Segmentations with Rearrangements

Aristides Gionis[*]
Yahoo! Research,
Barcelona, Spain
gionis@yahoo-inc.com

Evimaria Terzi
Basic Research Unit, HIIT
University of Helsinki, Finland
terzi@cs.helsinki.fi

January 26, 2007

Abstract

Sequence segmentation is a central problem in the analysis of sequential and time-series data. In this paper we introduce and we study a novel variation to the segmentation problem: in addition to partitioning the sequence we also seek to apply a limited amount of reordering, so that the overall representation error is minimized. Our problem formulation has applications in segmenting data collected from a sensor network where some of the sensors might be slightly out of sync, or in the analysis of newsfeed data where news reports on a few different topics are arriving in an interleaved manner. We formulate the problem of segmentation with rearrangements and we show that it is an NP-hard problem to solve or even approximate. We then proceed to devise effective algorithms for the proposed problem, combining ideas from linear programming, dynamic programming, and outlier-detection algorithms in sequences. We perform extensive experimental evaluation on synthetic and real datasets that demonstrates the efficacy of the suggested algorithms.

1 Introduction

A central problem related to time-series analysis is the construction of a compressed and concise representation of the data, so that the time-series data can be handled efficiently. One commonly used such representation is the *piecewise-constant* approximation. A piecewise-constant representation approximates a time series T of length n using k non-overlapping and contiguous segments that span the whole sequence. Each segment is represented by a single (constant) point, e.g., the mean of the points in the segment. We call mean this point the *representative* of the segment, since it represents all the points in the segment. The error in this approximate representation is measured using some *error function*, usually the sum of squares of the differences between all points in the segment and their representative point. Different error functions may be used depending on the application. For a given error function, the goal is to find the segmentation of the sequence into k segments and the corresponding k representatives so that the error in the representation of the underlying data is minimized. We call this problem the SEGMENTATION problem. Segmentation problems, particularly for multivariate time series, arise in many data mining applications, including bioinformatics, environmental sciences, weather prediction, and context recognition in mobile devices.

The input data for time-series segmentation problems is usually a sequence T consisting of n points $\{t_1, \ldots, t_n\}$ with $t_i \in \mathbb{R}^d$. The majority of the related work primarily focuses on finding a segmentation S of T taking for granted the order of the points in T. That is, it is assumed that the correct order of the points coincides with the input order. In this paper we assume that the order of the points in T is only *approximately* correct. We thus consider the case where the points need a "gentle" rearrangement (reordering). This reordering will result in another sequence T', which consists of the same points as T. Our focus is to find the rearrangement of the points such that the segmentation error of the reordered sequence T' is minimized. We call this alternative formulation of the segmentation problem SEGMENTATION WITH REARRANGEMENTS.

In this paper we formally define the SEGMENTATION WITH REARRANGEMENTS problem and some of its variants. We show that under several conditions the problem is NP-hard and, even worse, hard to approximate. Furthermore, we devise heuristics that are intuitive and work well in practice.

The following example motivates the SEGMENTATION WITH REARRANGEMENTS problem. Consider a setting where there are many sensors, located at different places, reporting their measurements to a central server. Assume that at all time points the sensors are functioning properly and they send correct and useful data to the server. However, due to communication delays or malfunctions in the network, the data points do not arrive to the server in the right order. In such a case, if the segmentation algorithm (or more general any data analysis algorithm employed) takes the data

[*] Work done while the author was with Basic Research Unit, HIIT, University of Helsinki, Finland.

order for granted it may produce misleading results. On the other hand, choosing to ignore data points that seem to be incompatible with their neighbors leads to omitting possibly valuable information. Omitting some data points is, of course, a type of correction, however, in this paper we take the approach of correcting the data by reordering instead of omitting data points.

The rest of the paper is organized as follows. We start by reviewing the related work in Section 2. In Section 3 we introduce our notation and we give the basic problem definition. We discuss the complexity of the problem in Section 4, and we present our algorithms in Section 5. We discuss the experimental evaluation of our algorithms in Section 6, and finally, Section 7 is a short conclusion.

2 Related work

The work presented in this paper is closely related to the literature on segmentation algorithms. Despite the fact that the basic SEGMENTATION problem can be solved optimally in polynomial time, the quadratic performance of the optimal algorithm makes the use of the algorithm prohibitive for large datasets. For this reason several suboptimal, though more efficient algorithms have been proposed in the literature. Some of them have provable approximation bounds [8, 18], while others have proved to work very well in practice [5, 11, 14, 13, 17]. Variants of the basic SEGMENTATION problem have been studied in different contexts, for example [2, 7, 9]. The main idea in these cases is to find a segmentation of the input sequence into k segments, subject to some constraints imposed on the output representatives. Finally, reordering techniques over the dimensions of multidimensional data have been proposed in [20]. The goal of this technique is mainly to improve the performance of indexing structures that allow for more efficient answering of similarity (e.g., k-NN queries).

The SEGMENTATION WITH REARRANGEMENTS problem is a classical segmentation problem, where all the points appearing in the sequence are assumed to have the correct values, but the structure of the sequence itself may be erroneous. Thus, in this problem we are given the additional "freedom" to move points around in order to improve the segmentation error. To the best of our knowledge, the problem of SEGMENTATION WITH REARRANGEMENTS has not been studied in the literature. Slightly related is the work on identifying "unexpected" or surprising behavior of the data used for various data-mining tasks. The mainstream approaches for such problems find the data points that exhibit surprising or unexpected behavior and remove them from the dataset. These points are usually called *outliers*, and their removal from the dataset allows for a cheapest (in terms of model cost) and more concise representation of the data. For example, [12, 16] study the problem of finding outliers (or *deviants*) in time-series data. More specifically, the goal is to find the best set of deviants that if removed from the dataset, the histogram built using the rest of the points has the smallest possible error. Although our problem definition is different from the one presented in [12, 16] we use some of their techniques in our methodology. Similarly, the problem of finding outliers in order to improve the results of clustering algorithms has been studied in [3].

3 Problem formulation

Let $T = (t_1, t_2, \ldots, t_n)$ be a d-dimensional sequence of length n with $t_i \in \mathbb{R}^d$, i.e., $t_i = (t_{i1}, t_{i2}, \ldots, t_{id})$.

A *k-segmentation* S of a sequence of length n is a partition of $\{1, 2, \ldots, n\}$ into k non-overlapping contiguous subsequences (segments), $S = \{s_1, s_2, \ldots, s_k\}$. Each segment s_i consists of $|s_i|$ points. The representation of sequence T when segmentation S is applied to it, collapses the values of the sequence within each segment s into a single value μ_s (e.g., the mean). We call this value the *representative* of the segment, and each point $t \in s$ is "represented" by the value μ_s. Collapsing points into representatives results in less accuracy in the sequence representation. We measure this loss in accuracy using the error function E_p, defined as

$$E_p(T,S) = \left(\sum_{s\in S}\sum_{t\in s}|t-\mu_s|^p\right)^{\frac{1}{p}}, \tag{3.1}$$

where T denotes the sequence and S the segmentation. We consider the cases where $p = 1, 2$. For simplicity, we will sometimes write $E_p(S)$ instead of $E_p(T, S)$, when the sequence T is implied.

The SEGMENTATION problem asks for the segmentation that minimizes the error E_p. The optimal representative of each segment depends on p. For $p = 1$ the optimal representative for each segment is the median of the points in the segment; for $p = 2$ the optimal representative of the points in a segment is their mean.

3.1 The SEGMENTATION problem

We now give a formal definition of the SEGMENTATION problem, and we describe the optimal algorithm for solving it. Let $\mathcal{S}_{n,k}$ denote the set of all k-segmentations of sequences of length n. For some sequence T, and for error measure E_p, we define the optimal segmentation as

$$S_{\text{opt}}(T,k) = \arg\min_{S\in\mathcal{S}_{n,k}} E_p(T,S).$$

That is, S_{opt} is the k-segmentation S that minimizes the $E_p(T, S)$. For a given sequence T of length n the formal definition of the k-segmentation problem is the following:

PROBLEM 1. (SEGMENTATION) *Given a sequence T of length n, an integer value k, and the error function E_p, find $S_{opt}(T, k)$.*

Problem 1 is known to be solvable in polynomial time [1]. The solution consists of a standard dynamic-programming (DP) algorithm that runs in time $O(n^2k)$. The main recurrence of the dynamic-programming algorithm is the following:

$$(3.2)\quad E_p(S_{\text{opt}}(T\left[1\dots n\right], k)) = \\ \min_{j<n} \quad \{E_p\left(S_{\text{opt}}(T\left[1\dots j\right], k-1)\right) \\ + E_p\left(S_{\text{opt}}(T\left[j+1,\dots,n\right], 1)\right)\},$$

where $T\left[i\dots,j\right]$ denotes the subsequence of T that contains all points in positions from i to j, with i, j included.

3.2 The Segmentation with Rearrangements problem

We now define the Segmentation with Rearrangements problem. Assume again an input sequence $T = \{t_1, \dots, t_n\}$. We associate the input sequence with the identity permutation τ i.e., the i-th observation is positioned in the i-th position in the sequence. Our goal is to find another permutation π of the data points in T. There are many possible ways to permute the points of the initial sequence T. Here we allow two types of rearrangements of the points, namely, *bubble-sort swaps* (or simply *swaps*) and *moves*.

- Bubble-Sort Swaps: A bubble-sort swap $\mathcal{B}(i, i+1)$ when applied to sequence $T = \{t_1, \dots, t_n\}$ causes the following rearrangement of the elements of T: $\mathcal{B}(i, i+1) \circ T = \{t_1, \dots, t_{i+1}, t_i, t_{i+2}, t_n\}$.
- Moves: A move corresponds to a single-element transposition [10]. That is, a move $\mathcal{M}(i \to j)$ $(i < j)$ when applied to sequence $T = \{t_1, \dots, t_n\}$ causes the following rearrangement of the elements in T: $\mathcal{M}(i \to j) \circ T = \{t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_i, t_j, t_{j+1}, \dots, t_n\}$.

We usually apply a series of swaps or moves to the initial sequence. We denote by $\overline{\mathcal{B}}$ such a sequence of bubble-sort swaps and by $\overline{\mathcal{M}}$ a sequence of single-element transpositions. When a sequence of swaps (or moves) is applied to the input sequence T we obtain a new sequence $T_{\overline{\mathcal{B}}} = \overline{\mathcal{B}} \circ T$ (or $T_{\overline{\mathcal{M}}} = \overline{\mathcal{M}} \circ T$). Finally we denote by $|\overline{\mathcal{B}}|$ (or $|\overline{\mathcal{M}}|$) the number of bubble-sort swaps (or moves) included in the sequence $\overline{\mathcal{B}}$ (or $\overline{\mathcal{M}}$).

We use the generic term *operation* to refer to either swaps or moves. The transformation of the input sequence T using a series of operations $\overline{O}$ (all of the same type) is denoted by $\overline{O} \circ T = T_{\overline{O}}$. Given the above notational conventions, we are now ready to define the generic Segmentation with Rearrangements problem.

Problem 2. (Segmentation with Rearrangements) *Given sequence T of length n, integer values C and k, and error function E, find a sequence of operations $\overline{O}$ such that*

$$\overline{O} = \arg\min_{\overline{O}'} E\left(S_{opt}\left(T_{\overline{O}'}, k\right)\right),$$

with the restriction that $|\overline{O}| \leq C$.

When the operations are restricted to bubble-sort swaps the corresponding segmentation problem is called Segmentation with Swaps, while when the operations are restricted to moves, we call the corresponding segmentation problem Segmentation with Moves.

We mainly focus on the E_p error measure, and particularly we are interested in the cases where $p = 1, 2$ (see Equation (3.1)).

4 Problem complexity

Although the basic k-segmentation problem (Problem 1) is solvable in polynomial time the alternative formulations we study in this paper are NP-hard.

We first argue the NP-hardness of the Segmentation with Swaps and Segmentation with Moves for the case of error function E_p with $p = 1, 2$ and for sequence $T = \{t_1, \dots, t_n\}$ with $t_i \in \mathbb{R}^d$ and $d \geq 2$. Consider for example, the Segmentation with Swaps problem with $C = n^2$. This value for C allows us to move any point freely to arbitrary position, irrespective to its initial location. Thus the Segmentation with Swaps problem with $C = n^2$ is the well-studied *clustering problem*. For $p = 1$ it is the Euclidean k-median problem, and for $p = 2$ it is the k-means problem of finding k points such that the sum of distances to the closest point is minimized. Both these problems can be solved in polynomial time for 1-dimensional data [15], and both are NP-hard for dimensions $d \geq 2$ [6]. Similar observation holds for the Segmentation with Moves problem when setting $C = n$. In this case again the Segmentation with Moves problem becomes equivalent to the k-median (for $p = 1$) and the k-means (for $p = 2$) clustering problems. Similar arguments show that the windowed versions of both these problems are NP-hard.

Lemma 4.1. *The* Segmentation with Swaps *and the* Segmentation with Moves *problems are NP-hard, for $p = 1, 2$ and dimensionality $d \geq 2$.*

Proof. Assume the Segmentation with Swaps problem with $C \geq n^2$. In this case, the budget C on the number of swaps allows us to move every point in any position in the sequence. That is, the initial ordering of the points is indifferent. Then, the Segmentation with Swaps problem becomes equivalent to clustering into k clusters. Thus, for $C \geq n^2$ and $p = 1$, solving Segmentation with Swaps would be equivalent to solving the k-median clustering problem. Similarly, for $p = 2$ the problem of Segmentation with Swaps becomes equivalent to k-means clustering.

The NP-hardness proof of the Segmentation with Moves problem is identical. The only difference is that Segmentation with Moves becomes equivalent to clustering when our budget is $C \geq n$. □

We now turn further our attention to the SEGMENTATION WITH SWAPS problem and we study its complexity for $d = 1$. We have the following lemma.

LEMMA 4.2. *For error function E_p, with $p = 1, 2$, the* SEGMENTATION WITH SWAPS *problem is NP-hard even for dimension $d = 1$.*

Proof. The result is by reduction from the GROUPING BY SWAPPING problem [6], which is stated as follows:

INSTANCE: Finite alphabet Σ, string $x \in \Sigma^*$, and a positive integer K.

QUESTION: Is there a sequence of K or fewer adjacent symbol interchanges that converts x into a string y in which all occurrences of each symbol $a \in \Sigma$ are in a single block, i.e., y has no subsequences of the form aba for $a, b \in \Sigma$ and $a \neq b$?

Now, if we can solve SEGMENTATION WITH SWAPS in polynomial time, then we can solve GROUPING BY SWAPPING in polynomial time as well. Assume string x input to the GROUPING BY SWAPPING problem. Create an one-to-one mapping f between the letters in Σ and a set of integers, such that each letter $a \in \Sigma$ is mapped to $f(a) \in \mathbb{N}$. Therefore, the input string $x = \{x_1, \ldots, x_n\}$ is transformed to an 1-dimensional integer sequence $f(x) = \{f(x_1), \ldots, f(x_n)\}$, such that each $f(x_i)$ is an 1-dimensional point. The question we ask is whether the algorithm for the SEGMENTATION WITH SWAPS can find a segmentation with error $E_p = 0$ by doing at most K swaps. If the answer is "yes", then the answer to the GROUPING BY SWAPPING problem is also "yes" and vice versa. □

A corollary of the above lemma is that we cannot hope for an approximation algorithm for the SEGMENTATION WITH SWAPS problem with bounded approximation ratio. Lets denote by $E^{\mathcal{A}}$ the error induced by an approximation algorithm $\mathcal{A}$ of the SEGMENTATION WITH SWAPS problem and by E^* the error of the optimal solution to the same problem. The following corollary shows that there does not exist an approximation algorithm for the SEGMENTATION WITH SWAPS problem such that $E^{\mathcal{A}} \leq \alpha \cdot E^*$ for any $\alpha > 1$, unless $P = NP$.

COROLLARY 4.1. *There is no approximation algorithm $\mathcal{A}$ for the* SEGMENTATION WITH SWAPS *problem such that $E_p^{\mathcal{A}} \leq \alpha \cdot E_p^*$, for $p = 1, 2$ and $\alpha > 1$, unless* P = NP.

Proof. We will prove the corollary by contradiction. Assume that an approximation algorithm $\mathcal{A}$ with approximation factor $\alpha > 1$ exists for the SEGMENTATION WITH SWAPS problem. This would mean that for every instance of the SEGMENTATION WITH SWAPS problem it holds that

$$E_p^{\mathcal{A}} \leq \alpha \cdot E_p^*.$$

Now consider again the proof of Lemma 4.2 and the GROUPING BY SWAPPING problem. Assume the string $x \in \Sigma^*$ the input to the GROUPING BY SWAPPING problem and let f be the one-to-one transformation of sequence x to the sequence of integers $f(x)$. The instance of the GROUPING BY SWAPPING problem with parameter K has an affirmative answer if and only if the SEGMENTATION WITH SWAPS problem has an affirmative answer for error $E_p = 0$ and $C = K$. This instance of the SEGMENTATION WITH SWAPS has error $E_p^* = 0$. Therefore, if we feed this instance to the approximation algorithm $\mathcal{A}$ it would output a solution with $E_p^{\mathcal{A}} = 0$ and thus using algorithm $\mathcal{A}$ we could decide the GROUPING BY SWAPPING problem. However, since GROUPING BY SWAPPING is NP-hard, the assumption of the existence of the polynomial-time approximation algorithm $\mathcal{A}$ is contradicted. □

5 Algorithms

In this section we give a description for our algorithmic approaches for the generic SEGMENTATION WITH REARRANGEMENTS problem. When necessary we focus our discussion to the specific rearrangement operations that we are considering, namely the moves and the swaps.

5.1 The SEGMENT&REARRANGE **algorithm** Since our problem is NP-hard, we propose algorithms that are suboptimal. The general algorithmic idea, which we will describe in this subsection and expand in the sequel, is summarized in Algorithm 1. We call this algorithm SEGMENT&REARRANGE and it consists of two steps. In the first step it fixes a segmentation of the sequence. This segmentation is the optimal segmentation of the input sequence T. Any segmentation algorithm can be used in this step including the optimal dynamic-programming algorithm (see recursion (3.2)), or any of the faster segmentation heuristics proposed in the literature. Once the segmentation S of T into k segments is fixed, the algorithm proceeds to the second step called the *rearrangement* step. Given input sequence T and its segmentation S, the goal of this step is to find a good set of rearrangements of points, so that the total segmentation error of the rearranged sequence is minimized. We call this subproblem the REARRANGEMENT problem. Note that the REARRANGEMENT problem assumes a fixed segmentation. Once the algorithm has decided upon the rearrangements that need to be made, it segments the rearranged sequence and outputs the obtained segmentation.

In the rest of our discussion we focus in developing a methodology for the REARRANGEMENT problem. Consider all segments of segmentation $S = \{s_1, \ldots, s_k\}$ as possible new locations for every point in T. Each point $t_j \in T$ is associated with a gain (or loss) p_{ij} that is incurred to the

Algorithm 1 The SEGMENT&REARRANGE algorithm.

```
Input: Sequence $T$ of $n$ points, number of segments $k$,
    number of operations $C$.
Ouput: A rearrangement of the points in $T$ and a segmentation of the new sequence into $k$ segments.
Segment: $S = S_{\text{opt}}(T, k)$
Rearrange: $\overline{O} = \text{rearrange}(T, S, C)$
Segment: $S' = S_{\text{opt}}(T_{\overline{O}}, k)$
```

	s_1	s_2	$\ldots$	s_k
t_1	$\langle w_{11}, p_{11} \rangle$	$\langle w_{21}, p_{21} \rangle$	$\ldots$	$\langle w_{k1}, p_{k1} \rangle$
t_2	$\langle w_{12}, p_{12} \rangle$	$\langle w_{22}, p_{22} \rangle$	$\ldots$	$\langle w_{k2}, p_{k2} \rangle$
$\vdots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
t_n	$\langle w_{1n}, p_{1n} \rangle$	$\langle w_{2n}, p_{2n} \rangle$	$\ldots$	$\langle w_{kn}, p_{kn} \rangle$

Table 1: The rearrangement table for fixed segmentation $S = \{s_1, \ldots, s_k\}$.

segmentation error if we move t_j from its current position *pos*(t_j, T) to segment s_i. Note that the exact position within the segment in which point t_j is moved does not affect the gain (loss) in the segmentation error. Let λ_j be the representative of the segment of S where t_j is initially located and μ_i the representative of segment s_i. Then, for a fixed segmentation S the gain p_{ij} is

$$p_{ij} = |t_j - \mu_i|^p - |t_j - \lambda_j|^p.$$

Moreover, point t_j is associated with cost (weight) w_{ij}, which is the *operational* cost of moving point t_j to segment s_i. If we use a_i, b_i to denote the start and the end points of the segment s_i, then the operational cost in the case of a move operation is

$$w_{ij} = \begin{cases} 1, & \text{if } t_j \notin s_i, \\ 0, & \text{otherwise.} \end{cases}$$

For the case of swaps the operational cost is

$$w_{ij} = \begin{cases} \min\{|a_i - \textit{pos}(t_j, T)|, & \\ \qquad |b_i - \textit{pos}(t_j, T)|\}, & \text{if } t_j \notin s_i, \\ 0, & \text{otherwise.} \end{cases}$$

Note that since the segmentation is fixed, when repositioning the point t_j to the segment s_i (if $t_j \notin s_i$) it is enough to make as many swaps as are necessary to put the point in the beginning or ending positions of the segment (whichever is closer).

Given the above definitions the REARRANGEMENT problem can be easily formulated with the following integer program.

$$\begin{array}{ll} \text{maximize} & z = \sum_{i=1}^{k} \sum_{t_j \in T} p_{ij} x_{ij} \\ \text{subject to:} & \sum_{i=1}^{k} \sum_{t_j \in T} w_{ij} x_{ij} \leq C \quad \text{(constraint 1)} \\ & \sum_{i=1}^{k} x_{ij} \leq 1, \quad \text{for every } j \quad \text{(constraint 2)} \\ & x_{ij} \in \{0, 1\}, \quad i = \{1, \ldots, k\},\ t_j \in T. \end{array}$$

	Knapsack
α_1	$\langle w_1, p_1 \rangle$
α_2	$\langle w_2, p_2 \rangle$
$\vdots$	$\ldots$
α_n	$\langle w_n, p_n \rangle$

Table 2: The tabular formulation of the KNAPSACK problem.

That is, the objective is to maximize the gain in terms of error by moving the points into new segments. At the same time we have to make sure that at most C operations are made (constraint 1), and each point t_j is transferred to at most one new location (constraint 2). Table 1 gives a tabular representation of the input to the REARRANGEMENT problem. We call this table the *rearrangement* table. The table has n rows and k columns and each cell is associated with a pair of values $\langle w_{ij}, p_{ij} \rangle$, where w_{ij} is the operational cost of rearranging point t_j to segment s_i and p_{ij} corresponds to the gain in terms of segmentation error that will be achieved by such a rearrangement. The integer program above implies that the solution to the REARRANGEMENT problem contains at most one element from each row of the rearrangement table.

One can observe that the REARRANGEMENT problem is a generalization of the KNAPSACK. In KNAPSACK we are given a set of n items $\{\alpha_1, \ldots, \alpha_n\}$ and each item α_j is associated with a weight w_j and a profit p_j. The knapsack capacity B is also given as part of the input. The goal in the KNAPSACK problem is to find a subset of the input items with total weight bounded by B, such that the total profit is maximized. We can express this with the following integer program.

$$\begin{array}{ll} \text{maximize} & z = \sum_{i=1}^{n} p_i x_i \\ \text{subject to:} & \sum_{i=1}^{n} w_i x_i \leq B \\ & x_i \in \{0, 1\}, i = 1, 2, \ldots, k. \end{array}$$

The tabular representation of the KNAPSACK is shown in Table 2. Note that this table, unlike the rearrangement table, has just a single column. Each element is again associated with a weight and a profit and is either selected to be in the knapsack or not. It is known that the KNAPSACK problem is NP-hard [6]. However, it does admit a pseudopolynomial-time algorithm that is based on dynamic programming [19].

The similarity between REARRANGEMENT and KNAPSACK leads us to apply algorithmic techniques similar to those applied for KNAPSACK. First observe that, in our case, the bound C on the number of operations that we can afford is an integer number. Moreover, for all i, j, we have that $w_{ij} \in \mathbb{Z}^+$ and $p_{ij} \in \mathbb{R}$. Denote now by $A[i, c]$ the maximum gain in terms of error that we can achieve if we consider points t_1 up to t_i and afford a total cost(weight) up to $c \leq C$. Then, the following recursion allows us to compute all the values $A[i, c]$

$$A[i,c] = \max \quad \{A[i-1,c], \max_{1\leq k'\leq k}(A[i-1,c-w_{k'i}]+p_{k'i})\}. \tag{5.3}$$

.

The first term of the outer $\max$ corresponds to the gain we would obtain by not rearranging the i-th element, while the second term corresponds to the gain we would have by rearranging it.

LEMMA 5.1. *Recurrence (5.3) finds the optimal solution to the* REARRANGEMENT *problem.*

Proof. First observe that by construction the recursion produces a solution that has cost at most C. For proving the optimality of the solution we have to show that the output solution is the one that has the maximum profit. The key claim is the following. For every $i \in \{1, \dots, n\}$, the optimal solution of weight at most c is a superset of the optimal solution of weight at most $c - w_{k'i}$, for any $k' \in \{1, \dots, k\}$. We prove this claim by contradiction. Let $A[i, c]$ be the gain of the optimal solution that considers points t_1 to t_i and let $A[i-1, c-w_{k'i}]$ be the gain of the subset of this solution with weight at most $c - w_{k'i}$. Note that $A[i-1, c-w_{k'i}]$ is associated with a set of rearrangements of points t_1 to t_{i-1}. Now assume that $A[i-1, c-w_{k'i}]$ is not the maximum profit of weight $c - w_{k'i}$. That is, assume that there exists another set of points from $\{t_1, \dots, t_{i-1}\}$, that if rearranged gives gain $A' > A[i-1, c-w_{k'i}]$ with weight still less than $c - w_{k'i}$. However, if such set of rearrangements exist then the profit of $A[i, c]$ would not have been optimal and therefore we reach a contradiction. □

The following theorem gives the running time of the dynamic-programming algorithm that evaluates recursion (5.3).

THEOREM 5.1. *For arbitrary gains p_{ij}, costs w_{ij} and integer C, recursion (5.3) defines an $O(nkC^2)$ time algorithm. This is a pseudopolynomial algorithm for the* REARRANGEMENT *problem.*

Proof. The dynamic-programming table A is of size $O(nC)$ and each step requires kC operations. Thus, the total cost is $O(nkC^2)$. Since C is an integer provided as input the algorithm is pseudopolynomial. □

An immediate corollary of Theorem 5.1 is the following.

COROLLARY 5.1. *For the special case of the* REARRANGEMENT *problem where we consider only moves (or only bubble-sort swaps), recurrence (5.3) computes the optimal solution in polynomial time.*

Proof. In the special cases in question, the weights w_{ij} are integers and bounded by n. Similarly C is also polynomially bounded by n. This is because we do not need more than n moves (or n^2 swaps). Therefore, for the special case of moves and swaps the dynamic-programming algorithm runs in time polynomial in n. □

In the case of move operations the REARRANGEMENT problem is even simpler. Recall that in the case of moves we have $w_{ij} \in \{0, 1\}$ for all i, j. This is because $w_{ij} = 1$ for every $t_j \notin s_i$ and $w_{ij} = 0$ if $t_j \in s_i$. Therefore, the REARRANGEMENT problem can be handled efficiently in terms of the rearrangement table (Table 1). Let p_i be the largest profit obtained by moving point t_i and let k' be the cell of the i-th row that indeed gives this maximum profit. Furthermore, let $w_i = w_{k'i}$. The rearrangement requires the movement of the C points with the highest $p_i w_i$ values to the indicated segment. This can be done simply by sorting the n points of T with respect to their $p_i w_i$ values in time $O(n \log n)$. Alternatively, we can do it simply in two passes ($O(n)$ time) by finding the point with the C-th largest $p_i w_i$ value (this can be done in linear time [4]) and then moving all the points with values higher or equal to this.

5.2 Pruning the candidates for rearrangement In the previous subsection we have considered all points in T as candidate points for rearrangement. Here we restrict this set of candidates. Algorithm 2 is an enhancement of Algorithm 1. The first step of the two algorithms are the same. The second step of the TRUNCATEDSEGMENT&REARRANGE algorithm finds a set of points $D \subseteq T$ to be considered as candidates for rearrangement. Note that the cardinality of this set is bounded by C, the total number of operations allowed. In that way, we can reduce the complexity of the actual rearrangement step, since we are focusing on the relocation of a smaller set of points.

We argue that it is rational to assume that the points to be rearranged are most probably the points that have values different from their neighbors in the input sequence T. Notice that we use the term "neighborhood" here to denote the closeness of points in the sequence T rather than the Euclidean space.

Example. Consider the 1-dimensional sequence $T = \{10, 9, 10, 10, 1, 1, 10, 1, 1, 1\}$. It is apparent that this sequence consists of two rather distinct segments. Notice that if we indeed segment T into two segments we obtain segmentation with segments $s_1 = \{10, 9, 10, 10\}$ and

Algorithm 2 The TRUNCATEDSEGMENT&REARRANGE algorithm.

```
Input: Sequence $T$ of $n$ points, number of segments $k$, number of operations $C$.
Ouput: A rearrangement of the points in $T$ and a segmentation of the new sequence into $k$ segments.
Prune: $D = \text{prune}(T)$
Segment: $S = S_{\text{opt}}(T - D, k)$
Rearrange: $\overline{O} = \text{rearrange}(D, S, C)$
Segment: $S' = S_{\text{opt}}(T_{\overline{O}}, k)$
```

$s_2 = \{1, 1, 10, 1, 1, 1\}$ with error $E_1(T, 2) = 10$. One can also observe that the seventh point with value 10 is an outlier w.r.t. its neighborhood (all the points close to it have value 1, while this point has value 10). Intuitively, it seems the the seventh point has arrived early. Therefore, moving this point at its "correct" position, is expected to be beneficial for the error of the optimal segmentation on the new sequence T'. Consider the move operation $\mathcal{M}(7 \to 4)$, then $T' = \mathcal{M}(7 \to 4) \circ T = \{10, 9, 10, 10, 10, 1, 1, 1, 1, 1\}$. The two-segment structure is even more pronounced in sequence T'. The segmentation of T' into two segments would give the segments $s'_1 = \{10, 9, 10, 10, 10\}$ and $s'_2 = \{1, 1, 1, 1, 1\}$ and total error $E'_1 = 1$. □

We use the outliers of the sequence T as candidate points to be moved in new locations. For this we should first clearly define the concept of an outlier by adopting the definitions provided in [12, 16].

Following the work of [16] we differentiate between two types of outliers, namely the *deviants* and the *pseudodeviants* (a.k.a. *pdeviants*). Consider sequence T, integers k and ℓ and error function E_p. The optimal set of ℓ deviants for the sequence T and error E_p is the set of points D such that

$$D = \arg \min_{D' \subseteq T, |D'| = \ell} E_p \left(T - D, k\right).$$

In order to build intuition about deviants, lets turn our attention to a single segment s_i. The total error that this segment contributes to the whole segmentation, before any deviants are removed from it, is $E_p(s_i, 1)$. For point t and set of points P we use $d_p(t, P)$ to denote the distance of the point t to the set of points in P. For $p = 1$ this is $d_1(t, P) = |t - \textit{median}(P)|$, where *median*$(P)$ is the median value of the points in P. Similarly, for $p = 2$, $d_2(t, P) = |t - \textit{mean}(P)|^2$, where *mean*$(P)$ is the mean value of the points in P. The optimal set of ℓ_i deviants of s_i are the set of points $D_i \in s_i$ with $|D_i| = \ell_i$ such that

$$D_i = \arg\max_{D'_i} \sum_{t \in D'_i} d_p(t, s_i - D_i). \tag{5.4}$$

Example. Consider segment $s = \{20, 10, 21, 9, 21, 9, 20, 9\}$ and let $\ell = 4$. Then, the optimal set of 4 deviants is $D = \{20, 21, 21, 20\}$, and $E_2(s - D, 1) = 0.75$. □

A slightly different notion of outliers is the so-called *pseudodeviants*. Pseudodeviants are faster to compute but have slightly different notion of optimality than deviants. The differences between the two notions of outliers becomes apparent when we focus our attention to the deviants of a single segment s_i, where the representative μ_i of the segment is computed *before* the removal of any deviant points. Then the optimal set of ℓ_i pseudodeviants of segment s_i is

$$\hat{D}_i = \arg\max_{D'_i} \sum_{t \in D'_i} d_p(t, s_i). \tag{5.5}$$

Example. Consider again the segment $s = \{$20,10,21,9,21, 9,20,9$\}$ and let $\ell = 4$, as in the previous example. The set of pseudodeviants in this case is $\hat{D} = \{21, 9, 9, 9\}$, with error $E_2(s - \hat{D}, 1) = 80.75$. □

From the previous two examples it is obvious that there are cases where the set of deviants and the set of pseudodeviants of a single segment (and thus of the whole sequence) may be completely different. However, finding the optimal set of pseudodeviants is a much more easy algorithmic task.

In order to present a generic algorithm for finding deviants and pseudodevians we have to slightly augment our notation. So far the error function E was defined over the set of possible input sequences $\mathcal{T}$ and integers $\mathbb{N}$ that represented the possible number of segments. Therefore, for $T \in \mathcal{T}$ and $k \in \mathbb{N}$, $E(T, k)$ was used to represent the error of the optimal segmentation of sequence T into k segments. We now augment the definition of function E with one more argument, the number of outliers. Now we write $E(T, k, \ell)$ to denote the minimum error of the segmentation of a sequence $T - D$, where D is the set of outliers of T with cardinality $|D| = \ell$. Finally, when necessary, we overload the notation so that for segmentation S with segments $\{s_1, \ldots, s_k\}$ we use s_i to represent the points included in segment s_i.

The generic dynamic-programming recursion that finds the optimal set D of ℓ deviants (or pseudodeviants) of sequence T is

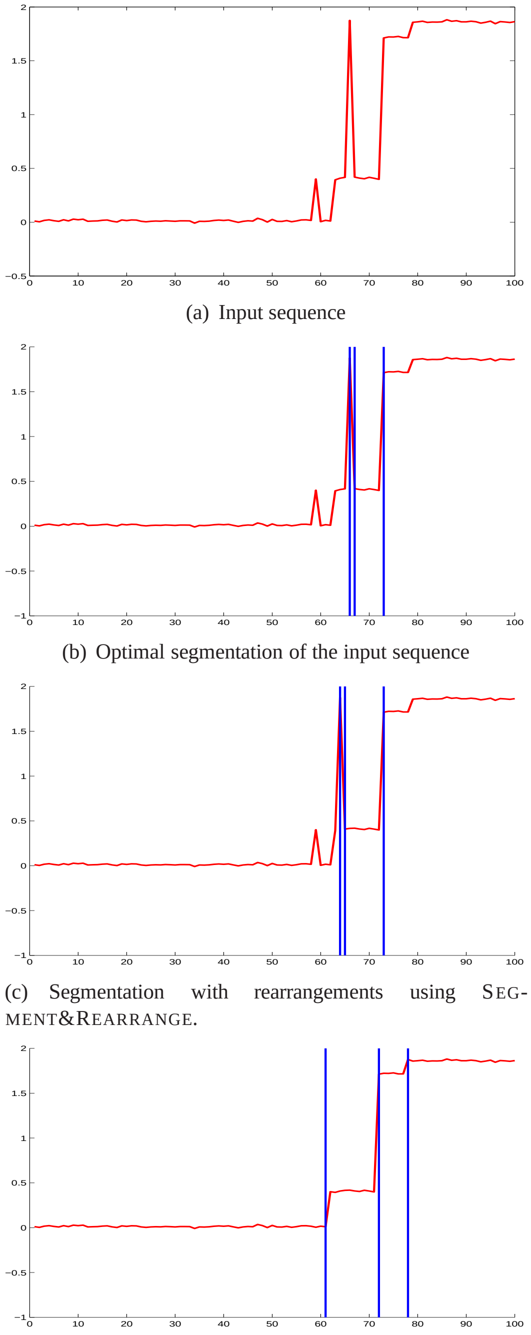
$$\begin{aligned} E_p\left(T\left[1,n\right], k, \ell\right) = \\ \min_{\substack{1 \le i \le n \\ 0 \le j \le \ell}} \quad & \{E_p\left(T\left[1,i\right], k-1, j\right) \\ & + E_p\left(T\left[i+1,n\right], 1, \ell - j\right)\}. \end{aligned} \tag{5.6}$$

The recursive formula (5.6) finds the best allocation of outliers between the subsequences $T[1, i]$ and $T[i + 1, n]$. Note that the recursion can be computed in polynomial time if both the terms of the sum can be computed in polynomial time. Let C_1 be the cost of computing the first term of the sum and C_2 the computational cost of the second term. Then, evaluating (5.6) takes time $O\left(n^2\ell\left(C_1 + C_2\right)\right)$. Time C_1 is constant since it is just a table lookup. Time C_2 is the time required to evaluate Equations (5.4) and (5.5) for deviants and pseudodeviants respectively. From our previous discussion on the complexity of Equations (5.4) and (5.5) we can conclude that recursion (5.6) can compute in polynomial time the pseudodeviants of a sequence, irrespective of the dimensionality of the input data. For the case of deviants and one-dimensional data Recurrence 5.6 can compute the optimal set of deviants in time $O(n^2\ell^2)$. In the case of pseudodeviants and arbitrary dimensionality, the time is $O(n^3\ell)$.

Although the pseudodeviant-extraction phase is rather expensive, we note that the expensive step of outlier detection is independent of the rearrangement step, so in practice one can use a faster and less accurate algorithm for finding the outliers. We defer the discussion of whether the quality of the results in arbitrary datasets is improved when we apply the outlier-detection algorithm in the experimental section. Here we just give an indicative (maybe contrived) example (see Figure 1) of a case where the extraction of outliers before proceeding in the rearrangement step proves useful.

Example. Consider the input sequence shown in Figure 1(a). The arrangement of the input points is such that the optimal segmentation of the sequence assigns the 66-th point, say p_{66}, of the sequence alone in one small segment (Figure 1(b)). Given the optimal segmentation of the input sequence, the SEGMENT&REARRANGE algorithm does not consider moving p_{66}. This is because there cannot exist another segment whose representative would be closer to the p_{66} than the point itself. As a result, the segmentation with rearrangements we obtain using the SEGMENT&REARRANGE algorithm and allowing at most 2 moves is the one shown in Figure 1(c). The error of this segmentation is 0.388 which is a 50% improvement over the error of the optimal segmentation of the input sequence without rearrangements. However, the TRUNCATEDSEGMENT&REARRANGE algorithm immediately identifies that points p_{59} and p_{66} are the ones that differ from their neighbors and it focuses on repositioning just these two points. Furthermore, the segment representatives of the segmentation used for the rearrangement are calculated by ignoring the outliers and therefore, the algorithm produces the output shown in Figure 1(d). This output has error 0.005 that is almost 100% improvement over the error of the optimal segmentation (without rearrangements). □

5.3 The GREEDY algorithm The SEGMENT & REARRANGE and the TRUNCATEDSEGMENT & REARRANGE algorithms were focusing on finding a sequence of rearrangement operations of cost at most C, that improved the error of the segmentation of the input sequence T. That is, the al-



(a) Input sequence

(b) Optimal segmentation of the input sequence

(c) Segmentation with rearrangements using SEGMENT&REARRANGE.

(d) Segmentation with rearrangements using TRUNCATED-SEGMENT&REARRANGE.

Figure 1: Pathological example where the TRUNCATEDSEGMENT&REARRANGE algorithm is useful.

gorithms were initially fixing a segmentation, and then they were deciding on all the possible operations that could improve the error of this specific segmentation. In this section, we describe the GREEDY algorithm that (a) rearranges one point at a time and (b) readjusts the segmentation that guides the rearrangement of the points after every step. The pseudocode of the GREEDY algorithm is given in Algorithm 3.

Algorithm 3 The GREEDY algorithm.

```
Input: Sequence $T$ of $n$ points, number of segments $k$, number of operations $C$.
Ouput: A rearrangement of the points in $T$ and a segmentation of the new sequence into $k$ segments.
$c \leftarrow 0$
$S \leftarrow S_{\text{opt}}(T,k)$
$E_1 \leftarrow E\left(S_{\text{opt}}(T,k)\right)$, $E_0 \leftarrow \infty$
while $c \leq C$ and $(E_1 - E_0) < 0$ do
    $E_0 \leftarrow E_1$
    $\langle \overline{O}, p\rangle \leftarrow \text{LEP}(T,S,C-c)$
    $c \leftarrow c + |\overline{O}|$
    $T \leftarrow \overline{O} \circ T$
    $S \leftarrow S_{\text{opt}}(T,k)$
    $E_1 \leftarrow E\left(S_{\text{opt}}(T,k)\right)$
end while
```

At each step the GREEDY algorithm decides to rearrange point p such that the largest reduction in the segmentation error of the rearranged sequence is achieved. The decision upon which point to be rearranged and the sequence of rearrangements is made in the LEP (Least Error Point) routine. The same routine ensures that at every step the condition $\left(c + |\overline{O}| \leq C\right)$ is satisfied. Note that the decision of the point to be moved is guided by the recomputed segmentation of the rearranged sequence and it is a tradeoff between the error gain due to the rearrangement and the operational cost of the rearrangement itself. The algorithm terminates either when it has made the maximum allowed number of operations, or when it cannot improve the segmentation cost any further.

If t is the time required by the LEP procedure and I the number of iterations of the algorithm, then the GREEDY algorithm needs $O(It)$ time. The LEP procedure creates the rearrange matrix of size $n \times k$ and among the entries with weight less than the remaining allowed operations, it picks the one with the highest profit. This can be done in $O(nk)$ time and therefore the overall computational cost of the GREEDY algorithm is $O(Ink)$.

6 Experiments

In this section we compare experimentally the algorithms we described in the previous sections. We use the *error ratio* as a measure of the qualitative performance. That is, if an algorithm $\mathcal{A}$ produces a k-segmentation with error $E^{\mathcal{A}}$ and the optimal k-segmentation (without rearrangements) has error E^*, then the error ratio is defined to be $r = E^{\mathcal{A}}/E^*$. In our case, the error ratio is by definition less than 1, since we only do rearrangements that reduce the segmentation error. When the value of r is small even for small number of rearrangements, then we can conclude that segmenting with rearrangements is meaningful for a given dataset.

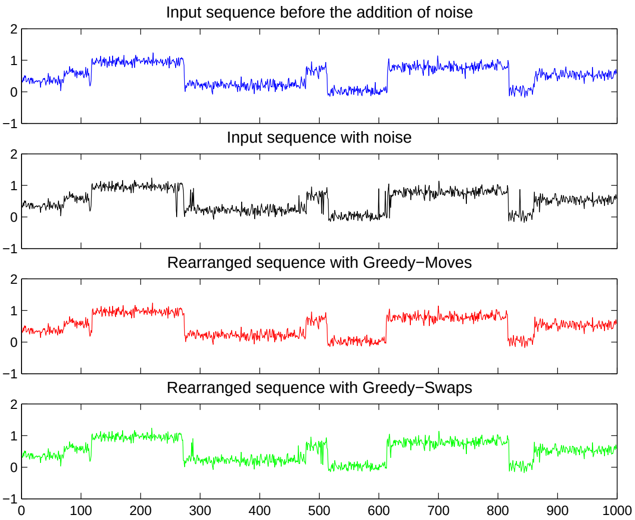


Figure 3: Anecdotal evidence of algorithms' qualitative performance.

We study the quality of the results for the different algorithms and rearrangement types. For the study we use both synthetic and real datasets, and we explore the cases where segmentation with rearrangements is meaningful.

6.1 Experiments with synthetic data

The synthetic data are generated as follows. First, we generate a ground-truth dataset (Step 1). Then, we rearrange some of the points of the dataset in order to produce the rearranged (or *noisy*) dataset (Step 2). The output of Step 2 is used as input for our experiments.

For Step 1 we first fix the dimensionality d of the data. Then we select k segment boundaries, which are common for all the d dimensions. For the j-th segment of the i-th dimension we select a mean value μ_{ij}, which is uniformly distributed in $[0,1]$. Points are then generated by adding a noise value sampled from the normal distribution $\mathcal{N}(\mu_{ij},\sigma^2)$. For the experiments we present here we have fixed the number of segments $k = 10$. We have generated datasets with $d = 1,5,10$, and standard deviations varying from 0.05 to 0.9.

Once the ground-truth dataset is generated we proceed with rearranging some of its points. There are two parameters that characterize the rearrangements, *np* and l. Parameter *np* determines how many points are moved from their initial location, while l determines for every moved point its new position on the sequence. More specifically, for every point p_i (located at position i) that is moved, its new position is uniformly distributed in the interval $[i - l, i + l]$.

Figure 2 shows the values of the error ratio achieved by the different combinations of algorithms and rearrangement types. RS-Moves and RS-Swaps correspond to the SEGMENT&REARRANGE algorithm for moves and swaps respectively. Similarly, G-Moves and G-Swaps refer to the GREEDY algorithm with moves and swaps. The results are shown for noisy datasets and for fixed *np* and l (*np* $= 16$, $l = 20$). Similar results were obtained for other combinations of *np* and l values. Notice that for segmenting with rearrangements we use the number of swaps and moves that are implied for the data-generation process. That is, when *np* $= 16$ and $\ell = 20$, we allow at most 16 moves and $16 \times 20 = 360$ swaps. For the synthetic datasets there are no significant differences in the quality of the results obtained by SEGMENT&REARRANGE and GREEDY algorithms. We observe though, that swaps give usually worse results than an equivalent number of moves. This effect is particularly pronounced in the 1-dimensional data. This is because in low dimensions the algorithms (both SEGMENT&REARRANGE and GREEDY) are susceptible to err and prefer swaps that are more promising locally but do not lead to a good overall rearrangement.

Figure 3 shows the effect of rearrangements on a noisy input sequence. There are four series shown in Figure 3. The first is the generated ground-truth sequence. (This is the sequence output by Step 1 of the data-generation process). The second sequence is the rearranged (noisy) sequence (and thus the output of Step 2 of the data-generation process). In this sequence the existence of rearranged points is evident. Notice, for example, intervals $[250 - 350]$, $[450 - 550]$, $[580 - 610]$ and $[800 - 850]$. The sequence recovered by the GREEDY algorithm with moves is almost identical to the input sequence before the addition of noise. However, the same algorithm with swaps does not succeed in finding all the correct rearrangements that need to be done.

Figure 4 shows the effect of pruning in the quality of the segmentation results. In this case we compare the quality of four segmentation outputs; the segmentation obtained by the SEGMENT&REARRANGE algorithm (SR), the segmentation obtained by the GREEDY algorithm (G) and the one obtained by the TRUNCATEDSEGMENT&REARRANGE (TSR). We also show the error ratio achieved by segmenting the ground-truth sequence using the conventional optimal segmentation algorithm (GT). Observe first that surprisingly our methods give better results than the segmentation of the ground-truth sequence. This means that the methods find rearrangements that needed to be done in the input sequence but were not generated by our rearrangement step during data generation. Also note that focusing on rearrangement of pseudodeviants does not have a considerably bad effect on the quality of the obtained segmentation. However it does not lead to improvements either, as we had expected when studying pathological cases.

6.2 Experiments with real data

Figure 5 shows the error ratio for different combinations of algorithms and rearrangement types, for a set of real datasets (`attas`, `phone`, `powerplant`, `robot_arm`, `shuttle` and `soiltemp`). The datasets along with their description can be downloaded from the UCR time-series data mining archive.[1] We plot the results as a function of the number of allowed rearrangement operations. In this case, since we are ignorant of the data-generation process, we use in all experiments the same number of moves and swaps. Under these conditions the effect of moves is expected to be larger, which indeed is observed in all cases. Furthermore, for a specific type of rearrangements the SEGMENT&REARRANGE and GREEDY algorithms give results that are always comparable. Finally, notice that there are cases where moving just 128 points of the input sequence leads to a factor 0.5 error improvement. For example, this is the case in *attas* and *soiltemp* datasets, where 128 points correspond to just 10% and 5% of the data points respectively.

7 Conclusions

In this paper we have introduced and studied the SEGMENTATION WITH REARRANGEMENTS problem. In particular we have considered two types of rearrangement operations, namely moves and bubble-sort swaps. We have shown that in most of the cases, the problem is hard to solve with polynomial-time algorithms. For that reason we discussed a set of heuristics that we experimentally evaluated using a set of synthetic and real time-series datasets. For each one of those heuristics we considered their potentials and shortcomings under different types of input data. Experimental evidence showed that allowing a small number of rearrangements my significantly reduce the error of the output segmentation.

Acknowledgments

We would like to thank Taneli Mielikäinen for many useful discussions and suggestions.

References

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

[1] `http://www.cs.ucr.edu/~eamonn/TSDMA/`

[2] E. Bingham, A. Gionis, N. Haiminen, H. Hiisilä, H. Mannila, and E. Terzi. Segmentation and dimensionality reduction. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2006.

[3] M. Charikar, S. Khuller, D. M. Mount, and G. Narasimhan. Algorithms for facility location problems with outliers. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 642–651, 2001.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[5] 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.

[6] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[7] A. Gionis and H. Mannila. Finding recurrent sources in sequences. In *International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 123–130, 2003.

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

[9] N. Haiminen and A. Gionis. Unimodal segmentation of sequences. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 106–113, 2004.

[10] T. Hartman and R. Sharan. A 1.5-approximation algorithm for sorting by transpositions and transreversals. *Journal of Computer and System Sciences (JCSS)*, 70(3):300–320, 2005.

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

[12] H. V. Jagadish, N. Koudas, and S. Muthukrishnan. Mining deviants in a time series database. In *Proceedings of Very Large Data Bases (VLDB) Conference*, pages 102–113, 1999.

[13] E. J. Keogh and M. J. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 239–243, 1998.

[14] E. J. Keogh and P. Smyth. A probabilistic approach to fast pattern matching in time series databases. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 24–30, 1997.

[15] N. Megiddo, E. Zemel, and S. L. Hakimi. The maximum coverage location problem. *SIAM Journal on Algebraic and Discrete Methods*, 4:253–261, 1983.

[16] S. Muthukrishnan, R. Shah, and J. S. Vitter. Mining deviants in time series data streams. In *Proceedings of Statistical and Scientific Database Management (SSDBM)*, pages 41–50, 2004.

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

[18] E. Terzi and P. Tsaparas. Efficient algorithms for sequence segmentation. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2006.

[19] V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[20] M. Vlachos, S. Papadimitriou, Z. Vagena, and P. S. Yu. Riva: Indexing and visualization of high-dimensional data via dimension reorderings. In *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, 2006.

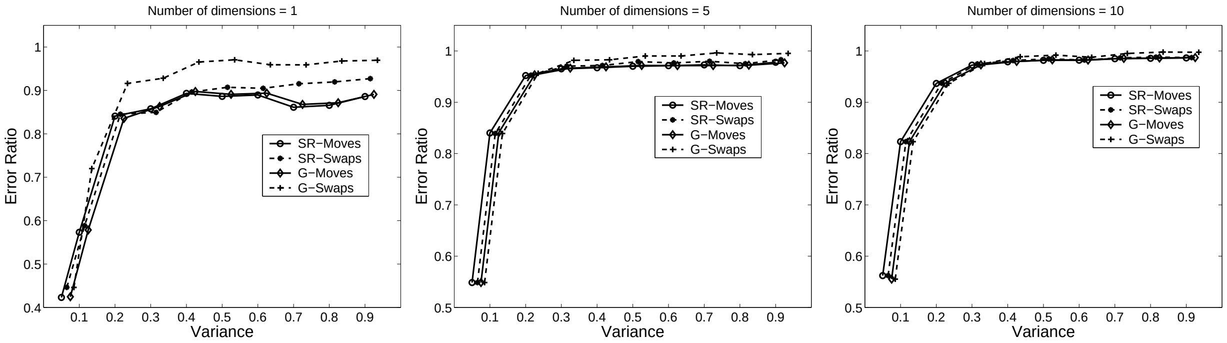


Figure 2: Error ratio for different algorithms and rearrangement types on synthetic datasets with dimensions $d = 1, 5$ and 10.

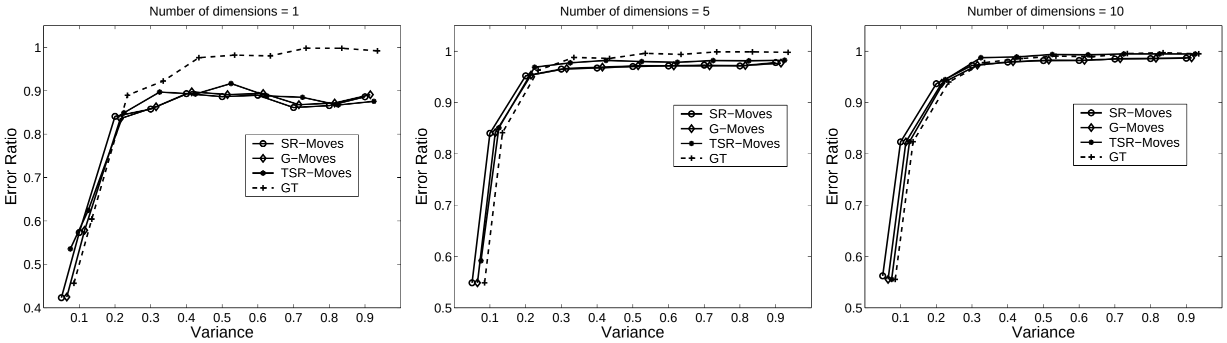


Figure 4: Error ratio with pruning for synthetic datasets with dimensions $d = 1, 5$ and 10.

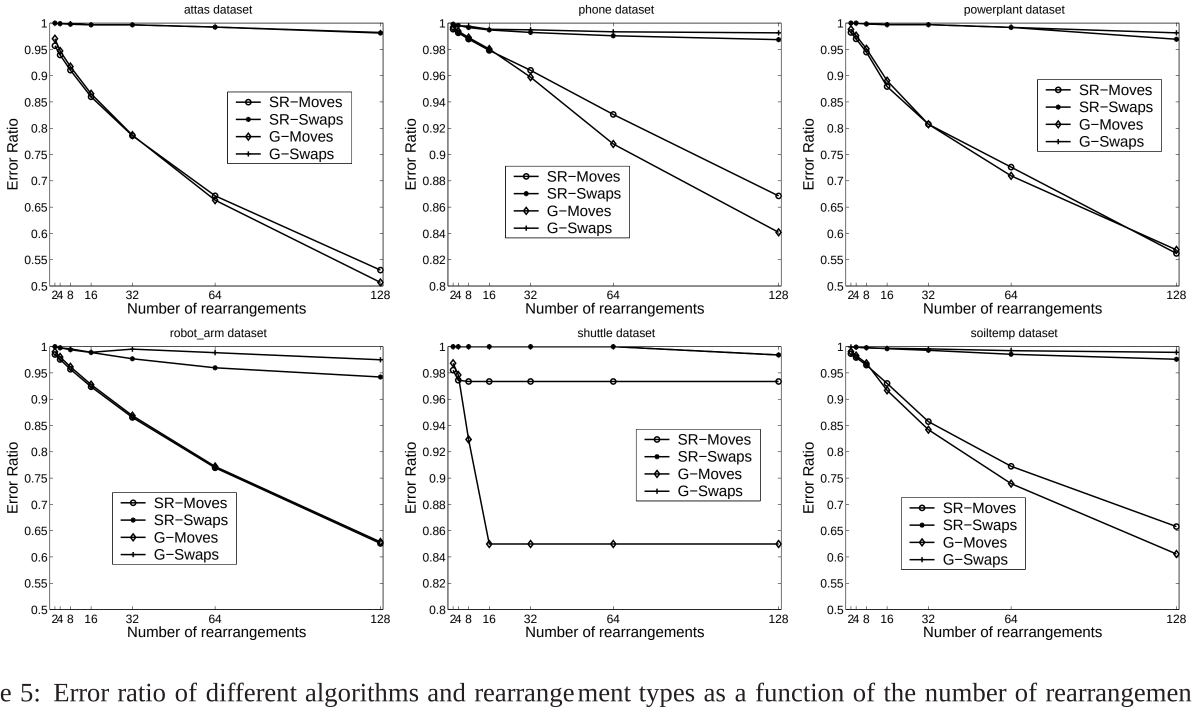


Figure 5: Error ratio of different algorithms and rearrangement types as a function of the number of rearrangements for different real datasets.