Department of Computer Science
Series of Publications A
Report A-2006-5

Problems and Algorithms for Sequence Segmentations

Evimaria Terzi

Academic Dissertation

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XII, University Main Building, on December 18th, 2006, at 12 o'clock noon.

University of Helsinki
Finland

Copyright © 2006 Evimaria Terzi

ISSN 1238-8645
ISBN 952-10-3519-6 (paperback)
ISBN 952-10-3520-X (PDF)
http://ethesis.helsinki.fi/

Computing Reviews (1998) Classification: E.4, H.2.8

Helsinki University Printing House
Helsinki, November 2006 (140 pages)

Problems and Algorithms for Sequence Segmentations

Evimaria Terzi
Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
Evimaria.Terzi@cs.helsinki.fi
http://www.cs.helsinki.fi/u/terzi

Abstract

The analysis of sequential data is required in many diverse areas such as telecommunications, stock market analysis, and bioinformatics. A basic problem related to the analysis of sequential data is the sequence segmentation problem. A sequence segmentation is a partition of the sequence into a number of non-overlapping segments that cover all data points, such that each segment is as homogeneous as possible. This problem can be solved optimally using a standard dynamic programming algorithm.

In the first part of the thesis, we present a new *approximation algorithm* for the sequence segmentation problem. This algorithm has smaller running time than the optimal dynamic programming algorithm, while it has bounded approximation ratio. The basic idea is to divide the input sequence into subsequences, solve the problem optimally in each subsequence, and then appropriately combine the solutions to the subproblems into one final solution.

In the second part of the thesis, we study alternative segmentation models that are devised to better fit the data. More specifically, we focus on *clustered segmentations* and *segmentations with rearrangements*. While in the standard segmentation of a multidimensional sequence all dimensions share the same segment boundaries, in a clustered segmentation the multidimensional sequence is segmented in such a way that dimensions are allowed to form clusters. Each cluster of dimensions is then segmented separately. We formally define the problem of clustered segmentations and we experimentally show that segmenting sequences using this segmentation model, leads to solutions with smaller error for the same model cost. Segmentation with rearrangements is 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. We formulate the problem of segmentation with rearrangements and we show that it is an NP-hard problem to solve or even to approximate. We devise effective algorithms for the proposed problem, combining ideas from dynamic programming and outlier detection algorithms in sequences.

In the final part of the thesis, we discuss the problem of *aggregating* results of segmentation algorithms on the same set of data points. In this case, we are interested in producing a partitioning of the data that agrees as much as possible with the input partitions. We show that this problem can be solved optimally in polynomial time using dynamic programming. Furthermore, we show that not all data points are candidates for segment boundaries in the optimal solution.

Computing Reviews (1998) Categories and Subject Descriptors:

E.4 Coding and Information Theory: Data Compaction and Compression

H.2.8 Database Applications: Data mining

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Segmentation, Dynamic programming, Approximation algorithms

Acknowledgements

My very special thanks go to my advisor, Heikki Mannila, for entrusting me with the freedom to explore my interests and for teaching me the importance of intellectual curiosity and scientific honesty. I benefited a lot from his scientific insights and wise advice, but I also enjoyed our informal discussions and his way of dealing with my Mediterranean temperament.

An earlier draft of the dissertation was reviewed by Erkki Mäkinen and Jorma Tarhio, whom I thank for their useful comments.

During the last years I was lucky to work with many special people who unconditionally taught me a great deal of things. I am deeply grateful to my mentor in IBM and Microsoft, Rakesh Agrawal, for his generosity with his time and ideas. Although our common work is not part of this thesis, I have to thank him for teaching me that I should have the courage to finish the projects I start. I am also especially thankful to Floris Geerts, Aris Gionis and Panayiotis Tsaparas for their mentorship and their friendship. I fondly remember our discussions with Floris on "random" problems, Aris' ability to spontaneously spot my mistakes, and Panayiotis' wealth of conjectures. I already miss our vivid discussions, along with the feeling of "safety" I had when working with them. I learned a lot from my interactions with Mikko Koivisto and Taneli Mielikäinen. I thank them for this. I also thank Niina Haiminen for fun discussions on the "S" project. It was a pleasure to work with Elisa Bertino, Sreenivas Gollapudi, Ashish Kamra and Hannu Toivonen and learn from them.

The Department of Computer Science and BRU proved a great host for me for the last years. I thank Jukka Paakki and Esko Ukkonen for making this happen. The financial support of ComBi

graduate school is gratefully acknowledged. I am also indebted to Inka Kujala and Teija Kujala for making my life easier and for dealing with all the bureaucratic problems I did not want to deal with.

I benefited a lot and in many different ways from the friendship of Aimee, Anne, Eli, Kismat, Kristin, Mika, Oriana, Sophia, and Vasilis. I want to thank them for giving me their alternative perspectives. Aris, Panayiotis and Vasilis deserve a separate mention for making the Helsinki experience special.

Last but not least, I want to express my immeasurable gratitude to my family for providing me with the luxury to take them for granted.

Evimaria Terzi, November 2006.

Contents

1 Introduction **1**

2 Preliminaries **5**

2.1 The sequence segmentation problem 5
2.2 Optimal algorithm for sequence segmentation 8
2.3 Related work 11
2.3.1 Algorithms and applications of sequence segmentation 11
2.3.2 Variants of the `Segmentation` problem 13

3 Approximate algorithms for sequence segmentation **15**

3.1 The Divide&Segment algorithm 16
3.1.1 Analysis of the DnS algorithm. 19
3.2 Recursive DnS algorithm 22
3.3 Experiments 28
3.3.1 Segmentation heuristics 29
3.3.2 Experimental setup 30
3.3.3 Performance of the DnS algorithm 30
3.3.4 Exploring the benefits of the recursion 35
3.4 Applications to the (k, h)-segmentation problem . . 36
3.4.1 Algorithms for the (k, h)-segmentation problem 38
3.4.2 Applying DnS to the (k, h)-segmentation problem 39
3.5 Conclusions 41

4 Clustered segmentations **43**

4.1 Problem definition 46
4.1.1 Connections to related work 46

4.1.2 Problem complexity 47
4.2 Algorithms 49
4.2.1 The D_E distance function between segmentations 51
4.2.2 The D_P distance function between segmentations 51
4.2.3 The SamplSegm algorithm 54
4.2.4 The IterClustSegm algorithm 55
4.3 Experiments 56
4.3.1 Ensuring fairness in model comparisons . . . 56
4.3.2 Experiments on synthetic data 57
4.3.3 Experiments on timeseries data 59
4.3.4 Experiments on mobile sensor data 61
4.3.5 Discussion 62
4.4 Conclusions 66

5 Segmentation with rearrangements 67
5.1 Problem formulation 69
5.2 Problem complexity 70
5.3 Algorithms 72
5.3.1 The Segment&Rearrange algorithm . . . 73
5.3.2 Pruning the candidates for rearrangement . . 78
5.3.3 The Greedy algorithm 82
5.4 Experiments 85
5.4.1 Experiments with synthetic data 85
5.4.2 Experiments with real data 91
5.5 Conclusions 91

6 Segmention aggregation 93
6.1 Application domains 95
6.2 The `Segmentation Aggregation` problem 97
6.2.1 Problem definition 97
6.2.2 The disagreement distance 98
6.2.3 Computing the disagreement distance 99
6.3 Aggregation algorithms 101
6.3.1 Candidate segment boundaries 102
6.3.2 Finding the optimal aggregation for D_A . . . 104
6.3.3 Greedy algorithms 106
6.4 Experiments 110

6.4.1 Comparing aggregation algorithms 112
6.4.2 Experiments with haplotype data 112
6.4.3 Robustness experiments 114
6.4.4 Experiments with reality-mining data 117
6.5 Alternative distance functions: Entropy distance . . 120
6.5.1 The entropy distance 120
6.5.2 Computing the entropy distance 122
6.5.3 Restricting the candidate boundaries for D_H 123
6.5.4 Finding the optimal aggregation for D_H . . . 125
6.6 Alternative distance function: Boundary mover's distance 127
6.7 Conclusions 128

7 Discussion **129**

References **131**

CHAPTER 1

Introduction

The abundance of sequential datasets that are available comes along with the need of techniques suitable for their analysis. Such datasets appear in a large range of diverse applications like telecommunications, stock market analysis, bioinformatics, text processing, click-stream mining and many more. The nature of these datasets has intrigued the data mining community to develop methodologies that can handle sequential data with large number of data points.

The work we present in this thesis is motivated by a specific type of sequential data analysis, namely, *sequence segmentation*. Given an input sequence a segmentation divides the sequence into k non-overlapping and contiguous pieces that are called *segments*. Each segment is usually represented by a *model* that concisely describes the data points appearing in the segment. Many different models of varying complexity can be used for this. We focus on the simplest model, namely the piecewise constant approximation. In this model each segment is represented by a single point, e.g., the mean of the points in the segment. We call this point the *representative* of the segment, since it represents the points in the segment. The error of this approximate representation is measured using some *error function*, e.g. the sum of squares. Different error functions are suitable for different applications. For a given error function, the goal is to find the segmentation of the sequence and the corresponding representatives that minimize the error in the representation of the underlying data. We call this problem the *sequence segmentation* problem.

Sequence segmentation gives a precise though coarse represen-

tation of the input data. Thus, segmentation can be perceived as a *data compression* technique. That is, a segmentation of a long sequence, reduces its representation to just k representatives along with the boundaries that define the segments. Furthermore, these representatives are picked in such a way that they provide a good approximation of the points appearing in the segment. From the point of view of *structure discovery*, segmentation allows for a high-level view of the sequence's structure. Moreover, it provides useful information, directing more detailed studies to focus on homogeneous regions, namely the segments. Finally, there are many sequences that appear to have an *inherent segmental structure* e.g., haplotypes or other genetic sequences, sensor data, etc. The analysis of these data using segmentation techniques is a natural choice. Examples of segmentation-based analysis of such data appear in [BLS00, KPV+03, Li01, RMRT00, SKM02, HKM+01].

The focus of this thesis is the segmentation problem and some of its variants. We do not narrow our attention on a specific data analysis task, but rather on a few general themes related to the basic segmentation problem. We start in Chapter 2 by giving a formal description of the basic segmentation problem and describing the optimal dynamic programming algorithm for solving it. We also give an overview of related work in the context of sequence segmentation. The related research efforts follow usually one of the following two trends.

- Propose algorithms for the basic segmentation problem that are faster than the optimal dynamic programming algorithm. Usually, these algorithms are fast and give high quality results.

- Propose new variants of the basic segmentation problem. These variants usually impose some constraints on the structure of the representatives of the segments.

In most of the applications that require the analysis of sequential data, the datasets are massive. Therefore, segmenting them using the optimal dynamic programming algorithm that has complexity quadratic in the number of data points is prohibitive. The need for more efficient segmentation algorithms motivates the work of Chapter 3. In this chapter we present an efficient approximation

algorithm for the basic segmentation problem. The algorithm is based on the idea of dividing the problem into smaller subproblems, solving these subproblems optimally, and then combining the optimal subsolutions to form the solution of the original problem. Parts of this chapter have also appeared in [TT06].

The study of the properties of sequential data reveals that their structure is more complex than the one captured by the simple segmentation model. Although the basic k-segmentation model is adequate for simple data analysis tasks, it seems reasonable to extend it in order to describe and discover the complex structure of the underlying data. This observation motivates Chapters 4 and 5, which study two variants of the basic segmentation problem. In Chapter 4 we introduce the *clustered segmentation* problem, which only applies to multidimensional sequences. In a clustered segmentation, which we have initially introduced in [GMT04], we allow the dimensions of the sequence to form clusters, and segmentation is applied independently to each such cluster. Intuition and experimental evidence shows that this segmentation model is better for certain data mining applications. That is, for the same model cost, it provides a more accurate representation of the underlying data.

In Chapter 5, we study another variant that we call *segmentation with rearrangements*. In many datasets, there are data points that appear to be significantly different from their temporal neighbors. The existence of such points inevitably increases the segmentation error. The main trend so far, has been towards removing these points from the dataset and characterizing them as outliers. When allowing rearrangements we assume that these points are valid data points that they just came out of order. The goal is to find the right position of such misplaced points and then segment the rearranged sequence. We study this problem for different rearrangement strategies and propose some algorithms for dealing with it.

The plethora of segmentation algorithms and of sequences that exhibit an inherent segmental structure, motivates the *segmentation aggregation* problem that we study in Chapter 6. A preliminary version of this study has appeared in [MTT06]. This problem takes as input many different partitions of the same sequence. For example, different segmentation algorithms produce different partitions of the same underlying data points. In such cases, we are interested in producing an aggregate partition, i.e., a segmentation that agrees

as much as possible with the input segmentations. We show that this problem can be solved optimally using dynamic programming for two different (and widely used) distance functions.

We summarize the results of the thesis in Chapter 7, where we also discuss some open problems.

CHAPTER 2

Preliminaries

In the first part of this chapter we define the sequence segmentation problem, and establish the notational conventions that we will (mostly) follow throughout the thesis. In the second part, we give an overview of the related work on problems and algorithms for sequence segmentations.

2.1 The sequence segmentation problem

The input to a segmentation problem is a sequence $T = \{t_1, \ldots, t_n\}$, of finite length n. In principle, the points of the sequence can belong to any domain. We focus our discussion on real multidimensional sequences. In this case, a sequence T consists of n d-dimensional points, that is, $t_i \in \mathbb{R}^d$. We denote by $\mathcal{T}_n$ all such sequences of length n.

Given an integer k, a segmentation partitions T into k contiguous and non-overlapping parts that we call *segments*. The partition is called a *k-segmentation* (or simply a *segmentation*). The partitioning is usually done in such a way that each segment is as *homogeneous* as possible. Homogeneity can be defined in different ways. A segment is considered homogeneous if, for example, it is simple to describe, or if it can be generated by a simple generative model.

Given a sequence T, a k-segmentation S of T can be defined using $(k+1)$ segment *boundaries* (or *breakpoints*). That is, $S = \{s_0, \ldots, s_k\}$, where $s_i \in T$, $s_i < s_{i+1}$ for every i, and $s_0 = 0$ and

$s_k = n$. We define the i-th segment $\bar{s}_i$ of S to be the interval $\bar{s}_i = (s_{i-1}, s_i]$. Note that there is an one-to-one mapping between the end boundaries of each segment and the segments themselves. Each segment consists of $|\bar{s}_i|$ points. We use $\mathcal{S}_n$ to denote the family of all possible segmentations of sequences of length n, and $\mathcal{S}_{n,k}$ to denote all possible segmentations of sequences of length n into k segments.

All the points within each segment, $\bar{s}$, are assumed to be generated by the same generative model $M_{\bar{s}}$. The model $M_{\bar{s}}$ is used for describing the points in $\bar{s}$. Usually, models from the same class are picked for describing all the segments of a segmentation. Different choices of models lead to different segmentation paradigms. We focus on a very simple model that represents all the points in a segment by a constant d-dimensional vector that we call the *representative*, $\mu_{\bar{s}}$. In this way, each point $t \in \bar{s}$ is replaced by $\mu_{\bar{s}}$. The representative collapses the values of the sequence within each segment $\bar{s}$ into a single value $\mu_{\bar{s}}$ (e.g., the mean value of the points appearing in the segment). Collapsing points into representatives results in a less accurate though more compact representation of the sequence. We measure this loss in accuracy using an error function E. The error function is a means for assessing the quality of a segmentation of a sequence, and it measures of the homogeneity of each segment. The error function is a mapping $\mathcal{T}_n \times \mathcal{S}_n \rightarrow \mathbb{R}$. For a given input sequence, we sometimes abuse notation and omit the first argument of the error function.

For a sequence $T \in \mathcal{T}_n$, and an error function E, we define the optimal k-segmentation of T as

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

That is, S_{opt} is the k-segmentation S that minimizes the function $E(T,S)$.

For a given sequence T of length n the definition of the generic k-segmentation problem is as follows.

Problem 2.1 (The `Segmentation` problem) *Given a sequence T of length n, an integer value k, and the error function E, find $S_{opt}(T,k)$.*

Given a segment $\bar{s}_i$ with boundaries s_{i-1}, s_i, the error for the segment $\bar{s}_i$, $\sigma(s_{i-1}, s_i)$, captures how well the model $\mu_{\bar{s}_i}$ fits the

data. For example, the error of a segment can be the sum of the distances of the points to the representatives, or the likelihood of the model used for the segment's description. For sequence T and error function E, the optimal k-segmentation of the sequence is the one that *minimizes* the total representation error

$$E(T,S) \quad = \quad \bigoplus_{\bar{s}\in S} \sigma(s_{i-1},s_i), \qquad (2.1)$$

where $\bigoplus \in \{\sum, \prod, \min, \max\}$. The most important observation is that Equation (2.1) defines a *decomposable* error function. That is, the error of the segmentation is decomposed to the error of each segment. This observation is a key property for proving the optimality of the algorithmic techniques used for solving the `Segmentation` problem.

One straightforward instantiation of 2.1 is the sum of squares error, that is,

$$E(T,S) \quad = \quad \sum_{\bar{s}\in S}\sum_{t\in\bar{s}} |t-\mu_{\bar{s}}|^2.$$

In this case, the error of the segmentation is measured as the squared Euclidean distance of each point from the corresponding representative.

We now turn our attention to error function σ, namely the error of each segment. This error is usually measured using the L_p metric. Given a vector $v=(v_1,\ldots,v_m)$ the L_p norm of the vector is

$$||v||_p = \left(\sum_{i=1}^{m} |v_i|^p\right)^{\frac{1}{p}}.$$

The d_p (or L_p) distance between two points $x,y\in\mathbb{R}^d$ is then defined as $d_p(x,y)=||x-y||_p$. For $p=2$, the the d_p distance corresponds to the Euclidean metric, for $p=1$ to the Manhattan metric and for $p\rightarrow\infty$ to the maximum metric. For ease of exposition we will usually use the d_p^p distance instead of d_p. Given a segment $\bar{s}_i$ with representative $\mu_{\bar{s}_i}$, the error of a segment using distance function d_p^p is

$$\sigma_p(s_{i-1},s_i) \quad = \quad \sum_{t\in\bar{s}_i} d_p(t,\mu_{\bar{s}_i})^p = \sum_{t\in\bar{s}_i} |t-\mu_{\bar{s}_i}|^p\,.$$

The error of the corresponding segmentation S when applied to sequence T is

$$E_p(T,S)^p = \sum_{i=1}^{k} \sigma_p(s_{i-1}, s_i). \tag{2.2}$$

We mainly concentrate on $p = 1$ and $p = 2$. For segment $\bar{s}_i$ and $p = 1$, the representative $\mu_{\bar{s}_i}$ that minimizes $\sigma_1(s_{i-1}, s_i)$ is the median value of the points in $\bar{s}_i$. For $p = 2$ the representative that minimizes $\sigma_2(s_{i-1}, s_i)$ is the mean value of the points in $\bar{s}_i$.

2.2 Optimal algorithm for sequence segmentation

Problem 2.1 can be solved optimally using dynamic programming (DP) for a wide range of decomposable and polynomially computable error functions E. This has been initially observed in [Bel61]. Let T be the input sequence of length n and k the desired number of segments. If we denote by $T[i,j]$ the part of the sequence that starts at position i and ends at position j (with $i < j$), then the main recurrence of the dynamic programming algorithm is

$$E\left(S_{\text{opt}}\left(T\left[1,n\right],k\right)\right) = \min_{j<n} \bigoplus \left\{E\left(S_{\text{opt}}(T\left[1,j\right],k-1)\right), \sigma(j+1,n)\right\}. \tag{2.3}$$

Recursion (2.3) says that the optimal segmentation of the whole sequence T into k segments, consists of an optimal segmentation of the subsequence $T[1,j]$ into $k-1$ segments and a single segment that spans the subsequence $T[j+1,n]$. The optimality of the resulting segmentation can be proved for many different $\bigoplus$ operators and error functions. We focus the rest of the discussion on the cases where $\bigoplus$ is $\sum$ and E is either E_1 or E_2. For the rest of the thesis we abuse our terminology and call this instance the `Segmentation` problem. The specific error function used in every occasion will usually become clear from the context.

For an input sequence of length n, and a k-segmentation of this sequence, the dynamic programming table consists of $n \times k$ entries.

Therefore, the complexity of the dynamic programming algorithm is at least $O(nk)$. Consider now the evaluation of a single entry of the dynamic programming table. This would require checking $O(n)$ entries of the previous row plus one evaluation of the σ function for the last segment. Assume that we need time T_σ to evaluate σ for a single segment. Then, the total time required for evaluating a single entry is $O(nT_\sigma)$, and the overall complexity of the dynamic programming recursion is $O(n^2kT_\sigma)$. Consider segment $\bar{s}_i$ and error function

$$\sigma_2(s_{i-1}, s_i) = \sum_{t \in \bar{s}_i} d_2(t, \mu_{\bar{s}_i})^2 = \sum_{t \in \bar{s}_i} |t - \mu_{\bar{s}_i}|^2.$$

Then, T_σ needs $O(n)$ time and therefore the overall complexity is $O(n^3k)$.

This cubic complexity makes the dynamic programming algorithm prohibitive to use in practice. However, Equation (2.3) can be evaluated in $O(n^2k)$ total time using the following simple observation. For any segment $\bar{s}_i$ we have that

$$\begin{aligned}
\sigma_2(s_{i-1}, s_i) &= \sum_{t \in \bar{s}_i} d_2(t, \mu_{\bar{s}_i})^2 \\
&= \sum_{t \in \bar{s}_i} |t - \mu_{\bar{s}_i}|^2 \\
&= \sum_{t \in \bar{s}_i} t^2 - 2\mu_{\bar{s}_i} \sum_{t} t + |\bar{s}_i| \mu_{\bar{s}_i}^2 \\
&= \sum_{t \in \bar{s}_i} t^2 - \frac{1}{|\bar{s}_i|} \left(\sum_{t \in \bar{s}_i} t \right)^2 .
\end{aligned}$$

At each point $i \in \{1, \ldots, n\}$ we can keep the cumulative sum (cs) and the cumulative sum of squares (css) for all the points from 1 up to i. That is, for each i we have $\mathrm{cs}[i] = \sum_{j=1}^{i} t_i$ and $\mathrm{css}[i] = \sum_{j=1}^{i} t_i^2$. Then, it is straightforward that for any segment $T[i, j]$ function $\sigma_2(i, j)$ can be computed in constant time using

$$\sigma_2(i, j) = (\mathrm{css}[j] - \mathrm{css}[i-1]) - \frac{1}{j - i + 1} (\mathrm{cs}[j] - \mathrm{cs}[i-1])^2 .$$

This results in total time complexity $O(n + n^2k)$. The addition of $O(n)$ time comes from the evaluation of cs and css tables.

We now turn our attention to E_1 error function. In this case, for the evaluation of σ_1, we need to precompute the medians of each possible segment $T[i,j]$. This can be done in total $O(n^2 \log n)$ time. Consider an 1-dimensional sequence and a segment $T[i,j]$ for which we have already evaluated its median. Additionally, assume that the points in $T[i,j]$ are already sorted according to their values. Then, finding the median of the segment $T[i,j+1]$ can be done in $O(\log n)$ time by performing a binary search on the already sorted points in $T[i,j]$. This binary search is necessary for finding the position of the $(j+1)$-st point in this larger set of sorted points. Then, finding the median of the sorted set of points can be done in constant time. Therefore, the total time required for evaluating 2.3 for the E_1 error function is $O(n^2 \log n + n^2 k)$. However, for the rest of the discussion we assume that computing the medians is a preprocessing step and therefore the dynamic programming algorithm for the E_1 error function requires just $O(n^2 k)$ time.

In the `Segmentation` problem (Problem 2.1), the number of segments k is given in advance. If k is not restricted, the trivial n-segmentation achieves zero cost. A popular way for dealing with variable k is to add a penalization factor for choosing large values of k. For example, we can define the optimal segmentation to be

$$S_{\text{opt}}(T) = \arg \min_{k \in \mathbb{N}, S \in \mathcal{S}_{n,k}} E(T,S) + k\gamma,$$

where γ is the penalty for every additional segment being used. A Bayesian approach for selecting the penalty value would make it proportional to the *description length* [Ris78] of the *segmentation model*. For instance, assuming that for each segment we need to specify one boundary point and d values (one per dimension), we can choose $\gamma = (d+1)\log(dn)$. An important observation is that the same dynamic programming algorithm with no additional time overhead can be used to compute the optimal segmentation for the variable-k version of the problem. For the rest of this thesis (and mainly for clarity of exposition) we will assume that the number of segments, k, is given as part of the input.

2.3 Related work

In this section we present a review of the related work on sequence segmentation algorithms and their applications. Furthermore, we briefly discuss some variants of the `Segmentation` problem that have appeared in the literature. These variants mainly focus on providing more accurate models for existing data.

2.3.1 Algorithms and applications of sequence segmentation

Although the dynamic programming algorithm that uses Recursion (2.3) solves the `Segmentation` problem optimally, its quadratic running time makes it prohibitive for long sequences that usually appear in practice. Several faster algorithms have been proposed in the literature. These algorithms give high-quality segmentations, and have proved to be very useful in practice. The attempts for speedup include both heuristic as well as approximation algorithms.

A top-down greedy algorithm is proposed in [DP73, SZ96]. The algorithm starts with an unsegmented sequence and introduces one segment boundary at each step. The new boundary is the one that reduces the most the error of the segmentation at the current step. The running time of the algorithm is $O(nk)$. Similarly, [KS97, KP98] propose a bottom-up greedy algorithm, which starts with all points being at different segments. At each step the algorithm greedily merges segments until a segmentation with k segments is obtained. The running time of the algorithm is $O(n \log n)$.

Local search techniques for the `Segmentation` problem are proposed in [HKM$^+$01]. The core of these algorithms is the idea of starting with an arbitrary segmentation and then moving segment boundaries in order to provide better-quality segmentations. If there is no movement that can improve the quality of the segmentation the algorithms stop and output the best solution found.

In [GKS01] an approximation algorithm for the `Segmentation` problem and for the E_2 error function is provided. The idea is to suppress the entries of the optimal dynamic programming table so that each row is approximately represented. The algorithm gives solutions with error at most $(1+\epsilon)$ times that of the optimal solution, and runs in time $O(nk\frac{k}{\epsilon}\log n)$. For the segmentation problem with $\bigoplus = \max$ and E_∞ error function, [OM95] presents an $O\left(k\left(n-k\right)\right)$

algorithm. The algorithm is tailored to this specific error function and does not generalize easily to other choices of error measures.

A variant of the basic segmentation problem takes as input the sequence and a threshold value t and asks for a segmentation of the sequence such that the error of each segment is at most t. There exists a simple algorithm for dealing with this variation of the problem that is called the sliding-window (SW) algorithm. The main idea in SW is to fix the left boundary of a segment and then try to place the right boundary as far as possible. When the error of the current segment exceeds the given threshold value, then the current segment is fixed and the algorithm proceeds by introducing a new segment. The process is repeated until the end of the sequence is reached. Note that the number of segments in the resulting segmentation is not fixed in this case. Several variants of the basic SW algorithm have been proposed [KCHP01, KJM95, SZ96].

Segmentation of event sequences rather than real time series has been considered in [MS01] and [Kle02]. In event sequences each event appearing in the sequence is associated with a timestamp. The segments correspond to event intervals with homogeneous event density. In [MS01] segmentation of event sequences is found using an MCMC (Markov Chain Monte Carlo) algorithm. The algorithm tries to find the segmentation model that best represents the given data by sampling the space of all segmentation models. The work of [Kle02] assumes that the events in each interval are generated with an exponential probability distribution function. The objective function is the likelihood to fit the model plus a penalty of changing intensity intervals. Given this model, a dynamic programming algorithm is used to find the best sequence of intervals.

Several application domains have benefited from sequence segmentation algorithms or even called for new algorithmic ideas related to segmentations. For example, segmentation of genomic sequences [ARLR02, BGGC+00, BGRO96, KPV+03, LBGHG02, SKM02] is an active research area. Segmentation of timeseries data has proved useful for performing clustering and classification tasks [KP98], prediction [LM03], or for computing timeseries similarity [LKLcC03]. Finally, segmentation has been a useful tool for the analysis of sensor data streams [PVK+04] as well as data coming from context-aware devices [HKM+01].

2.3.2 Variants of the `Segmentation` problem

In this section we summarize related work that focuses on studying variants of the basic segmentation problem. The goal of such variants is to find a segmentation model that gives a better representation of the data. For example, in the *(k, h)-segmentation* problem the goal is to find a segmentation of the input sequence into k segments such that there are at most h distinct segment representatives, with $h < k$. The problem is introduced in [GM03]. Conceptually, it is a special case of the Hidden Markov Model (HMM) discovery [RJ86]. The problem is shown to be NP-hard and constant-factor approximation algorithms for the L_1 and L_2 distance metrics are provided. The core of the algorithmic techniques used is a mixture of dynamic programming and clustering algorithms.

The *unimodal segmentation* problem ([HG04]) asks for a partition of the input sequence in k segments so that the segment representatives satisfy monotonicity or unimodality constraints. The problem is shown to be solvable in polynomial time. The algorithmic solution is a combination of a well-known unimodal regression algorithm [Fri80], with simple dynamic programming.

Finally, the *basis segmentation* problem has been introduced and studied in [BGH$^+$06]. Given a multidimensional time series, the basis segmentation problem asks for a small set of latent variables and a segmentation of the series such that the segment representatives can be expressed as a linear combination of the latent variables. The set of these latent variables are called the *basis*. The work in [BGH$^+$06] presents constant-factor approximation algorithms for E_2 error measure. These algorithms combine segmentation (dynamic programming) and dimensionality-reduction techniques (Principal Component Analysis or PCA).

In [JKM99, MSV04] the authors deal formally with the problem of *segmentation with outliers*. Their objective is to find a k-segmentation of the sequence that represents well the majority of the data points and ignores the points that behave as outliers (or deviants). The problem can be solved using a dynamic programming algorithm for one-dimensional sequences and E_2 error function. The complexity of this problem for higher dimensions remains unknown.

CHAPTER 3

Approximate algorithms for sequence segmentation

In Chapter 2 we showed that the `Segmentation` problem can be solved optimally in $O(n^2k)$ time using dynamic programming, where n is the number of points of the input sequence. In this chapter, we present a new Divide&Segment (DnS) algorithm for the `Segmentation` problem. The DnS algorithm has subquadratic running time, $O(n^{4/3}k^{5/3})$, and it is a 3-approximation algorithm for E_1 and E_2 error functions. That is, the error of the segmentation it produces is provably no more than 3 times that of the optimal segmentation. Additionally, we explore several more efficient variants of the algorithm and we quantify the accuracy/efficiency tradeoff. More specifically, we define a variant that runs in time $O(n \log\log n)$ and has approximation ratio $O(\log n)$. All these algorithms have sublinear space complexity, which makes them applicable to problems where the data needs to be processed in an streaming fashion. We also propose an algorithm that requires logarithmic space and linear time, albeit, with no approximation guarantees. Parts of this chapter have already been presented in [TT06].

Experiments on both real and synthetic datasets demonstrate that in practice our algorithms perform significantly better than their worst case theoretical upper bounds. In many cases our algorithms give results equivalent to the optimal algorithm. We also compare the proposed algorithms against popular heuristics that are known to work well in practice. Finally, we show that the proposed algorithms can be applied to variants of the basic segmenta-

tion problem, like for example the one defined in [GM03]. We show that for this problem we achieve similar speedups for the existing approximation algorithms, while maintaining constant approximation factors.

In this chapter, we mainly focus on the `Segmentation` problem for E_1 and E_2 error functions. The input is assumed to be a discrete, real-valued sequence of length n. That is, we mainly focus on the following problem.

Problem 3.1 (The `Segmentation` problem) *Given a sequence $T \in \mathcal{T}_n$, find segmentation $S \in \mathcal{S}_{n,k}$ that minimizes the error*

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

with $p = 1$ or $p = 2$.

3.1 The Divide&Segment algorithm

In this section we describe the Divide&Segment (DnS) algorithm for Problem 3.1. The algorithm is faster than the standard dynamic programming algorithm and its approximation factor is constant. The main idea of the algorithm is to divide the problem into smaller subproblems, solve the subproblems optimally and combine their solutions to form the final solution. Recurrence (2.3) is a building component of DnS. The output of the algorithm is a k-segmentation of the input sequence. Algorithm 1 outlines DnS. In step 1, the input sequence T is partitioned into χ disjoint subsequences. Each subsequence is segmented optimally using dynamic programming. For subsequence T_i, the output of this step is a segmentation S_i of T_i and a set M_i of k weighted points. These are the representatives of the segments of segmentation S_i, weighted by the length of the segment they represent. All the χk representatives of the χ subsequences are concatenated to form the (weighted) sequence T'. Then, the dynamic programming algorithm is applied on T'. The k-segmentation of T' is output as the final segmentation.

The following example illustrates the execution of DnS.

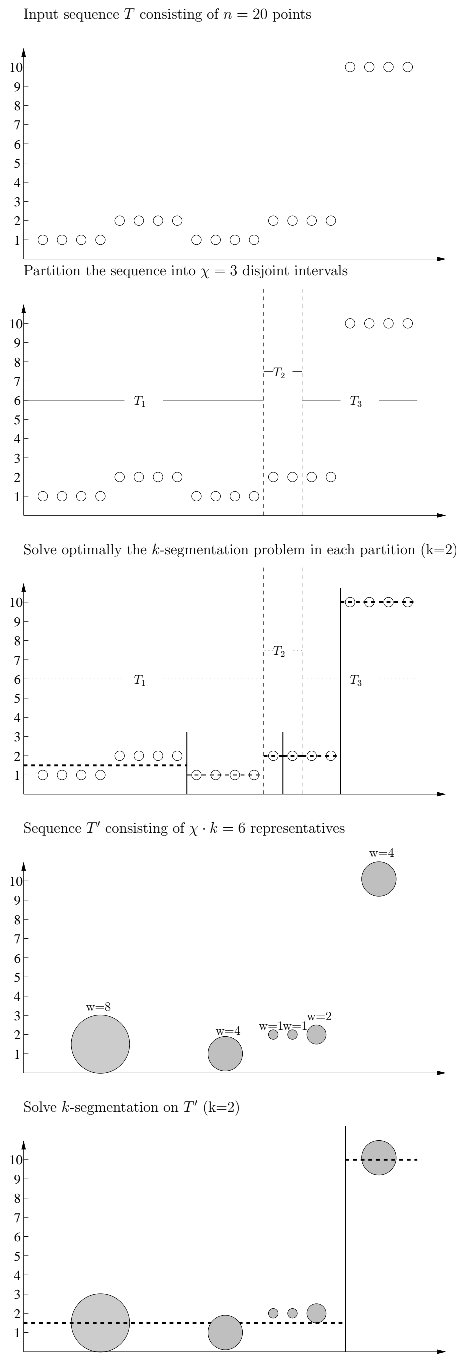


Figure 3.1: Illustration of the DNS algorithm.

Algorithm 1 The DnS algorithm

Input: Sequence T of n points, number of segments k, value χ.
Ouput: A segmentation of T into k segments.
1: Partition T arbitrarily into χ disjoint intervals $T_1, \ldots, T_\chi$.
2: **for all** $i \in \{1, \ldots, \chi\}$ **do**
3: $(S_i, M_i) = \mathrm{DP}(T_i, k)$
4: **end for**
5: Let $T' = M_1 \oplus M_2 \oplus \cdots \oplus M_\chi$ be the sequence defined by the concatenation of the representatives, weighted by the length of the interval they represent.
6: Return the optimal segmentation of (S,M) of T' using the dynamic programming algorithm.

Example 3.1 *Consider the time series of length* $n = 20$ *that is shown in Figure 3.1. We show the execution of the* DnS *algorithm for* $k = 2$*, using* $\chi = 3$*. In step 1 the sequence is divided into three disjoint and contiguous intervals* T_1*,* T_2 *and* T_3*. Subsequently, the dynamic programming algorithm is applied to each one of those intervals. The result of this are the six weighted points on which dynamic programming is applied again. For this input sequence, the output* 2*-segmentation found by the* DnS *algorithm is the same as the optimal segmentation.*

The running time of the algorithm is easy to analyze.

Theorem 3.1 *The running time of the* DnS *algorithm is* $O(n^{4/3}k^{5/3})$ *for* $\chi = (\frac{n}{k})^{2/3}$.

Proof. Assume that DnS partitions T into χ equal-length intervals. The running time of the DnS algorithm as a function of χ is

$$\begin{aligned} R(\chi) &= \chi \left(\frac{n}{\chi}\right)^2 k + (\chi k)^2 k \\ &= \frac{n^2}{\chi} k + \chi^2 k^3. \end{aligned}$$

The minimum of function $R(\chi)$ is achieved when $\chi_0 = \left(\frac{n}{k}\right)^{\frac{2}{3}}$ and this gives $R(\chi_0) = 2n^{4/3}k^{5/3}$. □

We note that the DnS algorithm can also be used in the case where the data must be processed in a streaming fashion. Assuming that we have an estimate of the size of the sequence n, then the algorithm processes the points in batches of size n/χ. For each such batch it computes the optimal k-segmentation, and stores the representatives. The space required is $M(\chi) = n/\chi + \chi k$. This is minimized for $\chi = \sqrt{n/k}$, resulting in space $M = 2\sqrt{nk}$.

3.1.1 Analysis of the DnS algorithm.

For the proof of the approximation factor of the DnS algorithm we first show the following lemma.

Lemma 3.1 *Let $S_i = S_{opt}(T_i, k)$, for $i = 1, \ldots, \chi$, and $S_{opt} = S_{opt}(T, k)$. If $\bar{t}$ is the representative assigned to point $t \in T$ by segmentation S_i after the completion of the for loop (Step 2) of the* DnS *algorithm, then we have*

$$\sum_{t \in T} d_p(t, \bar{t})^p = \sum_{i=1}^{\chi} E_p\left(T_i, S_i\right)^p \leq E_p\left(T, S_{opt}\right)^p.$$

Proof. For each interval T_i consider the segmentation points of S_{opt} that lie within T_i. These points together with the starting and ending points of interval T_i define a segmentation of T_i into k'_i segments with $k'_i \leq k$. Denote this segmentation by S'_i. Then, for every interval T_i and its corresponding segmentation S'_i defined as above we have $E_p(T_i, S_i) \leq E_p(T_i, S'_i)$. This is true since S_i is the optimal k-segmentation for subsequence T_i and $k'_i \leq k$. Thus we have

$$E_p\left(T_i, S_i\right)^p \leq E_p\left(T_i, S'_i\right)^p.$$

Summing over all T_i's we get

$$\begin{aligned}
\sum_{t \in T} d_p(t, \bar{t})^p &= \sum_{i=1}^{\chi} E_p\left(T_i, S_i\right)^p \\
&\leq \sum_{i=1}^{\chi} E_p\left(T_i, S'_i\right)^p \\
&= E_p\left(T, S_{\text{opt}}\right)^p.
\end{aligned}$$

□

We are now ready to prove the approximation factors for E_1 and E_2 error measures.

Theorem 3.2 *For a sequence T and error measure E_1 let* $\textsc{Opt}_1 = E_1(S_{opt}(T,k))$ *be the E_1-error for the optimal k-segmentation. Also let* $\textsc{DnS}_1$ *be the E_1-error for the k-segmentation output by the* DnS *algorithm. We have that* $\textsc{DnS}_1 \leq 3\,\textsc{Opt}_1$.

Proof. Let S be the segmentation of sequence T output by the $\textsc{DnS}(T,k,\chi)$ algorithm, and let μ_t be the representative assigned to some point $t \in T$ in S. Also, let λ_t denote the representative of t in the optimal segmentation $S_{\text{opt}}(T,k)$. The E_1-error of the optimal segmentation is

$$\textsc{Opt}_1 = E_1(S_{\text{opt}}(T,k)) = \sum_{t\in T} d_1(t,\lambda_t)\ .$$

The E_1 error of the DnS algorithm is given by

$$\textsc{DnS}_1 = E_1(T,S) = \sum_{t\in T} d_1(t,\mu_t).$$

Now let $\bar{t}$ be the representative of the segment to which point t is assigned after the completion of the for loop in Step 2 of the DnS algorithm. Due to the optimality of the dynamic programming algorithm in Step 4 we have

$$\sum_{t\in T} d_1(\bar{t},\mu_t) \quad \leq \quad \sum_{t\in T} d_1(\bar{t},\lambda_t)\ . \tag{3.1}$$

We can now obtain the desired result as follows

$$\begin{aligned}
\textsc{DnS}_1 &= \sum_{t\in T} d_1(t,\mu_t) \\
&\leq \sum_{t\in T} \big(d_1(t,\bar{t}) + d_1(\bar{t},\mu_t)\big) &(3.2)\\
&\leq \sum_{t\in T} \big(d_1(t,\bar{t}) + d_1(\bar{t},\lambda_t)\big) &(3.3)\\
&\leq \sum_{t\in T} \big(d_1(t,\bar{t}) + d_1(\bar{t},t) + d_1(t,\lambda_t)\big) &(3.4)\\
&\leq 2\sum_{t\in T} d_1(t,\lambda_t) + \sum_{t\in T} d_1(t,\lambda_t) &(3.5)\\
&= 3\,\textsc{Opt}_1\ .
\end{aligned}$$

Inequalities (3.2) and (3.4) follow from the triangular inequality, inequality (3.3) follows from Equation (3.1), and inequality (3.5) follows from Lemma 3.1. □

Next we prove the 3-approximation result for E_2. For this, we need the following simple fact.

Fact 3.1 *(DOUBLE TRIANGULAR INEQUALITY) Let d be a distance metric. Then for points x, y and z and $p \in \mathbb{N}^+$ we have*

$$d(x,y)^2 \leq 2\,d(x,z)^2 + 2\,d(z,y)^2 \ .$$

Theorem 3.3 *For a sequence T and error measure E_2, let $\textsc{Opt}_2 = E_2(S_{opt}(T,k))$ be the E_2-error for the optimal k-segmentation. Also let $\textsc{DnS}_2$ be the E_2-error for the k-segmentation output by the DNS algorithm. We have $\textsc{DnS}_2 \leq 3\,\textsc{Opt}_2$.*

Proof. Consider the same notation as in Theorem 3.2. The E_2 error of the optimal dynamic programming algorithm is

$$\textsc{Opt}_2 = E_2(S_{\text{opt}}(T,k)) = \sqrt{\sum_{t\in T} d_2(t,\lambda_t)^2} \ .$$

Let S be the output of the $\textsc{DnS}(T,k,\chi)$ algorithm. The error of the DNS algorithm is given by

$$\textsc{DnS}_2 = E_2(T,S) = \sqrt{\sum_{t\in T} d_2(t,\mu_t)^2} \ .$$

The proof continues along the same lines as the proof of Theorem 3.2 but uses Fact 3.1 and Cauchy-Schwarz inequality. Using the triangular inequality of d_2 we get

$$\begin{aligned}
\textsc{DnS}_2^2 &= \sum_{t\in T} d_2(t,\mu_t)^2 \\
&\leq \sum_{t\in T} \left(d_2(t,\bar{t}) + d_2(\bar{t},\mu_t)\right)^2 \\
&= \sum_{t\in T} d_2(t,\bar{t})^2 + \sum_{t\in T} d_2(\bar{t},\mu_t)^2 \\
&\quad +2\sum_{t\in T} d_2(t,\bar{t})\, d_2(\bar{t},\mu_t) \ .
\end{aligned}$$

From Lemma 3.1 we have that

$$\sum_{t\in T} d_2(t,\bar{t})^2 \leq \sum_{t\in T} d_2(t,\lambda_t)^2 = \textsc{Opt}_2^2 .$$

Using the above inequality, the optimality of dynamic programming in Step 4 of the algorithm, and Fact 3.1 we have

$$\begin{aligned}
\sum_{t\in T} d_2(\bar{t},\mu_t)^2 &\leq \sum_{t\in T} d_2(\bar{t},\lambda_t)^2 \\
&\leq 2\sum_{t\in T}\left(d_2(\bar{t},t)^2 + d_2(t,\lambda_t)^2\right) \\
&\leq 4\sum_{t\in T} d_2(t,\lambda_t)^2 \\
&= 4\,\textsc{Opt}_2^2 .
\end{aligned}$$

Finally using the Cauchy-Schwarz inequality we get

$$\begin{aligned}
2\sum_{t\in T} d_2(t,\bar{t})\, d_2(\bar{t},\mu_t) &\leq 2\sqrt{\sum_{t\in T} d_2(t,\bar{t})^2}\sqrt{\sum_{t\in T} d_2(\bar{t},\mu_t)^2} \\
&\leq 2\sqrt{\textsc{Opt}_2^2}\,\sqrt{4\,\textsc{Opt}_2^2} \\
&= 4\,\textsc{Opt}_2^2 .
\end{aligned}$$

Combining all the above we conclude that

$$\textsc{DnS}_2^2 \leq 9\,\textsc{Opt}_2^2 .$$

□

3.2 Recursive DnS algorithm

The DnS algorithm applies the "divide-and-segment" idea once, splitting the sequence into subsequences, partitioning each of subsequence optimally, and then merging the results. We now consider the recursive DnS algorithm (RDnS) which recursively splits each of the subsequences, until no further splits are possible. Algorithm 2 shows the outline of the RDnS algorithm.

The value B is a constant that defines the base case for the recursion. Alternatively, one could directly determine the depth ℓ of

---

Algorithm 2 The RDnS algorithm

Input: Sequence T of n points, number of segments k, value χ.
Ouput: A segmentation of T into k segments.

1: **if** $|T| \leq B$ **then**
2: Return the optimal partition (S,M) of T using the dynamic programming algorithm.
3: **end if**
4: Partition T into χ intervals $T_1, \ldots, T_\chi$.
5: **for all** $i \in \{1, \ldots, \chi\}$ **do**
6: $(S_i, M_i) = \text{RDnS}(T_i, k, \chi)$
7: **end for**
8: Let $T' = M_1 \oplus M_2 \oplus \cdots \oplus M_\chi$ be the sequence defined by the concatenation of the representatives, weighted by the length of the interval they represent.
9: Return the optimal partition (S,M) of T' using the dynamic programming algorithm.

---

the recursive calls to RDnS. We will refer to such an algorithm, as the ℓ-RDnS algorithm. For example, the simple DnS algorithm is 1-RDnS. We also note that at every recursive call of the RDnS algorithm the number χ of intervals into which we partition the sequence may be a function of sequence length. However, for simplicity we use χ instead of $\chi(n)$.

As a first step in the analysis of the RDnS we consider the approximation ratio of the ℓ-RDnS algorithm. We can prove the following theorem.

Theorem 3.4 *The ℓ-*RDnS *algorithm is an $O(2^\ell)$ approximation algorithm for the E_1-error function for Problem 2.1.*

Proof. The proof follows by induction on the value of ℓ. The exact approximation ratio is $2^{\ell+1} - 1$ for E_1 and the proof goes as follows.

From Theorem 3.2, we have that the theorem is true for $\ell = 1$. Assume now that it is true for some $\ell \geq 1$. We will prove it for $\ell + 1$. At the first level of recursion the $(\ell + 1)$-RDnS algorithm, breaks the sequence T into χ subsequences $T_1, \ldots, T_\chi$. For each one of these we call the ℓ-RDnS algorithm, producing a set R of χk representatives. Similar to the proof of Theorem 3.2, let $\bar{t} \in R$

denote the representative in R that corresponds to point t. Consider also the optimal segmentation of each of these intervals, and let O denote the set of χk representatives. Let $\widetilde{t} \in O$ denote the representative of point t in O. From the inductive hypothesis we have that

$$\sum_{t\in T} d_1(t,\bar{t}) \leq \left(2^{\ell+1}-1\right)\sum_{t\in T} d_1(t,\widetilde{t}).$$

Now let μ_t be the representative of point t in the segmentation output by the $(\ell+1)$-RDnS algorithm. Also let λ_t denote the representative of point t in the optimal segmentation. Let RDnS_1 denote the E_1-error of the $(\ell+1)$-RDnS algorithm, and Opt_1 denote the E_1-error of the optimal segmentation. We have that

$$\text{RDnS}_1 = \sum_{t\in T} d_1(t,\mu_t) \quad \text{and} \quad \text{Opt}_1 = \sum_{t\in T} d_1(t,\lambda_t).$$

From the triangular inequality we have that

$$\begin{aligned}\sum_{t\in T} d_1(t,\mu_t) &\leq \sum_{t\in T} d_1(t,\bar{t}) + \sum_{t\in T} d_1(\bar{t},\mu_t)\\ &\leq \left(2^{\ell+1}-1\right)\sum_{t\in T} d_1(t,\widetilde{t}) + \sum_{t\in T} d_1(\bar{t},\mu_t).\end{aligned}$$

From Lemma 3.1, and Equation (3.1), we have that

$$\sum_{t\in T} d_1(t,\widetilde{t}) \leq \sum_{t\in T} d_1(t,\lambda_t)$$

and

$$\sum_{t\in T} d_1(\bar{t},\mu_t) \leq \sum_{t\in T} d_1(\bar{t},\lambda_t).$$

Using the above inequalities and the triangular inequality we obtain

$$\begin{aligned}
\text{RDnS}_1 &= \sum_{t\in T} d_1(t,\mu_t) \\
&\leq \left(2^{\ell+1}-1\right)\sum_{t\in T} d_1(t,\lambda_t) + \sum_{t\in T} d_1(\bar{t},\lambda_t) \\
&\leq \left(2^{\ell+1}-1\right)\sum_{t\in T} d_1(t,\lambda_t) \\
&\quad + \sum_{t\in T} d_1(t,\bar{t}) + \sum_{t\in T} d_1(t,\lambda_t) \\
&\leq 2^{\ell+1}\sum_{t\in T} d_1(t,\lambda_t) + \left(2^{\ell+1}-1\right)\sum_{t\in T} d_1(t,\widetilde{t}) \\
&\leq \left(2^{\ell+2}-1\right)\sum_{t\in T} d_1(t,\lambda_t) \\
&= \left(2^{\ell+2}-1\right)\text{Opt}_1
\end{aligned}$$

□

The corresponding result for the E_2 follows similarly (see Theorem 3.5). Instead of using the binomial identity as in the proof of Theorem 3.3, we obtain a simpler recursive formula for the approximation error by applying the double triangular inequality.

Theorem 3.5 *The ℓ-*RDnS *algorithm is an $O(6^{\ell/2})$ approximation algorithm for the E_2-error function for Problem 2.1.*

Proof. The exact approximation ratio of the ℓ-RDnS algorithm for E_2-error function is $\sqrt{\frac{9}{5}6^\ell - \frac{4}{5}}$. We prove the theorem by induction on the levels of recursion ℓ. From Theorem 3.3 the claim holds for $\ell = 1$. Assume now that the claim is true for $\ell \geq 1$. We will prove it for $\ell + 1$. At the first level of recursion, the $(\ell+1)$-RDnS algorithm breaks sequence T into χ subsequences $T_1, \ldots, T_\chi$. For each one of these we recursively call the ℓ-RDnS algorithm that produces a set R of χk representatives. Let $\bar{t} \in R$ denote the representative of point t in R. Consider also the optimal segmentation of these intervals and denote by O the set of the χk optimal representatives. As before we denote by $\widetilde{t} \in O$ the representative of point t in O. From the inductive hypothesis we have that

$$\sum_{t\in T} d_2(t,\bar{t})^2 \leq \left(\frac{9}{5}6^\ell - \frac{4}{5}\right)\sum_{t\in T} d_2(t,\widetilde{t})^2.$$

Now let μ_t be the representative of point t in the segmentation output by the $(\ell+1)$-RDnS algorithm and λ_t the representative of point t in the optimal segmentation. If RDnS_2 denotes the E_2-error of the $(\ell+1)$-RDnS algorithm, and Opt_2 denote the E_2-error of the optimal segmentation. We have that

$$\text{RDnS}_2^2 = \sum_{t \in T} d_2(t, \mu_t)^2 \quad \text{and} \quad \text{Opt}_2^2 = \sum_{t \in T} d_2(t, \lambda_t)^2.$$

From the double triangular inequality (Fact 3.1) we have that

$$\begin{aligned}
\sum_{t \in T} d_2(t, \mu_t)^2 &\leq 2 \sum_{t \in T} d_2(t, \bar{t})^2 + 2 \sum_{t \in T} d_2(\bar{t}, \mu_t)^2 \\
&\leq 2 \left(\frac{9}{5} 6^\ell - \frac{4}{5} \right) \sum_{t \in T} d_2(t, \widetilde{t})^2 + 2 \sum_{t \in T} d_2(\bar{t}, \mu_t)^2.
\end{aligned}$$

From Lemma 3.1 we have that

$$\sum_{t \in T} d_2(t, \widetilde{t})^2 \leq \sum_{t \in T} d_2(t, \lambda_t)^2,$$

and from the optimality of the last step of the algorithm we have

$$\sum_{t \in T} d_2(\bar{t}, \mu_t)^2 \leq \sum_{t \in T} d_2(\bar{t}, \lambda_t)^2.$$

Using the above inequalities and the double triangular inequality we obtain

$$\begin{aligned}
\text{RDnS}_2^2 &= \sum_{t\in T} d_2(t,\mu_t)^2 \\
&\leq 2\left(\frac{9}{5}6^\ell - \frac{4}{5}\right)\sum_{t\in T} d_2(t,\lambda_t)^2 + 2\sum_{t\in T} d_2(\bar{t},\lambda_t)^2 \\
&\leq 2\left(\frac{9}{5}6^\ell - \frac{4}{5}\right)\sum_{t\in T} d_2(t,\lambda_t)^2 \\
&\quad +2\left(2\sum_{t\in T} d_2(t,\bar{t})^2 + 2\sum_{t\in T} d_2(t,\lambda_t)^2\right) \\
&\leq 2\left(\frac{9}{5}6^\ell - \frac{4}{5}\right)\sum_{t\in T} d_2(t,\lambda_t)^2 \\
&\quad +4\left(\frac{9}{5}6^\ell - \frac{4}{5}\right)\sum_{t\in T} d_2(t,\widetilde{t})^2 + 4\sum_{t\in T} d_2(t,\lambda_t)^2 \\
&\leq \left(\frac{9}{5}6^{\ell+1} - \frac{4}{5}\right)\sum_{t\in T} d_2(t,\lambda_t)^2 \\
&= \left(\frac{9}{5}6^{\ell+1} - \frac{4}{5}\right)\text{Opt}_2^2.
\end{aligned}$$

□

We now consider possible values for χ. First, we set χ to be a constant. We can prove the following theorem.

Theorem 3.6 *For any constant χ the running time of the* RDnS *algorithm is $O(n)$, where n is the length of the input sequence. The algorithm can operate on data that arrive in streaming fashion using $O(\log n)$ space.*

Proof. The running time of the RDnS algorithm is given by the recursion

$$R(n) = \chi R\left(\frac{n}{\chi}\right) + (\chi k)^2 k.$$

Solving the recursion gives $R(n) = O(n)$.

When the data arrive in a stream, the algorithm can build the recursion tree online, in a bottom-up fashion. We only need to maintain at most χk representatives at every level of the recursion

tree. The depth of the recursion is $O(\log n)$, resulting in $O(\log n)$ space overall. □

Therefore, for constant χ, we obtain an efficient algorithm, both in time and space. Unfortunately, we do not have any approximation guarantees, since the best approximation bound we can prove using Theorems 3.4 and 3.5 is $O(n)$. We can however obtain significantly better approximation guarantees if we are willing to tolerate a small increase in the running time. We set $\chi = \sqrt{n}$, where n is the length of the input sequence at each specific recursive call. That is, at each recursive call we split the sequence into $\sqrt{n}$ pieces of size $\sqrt{n}$.

Theorem 3.7 *For $\chi = \sqrt{n}$ the* RDnS *algorithm is an $O(\log n)$ approximation algorithm for Problem 2.1 for both E_1 and E_2 error functions. The running time of the algorithm is $O(n \log\log n)$, using $O(\sqrt{n})$ space, when operating in a streaming fashion.*

Proof. It is not hard to see that after ℓ recursive calls the size of the input segmentation is $O(n^{1/2^\ell})$. Therefore, the depth of the recursion is $O(\log\log n)$. From Theorems 3.4 and 3.5 we have that the approximation ratio of the algorithm is $O(\log n)$. The running time of the algorithm is given by the recurrence

$$R(n) = \sqrt{n}R\left(\sqrt{n}\right) + nk^3.$$

Solving the recurrence we obtain running time $O(n \log\log n)$. The space required is bounded by the size of the top level of the recursion, and it is $O(\sqrt{n})$. □

The following corollary is an immediate consequence of the proof of Theorem 3.7 and it provides an accuracy/efficiency tradeoff.

Corollary 3.1 *For $\chi = \sqrt{n}$, the ℓ-*RDnS *algorithm is an $O(2^\ell)$ approximation algorithm for the E_1-error function, and an $O(6^{\ell/2})$ approximation algorithm for the E_2-error function, with respect to Problem 2.1. The running time of the algorithm is $O(n^{1+1/2^\ell} + n\ell)$.*

3.3 Experiments

In this section we compare experimentally the different "divide-and-segment" algorithms with other segmentation algorithms proposed in the literature.

3.3.1 Segmentation heuristics

Since sequence segmentation is a basic problem particularly in time-series analysis, several algorithms have been proposed in the literature with the intention to improve the running time of the optimal dynamic programming algorithm. These algorithms have proved to be very useful in practice. However, no approximation bounds are known for them. For completeness we briefly describe them here.

The Top-Down greedy algorithm (TD) starts with the unsegmented sequence (initially there is just a single segment) and it introduces a new boundary at every greedy step. That is, in the i-th step it introduces the i-th segment boundary by splitting one of the existing i segments into two. The new boundary is selected in such a way that it minimizes the overall error. No change is made to the existing $i-1$ boundary points. The splitting process is repeated until the number of segments of the output segmentation reaches k. This algorithm, or variations of it with different stopping conditions are used in [BGRO96, DP73, LSL$^+$00, SZ96]. The running time of the algorithm is $O(nk)$.

In the Bottom-Up greedy algorithm (BU) initially each point forms a segment on its own. At each step, two consecutive segments that cause the smallest increase in the error are merged. The algorithm stops when k segments are formed. The complexity of the bottom-up algorithm is $O(n \log n)$. BU performs well in terms of error and it has been used widely in timeseries segmentation [HG04, PVK$^+$04].

The Local Iterative Replacement (LiR) and Global Iterative Replacement (GiR) are randomized algorithms for sequence segmentations proposed in [HKM$^+$01]. Both algorithms start with a random k-segmentation. At each step they pick one segment boundary (randomly or in some order) and search for the best position to put it back. The algorithms repeat these steps until they converge, i.e., they cannot improve the error of the output segmentation. The two algorithms differ in the types of replacements of the segmentation boundaries they are allowed to do. Consider a segmentation $s_1, s_2, \ldots, s_k$. Now assume that both LiR and GiR pick segment boundary s_i for replacement. LiR is only allowed to put a new boundary between points s_{i-1} and s_{i+1}. GiR is allowed to put a new segment boundary anywhere on the sequence. Both

algorithms run in time $O(In)$, where I is the number of iterations necessary for convergence.

Although extensive experimental evidence shows that these algorithms perform well in practice, there is no known guarantee of their worst-case error ratio.

3.3.2 Experimental setup

For the experimental study we compare the family of "divide-and-segment" algorithms with all the heuristics described in the previous section. We also explore the quality of the results given by RDnS compared to DnS for different parameters of the recursion (i.e., number of recursion levels, value of χ).

For the study we use two types of datasets: (a) synthetic and (b) real data. The synthetic data are generated as follows: First we 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 variance varying from 0.05 to 0.9.

The real datasets were downloaded from the UCR timeseries data mining archive [KF02][1].

3.3.3 Performance of the DnS algorithm

Figure 3.2 shows the performance of different algorithms for the synthetic datasets. In particular, we plot the error ratio $\frac{\text{A}}{\text{Opt}}$ for A being the error of the solutions found by the algorithms DnS, BU, TD, LiR and GiR. With Opt we denote the error of the optimal solution. The error ratio is shown as a function of the number of segments. In all cases, the DnS algorithm consistently outperforms all other heuristics, and the error it achieves is very close to that of the optimal algorithm. Note that in contrast to the steady behavior

[1] The interested reader can find the datasets at http://www.cs.ucr.edu/~eamonn/TSDMA/.

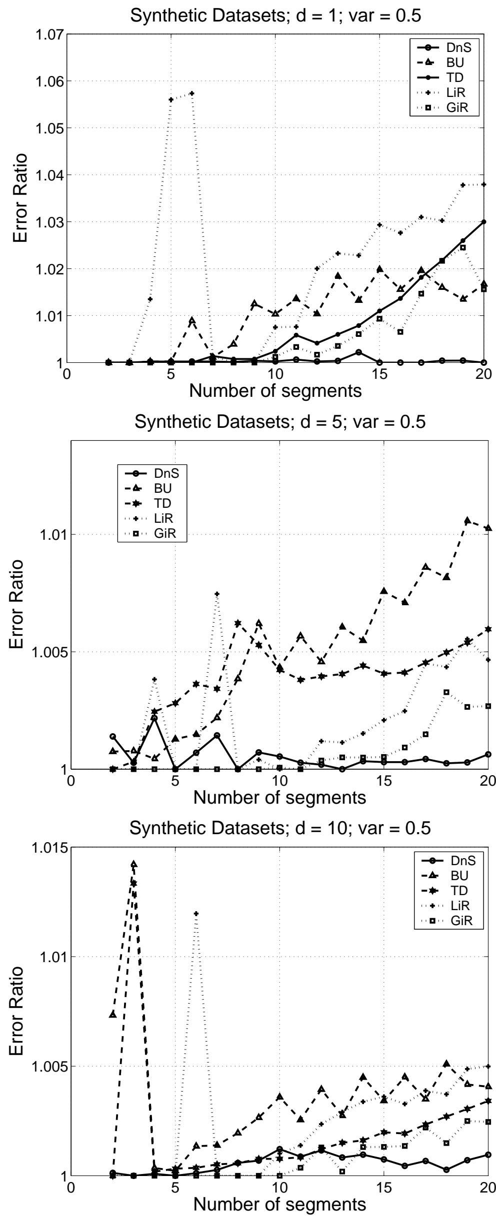


Figure 3.2: Synthetic datasets: error ratio of DnS, BU, TD, LiR and GiR algorithms with respect to Opt as a function of the number of segments.

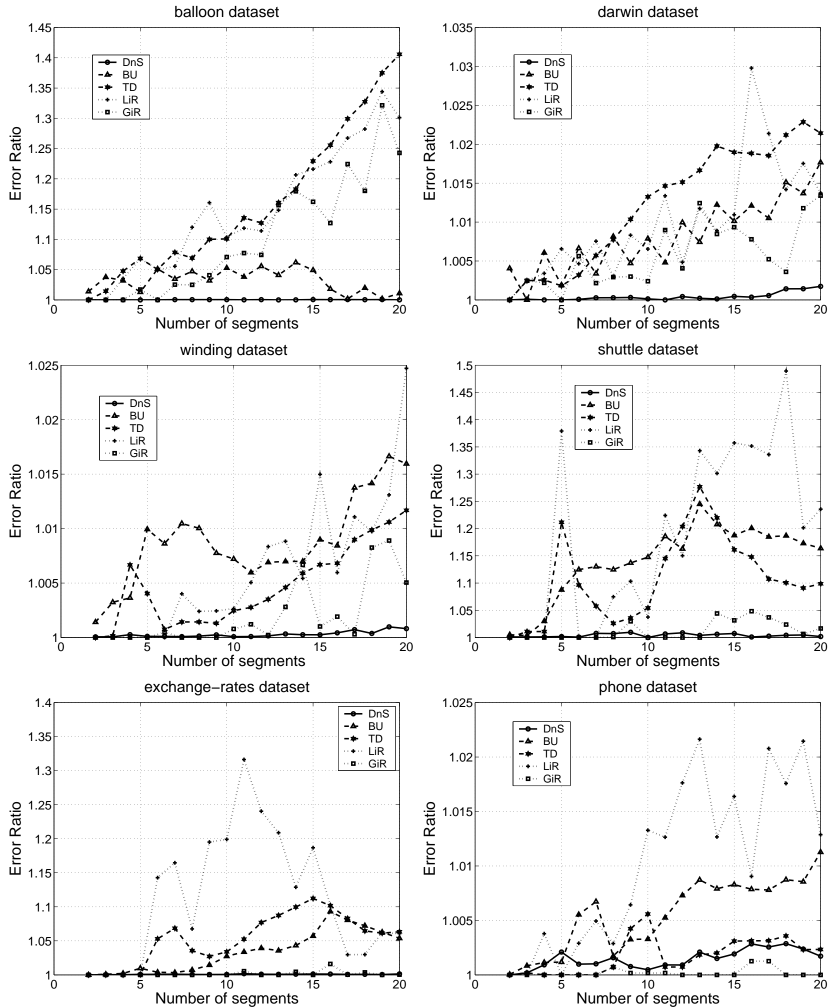


Figure 3.3: Real datasets: error ratio of DnS, BU, TD, LiR and GiR algorithms with respect to Opt as a function of the number of segments.

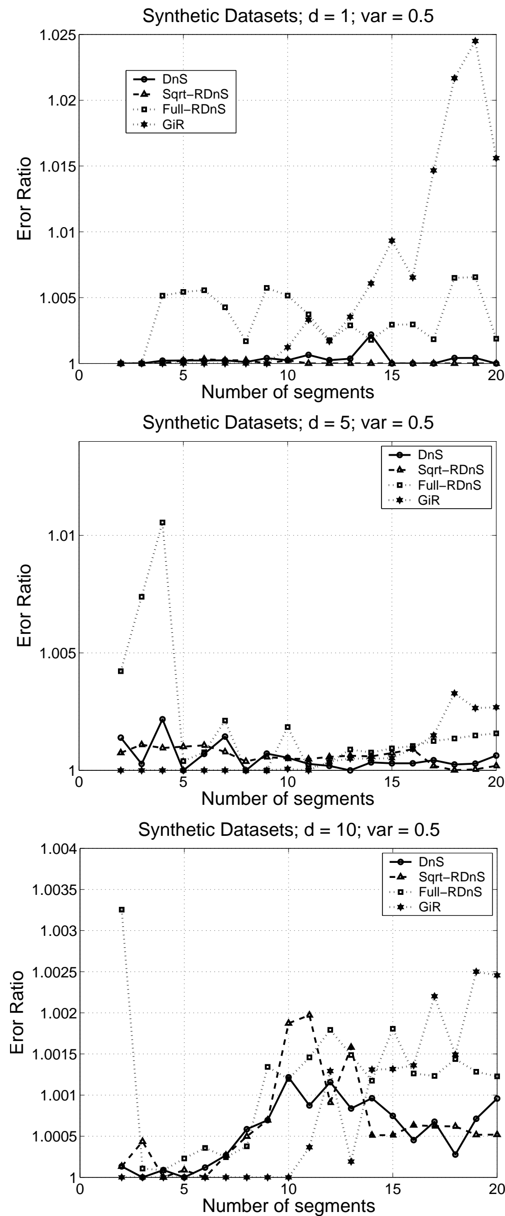


Figure 3.4: Synthetic datasets: error ratio of DnS and RDnS algorithms with respect to Opt as a function of the number of segments.

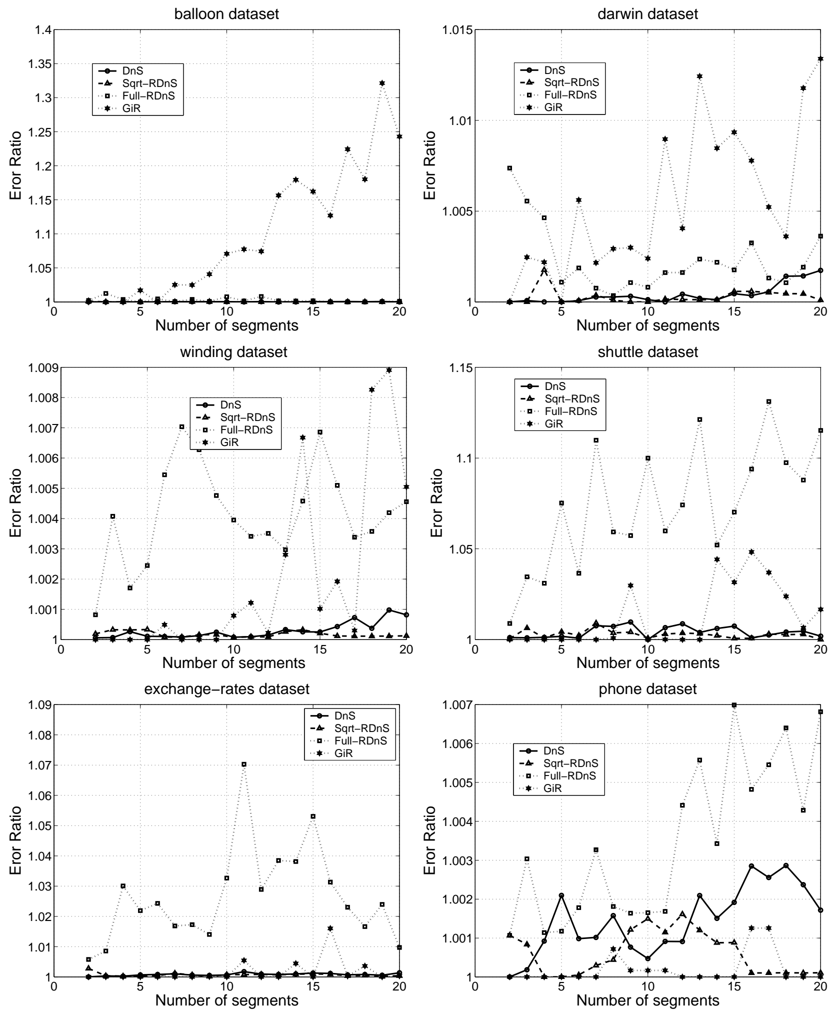


Figure 3.5: Real datasets: error ratio of DnS and RDnS algorithms with respect to Opt as a function of the number of segments.

of DnS the quality of the results of the other heuristics varies for the different parameters and no conclusions on their behavior on arbitrary datasets can be drawn.

This phenomenon is even more pronounced when we experiment with real data. Figure 3.3 is a sample of similar experimental results obtained using the datasets *balloon, darwin, winding, xrates* and *phone* from the UCR repository. The DnS performs extremely well in terms of accuracy, and it is again very robust across different datasets for different values of k. Mostly, GiR performs the best among the rest of the heuristics. However, there are cases (e.g., the balloon dataset) where GiR is severely outperformed.

3.3.4 Exploring the benefits of the recursion

We additionally compare the basic DnS algorithm with different versions of RDnS. The first one, Full-RDnS (full recursion), is the RDnS algorithm when we set the value of χ to be a constant. This algorithm runs in linear time (see Theorem 3.6). However, we have not derived any approximation bound for it (other than $O(n)$). The second one, Sqrt-RDnS, is the RDnS algorithm when we set χ to be $\sqrt{n}$. At every recursive call of this algorithm the parental segment of size s is split into $O(\sqrt{s})$ subsegments of the same size. This variation of the recursive algorithm runs in time $O(n \log \log n)$ and has approximation ratio $O(\log n)$ (see Theorem 3.7). We study experimentally the tradeoffs between the running time and the quality of the results obtained using the three different alternatives of "divide-and-segment" methods on synthetic and real datasets. We also compare the quality of those results with the results obtained using GiR algorithm. We choose this algorithm for comparison since it has proved to be the best among all the other heuristics. In Figure 3.4 we plot the error ratio of the algorithms as a function of the number of segments and the variance for the synthetic datasets. Figure 3.5 shows the experiments on real datasets.

From the results we can make the following observations. First, all the algorithms of the divide-and-segment family perform extremely well, giving results close to the optimal segmentation and usually better than the results obtained by GiR. The full recursion (Full-RDnS) can harm the quality of the results. However, we note that in order to study the full effect of recursion on the

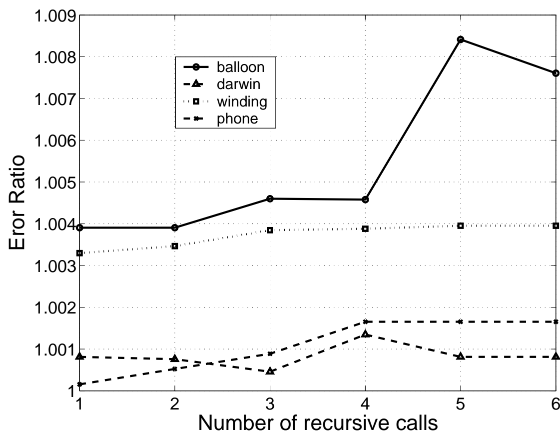


Figure 3.6: Real datasets: error ratio of ℓ-RDnS algorithm with respect to Opt as a function of the number recursion calls.

performance of the algorithm we set $\chi = 2$, the smallest possible value. We believe that for larger values of χ the performance of Full-RDnS will be closer to that of DnS (for which we have $\chi = (n/k)^{2/3}$). Finally, there are cases where Sqrt-RDnS (and in some settings Full-RDnS) performs even better than simple DnS. This phenomenon is due to the difference in the number and the positions of the splitting points the two algorithms pick for the division step. It appears that, in some cases, performing more levels of recursion helps the algorithm to identify better segment boundaries, and thus produce segmentations of lower cost.

Figure 3.6 shows how the error of the segmentation output by ℓ-RDnS changes for different number of recursion levels, for four real datasets (*balloon, darwing, phone* and *winding*). Note that even for 5 levels of recursion the ratio never exceeds 1.008.

3.4 Applications to the (k, h)-segmentation problem

Here, we discuss the application of the simple DnS algorithm for a variant of the `Segmentation` problem, namely the problem of finding the optimal (k, h)-segmentation introduced in [GM03]. Given a sequence, the (k, h)-segmentation of the sequence is a k-segmentation

that uses only $h < k$ distinct representatives. We have picked this problem to demonstrate the usefulness of the DnS algorithm because of the applicability of (k, h)-segmentation to the analysis of long genetic sequences. For that kind of analysis, efficient algorithms for the (k, h)-segmentation problem are necessary.

Let S be a (k, h)-segmentation of the sequence T. For each segment $\bar{s}$ of the segmentation S, let ℓ_s be the representative for this segment (there are at most h representatives). The error E_p of the (k, h)-segmentation is defined as follows

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

Let $\mathcal{S}_{n,k,h}$ denote the family of all segmentations of sequences of length n into k segments using h representatives. In a similar way to the k-segmentation, for a given sequence T of length n and error measure E_p, and for given $k, h \in \mathbb{N}$ with $h < k$, the optimal (k, h)-segmentation is defined as

$$S_{\text{opt}}(T, k, h) = \arg \min_{S \in \mathcal{S}_{n,k,h}} E_p\left(T, S\right). \tag{3.6}$$

Problem 3.2 (The (k, h)-segmentation problem) *Given a sequence T of length n, integer values k and h with $h < k \leq n$ and error function E_p, find $S_{opt}(T, k, h)$.*

Lets denote by $\textsc{Opt}_p(k)$ the cost of the optimal solution of the `Segmentation` problem on a sequence T using error function E_p and by $\textsc{Opt}_p(k, h)$ the cost of the optimal solution of the (k, h)-segmentation problem on the same sequence for the same error function. The following two observations are immediate consequences of the definition of Problem 3.2.

Observation 3.1 *For sequence T the error E_p $(p = 1, 2)$ of the optimal solution to the* `Segmentation` *problem is no more than the error of the optimal solution to the (k, h)-segmentation problem if $h \leq k$. Thus*

$$\textsc{Opt}_p(k) \quad \leq \quad \textsc{Opt}_p(k, h).$$

Observation 3.2 *For sequence T the error E_p ($p = 1, 2$) of the optimal clustering of the points in T into h clusters is no more than the error of the optimal solution to the (k, h)-segmentation problem given that $h \leq k$. Thus,*

$$\textsc{Opt}_p(n, h) \quad \leq \quad \textsc{Opt}_p(k, h).$$

3.4.1 Algorithms for the (k, h)-segmentation problem

The (k, h)-segmentation problem is known to be NP-hard for $d \geq 2$ and $h < k$, since it contains clustering as its special case [GM03]. Approximation algorithms, with provable approximation guarantees are presented in [GM03] and their running time is $O(n^2(k+h))$. We now discuss two of the algorithms presented in [GM03]. We subsequently modify these algorithms, so that they use the DnS algorithm as their subroutine.

Algorithm 3 The Segments2Levels algorithm.

Input: Sequence T of n points, number of segments k, number of levels h.

Ouput: A (k, h) segmentation of T into k segments using h levels.

1: Find segmentation S, solution to the `Segmentation` problem on T. Associate each segment $\bar{s}$ of S with its corresponding level $\mu_{\bar{s}}$.

2: Find a set of h levels, L, by solving (n, h) segmentation problem on T.

3: Assign each segment representative $\mu_{\bar{s}}$ to its closest level in L.

4: Return segmentation S and the assignment of representatives to levels in L.

Algorithm Segments2Levels: The algorithm (described in pseudocode in Algorithm 3) initially solves the k-segmentation problem obtaining a segmentation S. Then it solves the (n, h)-segmentation problem obtaining a set L of h levels. Finally, the representative μ_s of each segment $s \in S$ is assigned to the level in L that is the closest to μ_s.

Algorithm ClusterSegments: The pseudocode is given in Algorithm 4. As before, the algorithm initially solves the k-segmentation problem obtaining a segmentation S. Each segment $s \in S$ is represented by its representative μ_s weighted by the length of the seg-

ment $|s|$. Finally, a set L of h levels is produced by clustering the k weighted points into h clusters.

Algorithm 4 The ClusterSegments algorithm.

Input: Sequence T of n points, number of segments k, number of levels h.

Ouput: A (k,h) segmentation of T into k segments using h levels.

1: Find segmentation S, solution to the k-segmentation problem on T. Associate each segment $\bar{s}$ of S with its corresponding level $\mu_{\bar{s}}$.

2: Cluster the k weighted representatives in h clusters. Use the h cluster representatives (forming the set L) as the labels for the segment representatives.

3: Return segmentation S and the assignment of representatives to levels in L.

3.4.2 Applying DnS to the (k,h)-segmentation problem

Step 1 of both Segments2Levels and ClusterSegments algorithms uses the optimal dynamic programming algorithm for solving the k-segmentation problem. Using DnS instead we can achieve the a set of approximation results that are stated and proved in Theorems 3.8, 3.9.

Theorem 3.8 *If algorithm* Segments2Levels *uses* DnS *for obtaining the k-segmentation, and the clustering step is done using an α-approximation algorithm, then the overall approximation factor of* Segments2Levels *is* $(6+\alpha)$ *for both* E_1 *and* E_2*-error measures.*

Proof. We prove the statement for E_2. The proof for E_1 is similar.

Denote by $\mathrm{S2L}_2$ the E_2-error for the (k,h)-segmentation output by the Segments2Levels algorithm that uses DnS for producing the k-segmentation. Also let $\mathrm{Opt}_2(k,h)$ the E_2 cost of the optimal (k,h) segmentation.

For every point $t \in T$, let μ_t, c_t and ℓ_t denote the representative assigned to it after steps 1, 2 and 3 of the algorithm respectively. For each point $t \in T$ using the triangle inequality we have that

$$d_2(t,\ell_t)^2 \le (d_2(t,\mu_t) + d_2(\mu_t,\ell_t))^2$$

From the triangular inequality and due to the optimality of the assignment of levels to segments in Step 3 of the algorithm, we have

$$\begin{aligned}
\mathrm{S2L}_2^2 = \sum_{t\in T} d_2(t,\ell_t)^2 &\le \sum_{t\in T}\left(d_2(t,\mu_t)+d_2(\mu_t,\ell_t)\right)^2 \\
&\le \sum_{t\in T}\left(d_2(t,\mu_t)+d_2(\mu_t,c_t)\right)^2 .
\end{aligned}$$

Applying triangle inequality again we get

$$\begin{aligned}
\sum_{t\in T}\left(d_2(t,\mu_t)+d_2(\mu_t,c_t)\right)^2 &\le \sum_{t\in T}\left(d_2(t,\mu_t)+d_2(\mu_t,t)+d_2(t,c_t)\right)^2 \\
&= \sum_{t\in T}\left(2\,d_2(t,\mu_t)+d_2(t,c_t)\right)^2 \\
&= 4\sum_{t\in T} d_2(t,\mu_t)^2 + \sum_{t\in T} d_2(t,c_t)^2 \\
&\quad +4\sum_{t\in T} d_2(t,\mu_t) d_2(t,c_t) \\
&\le 4\times 9\left[\textsc{Opt}_2(k)\right]^2 + \alpha^2\left[\textsc{Opt}_2(n,h)\right]^2 \\
&\quad +4\sqrt{\sum_{t\in T} d_2(t,\mu_t)^2}\sqrt{\sum_{t\in T} d_2(t,c_t)^2} \\
&\le 36\left[\textsc{Opt}_2(k)\right]^2 + \alpha^2\left[\textsc{Opt}_2(n,h)\right]^2 \\
&\quad +12\alpha\textsc{Opt}_2(k)\ \textsc{Opt}_2(n,h) \\
&= \left(6\textsc{Opt}(k)+\alpha\textsc{Opt}(n,h)\right)^2 \\
&\le \left[(6+\alpha)\textsc{Opt}_2\right]^2 .
\end{aligned}$$

□

When the data points are of dimension 1 ($d = 1$) then clustering can be solved optimally using dynamic programming and thus the approximation factor is 7 for both E_1 and E_2 error measures. For $d > 1$ and for both E_1 and E_2 error measures the best α is $(1+\epsilon)$ using the algorithms proposed in [ARR98] and [KSS04] respectively.

Theorem 3.9 *Algorithm* ClusterSegments *that uses* DnS *for obtaining the k-segmentation, has approximation factor* 11 *for E_1-error and* $\sqrt{29}$ *for E_2-error measure.*

The proof of Theorem 3.9 is almost identical to the approximation proof of ClusterSegments algorithm presented in [GM03] and thus is omitted.

Notice that the clustering step of the ClusterSegments algorithm does not depend on n and thus one can assume that clustering can be solved optimally in constant time, since usually $k << n$. However, if this step is solved approximately using the clustering algorithms of [ARR98] and [KSS04], the approximation ratios of the ClusterSegments algorithm that uses DnS for segmenting, becomes $11 + \epsilon$ for E_1 and $\sqrt{29 + \epsilon}$ for E_2.

Given Theorem 3.1 and using the linear time clustering algorithm for E_2 proposed in [KSS04] and the linear time version of the algorithm proposed in [AGK$^+$01] for E_1 we get the following result:

Corollary 3.2 *Algorithms* Segments2Levels *and* ClusterSegments *when using* DnS *in their first step run in time* $O(n^{4/3}k^{5/3})$ *for both* E_1 *and* E_2 *error measure.*

3.5 Conclusions

In this chapter we described a family of approximation algorithms for the `Segmentation` problem. The most basic of those algorithms (DnS) works in time $O(n^{4/3}k^{5/3})$ and is a 3-approximation algorithm. We have described and analyzed several variants of this basic algorithm that are faster, but have worse approximation bounds. Furthermore, we quantified the accuracy versus speed tradeoff. Our experimental results on both synthetic and real datasets show that the proposed algorithms outperform other heuristics proposed in the literature and that the approximation achieved in practice is far below the bounds we obtained analytically. Finally, we have applied the DnS algorithm to other segmentation problems and obtained fast algorithms with constant approximation bounds.

CHAPTER 4

Clustered segmentations

In this chapter we study an alternative formulation of the `Segmentation` problem tailored for multidimensional sequences. More specifically, when segmenting a multidimensional signal, we allow different dimensions to form clusters such that: (a) the dimensions within the same cluster share the same segment boundaries and (b) the different clusters are segmented independently. Such a segmentation is different from the traditional segmentation approaches where all dimensions of a multidimensional signal are forced to share the same segment boundaries. We believe that in many settings, it is reasonable to assume that some dimensions are more correlated than others, and that concrete and meaningful states are associated with only small subsets of the dimensions.

An illustrating example of the above assumption is shown in Figure 4.1. The input sequence is the four-dimensional time series, shown in the top box. In the middle box, we show the globally optimal segmentation for the input time series when all dimensions share common segment boundaries. In this example, one can see that the global segmentation provides a good description of the sequence—in all dimensions most of the segments can be described fairly well using a constant value. In this segmentation some of the segments are not quite uniform. Furthermore, there are some segment boundaries introduced in relatively constant pieces of the sequence. The above representation problems can be alleviated if one allows different segment boundaries among subsets of dimensions, as shown in the lower box of Figure 4.1. We see that the segmentation after clustering the dimensions in pairs $\{1,2\}$ and $\{3,4\}$

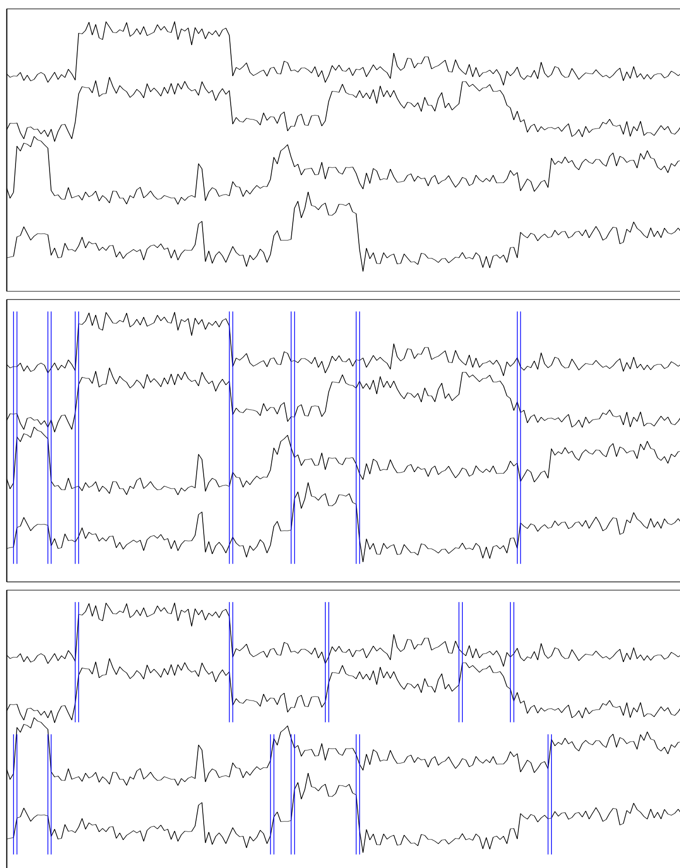


Figure 4.1: Example illustrating the usefulness of clustered segmentations.

gives a tighter fit and a more intuitive description of the sequence, even though the number of segments used for each cluster is smaller than the number of segments used in the global segmentation.

In our setting, we are looking for a segmentation of a multidimensional sequence when subsets of dimensions are allowed to be clustered and segmented separately from other subsets. We call this segmentation model *clustered segmentation*, and the problem of finding the best clustered segmentation of a multidimensional sequence the `Clustered Segmentation` problem.

Clustered segmentation can be used to extend and improve the quality of results in many application domains. For example, con-

sider the problem of "context awareness" as it arises in the area of mobile communications. The notion of context awareness can prove a powerful cue for improving the friendliness of mobile devices. In such a setting, certain context states might be independent of some sensor readings, therefore, a segmentation based on all sensors simultaneously would be a bad predictor.

The *haplotype block* structure discovery, is another problem from the biology domain, where the clustered segmentation model can be useful. One of the most important discoveries that came out of the analysis of genomic sequences is the *haplotype block structure*. To explain this notion, consider a collection of DNA sequences over n marker sites for a population of p individuals. The "haplotype block structure hypothesis" states that the sequence of markers can be segmented in blocks, such that in each block most of the haplotypes of the population fall into a small number of classes. The description of these haplotype blocks can be further used, for example, in the association of specific blocks with genetic-influenced diseases [Gus02]. From the computational point of view, the problem of discovering haplotype blocks can be viewed as a partitioning of a long multidimensional sequence into segments, such that, each segment demonstrates low diversity among the different dimensions (the individuals). Naturally, segmentation algorithms have been applied to this problem [DRS+01, KPV+03, PBH+01]. Since in this setting the different dimensions correspond to different individuals, applying the clustered segmentation model would allow for a clustering of the population into groups. Each such group is expected to have a distinct haplotype block structure. The existence of such subpopulations gives rise to a *mosaic structure* of haplotype blocks, which is a viable biological hypothesis [SHB+03, WP03].

In this chapter, we focus on formally defining the `Clustered Segmentation` problem and devising algorithms for solving it in practice. The results presented here have already appeared in [GMT04]. Note, that throughout this chapter we keep the problem definition rather generic. That is, we discuss the `Clustered Segmentation` problem and the corresponding algorithms for arbitrary (but easily computable) error functions E. Only when a concrete error function is necessary, we use the E_p error function for $p = 1, 2$.

4.1 Problem definition

Let $T = \{T_1, \ldots, T_d\}$ be a d-dimensional sequence, where T_i is the i-th dimension (signal, attribute, individual, etc.). We assume that each dimension is a sequence of n values. The positions on each dimension are naturally ordered, for example, in timeseries data the order is induced by the time attribute, while in genomic data the order comes from the position of each base in the DNA sequence. In addition, we assume that the dimensions of the sequence are aligned, that is, the values at the u-th position of all dimensions are semantically associated (e.g., they correspond to the same time).

The *clustered* version of segmentation problem is defined as follows.

Problem 4.1 (The `Clustered Segmentation` problem) *Given a d-dimensional sequence T with n values in each dimension, error function E defined on all subsequences and all dimensions of T, and integers k and c, find c k-segmentations $S_1, \ldots, S_c$, and an assignment of the d dimensions to these segmentations such that*

$$\sum_{i=1}^{d} \min_{1 \leq j \leq c} E(T_i, S_j)$$

is minimized.

In other words, we seek to partition the sequence dimensions into c clusters, and to compute the optimal segmentation in each one of these clusters in a way that the total error is minimized. Alternatively, we can use the Bayesian approach to define the variable-c version of the problem, where the optimal value for c is sought, but we assume that the value of c is given.

4.1.1 Connections to related work

The clustered segmentation problem, as stated above, is a form of timeseries clustering. We want to put in the same cluster time series that segment well together. The only difference is that in our case we assume that the different time series correspond to the different dimensions of the same sequence. There is abundance of work in

timeseries clustering like for example in [VLKG03, KGP01]. Several definitions for timeseries similarity have also been discussed in the literature, for example, see [BDGM97, FRM94, ALSS95]. A key difference, however, is that in our formulation different dimensions are assigned to the same cluster if they can be segmented well together, while most timeseries clustering algorithms base their grouping criteria in a more geometric notion of similarity.

The formulation of Problem 4.1 suggests that one can consider clustered segmentation as a k-median type of problem (e.g., see [LV92]), where $k = c$. However, the main difficulty with trying to apply k-median algorithms in our setting, is that the search space is extremely large. Furthermore, for k-median algorithms it is often the case that a "discretization" of the solution space can be applied (seek for solutions only among the input points). Assuming the triangle inequality, this discretization degrades the quality of the solution by a factor of at most 2. In our setting, however, the solution space (segmentations) is different from the input space (sequence), and also many natural distance functions between sequences and segmentations do not form a metric.

Finally, our problem is also related with the notion of *segmentation problems* as introduced by Kleinberg et al. [KPR98]. In [KPR98], starting from an optimization problem, the "segmented" version of that problem is defined by allowing the input to be partitioned in clusters, and considering the best solution for each cluster separately. To be precise, in the terminology of [KPR98] our problem should be called "`Segmented Segmentation`", since in our case the optimization problem is the standard segmentation problem. Even when starting from very simple optimization problems, their corresponding segmented versions turn out to be hard.

4.1.2 Problem complexity

In this section we demonstrate the computational hardness of the clustered segmentation problem. In our case, the optimization problem we start with is the $\mathcal{SEG}_{\mathrm{SUM}}$ problem. As usual we focus on the cases where the error function is E_p with $p = 1, 2$. We already know that in these cases the $\mathcal{SEG}_{\mathrm{SUM}}$ problem is solvable in $O(n^2k)$ time. Not surprisingly the corresponding `Clustered Segmentation` problem is NP-hard problem.

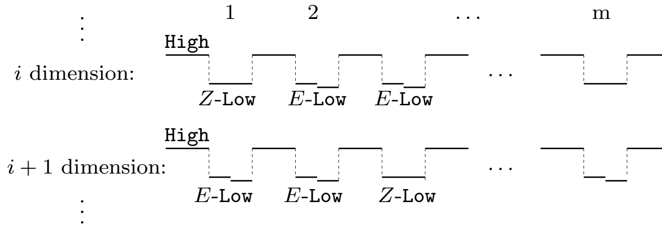


Figure 4.2: The construction used for transforming an instance of the `Set Cover` problem to an instance of the `Clustered Segmentation` problem in the proof of Theorem 4.1.

Theorem 4.1 *The* `Clustered Segmentation` *problem (Problem 4.1), with real-valued sequences, and cost function E_p, is NP-hard.*

Proof.

We give here the proof for error function E_1. The proof for error function E_2 is identical. The problem from which we obtain the reduction is the `Set Cover`, a well-known NP-hard problem [GJ79]. An instance of the `Set Cover` specifies a ground set U of n elements, a collection $\mathcal{C}$ of m subsets of U, and a number c. The question is whether there are c sets in the collection $\mathcal{C}$ whose union is the ground set U.

Given an instance of the `Set Cover`, we create an instance of the `Clustered Segmentation` as follows: we form a sequence with n dimensions. In each dimension there are $2m + 1$ "runs", all of equal number of points. There are two types of runs; *High* and *Low*, and these two types are alternating across the sequence. All *High* runs have all their values equal to 1. The *Low* runs are indexed from 1 to m and they in turn can be of two types: Z and E. Z-*Low* runs have all their values equal to 0. E-*Low* runs are split in two pieces, so that the E function on such a run incurs cost exactly ϵ. The construction can be seen schematically in Figure 4.2.

The instance of the `Set Cover` is encoded in the *Low* runs. If the j-th set of $\mathcal{C}$ contains the i-th element of U, then the j-th *Low* run of the i-th dimension is set to type E, otherwise it is set to type Z. Assume that in total there are L *Low* runs of type E. We would ask for clustered segmentation, in which each cluster has a k-segmentation with $k = 2m + 2$ segments. By setting ϵ to be very small, the $2m + 1$ segments would be separating the *High* from the

Low runs, and there would be the freedom to segment one more *E-Low*-type run in order to save an additional cost of ϵ per dimension. The question to ask is whether there is a clustered segmentation with c clusters that has cost at most $(L-n)\epsilon$. This is possible if and only if there is a solution to the original `Set Cover` problem.

For the one direction we need to show that if there exists a set cover of size c, then there exists a clustered segmentation with c clusters, $2m+2$ segments, and error less than $(L-n)\epsilon$. Given the set cover we can construct the clusters of the dimensions in such a way that dimensions i and j are put in the same cluster if there is a set in the set cover that contains elements i and j (ties are broken arbitrarily). It is apparent that this grouping of dimensions into c groups returns a clustered segmentation that has error at most $(L-n)\epsilon$.

For the reverse direction we have to show that if there is a clustered segmentation with c clusters, $2m+2$ segments per cluster, and error less than $(L-n)\epsilon$, then there is a set cover of size c. Since the error of the clustered segmentation is at most $(L-n)\epsilon$, at least n *E-Low* runs have been broken into two segments. Since we ask for $(2m+2)$-segmentation per cluster only one such break can be done in each dimension, segmenting one *E-Low* run into two. For this to happen, the clustering of dimensions has been done in such a way, that all the dimensions in the same cluster have the same *E-Low* segment been segmented. Given the solution of the `Clustered Segmentation` problem with c clusters we can obtain a solution of size c to the `Set Cover` problem as follows. Let $\mathcal{T}_i$ a set of dimensions grouped together in one of the c clusters of the clustered segmentation. For each such group pick a set $\mathcal{C}_{i'}$ from the original `Set Cover` problem such that $\mathcal{T}_i \subseteq \mathcal{C}_{i'}$. By construction, such a set always exists for every cluster of the solution.□

4.2 Algorithms

In this section we describe four algorithms for solving the `Clustered Segmentation` problem. The first two are based on the definition of two different distance measures between sequence segmentations. Then they use a standard clustering algorithm (e.g., K-means) on the pairwise distance matrix. The K-means is a hill

climbing algorithm that is not guaranteed to converge to a global optimum. However, it is widely used because it is efficient and it works very well in practice. Variations of the K-means algorithm have been proposed for timeseries clustering, as for example in [VLKG03].

The main idea of the K-means algorithm is the following: Given N points to be clustered and a distance function *dist* between them, the algorithm starts by selecting K random points as cluster centers and assigning the rest of the $N-K$ points to the closest cluster center, according to *dist*. In that way K clusters are formed. Within each cluster the cluster representative is selected and the process continues iteratively with these means as the new cluster centers, until convergence.

The two distance functions we define here are rather intuitive and simple. The first one, D_E, is based on the mutual exchange of optimal segmentations of the two sequences and the evaluation of the additional error such an exchange introduces. Therefore, two sequences are similar if the optimal segmentation of the one describes well the second, and vice versa. The second distance function, D_P evaluates the distance between two sequences by comparing the probabilities of each position in the sequence being a segment boundary.

The other two algorithms are randomized methods that cluster sequences using segmentations as "centroids". In particular, we use the notion of a distance between a segmentation and a sequence, which is the error induced to the sequence when the segmentation is applied to it. These algorithms treat the `Clustered Segmentation` problem as a model selection problem and they try to find the best such clustered segmentation model that describes the data. The first algorithm, SamplSegm, is a sampling algorithm and it is motivated by the theoretical work presented in [KPR98, Ind99, COP03]. The second, IterClustSegm, is an adaptation of the popular K-means algorithm. Both algorithms are simple and intuitive and they perform well in practice.

The optimal dynamic programming algorithm that uses Recursion (2.3) is used as a subroutine by all our methods. That is, whenever we decide upon a good grouping of the dimensions, we use this optimal dynamic programming algorithm to determine the optimal segmentation of this group.

4.2.1 The D_E distance function between segmentations

The goal of our clustering is to cluster together dimensions in such a way that similarly segmented dimensions are put in the same cluster, while the overall cost of the clustered segmentation is minimized. Intuitively this means that a distance function is appropriate if it quantifies how well the optimal segmentation of the one sequence describes the other one and vice versa. Based on exactly this notion of "exchange" of optimal segmentations of sequences, we define the distance function D_E in the following way.

Given two dimensions T_i, T_j and their corresponding optimal k-segmentations $S_i^*, S_j^* \in \mathcal{S}_{n,k}$, we define the distance of T_i from S_j^* denoted by $D_E(T_i, S_i^* | S_j^*)$ as

$$D_E(T_i, S_i^* | S_j^*) \quad = \quad E(T_i, S_j^*) - E(T_i, S_i^*).$$

However, in order for the distance to be symmetric we alternatively use the following definition of D_E.

$$D_E(T_i, S_i^*, T_j, S_j^*) \quad = \quad D_E(T_i, S_i^* | S_j^*) + D_E(T_j, S_j^* | S_i^*).$$

4.2.2 The D_P distance function between segmentations

Distance function D_P is based on comparing two dimensions (or in general two time series of the same length) via comparing the probability distributions of their points being segment boundaries. A similar approach to compute these probabilities is presented in [KPV+03].

The basic idea for D_P comes from the fact that we can associate with each dimension T_i (with $1 \leq i \leq d$), a probability distribution P_i. For a given point $t \in \{1, \ldots, n\}$ the value of $P_i(t)$ is defined to be the probability of the t-th point being a segment boundary in a k-segmentation of sequence T_i. The details of how this probability distribution is evaluated will be given shortly. Once every dimension T_i is associated with the probability distribution P_i, we can define the distance between two dimensions T_i and T_j as the *variational distance* between the corresponding probability distributions P_i and P_j[1]. Therefore, we define the distance function $D_P(T_i, T_j)$ between the dimensions T_i and T_j as

[1]Other measures for comparing distributions, for example the KL-divergence, could also be used for that.

$$D_P(T_i,T_j) = VarD(P_i,P_j) = \sum_{1\leq t\leq n} |P_i(t) - P_j(t)|. \quad (4.1)$$

We devote the rest of this paragraph to describe how we can evaluate the probability distributions P_i for every dimension T_i. We also discuss the intuition behind these calculations. In order to proceed we need to extend our notation, so that it handles segmentations of subsequences and "subsegmentations". Given a sequence T, we have already used $T[i,j]$, with $i < j$, to denote the subsequence of T that contains all points between positions i and j. Similarly for a segmentation S with boundaries $\{s_0,\ldots,s_k\}$ we use $S[i,j]$ to denote the set of boundaries $\{i,j\} \cup \{b \in S \mid i \leq b \leq j\}$. That is, $S[i,j]$ contains all the boundaries of S in positions between points i and j, as well as the boundaries i and j themselves. Finally, we denote by $\mathcal{S}[i,j]$ the family of all segmentations that can be defined on the interval between positions i and j.

Denote by $P_T(t)$ the probability that point t is a segment boundary in a k-segmentation of sequence T and by $\mathcal{S}_k^{(t)}$ the set of all k-segmentations of T that have a boundary at point t. Then, for a given sequence T, we are interested in computing

$$P_T(t) = \Pr(\mathcal{S}_k^{(t)}|T), \quad (4.2)$$

for every point $t \in T$.

Equation (4.2) uses conditional probabilities of segmentations given a specific sequence. By the definition of probability, Equation (4.2) can be rewritten as

$$\begin{aligned}\Pr(\mathcal{S}_k^{(t)}|T) &= \frac{\Pr(\mathcal{S}_k^{(t)},T)}{\Pr(\mathcal{S}_k,T)} \\ &= \frac{\sum_{S'\in\mathcal{S}_k^{(t)}} \Pr(S',T)}{\sum_{S\in\mathcal{S}_k} \Pr(S,T)}. \quad (4.3)\end{aligned}$$

For the joint probabilities of segmentation and sequence, $\Pr(S,T)$, it holds that

$$\Pr(S,T) = \frac{1}{Z} e^{-\sum_{\bar{s}\in S} E(T[\bar{s}],\bar{s})}, \quad (4.4)$$

where $\bar{s}$ is a segment of segmentation S and $T[\bar{s}]$ is the subsequence of T defined by the boundaries of segment $\bar{s}$. Also Z is a normalizing

constant that cancels out when substituting the joint probabilities into Equation (4.3). Equation (4.4) assigns a probability for every segmentation S when applied to sequence T. For every segmentation S it considers every segment $\bar{s} \in S$. The existence of each such segment introduces some error in the representation of the sequence T. The points of the sequence that are affected by the existence of $\bar{s}$ are the points in $T[\bar{s}]$. For error function E, this error valuates to $E(T[\bar{s}], \bar{s})$. Giving to this error a likelihood interpretation, we can say that when segment $\bar{s}$ causes large error in $T[\bar{s}]$, then a model (aka segmentation) that contains $\bar{s}$ is not a good model for sequence T, and thus it should be picked with small probability. Equation (4.4) suggests that the probability of a segment $\bar{s}$ existing in the picked segmentation is proportional to $e^{-E(T[\bar{s}],\bar{s})}$. That is, segmentations that consist of segments that cause large errors in the representation of the sequence T should also have small probability of being picked. The probability of a segmentation S is the product of the probabilities of the segments appearing to them and thus proportional to $e^{-\sum_{\bar{s}\in S} E(T[\bar{s}],\bar{s})}$. Substituting Equation (4.4) in (4.3) allows us to compute the probability of point t being a segment boundary.

Now we discuss how to evaluate Equation (4.3) algorithmically using dynamic programming. First, for a given segmentation S and segment $\bar{s}_i$ with boundaries s_{i-1} and s_i we define

$$q(s_{i-1}, s_i) = e^{-E(T[\bar{s}],\bar{s})}.$$

For any interval $[t, t'] \in T$ and for segmentations that have exactly i-segments (where $1 \leq i \leq k$) we also define

$$Q_i(t, t') = \sum_{S \in \mathcal{S}_i[t,t']} \prod_{\bar{s}_i \in S} q(s_{i-1}, s_i).$$

Since $\mathcal{S}_k^{(t)}$ contains all segmentations in the Cartesian product $\mathcal{S}_i[1, t] \times \mathcal{S}_{k-i}[t+1, n]$, for $1 \leq i \leq k$, we have that

$$\Pr(\mathcal{S}_k^{(t)} | T) = \sum_{1 \leq i < k} \frac{Q_i(1, t) Q_{k-i}(t+1, n)}{Q_k(1, n)}.$$

Overall, the dynamic programming recursions that allow us to compute the probabilities of points being segment boundaries when

considering segmentations with fixed number of segments i (with $1 \leq i \leq k$) are

$$Q_i(1,b) = \sum_{1 \leq a \leq b} Q_{i-1}(1,a-1)q(a,b) \quad (4.5)$$

and

$$Q_i(a,n) = \sum_{a \leq b \leq n} q(a,b)Q_{i-1}(b+1,n). \quad (4.6)$$

The above recursions give the following corollary.

Corollary 4.1 *For a given sequence T of length n, computing the probability of each point $t \in \{1,\ldots,n\}$ being a segment boundary can be done in time $O(n^2k)$ using the dynamic programming recursions defined in Equations (4.5) and (4.6).*

Once the probability distributions P_i are computed for every dimension T_i, the pairwise probabilistic distances between pairs of dimensions are evaluated using Equation (4.1).

4.2.3 The SamplSegm algorithm

The basic idea behind the SamplSegm approach is that if the data exhibit clustered structure, then a small sample of the data would exhibit the same structure. The reason is that for large clusters in the dataset one would expect that it is adequate to sample enough data, so that these clusters appear in the sample. At the same time, one can possibly afford to miss data from small clusters in the sampling process, because small clusters do not contribute much in the overall error. Our algorithm is motivated by the work of [KPR98], where the authors propose a sampling algorithm in order to solve the segmented version of the catalog problem. Similar ideas have been used successfully in [Ind99] for the problem of clustering in metric spaces.

For the `Clustered Segmentation` problem we adopt a natural sampling-based technique. We first sample uniformly at random a small set A of $r \log d$ dimensions, where r is a small constant. Then, we search exhaustively all possible partitions of A into c clusters

$C_1, \ldots, C_c$. For each cluster C_j we find the optimal segmentation $S_j \in \mathcal{S}_k$ for the set of dimensions in C_j. Each one of the remaining dimensions $T_i \notin A$, are assigned to cluster C_j that minimizes the error $E(T_i, S_j)$. The partitioning of the sample set A that causes the least total error is considered to be the solution found for the set A. The sampling process is repeated with different sample sets A for a certain number of times and the best result is reported as the output of the sampling algorithm.

When the size of the sample set is logarithmic in the number of dimensions, the overall running time of the algorithm is polynomial. In our experiments, we found that the method is accurate for data sets of moderate size, but it does not scale well for larger data sets.

4.2.4 The IterClustSegm algorithm

The IterClustSegm algorithm is an adaptation of the widely used K-means algorithm. The only difference here is that the cluster centers are replaced by the common segmentation of the dimensions in the cluster and the distance of a sequence from the cluster center is the error induced when the cluster's segmentation is applied to the sequence.

Therefore, in our case, the c centers correspond to c different segmentations. The algorithm is iterative and at the t-th iteration step it keeps an estimate for the solution segmentations $S_1^t, \ldots, S_c^t$, which is to be refined in the consecutive steps. The algorithm starts with a random clustering of the dimensions, and it computes the optimal k-segmentation for each cluster. At the $(t+1)$-th iteration step, each dimension T_i is assigned to the segmentation S_j^t for which the error $E(T_i, S_j^t)$ is minimized. Based on the newly obtained clusters of dimensions, new segmentations $S_1^{t+1}, \ldots, S_c^{t+1}$ are computed, and the process continues until there is no more improvement in the result. The complexity of the algorithm is $O(I(cd+cP(n,d)))$, where I is the number of iterations until convergence, and $P(n,d)$ is the complexity of segmenting a sequence of length n and d dimensions. For error function E_p with $p = 1, 2$, $P(n,d) = O(n^2kd)$.

4.3 Experiments

In this section we describe the experiments we performed in order to evaluate the validity of the clustered segmentation model and the behavior of the suggested algorithms. For our experiments we used both synthetically generated data, as well as real timeseries data. For all cases of synthetic data, the algorithms find the true underlying model that was used to generate the data. For the real data we found that in all cases clustered segmentations, output by the proposed algorithms, introduce less error in the representation of the data than the non-clustered segmentations.

4.3.1 Ensuring fairness in model comparisons

In the experimental results shown in this section we report the accuracy in terms of error. Our intention is to consider the error as a measure of comparing models; a smaller error indicates a better model. Note though that this can only be the case when the compared models have the same number of parameters. It would be unfair to compare the error induced by two models with different number of parameters, because the trivial model of each point described by itself would induce the the least error and would be the best.

For this reason when comparing the error of two representations we make sure that the underlying models have the same number of parameters. We guarantee this fairness in the comparison as follows. Consider a k-segmentation for a d-dimensional sequence $T \in \mathcal{T}_n$. If no clustering of dimensions is considered, the number of parameters that are necessary to describe this k-segmentation model is $k(d+1)$. This number comes from the fact that we have k segments and for each segment we need to specify its starting point and its mean, which is a d-dimensional point. Consider now a clustered segmentation of the sequence with c clusters and k' segments per cluster. The number of parameters for this model is $d + \sum_{i=1}^{c} k'(d_i+1) = d + k'(d+c)$, since, in addition to specifying the starting points and the representative points for each cluster, we also need d parameters to indicate the cluster that each dimension belongs to. In our experiments, in order to compare the errors induced by the two models we select parameters so that $k(d+1) =$

$d + k'(d + c)$. This ensures fairness in our comparisons.

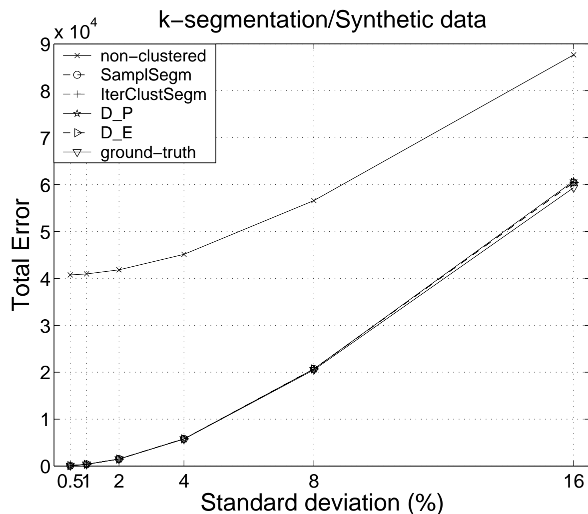
4.3.2 Experiments on synthetic data

We first describe our experiments on synthetic data. For the purpose of this experiment, we have generated sequence data from a known model, and the task is to test if the suggested algorithms are able to discover that model.

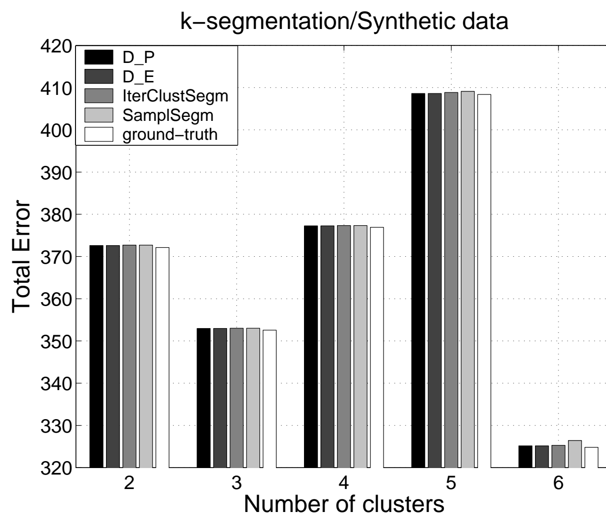
The synthetic datasets were generated as follows: the d dimensions of the generated sequence were divided in advance into c clusters. For each cluster we select k segment boundaries, which are common for all the dimensions in that cluster, and for the j-th segment of the i-th dimension we select a mean value μ_{ij}, which is uniformly distributed in $[0, 1]$. Points are then generated by adding a noise value sampled from the normal distribution $\mathcal{N}(\mu_{ij}, \sigma^2)$. An example of a small data set generated by this method is shown in Figure 4.1. For our experiments we fixed the values $n = 1000$ points, $k = 10$ segments, and $d = 200$ dimensions. We created different data sets using $c = 2, ..., 6$ clusters and with standard deviations varying from 0.005 to 0.16.

The results for the synthetically generated data are shown in Figure 4.3. One can see that the errors of the reported clustered segmentation models are typically very low for all of our algorithms. In most of the cases all proposed methods approach the *true* error value. Since our algorithms are randomized we repeat each one of them for 5 times and report the best found solution. Apart from the errors induced by the proposed algorithms, the figures include also two additional errors. The error induced by the non-clustered segmentation model with the same number of parameters and the error induced by the true model that has been used for generating the data ("ground-truth"). The first one is always much larger than the error induced by the models reported by our algorithms. In all the comparisons between the different segmentation models we take into consideration the fairness criterion discussed in the previous section.

As indicated in Figure 4.3(a) the difference in errors becomes smaller as the standard deviation of the dataset increases. This is natural since as standard deviation increases all dimensions tend to become uniform and the segmental structure disappears. The error



(a) Synthetic datasets: error of the segmentation as a function of the standard deviation used for data generation.



(b) Synthetic datasets: error of the segmentation as a function of the number of clusters used in the clustered segmentation model. The bar that corresponds to the non-clustered model is missing, since it gives error that his an order of magnitude larger.

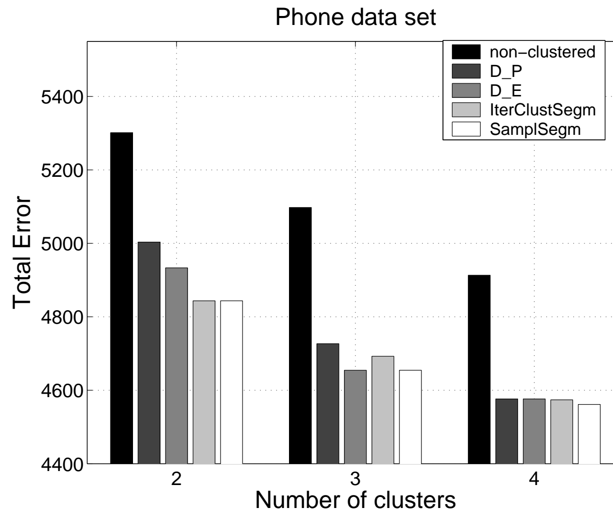
Figure 4.3: Error of segmentations on synthetic data sets.

of the clustered segmentation model as a function of the true underlying number of clusters is shown in Figure 4.3(b). The better performance of the clustered model is apparent. Notice that the error caused by the non-clustered segmentation is an order of magnitude larger than the corresponding clustered segmentation results and thus omitted from the plot.

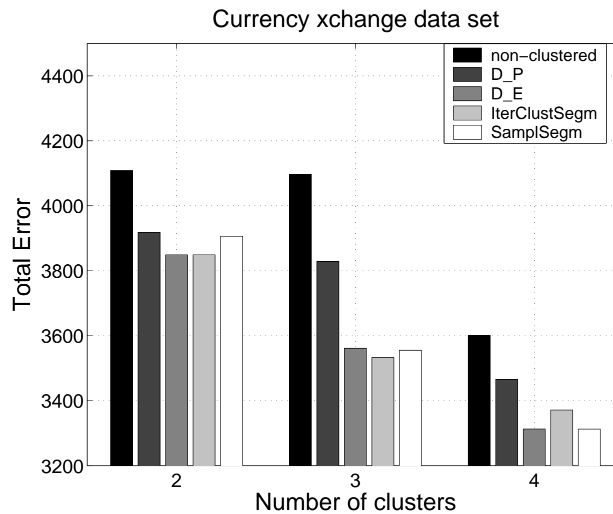
4.3.3 Experiments on timeseries data

Next, we tested the behavior of the clustered segmentation model on real timeseries data sets obtained by the UCR timeseries data mining archive [KF02]. We used the **phone** and the **spot_exrates** data sets of the archive. The **phone** data set consists of 8 dimensions each one corresponding to the value of a sensor attached to a mobile phone. For the clustered segmentation we used number of segments $k = 8$ and number of clusters $c = 2$, 3 and 4. For the non-clustered segmentation we used $k = 10$, 11, and 11, respectively so that we again guarantee a fair comparison. Figure 4.4(a) shows the error induced by the clustered and the non-clustered segmentations for different number of clusters. Apparently, the clustered segmentation model reduces by far the induced error. SamplSegm, IterClustSegm and clustering using D_E are all giving the same level of error, with clustering using D_P performing almost equally well, and in all cases better than the non-clustered segmentation.

Analogous results are obtained for the **spot_exrates** data set as illustrated in Figure 4.4(b). This data set contains the spot prices (foreign currency in dollars) and the returns for daily exchange rates of the 12 different currencies relative to the US dollar. There are 2566 daily returns for each of these 12 currencies, over a period of about 10 years (10/9/86 to 8/9/96). For our experiments we used number of segments $k = 10$ and number of clusters $c = 2$, 3, and 4 for the clustered segmentations. For the non-clustered segmentation we used $k = 12$, 12 and 13, respectively.



(a)



(b)

Figure 4.4: Real timeseries data: error of the segmentations as a function of the number of clusters used in the clustered segmentation model.

4.3.4 Experiments on mobile sensor data

Finally, we tested the behavior of the proposed methods on the benchmark dataset for context recognition described in [MHK+04].[2] The data were recorded using microphones and a sensor box, attached to a mobile phone. The combination was carried by the users during the experiments and the data were logged. The signals collected were transformed into 29 variables (dimensions) the values of which have been recorded for some periods of time.

The dataset basically contains 5 scenarios each one repeated for a certain number of times. The 29 variables recorded correspond to 29 dimensions and are related to *device position*, *device stability*, *device placement*, *light*, *temperature*, *humidity*, *sound level* and *user movement*.

The results of the clustered segmentations algorithms for scenario 1 and 2 are shown in Figures 4.5(a) and 4.5(b). For the rest of the scenarios the results are similar and thus omitted. Since some dimensions in the different scenarios are all constant we have decided to ignore them. Therefore, from a total of 29 dimensions we have considered 20 for scenario 1, 19 for scenario 2, 16 for scenario 3, 14 for scenario 4 and 15 for scenario 5.

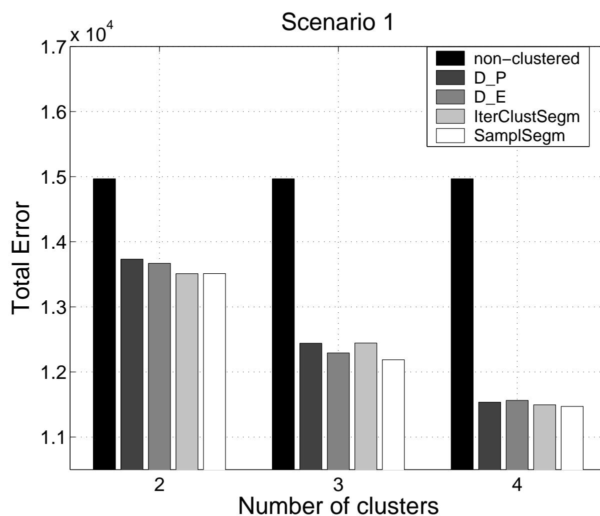
For the case of clustered segmentations we set $k = 5$ and $c = 2, 3, 4, 5$ for all the scenarios, while the corresponding values of k for the non-clustered segmentation that could guarantee fairness of comparison of the results was evaluated to be $k = 6, 7$ depending on the value of c and the number of dimensions in the scenario. The error levels using the different segmentation models are shown in Figures 4.5(a) and 4.5(b). In all cases, the clustered segmentation model found by any of our four methods has much lower error level than the corresponding non-clustered one. Some indicative clusterings of the dimensions of scenario 1 using the proposed methods are shown in Table 4.1. Notice that the clustering shown in Table 4.1 reflects an intuitive clustering of dimensions. The first cluster contains only dimensions related to the "Position" , the "Stability" and the "Placement" of the device. The second cluster puts together all time series related to the "Humidity" of the environment. The third cluster consists of dimensions related to the "Light" and

[2] The dataset is available at http://www.cis.hut.fi/jhimberg/contextdata/index.shtml

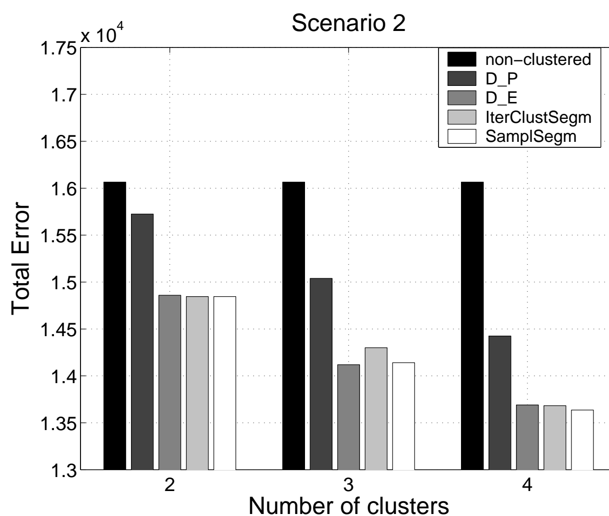
the “Sound Pressure” of the environment as well as the user movement. Finally, the last cluster contains all the dimensions related to the “Temperature” of the environment as well as the dimensions that corresponds to the “Running” dimensions that characterizes the user movement. This last result raises some suspicions, since running is the only user action related dimension that is clustered separately from the other two. However, this can be quite easily explained by observing Figure 4.6. It is obvious that segmentation-wise, dimensions “UserAction: Movement: Walking” and “‘UserAction: Movement: WalkingFast” are much closer to each other than they are with “UserAction: Movement: Running”. On the other hand, the latter dimension can be easily segmented using segmentation boundaries of dimensions “Environment:Temperature:Warm” and “Environment:Temperature:Cool”. Similar observations can be made also for the rest of the clusterings obtained using the other three proposed methods. Indicatively we show in Table 4.2 the clustering obtained using K-means algorithm for the same number of clusters and using L_1 as the distance metric between the different dimensions. There is no obvious correspondence between the clustering found using this method and the clustering of dimensions induced by their categorization.

4.3.5 Discussion

The experimental evaluation performed on both synthetic and real data indicates that the clustered segmentation model is a more precise alternative for describing the data at hand, and all the proposed methods find models that show much smaller error than the error of the equivalent non-clustered segmentation model, in all the considered cases. Overall, in the case of synthetic data the true underlying model, used for the data generation, is always found. For the real data, the true model is unknown and thus we base our conclusions on the errors induced by the two alternative models when the same number of parameters is used. The proposed algorithms are intuitive and they perform well in practice. For most of the cases they give equivalent results and they find almost the same models.



(a) Experiments with scenario 1.



(b) Experiments with scenario 2.

Figure 4.5: Context-recognition dataset: error of the segmentations as a function of the number of clusters used in the clustered segmentation model.

Cluster
Device:Position:DisplayDown, Device:Position:AntennaUp, Device:Stability:Stable, Device:Stability:Unstable, Device:Placement:AtHand
Environment:Humidity:Humid, Environment:Humidity:Normal, Environment:Humidity:Dry
Environment:Light:EU, Environment:Light:Bright, Environment:Light:Normal, Environment:Light:Dark, Environment:Light:Natural, Environment:SoundPressure:Silent, Environment:SoundPressure:Modest, UserAction:Movement:Walking, UserAction:Movement:WalkingFast
Environment:Temperature:Warm, Environment:Temperature:Cool, UserAction:Movement:Running

Table 4.1: The clustering of the dimensions of Scenario 1 into 4 clusters using IterClustSegm algorithm.

Cluster
Environment:Humidity:Dry
Environment:Light:Bright, Environment:Light:Natural, UserAction:Movement:WalkingFast
Device:Position:AntennaUp, Device:Stability:Unstable, Environment:Light:EU, Environment:Light:Normal, Environment:Temperature:Warm, Environment:Humidity:Humid, Environment:SoundPressure:Silent, UserAction:Movement:Walking
Device:Position:DisplayDown, Device:Stability:Stable, Device:Placement:AtHand, Environment:Light:Dark, Environment:Temperature:Cool, Environment:Humidity:Normal, Environment:SoundPressure:Modest, UserAction:Movement:Running’

Table 4.2: Clustering of the dimensions of Scenario 1 into 4 clusters using L_1 distance K-means.

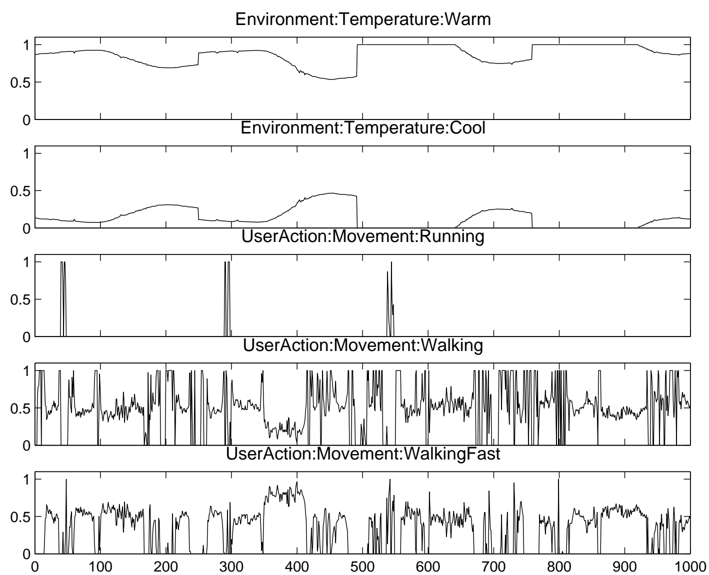


Figure 4.6: Subset of dimensions of Scenario 1 dataset; Visual inspection shows that UserAction:Movement:Running can be segmented well together with the "Environment" variables rather than the other "UserAction" variables.

4.4 Conclusions

In this chapter we have introduced the clustered segmentation problem where the task is to cluster the dimensions of a multidimensional sequence into c clusters so that the dimensions grouped in the same cluster share the same segmentation points. The problem when considered for real-valued sequences and cost function E_p is NP-hard. We described simple algorithms for solving the problem. All proposed methods perform well for both synthetic and real data sets consisting of timeseries data. In all the cases we experimented with, the clustered segmentation model seems to describe the datasets better than the corresponding non-clustered segmentation model with the same number of parameters.

CHAPTER 5

Segmentation with rearrangements

So far we have assumed as input a sequence T consisting of n points $\{t_1, \ldots, t_n\}$, with $t_i \in \mathbb{R}^d$. We have primarily focused on finding a segmentation S of T taking for granted the order of the points in T. That is, we have assumed that the correct order of the points coincides with the input order. In this chapter we assume that the order of the points in T is only approximately correct. That is, we 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`.

Motivating example: Consider the real-life example 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 that case segmenting, or in general analyzing, the data taking for granted the order in which the points arrive may result in misleading results. On the other hand, choosing to ignore data points that seem to be incompatible with their temporal neighbors leads to omitting possibly valuable information. In

such cases, it seems sensible to try to somehow "correct" the data before analyzing them. Omitting some data points is, of course, a type of correction. However, in this chapter we argue that there are cases where it makes more sense to attempt more gentle correction methodologies.

Related work: This chapter is closely related to the literature on segmentation algorithms. A thorough review of these algorithms can be found in Chapter 2. The `Segmentation with Rearrangements` problem is a classical `Segmentation` problem, where all the points appearing in the sequence are assumed to have correct values. However 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. Reordering techniques over the dimensions of multidimensional data have been proposed in [VPVY06]. 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). 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, [JKM99, MSV04] study the problem of finding outliers (or *deviants* as they are called) in timeseries data. More specifically, their 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 [JKM99, MSV04] 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 [CKMN01]. Finally, the task of detecting outliers itself has motivated lots of interesting research. The main idea there is again to find points or patterns that exhibit different behavior from the normal. Examples of such research efforts are presented in [CSD98, KNT00, PKGF03, ZKPF04].

5.1 Problem formulation

In this section we formally define the `Segmentation with Rearrangements` problem. Assume an input sequence $T = \{t_1, \ldots, t_n\}$. 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 $T = \{t_1, \ldots, t_n\}$ causes the following rearrangement of the elements of T: $\mathcal{B}(i, i+1) \circ T = \{t_1, \ldots, t_{i+1}, t_i, t_{i+2}, \ldots, t_n\}$.
- *Moves*: A move corresponds to a single-element transposition [HS05]. That is, a move $\mathcal{M}(i \rightarrow j)$ (with $i < j$) when applied to sequence $T = \{t_1, \ldots, t_n\}$ causes the following rearrangement of the elements in T: $\mathcal{M}(i \rightarrow j) \circ T = \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{j-1}, t_i, t_j, t_{j+1}, \ldots, t_n\}$.

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 $T_{\overline{\mathcal{B}}} \equiv \overline{\mathcal{B}} \circ T$ (or $T_{\overline{\mathcal{M}}} \equiv \overline{\mathcal{M}} \circ T$). 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 $\overline{O} \circ T \equiv T_{\overline{O}}$. Given the above notational conventions, we are now ready to define the generic `Segmentation with Rearrangements` problem.

Problem 5.1 **(`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`.

As in all the previous chapters we mainly focus on the E_p error measure, and particularly we are interested in the cases where $p = 1, 2$ (see Equation (2.2)).

5.2 Problem complexity

Although the basic k-segmentation problem (Problem 2.1) is solvable in polynomial time the alternative formulations we study in this chapter 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 $T = \{t_1, \ldots, t_n\}$ 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 [MZH83], and both are NP-hard for dimensions $d \geq 2$ [GJ79]. 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.

Lemma 5.1 *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 at 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 further focus our attention on the `Segmentation with Swaps` problem and we study its complexity for $d = 1$. We have the following lemma.

Lemma 5.2 *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 [GJ79], 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 $f(a) \in \mathbb{N}$. In this way, 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 5.1 *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 5.2 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. $\square$

5.3 Algorithms

In this section we give a description for our algorithmic approaches for the generic `Segmentation with Rearrangements` problem. Only

when necessary, we focus our discussion to the specific rearrangement operations that we are considering, namely the moves and the swaps.

5.3.1 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 section and expand in the sequel, is summarized in Algorithm 5. 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 (2.3)), 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.

Algorithm 5 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.

1: **Segment:** $S = S_{\text{opt}}(T, k)$
2: **Rearrange:** $\overline{O} = \text{rearrange}(T, S, C)$
3: **Segment:** $S' = S_{\text{opt}}(T_{\overline{O}}, k)$

In the rest of our discussion we focus in developing a methodology for the `Rearrangement` problem. Consider all segments of segmentation $S = \{\bar{s}_1, \ldots, \bar{s}_k\}$ as possible new locations for every point in T. Each point $t_j \in T$ is associated with a gain (or loss)

	$\bar{s}_1$	$\bar{s}_2$	...	$\bar{s}_k$
t_1	$\langle w_{11}, p_{11} \rangle$	$\langle w_{21}, p_{21} \rangle$	...	$\langle w_{k1}, p_{k1} \rangle$
t_2	$\langle w_{12}, p_{12} \rangle$	$\langle w_{22}, p_{22} \rangle$	...	$\langle w_{k2}, p_{k2} \rangle$
$\vdots$	...	...	...	...
t_n	$\langle w_{1n}, p_{1n} \rangle$	$\langle w_{2n}, p_{2n} \rangle$	...	$\langle w_{kn}, p_{kn} \rangle$

Table 5.1: The rearrangement table for fixed segmentation $S = \{\bar{s}_1, \ldots, \bar{s}_k\}$.

p_{ij} that is incurred to the segmentation error if we move t_j from its current position $pos(t_j, T)$ to segment $\bar{s}_i$. 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 $\bar{s}_i$. 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 $\bar{s}_i$. If we use a_i, b_i to denote the start and the end points of the segment $\bar{s}_i$, then the operational cost in the case of a move operation is

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

For the case of swaps the operational cost is

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

Note that since the segmentation is fixed, when repositioning the point t_j to the segment $\bar{s}_i$ (if $t_j \notin \bar{s}_i$) 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.

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

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 5.1 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 $\bar{s}_i$ 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.

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

The tabular representation of the `Knapsack` is shown in Table 5.2. 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 [GJ79]. However,

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

Table 5.2: The tabular formulation of the `Knapsack` problem.

it does admit a pseudopolynomial time algorithm that is based on dynamic programming [Vaz03].

The similarity between `Rearrangement` and `Knapsack` encourages us to apply algorithmic techniques similar to those applied for `Knapsack`. 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 $A[i, c]$

$$A[i,c] = \max \{A[i-1,c], \max_{1 \leq k' \leq k} (A[i-1, c-w_{k'i}] + p_{k'i})\}. \tag{5.1}$$

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 5.3 *Recurrence (5.1) 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 from t_1 up to

t_i and let $A[i-c, c-w_{k'i}]$ 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 rearrangements of points from $\{t_1, \dots, t_{i-1}\}$, that 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 $A[i, c]$ cannot be optimal, which is a contradiction. □

The following theorem gives the running time of the dynamic programming algorithm that evaluates Recursion (5.1).

Theorem 5.1 *For arbitrary gains p_{ij}, costs w_{ij} and integer C, Recursion (5.1) 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 runs in pseudopolynomial time. □

An immediate corollary of Theorem 5.1 is the following.

Corollary 5.2 *For the special case of the* `Rearrangement` *problem where we consider only moves (or only bubble-sort swaps), Recurrence (5.1) 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 $t_j \notin \bar{s}_i$ and $w_{ij} = 0$ if $t_j \in \bar{s}_i$. Therefore, the `Rearrangement` problem can be handled efficiently in terms of the rearrangement table (Table 5.1). 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 [CLR90]) and then moving all the points with values higher or equal to this. Therefore, the overall complexity of the SEGMENT&REARRANGE algorithm for the case of moves is $O(2n^2k + 2n)$. For the case of swaps the complexity is $O(2n^2k + nkC^2)$, which is still quadratic to the number of input points when C is a constant.

5.3.2 Pruning the candidates for rearrangement

In the previous section we have considered all points in T as candidate points for rearrangement. Here we restrict this set of candidates. Algorithm 6 is an enhancement of Algorithm 5. 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.

Algorithm 6 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.

1: **Prune:** $D = \text{prune}(T)$
2: **Segment:** $S = S_{\text{opt}}(T - D, k)$
3: **Rearrange:** $\overline{O} = \text{rearrange}(D, S, C)$
4: **Segment:** $S' = S_{\text{opt}}(T_{\overline{O}}, k)$

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 5.1 *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 $\bar{s}_1 = \{10, 9, 10, 10\}$ and $\bar{s}_2 = \{1, 1, 10, 1, 1, 1\}$ 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 to its "correct" position, is expected to be beneficial for the error of the optimal segmentation on the new sequence T'. Consider the move operation $\mathcal{M}(7 \rightarrow 4)$, then $T' = \mathcal{M}(7 \rightarrow 4) \circ T = \{10, 9, 10, 10, 10, 1, 1, 1, 1, 1\}$. The two-segment structure is even more pronounced in sequence T'. The segmentation of T' into two segments would give the segments $\bar{s}'_1 = \{10, 9, 10, 10, 10\}$ and $\bar{s}'_2 = \{1, 1, 1, 1, 1\}$ 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 [JKM99, MSV04].

Following the work of [MSV04] 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, we turn our attention to a single segment $\bar{s}_i$. The total error that this segment contributes to the whole segmentation, before any deviants are removed from it, is $E_p(\bar{s}_i, 1)$. 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 $d_1(t, P) = |t - \text{median}(P)|$, where median$(P)$ is the median value of the points in P. Similarly, for $p = 2$, $d_2(t, P) = |t - \text{mean}(P)|^2$, where mean$(P)$ is the mean value of the points in P. The optimal set of ℓ_i deviants of $\bar{s}_i$ are the set of points $D_i \in \bar{s}_i$ with $|D_i| = \ell_i$ such that

$$D_i = \arg\max_{D'_i} \sum_{t \in D'_i} d_p(t, \bar{s}_i - D_i). \tag{5.2}$$

Example 5.2 *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* $E_2(\bar{s} - D, 1) = 0.75$.

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 $\bar{s}_i$, 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 $\bar{s}_i$ is

$$\hat{D}_i = \arg\max_{D'_i} \sum_{t \in D'_i} d_p(t, \bar{s}_i). \tag{5.3}$$

Example 5.3 *Consider again the segment* $\bar{s} = \{20,\ 10,\ 21,\ 9,\ 21,\ 9,\ 20,\ 9\}$ *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* $E_2(\bar{s} - \hat{D}, 1) = 80.75$.

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 of length n, $\mathcal{T}_n$, and integers $\mathbb{N}$ that represented the possible number of segments. Therefore, for $T \in \mathcal{T}_n$ 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 $\{\bar{s}_1, \ldots, \bar{s}_k\}$ we use $\bar{s}_i$ to represent the points included in segment $\bar{s}_i$.

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

$$E_p\left(T\left[1,n\right],k,\ell\right) = \min_{\substack{1\le i\le n\\ 0\le j\le \ell}} \left\{E_p\left(T\left[1,i\right],k-1,j\right) + E_p\left(T\left[i+1,n\right],1,\ell-j\right)\right\}. \tag{5.4}$$

The recursive formula (5.4) 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 (5.4) 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 (5.2) and (5.3) for deviants and pseudodeviants respectively. From our previous discussion on the complexity of Equations (5.2) and (5.3) we can conclude that Recursion (5.4) 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 (5.4) can compute the optimal set of deviants in time $O(n^2\ell^2)$, using the data structures proposed in [MSV04]. For higher dimensions the complexity of evaluating Recursion (5.2) is unknown. In the case of pseudodeviants and arbitrary dimensionality, the time required for evaluating Recursion (5.4) is $O(n^3\ell)$.

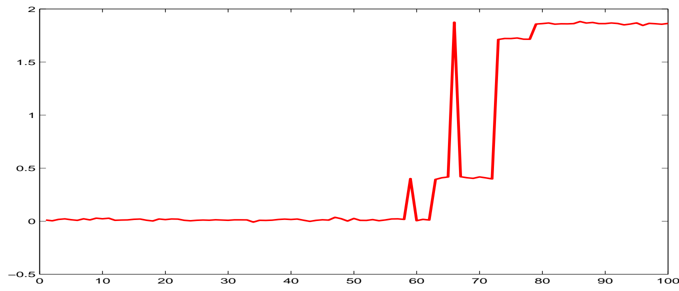
Therefore, the pruning step makes the complexity of TruncatedSegment&Rearrange algorithm cubic, in the case of pseudodeviants. This time requirement may be prohibitive for sequences of large of even medium size. Notice, however, that the expensive step of outlier detection is independent on the rearrangement step, so in practice one can use a faster and less accurate algorithm for finding the outliers. Once the set of outliers is extracted, finding the set of rearrangements can be done as before (or using the Greedy

algorithm described in the next section). We defer the discussion of whether the quality of the results compensates the increase in the computational cost in arbitrary datasets, for the experimental section. Here we just give an indicative (maybe contrived) example (see Figure 5.1) of a case where the extraction of outliers before proceeding in the rearrangement step proves useful.

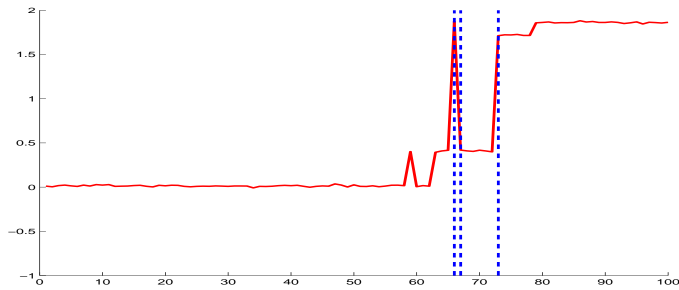
Example 5.4 *Consider the input sequence shown in Figure 5.1(a). The arrangement of the input points is such that the optimal segmentation of the sequence assigns the* 66*-th point, say* p_{66}*, of the sequence alone in one small segment (Figure 5.1(b)). Given the optimal segmentation of the input sequence, 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 5.1(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 5.1(d). This output has error* 0.005 *that is almost* 100% *improvement over the error of the optimal segmentation (without rearrangements).*

5.3.3 The Greedy algorithm

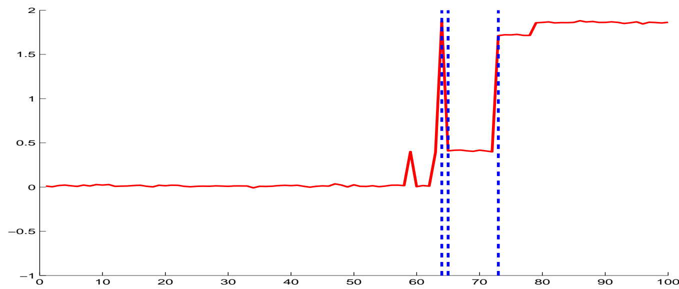
The Segment&Rearrange and the TruncatedSegment & Rearrange algorithms were focusing on finding a sequence of rearrangement operations of cost at most C, that could possibly improve the error of the segmentation of the input sequence T. That is, the algorithms were initially fixing a segmentation, and 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)



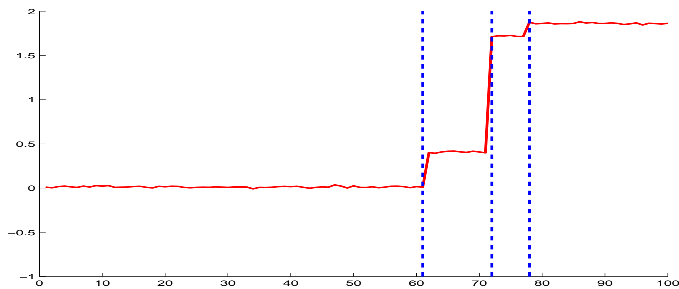
(a) Input sequence



(b) Optimal segmentation of the input sequence



(c) Segmentation with rearrangements using SEGMENT&REARRANGE.



(d) Segmentation with rearrangements using TRUNCATEDSEGMENT&REARRANGE.

Figure 5.1: Pathological example where the TRUNCATEDSEGMENT&REARRANGE algorithm is useful.

readjusts the segmentation that guides the rearrangement of the points after every step. The pseudocode of the GREEDY algorithm is given in Algorithm 7.

Algorithm 7 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.

1: $c \leftarrow 0$
2: $S \leftarrow S_{\text{opt}}(T,k)$
3: $E_1 \leftarrow E\left(S_{\text{opt}}(T,k)\right)$, $E_0 \leftarrow \infty$
4: **while** $c \leq C$ **and** $(E_1 - E_0) < 0$ **do**
5: $\quad E_0 \leftarrow E_1$
6: $\quad \langle \overline{O}, p \rangle \leftarrow \text{LEP}(T,S,C-c)$
7: $\quad c \leftarrow c + |\overline{O}|$
8: $\quad T \leftarrow \overline{O} \circ T$
9: $\quad S \leftarrow S_{\text{opt}}(T,k)$
10: $\quad E_1 \leftarrow E\left(S_{\text{opt}}(T,k)\right)$
11: **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 rearrange and what is 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 C_{LEP} is the time required by the LEP procedure and I the number of iterations of the algorithm, then the GREEDY algorithm needs $O\left(I\left(C_{\text{LEP}} + n^2k\right)\right)$ 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 there-

fore the overall computational cost of the GREEDY algorithm is $O\left(I\left(nk+n^2k\right)\right)$.

5.4 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 $r = E^{\mathcal{A}}/E^*$. 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.

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.

5.4.1 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 $d = 1,5$, 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

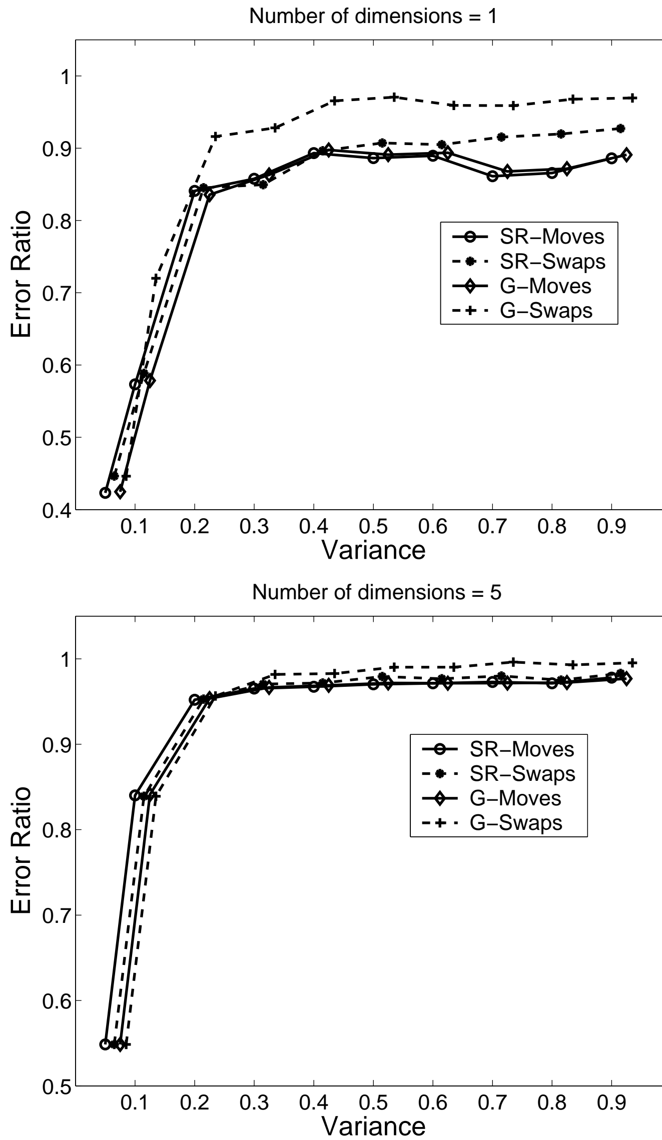


Figure 5.2: Synthetic datasets: error ratio of the SEGMENT&REARRANGE and GREEDY algorithms (with swaps and moves) as a function of the variance used for the data generation. The error ratio is evaluated using the error of the segmentation of the rearranged sequence divided by the error of the segmentation without rearrangements.

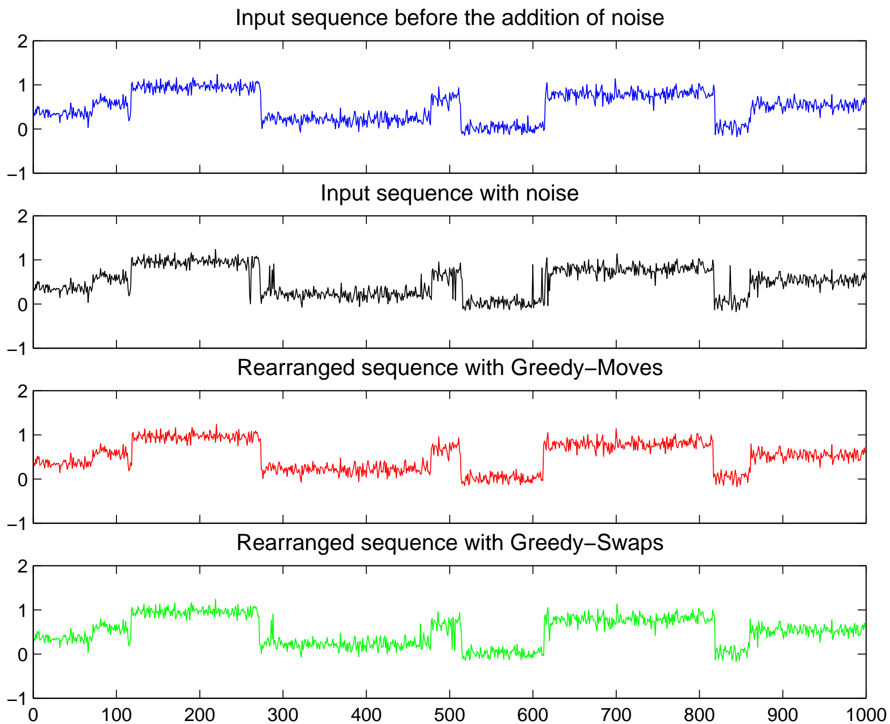


Figure 5.3: Anecdotal evidence of algorithms' qualitative performance. The addition of noise in the input sequence results in the existence of outliers (see for example positions 260, 510, 600 and 840 of the noisy sequence). Observe that GREEDY moves alleviate the effect of outliers completely, giving a perfect rearrangement. An almost perfect rearrangement is obtained using GREEDY algorithm with swaps.

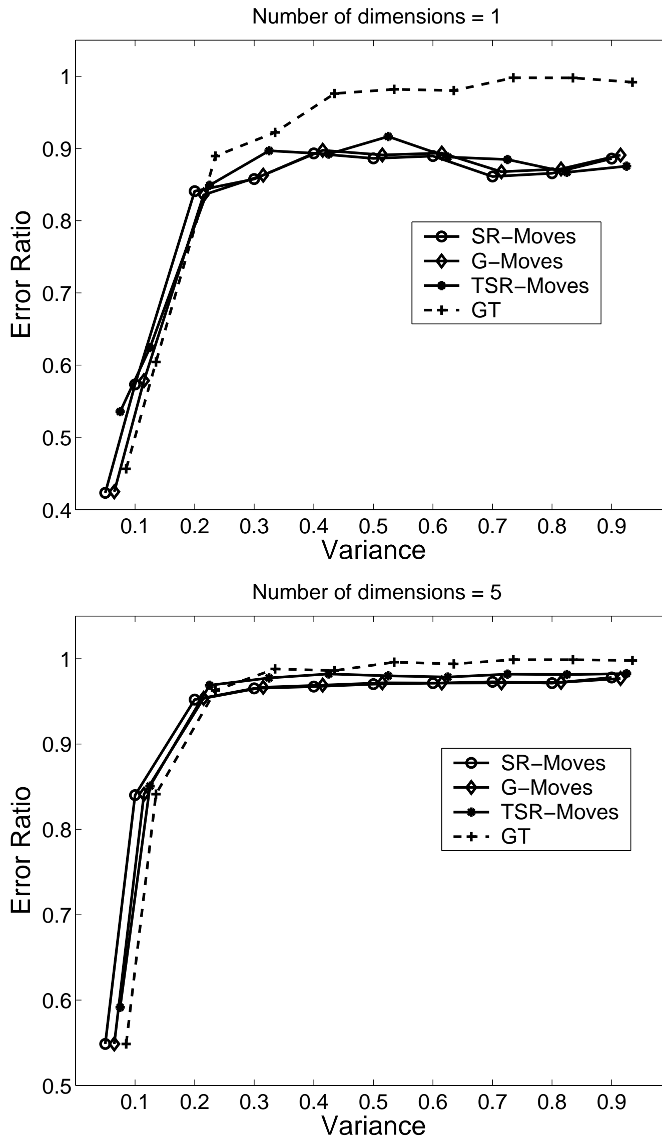


Figure 5.4: Synthetic datasets: error ratio of the Segment&Rearrange and Greedy and TruncatedSegment&Rearrange algorithms (with moves) as a function of the variance used for the data generation. The error ratio is evaluated using the error of the segmentation of the rearranged sequence divided by the error of the segmentation without rearrangements.

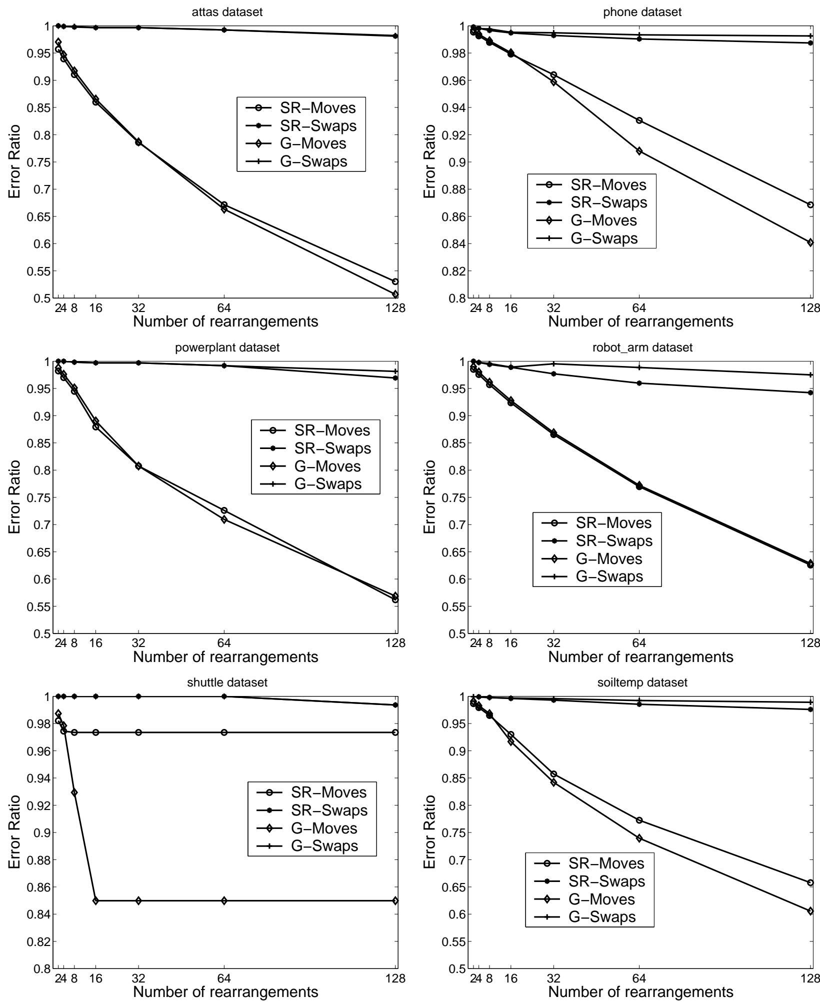


Figure 5.5: Real datasets: error ratio of the Segment&Rearrange and Greedy algorithms (with swaps and moves) as a function of the variance used for the data generation. The error ratio is evaluated using the error of the segmentation of the rearranged sequence divided by the error of the segmentation without rearrangements.

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 t_i (located at position i) that is moved, its new position is uniformly distributed in the interval $[i-l, i+l]$.

Figure 5.2 shows 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 5.3 shows the effect of rearrangements on a noisy input sequence. There are four series shown in Figure 5.3. 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 5.4 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.

5.4.2 Experiments with real data

Figure 5.5 shows the error ratio for different combinations of algorithms and rearrangement types, for a set of real datasets. The real datasets were downloaded from the UCR timeseries 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.

5.5 Conclusions

In this chapter we have introduced and studied the `Segmentation with Rearrangements` problem. In particular we have considered

[1] The interested reader can find the datasets at http://www.cs.ucr.edu/~eamonn/TSDMA/.

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 timeseries 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.

CHAPTER 6

Segmention aggregation

So far we have focused on segmentation algorithms that derive a good representation of the underlying sequential data. The plethora of segmentation algorithms and error functions raises naturally the question, given a specific dataset, what is the segmentation that better captures the underlying structure of the data? In this chapter, we try to answer this question by adopting an approach that assumes that all segmentations found by different algorithms are correct, each one in its own way. That is, each one of them reveals just one aspect of the underlying true segmentation. Therefore, we aggregate the information hidden in the segmentations by constructing a consensus output that reconciles optimally the differences among the given inputs. We call the problem of finding such a segmentation, the `Segmentation Aggregation` problem. We have initially introduced this problem in [MTT06].

This chapter discusses a different view on sequence segmentation. We segment a sequence via aggregation of already existing, but probably contradicting segmentations. The input to our problem is m different segmentations $S_1, \ldots, S_m$. The objective is to produce a single segmentation $\hat{S}$ that agrees as much as possible with the given m segmentations. In the discrete case we define a disagreement between two segmentations S and S' as a pair of points (x, y) such that S places points x and y in the same segment, while S' places them in different segments, or vice versa. If $D_A(S, S')$ denotes the total number of disagreements between S and S', then the segmentation aggregation asks for segmentation $\hat{S}$ that minimizes $C(\hat{S}) = \sum_{i=1}^{m} D_A(S_i, \hat{S})$. The continuous generalization

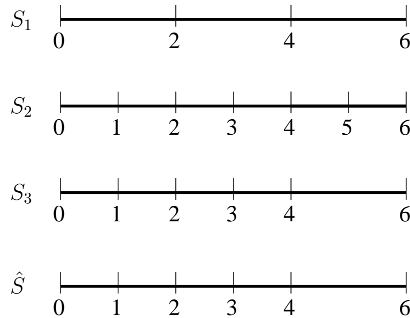


Figure 6.1: Segmentation aggregation that takes into consideration only the segment information.

considers unit intervals of the sequence instead of points.

Consider a sequence of length 6 and three segmentations of this sequence: S_1, S_2 and S_3 as shown in Figure 6.1. (The sequence can be viewed as consisting of the unit intervals $p_i = (i, i+1]$. Hence, the segments consist of unions of p_i's.) Segmentation S_1 has boundaries $\{0, 2, 4, 6\}$. That is, it places intervals p_0 and p_1 in the first segment, p_2 and p_3 in the second and p_4 and p_5 in the third. Similarly, S_2 places each unit interval in a different segment. Segmentation S_3 places p_0, p_1, p_2 and p_3 in different segments, while p_4 and p_5 are assigned in the same last segment of the segmentation. The segmentation $\hat{S}$ in the bottom of Figure 6.1 is the optimal aggregate segmentation for S_1, S_2 and S_3. The total cost of $\hat{S}$ is 3, since $\hat{S}$ has two disagreements with S_1 and one with S_2.

In this chapter we formally define the `Segmentation Aggregation` problem and show that it can be solved optimally in polynomial time using a dynamic programming algorithm. We then apply the segmentation aggregation framework to several problem domains and we illustrate its practical significance.

Related work: For a review of the related literature on segmentation algorithms see Chapter 2. Here, we additionally mention work related to aggregation of data mining results, which has recently emerged in several data mining tasks. The problem of aggregating clusterings has been studied under the names of clustering aggregation [GMT05], consensus clustering [ACN05, LB05] and cluster ensembles [FJ05, SG02]. Ranking aggregation has been studied from the viewpoints of algorithmics [ACN05], Web search [DKNS01],

databases [FKM+04], and machine learning [FISS03]. A third important group of aggregating data mining results is formed by voting classifiers such as bagging [Bre96] and boosting [CSS02]. The spirit of the work presented here is similar to those since we are also trying to aggregate results of existing data mining algorithms. Despite the fact that the segmentation problem has received considerable attention by the data mining community, to the best of our knowledge, the `Segmentation Aggregation` problem has not been studied previously.

6.1 Application domains

Segmentation aggregation is useful in many scenarios. We list some of them below.

Analysis of genomic sequences: A motivating problem of important practical value is the *haplotype block structure* problem. The block structure discovery in haplotypes is considered one of the most important discoveries for the search of structure in genomic sequences. To explain this notion, consider a collection of DNA sequences over n marker sites for a population of p individuals. Consider a marker site to be a location on the DNA sequence associated with some value. This value is indicative of the genetic variation of individuals in this location. The "haplotype block structure" hypothesis states that the sequence of markers can be segmented in blocks, so that, in each block most of the haplotypes of the population fall into a small number of classes. The description of these haplotypes can be used for further knowledge discovery, e.g., for associating specific blocks with specific genetic influenced diseases [Gus03].

As we have already mentioned in Chapter 4, the problem of discovering haplotype blocks in genetic sequences can be viewed as that of partitioning a multidimensional sequence into segments, such that, each segment demonstrates low diversity along the different dimensions. Different segmentation algorithms have been applied to good effect on this problem. However, these algorithms either assume different generative models for the haplotypes or optimize different criteria. As a result, they output block structures that can

be (slightly or completely) different. In this setting, the segmentation aggregation assumes that all models and optimization criteria contain useful information about the underlying haplotype structure, and aggregates their results to obtain a single block structure that is hopefully a better representation of the underlying truth.

Segmentation of multidimensional categorical data: The segmentation aggregation framework gives a natural way of segmenting multidimensional categorical data. Although the problem of segmenting multidimensional numerical data is rather natural, the segmentation problem of multidimensional categorical sequences has not been considered widely, mainly because such data are not easy to handle. Consider an 1-dimensional sequence of points that take nominal values from a finite domain. In such data, a segment is defined by consecutive points that take the same value. For example, the sequence $a\ a\ a\ b\ b\ b\ c\ c$, has 3 segments ($a\ a\ a$, $b\ b\ b$ and $c\ c$). When the number of dimensions in such data increases the corresponding segmentation problem starts getting more complicated. Each dimension has its own clear segmental structure. However, the segmentation of the sequence when all dimensions are considered simultaneously is rather difficult to be found by conventional segmentation algorithms. Similar difficulties in using standard segmentation algorithms appear when the multidimensional data exhibit a mix of nominal and numerical dimensions.

Robust segmentation results: Segmentation aggregation provides a concrete methodology for improving segmentation robustness by combining the results of different segmentation algorithms, which may use different criteria for the segmentation, or different initializations of the segmentation method. Note also that most of the segmentation algorithms are sensitive to erroneous or noisy data. However, such data are very common in practice. For example, sensors reporting measurements over time may fail (e.g., run out of battery), genomic data may have missing values (e.g., due to insufficient wet-lab experiments). Traditional segmentation algorithms show little robustness to such scenarios. However, when their results are adequately combined, via aggregation, the effect of missing or faulty data in the final segmentation is expected to be alleviated.

Clustering segmentations: Segmentation aggregation gives a

natural way to cluster segmentations. In such a clustering each cluster is represented by the aggregated segmentation and the cost of the clustering is the sum of the disagreements within the clusters. Segmentation aggregation defines a natural representative of a cluster of segmentations. Given a cluster of segmentations, its representative is the aggregation of the segmentations of the cluster. Furthermore, the disagreements distance is a metric. Hence, various distance-based data mining techniques can readily be applied to segmentations, and the distance function being a metric also provides approximation guarantees for many of them.

Summarization of event sequences: An important line of research is focusing on mining event sequences [AS95, GAS05, Kle02, MTV97]. An event sequence consists of a set of events of specific types that occur at certain points on a given timeline. For example, consider a user accessing a database at time points $t_1, t_2, \ldots, t_k$ within a day. Or a mobile phone user making phone calls, or transferring between different cells. Having the activity times of the specific user for a number of different days one could raise the question: *How does the user's activity on an average day look like?* One can consider the time points at which events occur as segment boundaries. In that way, forming the profile of the user's daily activity is mapped naturally to a segmentation aggregation problem.

6.2 The `Segmentation Aggregation` problem

6.2.1 Problem definition

Let T be a timeline of bounded length. Unless otherwise stated, we will assume that T is a continuous interval of length n. For the purpose of exposition we will also often talk about discrete timelines. A discrete timeline T of size n can be thought of as the timeline T been discretized into n subintervals of unit length.

Here, we recall some of the notational conventions we have been following, and which are important for the rest of the chapter. A segmentation S is a partition of T into continuous intervals (segments). Formally, we define $S = \{s_0, s_1, \ldots, s_k\}$, where $s_i \in T$ are the *breakpoints* (or *boundaries*) of the segmentation and it holds that $s_i < s_{i+1}$, for every i. We will always assume that $s_0 = 0$

and $s_k = n$. We define the i-th segment $\bar{s}_i$ of S to be the interval $\bar{s}_i = (s_{i-1}, s_i]$. The length of S, denoted by $|S|$ is the number of segments in S. Note that there is an one to one mapping between the end boundaries of the segments and the segments themselves. We will often abuse the notation and define a segmentation as a set of segments instead of a set of boundaries. In these cases we will always assume that the segments define a partition of the timeline, and thus they uniquely define a set of boundaries.

For a set of m segmentations $S_1, \ldots, S_m$ over the same timeline T we define their *union segmentation* to be the segmentation with boundaries $U = \bigcup_{i=1}^{m} S_i$. Assume that function D takes as input two segmentations (of the same timeline T) and evaluates how differently the two segmentations partition T. Given such a distance function, we can define the `Segmentation Aggregation` problem as follows.

Problem 6.1 (The `Segmentation Aggregation` problem) *Given a set of m segmentations $S_1, S_2, ..., S_m$ of timeline T, and a distance function D between them, find a segmentation $\hat{S}$ of T that minimizes the sum of the distances from all the input segmentations. That is,*

$$\hat{S} = \arg\min_{S \in \mathcal{S}_n} \sum_{i=1}^{m} D(S, S_m).$$

We define $C(\hat{S}) = \sum_{i=1}^{m} D(S, S_m)$ to be the *cost* of the aggregate segmentation.

Note that Problem 6.1 is defined independently of the distance function D used between segmentations. We focus our attention on the disagreement distance D_A. However, alternative distance functions are discussed in Sections 6.5 and 6.6.

6.2.2 The disagreement distance

In this section we formally define the notion of distance between two segmentations. Our distance function is based on similar distance functions proposed for clustering [GMT05]. The intuition of the distance function is drawn from the discrete case. Given two discrete timeline segmentations, the disagreement distance is the total number of pairs of points that are placed in the same segment

in one segmentation, while placed in different segments in the other. We now generalize the definition to the continuous case.

Let $P = \{\bar{p}_1, \ldots, \bar{p}_{\ell_p}\}$ and $Q = \{\bar{q}_1, \ldots, \bar{q}_{\ell_q}\}$ be two segmentations. Let $U = P \cup Q$ be their union segmentation with segments $\{\bar{u}_1, \ldots, \bar{u}_{\ell_u}\}$. Note that by definition of the union segmentation, for every $\bar{u}_i$ there exist segments $\bar{p}_k$ and $\bar{q}_t$ such that $\bar{u}_i \subseteq \bar{p}_k$ and $\bar{u}_i \subseteq \bar{q}_t$. We define $P(\bar{u}_i) = k$ and $Q(\bar{u}_i) = t$, to be the *labeling* of interval $\bar{u}_i \in U$ with respect to segmentations P and Q respectively. Similar to the discrete case we now define a disagreement when two segments $\bar{u}_i$, and $\bar{u}_j$ receive the same label in one segmentation, but different labels in the other. The disagreement is weighted by the product of the segment lengths $|\bar{u}_i||\bar{u}_j|$. Intuitively, the length captures the number of points contained in the interval, and the product the number of disagreements between the intervals.

Formally, the disagreement distance of P and Q on segments $\bar{u}_i$, and $\bar{u}_j$ is defined as follows

$$d_{P,Q}(\bar{u}_i, \bar{u}_j) = \begin{cases} |\bar{u}_i||\bar{u}_j|, & \text{if } P(\bar{u}_i) = P(\bar{u}_j) \text{ and } Q(\bar{u}_i) \neq Q(\bar{u}_j) \\ & \text{or } Q(\bar{u}_i) = Q(\bar{u}_j) \text{ and } P(\bar{u}_i) \neq P(\bar{u}_j) \\ 0, & \text{otherwise.} \end{cases}$$

The overall disagreement distance between two segmentations can now be defined naturally as follows.

Definition 6.1 *Let P and Q be two segmentations of timeline T and let U their union segmentation with segments $\{\bar{u}_1, \ldots, \bar{u}_t\}$. The disagreement distance, D_A, between P and Q is defined to be*

$$D_A(P, Q) = \sum_{(\bar{u}_i, \bar{u}_j) \in U \times U} d_{P,Q}(\bar{u}_i, \bar{u}_j).$$

It is rather easy to prove that the distance function D_A is a metric. This property is significant for applications of the distance function such as clustering, where good worst case approximation bounds can be derived in metric spaces.

6.2.3 Computing the disagreement distance

For two segmentations P and Q with ℓ_p and ℓ_q number of segments respectively, the distance $D_A(P, Q)$ can be computed trivially in

time $O\left((\ell_p + \ell_q)^2\right)$. Next we show that this can be done even faster in time $O\left(\ell_p + \ell_q\right)$. Furthermore, our analysis helps in building intuition on the general aggregation problem. This intuition will be useful in the following sections.

We first define the notion of *potential energy.*

Definition 6.2 *Let $\bar{v} \subseteq T$ be an interval of length $|\bar{v}|$ of timeline T. We define the* potential energy *of the interval to be*

$$E(\bar{v}) = \frac{|\bar{v}|^2}{2}. \tag{6.1}$$

Let P be a segmentation with segments $P = \{\bar{p}_1, \ldots, \bar{p}_{\ell_p}\}$. We define the potential energy *of segmentation P to be*

$$E(P) = \sum_{\bar{p}_i \in P} E(\bar{p}_i).$$

The potential energy computes the potential disagreements that the interval $\bar{v}$ can create. To better understand the intuition behind it we resort again to the discrete case. Let $\bar{v}$ be an interval in the discrete timeline, and let $|\bar{v}|$ be the number of points in $\bar{v}$. There are $|\bar{v}|(|\bar{v}|-1)/2$ distinct pairs of points in $\bar{v}$, all of which can potentially cause disagreements with other segmentations.

Each of the discrete points in the interval can be thought of as a unit-length elementary subinterval and there are $|\bar{v}|(|\bar{v}| - 1)/2$ pairs of those in $\bar{v}$, all of which are potential disagreements. Considering the continuous case is actually equivalent to focusing on very small (instead of unit length) subintervals. Let the length of these subintervals be ϵ with $\epsilon << 1$. Then, the number of potential disagreements caused by all the ϵ-length intervals in $\bar{v}$ is

$$\frac{|\bar{v}|/\epsilon\left(|\bar{v}|/\epsilon - 1\right)}{2} \cdot \epsilon^2 \quad = \quad \frac{|\bar{v}|^2 - \epsilon|\bar{v}|}{2} \rightarrow \frac{|\bar{v}|^2}{2}$$

when $\epsilon \rightarrow 0$.[1]

The potential energy of a segmentation is the sum of the potential energies of the segments it contains. Given the above definition we can show the following basic lemma.

[1] We can obtain the same result by integration. However, we feel that this helps to better understand the intuition of the definition.

Lemma 6.1 *Let P and Q be two segmentations and let U be their union segmentation. The distance $D_A(P,Q)$ can be computed by the closed formula*

$$D_A(P,Q) \quad = \quad E(P) + E(Q) - 2E(U). \tag{6.2}$$

Proof. For simplicity of exposition we will present the proof in the discrete case and talk in terms of points (rather than intervals), though the extension to intervals is straightforward. Consider the two segmentations P and Q and a pair of points $x, y \in T$. For point x, let $P(x)$ (respectively $Q(x)$) be the index of the segment that contains x in P (respectively in Q). By definition, the pair (x, y) introduces a disagreement if one of the following two cases is true:

Case 1: $P(x) = P(y)$ but $Q(x) \neq Q(y)$, or

Case 2: $Q(x) = Q(y)$ but $P(x) \neq P(y)$.

The term $E(P)$ in Equation (6.2) gives all the pairs of points that are in the same segments in segmentation P. Similarly, the term $E(U)$ stands for the pairs of points that are in the same segments in the union segmentation U. Their difference is the number of pairs that are in the same segment in P but not in the same segment in U. However, if for two points x, y it holds that $P(x) = P(y)$ and $U(x) \neq U(y)$, then $Q(x) \neq Q(y)$, since U is the union segmentation of P and Q. Therefore, the term $E(X) - E(U)$ counts all the disagreements due to Case 1. Similarly, the disagreements due to Case 2 are counted in the term $E(Q) - E(U)$. Therefore, Equation (6.2) gives the total number of disagreements between segmentations P and Q. □

Lemma 6.1 allows us to state the following theorem.

Theorem 6.1 *Given two segmentations P and Q with ℓ_p and ℓ_q segments respectively, $D_A(P,Q)$ can be computed in time $O(\ell_p + \ell_q)$.*

6.3 Aggregation algorithms

In this section we give exact and heuristic algorithms for the `Segmentation Aggregation` problem for distance function D_A. First,

we show that the optimal segmentation aggregation contains only segment boundaries in the union segmentation. That is, no new boundaries are introduced in the aggregate segmentation. Based on this observation we can construct a dynamic programming algorithm that solves the problem exactly even in the continuous setting. If N is the number of boundaries in the union segmentation, and m the number of input segmentations, the dynamic programming algorithm runs in time $O(N^2m)$. We also propose faster greedy heuristic algorithms that run in time $O\left(N(m+\log N)\right)$ and, as shown in the experimental section, give high quality results in practice.

6.3.1 Candidate segment boundaries

If U is the union segmentation of $S_1,\ldots,S_m$, then Theorem 6.2 establishes the fact that the boundaries of the optimal aggregation are subset of the boundaries appearing in U.

The consequences of the theorem are twofold. For the discrete version of the problem, where the input segmentations are defined over discrete sequences of n points, Theorem 6.2 restricts the search space of output aggregations. That is, only 2^N (instead of 2^n) segmentations are valid candidate aggregations. More importantly this pruning of the search space allows us to map the continuous version of the problem to a discrete combinatorial search problem and to apply standard algorithmic techniques for solving it.

Theorem 6.2 *Let $S_1,S_2,\ldots,S_m$ be the m input segmentations to the segmentation aggregation problem for D_A distance. Additionally, let U be their union segmentation. For the optimal aggregate segmentation $\hat{S}$, it holds that $\hat{S}\subseteq U$, that is, all the segment boundaries in $\hat{S}$ belong in U.*

Proof. The proof is by contradiction. Assume that the optimal aggregate segmentation $\hat{S}$ has cost $C(\hat{S}) = \sum_{i=1}^{m} D_A(\hat{S},S_i)$ and that $\hat{S}$ contains a segment boundary $j\in T$ such $j\notin U$. Assume that we have the freedom to move boundary j to a new position x_j. We will show that this movement will reduce the cost of the aggregate segmentation $\hat{S}$, which will contradict the optimality assumption.

We first consider a single input segmentation $S\in\{S_1,\ldots,S_m\}$ and denote its union with the aggregate segmentation $\hat{S}$ by U_S.

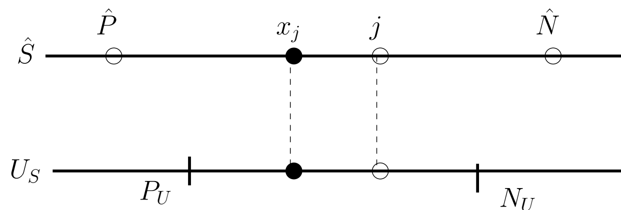


Figure 6.2: Boundary arrangement for the proof of Theorems 6.2 and 6.5.

Assume that we move boundary j to position x_j in segmentation $\hat{S}$. However, this movement is restricted to be within the smallest interval in U_S that contains j. (Similar arguments can be made for any segment in U_S.) Consider the boundary point arrangement shown in Figure 6.2. In this arrangement we denote by $\hat{P}$ (P_U) the first boundary of $\hat{S}$ (U_S) that is to the left of x_j and by $\hat{N}$ (N_U) the first boundary of $\hat{S}$ (U_S) that is to the right of x_j. We know by Lemma 6.1 that

$$D_A(S, \hat{S}) \quad = \quad E(S) + E(\hat{S}) - 2E(U_S),$$

or simply

$$D_A \quad = \quad E_S + E_{\hat{S}} - 2E_{U_S}.$$

Note that $E_{\hat{S}}$ and E_{U_S} both depend on the position of x_j, while E_S does not since $x_j \notin S$. Thus, by writing D_A as a function of x_j we have that

$$D_A(x_j) \quad = \quad E_S + E_{\hat{S}}(x_j) - 2E_{U_S}(x_j), \tag{6.3}$$

where

$$E_{\hat{S}}(x_j) \quad = \quad c_{\hat{S}} + \frac{(x_j - \hat{P})^2}{2} + \frac{(\hat{N} - x_j)^2}{2} \tag{6.4}$$

and

$$E_{U_S}(x_j) \quad = \quad c_U + \frac{(x_j - P_U)^2}{2} + \frac{(N_U - x_j)^2}{2}. \tag{6.5}$$

In Equations (6.4) and (6.5) the terms $c_{\hat{S}}$ and c_U correspond to constant factors that are independent of x_j. If we substitute Equations (6.4) and (6.5) into Equation (6.3) and take the first derivative

of this with respect to x_j we have that

$$\begin{aligned}\frac{dD_A(x_j)}{dx_j} &= 0 + \frac{2(x_j - \hat{P})}{2} - \frac{2(\hat{N} - x_j)}{2} \\ &\quad -2\frac{2(x_j - P_U)}{2} + 2\frac{2(N_U - x_j)}{2}.\end{aligned}$$

Taking the second derivative we get that

$$\frac{d^2 D_A(x_j)}{dx_j^2} = 1 + 1 - 2 - 2 = -2 < 0.$$

The second derivative being negative implies that the function is convex in the interval $[P_U, N_U]$ and therefore it will exhibit its local minima in the interval's endpoints. That is, $x_j \in \{P_U, N_U\}$ which contradicts the initial optimality assumption. Note that the above argument is true for all input segmentations in the particular interval and therefore it is also true for their sum. □

6.3.2 Finding the optimal aggregation for D_A

We now formulate the dynamic programming algorithm that solves optimally the `Segmentation Aggregation` problem. We first need to introduce some notation. Let $S_1, \ldots, S_m$ be the input segmentations, and let $U = \{u_0, u_1, \ldots, u_N\}$ be the union segmentation. Consider a *candidate* aggregate segmentation $A \subseteq U$, and let $C(A)$ denote the cost of A, that is, the sum of distances of A to all input segmentations. We write $C(A) = \sum_i C_i(A)$, where $C_i(A)$ is the distance between A and segmentation S_i. The optimal aggregate segmentation is the segmentation $\hat{S}$ that minimizes the cost $C(\hat{S})$. Also, we define a j*-restricted* segmentation A^j to be a candidate segmentation such that the next-to-last breakpoint is restricted to be the point $u_j \in U$. That is, the segmentation is of the form $A^j = \{u_0, \ldots, u_j, n\}$: it contains u_j, and does not contain any breakpoint $a_k > u_j$ (except for the last point of the sequence). We use $\mathcal{A}^j$ to denote the set of all j-restricted segmentations, and $\hat{S}^j$ to denote the one with the smallest cost. As an extreme example, for $j = 0$, $\hat{S}^0 = \{u_0, n\}$ consists of a single segment. Abusing slightly the notation, for $j = N$, where the next-to-last and the last segmentation breakpoint coincide to be u_N, we have that $\hat{S}^N = \hat{S}$, that is the optimal aggregate segmentation.

Now let A be a candidate segmentation, and let $u_k \in U$ be a boundary point in the union that does not belong to A, $u_k \notin A$. We define the *impact* of u_k to A to be the change (increase or decrease) in the cost that is caused by adding breakpoint u_k to the A, that is, $I(A, u_k) = C(A \cup \{u_k\}) - C(A)$. We have that $I(A, u_k) = \sum_i I_i(A, u_k)$, where $I_i(A, u_j) = C_i(A \cup \{u_k\}) - C_i(A)$.

We can now prove the following theorem.

Theorem 6.3 *The cost of the optimal solution for the* `Segmentation Aggregation` *problem can be computed using a dynamic programming algorithm with the recursion*

$$C(\hat{S}^j) = \min_{0 \leq k < j} \left\{ C(\hat{S}^k) + I(\hat{S}^k, u_j) \right\}. \tag{6.6}$$

Proof. For the proof of correctness it suffices to show that the impact of adding breakpoint u_j to a k-restricted segmentation is the same for all $A^k \in \mathcal{A}^k$. Then, Recursion (6.6) calculates the minimum cost aggregation correctly, since the two terms appearing in the summation are independent.

For proving the above claim pick any k-restricted segmentation A^k with boundaries $\{u_0, \ldots, u_k, n\}$ and segments $\{\bar{a}_1, \ldots, \bar{a}_{k+1}\}$. Assume that we extend A^k to $A^j = A^k \cup \{u_j\}$ by adding boundary $u_j \in U$. We focus first on a single input segmentation S_i and we denote by U_i^k the union of segmentation S_i with the k-restricted aggregation A^k. Then it is enough to show that $I_i(A^k, u_j)$ is independent of the selected A^k. If this is true for $I_i(A^k, u_j)$ and any i, then it will also be true for $I(A^k, u_j) = \sum_{i=1}^m I_i(A^k, u_j)$. By Lemma 6.1 we have that the impact of u_j is

$$\begin{aligned} I_i(A^k, u_j) &= C_i(A^k \cup \{u_j\}) - C_i(A^k) \\ &= E(A^k \cup \{u_j\}) + E(S_i) - 2E(U_i^j) \\ &\quad -E(A^k) - E(S_i) + 2E(U_i^k). \end{aligned} \tag{6.7}$$

Consider now the addition of boundary $u_j > u_k$. This addition splits the last segment of A^k into two subsegments $\bar{\beta}_1$ and $\bar{\beta}_2$ such that $|\bar{\beta}_1| + |\bar{\beta}_2| = |\bar{a}_{k+1}|$. Assume that $u_j \in S_i$. This means that the addition of u_j in the aggregation does not change segmentation

U_i^k. That is, $U_i^k = U_i^j$. Substituting into Equation (6.7) we have that

$$\begin{aligned} I_i(A^k, u_j) &= E(A^k) - E(\bar{a}_{k+1}) + E(\bar{\beta}_1) + E(\bar{\beta}_2) \\ &\quad + E(S_i) - 2E(U_i^k) - E(A^k) - E(S_i) + 2E(U_i^k) \\ &= -E(\bar{a}_{k+1}) + E(\bar{\beta}_1) + E(\bar{\beta}_2) \\ &= -\frac{|\bar{a}_{k+1}|^2}{2} + \frac{|\bar{\beta}_1|^2}{2} + \frac{|\bar{\beta}_2|^2}{2} \\ &= -|\bar{\beta}_1||\bar{\beta}_2|. \end{aligned} \tag{6.8}$$

In the case where $u_j \notin S_i$, the addition of u_j in the aggregation causes a change in segmentation U_i^k as well. Let segment $\bar{u}$ of U_i^k be split into segments $\bar{\gamma}_1$ and $\bar{\gamma}_2$ in segmentation U_i^j. The split is such that $|\bar{u}| = |\bar{\gamma}_1| + |\bar{\gamma}_2|$ The impact of boundary u_j now becomes

$$\begin{aligned} I_i(A^k, u_j) &= E(A^k) - E(\bar{a}_{k+1}) + E(\bar{\beta}_1) + E(\bar{\beta}_2) + E(S_i) \\ &\quad - 2E(U_i^k) + 2E(\bar{u}) - 2E(\bar{\gamma}_1) - 2E(\bar{\gamma}_2) \\ &\quad - E(A^k) - E(S_i) + 2E(U_i^k) \\ &= -E(\bar{a}_{k+1}) + E(\bar{\beta}_1) + E(\bar{\beta}_2) \\ &\quad + 2E(\bar{u}) - 2E(\bar{\gamma}_1) - 2E(\bar{\gamma}_2) \\ &= -\frac{|\bar{a}_{k+1}|^2}{2} + \frac{|\bar{\beta}_1|^2}{2} + \frac{|\bar{\beta}_2|^2}{2} + |\bar{u}|^2 - |\bar{\gamma}_1|^2 - |\bar{\gamma}_2|^2 \\ &= -|\bar{\beta}_1||\bar{\beta}_2| + 2|\bar{\gamma}_1||\bar{\gamma}_2|. \end{aligned} \tag{6.9}$$

In the update rules (6.8) and (6.9), the terms that determine the impact of boundary u_j do not depend on the boundaries of A^k in the interval $[u_0, u_k]$. They only depend on where the new boundary u_j is placed in the interval $(u_k, n]$. Therefore, the impact term is the same for all $A^k \in \mathcal{A}^k$. □

Computing the impact of every point can be done in $O(m)$ time (constant time is needed for each S_i) and therefore the total computation needed for the evaluation of the dynamic programming recursion is $O(N^2 m)$.

6.3.3 Greedy algorithms

The dynamic programming algorithm produces an optimal solution with respect to the disagreements distance, but it runs in time

quadratic in N, the size of the union segmentation. This makes it impractical for large datasets. We therefore need to consider faster heuristics.

The GreedyBU algorithm

In this paragraph we describe a greedy bottom-up (GreedyBU) approach to segmentation aggregation. The algorithm starts with the union segmentation U. Let $A_1 = U$ denote this initial aggregate segmentation. At the t-th step of the algorithm we identify the boundary b in A_t whose removal causes the maximum decrease in the distance between the aggregate segmentation and the input segmentations. By removing b we obtain the next aggregate segmentation A_{t+1}. If no boundary that causes cost reduction exists, the algorithm stops and it outputs the segmentation A_t.

At some step t of the algorithm, let $C(A_t)$ denote the cost of the aggregate segmentation A_t constructed so far, that is, the sum of distances from A_t to all input sequences. We have that $C(A_t) = \sum_\ell C_\ell(A_t)$, where $C_\ell(A_t) = D_A(A_t, S_\ell)$, the distance of A_t from the input segmentation S_ℓ. For each boundary point $b \in A_t$, we need to store the *impact* of removing b from A_t, that is, the change in $C(A_t)$ that the removal of boundary b causes. This may be negative, meaning that the cost decreases, or positive, meaning that the cost increases. We denote this impact by $G(b)$ and as before, it can be written as $G(b) = \sum_\ell G_\ell(b)$.

We now show how to compute and maintain the impact in an efficient manner. More specifically we show that at any step the impact for a boundary point b can be computed by looking only at *local* information; the segments adjacent to b. Furthermore, the removal of b affects the impact only of the adjacent boundaries in A_t, thus updates are also fast.

For the computation of $G(b)$ we make use of Lemma 6.1. Let $A_t = \{a_0, a_1, \ldots, a_b\}$ be the aggregate segmentation at step t, and let $S_\ell = \{s_0, s_1, \ldots, s_d\}$ denote one of the input segmentations. Also let $U_\ell = \{u_0, u_1, \ldots, u_e\}$ denote the union segmentation of A_t and S_ℓ. Then the distance from A_t to S_ℓ can be computed as

$$\begin{aligned} C_\ell(A_t) &= D_A(A_t, S_\ell) \\ &= E(S_\ell) + E(A_t) - 2E(U_\ell). \end{aligned}$$

The potential $E(S_\ell)$ does not depend on the aggregate segmentation A_t. Therefore, when removing a boundary point b, $E(S_\ell)$ remains unaffected. We only need to consider the effect of b on the potentials $E(A_t)$ and $E(U_\ell)$.

Assume that $b = a_j$ is the j-th boundary point of A_t. Removing a_j causes segments $\bar{a}_j$ and $\bar{a}_{j+1}$ to be merged, creating a new segment of size $|\bar{a}_j| + |\bar{a}_{j+1}|$ and diminishing two segments of size $|\bar{a}_j|$ and $|\bar{a}_{j+1}|$. Therefore, the potential energy of the resulting segmentation A_{t+1} is

$$\begin{aligned} E(A_{t+1}) &= E(A_t) + \frac{(|\bar{a}_j| + |\bar{a}_{j+1}|)^2}{2} - \frac{|\bar{a}_j|^2}{2} - \frac{|\bar{a}_{j+1}|^2}{2} \\ &= E(A_t) + |\bar{a}_j||\bar{a}_{j+1}|. \end{aligned}$$

The boundary b, that is removed from A_t, is also a boundary point of U_ℓ. If $b \in S_\ell$, then the boundary remains in U_ℓ even after it is removed from A_t. Thus, the potential energy $E(U_\ell)$ does not change. Therefore, the impact is $G_\ell(b) = |\bar{a}_j||\bar{a}_{j+1}|$

Consider now the case that $b \notin S_\ell$. Assume that $b = u_i$ is the i-th boundary of U, that separates segments $\bar{u}_i$ and $\bar{u}_{i+1}$. The potential energy of U_ℓ increases by $|\bar{u}_i||\bar{u}_{i+1}|$. Thus the total impact caused by the removal of b is $G_\ell(b) = |\bar{a}_j||\bar{a}_{j+1}| - 2|\bar{u}_i||\bar{u}_{i+1}|$.

Therefore, the computation of $G_\ell(b)$ can be done in constant time with the appropriate data structure for obtaining the lengths of the segments adjacent to b. Going through all input segmentations we can compute $G(b)$ in time $O(m)$.

Computing the impact of all boundary points takes time $O(Nm)$. Updating the costs in a naive way would result in an algorithm with cost $O(N^2m)$. However, we do not need to update all boundary points. Since the impact of a boundary point depends only on the adjacent segments only the impact values of the neighboring boundary points are affected. If $b = a_j$, we only need to recompute the impact for a_{j-1} and a_{j+1}, which can be done in time $O(m)$. Therefore, using a simple heap to store the benefits of the breakpoints we are able to compute the aggregate segmentation in time

$O(N(m + \log N))$. A similar methodology can also be used for a greedy top-down algorithm resulting in an algorithm with the same complexity.

The GreedyTD algorithm

In this paragraph we describe a greedy top-down (GreedyTD) algorithm for segmentation aggregation. Initially, the algorithm starts with the empty segmentation (the segmentation that has no other boundaries than the beginning and the end of the sequence). Let $A_1 = \{0, n\}$ be the initial aggregate segmentation. Segment boundaries are added to this empty segmentation one step at a time. That is, in the t-th step of the algorithm the t-th boundary is added. The boundaries are again picked from the union segmentation of the inputs $S_1, \ldots, S_m$, which we denote by U. At the t-th step of the algorithm we identify the boundary b in U whose addition causes the maximum decrease in the distance between the aggregate segmentation and the input segmentations. By adding b we obtain the next aggregate segmentation A_{t+1}. If no boundary that causes cost reduction exists, the algorithm stops and it outputs the segmentation A_t.

As before, we use $C(A_t)$ to denote the cost of the aggregate segmentation A_t, that is, the sum of distances from A_t to all input segmentations. Thus $C(A_t) = \sum_\ell C_\ell(A_t)$, where $D_\ell(A_t) = D_A(A_t, S_\ell)$ is the distance of A_t to segmentation S_ℓ. In this case, the *impact* of boundary $b \in U$ is the change in $C(A_t)$ that the addition of boundary b can cause. We denote the impact of boundary b by $G(b) = \sum_\ell G_\ell(b)$, where $G_\ell(b)$ is the impact that the addition of boundary b has on the cost induced by input segmentation S_ℓ.

We now show how to compute and maintain the impact of each candidate boundary point efficiently. Assume that we are at step t of the execution of the algorithm. Denote by $A_t = \{a_0, a_1, \ldots, a_b\}$ the aggregate segmentation at this step, and let $S_\ell = \{s_0, s_1, \ldots, s_d\}$ denote one of the input segmentations. Also let U_ℓ denote the union segmentation of A_t and S_ℓ. Then as in the previous paragraph, the distance of A_t from S_ℓ can be computed as

$$\begin{aligned} C_\ell(A_t) &= D_A(A_t, S_\ell) \\ &= E(S_\ell) + E(A_t) - 2E(U_\ell). \end{aligned}$$

Again the potential energy $E(S_\ell)$ does not depend on the aggregate segmentation and therefore it remains constant in all steps of the algorithm. We only need to consider the effect of adding b to the potential energies $E(A_t)$ and $E(U_\ell)$. The change in $E(A_t)$ is easy to compute. This is because the addition of boundary b will cause the split of a single segment in A_t. Let this segment be $\bar{a}_j$, with length $|\bar{a}_j|$. Now assume that adding b splits $\bar{a}_j$ into to segments $\bar{a}_{j_1}$ and $\bar{a}_{j_2}$ such that $|\bar{a}_{j_1}| + |\bar{a}_{j_2}| = |\bar{a}_j|$. Then potential energy of segmentation A_{t+1} will be

$$\begin{aligned} E(A_{t+1}) &= E(A_t) + \frac{|\bar{a}_{j_1}|^2}{2} + \frac{|\bar{a}_{j_2}|^2}{2} - \frac{(|\bar{a}_{j_1}| + |\bar{a}_{j_2}|)^2}{2} \\ &= E(A_t) - |\bar{a}_{j_1}||\bar{a}_{j_2}|. \end{aligned}$$

The boundary point b that has been added in A_t is also a boundary point in U_ℓ. However, if $b \in S_\ell$, then b was in U_ℓ already and therefore the potential energy $E(U_\ell)$ remains unchanged. If $b \notin S_\ell$, then the addition of b splits a segment $\bar{u}_i$ of U_ℓ in two smaller segments $\bar{u}_{i_1}$ and $\bar{u}_{i_2}$ such that $|\bar{u}_{i_1}| + |\bar{u}_{i_2}| = |\bar{u}_i|$. Then it holds that the new potential energy of U_ℓ after the addition of boundary b will be reduced by $|\bar{u}_{i_1}||\bar{u}_{i_2}|$. Thus, the impact of boundary point b is $G_\ell(b) = 2|\bar{u}_{i_1}||\bar{u}_{i_2}| - |\bar{a}_{j_1}||\bar{a}_{j_2}|$.

6.4 Experiments

In this section we experimentally evaluate our methodology. First, on a set of generated data we show that both DP and Greedy algorithms give results of high quality. Next, we show that how segmentation aggregation can prove useful in analysis of genomic sequences and particularly to the problem of haplotype blocks. In Section 6.4.3 we demonstrate how segmentation aggregation provides a robust framework for segmenting timeseries data that have erroneous or missing values. Finally, in Section 6.4.4 we analyze mobile phone users' data via the segmentation aggregation framework.

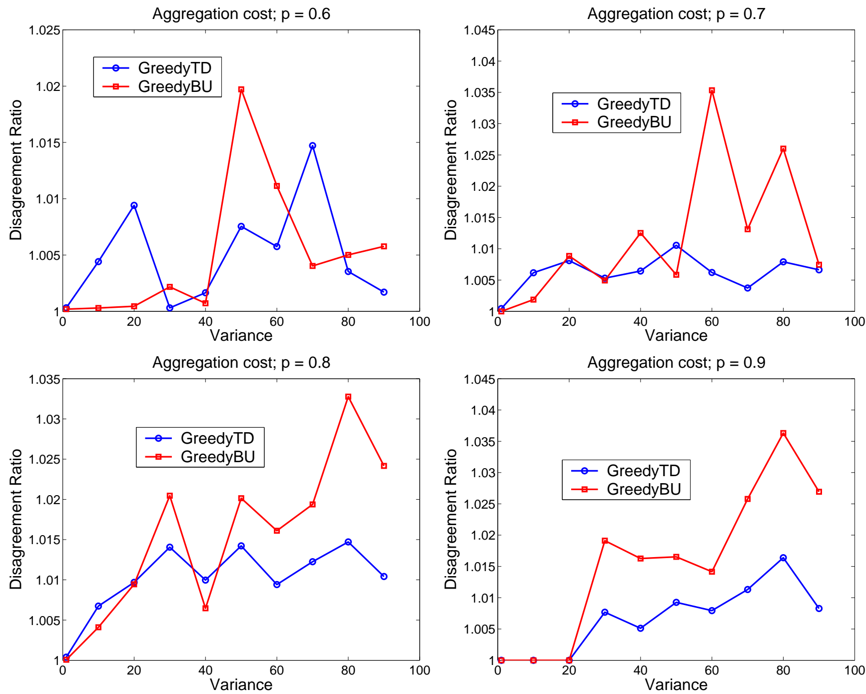


Figure 6.3: Synthetic datasets: disagreement ratio of GREEDY heuristics as a function of the variance used for data generation. The probability p of altering a single segment boundary is fixed in every experiment. The disagreement ratio is obtained by dividing the disagreement cost of the GREEDY algorithms with the disagreement cost of the optimal aggregation.

6.4.1 Comparing aggregation algorithms

For comparing aggregation algorithms we generate segmentation datasets as follows: first we create a random segmentation of a sequence of length 1000 by picking a random set of boundary points. We call the segmentation obtained in this way the *basis segmentation*. We use this basis segmentation as a template to create a dataset of 100 segmentations to be aggregated. Each such output segmentation is generated from the basis as follows: each segment boundary of the basis is kept identical in the output with probability $(1 - p)$, while it is altered with probability p. There are two types of changes a boundary is subject to: *deletion* and *translocation*. In the case of translocation, the new location of the boundary is determined by the *variance level*. For small variance levels the boundary is placed close to its old location, while for large values of v the boundary is placed further.

Figure 6.3 shows the ratio of the aggregation costs achieved by GreedyTD and GreedyBU with respect to the optimal DP algorithm. It is apparent that the greedy alternatives give in most of the cases results with cost very close (almost identical) to the optimal. We mainly show the results for $p > 0.5$, since for smaller values of p the ratio is always equal to 1. These results verify that not only the quality of the aggregation found by the greedy algorithms is close to this of the optimal, but also that the *structure* of the algorithms' outputs are very similar. All the results are averages over 10 independent runs.

6.4.2 Experiments with haplotype data

The basic intuition of the haplotype block problem as well as its significance in biological sciences and medicine have already been discussed in Section 6.1. Here we show how the segmentation aggregation methodology can be applied in this setting. The main problem with the haplotype block structure is that although numerous studies have confirmed its existence, the methodologies that have been proposed for finding the blocks leave multiple uncertainties, related to the number and the exact positions of their boundaries.

The main line of work related to haplotype block discovery consists of a series of segmentation algorithms. These algorithms usu-

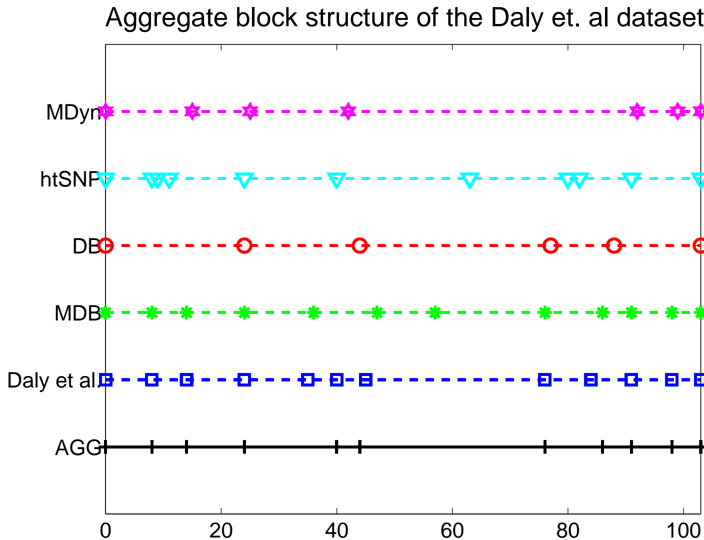


Figure 6.4: Block structure of haplotype data. AGG corresponds to the aggregate segmentation obtained by aggregating the block structures output by MDyn, htSPN, DB, MDB and Daly et. al. methods.

ally assume different optimization criteria for block quality and segment the data so that blocks of good quality are produced. Although one can argue for or against each one of these optimization functions, we again adopt the aggregation approach. That is, we aggregate the results of the different algorithms used for discovering haplotype blocks by doing segmentation aggregation.

For the experiments we use the published dataset of [DRS+01] and we aggregate the segmentations produced by the following five different methods:

1. *Daly et al.*: This is the original algorithm for finding blocks used in [DRS+01].

2. *htSNP*: This is a dynamic programming algorithm proposed in [ZCC+02]. The objective function uses the htSNP criterion proposed in [PBH+01].

3. *DB*: This is again a dynamic programming algorithm, though for a different optimization criterion. The algorithm is pro-

posed in [ZCC$^+$02], while the optimization measure is the *haplotype diversity* proposed by [JKB97].

4. *MDB*: This is a Minimum Description Length (MDL) method proposed in [AN03].

5. *MDyn*: This is another MDL-based method proposed by [KPV$^+$03].

Figure 6.4 shows the block boundaries found by each one of the methods. The solid line corresponds the block boundaries found by doing segmentation aggregation on the results of the aforementioned five methods. The aggregate segmentation has 11 segment boundaries, while the input segmentations have $12, 11, 6, 12$ and 7 segment boundaries respectively, with 29 of them being unique. Notice that in the result of the aggregation, block boundaries that are very close to each other in some segmentation methods (for example htSNP) disappear and in most of the cases they are replaced by a single boundary. Additionally, the algorithm does not always find boundaries that are in the majority of the input segmentations. For example, the eighth boundary of the aggregation appears only in two input segmentations, namely the results of Daly et al. and htSNP.

6.4.3 Robustness experiments

In this experiment we demonstrate the usefulness of the segmentation aggregation in producing robust segmentation results, insensitive to the existence of outliers in the data. Consider the following scenario, where multiple sensors are sending their measurements to a central server. It can be the case that some of the sensors may fail at certain points in time. For example, they may run out of battery or report erroneous values due to communication delays in the network. Such a scenario causes outliers (missing or erroneous data) to appear. The classical segmentation algorithms are sensitive to such values and usually produce "unintuitive" results. We here show that the segmentation aggregation is insensitive to the existence of missing or erroneous data via the following experiment: first, we generate a multidimensional sequence of real numbers that has an a priori known segmental structure. We fix the number of segments appearing in the data to be $k = 10$, while all the dimensions have

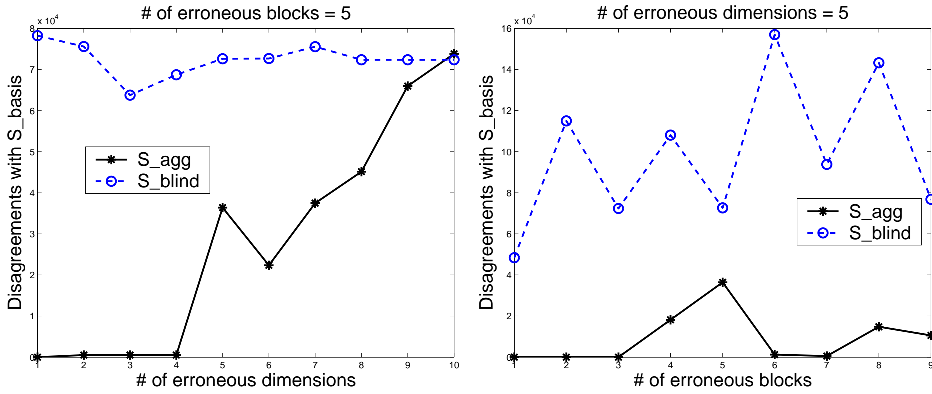


Figure 6.5: Disagreements of S_{agg} and S_{blind} with the true underlying segmentation S_{basis} as a function of the number of erroneous dimensions.

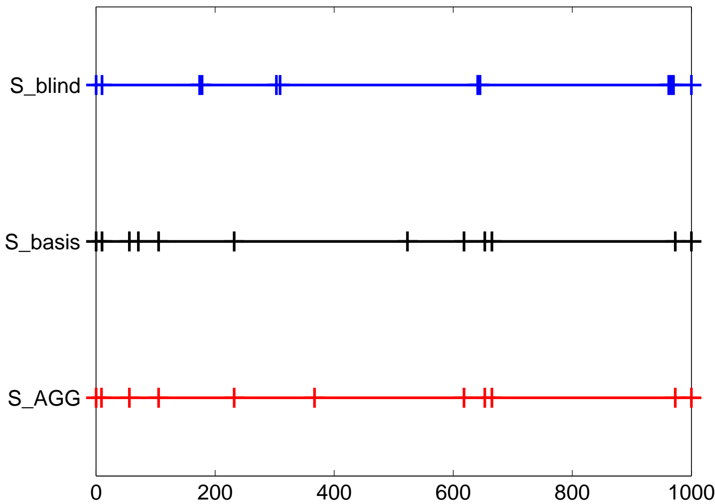


Figure 6.6: Anecdote illustrative of the insensitivity of the aggregation, S_AGG, to the existence of outliers in the data. The aggregate segmentation S_AGG is much more similar to the desired segmentation, S_basis, than the output of the "best" segmentation algorithm, S_blind.

the same segment boundaries. All the points in a segment are normally distributed around some randomly picked mean $\mu \in [9, 11]$. One can consider each dimension to correspond to data coming from a different sensor. We report the results from a dataset that has 1000 data points, and 10 dimensions.

Standard segmentation methods segment all dimensions together. We do the same using the variance of the segments to measure the quality of the segmentation. We segment all the dimensions together using the standard optimal dynamic programming algorithm for sequence segmentation [Bel61]. Let us denote by S_{basis} the segmentation of this data obtained by this dynamic programming algorithm.

We simulate the erroneous data as follows: first we pick a specific subset of dimensions on which we insert erroneous blocks of data. The cardinality of the subset varies from 1 to 10 (all dimensions). An erroneous block is a set of consecutive outlier values. Outlier values are represented by 0s in this example. We use small blocks of length at most 4 and we insert $1 - 10$ such blocks. This means that in the worst case we have at most 4% faulty data points. A standard segmentation algorithm would produce a segmentation of this data by blindly segmenting all dimensions together. Let us denote by S_{blind} the segmentation produced by the dynamic programming segmentation algorithm in this modified dataset. Alternatively, we segment each dimension separately in $k = 10$ segments and then we aggregate the results. We denote by S_{agg} the resulting aggregate segmentation.

Figure 6.5 reports the disagreements $D_A(S_{\text{agg}}, S_{\text{basis}})$ and $D_A(S_{\text{blind}}, S_{\text{basis}})$ obtained when we fix the number of erroneous blocks inserted in each dimension and vary the number of dimensions that are faulty, and vice versa. That is, we try to compare the number of disagreements between the segmentations produced by aggregation and by blindly segmenting all the dimensions together, to the segmentation that would have been obtained if the erroneous data were ignored. Our claim is that a "correct" segmentation should be as close as possible to S_{basis}. Figure 6.5 indeed demonstrates that the aggregation result is much closer to the underlying true segmentation, and thus the aggregation algorithm is less sensitive to the existence of outliers. Figure 6.6 further verifies this intuition by visualizing the segmentations S_{basis}, S_{agg} and S_{blind} for the case of

5 erroneous dimensions containing 5 blocks of consecutive outliers.

6.4.4 Experiments with reality-mining data

The reality-mining dataset [2]contains usage information of about a 97 mobile phone users. Large percentage of these users are either students, or faculty of the MIT Media Laboratory, while the rest are incoming students at the MIT Sloan Business School, located adjacent to the laboratory. The collected information includes call logs, Bluetooth devices in proximity, cell tower IDs, application usage, and phone status (such as charging and idle) etc. The data spans a period from September 2004 to May 2005. We mainly focus our analysis on the data related to the *callspan* of each user. The callspan data has information related to the actual times each user places a phonecall.

From this data we produce segmentation-looking inputs as follows: for each user, and each day during which he has been logged, we take the starting times reported in the callspan and we consider them as segment boundaries on the timeline of the day. Therefore, a user recorded for say 30 days is expected to have 30 different such segmentations associated with him.

Identifying single user's patterns

In our first experiment, we cluster the days of a single user. Since each day is represented as a segmentation of the 24-hour timeline, clustering the days corresponds to clustering these segmentations. We use distance D_A for comparing the different days. The definition of segmentation aggregation allows naturally to define the "mean" of a cluster to be the aggregation of the segmentations that are grouped together in the cluster. With this tool, we can extend classical k-means of Euclidean spaces to the space of segmentations.

Figure 6.7 shows the clustering of the days of a single user (who is classified as a professor in the dataset) over a period of 213 days starting from September 2004 to May 5th 2005 (not all days are recorded). The plot on the top shows the clustering of the days. The days are arranged sequentially and the different colors correspond

[2]The interested reader can find the datasets at http://reality.media.mit.edu/

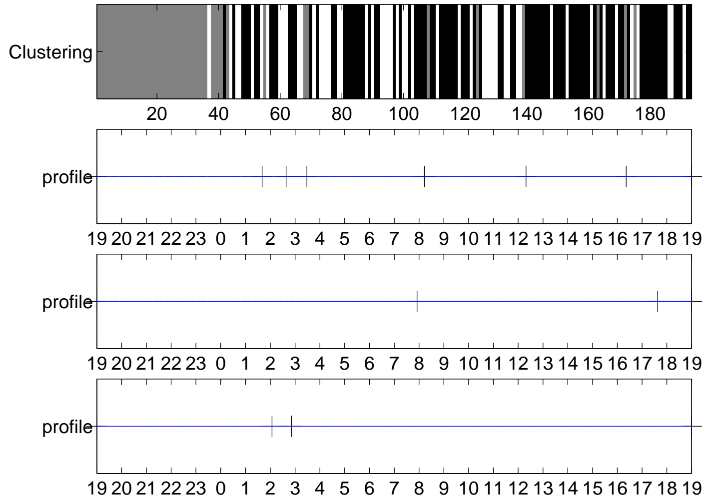


Figure 6.7: Clustering of a single user's logged days into three clusters and the corresponding cluster representatives.

to different clusters. It is apparent that at the beginning of the recorded period the patterns of the user are quite different from the patterns observed at later points in the study. More specifically, all the initial days form a single rather homogeneous cluster.

During the period corresponding to the days of this cluster the Media Lab subjects had been working towards the annual visit of the laboratory's sponsors [Eag05]. It had been previously observed that this had affected the subjects' schedules. We can thus conclude that our methodology captures this pattern as well. The rest of Figure 6.7 shows the representatives of each cluster. We observe that the representatives are rather distinct consisting of profiles where the users uses his phone either in morning hours, or in evening hours or both.

Finding groups of similar users

In the second experiment we try to build clusters of users that show similar patterns in their activities. For this, we build the profile of each user, by aggregating all the days he has been logged for. Next, we cluster the user profiles, using the k-means algorithm for

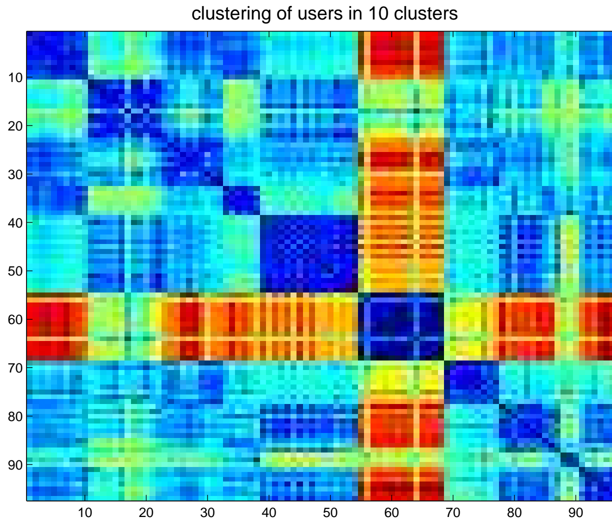


Figure 6.8: The clustering structure of the reality-mining user data.

segmentations, as discussed in the previous paragraph. Figure 6.8 gives visual evidence of the existence of some clusters of users in the dataset. The plot shows the distances between user profiles, in terms of disagreement distance. The rows and the columns of the distance matrix have been rearranged so that users clustered together are put in consecutive rows (columns). The darker the coloring of a cell at position (i, j) the more similar users i and j are. There are some evident clusters in the dataset, like for example the one consisting of users at positions $1-10$, $33-38$, $39-54$, $55-68$ and $69-77$. Notice, that the cluster containing users $55-68$ is not only characterized by strong similarity among its members but additionally the members of this cluster are very dissimilar to almost every other user in the dataset.

From those groups, the third one, consisting of rows $39-54$, seems to be very coherent. We further looked at the people constituting this group and found out that most of them are related (being probably students) to the Sloan Business School. More specifically, the positions of the people in the cluster, as reported in the dataset, appear to be

```
sloan, mlUrop, sloan, sloan, sloan, 1styeargrad,
sloan, sloan, sloan, 1styeargrad, mlgrad,
```

`sloan, sloan, sloan, staff, sloan.`

Similarly, the relatively large and homogeneous group formed by lines $1 - 10$ consists mostly from staff and professors.

Another interesting group of users is the one consisting from users $55 - 68$. Those users are not only very similar within themselves but they are also are very dissimilar to almost every other user in the dataset. This group though contains a rather diverse set of people, at least with respect to their positions. However there may be another link that makes their phone usage patterns similar.

6.5 Alternative distance functions: Entropy distance

In this section we introduce the *entropy distance* as an alternative measure for comparing segmentations.

6.5.1 The entropy distance

The *entropy distance* between two segmentations quantifies the information one segmentation reveals about the other. In general, the entropy distance between two random variables X and Y that take values in domains $\mathcal{X}$ and $\mathcal{Y}$, respectively, is defined as

$$D_H(X,Y) = H(X|Y) + H(Y|X),$$

where $H(\cdot|\cdot)$ is the conditional entropy function and

$$H(X|Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} \Pr(x,y) \log \Pr(x|y). \tag{6.10}$$

For segmentations this can be interpreted as follows. Consider a segmentation P of timeline T with segments $\{\bar{p}_1, \ldots, \bar{p}_{\ell_p}\}$, on a random experiment that picks a random segment $\bar{x} \in \{\bar{p}_1, \ldots, \bar{p}_{\ell_p}\}$. Then the probability that the i-th segment was picked is

$$\Pr(\bar{x} = \bar{p}_i) = \frac{|\bar{p}_i|}{|T|}.$$

In this way, every segmentation defines a sample space for a random variable and therefore the entropy of segmentation P, with segments is defined to be

$$H(P) = -\sum_{i=1}^{\ell_p} \Pr(\bar{x} = \bar{p}_i) \log \Pr(\bar{x} = \bar{p}_i).$$

Note that by definition of probability it also holds that for every segmentation P with ℓ_p segments, $\sum_{i=1}^{\ell_p} \Pr(\bar{x} = \bar{p}_i) = 1$. Furthermore, the entropy of a segmentation corresponds in some sense to the potential energy of the segmentation for the entropy distance function.

Now consider a pair of segmentations P and Q with segments $\{\bar{p}_1, \ldots, \bar{p}_{\ell_p}\}$ and $\{\bar{q}_1, \ldots, \bar{q}_{\ell_q}\}$, and define the conditional entropy of the one segmentation given the other. We associate with segmentation P random variable $\bar{x}$ that takes values in $\{\bar{p}_1, \ldots, \bar{p}_{\ell_p}\}$ and with segmentation Q random variable $\bar{y}$ that takes values in $\{\bar{q}_1, \ldots, \bar{q}_{\ell_q}\}$ The conditional entropy of segmentation P given segmentation Q can be computed as in Equation (6.10), that is,

$$H(P|Q) = \\ -\sum_{i=1}^{\ell_p} \sum_{j=1}^{\ell_q} \Pr\left(\bar{x} = \bar{p}_i, \bar{y} = \bar{q}_j\right) \log\left(\Pr\left(\bar{x} = \bar{p}_i | \bar{y} = \bar{q}_j\right)\right). \tag{6.11}$$

We can now define the entropy distance between two segmentations P and Q.

Definition 6.3 *Let P and Q be two segmentations of timeline T with segments $\{\bar{p}_1, \ldots, \bar{p}_{\ell_p}\}$ and $\{\bar{q}_1, \ldots, \bar{q}_{\ell_q}\}$ respectively. The* entropy distance*, D_H, between P and Q is defined to be*

$$D_H(P, Q) = H(P|Q) + H(Q|P), \tag{6.12}$$

where $H(P|Q)$ and $H(Q|P)$ are evaluated by Equation (6.11).

As the following lemma shows the entropy distance is also a metric and therefore we can again take advantage of this property for applications of D_H in clustering.

Lemma 6.2 *The entropy distance D_H is a metric.*

Proof. By simple observation and due to the fact that the entropy function is positive we can see that for any two segmentations P and Q it holds that $D_H(P,Q) \geq 0$ and $D_H(P,Q) = D_H(Q,P)$. We now show that D_H also satisfies the triangular inequality. That is, for three segmentations P, Q and S it holds that

$$D_H(P,Q) + D_H(Q,S) \geq D_H(P,S),$$

or alternatively,

$$H(P|Q) + H(Q|P) + H(Q|S) + H(S|Q) \geq H(P|S) + H(S|P).$$

Due to the symmetry of the above equation, it is enough to prove that

$$H(P|Q) + H(Q|S) \geq H(P|S).$$

This can be proved easily as follows

$$\begin{aligned} H(P|Q) + H(Q|S) &\geq H(P|Q,S) + H(Q|S) \\ &= H(P,Q|S) \\ &= H(Q|P,S) + H(P|S) \\ &\geq H(P|S). \end{aligned}$$

□

6.5.2 Computing the entropy distance

For two segmentations P and Q with ℓ_p and ℓ_q number of segments respectively, the distance $D_H(P,Q)$ can be computed trivially in time $O\left((\ell_p + \ell_q)^2\right)$. Next we show that this can be done faster in time $O\left(\ell_p + \ell_q\right)$. For showing this it is enough to establish the following lemma.

Lemma 6.3 *Let P and Q be two segmentations and U be their union segmentation. The distance $D_H(P,Q)$ can be computed by the following closed formula*

$$D_H(P,Q) \quad = \quad 2H(U) - H(P) - H(Q).$$

Proof. Assume that segmentation P has ℓ_p segments $\{\bar{p}_1,\dots \bar{p}_{\ell_p}\}$ and segmentation Q has ℓ_q segments $\{\bar{q}_1,\dots,\bar{q}_{\ell_q}\}$. We again associate with segmentation P random variable $\bar{x}$ that takes values in $\{\bar{p}_1,\dots \bar{p}_{\ell_p}\}$ and with segmentation Q random variable $\bar{y}$ that takes values in $\{\bar{q}_1,\dots,\bar{q}_{\ell_q}\}$. From Equation (6.12) we know that

$$D_H(P,Q) \quad = \quad H(P|Q) + H(Q|P),$$

and by Equation (6.11) we have that

$$\begin{aligned}
H(P|Q) &= -\sum_{i=1}^{\ell_p}\sum_{j=1}^{\ell_q} \Pr(\bar{x}=\bar{p}_i,\bar{y}=\bar{q}_j)\log\left(\Pr(\bar{x}=\bar{p}_i|\bar{y}=\bar{q}_j)\right) \\
&= -\sum_{i=1}^{\ell_p}\sum_{j=1}^{\ell_q} \Pr(\bar{x}=\bar{p}_i,\bar{y}=\bar{q}_j)\log\left(\Pr(\bar{x}=\bar{p}_i,\bar{y}=\bar{q}_j)\right) \\
&\quad +\sum_{i=1}^{\ell_p}\sum_{j=1}^{\ell_q} \Pr(\bar{x}=\bar{p}_i,\bar{y}=\bar{q}_j)\log\left(\Pr(\bar{y}=\bar{q}_j)\right) \\
&= H(U) + \sum_{j=1}^{\ell_q}\log\left(\Pr(\bar{y}=\bar{q}_j)\right)\sum_{i=1}^{\ell_p}\Pr(\bar{x}=\bar{p}_i,\bar{y}=\bar{q}_j) \\
&= H(U) + \sum_{j=1}^{\ell_q}\log\left(\Pr(\bar{y}=\bar{q}_j)\right)\Pr(\bar{y}=\bar{q}_j) \\
&= H(U) - H(Q).
\end{aligned}$$

Similarly, we can show that $H(Q|P) = H(U) - H(P)$. This proves the lemma. □

Lemma 6.3 allows us to state the following theorem.

Theorem 6.4 *Given two segmentations P and Q with ℓ_p and ℓ_q segments respectively, $D_H(P,Q)$ can be computed in time $O(\ell_p+\ell_q)$.*

6.5.3 Restricting the candidate boundaries for D_H

Theorem 6.2 says that in the `Segmentation Aggregation` problem with D_A distance function the boundaries of the optimal aggregation are restricted to the already existing boundaries in the input segmentations. A similar theorem can be proved for the entropy distance function D_H.

Theorem 6.5 *Let $S_1, S_2 \dots S_m$ be the m input segmentations to the* `Segmentation Aggregation` *problem for D_H distance and let U be their union segmentation. For the optimal aggregate segmentation $\hat{S}$, it holds that $\hat{S} \subseteq U$, that is, all the segment boundaries in $\hat{S}$ belong in U.*

Proof. The proof is along the same lines as the proof of Theorem 6.2. Assume that the optimal aggregate segmentation $\hat{S}$ has cost $C(\hat{S}) = \sum_{i=1}^{m} D_H(\hat{S}, S_i)$ and that $\hat{S}$ contains segment boundary $j \in T$ such that $j \notin U$. For the proof we will assume that we have the freedom to move boundary j to a new position x_j. We will show that this movement will reduce the cost of the aggregate segmentation $\hat{S}$, which will contradict the optimality assumption.

We first consider a single input segmentation $S \in \{S_1, \dots, S_m\}$ and denote its union with the aggregate segmentation $\hat{S}$ by U_S. Assume that we move boundary j to position x_j in segmentation $\hat{S}$. However, this movement is restricted to be within the smallest interval in U_S that contains j. (As before similar arguments can be made for all segments in U_S.) Consider the boundary point arrangement shown in Figure 6.2. In this arrangement we denote by $\hat{P}$ (P_U) the first boundary of $\hat{S}$ (U_S) that is to the left of x_j and by $\hat{N}$ (N_U) the first boundary of $\hat{S}$ (U_S) that is to the right of x_j. We know by Lemma 6.3 that

$$D_H(S, \hat{S}) = 2H(U_S) - H(S) - H(\hat{S}),$$

or simply

$$D_H = 2H_{U_S} - H_S - H_{\hat{S}}.$$

Note that $H_{\hat{S}}$ and H_{U_S} both depend on the position of x_j, while H_S does not since $x_j \notin S$. Thus, by writing D_H as a function of x_j we have that

$$D_H(x_j) = 2H_{U_S}(x_j) - H_{\hat{S}}(x_j) - H_S, \tag{6.13}$$

where

$$\begin{aligned} H_{\hat{S}}(x_j) = \; & c_{\hat{S}} - \frac{(x_j - \hat{P})}{n} \log\left(\frac{(x_j - \hat{P})}{n}\right) \\ & - \frac{(\hat{N} - x_j)}{n} \log\left(\frac{(\hat{N} - x_j)}{n}\right) \end{aligned}$$

and

$$
\begin{aligned}
H_{U_S}(x_j) &= c_U - \frac{(x_j - P_U)}{n}\log\left(\frac{(x_j - P_U)}{n}\right) \\
&\quad + \frac{(N_U - x_j)}{n}\log\left(\frac{(N_U - x_j)}{n}\right).
\end{aligned}
$$

In the above two equations $c_{\hat{S}}$ and c_U are terms that are independent of x_j. If we substitute the latter two equations into Equation (6.13) and we take the first derivative of this with respect to x_j we have that

$$
\begin{aligned}
\frac{dD_H(x_j)}{dx_j} &= \frac{1}{n}\big[1 + \log(x_j - \hat{P}) - 1 - \log(\hat{N} - x_j) \\
&\quad -2\log(x_j - P_U) + 2\log(N_U - x_j)\big] + 0.
\end{aligned}
$$

Taking the second derivative we get that

$$
\begin{aligned}
\frac{d^2 D_H(x_j)}{dx_j^2} &= \frac{1}{n}\left[\frac{1}{x_j - \hat{P}} + \frac{1}{\hat{N} - x_j} - 2\frac{2}{x_j - P_U} - 2\frac{1}{N_U - x_j}\right] \\
&\leq -\frac{1}{x_j - P_U} - \frac{1}{N_U - x_j} \leq 0.
\end{aligned}
$$

The last inequality holds because $x_j - \hat{P} \geq x - P_U$ which means that $\frac{1}{x_j - \hat{P}} \leq \frac{1}{x_j - P_U}$. Similarly, we have that $\hat{N} - x_j \geq N_U - x_j$ which means that $\frac{1}{\hat{N} - x_j} \leq \frac{1}{N_U - x_j}$.

The second derivative being negative implies that the function is convex in the interval $[P_U, N_U]$ and therefore it will exhibit its local minima in the interval's endpoints. That is, $x_j \in \{P_U, N_U\}$ which contradicts the initial optimality assumption. Note that the above argument is true for all input segmentations in the particular interval and therefore it is also true for their sum. □

6.5.4 Finding the optimal aggregation for D_H

In this section we show that the `Segmentation Aggregation` problem for distance function D_H between segmentations can also be solved optimally in polynomial time using a dynamic programming algorithm. The notational conventions used in this section are identical to the ones we used in Section 6.3.2. In fact, Recursion (6.6)

solves the `Segmentation Aggregation` problem for the D_H distance function as well. The proof that the recursion evaluates the aggregation with the minimum cost is very similar to the one given for Theorem 6.3 and thus omitted. However, we give some details of how the dynamic programming will be implemented for D_H.

Consider a k-restricted segmentation A^k with boundaries $\{u_0, \ldots, u_k, n\}$ and segments $\{\bar{a}_1, \ldots, \bar{a}_{k+1}\}$. Assume that we extend A^k to $A^j = A^k \cup \{u_j\}$ by adding boundary $u_j \in U$. We focus on a single input segmentation S_i and we denote by U_i^k the union of segmentation S_i with the k-restricted aggregation A^k. The calculation of the impact of point u_j on the other input segmentations is similar. By Lemma 6.3 we have that the impact of u_j is

$$\begin{aligned} I_i(A^k, u_j) &= C_i(A^k \cup \{u_j\}) - C_i(A^k) \\ &= 2H(U_i^j) - H(A^j) - H(S_i) \\ &\quad -2H(U_i^k) + H(A^k) + H(S_i). \end{aligned} \tag{6.14}$$

Consider now the addition of boundary $u_j > u_k$. This addition splits the last segment of A^k into two subsegments $\bar{\beta}_1$ and $\bar{\beta}_2$ such that $|\bar{\beta}_1|+|\bar{\beta}_2| = |\bar{a}_{k+1}|$. Now assume that $u_j \in S_i$. This means that the addition of u_j in the aggregation does not change segmentation U_i^k. That is, $U_i^k = U_i^j$. Substituting into Equation (6.14) we have that

$$\begin{aligned} I_i(A^k, u_j) &= -H(A^j) + H(A^k) \\ &= -\frac{|\bar{a}_{k+1}|}{n} \log\left(\frac{|\bar{a}_{k+1}|}{n}\right) \\ &\quad +\frac{|\bar{\beta}_1|}{n} \log\left(\frac{|\bar{\beta}_1|}{n}\right) + \frac{|\bar{\beta}_2|}{n} \log\left(\frac{|\bar{\beta}_2|}{n}\right). \end{aligned}$$

In the case where $u_j \notin S_i$, the addition of u_j in the aggregation causes a change in segmentation U_i^k. Let segment $\bar{u}$ of U_i^k be split into segments $\bar{\gamma}_1$ and $\bar{\gamma}_2$ in segmentation U_i^j. The split is such that $|\bar{u}| = |\bar{\gamma}_1| + |\bar{\gamma}_2|$ This would mean that the impact of boundary u_j now becomes

$$\begin{aligned} I_i(A^k, u_j) &= 2H(U_i^j) - H(A^j) - 2H(U_i^k) + H(A^k) \\ &= -\frac{|\bar{a}_{k+1}|}{n} \log\left(\frac{|\bar{a}_{k+1}|}{n}\right) \\ &\quad + \frac{|\bar{\beta}_1|}{n} \log\left(\frac{|\bar{\beta}_1|}{n}\right) + \frac{|\bar{\beta}_2|}{n} \log\left(\frac{|\bar{\beta}_2|}{n}\right) \\ &\quad - 2\frac{|\bar{\gamma}_1|}{n} \log\left(\frac{|\bar{\gamma}_1|}{n}\right) - 2\frac{|\bar{\gamma}_2|}{n} \log\left(\frac{|\bar{\gamma}_2|}{n}\right) \\ &\quad + \frac{|\bar{u}|}{n} \log\left(\frac{|\bar{u}|}{n}\right). \end{aligned}$$

Note that computing the impact of every point can be done in $O(m)$ time and therefore the total computation needed for the evaluation of the dynamic programming recursion is $O(N^2 m)$.

6.6 Alternative distance function: Boundary mover's distance

The *Boundary Mover's Distance* (D_B)[3] compares two segmentations P and Q considering only the distances between their boundary points. Let the boundary points of P and Q be $\{p_0, \ldots, p_k\}$ and $\{q_0, \ldots, q_\ell\}$. We define the Boundary Mover's distance between of P with respect to Q to be

$$D_B(P \mid Q) = \sum_{p_i \in P} \min_{q_j \in Q} |p_i - p_j|^p .$$

Two natural choices for p are $p = 1$ and $p = 2$. For $p = 1$ the Boundary Mover's distance is the Manhattan distance between the segment boundaries, while for $p = 2$ it is the sum-of-squares distance.

The `Segmentation Aggregation` problem for distance D_B with m input segmentations $S_1, \ldots, S_m$ asks for an aggregate segmentation $\hat{S}$ of at most t boundaries such that

$$\hat{S} = \arg\min_{S \in \mathcal{S}} D_B(S_i \mid S).$$

[3] The name is a variation of the known Earth Mover's Distance [RTG00]

Notice that in this alternative definition of the `Segmentation Aggregation` problem we have to restrict the number of boundaries that can appear in the aggregation. Otherwise, the optimal $\hat{S}$ will contain the union of boundaries that appear in P and Q - such a segmentation will have total cost equal to 0. One can easily see that this alternative definition of the segmentation aggregation problem can also be solved optimally in polynomial time. More specifically, the problem of finding the best aggregation with at most t segment boundaries can be reduced to one-dimensional clustering that can be solved using dynamic programming. For the mapping, consider that the boundaries of the input segmentations to be the points to be clustered, and the boundaries of the aggregation to be the cluster representatives. We also note that for $p = 1$ all the boundaries of the aggregate segmentation appear in the union segmentation too.

6.7 Conclusions

We have presented a novel approach to sequence segmentation, that is based on the idea of aggregating existing segmentations. The utility of segmentation aggregation has been extensively discussed via a set of useful potential applications. We have formally defined the segmentation aggregation problem and showed some of its interesting properties. From the algorithmic point of view, we showed that we can solve it optimally in polynomial time using dynamic programming. Furthermore, we designed and experimented with greedy algorithms for the problem, which in principle are not exact, but in practice they are both fast and give results of high quality (almost as good as the optimal). The practical utility of the problem and the proposed algorithms has been illustrated via a broad experimental evaluation that includes applications of the framework on genomic sequences and users' mobile phone data. We additionally demonstrated that segmentation aggregation is a noise and error-insensitive segmentation method that can be used to provide robust segmentation results.

CHAPTER 7

Discussion

In this thesis we presented a set of problems related to sequence segmentations and discussed some algorithmic techniques for dealing with them. Initially, we focused on the segmentation problem in its simplest formulation. However, in the subsequent chapters we presented different segmentation models that proved useful in modeling real-life datasets.

More specifically, the contributions of the thesis include the fast approximate algorithm for the basic segmentation problem that was presented in Chapter 3. The algorithm's simplicity makes it attractive to use in practice. The open question that remains of interest is whether techniques similar to the DnS framework can be employed to accelerate other standard dynamic programming algorithms.

Chapters 4 and 5 introduced new variants of the basic segmentation problem. Experimental evidence showed that these new segmentation models are useful for real datasets. The common characteristic of these models is that they are rather simple and intuitive. The existence of approximation algorithms for finding clustered segmentations is still an open problem. We have proved that `Segmentation with Swaps` is not only NP-hard but also hard to approximate. The question of whether this result carries over to `Segmentation with Moves` remains open.

Aggregating results of data mining algorithms has attracted lots of attention in recent years. A natural problem when dealing with sequence segmentation algorithms is the one of aggregating their results. The aggregation should be made in such a way that the strengths of the different segmentation algorithms are exploited,

while their weaknesses are suppressed. In Chapter 6 we introduced the `Segmentation Aggregation` problem and showed that it is solvable in polynomial time for three different definitions of distance functions between segmentations. An interesting future direction of this work is towards defining a class of distance functions, such that the results presented in Chapter 6 carry over to any function in this class.

References

[ACN05] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: ranking and clustering. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, pages 684–693, 2005.

[AGK+01] Vijay Arya, Naveen Garg, Rohit Khandekar, Kamesh Munagala, and Vinayaka Pandit. Local search heuristic for k-median and facility location problems. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, pages 21–29, 2001.

[ALSS95] Rakesh Agrawal, King-Ip Lin, Harpreet S. Sawhney, and Kyuseok Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proceedings of Very Large Data Bases (VLDB) Conference*, pages 490–501, 1995.

[AN03] Eric C. Anderson and John Novembre. Finding haplotype block boundaries by using the minimum-description-length principle. *American Journal of Human Genetics*, 73, 2003.

[ARLR02] Rajeev K. Azad, J. Subba Rao, Wentian Li, and Ramakrishna Ramaswamy. Simplifying the mosaic description of DNA sequences. *Physical Review E, 66, article 031913*, 2002.

[ARR98] Sanjeev Arora, Prabhakar Raghavan, and Satish Rao. Approximation schemes for Euclidean medians and related problems. In *Proceedings of the Symposium*

on the Theory of Computing (STOC), pages 106–113, 1998.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 3–14, 1995.

[BDGM97] Béla Bollobás, Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. Time-series similarity problems and well-separated geometric sets. In *Symposium on Computational Geometry*, pages 454–456, 1997.

[Bel61] Richard Bellman. On the approximation of curves by line segments using dynamic programming. *Communications of the ACM*, 4(6), 1961.

[BGGC+00] P. Bernaola-Galván, I. Grosse, P. Carpena, J. Oliver, R. Román-Roldán, and H. Stanley. Finding borders between coding and non-coding regions by an entropic segmentation method. *Physical Review Letters*, 85(6):1342–1345, 2000.

[BGH+06] Ella Bingham, Aristides Gionis, Niina Haiminen, Heli Hiisilä, Heikki Mannila, and Evimaria Terzi. Segmentation and dimensionality reduction. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2006.

[BGRO96] P. Bernaola-Galvan, R. Roman-Roldan R, and J.L. Oliver. Compositional segmentation and long-range fractal correlations in dna sequences. *Phys. Rev. E. Stat. Phys. Plasmas Fluids Relat. Interdiscip. Topics*, 53(5):5181–5189, 1996.

[BLS00] Harmen J. Bussemaker, Hao Li, and Eric D. Siggia. Regulatory element detection using a probabilistic segmentation model. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology*, pages 67–74, 2000.

[Bre96] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[CKMN01] Moses Charikar, Samir Khuller, David M. Mount, and Giri Narasimhan. Algorithms for facility location problems with outliers. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 642–651, 2001.

[CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[COP03] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, pages 30–39, 2003.

[CSD98] Soumen Chakrabarti, Sunita Sarawagi, and Byron Dom. Mining surprising patterns using temporal description length. In *Proceedings of Very Large Data Bases (VLDB) Conference*, pages 606–617, 1998.

[CSS02] Michael Collins, Robert E. Schapire, and Yoram Singer. Logistic regression, adaboost and Bregman distances. *Machine Learning*, 48(1-3):253–285, 2002.

[DKNS01] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of International World Wide Web Conferences (WWW)*, pages 613–622, 2001.

[DP73] D.H. Douglas and T.K. 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.

[DRS$^+$01] Mark J. Daly, John D. Rioux, Stephen F. Schaffner, Thomas J. Hudson, and Eric S. Lander. High-resolution haplotype structure in the human genome. *Nature Genetics*, 29:229–232, 2001.

[Eag05] Nathan Eagle. Machine perception and learning of complex social systems. *PhD Thesis, Program in Media Arts and Sciences, Massachusetts Institute of Technology*, 2005.

[FISS03] Yav Freund, Raj Iyer, Rober E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, 4:933–969, 2003.

[FJ05] Ana L.N. Fred and Anil K. Jain. Combining multiple clusterings unsing evidence accumulation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(6):835–850, 2005.

[FKM+04] Ronald Fagin, Ravi Kumar, Mohammad Mahdian, D. Sivakumar, and Erik Vee. Comparing and aggregating rankings with ties. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 47–58, 2004.

[Fri80] M. Frisén. Unimodal regression. *The Statistician*, 35:304–307, 1980.

[FRM94] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time series databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 419–429, 1994.

[GAS05] Robert Gwadera, Mikhail J. Atallah, and Wojciech Szpankowski. Reliable detection of episodes in event sequences. *Knowledge and Information Systems*, 7(4):415–437, 2005.

[GJ79] M.R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[GKS01] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, pages 471–475, 2001.

[GM03] Aristides Gionis and Heikki Mannila. Finding recurrent sources in sequences. In *International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 123–130, 2003.

[GMT04] Aristides Gionis, Heikki Mannila, and Evimaria Terzi. Clustered segmentations. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD). Workshop on Mining Temporal and Sequential Data (TDM)*, 2004.

[GMT05] Aristides Gionis, Heikki Mannila, and Panayiotis Tsaparas. Clustering aggregation. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 341–352, 2005.

[Gus02] Dan Gusfield. Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In *International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 166–175, 2002.

[Gus03] Dan Gusfield. An overview of haplotyping via perfect phylogeny: Theory, algorithms and programs. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, 2003.

[HG04] Niina Haiminen and Aristides Gionis. Unimodal segmentation of sequences. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 106–113, 2004.

[HKM+01] Johan Himberg, Kalle Korpiaho, Heikki Mannila, Johanna Tikanmäki, and Hannu 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.

[HS05] Tzvika Hartman and Roded Sharan. A 1.5-approximation algorithm for sorting by transpositions and transreversals. *Journal of Computer and System Sciences*, 70(3):300–320, 2005.

[Ind99] Piotr Indyk. A sublinear time approximation scheme for clustering in metric spaces. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, pages 154–159, 1999.

[JKB97] N.L. Johnson, Z. Kotz, and N. Balakrishnan. *Discrete multivariate distributions.* John Wiley & Sons, 1997.

[JKM99] H. V. Jagadish, Nick Koudas, and S. Muthukrishnan. Mining deviants in a time series database. In *Proceedings of Very Large Data Bases (VLDB) Conference*, pages 102–113, 1999.

[KCHP01] Eamonn J. Keogh, Selina Chu, David Hart, and Michael J. Pazzani. An online algorithm for segmenting time series. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 289–296, 2001.

[KF02] Eamonn Keogh and Theodoros Folias. The UCR time series data mining archive, 2002.

[KGP01] Konstantinos Kalpakis, Dhiral Gada, and Vasundhara Puttagunta. Distance measures for effective clustering of arima time-series. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 273–280, 2001.

[KJM95] A. Koski, M. Juhola, and M. Meriste. Syntactic recognition of ECG signals by attributed finite automata. *Pattern Recognition*, 28(12):1927–1940, 1995.

[Kle02] Jon M. Kleinberg. Bursty and hierarchical structure in streams. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 91–101, 2002.

[KNT00] Edwin M. Knorr, Raymond T. Ng, and Vladimir Tucakov. Distance-based outliers: algorithms and applications. *The VLDB Journal*, 8(3-4):237–253, 2000.

[KP98] Eamonn J. Keogh and Michael 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.

[KPR98] Jon Kleinberg, Christos Papadimitriou, and Prabhakar Raghavan. Segmentation problems. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, pages 473–482, 1998.

[KPV+03] Mikko Koivisto, Markus Perola, Teppo Varilo, et al. An MDL method for finding haplotype blocks and for estimating the strength of haplotype block boundaries. In *Pacific Symposium on Biocomputing*, pages 502–513, 2003.

[KS97] Eamonn J. Keogh and Padhraic 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.

[KSS04] Amit Kumar, Yogish Sabharwal, and Sandeep Sen. A simple linear time $(1+\epsilon)$ -approximation algorithm for k-means clustering in any dimensions. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, pages 454–462, 2004.

[LB05] Tilman Lange and Joachim M. Buhman. Combining partitions by probabilistic label aggregation. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 147–156, 2005.

[LBGHG02] W. Li, P. Bernaola-Galván, F. Haghighi, and I. Grosse. Applications of recursive segmentation to the analysis of dna sequences. *Computers and Chemistry*, 26:491–510, 2002.

[Li01] Wentian Li. Dna segmentation as a model selection process. In *International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 204–210, 2001.

[LKLcC03] Jessica Lin, Eamonn J. Keogh, Stefano Lonardi, and Bill Yuan chi Chiu. A symbolic representation of time series, with implications for streaming algorithms. In

Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD), pages 2–11, 2003.

[LM03] Iosif Lazaridis and Sharad Mehrotra. Capturing sensor-generated time series with quality guarantees. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 429–, 2003.

[LSL+00] V. Lavrenko, M. Schmill, D. Lawrie, P. Ogilvie, D. Jensen, and J. Allan. Mining concurrent text and time series. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD). Workshop on Text Mining*, pages 37–44, 2000.

[LV92] Jyh-Han Lin and Jeffrey Scott Vitter. ϵ-approximations with minimum packing constraint violation. In *Proceedings of the Symposium on the Theory of Computing (STOC)*, pages 771–782, 1992.

[MHK+04] Jani Mantyjarvi, Johan Himberg, Petri Kangas, Urpo Tuomela, and Pertti Huuskonen. Sensor signal data set for exploring context recognition of mobile devices. In *Benchmarks and databases for context recognition in PERVASIVE 2004*, 2004.

[MS01] Heikki Mannila and Marko Salmenkivi. Finding simple intensity descriptions from event sequence data. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 341–346, 2001.

[MSV04] S. Muthukrishnan, Rahul Shah, and Jeffrey Scott Vitter. Mining deviants in time series data streams. In *Proceedings of Statistical and Scientific Database Management (SSDBM)*, pages 41–50, 2004.

[MTT06] Taneli Mielikäinen, Evimaria Terzi, and Panayiotis Tsaparas. Aggregating time partitions. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2006.

[MTV97] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[MZH83] Nimrod Megiddo, Eitan Zemel, and Seifollah Louis Hakimi. The maximum coverage location problem. *SIAM Journal on Algebraic and Discrete Methods*, 4:253–261, 1983.

[OM95] Bjørn Olstad and Fredrik Manne. Efficient partitioning of sequences. *IEEE Transactions on Computers*, 44(11):1322–1326, 1995.

[PBH+01] Nila Patil, Anthony J. Berno, David A. Hinds, et al. Blocks of limited haplotype diversity revealed by high-resolution scanning of human chromosome 21. *Science*, 294:1669–1670, 2001.

[PKGF03] Spiros Papadimitriou, Hiroyuki Kitagawa, Phillip B. Gibbons, and Christos Faloutsos. LOCI: Fast outlier detection using the local correlation integral. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 315–, 2003.

[PVK+04] Themistoklis Palpanas, Michail Vlachos, Eamonn J. Keogh, Dimitrios Gunopulos, and Wagner Truppel. Online amnesic approximation of streaming time series. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 338–349, 2004.

[Ris78] Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[RJ86] L. R. Rabiner and B. H. Juang. An introduction to Hidden Markov Models. *IEEE ASSP Magazine*, pages 4–15, January 1986.

[RMRT00] Vasily Ramensky, Vsevolod Makeev, Mikhail A. Roytberg, and Vladimir Tumanyan. Dna segmentation through the bayesian approach. *Journal of Computational Biology*, 7(1-2):215–231, 2000.

[RTG00] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[SG02] Alexander Strehl and Joydeep Ghosh. Cluster ensembles – a knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*, 3, 2002.

[SHB+03] Russell Schwartz, Bjarni V. Halldórsson, Vineet Bafna, Andrew G. Clark, and Sorin Istrail. Robustness of inference of haplotype block structure. *Journal of Computational Biology*, 10(1):13–19, 2003.

[SKM02] Marko Salmenkivi, Juha Kere, and Heikki Mannila. Genome segmentation using piecewise constant intensity models and reversible jump MCMC. In *Proceedings of the European Conference on Computational Biology (ECCB)*, pages 211–218, 2002.

[SZ96] Hagit Shatkay and Stanley B. Zdonik. Approximate queries and representations for large data sequences. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 536–545, 1996.

[TT06] Evimaria Terzi and Panayiotis Tsaparas. Efficient algorithms for sequence segmentation. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2006.

[Vaz03] Vijay Vazirani. *Approximation Algorithms*. Springer, 2003.

[VLKG03] Michail Vlachos, Jessica Lin, Eamonn Keogh, and Dimitrios Gunopulos. A wavelet-based anytime algorithm for k-means clustering of time series. In *SIAM International Conference on Data Mining (SDM). Workshop on Clustering High Dimensionality Data and Its Applications*, 2003.

[VPVY06] Michail Vlachos, Spiros Papadimitriou, Zografoula Vagena, and Philip 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.

[WP03] J.D. Wall and J.K. Pritchard. Assessing the performance of the haplotype block model of linkage disequilibrium. *American Journal of Human Genetics*, 73:502–515, 2003.

[ZCC+02] K. Zhang, M. Cheng, T. Chen, M.S. Waterman, and F. Sun. A dynamic programming algorithm for haplotype block partitioning. *PNAS*, 99:7335–7339, 2002.

[ZKPF04] Cui Zhu, Hiroyuki Kitagawa, Spiros Papadimitriou, and Christos Faloutsos. OBE: Outlier by example. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 222–234, 2004.

TIETOJENKÄSITTELYTIETEEN LAITOS
PL 68 (Gustaf Hällströmin katu 2 b)
00014 Helsingin yliopisto

DEPARTMENT OF COMPUTER SCIENCE
P.O. Box 68 (Gustaf Hällströmin katu 2 b)
FIN-00014 University of Helsinki, FINLAND

JULKAISUSARJA **A**

SERIES OF PUBLICATIONS **A**

Reports may be ordered from: Kumpula Science Library, P.O. Box 64, FIN-00014 University of Helsinki, FINLAND.

A-1996-4 H. Ahonen: Generating grammars for structured documents using grammatical inference methods. 107 pp. (Ph.D. thesis).

A-1996-5 H. Toivonen: Discovery of frequent patterns in large data collections. 116 pp. (Ph.D. thesis).

A-1997-1 H. Tirri: Plausible prediction by Bayesian inference. 158 pp. (Ph.D. thesis).

A-1997-2 G. Lindén: Structured document transformations. 122 pp. (Ph.D. thesis).

A-1997-3 M. Nykänen: Querying string databases with modal logic. 150 pp. (Ph.D. thesis).

A-1997-4 E. Sutinen, J. Tarhio, S.-P. Lahtinen, A.-P. Tuovinen, E. Rautama & V. Meisalo: Eliot – an algorithm animation environment. 49 pp.

A-1998-1 G. Lindén & M. Tienari (eds.): Computer Science at the University of Helsinki 1998. 112 pp.

A-1998-2 L. Kutvonen: Trading services in open distributed environments. 231 + 6 pp. (Ph.D. thesis).

A-1998-3 E. Sutinen: Approximate pattern matching with the q-gram family. 116 pp. (Ph.D. thesis).

A-1999-1 M. Klemettinen: A knowledge discovery methodology for telecommunication network alarm databases. 137 pp. (Ph.D. thesis).

A-1999-2 J. Puustjärvi: Transactional workflows. 104 pp. (Ph.D. thesis).

A-1999-3 G. Lindén & E. Ukkonen (eds.): Department of Computer Science: annual report 1998. 55 pp.

A-1999-4 J. Kärkkäinen: Repetition-based text indexes. 106 pp. (Ph.D. thesis).

A-2000-1 P. Moen: Attribute, event sequence, and event type similarity notions for data mining. 190+9 pp. (Ph.D. thesis).

A-2000-2 B. Heikkinen: Generalization of document structures and document assembly. 179 pp. (Ph.D. thesis).

A-2000-3 P. Kähkipuro: Performance modeling framework for CORBA based distributed systems. 151+15 pp. (Ph.D. thesis).

A-2000-4 K. Lemström: String matching techniques for music retrieval. 56+56 pp. (Ph.D.Thesis).

A-2000-5 T. Karvi: Partially defined Lotos specifications and their refinement relations. 157 pp. (Ph.D.Thesis).

A-2001-1 J. Rousu: Efficient range partitioning in classification learning. 68+74 pp. (Ph.D. thesis)

A-2001-2 M. Salmenkivi: Computational methods for intensity models. 145 pp. (Ph.D. thesis)

A-2001-3 K. Fredriksson: Rotation invariant template matching. 138 pp. (Ph.D. thesis)

A-2002-1 A.-P. Tuovinen: Object-oriented engineering of visual languages. 185 pp. (Ph.D. thesis)

A-2002-2 V. Ollikainen: Simulation techniques for disease gene localization in isolated populations. 149+5 pp. (Ph.D. thesis)

A-2002-3 J. Vilo: Discovery from biosequences. 149 pp. (Ph.D. thesis)

A-2003-1 J. Lindström: Optimistic concurrency control methods for real-time database systems. 111 pp. (Ph.D. thesis)

A-2003-2 H. Helin: Supporting nomadic agent-based applications in the FIPA agent architecture. 200+17 pp. (Ph.D. thesis)

A-2003-3 S. Campadello: Middleware infrastructure for distributed mobile applications. 164 pp. (Ph.D. thesis)

A-2003-4 J. Taina: Design and analysis of a distributed database architecture for IN/GSM data. 130 pp. (Ph.D. thesis)

A-2003-5 J. Kurhila: Considering individual differences in computer-supported special and elementary education. 135 pp. (Ph.D. thesis)

A-2003-6 V. Mäkinen: Parameterized approximate string matching and local-similarity-based point-pattern matching. 144 pp. (Ph.D. thesis)

A-2003-7 M. Luukkainen: A process algebraic reduction strategy for automata theoretic verification of untimed and timed concurrent systems. 141 pp. (Ph.D. thesis)

A-2003-8 J. Manner: Provision of quality of service in IP-based mobile access networks. 191 pp. (Ph.D. thesis)

A-2004-1 M. Koivisto: Sum-product algorithms for the analysis of genetic risks. 155 pp. (Ph.D. thesis)

A-2004-2 A. Gurtov: Efficient data transport in wireless overlay networks. 141 pp. (Ph.D. thesis)

A-2004-3 K. Vasko: Computational methods and models for paleoecology. 176 pp. (Ph.D. thesis)

A-2004-4 P. Sevon: Algorithms for Association-Based Gene Mapping. 101 pp. (Ph.D. thesis)

A-2004-5 J. Viljamaa: Applying Formal Concept Analysis to Extract Framework Reuse Interface Specifications from Source Code. 206 pp. (Ph.D. thesis)

A-2004-6 J. Ravantti: Computational Methods for Reconstructing Macromolecular Complexes from Cryo-Electron Microscopy Images. 100 pp. (Ph.D. thesis)

A-2004-7 M. Kääriäinen: Learning Small Trees and Graphs that Generalize. 45+49 pp. (Ph.D. thesis)

A-2004-8 T. Kivioja: Computational Tools for a Novel Transcriptional Profiling Method. 98 pp. (Ph.D. thesis)

A-2004-9 H. Tamm: On Minimality and Size Reduction of One-Tape and Multitape Finite Automata. 80 pp. (Ph.D. thesis)

A-2005-1 T. Mielikäinen: Summarization Techniques for Pattern Collections in Data Mining. 201 pp. (Ph.D. thesis)

A-2005-2 A. Doucet: Advanced Document Description, a Sequential Approach. 161 pp. (Ph.D. thesis)

A-2006-1 A. Viljamaa: Specifying Reuse Interfaces for Task-Oriented Framework Specialization. 285 pp. (Ph.D. thesis)

A-2006-2 S. Tarkoma: Efficient Content-based Routing, Mobility-aware Topologies, and Temporal Subspace Matching. 198 pp. (Ph.D. thesis)

A-2006-3 M. Lehtonen: Indexing Heterogeneous XML for Full-Text Search. 185+3pp.(Ph.D. thesis).

A-2006-4 A. Rantanen: Algorithms for ^{13}C Metabolic Flux Analysis. 92+73pp.(Ph.D. thesis).