let us define

$$X = \left\{ \frac{1}{101 - j} \sum_{i=j}^{100} D_i \mid 1 \leq j \leq 100 \right\} \quad \text{and}$$

$$Y = \left\{ \frac{1}{j - 100} \sum_{i=101}^{j} D_i \mid 101 \leq j \leq 200 \right\},$$

that is, X contains the averages from the right side of the first segment and Y contains the averages from the left side of the second segment. Let us define $r_1 = \min X$, $r_2 = \max X$, $l_1 = \min Y$, $l_2 = \max Y$. We see that $r_1 \approx -1$, $r_2 \approx 1.8$, $l_1 \approx -1.8$, and $l_2 \approx 1$. That is, the intervals $[r_1, r_2]$ and $[l_1, l_2]$ intersect. We will show in such case that not only we can safely ignore the segmentation $[1, 100], [101, 200]$ but we also will show that even if we augment

the sequence with additional data points, index 101 will never be part of the optimal segmentation with 2 segments. This pruning allows us to speedup the dynamic programming.

In general, if the extreme values of averages $[r_1, r_2]$ and $[l_1, l_2]$ computed from neighboring segments intersect, we know that the segmentation is suboptimal. On the other hand, the optimal segmentation with 4 segments for data in Figure 1(a), $[1, 70], [71, 100], [101, 130], [131, 200]$, uses index 101. We do not violate our condition since the extreme values of averages $[r_1, r_2]$ computed *only* from the second segment and extreme values of averages $[l_1, l_2]$ computed *only* from the third segment no longer intersect.

Using this idea, we will build an efficient pruning technique for segmenting data using one-dimensional log-linear models. We empirically demonstrate that this approach can reduce the computational load by several orders of magnitude compared to the standard approach.

The remaining paper is organized as follows. In Section 2 we give preliminary notation and define the segmentation problem. In Section 3 we give the key result which allows us to prune segments. Sections 4–5 are devoted to a segmentation algorithm. We present our experiments in Section 6 and related work in Section 7. Finally, we conclude the paper with discussion in Section 8.

2 Segmentation for log-linear models

In this section we give preliminaries and define the segmentation problem.

A *sequence* $D = (D_1, \ldots, D_L)$ is a sequence of real vectors of length M, $D_i \in \mathbb{R}^M$. A *segment* $I = [b, e]$ consists of two integers such that $1 \leq b \leq e \leq L$. We will write $k \in I$ whenever $b \leq k \leq e$ for an integer k. We define $D[b, e] = (D_b, \ldots, D_e)$ to be the subsequence corresponding to that segment. A *segmentation* P is a list of disjoint segments that cover D, that is, $P = (I_1, \ldots, I_K)$ such that the first segment I_1 starts at 1, the last segment I_K ends at $|D|$ and $I_k = [a, b]$ begins right after $I_{k-1} = [c, d]$, that is $a = d + 1$.

Our goal is to find a segmentation that maximizes the likelihood of a log-linear model of each individual segment. By log-linear models, also known as exponential family, we mean models whose probability density function can we written as

$$p(x \mid r) = q(x) \exp\big(Z(r) + r^T S(x)\big)\,,$$

where $S : \mathbb{R}^M \to \mathbb{R}^N$ is a function mapping x to a vector in $\mathbb{R}^N$, $r \in \mathbb{R}^N$ is the parameter vector of the model, and $Z(r)$ is the normalization constant. Many standard distributions are log-linear, for example, Poisson, Gamma, Bernoulli, Binomial, and Gaussian (both with fixed or unknown variance). We will argue later in this section that using a Gaussian distribution with a fixed variance is equivalent to minimizing L_2 error.

Assume that we are given a segmentation P and for each segment $I \in P$, we have a parameter vector r_I. Let us now consider the log-likelihood

$$\log \prod_{I \in P} \prod_{k \in I} p(D_k \mid r_I) = \sum_{I \in P} \sum_{k \in I} \log q(D_k) + Z(r_I) + r_I^T S(D_k)$$
$$= \sum_{k=1}^{|D|} \log q(D_k) + \sum_{I \in P} \sum_{k \in I} Z(r_I) + r_I^T S(D_k),$$

for this segmentation of D. Note that the first term in the right-hand side does not depend on the parameters nor on the segmentation. Consequently, we can ignore it. In addition, note that we can safely assume that $S(x) = x$. If this is not the case, we can always transform sequence D into $D' = (S(D_1), \ldots, S(D_L))$. From now on we will assume that $S(x) = x$.

For notational simplicity, let us define

$$c(D) = \sum_{i=1}^{|D|} D_i \quad \text{and} \quad av(D) = \frac{c(D)}{|D|}$$

to be the sum and the average of data points in D. If D is clear from the context, we will often write $c(i,j)$ and $av(i,j)$ to mean $c(D[i,j])$ and $av(D[i,j])$. As shorthand, we write $av(j)$ and $c(j)$ to mean $c(1,j)$ and $av(1,j)$.

We define the score of a single segment given a parameter vector as

$$sc(D \mid r) = |D| Z(r) + r^T c(D) \quad .$$

We define the score for a segmentation P as

$$sc(P; D) = \sum_{I \in P} sc(D[I]), \text{ where } sc(D) = \sup_r sc(D \mid r),$$

that is, $sc(P; D)$ is a sum of the optimal scores of individual segments. We see that optimizing $sc(P; D)$ is equivalent to maximizing likelihood of the log-linear model.

We are now ready to state our optimization problem.

Problem 1 Given a sequence D, a log-linear model, and an integer K, find a segmentation P with K segments maximizing $sc(P; D)$.

Example 1 Let us now consider a Gaussian distribution with identity covariance matrix. This distribution is log-linear since we can rewrite

$$(2\pi)^{-M/2} e^{-0.5\|x-\mu\|^2} \quad \text{as} \quad e^{-0.5\|x\|^2} (2\pi)^{-M/2} e^{-0.5\|\mu\|^2 + \mu^T x} \quad .$$

The log-likelihood of a Gaussian distribution for a segmentation P is

$$\log \prod_{I \in P} \prod_{x \in D[I]} (2\pi)^{-M/2} e^{-0.5\|x-\mu_I\|^2}$$
$$= \sum_{I \in P} \sum_{x \in D[I]} -M/2 \log 2\pi - 0.5 \|x - \mu_I\|^2$$
$$= -|D| M/2 \log 2\pi - 0.5 \sum_{I \in P} \sum_{x \in D[I]} \|x - \mu_I\|^2 \quad .$$

The optimal value for μ_I is an average of data points in $D[I]$. The first term of the right-hand side is constant while the second term is the L_2 error. Consequently, selecting a segmentation that maximizes log-likelihood is equivalent to finding a segmentation that minimizes the L_2 error, a typical choice for an error function.

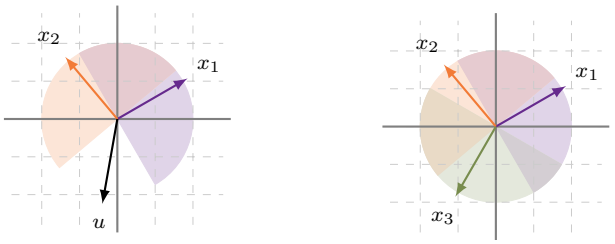
The optimal segmentation can be found with a dynamic program (Bellman, 1961). In order to see this, let $P = (I_1, \ldots, I_K)$ be the optimal segmentation with K segments. Let c be the last index of I_{K-1}. Then $(I_1, \ldots, I_{K-1})$ is the optimal segmentation for $D[1, c]$. We can find the optimal segmentation with K segments by first computing the optimal segmentation with $K-1$ segments for each $D[1, c]$ and then testing which segment of form $(c, |D|)$ we need to add to produce the optimal segmentation with K segments. This leads to an algorithm of time complexity $O(K|D|^2)$. The goal of this paper is to provide an optimization of this dynamic program.

3 Necessary Condition for Optimal Segmentation

In this section we give a key result of this paper. This result allows us to prune candidates that will not be included in the optimal segmentation and hence speedup the dynamic program.

In order to do so, let V be a set of vectors in $\mathbb{R}^N$. We say that V is a *cover* if for any $y \in \mathbb{R}^N$, there is a $v \in V$ such that $y^T v \geq 0$. See Figure 2 for an example. Given two sequences D and E we define $\mathit{diff}(D, E)$ to be the *difference set* for D and E as

$$\mathit{diff}(D, E) = \{av(D[k, |D|]) - av(E[1, l]) \mid 1 \leq k \leq |D|, 1 \leq l \leq |E|\} \quad .$$



(a) non-cover (b) cover

Fig. 2 An example of a non-cover and a cover. In Figure 2(a) $\{x_1, x_2\}$ is not a cover since u is outside the half-planes induced by x_1 and x_2. In Figure 2(b) $\{x_1, x_2, x_3\}$ is a cover

We are now ready to state the key result of the paper. For readability, we postpone the proof to Appendix A.1.

Theorem 1 *Let P be a segmentation. There is a segmentation P' such that $sc(P') \geq sc(P)$ and $diff(D[I], D[J])$ is not a cover for any two consecutive segments I and J in P'.*

4 Segmentation for one-dimensional models

In the previous section we saw a necessary condition for optimal segmentation. This involves checking whether the difference set of consecutive segments is a cover. In this section and the next section we show that we can efficiently check this condition if our linear model is one-dimensional, that is, if data points D_i are real numbers.

In order to show this, let D be a sequence. We define a *left interval* to be an interval

$$int_L(D) = \left(\min_{1 \leq i \leq |D|} av(1,i) \, , \max_{1 \leq i \leq |D|} av(1,i) \right)$$

of extreme values of $av(1,i)$. Similarly, we define a *right interval* to be

$$int_R(D) = \left(\min_{1 \leq i \leq |D|} av(i,|D|) \, , \max_{1 \leq i \leq |D|} av(i,|D|) \right) \quad .$$

We can now express the condition using these intervals.

Theorem 2 *Assume two sequences, D and E and let S be a one-dimensional statistic. Then $diff(D,E)$ is a cover if and only if the intervals $int_R(D)$ and $int_L(E)$ intersect.*

Proof Let $int_R(D) = (x,y)$ and $int_L(E) = (u,v)$. $diff(D,E)$ is a cover if and only if there are a, b, c, and d such that $av(D[a,|D|]) \leq av(E[1,b])$ and $av(D[c,|D|]) \geq av(E[1,d])$. This is equivalent to $x \leq v$ and $y \geq u$, which is equivalent to $int_R(D)$ and $int_L(E)$ intersecting. □

We can now use this result to design an efficient algorithm. Assume that we already have computed for each j the optimal segmentation with $K-1$ segments, say P_j covering $D[1,j]$. We now want to find an optimal segmentation with K segments covering $D[1,i]$. In order to do so we need to augment each P_{j-1} for $j \leq i$ with a segment $[j,i]$, and pick the optimal segmentation. Assume that the intervals $int_L(D[j,i])$ and $int_R(D[c,j-1])$ intersect, where c is the starting point of the last segment in P_{j-1}. Then Theorems 1 and 2 imply that we can safely ignore the segmentation P_{j-1} augmented with $[j,i]$. Moreover, if the intervals intersect when segmenting $D[1,i]$, they will also intersect when segmenting $D[1,k]$ for $k > i$. Hence, as soon as $int_L(D[j,i])$ and $int_R(D[c,j-1])$ intersect, we can ignore j as a candidate for the starting point of the last segment. We present the pseudo-code for this approach as Algorithm 1.

Let us next analyze the time and memory complexity of Algorithm 1. Let L be the maximal size of C. It is easy to see that we can compute $sc(D[j,i])$ and $int_L(D[j,i])$ in constant time by keeping and updating the sum $c(j,i)$ for every

Algorithm 1: SEGMENT(s, r, D) builds the optimal K-segmentation for D using optimal segmentations with $K-1$ segments

```
input  : scores s for optimal segmentation with K − 1 segments, corresponding
         right intervals r, sequence D
output : scores u for optimal K-segmentation, right intervals v for optimal
         K-segmentation
C ← ∅;
foreach i = 1, ..., |D| do
    add i to C;
    foreach j ∈ C do
        update l(j) to be int_L(D[j, i]);
        if r(j − 1) and l(j) intersect then
            delete j from C;
    c ← arg max_{j∈C} s(j − 1) + sc(D[j, i]);
    u(i) ← s(c − 1) + sc(D[c, i]);
    v(i) ← int_R(D[c, i]);
return u, v;
```

$j \in C$. The only non-trivial part of Algorithm 1 is computing the right interval $int_R(D[c, i])$. In the next section, we will show how to compute the right interval in amortized $O(L)$ time, hence the execution time of the algorithm is in $O(L|D|)$. Moreover, we will show that the total memory requirement for computing the right interval is in $O(|D|)$ which will make the memory usage of the algorithm $O(|D|)$.

5 Computing the Right Interval

In this section we show how to compute the right interval, as needed in Algorithm 1. We will focus on how to compute the maximal value of the right interval; we can compute the minimal value using exactly the same framework.

5.1 Computing the Borders

Our first goal, given a sequence D and integer i, is to find j such that $av(j, i)$ is maximal. Naturally, if we have to do so from scratch, we have no other option but to test every $1 \leq j \leq i$. However, since segmentation needs the maximal average for every i we can use information from previous scans to find the optimal j more quickly.

We will now present the main results by Calders et al (2007) in which the authors considered efficiently finding the maximal average from a stream of data points. In the next section we will modify this approach to make it more memory-efficient.

Given a sequence D we say that $1 \leq i \leq |D|$ is a *border* if there is a (possibly empty) sequence E such that if we define F to be D concatenated

with E, then

$$av(F[i,|F|]) = \max_{1 \leq j \leq |F|} av(F[j,|F|]) \quad .$$

We define $borders(D)$ to be the sorted list of border points of D.

Let D be a sequence. Further, Let $1 \leq i \leq j \leq |D|$ and let $(b_1, \ldots, b_M) = borders(D[i,j])$. Whenever D is clear from the context, we define $borders(i,j)$ to be $(b_1 + i - 1, \ldots, b_M + i - 1)$. Further, we will write $borders(i)$ instead of $borders(i,|D|)$.

The following theorem states that a maximal average can be found by simply taking the largest border.[1]

Theorem 3 (see Calders et al, 2007) *Assume a sequence D. Let $j = \max borders(D)$. Then*

$$av(j,|D|) = \max_{1 \leq k \leq |D|} av(k,|D|) \quad .$$

We can describe the borders using the following theorem.

Theorem 4 (see Calders et al, 2007) *An integer i is a border for D if and only if there are no a and b, $a < i \leq b$ such that $av(a, i-1) \geq av(i,b)$.*

Example 2 Assume that we are given a sequence $D = (2,0,1,2,1,1,9,2,5,0)$, and that $S(x) = x$. According to Theorem 4, index $3 \notin borders(D)$, since $av(1,2) = 1 = av(3,3)$. The borders are $(1,4,7) = borders(D)$.

Our next step is to revise the algorithm given by Calders et al (2007) for constructing $borders(1,i)$ from $borders(1,i-1)$. The key idea for update is given in the following theorem.

Theorem 5 (see Calders et al, 2007) *Let us assume a list of borders $(b_1, \ldots, b_M) = borders(1, i-1)$. Define $b_{M+1} = i$. Define N, $2 \leq N \leq M+1$, to be the maximal integer such that $av(b_{N-1}, i) < av(b_N, i)$. If such N does not exist, we set $N = 1$. Then, $(b_1, \ldots, b_N) = borders(1,i)$.*

The update algorithm (given as Algorithm 2) starts with the previous borders $(b_1, \ldots, b_M) = borders(1, i-1)$ and adds i as a border. Then the algorithm tests whether the average of the second last border is larger than the average of the last border. If so, then the condition in Theorem 5 is violated, and we remove the last border and repeat the test. The correctness of UPDATE is given by Calders et al (2007). Note that we can compute the needed averages in constant time, for example, by precalculating a sequence $(c(1), \ldots, c(|D|))$.

[1] Calders et al (2007) deal only with binary sequences but we can easily extend these results to the general case.

Algorithm 2: UPDATE, updates $borders(1, i-1)$ to $borders(1, i)$

input : borders $(b_1, \ldots, b_M) = borders(1, i-1)$
output : updated borders $borders(1, i)$

$b_{M+1} \leftarrow i$;
$M \leftarrow M + 1$;
while $M > 1$ **and** $av(b_{M-1}, i) \geq av(b_M, i)$ **do** $M \leftarrow M - 1$;
return $(b_1, \ldots, b_M)$;

5.2 Computing Borders Simultaneously

We can use borders to discover the right interval for a single segment. However, recall that in Algorithm 1 we need to be able to compute the right interval for any $D[c, i]$, where $c \in C$ is the current set of candidates for a segment. A naïve approach would be to compute borders separately for each $D[c, i]$. This leads to $O(L|D|)$ memory and time consumption, where L is the maximum size of C during evaluation of SEGMENT. Here, we will modify the border update algorithm such that its total memory consumption is $O(|D|)$. This will guarantee that the memory consumption of SEGMENT is $O(|D|)$.

Example 3 Let us continue Example 2. We have $borders(1, 10) = (1, 4, 7)$, $borders(3, 10) = (3, 4, 7)$, $borders(8, 10) = (8, 9)$, and $borders(10, 10) = (10)$. Note that $borders(1, 10)$ and $borders(3, 10)$ have a common tail sequence, namely, $(4, 7)$. We can generalize this observation.

The following key result states that when two border lists, say $borders(i)$ and $borders(j)$ share a common border, the subsequent borders are equivalent.

Theorem 6 *Let D be a sequence and let $1 \leq i, j \leq |D|$ be two indices. Assume that $a \in borders(i) \cap borders(j)$. Let $b \geq a$. Then $b \in borders(i)$ if and only if $b \in borders(j)$.*

Proof We will show that $\{b \in borders(i, k) \mid b \geq a\} = \{b \in borders(j, k) \mid b \geq a\}$ using induction over k. The result follows by setting $k = |D|$.

Since $\{b \in borders(i, a) \mid b \geq a\} = \{a\} = \{b \in borders(j, a) \mid b \geq a\}$, the first $k = a$ step follows.

Assume that the result holds for $k-1$. Let $(b_1, \ldots, b_M) = borders(i, k-1)$ and $(c_1, \ldots, c_K) = borders(j, k-1)$. Let also $b_{M+1} = c_{K+1} = k$. Let x and y be such that $b_x = a = c_y$.

Theorem 5 states that there is an integer N such that $(b_1, \ldots, b_N) = borders(i, k)$ and an integer L such that $(c_1, \ldots, c_L) = borders(j, k)$. Since $a \in borders(i)$, we must have $N \geq x$ and similarly $L \geq y$. Note that, by the induction assumption, we have $(b_x, \ldots, b_{M+1}) = (c_y, \ldots, c_{K+1})$. This implies that UPDATE will process exactly the same input, and deletes exactly the same number of entries, that is, implies that $M - y \leq N - x$. This proves the induction step. □

This theorem allows us to group border lists into a tree. Let D be a sequence and let C be a set of indices. We define a border tree $T = btree(D, C)$ as follows:

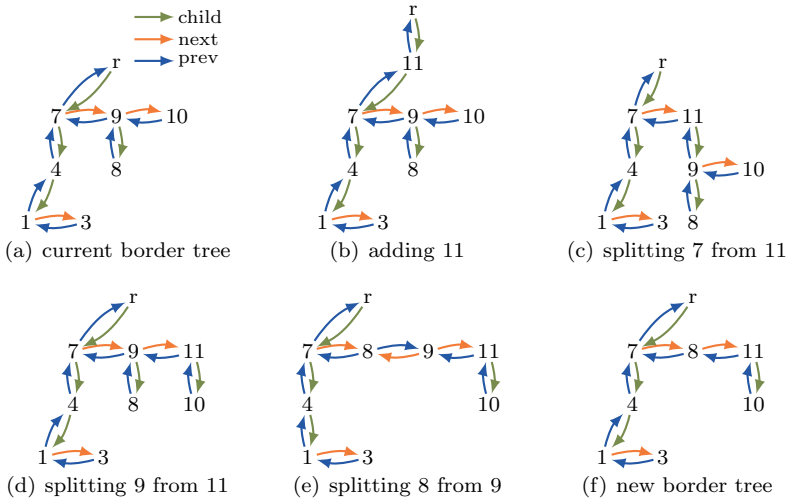


Fig. 3 Border trees related to Example 4, demonstrating how these trees are updated. Figure 3(a) is given as input to UpdateTree. First, UpdateTree adds a new node to the tree, shown in Figure 3(b), then proceeds to prune obsolete borders, resulting in a new border tree, given in Figure 3(f)

The non-root nodes of the tree consists of the borders from $borders(c)$ for each $c \in C$, that is,

$$V(T) = \{b \mid b \in borders(c) \text{ for some } c \in C\} \quad .$$

There is an edge from a node m to a node n if and only if there is $c \in C$ such that $(b_1, \ldots, b_M) = borders(c)$, $n = b_j$ and $m = b_{j+1}$. Note that this is well-defined since Theorem 6 states that if we have a node n, essentially a border, shared by several border lists, then each border list will have the exactly same next border, which is represented by the parent of n. Finally, the last border from each $borders(c)$ is a child of a root, which we will denote by r. Note that, for each $c \in C$, a path from c to r in T is equal to $(c = b_1, \ldots, b_M, r)$, where $(b_1, \ldots, b_M) = borders(c)$.

Given a node a in $btree(D, C)$ we write $children(a)$ to be the child nodes of a. We assume that $btree(D, C)$ is constructed so that the children are ordered from smallest to largest. In order to be able to modify the tree quickly, we store the tree structure as follows. Each node can have 3 pointers at most: a pointer to a right sibling, a pointer to a left sibling or to the parent, if there is no left sibling, and a pointer to the first child, see Figure 3(a) as an example.

Our next step is to show how to extract the maximal average, and by doing so compute the right interval. In order to do so we need the following results.

Theorem 7 *Let D be a sequence and let $1 \leq i \leq j \leq |D|$ be two indices. If $a \in borders(i)$ and $a \geq j$, then $a \in borders(j)$.*

Proof Assume that $a \notin borders(j)$. Then Theorem 4 implies that there are $j \leq x < a \leq y \leq |D|$ such that $av(x, a-1) \geq av(a, y)$. Since $i \leq x$, Theorem 4 immediately implies $a \notin borders(i)$. □

Corollary 1 *Let D be a sequence and let $1 \leq i \leq j \leq |D|$ be two indices. If $a = \max borders(i)$ and $a \geq j$, then $a = \max borders(j)$.*

Proof Theorem 7 implies $a \in borders(j)$. Since both border lists share a they also share any border larger than a. If both $b \in borders(j)$ and $b > a$, then Theorem 6 implies $b \in borders(i)$, which is a contradiction. Consequently, $a = \max borders(j)$. □

Corollary 2 *Let D be a sequence and let C be a set of indices. Let $btree(D, C)$ be a border tree and let r be its root. Select $c \in C$ and let $a \in children(r)$ be the smallest index such that $c \leq a$. Then $av(a, |D|) \geq av(b, |D|)$ for any $b \geq c$.*

Proof Let $b = \max borders(c)$ be the maximal border. Theorem 3 states that we need to prove $a = b$. We see immediately that $a \leq b$. Let d be such that $a = \max borders(d)$. If $d \leq c$, then, since $c \leq a$, Corollary 1 implies $a = \max borders(c) = b$. On the other hand, if $d > c$, then since $d \leq a \leq b$, Corollary 1 implies $b = \max borders(d) = a$. □

Corollary 2 gives a way to find the maximal average. Given $btree(D, C)$ and $c \in C$, we look for the smallest child of root, say a, such that $a \geq c$.

Our next step is to update a border tree from $T = btree(D[1, i-1], C)$ to $btree(D[1, i], C)$, an update step similar to Algorithm 2. We start by first adding a node i between a root and its children. This corresponds to the first two lines in Algorithm 2. After this we modify the tree such that Theorem 5 holds for every path from $c \in C$ to the root. In Algorithm 2 we simply deleted indices that were no longer borders. However, since a single node n can be shared by several border lists we cannot just delete it, since it might be the case that it is still used by another border list. Instead, we reattach children of n violating Theorem 5 to the root; effectively removing n from the border lists in which n is no longer a border. We give the pseudo-code in Algorithm 3.

Example 4 Let us continue Examples 2–3. Assume that we have a sequence given in Example 2 and that we have $C = \{1, 3, 8, 10\}$. Based on borders given in Example 3, the border tree is given in Figure 3(a). Assume that we see a new data point, $D_{11} = 1$. We have $borders(1, 11) = (1, 4, 7)$, $borders(3, 11) = (3, 4, 7)$, $borders(8, 11) = (8)$, and $borders(10, 11) = (10, 11)$.

We begin updating the tree by first adding node 11 between the root and its children, see Figure 3(b). We continue by checking the first child of 11: node 7, and reattach it to r, see Figure 3(c). After this, we check the first child of 7, node 4 and leave it unmodified. We continue by reattaching 9 to the root, see Figure 3(d), and similarly node 8, see Figure 3(e). Since node 9 is now a leaf and $9 \notin C$, we can delete it. Finally, we leave 10 attached to 11. The final tree, which corresponds to the correct border tree, is given in Figure 3(f).

Algorithm 3: UPDATETREE(T, C, D, i)

```
   input  : A tree T = btree(D[1, i − 1], C), a set of candidates C, a sequence D, an
            index i
   output : border tree btree(D[1, i], C)
 1 add node i between the root and its children;
 2 a ← i;
 3 while a exists do
 4     n ← next sibling of a;
 5     if a is a leaf then
 6         if a ∉ C then delete a from T;
 7     else
 8         b ← first child of a;
 9         if av(b, i) ≥ av(a, i) then
10             detach b from a;
11             attach b to the root left to a;
12             n ← b;
13     a ← n;
14 return T;
```

Theorem 8 *Let $T = btree(D[1, i-1], C)$. Algorithm* UPDATETREE(T, C, D, i) *outputs $btree(D[1, i], C)$.*

See Appendix for the proof.

In addition to UPDATETREE, we need a routine for updating the tree when an index c is deleted from C. This is needed when SEGMENT deletes a candidate for the optimal segmentation. In order to update we simply check whether c is a leaf, if it is, then we delete it, and recursively test the parent of c.

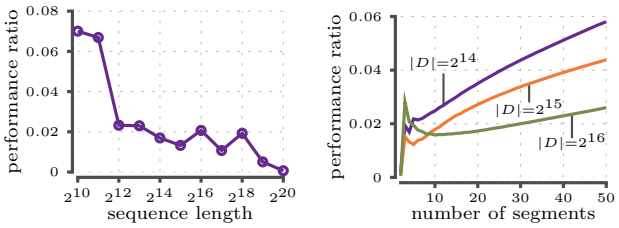
Finally, let us address memory and time complexity of a border tree. First of all, we have $|D|$ nodes at maximum, hence we need $O(|D|)$ memory. Let L be the maximum number of $|C|$. Let K_i be the number of nodes removed during UPDATETREE(T, D, C, i). If we do not modify the tree during the while-loop, then we execute the while-loop only once, since there is only child of r, namely i. Note that by the end of each UPDATETREE(T, D, C, i), root r can have at most L children. This means that at maximum we have done $L + K_i$ reattachments. Each reattachment increases the while-loop executions by 2: we need to check the child attached to the root and we need to check whether the parent has more children that need to be reattached. Hence, the while-loop is executed at most $2(L+K_i)+1$ times during UPDATETREE(T, D, C, i). Thus total time complexity is $O(|D|L + \sum_{i=1}^{|D|} K_i)$. Note that once a node is deleted it will not be introduced again. Hence, $\sum_{i=1}^{|D|} K_i \leq |D|$. This gives us a total execution time of $O(|D|L)$.

6 Experiments

In this section we empirically evaluate our approach on synthetic and real-world datasets.[2]

Synthetic data Our main contribution to the paper is the speedup of the dynamic program for finding the optimal segmentation when using one-dimensional log-linear models. We measure the efficiency by the total number of comparisons needed in Line 8 of Algorithm 1. We define a *performance ratio* by normalizing this number by the number of comparisons that we would have made if we would not use any pruning. This ensures that the ratio is between 0 and 1, smaller values indicating faster performance. Note that if we do not use any pruning, the total number of comparisons is $O(K|D|^2)$.

We begin by generating sequences of random samples drawn from the Gaussian distribution with 0 mean and 1 variance. We generated 11 sequences of lengths 2^k for $k = 10, \ldots, 20$ and computed the performance ratio of our segmentation using 4 segments of Gaussian distributions (as given in Example 1). From results given in Figure 4(a) we see that we obtain speedups of 1 order of magnitude for the smallest data, up to 3 orders of magnitude for longer data: the ratio for the largest sequence is 0.0007. Note that the ratios become smaller as the sequence becomes larger. The reason is that when considering longer segments, it becomes more likely that we can delete candidates, making the algorithm relatively faster. The absolute computation time grows with the length of a sequence, $11ms$, $1.3s$, and 20 minutes for sequences of length 2^{10}, 2^{15}, and 2^{20}, respectively.

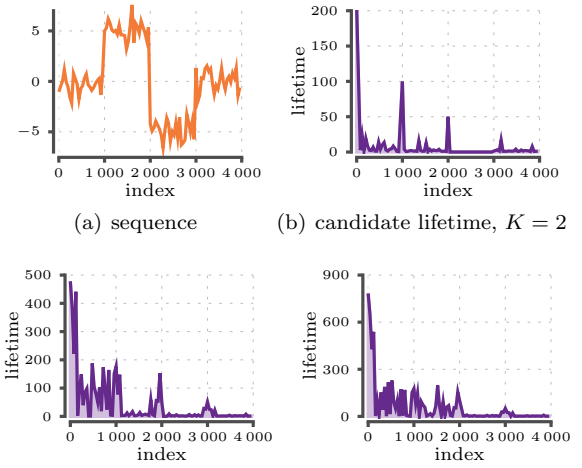


(a) speedup vs. sequence length (b) speedup vs. # of segments

Fig. 4 Performance ratio, total number of score comparisons (see Algorithm 1, Line 8), normalized between 0 and 1, as a function of sequence length 4(a), using 4 segments, and as a function of number of segments 4(b). Smaller values are better

Our second experiment is to study the performance ratio as a function of segments. We sampled 3 sequences from a Gaussian distribution, with 0 mean and 1 variance, of sizes 2^{14}, 2^{15}, 2^{16}. For each sequence we computed segmentations up to 50 segments. From the results given in Figure 4(b) we

[2] The implementation of the algorithm is given at http://adrem.ua.ac.be/segmentation



(a) sequence (b) candidate lifetime, $K = 2$

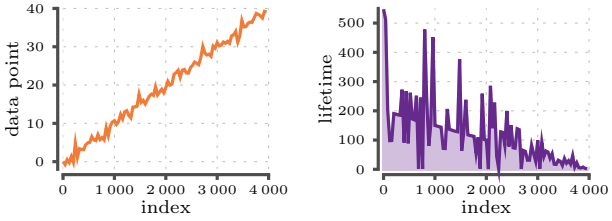
(c) candidate lifetime, $K = 3$ (d) candidate lifetime, $K = 4$

Fig. 5 Sequence of 4 Gaussian segments and candidate lifetimes, how many iterations is needed for a candidate to be deleted, when computing a segmentation with K segments from a segmentations of $K - 1$ segments, where $K = 2, 3, 4$. Smaller values imply lower computational burden

see that the performance ratio becomes worse as we increase the number of segments. The reason is that when segments become shorter, consequently, the right intervals are more compact and have less chance of being intersected with the left interval. Nevertheless, we get 0.06, 0.04, and 0.02 for performance ratios for our sequences when using 50 segments. The peak at 3 segments suggest that discovering segmentation with 3 segments is particularly expensive. To see why this is happening, first note that the first segment always starts from the beginning. This implies that when looking for a segmentation with 2 segments for a sequence $D[1, i]$, the second segment will be typically either really short or really long as its mean needs to differ from the mean of the first segment. If the second segment is short, it will have an abnormal right interval, consequently, the interval has a smaller chance of overlapping with the left interval of the next segment.

Our next step is to study how candidates for segments are distributed. A candidate c is added to C on Line 3 and deleted from C on Line 7 in Segment. The candidate is added when the counter i is equal to c and let us assume that it is deleted when the counter is equal to j. If c is not deleted, after the for-loop in Segment, we simply set $j = |D| + 1$. We define a *lifetime* of a candidate c to be $j - c$, that is, a candidate lifetime is how often it has been used in the maximization step on Line 8. The smaller the value, the less computational burden a candidate is producing. In the worst case, that is, without any pruning, the lifetime for a candidate c is equal to $|D| + 1 - c$.

To study candidate lifetimes we generate a sequence of 4 000 samples, consisting of 4 segments of Gaussian distribution with 0, 5, −5, and 0 means, respectively, and variance of 1 (see Figure 5(a)). We computed segmentations up to 4 segments and present the lifetimes in Figure 5.[3] We see that there are four major spikes in lifetimes, at the beginning of the sequence and around each change point. Let us consider a spike at 2 000 for $K = 4$. A candidate on the left side of the spike has a longer lifetime because the left interval of the next segment is shifted and it is less likely that it will intersect with the right interval. On the other hand, a candidate on the right side of the spike has a longer lifetime because the segment is short and the right interval has a higher chance of being abnormal. The same rationale applies to spike at the beginning of the sequence. The spikes grow with increasing number of segments, nevertheless they are shallow, implying that we have significant speedup. In fact, the performance ratios are 0.004, 0.01, 0.02 for segmentations with $K = 2, 3, 4$ segments, respectively.



(a) sequence (b) candidate lifetime, $K = 4$

Fig. 6 Sequence sampled from a Gaussian distribution with a slowly increasing mean and candidate lifetimes, how many iterations is needed for a candidate to be deleted, when computing a segmentation with K segments from a segmentations of $K - 1$ segments, for segmentation with K segments

Finally, let us demonstrate the limitations of our approach. We generate a sequence of 4 000 samples, where a sample i is generated from a Gaussian distribution with a mean of $i/100$ and variance of 1, see Figure 6(a). The performance ratio of segmentation with 4 segments is 0.06, the lifetimes are given in Figure 6(b). While we see a good performance for this data, when we increase the slope (or equivalently, lower variance) the performance ratio becomes worse. The worst case scenario is a genuinely monotonically increasing (or decreasing) sequence, that is, $D_{i+1} > D_i$. In such case, the left intervals and the right intervals will never overlap and no candidate will be pruned. We should point out that applying segmentation for a monotonic sequence in the first place is questionable as such sequence does not fit well the segmentation probabilistic model, and it might be beneficial to detrend the data to obtain a better segmentation.

[3] For clarity sake, figures show average lifetimes of bins containing 40 points

Real-world data We continue our experiments using real-world data sets. We considered 3 different datasets.[4] The first dataset, *Marotta*, is Space Shuttle Marotta Valve time series, consisting of 5 energize/de-energize cycles (TEK17). The second dataset, *Power*, consists of a power consumption of a Dutch research facility during the year 1997. The third dataset consists of two-dimensional time series extracted from videos of an actor performing various acts with and without a replica gun. Since this sequence is two-dimensional, we split the dimensions into *Video1* and *Video2*. The sequence lengths are given in Table 1.

Table 1 Characteristics of real-world datasets and performance of the algorithm with 20 segments. The last column is the time needed to compute the optimal segmentation using traditional dynamic program

Data	length	performance	time (s)	baseline time (s)
Marotta	5 000	0.04	0.6	13
Power	35 040	0.03	19.5	600
Video1	11 251	0.1	6.7	62
Video2	11 251	0.14	9.7	62

We study the performance by computing segmentations with 20 segments for each data and comparing it against the traditional dynamic program, that is, without deleting any candidates. From the results, given in Table 1, we see that our approach has a significant advantage over a baseline approach, for example, with *Power* dataset we find an optimal solution in 20 seconds while the baseline approach requires 10 minutes.

Finally, let us look at some of the discovered segmentations. In Figure 7 we present a segmentation of *Marotta* with 11 segments. The segments align with high and low energy states. Note that the 3rd high energy segment is more shallow than the other high energy segments. This cycle contains an anomaly as pointed out by Keogh et al (2005) resulting in a shorter high energy segment. In Figure 8 we show a segmentation with 3 segments of the power consumption. We can see that the mean of the middle segment is lower than the other means, indicating a summer season.

7 Related Work

Segmentation is an instance of a larger problem setting, called change point detection, see (Basseville and Nikiforov, 1993), for introduction. We can divide the problem settings broadly into two categories: offline and online. Although these settings have conceptually the same goal, the setup details make it different from an algorithmic point of view. In online change point detection

[4] The datasets were obtained from http://www.cs.ucr.edu/~eamonn/discords/

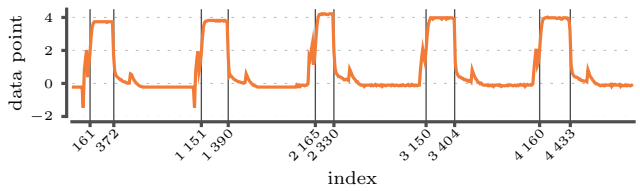


Fig. 7 Segmentation with 11 segments of Space Shuttle Marotta Valve time series

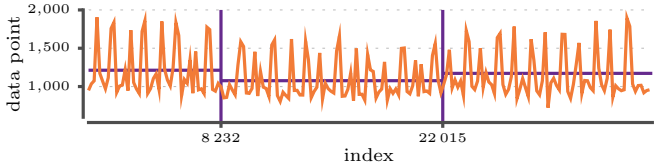


Fig. 8 Segmentation with 3 segments of the *Power* dataset. The horizontal lines represent the means of the individual segments.

(see Kifer et al (2004), for example) the data arrives in a stream fashion, typically there is no budget for how many change points are allowed, and the decision needs te be made within some time frame, whereas in segmentation, offline change point detection, new datapoints can change early segments. A typical goal for online change detection is to alert the system or a user of a change, whereas in segmentation the only goal is to summarize the sequence.

Popular approaches for segmentation are top-down approaches where we select greedily a new change-point (see Shatkay and Zdonik (1996); Bernaola-Galván et al (1996); Douglas and Peucker (1973); Lavrenko et al (2000), for example) and bottom-up approaches where at the beginning each point is a segment, and points are combined in a greedy fashion (see Palpanas et al (2004), for example). A randomized heuristic was suggested by Himberg et al (2001), where we start from a random segmentation and optimize the segment boundaries. These approaches, although fast, are heuristics and have no theoretical guarantees of the approximation quality. A divide-and-segment approach, an approximation algorithm with theoretical guarantees on the approximation quality was given by Terzi and Tsaparas (2006).

Modifications of the original segmentation problem have been also studied. Discovering recurrent sources is a setup where one limits the amount of distinct means of the segments to be H such that $H < K$, where K is the number of allowed segments has been suggested (Gionis and Mannila, 2003). Haiminen and Gionis (2004) study unimodal sequences, where means of the centroids (of one-dimensional sequence) are required to follow a unimodal curve, that is, the means should only rise to some point and then only decline afterwards. For a survey of the segmentation algorithms, see Chapter 8 in (Džeroski et al, 2011).

8 Discussion and Conclusions

In this paper we introduced a pruning technique to speedup the dynamic program used for solving the segmentation problem. We demonstrated on both synthetic and real-world data that we gain a significant speedup by using our pruning technique.

We should point out that our pruning is online, that is, the decision to delete a candidate is based only on current and past data points. We believe that we can speedup the algorithm further by applying additional pruning techniques based on future data points, such as (Gedikli et al, 2010). In addition, we conjecture that these optimizations may prove to be useful in other setups, such as, discovering HMMs or CRFs, where dynamic programs are used in order to optimize the model. We leave these studies as future work.

Segmentation requires a parameter, namely the number of segments. One approach to remove this parameter is by using model selection techniques, such as, BIC (Schwarz, 1978) or MDL (Grünwald, 2007). We conjecture that using these techniques not only remove the parameter but can be also used for further speedup.

Our algorithm is limited only to handle one-dimensional case. However, the key result, Theorem 1, actually handles the multi-dimensional case. The reason why we limit ourselves to one-dimensional case is that we were able to verify the sufficient conditions in Theorem 1 with relative ease. We leave studying applying Theorem 1 more generally as future work. While we are skeptical whether it is possible verify the conditions in Theorem 1 exactly, we believe that it is possible to find more conservative conditions that can be easily checked and that will imply the conditions in Theorem 1.

Acknowledgements

Nikolaj Tatti was partly supported by a Post-Doctoral Fellowship of the Research Foundation – Flanders (FWO).

References

Basseville M, Nikiforov IV (1993) Detection of Abrupt Changes — Theory and Application. Prentice-Hall

Bellman R (1961) On the approximation of curves by line segments using dynamic programming. Communications of the ACM 4(6)

Bernaola-Galván P, Román-Roldán R, Oliver JL (1996) Compositional segmentation and long-range fractal correlations in dna sequences. Physical Review E Statistical Physics Plasmas Fluids And Related Interdisciplinary Topics 53(5):5181–5189

Calders T, Dexters N, Goethals B (2007) Mining frequent itemsets in a stream. In: ICDM, pp 83–92

Douglas D, Peucker T (1973) Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Canadian Cartographer 10(2):112–122

Džeroski S, Goethals B, Panov P (eds) (2011) Inductive Databases and Constraint-based Data Mining. Springer

Gedikli A, Aksoy H, Unal NE, Kehagias A (2010) Modified dynamic programming approach for offline segmentation of long hydrometeorological time series. Stochastic Environmental Research and Risk Assessment 24(5)

Gionis A, Mannila H (2003) Finding recurrent sources in sequences. In: Proceedings of the seventh annual international conference on Research in computational molecular biology, RECOMB '03, pp 123–130

Grünwald P (2007) The Minimum Description Length Principle. MIT Press

Haiminen N, Gionis A (2004) Unimodal segmentation of sequences. In: ICDM, pp 106–113

Himberg J, Korpiaho K, Mannila H, Tikanmäki J, Toivonen H (2001) Time series segmentation for context recognition in mobile devices. In: ICDM, pp 203–210

Keogh EJ, Lin J, Fu AWC (2005) HOT SAX: Efficiently finding the most unusual time series subsequence. In: ICDM, pp 226–233

Kifer D, Ben-David S, Gehrke J (2004) Detecting change in data streams. In: VLDB, pp 180–191

Lavrenko V, Schmill M, Lawrie D, Ogilvie P, Jensen D, Allan J (2000) Mining of concurrent text and time series. In: KDD Workshop on Text Mining, pp 37–44

Palpanas T, Vlachos M, Keogh EJ, Gunopulos D, Truppel W (2004) Online amnesic approximation of streaming time series. In: ICDE, pp 339–349

Schwarz G (1978) Estimating the dimension of a model. Annals of Statistics 6(2):461–464

Shatkay H, Zdonik SB (1996) Approximate queries and representations for large data sequences. In: ICDE, pp 536–545

Terzi E, Tsaparas P (2006) Efficient algorithms for sequence segmentation. In: SIAM Data Mining

A Proofs

A.1 Proof of Theorem 1

Theorem 1 will follow from the following theorem.

Theorem 9 *Let $D = (D_1, \ldots, D_e)$. Let $1 \leq m < e$. Assume that diff$(D[1,m], D[m+1,e])$ is a cover. Then there exists $n > m$ such that $sc([1,n],[n+1,e]) > sc([1,m],[m+1,e])$ or there exists $l < m$ such that $sc([1,l],[l+1,e]) \geq sc([1,m],[m+1,e])$.*

In order to prove the theorem we will introduce some helpful notation. First, given a parameter vector s and r, we define

$$h(k \mid s, r) = sc([1,k] \mid s) + sc([k+1,e] \mid r) \quad .$$

Note that $h(k \mid s, r) \leq sc([1, k], [k + 1, e])$. We also define

$$g(l, \delta \mid s, r) = l(Z(s) - Z(r) + (s - r)^T \delta) \quad .$$

This function is essentially the difference between two scores.

Lemma 1 *Let $k > l$. We have $h(k \mid s, r) - h(l \mid s, r) = g(k - l, av(l + 1, k) \mid s, r)$.*

Proof Note that

$$\begin{aligned} h(k \mid s, r) &= kZ(s) + s^T c(k) + (e - k)Z(r) + r^T (c(e) - c(k)) \\ &= k(Z(s) - Z(r)) + (s - r)^T c(k) + eZ(r) + r^T c(e) \quad . \end{aligned}$$

The last two terms do not depend on k. This allows us to write

$$\begin{aligned} h(k \mid s, r) - h(l \mid s, r) &= k(Z(s) - Z(r)) + (s - r)^T c(k) - l(Z(s) - Z(r)) - (s - r)^T c(l) \\ &= (k - l)\big(Z(s) - Z(r) + (s - r)^T \frac{c(k) - c(l)}{k - l}\big) = g(k - l, av(l + 1, k) \mid s, r) \quad . \end{aligned}$$

This completes the proof. □

Proof (Proof of Theorem 9) Write $y = sc([1, m], [m + 1, e])$ and define

$$x = \max_{k<m} sc([1, k], [k + 1, e]) \quad \text{and} \quad z = \max_{k>m} sc([1, k], [k + 1, e]) \quad .$$

We need to show that either $x \geq y$ or $z > y$. Assume that $z \leq y$. Fix $\epsilon > 0$. By definition, there exist s and r such that

$$sc([1, m] \mid s) + sc([m + 1, e] \mid r) \geq y - \epsilon \quad .$$

From now on we will write $h(k)$ to mean $h(k \mid s, r)$ and $g(k, \delta)$ to mean $g(k, \delta \mid s, r)$. We must have $h(m) + \epsilon \geq y \geq z$ or, equivalently, $\epsilon \geq z - h(m)$.

Since *diff*$(D[1, m], D[m + 1, e])$ is a cover, there exist integers l and n, $0 \leq l < m < n \leq e$, such that $(\alpha - \beta)^T (s - r) \geq 0$, where $\alpha = av(m + 1, n)$ and $\beta = av(l + 1, m)$.

Define $c = (n - m)/(m - l)$. We now have

$$\begin{aligned} \epsilon &\geq z - h(m) \geq h(n) - h(m) = g(n - m, \alpha) = cg(m - l, \alpha) \\ &= cg(m - l, \beta) + c(m - l)(s - r)^T (\alpha - \beta) \geq cg(m - l, \beta) \\ &= c(h(m) - h(l)) \geq c(h(m) - x) \geq c(y - \epsilon - x), \end{aligned}$$

which implies $y - x \leq \epsilon(1 + c^{-1}) \leq \epsilon(1 + e)$. Since this holds for any $\epsilon > 0$, we conclude that $y \leq x$. This proves the theorem. □

Proof (Proof of Theorem 1) Let P a segmentation and let I and J be two consecutive segments such that *diff*$(D[I], D[J])$ is a cover. We can now apply Theorem 9 to find alternative segments I' and J' such that if we define P' by replacing I and J from P with I' and J' then either $sc(P' \mid D) > sc(P' \mid D)$ or $sc(P' \mid D) \geq sc(P' \mid D)$ and I' ends before I. We repeat this until no consecutive segments constitute a cover. This repetition ends because no segmentation will occur twice during these steps and there is a finite number of segmentations. The reason why no segmentation occur twice is because either the score properly increases or the score stays the same and we move a breakpoint to the left. □

A.2 Proof of Theorem 8

Let U be the resulting tree from UpdateTree(T, C, D, i). To prove the theorem we need to show that the paths of U from leafs to the root consists of borders, there are no nodes in U outside the borders, and that children are ordered. We will prove these results in a series of lemmata.

Lemma 2 *Let T' be a tree after we have added a node i in* UpdateTree. *Let $n \neq i$ be a node in T' and let m be its parent. Let $c \in C$ be such that $n \in borders(c, i-1)$. If $m \notin borders(c, i)$, then n will cease to be a child of m during some stage of* UpdateTree.

Proof Let r be a root node of T'. Consider a pre-order of nodes of T', that is, parents and earlier siblings come first. We will prove the lemma using induction on the pre-order.

To prove the first step, let n be the first child of i. If $i \notin borders(c, i)$, then Theorem 5 implies that $av(n, i) \geq av(i, i)$ which is exactly the test on Line 9. Hence, n will be disconnected from i.

Let us now prove the induction step. Let p be the parent of m in T'. Assume that $p \neq r$. Note that p is the border next to m in $borders(c, i-1)$. Theorem 5 implies that $p \notin borders(c, i)$, hence the induction assumption implies that m and p are disconnected and m becomes a child of r at some point.

Assume now that n is not the first child of m and let q be the sibling left to n, and let p be such that $q \in borders(p, i-1)$. Theorem 3 implies that $av(q, m-1) \geq av(j, m-1)$ for any $q \leq j < m$. Since $n > q$, we must have $av(q, m-1) \geq av(n, m-1) \geq av(m, i)$, which implies that $m \notin borders(p, i)$. Again, the induction assumption implies that q and m will be disconnected. Consequently, n will be the first child of m at some point.

Note that while moving m or left siblings of n to be children of r we move the current node a in UpdateTree to the left. Hence, there will be a point where $a = m$ and n is the first child of m. Theorem 5 implies that $av(n, i) \geq av(m, i)$ which is exactly the test on Line 9. Hence, n will be disconnected from m. This proves the lemma. □

Lemma 3 *For every $c \in C$, a path in U from c to a child of the root node r equals $borders(c, i)$.*

Proof Fix $c \in C$ and let $(b_1, \ldots, b_M) = borders(c, i-1)$ and define $b_{M+1} = i$. Theorem 5 implies that there is $1 \leq N \leq M+1$ such that $(b_1, \ldots, b_N) = borders(c, i)$.

After adding i to T, UpdateTree will not add new nodes into the path from c to r. Lemma 2 now implies that the path from c to r will be $(b_1, \ldots, b_K)$, where $K \leq N$. If $N = 1$, then immediately $K = 1$. To conclude that $K = N$ in general, assume that $N > 1$ and assume that at some point in UpdateTree we have $a = b_N$ and $b = b_{N-1}$. Then, according to Theorem 5, the test on Line 9 will fail and b_{N-1} remains as a child of b_N. □

Lemma 4 *Let n be a node in U, then there is $c \in C$ such that $n \in borders(c, i)$.*

Proof Let m be a node that occurs in T but not in $btree(D[1, i], C)$. The lemma will follow if we can show that m is not in U. Let n be the last child of m. Lemma 2 implies that at some point n will be disconnected from m and we will visit m when it is a leaf, since $m \notin C$, we will delete m. □

Lemma 5 *Consider a post-order of nodes of $T = btree(D[1, i-1], C)$, that is, parents and later siblings come first. Node values decrease with respect to this order.*

Proof We will prove that the following holds: Let n be a node and let m be its left sibling. Let q be the smallest child of n. Then $m < q$. Note that this automatically proves the lemma.

Note that $q \in C$. To prove that $m < q$, let $c \in C$ such that $m \in borders(c, i-1)$. If $c \geq q$, then since $n > m \geq c$, Theorem 7 implies that $n \in borders(c, i-1)$ which is a contradiction. Consequently, $c < q$. If $q \leq m$, then again Theorem 7 implies that $m \in borders(q, i-1)$ which is a contradiction. This proves that $m < q$. □

Lemma 6 *Child nodes of each node in U are ordered from smallest to largest.*

Proof UpdateTree modifies the tree by moving the first child of a node a to be the left sibling of a. This does not change the post-order of the nodes. This implies that, since node values decrease with respect to the post-order in T, they will also decrease in U. This proves the lemma. □