Strongly polynomial efficient approximation scheme for segmentation

Nikolaj Tatti

F-Secure, HIIT, Aalto University, Finland

Abstract

Partitioning a sequence of length n into k coherent segments (SEG) is one of the classic optimization problems. As long as the optimization criterion is additive, SEG can be solved exactly in $\mathcal{O}(n^2 k)$ time using a classic dynamic program. Due to the quadratic term, computing the exact segmentation may be too expensive for long sequences, which has led to development of approximate solutions. We consider an existing estimation scheme that computes $(1 + \epsilon)$ approximation in polylogarithmic time. We augment this algorithm, making it strongly polynomial. We do this by first solving a slightly different segmentation problem (MAXSEG), where the quality of the segmentation is the maximum penalty of an individual segment. By using this solution to initialize the estimation scheme, we are able to obtain a strongly polynomial algorithm. In addition, we consider a cumulative version of SEG, where we are asked to discover the optimal segmentation for each prefix of the input sequence. We propose a strongly polynomial algorithm that yields $(1 + \epsilon)$ approximation in $\mathcal{O}(nk^2/\epsilon)$ time. Finally, we consider a cumulative version of MAXSEG, and show that we can solve the problem in $\mathcal{O}(nk \log k)$ time.

1. Introduction

Partitioning a sequence into coherent segments is one of the classic optimization problems, with applications in various domains, such as discovering context in mobile devices [12], similarity search in time-series databases [13], and bioinformatics [15, 17].

More formally, we are given a sequence of length n and a penalty function of a segment, and we are asked to find a segmentation with k segments such that the sum of penalties is minimized (SEG). As long as the score is additive, SEG can be solved exactly in $\mathcal{O}(n^2 k)$ time using a classic dynamic program [2].

Due to the quadratic term, computing the exact segmentation may be too expensive for long sequences, which has led to development of approximate solutions. Guha et al. [10] suggested an algorithm that yields $(1 + \epsilon)$ ap-

proximation in $\mathcal{O}(k^3 \log^2 n + k^3 \epsilon^{-2} \log n)$ time, if we can compute the penalty of a single segment in constant time.[1] This method assumes that we are dealing with an integer sequence that is penalized by L_2-error. Without these assumptions the computational complexity deteriorates to $\mathcal{O}(k^3 \log \frac{\theta}{\alpha} \log n + k^3 \epsilon^{-2} \log n)$, where θ is the cost of the optimal solution and α is the smallest possible non-zero penalty. Consequently, without the aforementioned assumptions the term $\mathcal{O}(\frac{\theta}{\alpha})$ may be arbitrarily large, and the method is not strongly polynomial.

In this paper we demonstrate a simple approach for how to augment this method making it strongly polynomial. The reason for having $\log \frac{\theta}{\alpha}$ term is that the algorithm needs to first find a 2-approximation. This is done by setting α to be the smallest non-zero cost and then increasing exponentially its value until an appropriate value

---

[1] Note that Guha et al. [10] refers to this problem as histogram construction.

Email address: `nikolaj.tatti@aalto.fi` (Nikolaj Tatti)

is discovered (we can verify whether the value is appropriate in $\mathcal{O}(k^3 \log n)$ time). Instead of using the smallest non-zero cost, we first compute a k-approximation of the segmentation cost, say η, and then set $\alpha = \eta/k$. This will free us of integrality assumptions, reducing the computation time to $\mathcal{O}(k^3 \log k \log n + k^3 \epsilon^{-2} \log n)$. Moreover, we no longer need to discover the smallest non-zero cost, which can be non-trivial.

In order to discover k-approximation we consider a different segmentation problem, where the score of the whole segmentation is not the sum but the maximum value of a single segment (MAXSEG). We show in Section 4 that the segmentation solving MAXSEG yields the needed k-approximation for SEG. Luckily, we can solve MAXSEG in $\mathcal{O}(k^2 \log^2 n)$ time by using an algorithm by Guha and Shim [7].

The method by Guha et al. [10] only computes a k-segmentation for whole sequence. We consider a cumulative variant of the segmentation problem, where we are asked to compute an ℓ-segmentation for all prefixes and for all $\ell \leq k$. As our second contribution, given in Section 5, we propose a *strongly polynomial* algorithm that yields $(1 + \epsilon)$ approximation in $\mathcal{O}(nk^2/\epsilon)$ time. Finally, in Section 6 we also consider a cumulative variant of MAXSEG problem, for which we propose an exact algorithm with computational complexity of $\mathcal{O}(nk \log k)$.

2. Related work

As discussed earlier, our approach is based on improving method given by Guha et al. [10], that achieves an $(1 + \epsilon)$ approximation in $\mathcal{O}(k^3 \log^2 n + k^3 \epsilon^{-2} \log n)$ time, under some assumptions. Terzi and Tsaparas [20] suggested an approximation algorithm that yields 3-approximation in $\mathcal{O}(n^{4/3} k^{5/3})$ time, assuming L_2 error.

We also consider an algorithm for the cumulative version of the problem. The main idea behind the algorithm is inspired by Guha et al. [8], where the algorithm requires $\mathcal{O}(nk^2/\epsilon \log n)$ time as well as integrality assumptions. We

achieve $\mathcal{O}(nk^2/\epsilon)$ time and, more importantly, we do not need any integrality assumptions, making the algorithm strongly polynomial.

Fast heuristics that do not yield approximation guarantees have been proposed. These methods include top-down approaches, based on splitting segments (see Shatkay and Zdonik [18], Bernaola-Galván et al. [3], Douglas and Peucker [4], Lavrenko et al. [14], for example), and bottom-up approaches, based on merging segments (see Palpanas et al. [16], for example). A different approach was suggested by Himberg et al. [12], where the authors optimize boundaries of a random segmentation.

If the penalty function is concave, then we can discover the exact optimal segmentation in $\mathcal{O}(k(n + k))$ time using the SMAWK algorithm [1, 6]. This is the case when segmenting a monotonic one-dimensional sequence and using L_1-error as a penalty [11, 5].

If we were to evaluate the segmentation by the maximum penalty of a segment, instead of the sum of all penalties, then the problem changes radically. Guha and Shim [7] showed that we can compute the exact solution in $\mathcal{O}(n + k^2 \log^3 n)$ time, when using the L_∞ penalty. Guha et al. [9] also showed that we can compute the cumulative version in $\mathcal{O}(kn \log^2 n)$ time,[2] which we improve in Section 6.

3. Preliminaries

Segmentation problem. Throughout the paper we will assume that we are given an integer n and a *penalty function* p that maps two integers $1 \leq a \leq b \leq n$ to a positive real number.

The most common selection for the penalty function is an L_q distance of individual points in a sequence segment to the optimal centroid, that is, given a sequence of real

---

[2]or in $\mathcal{O}(kn \log n)$ time if we assume that we can compute the penalty of a segment in constant time.

numbers, $z_1, \ldots, z_{n+1}$, the score is equal to

$$L_q(a, b) = \min_{\mu} \sum_{i=a}^{b-1} \|z_i - \mu\|_q \quad . \qquad (1)$$

Since we do not need any notion of sequence in this paper, we abstract it out, and speak directly only about penalty function.

Throughout the paper, we will assume that

1. for any $1 \le a_1 \le a_2 \le b_2 \le b_1$, we have $p(a_2, b_2) \le p(a_1, b_1)$. Also, $p(a_1, a_1) = 0$,

2. we can compute the penalty in constant time,

3. we can perform arithmetic operations to the penalty in constant time, as well as compare the scores.

The first assumption typically holds. For example, any L_p-error will satisfy this assumption, as well as any log-likelihood-based errors [19]. The third assumption is a technicality used by the definition of strongly polynomial time. The second assumption is the most limiting one. It holds for L_2-error: we can compute the penalty in constant time by precomputing cumulative mean and the second moment. It also holds for log-linear models [19]. However, for example, we need $\mathcal{O}(\log n)$ time to compute L_∞-error [7]. In such a case, we need to multiply the running time by the time needed to compute the penalty. We will ignore the running time needed for any precomputation as this depends on the used penalty.

Given an integer k and an interval $[i, j]$, where i and j are integers with $1 \le i \le j \le n$, a *k-segmentation* B *covering* $[i, j]$ is a sequence of $k + 1$ integers

$$B = (i = b_0 \le b_1 \le \cdots \le b_k = j) \quad .$$

We omit k and simply write segmentation, whenever k is known from context.

Let p be a penalty function for individual segments. Given a segmentation B, we extend the definition of p and define $p(B)$, the penalty for the segmentation B, as

$$p(B) = \sum_{i=1}^{k} p(b_{i-1}, b_i) \quad .$$

We can now state the classic segmentation problem:

Problem 3.1 (SEG)**.** *Given a penalty p and an integer k find a segmentation B covering $[1, n]$ that minimizes $p(B)$.*

Since the penalty score is the sum of the interval penalties, we can solve SEG with a dynamic program given by Bellman [2]. The computational complexity of this program is $\mathcal{O}(n^2 k)$, which is prohibitively slow for large n.

We will also consider a cumulative variant of the segmentation problem defined as follows.

Problem 3.2 (ALLSEG)**.** *Given a penalty function p and an integer k find an ℓ-segmentation B covering $[1, i]$ that minimizes $p(B)$, for every $i = 1, \ldots, n$ and $\ell = 1, \ldots k$.*

Note that Bellman [2] in fact solves ALLSEG, and uses the solution to solve SEG. However, the state-of-the-art approximation algorithm, by Guha et al. [10], solving SEG does not solve ALLSEG, hence we will propose a separate algorithm.

Approximation algorithm. Our contribution is an additional component to the approximation algorithm by Guha et al. [10], a state-of-the-art approximation algorithm for estimating SEG. We devote the rest of this section to explaining the technical details of this algorithm, and what is the issue that we are addressing.

Guha et al. [10] showed that under some assumptions it is possible to obtain a $(1 + \epsilon)$ approximation to SEG in $\mathcal{O}(k^3 \log^2(n) + k^3 \epsilon^{-2} \log(n))$ time. What makes this algorithm truly remarkable is that it is polylogarithmic in n, while the exact algorithm is quadratic in n. Here we assume that we have a constant-time access to the score p. If one uses L_2 error as a penalty, then one is forced to precompute the score, which requires additional $\mathcal{O}(n)$ time. However, this term depends neither on k nor on ϵ.

The key idea behind the approach by Guha et al. [10] is a sub-routine, ORACLE(δ, u), that relies on two parameters δ and u. ORACLE constructs a k-segmentation with the following property.[3]

---

[3]The actual subroutine is too complex to be described in compact space. For details, we refer reader to [10].

Proposition 3.1. *Suppose that p is a penalty function, and k is an integer. Let θ be the cost of an optimal k-segmentation covering $[1, n]$. Let δ and u be two positive real numbers such that $\theta + \delta \leq u$. Then $\text{ORACLE}(\delta, u)$ returns a k-segmentation with a cost of τ such that $\tau \leq \theta + \delta$. $\text{ORACLE}(\delta, u)$ runs in $\mathcal{O}\left(k^3 \frac{u^2}{\delta^2} \log n\right)$ time.*

Proposition 3.1 describes the trade-off between the accuracy and computational complexity: We can achieve good accuracy with small δ, and large enough u, but we have to pay the price in running time.

We will now describe how to select u and δ in a smart way. Let θ be the cost of an optimal k-segmentation. Assume that we know an estimate of θ, say η, such that $\eta \leq \theta \leq 2\eta$. Let us set

$$u = (2 + \epsilon)\eta \quad \text{and} \quad \delta = \epsilon\eta \quad .$$

As $\theta + \delta \leq u$, Proposition 3.1 guarantees that $\text{ORACLE}(\delta, u)$ returns a k-segmentation with a cost of τ such that

$$\tau \leq \theta + \delta = \theta + \epsilon\eta \leq \theta + \epsilon\theta = (1 + \epsilon)\theta \quad .$$

In other words, the resulting segmentation yields a $(1 + \epsilon)$ approximation guarantee. The computational complexity of $\text{ORACLE}(\delta, u)$ is equal to $\mathcal{O}\left(k^3 \epsilon^{-2} \log n\right)$.

The difficult part is to discover η such that $\eta \leq \theta \leq 2\eta$. This can be also done using the oracle (see Algorithm 1[4]). Assume that we know a *lower bound* for θ, say α. We first set $\eta = \alpha$ and check using ORACLE to see if η is too small. If it is, then we increase the value and repeat.

To see why ESTIMATE works, assume that we are at a point in the while-loop where $2\eta < \theta$. Then $\tau \geq \theta > 2\eta$ and the while-loop is not terminated. This guarantees that for the final η, we have $\theta \leq 2\eta$. Let us show that $\eta \leq \theta$. If the while-loop is terminated while $\eta < \theta$, then there is nothing to prove. Assume otherwise, that is, at some point

---

[4] The original pseudo-code given by Guha et al. [10] contains a small error, and only yields $\eta \leq \theta < 4\eta$. Here we present the corrected variant.

Algorithm 1: $\text{ESTIMATE}(\alpha)$, computes η such that $\eta \leq \theta \leq 2\eta$, where θ is the optimal cost. Requires $\alpha \leq \theta$ as an input parameter.

1 $\eta \leftarrow \alpha$;

2 $\tau \leftarrow$ the cost of the solution by $\text{ORACLE}(\eta/2, 2\eta)$;

3 while $\tau > 2\eta$ **do**

4 $\quad$ $\eta \leftarrow 1.5\eta$;

5 $\quad$ $\tau \leftarrow$ the cost of the solution by $\text{ORACLE}(\eta/2, 2\eta)$;

6 return η;

---

we have $\eta \leq \theta \leq 1.5\eta$. Since $\theta + \eta/2 \leq 2\eta$, Proposition 3.1 guarantees that $\tau \leq \theta + \eta/2 \leq 2\eta$, and we exit the while-loop with $\eta \leq \theta$.

The computational complexity of a single ORACLE call is $\mathcal{O}\left(k^3 \log n\right)$, and the total computational complexity of ESTIMATE is $\mathcal{O}\left(\log(\frac{\theta}{\alpha})k^3 \log n\right)$. At this point, Guha et al. [10] assume (implicitly) that the penalty function is L_2 error of a sequence (see Eq. 1), and the values in the sequence are integers encoded with a standard bit representation in at most $\mathcal{O}(\log n)$ space. These assumptions have two consequences: (i) we can use $\alpha = 1/2$, the smallest non-zero cost for a segmentation, and (ii) the number of iterations $\mathcal{O}\left(\log \frac{\theta}{\alpha}\right)$ is bounded by $\mathcal{O}(\log n)$. This leads to a computational complexity of $\mathcal{O}\left(k^3 \log^2 n\right)$.

The L_2 assumption is not critical since the same argument can be done for many other cost functions, however one is forced to find an appropriate α for each case individually. Moreover, there are penalty functions for which this argument does not work, for example, $p(a, b) = \exp\left(L_2(a, b)\right)$, where L_2 is given in Eq. 1.

The more critical assumption is that the numbers in a sequence are integers, and this assumption can be easily violated if we have a sequence of numbers represented in a floating-point format. If this is the case, then we can no longer select $\alpha = 1/2$. In fact, there is no easy way of selecting α such that (i) we are sure that α is less than the optimal segmentation score, and (ii) the number of loops

---

needed by ESTIMATE, $\mathcal{O}\left(\log \frac{\theta}{\alpha}\right)$, is bounded by a (slowly increasing) function of n.

In the following section, we will show how to select α such that the number of loops in ESTIMATE remains small. More specifically, we demonstrate how to select α such that $\alpha \leq \theta \leq k\alpha$. This immediately implies that the computational complexity of ESTIMATE reduces to $\mathcal{O}\left(k^3 \log(k) \log(n)\right)$. More importantly, we do not need any awkward assumptions about having a sequence of only integer values, making this algorithm strongly polynomial. Moreover, this procedure works on any penalty function, hence a finding appropriate α manually is no longer needed.

4. Strongly polynomial scheme for segmentation

To find α, the parameter for ESTIMATE, we consider a different optimization problem, where the segmentation is evaluated by its most costly segment.

Problem 4.1 (MAXSEG)**.** *Given a penalty p and an integer k, find a k-segmentation B covering $[1, n]$ that minimizes*

$$p_{max}(B) = \max_{1 \leq j \leq k} p(b_{j-1}, b_j) \quad .$$

The next proposition states why solving MAXSEG helps us to discover α.

Proposition 4.1. *Suppose that p is a penalty function, and k is an integer. Let B be a solution to SEG, and let B' be a solution to MAXSEG. Then*

$$p(B')/k \leq p(B) \leq p(B') \quad .$$

Proof. To prove the first inequality write

$$p(B') = \sum_{i=1}^{k} p\left(b'_{i-1}, b'_i\right) \leq k \max_{1 \leq i \leq k} p\left(b'_{i-1}, b'_i\right)$$

$$= k p_{max}(B') \leq k p_{max}(B) \leq k p(B) .$$

The claim follows as the second inequality is trivial. $\square$

By solving MAXSEG and obtaining a solution B', we can set $\alpha = p(B)/k$ for ESTIMATE. The remaining problem is how to solve MAXSEG in sub-linear time.

Here we can reuse an algorithm given by Guha and Shim [7]. This algorithm is designed to solve an instance of MAXSEG, where the penalty function is L_∞, which is

$$p(a, b) = \min_{\mu} \sum_{i=a}^{b-1} |x_i - \mu| \quad .$$

Luckily, the same algorithm and the proof of correctness, see Lemma 3 in [7], is valid for any monotonic penalty.[5] The algorithm runs in $\mathcal{O}\left(k^2 \log^2 n\right)$ time. Thus, the total running time to obtain a segmentation with $(1 + \epsilon)$ approximation guarantee is

$$\mathcal{O}\left(k^2 \log^2(n) + k^3 \log(k) \log(n) + k^3 \epsilon^{-2} \log(n)\right) \quad .$$

5. Strongly polynomial and linear-time scheme for cumulative segmentation

In this section we present a strongly polynomial algorithm that approximates ALLSEG in $\mathcal{O}\left(nk^2/\epsilon\right)$ time. Let $o[i, \ell]$ be the cost for an optimal ℓ-segmentation covering $[1, i]$. The optimal segmentation is computed with a dynamic program based on the identity

$$o[i, \ell] = \min_{j \leq i} o[j, \ell - 1] + p(j, i) \quad .$$

The integer j yielding the optimal cost will be the starting point of the last segment in an optimal segmentation. To speed-up the discovery of j, we will not test every $j \leq i$, but instead we will use a small set of candidates, say A. So, instead of computing a single entry in $\mathcal{O}(n)$ time, we only need $\mathcal{O}(|A|)$ time.

The set A depends on i and ℓ, and we update it as we change i and ℓ. The key point here is to keep A very small, in fact, $|A| \in \mathcal{O}(k/\epsilon)$, while having enough entries to yield the approximation guarantee.

---

[5] For the sake of completeness, we revisit this algorithm in supplementary material.

Our approach works as follows (see Algorithm 2 for the pseudo-code): the algorithm loops over ℓ and i with i being the inner for-loop, and maintains a set of candidates A. There are 3 main steps inside the inner for-loop:

(1) Test the current candidates A.

(2) Test $a = 1 + \max A$, and add a to A. Repeat this step until $a = i$ or if the current score $s[i, \ell]$ becomes smaller than $s[a, \ell - 1]$. In the latter case, we can stop because $s[a', \ell - 1] \geq s[i, \ell]$ for any $a' > a$, so we know that there are no candidates that can improve $s[i, \ell]$.

(3) For every consecutive triplet $a_j, a_{j+1}, a_{j+2} \in A$ such that $s[a_{j+2}, \ell - 1] - s[a_j, \ell - 1] \leq s[i, \ell]\frac{\epsilon}{k+\ell\epsilon}$, remove a_j (see Algorithm 3). This will keep $|A|$ small for the next round while yielding the approximation guarantee.

Algorithm 2: ALL-DP(k, ϵ), computes a table s such that $s[i, \ell]$ is a $(1+\epsilon)$ approximation of the cost of an optimal ℓ-segmentation covering $[1, i]$.

1 $s[i, 1] \leftarrow p(1, i)$ for $i = 1, \ldots, n$;
2 **foreach** $\ell = 2, \ldots, k$ **do**
3     $A \leftarrow \{1\}$;
4   **foreach** $i = 1, \ldots, n$ **do**
5        $s[i, \ell] \leftarrow \min_{a \in A} s[a, \ell - 1] + p(a, i)$;
6        $a \leftarrow 1 + \max A$;
7   **while** $a \leq i$ **and** $s[a, \ell - 1] \leq s[i, \ell]$ **do**
8           $s[i, \ell] \leftarrow \min(s[i, \ell], s[a, \ell - 1] + p(a, i))$;
9   insert a to A;
10          $a \leftarrow a + 1$;
11       $A \leftarrow$ SPARSIFY$(A, s[i, \ell]\frac{\epsilon}{k+\ell\epsilon}, \ell)$;
12 **return** s;

Our next step is to prove the correctness of ALL-DP.

Proposition 5.1. *Let $o[i, \ell]$ be the cost of an optimal ℓ-segmentation covering $[1, i]$. Let $s =$ ALL-DP(k, ϵ) be the solution returned by the approximation algorithm. Then for any $i = 1, \ldots, n$ and $\ell = 1, \ldots, k$,*

$$s[i, \ell] \leq o[i, \ell]\left(1 + \frac{\epsilon\ell}{k}\right) \quad .$$

Algorithm 3: SPARSIFY(A, δ, ℓ), sparsifies $A = a_1, \ldots, a_{|A|}$ using $s[a_i, \ell - 1]$ and δ.

1 $j \leftarrow 1$;
2 **while** $j \leq |A| - 2$ **do**
3   **if** $s[a_{j+2}, \ell - 1] - s[a_j, \ell - 1] \leq \delta$ **then**
4   remove a_{j+1} from A, and update the indices;
5   **else**
6        $j \leftarrow j + 1$;

Before proving the claim, let us introduce some notation that will be used throughout the remainder of the section. Assume that ℓ is fixed, and let A_i be the set A at the beginning of the ith round of ALL-DP. Let $m_i = \max A_i$. The entries in A_i are always sorted, and we will often refer to these entries as a_j. Let us write A_i' to be the set A which is given to SPARSIFY during the ith round.

To prove the claim, we need two lemmas. First we show that the score is increasing as a function of i.

Lemma 5.1. *The score is monotone, $s[i, \ell] \leq s[i + 1, \ell]$.*

Proof. We will prove the lemma by induction on ℓ. The $\ell = 1$ case holds since $s[i, \ell] = p(1, i)$. Assume that the lemma holds for $\ell - 1$.

Algorithm 2 uses the indices in A_i' (Lines 5 and 8) for computing $s[i, \ell]$, that is,

$$s[i, \ell] = \min_{a \in A_i'} s[a, \ell - 1] + p(a, i) \quad . \qquad (2)$$

Let B be the ℓ-segmentation corresponding to the cost $s[i + 1, \ell]$. Let b be the starting point of the last segment of B. If $b \leq m_{i+1}$, then b was selected during Line 5, that is, $b \in A_{i+1} \subseteq A_i'$, so Eq. 2 implies that

$$s[i, \ell] \leq s[b, \ell - 1] + p(b, i)$$
$$\leq s[b, \ell - 1] + p(b, i + 1) = s[i + 1, \ell] \quad .$$

On the other hand, if $b > m_{i+1}$, then, due to the while-loop in ALL-DP, either

$$s[i, \ell] < s[m_{i+1} + 1, \ell - 1] \qquad (3)$$

6

or $i = m_{i+1} \in A_i'$, and so Eq. 2 implies that

$$s[i, \ell] \leq s[i, \ell - 1] = s[m_{i+1}, \ell - 1] \quad . \qquad (4)$$

Due to the induction hypothesis on ℓ, the right-hand sides of Eqs. 3–4 are bound by $s[b, \ell - 1]$. Since,

$$s[b, \ell - 1] \leq s[b, \ell - 1] + p(b, i + 1) = s[i + 1, \ell],$$

we have proved the claim. □

The second lemma essentially states that A_i is dense enough to yield an approximation. To state the lemma, let $\delta_i = s[i, \ell] \times \epsilon/(k + \ell\epsilon)$ be the value of δ in SPARSIFY during the ith round. For simplicity, we also define $\delta_0 = 0$.

Lemma 5.2. *For every $b \in [1, m_i]$, there is $a_j \in A_i$ s.t.*

$$s[a_j, \ell - 1] + p(a_j, i) \leq s[b, \ell - 1] + p(b, i) + \delta_{i-1} \quad .$$

Proof. We say that a sorted list of indices $X = x_1, \ldots, x_{|X|}$ with $x_1 = 1$ is δ-dense if $s[x_j, \ell - 1] \leq \delta + s[x_{j-1}, \ell - 1]$ or $x_j = x_{j-1} + 1$ for any $x_j \in X_i$. Note that if X is δ-dense, then so is SPARSIFY(X, δ, ℓ) due to the if-condition in Algorithm 3.

We claim that A_i is δ_{i-1}-dense. We will prove this by induction on i. This is vacuosly true for $i = 1$. Assume that A_{i-1} is δ_{i-2}-dense. It is trivial to see that $A_{i-1}' = A_{i-1} \cup [m_{i-1} + 1, m_i]$, thus A_{i-1}' is δ_{i-2}-dense. Since $\delta_i = s[i, \ell] \times \epsilon/(k + \ell\epsilon)$, Lemma 5.1 guarantees that $\delta_{i-2} \leq \delta_{i-1}$, so A_{i-1}' is δ_{i-1}-dense. Finally, A_i is δ_{i-1}-dense, since $A_i = $ SPARSIFY$(A_{i-1}', \delta_{i-1}, \ell)$.

Let $a_j \in A_i$ be the smallest index that is larger than or equal to b. If $b = a_j$, then the lemma follows immediately. If $b < a_j$, then $j > 1$ and

$$s[a_j, \ell - 1] \leq \delta_{i-1} + s[a_{j-1}, \ell - 1] \leq \delta_{i-1} + s[b, \ell - 1],$$

where the second inequality follows from the fact that $b > a_{j-1}$ and Lemma 5.1.

The result follows as $p(a, i) \leq p(b, i)$, for any $a \geq b$. □

We can now prove the main result.

Proof of Proposition 5.1. Write $\gamma = 1 + \frac{\epsilon(\ell-1)}{k}$.

We will prove the claim using induction on ℓ and i. The $\ell = 1$ or $i = 1$ cases is trivial.

To prove the general case, let B be an optimal ℓ-segmentation covering $[1, i]$, and let b be the starting point of the last segment of B. We consider two cases.

Case (i): Assume that $b \leq m_i$, then according to Lemma 5.2, there is $a \in A_i$ for which

$$s[a, \ell - 1] + p(a, i) \leq s[b, \ell - 1] + p(b, i) + \delta_{i-1} \quad . \qquad (5)$$

Recall that $\delta_{i-1} = s[i - 1, \ell]\frac{\epsilon}{k+\ell\epsilon}$, due to Line 11 in Alg. 2. Using the induction hypothesis on i, we can bound δ_{i-1},

$$\delta_{i-1} = s[i-1, \ell]\frac{\epsilon}{k+\ell\epsilon} \leq o[i-1, \ell]\left(1 + \epsilon\frac{\ell}{k}\right)\frac{\epsilon}{k+\ell\epsilon}$$
$$= o[i-1, \ell]\frac{\epsilon}{k} \leq o[i, \ell]\frac{\epsilon}{k} \quad . \qquad (6)$$

We can now combine the previous inequalities and the induction hypothesis on ℓ, which gives us

$$\begin{aligned}
s[i, \ell] &\leq \min_{x \in A_i} s[x, \ell - 1] + p(x, i) && \text{(Line 5 in Alg. 2)} \\
&\leq s[a, \ell - 1] + p(a, i) && (a \in A_i) \\
&\leq s[b, \ell - 1] + p(b, i) + o[i, \ell]\frac{\epsilon}{k} && \text{(Eqs. 5–6)} \\
&\leq o[b, \ell - 1]\gamma + p(b, i) + o[i, \ell]\frac{\epsilon}{k} && \text{(induction)} \\
&\leq o[b, \ell - 1]\gamma + p(b, i)\gamma + o[i, \ell]\frac{\epsilon}{k} && (\gamma \geq 1) \\
&= o[i, \ell]\gamma + o[i, \ell]\frac{\epsilon}{k} = o[i, \ell]\left(1 + \frac{\epsilon\ell}{k}\right) && .
\end{aligned}$$

Case (ii): Assume that $b > m_i$. If $b \in A_i'$, then

$$s[i, \ell] = \min_{x \in A_i'} s[x, \ell-1] + p(x, i) \leq s[b, \ell-1] + p(b, i) \quad . \quad (7)$$

Assume $b \notin A_i'$. This is only possible if the second condition in the while-loop failed, that is, there is $a \leq b$ with

$$s[i, \ell] \leq s[a, \ell - 1] \leq s[b, \ell - 1] \leq s[b, \ell - 1] + p(b, i) \quad . \quad (8)$$

7

In both cases, the induction hypothesis on ℓ gives us

$$s[i, \ell] \le s[b, \ell - 1] + p(b, i) \qquad \text{(Eqs. 7–8)}$$

$$\le o[b, \ell - 1]\gamma + p(b, i) \qquad \text{(induction)}$$

$$\le o[b, \ell - 1]\gamma + p(b, i)\,\gamma \qquad (\gamma \ge 1)$$

$$= o[i, \ell]\gamma \le o[i, \ell]\left(1 + \frac{\epsilon \ell}{k}\right) \quad .$$

This proves the induction step and the proposition. $\qquad \square$

Finally, let us prove the running time of ALL-DP.

Proposition 5.2. ALL-DP(k, ϵ) *needs* $\mathcal{O}\!\left(\frac{k^2}{\epsilon}n\right)$ *time.*

We will adopt the same notation as with the proof of Proposition 5.1. For simplicity, we also define $s[0, \ell] = 0$ for any $\ell = 1, \ldots, k$.

To prove the result we need two lemmas. The first lemma will be used to prove the second lemma.

Lemma 5.3. $s[m_i, \ell - 1] \le s[i - 1, \ell]$, *where* $m_i = \max A_i$.

Proof. We prove the lemma using induction on i. The case $i = 1$ is trivial since $A_1 = \{1\}$ and $s[1, \ell - 1] = (\ell - 1)p(1, 1) = 0 = s[0, \ell]$.

Assume that the claim holds for $i - 1$. Now, Lemma 5.1 guarantees that $s[m_{i-1}, \ell - 1] \le s[i - 2, \ell] \le s[i - 1, \ell]$. The while-loop condition in ALL-DP guarantees that $s[m, \ell - 1] \le s[i - 1, \ell]$, for $m = \max A'_{i-1}$. Since SPARSIFY never deletes the last element, we have $m = m_i$, which proves the lemma. $\qquad \square$

Next, we bound the number of items in A_i.

Lemma 5.4. $|A_i| \le 2 + 2(k + \ell \epsilon)/\epsilon \in \mathcal{O}(k/\epsilon)$.

Proof. Let $r = \lfloor (|A_i| + 1)/2 \rfloor$ be the number of entries in A_i with odd indices. Due to SPARSIFY, A_i cannot contain two items, say a_j and a_{j+2}, such that $s[a_{j+2}, \ell - 1] \le s[a_j, \ell - 1] + \delta_{i-1}$. Using this inequality on entries with odd incides we conclude that $s[m_i, \ell - 1] \ge (r - 1)\delta_{i-1}$. Lemma 5.3 allows us to bound r with

$$r - 1 \le \frac{s[m_i, \ell - 1]}{\delta_{i-1}} \le \frac{s[i - 1, \ell]}{\delta_{i-1}} = \frac{k + \ell \epsilon}{\epsilon} \in \mathcal{O}\!\left(\frac{k}{\epsilon}\right) \quad .$$

Since $|A_i| \le 2r$, the lemma follows. $\qquad \square$

We can now prove the main claim.

Proof of Proposition 5.2. Computing $s[i, \ell]$ in ALL-DP requires at most $|A'_i|$ comparisons, and SPARSIFY requires at most $|A'_i|$ while-loop iterations. We have $|A'_i| = |A_i| + m_{i+1} - m_i$. For a fixed ℓ, Lemma 5.4 implies that the total number of comparisons is

$$\sum_{i=1}^{n} |A'_i| = \sum_{i=1}^{n} |A_i| + m_{i+1} - m_i$$

$$\le n + \sum_{i=1}^{n} |A_i| \in \mathcal{O}\!\left(\frac{nk}{\epsilon}\right) \quad .$$

The result follows since $\ell = 1, \ldots, k$. $\qquad \square$

The idea behind this approach is similar to the algorithm suggested by Guha et al. [8]. The main difference is how the candidate list is formed: Guha et al. [8] constructed the candidate list by only adding new entries to it, whereas we are also deleting the values allowing us to keep A much smaller.

6. A linear-time algorithm for cumulative maximum segmentation

In previous section, we presented a technique for computing a cumulative segmentation. Our final contribution is a fast algorithm to solve cumulative version of the maximum segmentation problem MAXSEG.

Problem 6.1 (ALLMAXSEG). *Given a penalty function* p *and an integer* k, *find an* ℓ-*segmentation* B *covering* $[1, i]$ *that minimizes*

$$p_{max}(B) = \max_{1 \le j \le \ell} p(b_{j-1}, b_j),$$

for every $i = 1, \ldots, n$ *and every* $\ell = 1, \ldots, k$.

We should point out that we can apply the algorithm by Guha and Shim [7] used to solve MAXSEG for every i and ℓ, which would give us the computational complexity of $\mathcal{O}\!\left(nk^3 \log^2 n\right)$. Alternatively, we can use an algorithm

8

Algorithm 4: ALL-MS(k), solves ALLMAXSEG.

1 $b_i \leftarrow 1$, for $i = 0, \ldots, k$;

2 $b_{k+1} \leftarrow n$;　　　　{sentinel, discarded in the end}

3 $\tau \leftarrow 0$;

4 $s[1, \ell] \leftarrow 0$, for $\ell = 1, \ldots, k$;

5 while $b_1 \leq n$ **do**

6　　$\ell \leftarrow \arg \min\limits_{1 \leq j \leq k} \{ p(b_{j-1}, b_j + 1) \mid b_j < b_{j+1} \}$;

7　　$b_\ell \leftarrow b_\ell + 1$;

8　　$\tau \leftarrow \max(\tau, p(b_{\ell-1}, b_\ell))$;

9　　$s[b_\ell, \ell] \leftarrow \tau$;

10 return s;

---

given by Guha et al. [9] that provides us with the computational complexity of $\mathcal{O}(kn \log n)$.[6] We will present a faster algorithm, running in $\mathcal{O}(nk \log k)$ time, making the algorithm linear-time with respect to n.

Our algorihm works as follows (see Algorithm 4): We start with a k-segmentation $B = (b_0 = 1, \ldots, b_k = 1)$. It turns out that we can choose b_ℓ so that an ℓ-segmentation, say $B' = (b'_0 = b_0, \ldots, b'_{\ell-1} = b_{\ell-1}, b'_\ell = b_\ell + 1)$ is an optimal ℓ-segmentation covering $[1, b'_\ell]$. We increase value of b_ℓ by 1, and repeat until there are no indices that cannot be moved right, that is, $b_1 = n$. At this point, every b_ℓ has visited every integer between 1 and n, so we have discovered all optimal segmentations.

To guarantee that $(b'_0, \ldots, b'_\ell)$ is optimal, we need to select ℓ carefully. Here, we choose ℓ to be the index minimizing $p(b_{\ell-1}, b_\ell + 1)$. The main reason for choosing such a value comes from the next lemma.

Lemma 6.1. *Assume an ℓ-segmentation $B = b_0, \ldots, b_\ell$ covering $[1, i]$. Let τ be such that $\tau \leq p(b_{c-1}, b_c + 1)$, for every $c = 1, \ldots, \ell$. Let B' be an ℓ-segmentation covering $[1, j]$ with $j > i$. Then $p_{max}(B') \geq \tau$.*

Proof. Assume that $p_{max}(B') < \tau$. We claim that $b'_c \leq b_c$ for every $c = 1, \ldots, \ell$, and this claim leads to $j = b'_\ell \leq b_\ell = i$, which is a contradiction.

We will prove the claim by induction. To prove the induction base, note that $b'_1 \leq b_1$, as otherwise $p(b'_0, b'_1) \geq p(b_0, b_1 + 1) \geq \tau$. To prove the induction step, assume that $b'_{c-1} \leq b_{c-1}$. Then, $b'_c \leq b_c$, as otherwise $p(b'_{c-1}, b'_c) \geq p(b_{c-1}, b_c + 1) \geq \tau$. □

The lemma can be used as follows: Assume a current k-segmentation $B = (b_0 = 1, \ldots, b_k = 1)$, and let ℓ be the index minimizing $\tau = p(b_{\ell-1}, b_\ell + 1)$. Let $B' = (b'_0 = b_0, \ldots, b'_{\ell-1} = b_{\ell-1}, b'_\ell = b_\ell + 1)$ be the resulting ℓ-segmentation. Assume that $p_{max}(B') = \tau$. Lemma 6.1 implies that there is no segmentation with cost smaller than τ covering $[1, b'_\ell]$. This makes automatically B' optimal. The proof for the case $p_{max}(B') < \tau$ is more intricate, and it is handled in the next proposition.

Proposition 6.1. ALL-MS(k) *solves* ALLMAXSEG.

Proof. Define τ_i to be the value of τ at the end of the ith iteration. Let also $B^i = b^i_1, \ldots, b^i_k$ be the values of $b_1, \ldots, b_k$ at the end of the ith iteration.

Fix i and assume that during the ith iteration we updated b_ℓ. Let τ^* be the optimal cost for ℓ-segmentation covering $[1, b^i_\ell]$. Since $b^i_0, \ldots, b^i_\ell$ covers $[1, b^i_\ell]$ and the cost of individual segments is bounded by τ_i, we have $\tau^* \leq \tau_i$.

To prove the optimality of $b^i_0, \ldots, b^i_\ell$, we need to show that $\tau^* \geq \tau_i$. Let j be the largest index such that $\tau_j < \tau_i$. If such value does not exist, then $\tau_i = 0 \leq \tau^*$. Clearly, $j < i$. Assume that we update $b_{\ell'}$ during the $(j+1)$th iteration. Then, by definition of ℓ',

$$p\left(b^j_{c-1}, b^j_c + 1\right) \geq p\left(b^j_{\ell'-1}, b^j_{\ell'} + 1\right) = \tau_{j+1} = \tau_i,$$

for any $c = 1, \ldots, \ell$. Since $b^j_\ell \leq b^{i-1}_\ell < b^i_\ell$, Lemma 6.1 implies that an ℓ-segmentation covering $[1, b^i_\ell]$ must have a cost of a least τ_i. This proves the proposition. □

We finish with the computational complexity analysis.

9

Proposition 6.2. ALL-MS *runs in $\mathcal{O}(nk \log k)$ time.*

Proof. Since b_j can move only to the right, the while-loop is evaluated at most $\mathcal{O}(kn)$ times. Computing j at each iteration can be done by maintaining a priority queue of size $\mathcal{O}(k)$. After updating b_j, updating the queue can be done in $\mathcal{O}(\log k)$ time. $\qquad\square$

References

[1] A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1-4):195–208, 1987.

[2] R. Bellman. On the approximation of curves by line segments using dynamic programming. *Communications of the ACM*, 4 (6):284–284, 1961.

[3] P. Bernaola-Galván, R. Román-Roldán, and J. L. Oliver. 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, 1996.

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

[5] R. Fleischer, M. J. Golin, and Y. Zhang. Online maintenance of k-medians and k-covers on a line. *Algorithmica*, 45(4):549–567, 2006.

[6] Z. Galil and K. Park. A linear-time algorithm for concave one-dimensional dynamic programming. *Information Processing Letters*, 33(6):309–311, 1990.

[7] S. Guha and K. Shim. A note on linear time algorithms for maximum error histograms. *IEEE Transactions on Knowledge and Data Engineering*, 19(7):993–997, 2007.

[8] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 471–475, 2001.

[9] S. Guha, K. Shim, and J. Woo. REHIST: relative error histogram construction algorithms. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, pages 300–311, 2004.

[10] S. Guha, N. Koudas, and K. Shim. Approximation and streaming algorithms for histogram construction problems. *ACM Transactions of Database Systems*, 31(1):396–438, 2006.

[11] R. Hassin and A. Tamir. Improved complexity bounds for location problems on the real line. *Operations Research Letters*, 10 (7):395–402, 1991.

[12] J. Himberg, K. Korpiaho, H. Mannila, J. Tikanmäki, and H. Toivonen. Time series segmentation for context recognition in mobile devices. In *Proceedings of 1st IEEE International Conference on Data Mining (ICDM)*, pages 203–210, 2001.

[13] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *SIGMOD Record*, 30(2):151–162, 2001.

[14] V. Lavrenko, M. Schmill, D. Lawrie, P. Ogilvie, D. Jensen, and J. Allan. Mining of concurrent text and time series. In *KDD-2000 Workshop on Text Mining*, pages 37–44, 2000.

[15] W. Li. DNA segmentation as a model selection process. In *Proceedings of the 5th annual international conference on Computational biology (RECOMB)*, pages 204–210. ACM, 2001.

[16] T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *Proceedings of 20th International Conference on Data Engineering (ICDE)*, pages 339–349, 2004.

[17] M. Salmenkivi, J. Kere, and H. Mannila. Genome segmentation using piecewise constant intensity models and reversible jump MCMC. *Bioinformatics*, 18(Suppl 2):S211–S218, 2002.

[18] H. Shatkay and S. B. Zdonik. Approximate queries and representations for large data sequences. In *Proceedings of 12th International Conference on Data Engineering (ICDE)*, pages 536–545, 1996.

[19] N. Tatti. Fast sequence segmentation using log-linear models. *Data Mining Knowledge Discovery*, 27(3):421–441, 2013.

[20] E. Terzi and P. Tsaparas. Efficient algorithms for sequence segmentation. In *Proceedings of the 6th SIAM International Conference on Data Mining (SDM)*, pages 316–327, 2006.

Supplementary material for strongly polynomial efficient approximation scheme for segmentation

Nikolaj Tatti

F-Secure, HIIT, Aalto University, Finland

Abstract

In this supplementary material we revisit an algorithm proposed by Guha and Shim [1], and show that this algorithm can be used to solve the maximum segmentation problem.

1. Maximum segmentation

Problem 1.1 (MAXSEG). *Given a penalty p and an integer k, find a k-segmentation B covering $[1, n]$ that minimizes*

$$p_{max}(B) = \max_{1 \leq j \leq k} p(b_{j-1}, b_j) \quad .$$

This optimization problem can be solved with an algorithm given by Guha and Shim [1]. This algorithm is designed to solve an instance of MAXSEG, where the penalty function is L_∞ is

$$p(a, b) = \min_\mu \sum_{i=a}^{b-1} |x_i - \mu| \quad .$$

Luckily, the same algorithm and the proof of correctness, see Lemma 3 in [1], is valid for any monotonic penalty.

For the sake of completeness we present this algorithm in Algorithms 1–2.

The main idea is based on the following observation. Let B be the optimal segmentation with the cost of $p_{max}(B) = \tau$. Then $p(b_0, b_1) = \tau$ or $p_{max}(B') = \tau$, where $B' = b_1, \ldots, b_k$ is a $(k-1)$-segmentation covering $[b_1, n]$. In the first case, we can safely assume that $b_1 = c$, where c is the smallest index for which there is a $(k-1)$-segmentation covering $[b_1, n]$ with a maximum cost of $p(b_0, b_1)$. In the second case, we can safely assume that $b_1 = c - 1$, that is, the largest index for which there is *no* $(k-1)$-segmentation covering $[b_1, n]$ with a maximum cost of $p(b_0, b_1)$.

This gives rise to the main loop: compute c, and record $\Delta_1 = p(b_0, c)$, then recurse and discover the best $(k-1)$ segmentation covering $[c-1, n]$, with a cost of, say, Δ_2. Then, the correct cost is $\min(\Delta_1, \Delta_2)$. For the complete proof of correctness see Lemma 3 in [1].[1]

For completeness, we provide the proof of correctness. In order to do so, let us first define

$$f(b; i, k) = [\text{GREEDY}(b, k-1, p(i, b)) = n],$$

Email address: `nikolaj.tatti@aalto.fi` (Nikolaj Tatti)

[1]In pseudo-code given in [1], an incorrect step, $i \leftarrow c$, is used. However, in the proof of correctness, Lemma 3, correct value is used.

---

returning true or false depending whether the statement inside the brackets is valid.

Algorithm 1: GREEDY(b, k, τ) computes the largest index that can be reached with a k-segmentation starting from b with a cost $p_{max} \leq \tau$.

1 foreach $\ell = 1, \ldots, k$ **do**
2 $b \leftarrow \max_j \{b \leq j \leq n \mid p(b, j) \leq \tau\}$;
                                {use binary search}
3 return b;

Algorithm 2: MS-FAST(k), computes the cost of k-segmentation solving MAXSEG.

1 $\Delta \leftarrow \infty$;
2 $i \leftarrow 1$;
3 foreach $\ell = k, \ldots, 1$ **do**
4 $c \leftarrow \min \{b \geq i \mid \text{GREEDY}(b, \ell - 1, p(i, b)) = n\}$;
                                {use binary search}
5 $\Delta \leftarrow \min(\Delta, p(i, c))$;
6   **if** $i = c$ **then return** Δ;
7 $i \leftarrow c - 1$;
8 return Δ;

Proposition 1.1. *$f(b; i, k)$ returns true if and only if there is a k-segmentation B covering $[i, n]$ with $p_{max}(B) \leq p(i, b)$.*

Proof. The only if direction is trivial.

To prove the if direction, assume that B satisfies the conditions of the proposition, and let C be the segmentation constructed by GREEDY$(b, k-1, p(i, b))$. Note that $b_1 \leq c_1$. If $b_1 < c_1$, then we can move b_1 to the right, without violating the conditions. Thus, we can safely assume that $b_1 = c_1$. By doing this recursively, we can safely assume that $b_i = c_i$. This guarantees that f returns true. □

Proposition 1.2. MS-FAST *returns the cost of the optimal segmentation.*

Proof. Let us write i_ℓ, c_ℓ to be the values of variables during the foreach-loop of MS-FAST. Note that ℓ goes from k to 1, or terminated early.

Write $\tau_\ell = p(i_\ell, c_\ell)$. Define C_ℓ to be the segmentation discovered by GREEDY$(c_\ell, \ell - 1, p(i_\ell, c_\ell))$ preceding by the segment (i_ℓ, c_ℓ).

Let B_ℓ be the optimal segmentation covering $[i_\ell, n]$ with the cost of ρ_ℓ. Define C'_ℓ to be $B_{\ell-1}$ preceding by the segment $(i_\ell, c_\ell - 1) = (i_\ell, i_{\ell-1})$.

By definition of c_ℓ, $\tau_\ell = p(i_\ell, c_\ell) = p_{max}(C_\ell)$. Since C_ℓ covers $[i_\ell, n]$, we also have

$$\tau_\ell = p(i_\ell, c_\ell) = p_{max}(C_\ell) \geq \rho_\ell \quad .$$

By the minimality of c_ℓ, $p(i_\ell, c_\ell - 1) < p_{max}(B_{\ell-1})$, and since C'_ℓ covers $[i_\ell, n]$, we also have

$$\rho_{\ell-1} = p_{max}(B_{\ell-1}) = p_{max}(C'_\ell) \geq \rho_\ell \quad .$$

Assume that the for-loop is terminated early due to the $i_\ell = c_\ell$ condition. Then $0 = \tau_\ell \geq \rho_\ell \geq \rho_k$, that is, there is a k-segmentation covering $[1, n]$ with zero cost. Since the algorithm outputs 0, this proves the special case, so we can safely assume that the for-loop is not terminated early.

Define $\eta_\ell = \min_{j \leq \ell} \tau_j$. Note that η_k is the output of the algorithm, so to prove the result, we claim that $\eta_\ell = \rho_\ell$ which we will prove by induction on ℓ. The case $\ell = 1$ is trivial.

Fix ℓ and let b be the ending point of the first segment in B_ℓ. We consider two cases.

Assume that $f(b; i_\ell, \ell)$ is true. Then we must have $c_\ell \leq b$. Thus, $\tau_\ell = p(i_\ell, c_\ell) \leq p(i_\ell, b) \leq \rho_\ell$. So, $\tau_\ell = \rho_\ell$. The induction hypothesis states that $\eta_{\ell-1} = \rho_{\ell-1} \geq \rho_\ell$. Hence, $\eta_\ell = \min(\eta_{\ell-1}, \tau_\ell) = \rho_\ell$.

Assume that $f(b; i_\ell, \ell)$ is false. Then we must have $c - 1 \geq b$. In other words, $B_{\ell-1}$ need to cover less than the remaining segments of B_ℓ. Thus $\rho_{\ell-1} \leq \rho_\ell$. Since, $\tau_\ell \geq \rho_\ell$ and $\rho_{\ell-1} \geq \rho_\ell$, the induction hypothesis implies that

$$\eta_\ell = \min(\tau_\ell, \eta_{\ell-1}) = \min(\tau_\ell, \rho_{\ell-1}) = \min(\tau_\ell, \rho_\ell) = \rho_\ell,$$

this proves the induction step. $\qquad \square$

To show the computational complexity, note that for a fixed ℓ we need $\mathcal{O}(\log n)$ evaluations of GREEDY to compute c, each evaluation requiring $\mathcal{O}(k \log n)$ time. Thus, computing a single c requires $\mathcal{O}(k \log^2 n)$. We need to repeat this $\mathcal{O}(k)$ times, which gives us a total running time of $\mathcal{O}(k^2 \log^2 n)$.

References

[1] S. Guha and K. Shim. A note on linear time algorithms for maximum error histograms. *IEEE Transactions on Knowledge and Data Engineering*, 19(7):993–997, 2007.