

IMPORTANT EXTREMA OF TIME SERIES:
THEORY AND APPLICATIONS

by

Harith Suman Gandhi

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Eugene Fink, Ph.D.
Dmitry Goldgof, Ph.D.
Sudeep Sarkar, Ph.D.

Date of Approval: January 22, 2004

Keywords: time series,extrema,importance,derivative series,distance

© Copyright 2004, Harith Suman Reddy Gandhi

Dedication

To my parents, Mohan Reddy Gandhi and Sulochana Gandhi, whose values and discipline in life serve as great examples to me.

Acknowledgements

I am greatly indebted to my Major Professor, Dr. Eugene Fink for his guidance, suggestions, and valuable comments all through out my work. This work would not have been complete without his help. His constant feedback and suggestions during the weekly discussions for past two years has been of great value to my work. His thorough reading of my work and keen editorial remarks helped me a lot during my thesis writing.

My gratitude goes to my colleagues and management at Nielsen Media Research for sharing my load of work at office and providing me the flexible work hours to attend the college and complete my thesis. I thank my colleagues at Cognizant Technology Solutions for their continuous prompting and encouragement.

I am very grateful to my parents, Mohan Reddy Gandhi and Sulochana Gandhi, my brother Sharath Chandan Gandhi, and my wife Madhuri Gandhi for their belief in me. I deeply appreciate their constant motivation and assistance during my work.

Table of Contents

List of Tables	ii
List of Figures	iii
Abstract	vi
Chapter 1 Introduction	1
Chapter 2 Previous work	3
Chapter 3 Basic concepts	6
3.1 Related structures and algorithms	6
3.2 Representation of time series	8
3.3 Distance measures	10
Chapter 4 Important points	13
4.1 Extrema	13
4.2 Important extrema	16
4.3 Importance levels	21
4.4 Compression rate	27
Chapter 5 Indexing trees	31
Chapter 6 Distance computation	35
Chapter 7 Pattern retrieval	42
Chapter 8 Concluding remarks	45
Notation	47
References	47

List of Tables

3.1	Main operations supported by stacks and doubly linked lists; the running time of all operations is constant. The linked list also allows a fast traversal in either direction by using the pointers to adjacent elements, denoted <i>prev[element]</i> and <i>next[element]</i>	7
3.2	Main operations supported by red-black trees; the running time of these operations is logarithmic in the number of elements. The tree also includes a doubly linked list of elements in sorted order, and it supports all linked-list operations listed in Table 3.1(b).	7
3.3	Main elements of the structures that represent a time series. Note that the extremal-point structure includes the five elements of the point structure and two additional elements.	8
5.1	Main elements of the structure representing a point of a series.	32
8.1	Notation.	46

List of Figures

1.1	Example of a time series and its extrema. We show the importance of each extremum and mark the extrema with importance greater than 1.	1
3.1	Example of a linked list and red-black tree.	7
3.2	Representation of a time series. We show the representation of an uncompressed series, given in Figure 1.1, and the representation of its compressed version, marked by circles in Figure 1.1.	9
3.3	Example of the distance between two time series. If the distance function for real values is $ a - b $, then the l_1 distance between these series is 2.0, and l_2 distance is 2.1.	12
4.1	Compression by extracting all extrema. We show strict extrema by circles, left and right extrema by half-circles, and end-points by squares.	13
4.2	Minima in a series.	14
4.3	Identifying all extrema. The procedure process a series a_1, \dots, a_n and uses a global variable n to represent its size. It outputs the values, indices, and types of extrema.	15
4.4	Important minima.	18
4.5	Important extrema for the distance $ a - b $ with $R = 3$. We show the strict extrema by circles, left and right extrema by half-circles, and end-points by squares.	18
4.6	Compression procedure. We process a series a_1, \dots, a_n and use a global variable n to represent its size. The procedure outputs the values, indices, and types of the selected important points.	19
4.7	This figure is continuation of Figure 4.6.	20
4.8	Original series restored from the compressed series in Figure 1.1.	20
4.9	Example series where reduction of R leads to larger distance between original series and restored series. If we use the distance function $ a - b $ with $R = 6$, the distance between original series and the restored series is 0.6071. Whereas if we use the same distance function with $R = 5$, the distance is 1.2857.	20
4.10	Compression by selecting the important extrema in the first-derivative series (a) and second-derivative series (b).	21
4.11	Importances of the extrema for the distance $ a - b $. We show the strict, left, right, and flat importance for the same series.	22

4.12	Computation of the strict importance of a minimum, as described in Lemma 4.5. To determine the importance of a_i , we identify the maximal segments a_{il}, \dots, a_{i-1} , and a_{i+1}, \dots, a_{ir} whose values are strictly greater than a_i , determine the maximal values a_{lm} and a_{rm} on both sides of a_i , and compute the importance of a_i as the smaller of distances $dist(a_i, a_{lm})$ and $dist(a_i, a_{rm})$	23
4.13	Importance calculation procedure. We process a series a_1, \dots, a_n and use a global variable n to represent its size. We use global stack S_1 to hold all the local minima and stacks S_2, S_3, S_4 to hold their respective maximal values. We use a linked list L to hold the points in the order of their indices. The procedure outputs the importances and indices of all minima.	26
4.14	Illustration of different maximal values corresponding to a minimum a_i . The point a_{lx} corresponds to the maximum value in the segment a_{sl}, \dots, a_{i-1} , whose values are strictly greater than a_i . Similarly, the point a_{rx} corresponds to the maximum value in the segment a_{i+1}, \dots, a_{sr} , whose values are strictly greater than a_i . The point a_{lm} corresponds to the maximum value in the segment a_{il}, \dots, a_{i-1} , whose values are no smaller than a_i	27
4.15	Importance of points based on the first-derivative series (a) and second-derivative series (b).	27
4.16	Three-pass algorithm for compression. The algorithm makes three passes through the series and produces the output in linear time.	28
4.17	Original series and series with 40% compression with different distance functions.	29
4.18	The modified version of SET-IMPORTANCES in one-pass algorithm. The algorithm keeps only the s most important points in the red-black tree, the other points are deleted from the tree as well as from the list L	30
5.1	Time series with importance of extrema.	32
5.2	Red-black tree with an auxiliary structure for the series in Figure 5.1. The left superior of a point is the nearest left neighbor in the series with strictly greater importance and the right superior is the nearest right neighbor with equal or greater importance.	32
5.3	Procedure for selecting s most important points from an importance tree in the sorted order of their indices.	33
5.4	Procedure for building the auxiliary structure. We process the series with known importance of points and use a global variable n to represent its size. The procedure sets left superior and right superior for each extremum in the series.	33
6.1	Distance bounds between two compressed series. For a real value r , the lower bound of the distance between two compressed series is $\sqrt[r]{\frac{e_1 \cdot g_1^r + e_2 \cdot g_2^r + e_3 \cdot g_3^r}{e_1 + e_2 + e_3}}$, and the upper bound is $\sqrt[r]{\frac{e_1 \cdot o_1^r + e_2 \cdot o_2^r + e_3 \cdot o_3^r}{e_1 + e_2 + e_3}}$	36
6.2	Procedure for finding distance bounds between two compressed series. The values D_l and D_u represent the lower and upper boundaries of the distance between two compressed series.	37

6.3	Procedure for finding the approximate distance between two series using their pre-computed indexing trees. The procedure starts with highly compressed versions of the input series by selecting 3 most important points from each series and finds the distance range between them. If the range is not within the given accuracy Δ , it increases the number of important points from each series by a factor of two in each subsequent step and re-calculates distance range between them. If the approximation can not be found after using all the extrema from both the series, the procedure outputs the actual distance using all the points from both the series.	38
6.4	At each step, the procedure retrieves next points from two series, and adjusts the minimum distance between the series, then it check to see whether the minimum distance is greater than the threshold T	40
6.5	Procedure for performing the threshold test between two series using their pre-computed indexing trees. At each step the procedure increases the number of points by a factor of two and re-calculate the upper and lower bounds of the distance between series; it then performs the threshold test. If the use of <i>all</i> important points does not provide sufficient accuracy, the procedure eventually computes the exact distance and performs the test.	41
7.1	Procedure for identifying all series that are within a given distance T from a given pattern series.	42
7.2	Procedure for identifying the nearest series from a given pattern.	43
7.3	Procedure for identifying z closest series that are within a given distance T from a given pattern series. We use <i>tree</i> to hold the pointers of z closest series. The <i>tree</i> is indexed on the distance of the series from the pattern series.	44

IMPORTANT EXTREMA OF TIME SERIES:

THEORY AND APPLICATIONS

Harith Suman Gandhi

ABSTRACT

We describe techniques for fast compression of time series and hierarchical indexing of compressed series based on the assignment of importance levels to the extrema of time series and their derivatives. We formalize the distance functions used in compression and retrieval techniques. We describe retrieval techniques that use the developed compression and indexing techniques for fast retrieval of series from a database that match a given pattern.

Chapter 1

Introduction

We view a *time series* as a sequence of values measured at equal time intervals; for example, the series in Figure 1.1 includes values 1, 3, 3, 5, and so on. Examples of time series include stock prices, weather data, and electrocardiograms. We have investigated techniques for compression of time series and hierarchical indexing of compressed series. This investigation is a continuation of the work by Pratt and Fink on the indexing and fast retrieval of time series [Pratt, 2001; Pratt and Fink, 2002; Fink and Pratt, 2003].

First, we describe a procedure for compressing time series by extraction of certain *important extrema*, that is, major minima and maxima. For example, we can compress the series in Figure 1.1 by extracting the circled extrema and discarding the other points.

Second, we give a technique for hierarchical indexing of a compressed series based on the assignment of importance levels to its extrema. For instance, we can assign the importance levels as shown in Figure 1.1, and then index extrema by their importance.

Third, we present algorithms that use the developed indexing structures for determining whether two given series are similar, and for retrieving the series similar to a given pattern from a database. We show that the hierarchical indexing leads to the reduction of the retrieval time from linear to near-logarithmic in the size of

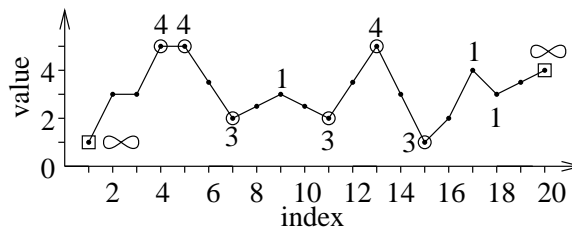


Figure 1.1: Example of a time series and its extrema. We show the importance of each extremum and mark the extrema with importance greater than 1.

the database.

Chapter 2

Previous work

We review related work on the comparison and indexing of time series.

Feature sets. Researchers have considered various feature sets for compressing time series and measuring similarity between them. They have extensively studied Fourier transforms, which allow fast compression [Sheikholeslami *et al.*, 1998; Singh and McAtackney, 1998; Stoffer, 1999; Yi *et al.*, 2000]; however, this technique has several disadvantages. In particular, it smoothes local minima and maxima, which may lead to a loss of important information, and it does not work well for erratic series [Ikeda *et al.*, 1999]. Chan *et al.* [1999; 2001] applied Haar wavelet transforms [Burrus *et al.*, 1997; Graps, 1995] to time series and showed several advantages of this technique over Fourier transforms.

Guralnik and Srivastava [1999] considered the problem of detecting a change in the trend of a data stream, and developed a technique for finding “change points” in a series. Last *et al.* [2001] proposed a general framework for knowledge discovery in time series, which included representation of a series by its key features, such as slope and signal-to-noise ratio. They described a technique for computing these features and identifying the points of change in the feature values. Pratt and Fink presented a procedure for finding major extrema in a series, which are a special case of change points [Pratt, 2001; Pratt and Fink, 2002].

Researchers have also studied the use of small alphabets for compression of series, and applied string matching [Gusfield, 1997] to the pattern search [Agrawal *et al.*, 1995; Huang and Yu, 1999; André-Jönsson and Badal, 1997; Lam and Wong, 1998; Park *et al.*, 1999; Qu *et al.*, 1998]. For example, Guralnik *et al.* [1998] compressed stock prices using a nine-letter alphabet. Singh and McAtackney [1998] represented stock prices, particle dynamics, and stellar light intensity by a three-letter alphabet. Lin and Risch [1998] used a two-letter alphabet to encode major spikes in a series.

Das *et al.* [1998] utilized an alphabet of primitive shapes for efficient compression. These techniques give a high compression rate, but their descriptive power is limited, which makes them inapplicable in many domains.

Perng *et al.* [2000] investigated a compression technique based on extraction of “landmark points,” which included local minima and maxima. Keogh and Pazzani [1998] used the endpoints of best-fit line segments to compress a series. Keogh *et al.* [2001a] reviewed and compared the compression techniques based on approximation of a time series by a sequence of straight segments. We describe an alternative compression technique, based on selection of important minima and maxima, and give a fast procedure for finding important points.

Similarity measures. Several researchers have defined similarity as the distance in a feature space. For example, Caraca-Valente and Lopez-Chavarrias [2000] used Euclidean distance between feature vectors containing angle of knee movement and muscle strength, and Lee *et al.* [2000] applied Euclidean distance to compare feature vectors containing color, texture, and shape of video data. This technique works well when all features have the same units of scale [Goldin and Kanellakis, 1995]; however, it is often ineffective for combining disparate features.

An alternative definition of similarity is based on bounding rectangles; two series are similar if their bounding rectangles are similar. This measure allows fast pruning of clearly dissimilar series [Lee *et al.*, 2000; Perng *et al.*, 2000], but it is less effective for selecting the most similar series among close candidates.

The envelope-count technique is based on dividing a series into short segments, called envelopes, and defining a yes/no similarity for each envelope. Two series are similar within an envelope if their point-by-point differences are within a certain threshold. The overall similarity is measured by the number of envelopes where the series are similar [Agrawal *et al.*, 1996]. This measure allows fast computation of similarity, and it can be adapted for noisy and missing data [Bollobas *et al.*, 1997; Das *et al.*, 1997].

Finally, we can measure a point-by-point similarity of two series and aggregate these measures, which often requires interpolation to obtain missing points. Keogh and Pazzani [1998] used linear interpolation with this technique, and Perng

et al. [2000] applied cubic approximation. Keogh and Pazzani [2000] also described a point-by-point similarity with modified Euclidean distance, which does not require interpolation.

Indexing and retrieval. Researchers have studied a variety of techniques for indexing of time series. They have utilized several advanced structures, such as *kd*-trees, R-trees, and grids. For example, Deng [1998] applied *kd*-trees to arrange series by their significant features, Chan and Fu [1999] combined wavelet transforms with R-trees, and Bozkaya and her colleagues [1999; 1997] used vantage-point trees for indexing series by numeric features.

Park *et al.* [2001b] indexed series by their local extrema and by properties of the segments between consecutive extrema. Li *et al.* [1998] proposed a retrieval technique based on a multi-level abstraction hierarchy of features. Aggarwal and Yu [2000] considered grids, but found that their performance in a multi-dimensional space is often no better than exhaustive search. They also reviewed the use of compression with exhaustive-search retrieval, and concluded that exhaustive search among compressed series is often faster than sophisticated indexing. Pratt and Fink also experimented with exhaustive search among compressed series, and showed that it allowed fast retrieval, although its theoretical time complexity was linear in the database size [Pratt, 2001; Fink and Pratt, 2003].

We describe a new technique for hierarchical indexing and retrieval of time series, by assigning importance levels to its extrema, which does not involve explicit dimension reduction. Although the worst-case complexity of the proposed retrieval procedure is linear in the database size, the average-case time is close to logarithmic.

Chapter 3

Basic concepts

We review basic algorithms and data structures used in the described work, including standard structures (Section 3.1) and representation of a time series (Section 3.2). We then define a distance between two equal-length time series and give its basic properties (Section 3.3).

3.1 Related structures and algorithms

We review the basic data structures and algorithms used in the described work, which include stacks, linked lists, red-black trees [Bayer, 1972; Guibas and Sedgwick, 1978], range trees [Edelsbrunner, 1981], and the computation of order statistics [Hoare, 1961; Blum *et al.*, 1973]. A more detailed description of range trees is available in Samet's [1990a, 1990b] books, and a description of other structures in the textbook by Cormen *et al.* [2001].

Stacks and linked lists. A *stack* is a “last-in first-out” structure, which allows adding a new element and removing the last added element in constant time. A *doubly linked list* is a sequence of elements (Figure 3.1a), which allows traversing the elements in either direction, and adding and removing elements on either end. Every element in the list includes pointers to the previous and next elements, denoted $prev[element]$ and $next[element]$. In Table 3.1, we summarize the main operations supported by stacks and linked lists.

Red-black trees. A *red-black tree* is a data structure for maintaining a sorted list, which allows fast retrieval, insertion, and deletion of elements. This tree is a variety of binary-search tree (Figure 3.1b), which includes a mechanism for balancing the tree after insertions and deletions. If a tree includes n elements, its height is at most $2 \cdot \lg(n + 1)$, and the running time of the main operations is $\Theta(n)$. In Table 3.2, we

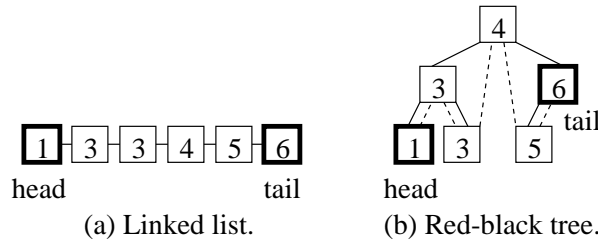


Figure 3.1: Example of a linked list and red-black tree.

(a) Operations on a stack structure:

EMPTY(*stack*) return TRUE if the stack is empty
 TOP(*stack*) return the top element of a nonempty stack
 PUSH(*stack*) add a new element to the top of the stack
 POP(*stack*) return the top element and remove it from the stack

(b) Operations on a doubly linked list:

EMPTY(*list*) return TRUE if the list is empty
 GET-FIRST(*list*), GET-LAST(*list*) return the first or last element of a nonempty list
 ADD-FIRST(*list*), ADD-LAST(*list*) add a new element in the beginning or end of the list
 DEL-FIRST(*list*), DEL-LAST(*list*) return the first or last element and remove it from the list

Table 3.1: Main operations supported by stacks and doubly linked lists; the running time of all operations is constant. The linked list also allows a fast traversal in either direction by using the pointers to adjacent elements, denoted $prev[element]$ and $next[element]$.

summarize the main operations supported by red-black trees.

The tree also includes a secondary structure, which is a doubly linked list of all elements in sorted order, shown by dashed lines in Figure 3.1(b). It allows constant-time retrieval of the minimal and maximal elements, and fast traversal of elements in sorted order.

Order statistics. The k th order statistic of an array $a[1..n]$ is the k th smallest element, that is, the element that would be in the k th place after sorting the array. An order-statistic algorithm, SELECT(a, n, k), inputs a given array and the k value, and returns the index of the k th smallest element, without sorting the given array. Hoare [1961] developed an algorithm for finding the k th order statistic with linear

FIND(*tree, index*) return an element with a given index
 INSERT(*tree, element*) add a new element to the tree
 DELETE(*tree, element*) delete an element from the tree

Table 3.2: Main operations supported by red-black trees; the running time of these operations is logarithmic in the number of elements. The tree also includes a doubly linked list of elements in sorted order, and it supports all linked-list operations listed in Table 3.1(b).

(a) Elements of the point structure:	
<i>index</i> [<i>point</i>]	index of the point in the original time series (positive integer)
<i>value</i> [<i>point</i>]	value of the point (real value)
<i>type</i> [<i>point</i>]	type of the point (end-point, minimum, maximum, or non-extremal)
<i>next</i> [<i>point</i>]	pointer to the next point in the compressed series
<i>prev</i> [<i>point</i>]	pointer to the previous point in the compressed series
(b) Additional elements of the extremal-point structure:	
<i>side</i> [<i>point</i>]	type of the extremum (strict, left, right, or flat)
<i>imp</i> [<i>point</i>]	importance of the extremum (positive real value)
(c) Elements of the uncompressed-series structure:	
<i>full-size</i> [<i>series</i>]	number of points
<i>points</i> [<i>series</i>]	array of all points
(d) Elements of the compressed-series structure:	
<i>full-size</i> [<i>series</i>]	number of points in the original series
<i>comp-size</i> [<i>series</i>]	number of points in the compressed series
<i>points</i> [<i>series</i>]	red-black tree of all points in the compressed series, indexed by their place in the original series

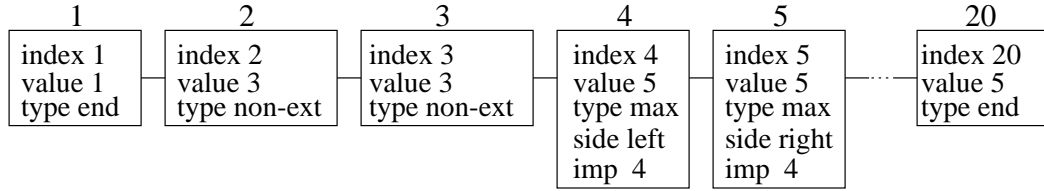
Table 3.3: Main elements of the structures that represent a time series. Note that the extremal-point structure includes the five elements of the point structure and two additional elements.

average-case time. Later, Blum *et al.* [1973] proposed an algorithm with the worst-case linear time, and Floyd and Rivest [1975] improved the algorithm for a fast average-case computation.

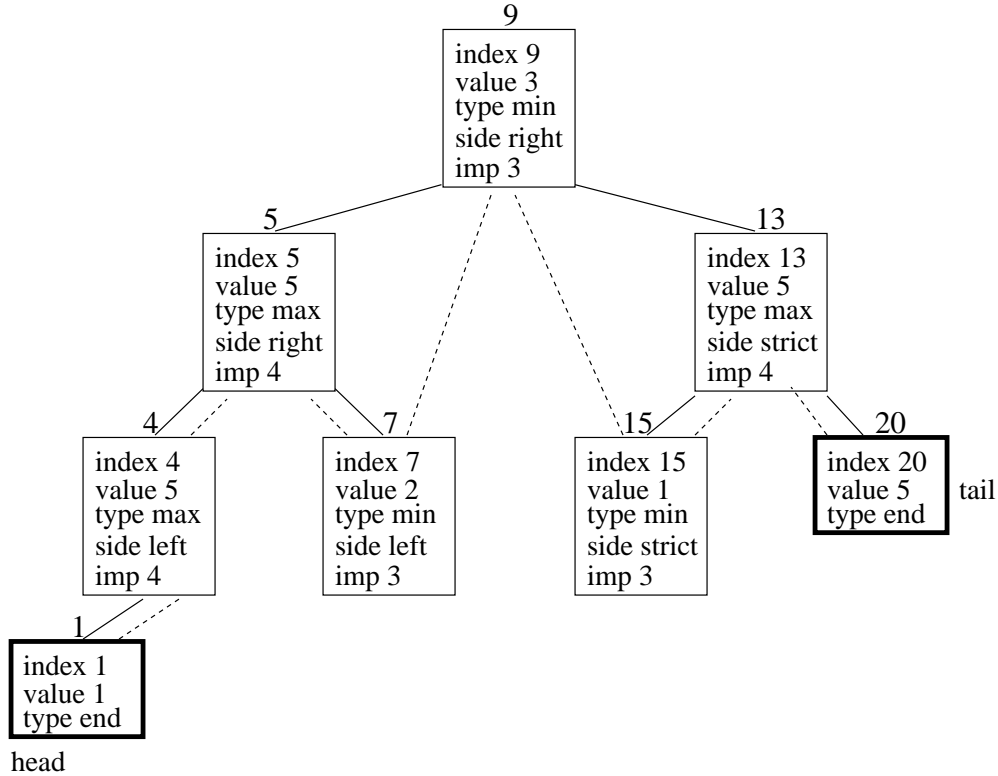
3.2 Representation of time series

We represent a time series as a collection of point structures, arranged into an array or red-black tree. A point structure describes a specific point of the time series, including its index, value, type, and pointers to the previous and next point. We summarize the main elements of the point and series structures in Table 3.3.

An *index* of a point is its position in an original uncompressed series a_1, \dots, a_n , which is an integer *value* between 1 and n , and the value of the point is the corresponding real value. The *type* shows whether the point is an end-point of the series, minimum, maximum, or a non-extremal point. For the extremal points, we may also optionally store their types and importance levels. We will explain the notion of extrema type in Sections 4.1 and 4.2, and the computation of importance levels in Section 4.3.



(a) Array representation: Twenty-point uncompressed series.



(b) Red-black tree representation: Eight-point compressed series.

Figure 3.2: Representation of a time series. We show the representation of an uncompressed series, given in Figure 1.1, and the representation of its compressed version, marked by circles in Figure 1.1.

The data structure for an uncompressed series includes an array of points, and the index of each point structure corresponds to its position in the array. For example, we represent a twenty-point series in Figure 1.1 by a twenty-element array shown in Figure 3.2(a).

The structure for a compressed series includes a red-black tree instead of an array, which indexes the points by their positions in the original uncompressed series. Note that the index of each point structure in the compressed series corresponds to its position in the original series. For instance, if we compress the series in Figure 1.1 by

extracting the circled extrema, then we represent it by the eight-element tree shown in Figure 3.2(b).

A series may optionally include a red-black tree that arranges its extrema by importance, which allows fast retrieval of major extrema. This tree includes the two end-points of the series and the extremal points, but it does not include the non-extremal points; the importance of the end-points is considered infinite. We will describe this tree and related secondary structures in Chapter 5.

3.3 Distance measures

We introduce the notion of the distance between two numbers, define the distance between two equal-length series as the mean of their point-by-point distances, and give basic properties of distances.

Definition 3.1 (Distance between real values) *A distance between real values is a two-argument function, denoted $dist(a, b)$, that satisfies the following conditions:*

- For every value a , $dist(a, a) = 0$.
- For every two values a and b , $dist(a, b) = dist(b, a)$.
- For every three values a , b , and c , if $a < b < c$, then $dist(a, b) \leq dist(a, c)$ and $dist(b, c) \leq dist(a, c)$.

We do not assume that the distance between two distinct values is greater than zero; that is, $dist(a, b)$ may be zero for distinct a and b . Also, we do not assume that a distance function satisfies the triangle inequality, that is, $dist(a, c)$ may be greater than $dist(a, b) + dist(b, c)$. We give three examples of distance functions.

Lemma 3.1 *The functions $|a - b|$, $\frac{|a-b|}{|a|+|b|}$, and $\frac{|a-b|}{\max(|a|,|b|)}$ are distance functions.*

We next describe a general method for combining distance functions into a more complex distance function.

Definition 3.2 (Distance composition) *A distance-composition function is a real-value function $f(d_1, \dots, d_q)$ on non-negative real arguments such that $f(0, \dots, 0) = 0$ and f is monotonically increasing on each of its arguments.*

Lemma 3.2 *If $dist_1, \dots, dist_q$ are distance functions, and $f(d_1, \dots, d_q)$ is a distance-composition function, then $f(dist_1(a, b), \dots, dist_q(a, b))$ is a distance function.*

Proof. We denote the composition of distances by $dist$; that is, $dist(a, b) = f(dist_1(a, b), \dots, dist_q(a, b))$. We show that $dist$ satisfies the three properties of distances stated in Definition 3.1.

- For every value a , $dist(a, a) = f(dist_1(a, a), \dots, dist_q(a, a)) = f(0, \dots, 0) = 0$.
- For every a and b , $dist(a, b) = f(dist_1(a, b), \dots, dist_q(a, b)) = f(dist_1(b, a), \dots, dist_q(b, a)) = dist(b, a)$.
- For every three values a, b , and c , if $a < b < c$, then $dist(a, b) = f(dist_1(a, b), \dots, dist_q(a, b)) \leq f(dist_1(a, c), \dots, dist_q(a, c)) = dist(a, c)$, and $dist(b, c) = f(dist_1(b, c), \dots, dist_q(b, c)) \leq f(dist_1(a, c), \dots, dist_q(a, c)) = dist(a, c)$. □

We next give two special cases of distance composition, which have been used in the implemented system.

Corollary 3.3 *If $dist_1, \dots, dist_q$ are distance functions, and w_1, \dots, w_q are positive weights, then*

- $\max_{j=1}^q w_j \cdot dist_j(a, b)$ is a distance function, and
- for every positive r , $\sqrt[r]{\sum_{j=1}^q w_j \cdot (dist_j(a, b))^r}$ is also a distance function.

Proof. The functions $\max_{j=1}^q w_j \cdot dist_j$ and $\sqrt[r]{\sum_{j=1}^q w_j \cdot dist_j^r}$ are both distance-composition functions, which implies that the respective compositions are distance functions by Lemma 3.2. □

We define the distance between two equal-length series as the l_r distance in multi-dimensional space. The definition includes two parameters, a distance function $dist$ for real values and a positive number r .

Definition 3.3 (Distance between time series) *For two equal-length series, a_1, \dots, a_n and b_1, \dots, b_n , and a distance function $dist$ for real values, the corresponding distance l_r for the series is*

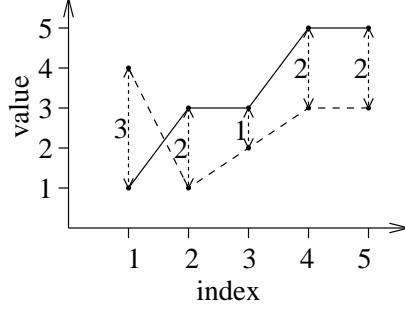


Figure 3.3: Example of the distance between two time series. If the distance function for real values is $|a - b|$, then the l_1 distance between these series is 2.0, and l_2 distance is 2.1.

$$l_r(a_{1..n}, b_{1..n}, dist) = \sqrt[r]{\frac{1}{n} \cdot \sum_{i=1}^n (dist(a_i, b_i))^r}.$$

For example, consider the two series in Figure 3.3, and suppose that we define the distance between real values as $|a - b|$, then, the l_1 distance between these series is $(3 + 2 + 1 + 2 + 2) / 5 = 2.0$, and the l_2 distance is $\sqrt{(3^2 + 2^2 + 1^2 + 2^2 + 2^2)/5} = 2.1$.

We now give a decomposition result for a distance function defined through linear combination of distances, which helps to simplify the distance computation.

Lemma 3.4 *Suppose that $dist_1, \dots, dist_q$ are distance functions for real values, and w_1, \dots, w_q are positive weights. Then, for every two sequences a_1, \dots, a_n and b_1, \dots, b_n , and every positive number r , we have:*

$$l_r(a_{1..n}, b_{1..n}, \sum_{j=1}^q w_j \cdot dist_j) = \sqrt[r]{\sum_{j=1}^q w_j^r \cdot (l_r(a_{1..n}, b_{1..n}, dist_j))^r}.$$

Proof.

$$\begin{aligned} l_r(a_{1..n}, b_{1..n}, \sum_{j=1}^q w_j \cdot dist_j) &= \sqrt[r]{\frac{1}{n} \cdot \sum_{i=1}^n \sum_{j=1}^q (w_j \cdot dist_j(a_i, b_i))^r} \\ &= \sqrt[r]{\sum_{j=1}^q w_j^r \cdot \frac{1}{n} \cdot \sum_{i=1}^n (dist_j(a_i, b_i))^r} \\ &= \sqrt[r]{\sum_{j=1}^q w_j^r \cdot (l_r(a_{1..n}, b_{1..n}, dist_j))^r}. \end{aligned}$$

□

Chapter 4

Important points

We describe algorithms for compressing time series by extracting major extrema and discarding the other points. First, we define the main types of extrema, and describe an algorithm that identifies all extrema (Section 4.1). Then, we introduce the notion of important extrema and give a procedure for identifying them (Section 4.2). We also define the importance level of an extremum and present a technique for determining the importances of extrema (Section 4.3). Finally, we give an algorithm that compresses a series at a given rate (Section 4.4).

4.1 Extrema

We begin with a simple compression based on the extraction of all extrema (Figure 4.1). We distinguish four types of extrema, called strict, left, right, and flat extrema (Figure 4.2). We give the definitions of strict, left, right, and flat minima; the definitions for maxima are similar.

Definition 4.1 (Minima) *Suppose that a_1, \dots, a_n is a time series, and a_i is a point in this series.*

- a_i is a strict minimum if $a_i < a_{i-1}$ and $a_i < a_{i+1}$.

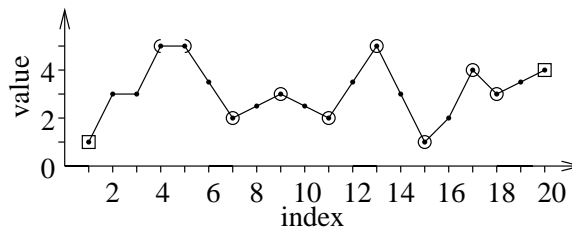


Figure 4.1: Compression by extracting all extrema. We show strict extrema by circles, left and right extrema by half-circles, and end-points by squares.

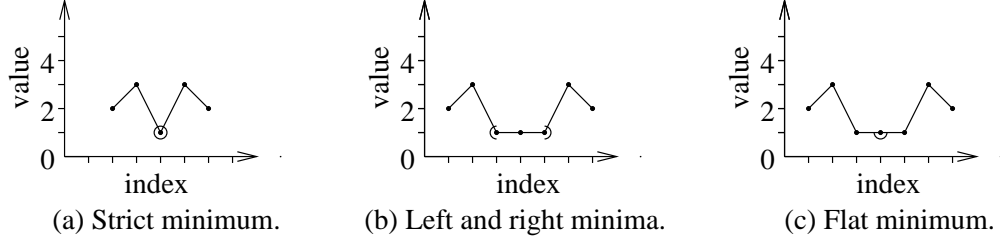


Figure 4.2: Minima in a series.

- a_i is a left minimum if $a_i < a_{i-1}$ and there is an index right $> i$ such that $a_i = a_{i+1} = \dots = a_{\text{right}} < a_{\text{right}+1}$.
- a_i is a right minimum if $a_i < a_{i+1}$ and there is an index left $< i$ such that $a_{\text{left}-1} > a_{\text{left}} = \dots = a_{i-1} = a_i$.
- a_i is a flat minimum if there are indices left $< i$ and right $> i$ such that $a_{\text{left}-1} > a_{\text{left}} = \dots = a_i = \dots = a_{\text{right}} < a_{\text{right}+1}$.

In Figure 4.3, we give a procedure that identifies all extrema and determines their types; it takes linear time and constant memory. This procedure can process new points as they arrive, without storing the original series; for example, it can process a live electrocardiogram without waiting until the end of the data collection.

We can compress a series by extracting its strict, left, and right extrema, along with the two end-points, and discarding the flat extrema and non-extremal points (Figure 4.1). We now state an important property of this compression, called *monotonicity*, used in the indexing and retrieval of time series (Chapter 5).

Definition 4.2 (Monotonic compression) Suppose that a_1, \dots, a_n is a time series, and a_{i_1}, \dots, a_{i_s} is its compressed version. The compression is monotonic if, for every consecutive indices i_c and i_{c+1} in the compressed series and every index i in the original series, if $i_c < i < i_{c+1}$, then either $a_{i_c} \leq a_i \leq a_{i_{c+1}}$ or $a_{i_{c+1}} \leq a_i \leq a_{i_c}$.

Lemma 4.1 If we compress a series by selecting all strict, left, and right extrema, along with the end-points, then the resulting compression is monotonic.

ALL-EXTREMA — Top-level function for finding extrema.

The input is a time series a_1, \dots, a_n ; the output is the values, indices, and types of all extrema.

```
i = 2
while i < n and  $a_i = a_1$  do i = i + 1
if i < n and  $a_i < a_1$  then i = FIND-MINIMUM(i)
while i < n do
    i = FIND-MAXIMUM(i)
    i = FIND-MINIMUM(i)
```

FIND-MINIMUM(*i*) — Find the first minimum after the *i*th point.

```
left = i
while i < n and  $a_i \geq a_{i+1}$  do
    i = i + 1
    if  $a_{left} > a_i$  then left = i
if i < n then OUTPUT-EXTREMUM(left, i, “min”);
return i + 1
```

FIND-MAXIMUM(*i*) — Find the first maximum after the *i*th point.

```
left = i
while i < n and  $a_i \leq a_{i+1}$  do
    i = i + 1
    if  $a_{left} < a_i$  then left = i
if i < n then OUTPUT-EXTREMUM(left, i, “max”)
return i + 1
```

OUTPUT-EXTREMUM(*left*, *right*, *type*) — Output the extrema.

```
if left = right
    then output ( $a_{right}$ , right, type, “strict”)
    else output ( $a_{left}$ , left, type, “left”)
        for flat = left + 1 to right - 1 do
            output ( $a_{flat}$ , flat, type, “flat”)
            output ( $a_{right}$ , right, type, “right”)
```

Figure 4.3: Identifying all extrema. The procedure process a series a_1, \dots, a_n and uses a global variable n to represent its size. It outputs the values, indices, and types of extrema.

4.2 Important extrema

We can achieve a higher compression rate by selecting only certain important extrema. We control the compression rate with a positive parameter R ; an increase of R leads to the selection of fewer points. We give a definition of important minima, illustrated in Figure 4.4; the definition of important maxima is similar.

Definition 4.3 (Important minimum) *A point a_i of a series a_1, \dots, a_n is an important minimum if there are indices il and ir , where $il < i < ir$, such that*

- a_i is a minimum among a_{il}, \dots, a_{ir} , and
- $\text{dist}(a_i, a_{il}) \geq R$ and $\text{dist}(a_i, a_{ir}) \geq R$.

Intuitively, a_i is an important minimum if it is the minimal value of some segment a_{il}, \dots, a_{ir} , and the end-point values of this segment are much larger than a_i . We next define strict, left, right, and flat important minima.

Definition 4.4 (Strict important minimum) *A point a_i of a series a_1, \dots, a_n is a strict important minimum if there are indices il and ir , where $il < i < ir$, such that*

- a_i is strictly smaller than a_{il}, \dots, a_{i-1} and a_{i+1}, \dots, a_{ir} , and
- $\text{dist}(a_i, a_{il}) \geq R$ and $\text{dist}(a_i, a_{ir}) \geq R$.

We give an example of a strict important minimum in Figure 4.4(a); intuitively, a point is a strict important minimum if it is a strict minimum of some segment, and the end-point values of this segment are much larger.

Definition 4.5 (Left important minimum) *A point a_i of a series a_1, \dots, a_n is a left important minimum if it is not a strict important minimum, and there are indices il and ir , where $il < i < ir$, such that*

- a_i is strictly smaller than a_{il}, \dots, a_{i-1} ,
- a_i is no larger than a_{i+1}, \dots, a_{ir} , and
- $\text{dist}(a_i, a_{il}) \geq R$ and $\text{dist}(a_i, a_{ir}) \geq R$.

The definition of a right important minimum is similar; we show left and right important minima in Figure 4.4(b).

Definition 4.6 (Flat important minimum) *A point of a series is a flat important minimum if it is an important minimum, but not strict, left, or right important minimum.*

We illustrate this definition in Figure 4.4(c); intuitively, a point is a flat important minimum if it is one of several equally important minima in some segment a_{il}, \dots, a_{ir} .

In Figure 4.6, we give a procedure that identifies the important extrema for a given R , and determines the type of each extremum. This procedure is an extended version of the compression algorithm developed by Pratt and Fink [2002].

We can compress a series by selecting its strict, left, and right important extrema for a given R , along with the two end-points, and discarding the other points. In Figure 4.5, we show the selected extrema for the distance $|a - b|$ with $R = 3$. Note that the resulting compression may not be monotonic; for example, the points a_7 and a_{11} in Figure 4.5 are consecutive important points, but the value of a_9 is *not* between the values of a_7 and a_{11} .

We can achieve a higher compression rate by selecting only strict and left extrema, or only strict and right extrema. The resulting compression is monotonic; however, it may not preserve information about flat and near-flat regions of a sequence, such as the segment a_7, \dots, a_{11} in Figure 4.5.

Lemma 4.2 *If we compress a series by selecting all strict important extrema, all left (or right) important extrema, and the end-points, then the resulting compression is monotonic.*

If we compress a series by selecting only the left and strict extrema, we call it the *left compression*. Similarly, if we use the right and strict extrema, we call it the *right compression*. If we use strict, left, and right extrema, the compression is called *symmetric*.

Recompression with larger R . We can recompress an already compressed series with a larger R using the same distance function. Recompression produces the same compressed series as the compression of the original series with the larger R . The compressed series produced using the procedure in Figure 4.6 has indices, values, and

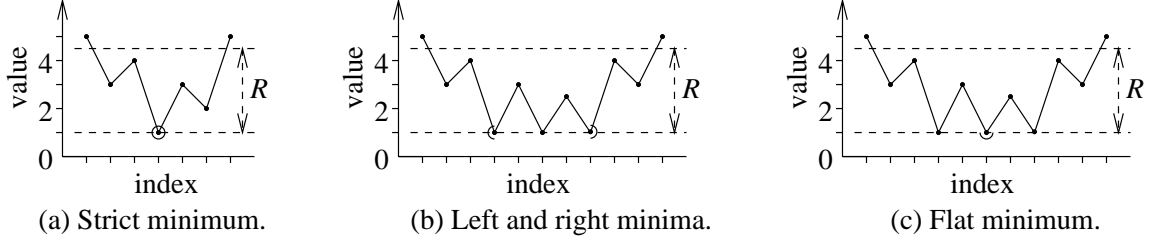


Figure 4.4: Important minima.

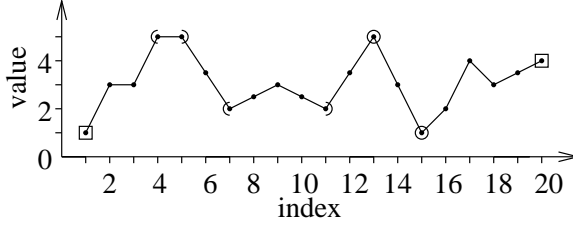


Figure 4.5: Important extrema for the distance $|a - b|$ with $R = 3$. We show the strict extrema by circles, left and right extrema by half-circles, and end-points by squares.

types of the important points in the original series. We denote the number of points in a compressed series by s , and assume that they are indexed from 1 to s . We use the procedure in Figure 4.6 with minor modifications for recompression. We input the compressed series just like the original series. For a selected important point in the input compressed series, the procedure needs to be modified to output the index of the point in the original series instead of the index in the compressed series.

Uncompression: The described compression is *lossy*, which means that we cannot restore the original series from a compressed version. If we need to construct an approximate version of the original series, we can use linear interpolation, illustrated in Figure 4.8. Specifically, we connect consecutive important points by line segments, and add points on these segments. We next give a bound on the difference between the original series and its restored version.

Lemma 4.3 *Suppose that we apply symmetric, left, or right compression to a series a_1, \dots, a_n , and then use linear interpolation to restore its approximate version b_1, \dots, b_n from the compressed series. Then, the l_r distance between the original and restored series is smaller than R ; that is, $l_r(a_{1..n}, b_{1..n}, dist) < R$.*

If we reduce the parameter R , then we select more important points, which usually leads to a closer approximation of the original series. Thus, a smaller R value

IMPORTANT-POINTS — Top-level function for finding important points. The input is a time series a_1, \dots, a_n , distance function $dist$, and R value; the output is the values, indices, and types of the selected important points.

```

i = FIND-FIRST
if i < n and  $a_i < a_1$  then i = FIND-MINIMUM(i)
while i < n do
    i = FIND-MAXIMUM(i)
    i = FIND-MINIMUM(i)

```

FIND-FIRST — Find the first important point.

```

i = 1; iMinLeft = 1; iMinRight = 1; iMaxLeft = 1; iMaxRight = 1
while i < n and  $dist(a_{i+1}, a_{iMaxLeft}) < R$  and  $dist(a_{i+1}, a_{iMinLeft}) < R$  do
    i = i + 1
    if  $a_{iMinLeft} > a_i$  then iMinLeft = i
    if  $a_{iMinRight} \geq a_i$  then iMinRight = i
    if  $a_{iMaxLeft} < a_i$  then iMaxLeft = i
    if  $a_{iMaxRight} \leq a_i$  then iMaxRight = i
    i = i + 1
if i < n and iMinLeft > 1 and  $a_i > a_1$  then
    OUTPUT-EXTREMUM(iMinLeft, iMinRight, “min”)
if i < n and iMaxLeft > 1 and  $a_i < a_1$  then
    OUTPUT-EXTREMUM(iMaxLeft, iMaxRight, “max”)
return i

```

FIND-MINIMUM(*i*) — Find the first important minimum after the *i*th point.

```

left = i; right = i
while i < n and ( $a_{i+1} < a_{left}$  or  $dist(a_{i+1}, a_{left}) < R$ ) do
    i = i + 1
    if  $a_{left} > a_i$  then left = i
    if  $a_{right} \geq a_i$  then right = i
if i < n then OUTPUT-EXTREMUM(left, right, “min”)
return i + 1

```

FIND-MAXIMUM(*i*) — Find the first important maximum after the *i*th point.

```

left = i; right = i
while i < n and ( $a_{i+1} > a_{left}$  or  $dist(a_{left}, a_{i+1}) < R$ ) do
    i = i + 1
    if  $a_{left} < a_i$  then left = i
    if  $a_{right} \leq a_i$  then right = i
if i < n then OUTPUT-EXTREMUM(left, right, “max”)
return i + 1

```

Figure 4.6: Compression procedure. We process a series a_1, \dots, a_n and use a global variable n to represent its size. The procedure outputs the values, indices, and types of the selected important points.

```

OUTPUT-EXTREMUM(left, right, type) — Output important points.
if left = right
  then output (aright, right, type, “strict”)
  else output (aleft, left, type, “left”)
    for flat = left + 1 to right - 1 do
      if aflat = aleft then output (aflat, flat, type, “flat”)
    output (aright, right, type, “right”)

```

Figure 4.7: This figure is continuation of Figure 4.6.

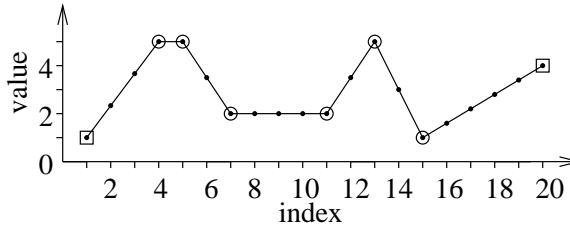


Figure 4.8: Original series restored from the compressed series in Figure 1.1.

usually leads to a smaller distance between the original and restored series; however, some series violate this rule. In Figure 4.9, we illustrate a situation in which the reduction of R leads to a less accurate compression. In this example, if we use the distance function $|a - b|$ and $R = 6$, then the l_1 distance between the original series and its restored version is 0.6071. If we use the same distance function with $R = 5$, then the l_1 distance between the original and restored series is 1.2857.

Derivative series. When compressing a time series, we may need to preserve not only its major extrema, but also major changes of its slope. For example, if the original series are as shown in Figure 4.10, then the compressed series should include the circled points. We can identify these points by finding the extrema in the first

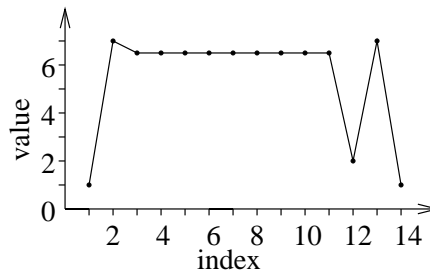


Figure 4.9: Example series where reduction of R leads to larger distance between original series and restored series. If we use the distance function $|a - b|$ with $R = 6$, the distance between original series and the restored series is 0.6071. Whereas if we use the same distance function with $R = 5$, the distance is 1.2857.

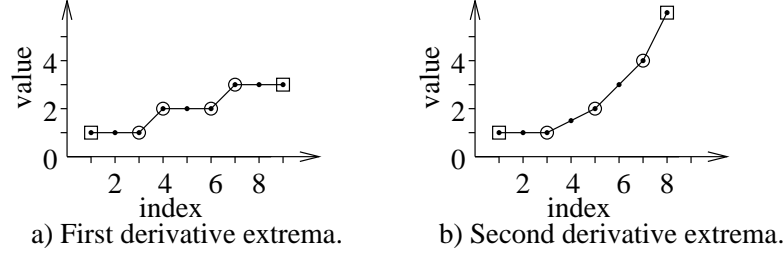


Figure 4.10: Compression by selecting the important extrema in the first-derivative series (a) and second-derivative series (b).

and second derivatives of the given series.

Definition 4.7 For a given series a_1, \dots, a_n , its first derivative is a series a'_1, \dots, a'_{n-1} , where $a'_i = a_{i+1} - a_i$ for every $i \leq n - 1$, and its second derivative is a series a''_1, \dots, a''_{n-2} , where $a''_i = a'_{i+1} - a'_i$ for every $i \leq n - 2$.

We can compress a series by identifying all strict, left, and right important extrema in the series itself, and in its first and second derivative. Note that the distance function and threshold for selecting the important extrema of a series may differ from the distance function and threshold for the derivative.

If a'_i is an important extremum of the first-derivative series, then the compressed series includes the two points of the original series used in defining a'_i , that is, a_i and a_{i+1} . Similarly, if a''_i is an important extremum in the second derivative, then the compressed series includes the three related points of the original series, a_i , a_{i+1} , and a_{i+2} .

In Figure 4.10(a), we illustrate the compression by selecting all extrema of the first derivative. In Figure 4.10(b), we show the result of selecting all extrema of the second derivative.

4.3 Importance levels

We next define the numerical importance of extrema, and give an algorithm for determining the importance of all extrema in a series.

Definition 4.8 (Importance) If a point is a strict (left, right, flat) extremum for compression with some value of R , then its strict (left, right, flat) importance is the maximal value of R for which it is a strict (left, right, flat) extremum.

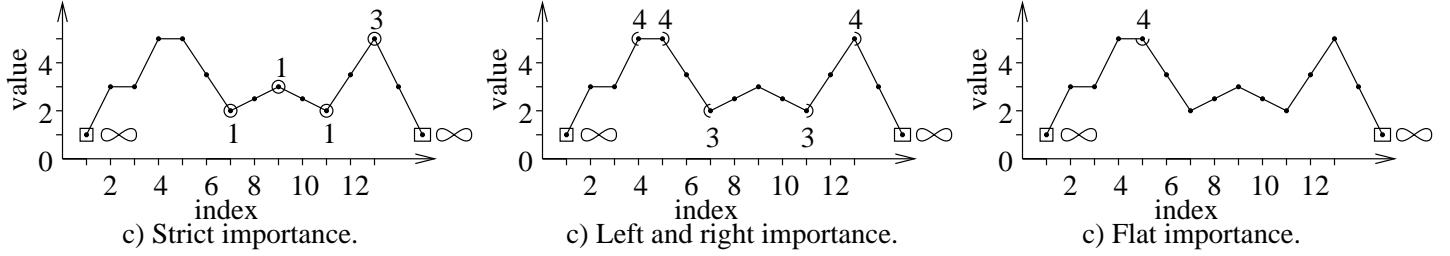


Figure 4.11: Importances of the extrema for the distance $|a - b|$. We show the strict, left, right, and flat importance for the same series.

In other words, the strict (left, right, flat) importance of an extremum is I if it is a strict (left, right, flat) extremum for compression with the R value equal to I , but not for any $R > I$. Observe that the importance of an extremum depends on a specific distance function; in Figure 4.11, we show strict, left, right, and flat importances for the $|a - b|$ distance.

If a point is not a strict (left, right, flat) important extremum for any value of R , we say that it has no strict (left, right, flat) importance. For example, the point a_7 in Figure 4.11 has no right or flat importance, whereas its strict importance is 1, and left importance is 3.

If we use the strict, left, and right important extrema in compression, then we define the importance of an extremum as the maximum of its strict, left, and right importances. If we use only the strict and left extrema, then we define the importance as the maximum of the strict and left importances. For convenience, we define the importance of the end-points as infinity, which means that we always include the end-points in a compressed series.

We now give basic properties of the importances.

Lemma 4.4

- *An extremum has a strict importance if and only if it is a strict extremum.*
- *A left (right) extremum has a left (right) importance; furthermore, if an extremum has a left (right) importance, then it is either a left (right) or strict extremum.*
- *A flat extremum has a flat importance; furthermore, it has no strict, left, or right importance.*

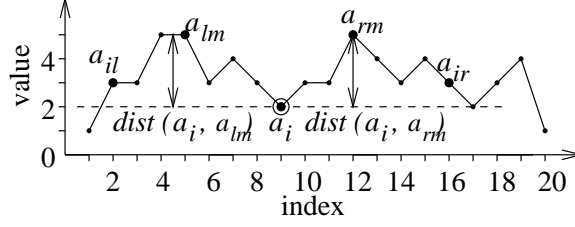


Figure 4.12: Computation of the strict importance of a minimum, as described in Lemma 4.5. To determine the importance of a_i , we identify the maximal segments a_{il}, \dots, a_{i-1} , and a_{i+1}, \dots, a_{ir} whose values are strictly greater than a_i , determine the maximal values a_{lm} and a_{rm} on both sides of a_i , and compute the importance of a_i as the smaller of distances $\text{dist}(a_i, a_{lm})$ and $\text{dist}(a_i, a_{rm})$.

We give a simple procedure for determining the strict importance of a minimum, stated as a lemma. We illustrate this procedure in Figure 4.12; determining the strict importance of a maximum is similar.

Lemma 4.5 (Strict importance) *Suppose that a_i is a strict minimum, and let a_{il}, \dots, a_{i-1} and a_{i+1}, \dots, a_{ir} be the maximal segments to the left and to the right of a_i whose values are strictly greater than a_i . Let a_{lm} be the maximal value among a_{il}, \dots, a_{i-1} , and let a_{rm} be the maximal value among a_{i+1}, \dots, a_{ir} ; then, the strict importance of a_i is equal to the minimum of the distances $\text{dist}(a_i, a_{lm})$ and $\text{dist}(a_i, a_{rm})$.*

We can use similar procedures for determining the left, right, and flat importances of the minima and maxima. We state the results for the left and flat importances of minima; the other procedures are similar.

Lemma 4.6 (Left importance) *Suppose that a_i is a strict or left minimum, and let a_{il}, \dots, a_{i-1} be the maximal segment to the left of a_i whose values are strictly greater than a_i , and a_{i+1}, \dots, a_{ir} be the maximal segment to the right of a_i whose values are no smaller than a_i . If all values among a_{i+1}, \dots, a_{ir} are strictly greater than a_i , then a_i has no left importance.*

Otherwise, let a_{rt} be the first point among a_{i+1}, \dots, a_{ir} with value equal to a_i . Furthermore, let a_{lm} be the maximal value among a_{il}, \dots, a_{i-1} , and let a_{rm} be the maximal value among a_{rt+1}, \dots, a_{ir} . If some value among a_{i+1}, \dots, a_{rt-1} is no smaller than $\min(a_{lm}, a_{rm})$, then a_i has no left importance; otherwise, its left importance is equal to the minimum of the distances $\text{dist}(a_i, a_{lm})$ and $\text{dist}(a_i, a_{rm})$.

Lemma 4.7 (Flat importance) *Suppose that a_i is a strict, left, right, or flat minimum, and let a_{i_l}, \dots, a_{i-1} and a_{i+1}, \dots, a_{i_r} be the maximal segments to the left and to the right of a_i whose values are no smaller than a_i . If all values among a_{i_l}, \dots, a_{i-1} are strictly greater than a_i , or all values among a_{i+1}, \dots, a_{i_r} are strictly greater than a_i , then a_i has no flat importance.*

Otherwise, let a_{i_t} be the last point among a_{i_l}, \dots, a_{i-1} with value equal to a_i , and a_{i_r} be the first point among a_{i+1}, \dots, a_{i_r} with value equal to a_i . Furthermore, let $a_{i_{lm}}$ be the maximal value among $a_{i_l}, \dots, a_{i_t-1}$, and let $a_{i_{rm}}$ be the maximal value among $a_{i_{r+1}}, \dots, a_{i_r}$. If some value among $a_{i_t+1}, \dots, a_{i-1}$ or among $a_{i+1}, \dots, a_{i_r-1}$ is no smaller than $\min(a_{i_{lm}}, a_{i_{rm}})$, then a_i has no flat importance; otherwise, its flat importance is equal to the minimum of the distances $\text{dist}(a_i, a_{i_{lm}})$ and $\text{dist}(a_i, a_{i_{rm}})$.

The following properties of importance values readily follow from the procedures for computing the importance.

Lemma 4.8

- *If a point has strict (left, right, flat) importance for some distance function, then it has strict (left, right, flat) importance for any other distance function.*
- *If a point has right importance, then it has no left importance; similarly, if a point has left importance, then it has no right importance.*
- *If a point has a strict and left (right) importance, then its strict importance is strictly smaller than its left (right) importance.*
- *If a point has a strict and flat importance, then its strict importance is strictly smaller than its flat importance.*
- *If a point has a left (right) and flat importance, then its left (right) importance is strictly smaller than its flat importance.*

We next show that, if we define a new distance function through a linear combination of distances, then we can calculate the importance of extrema for the new distance based on the importances for the original distances. This result helps to re-compute the importance of extrema for user-defined distance functions.

Lemma 4.9 *Suppose that $dist_1, \dots, dist_q$ are distance functions for real values, and $f(d_1, \dots, d_q)$ is a distance-composition function. Suppose further that the strict (left, right, flat) importance of a given extremal point for the $dist_1$ distance is I_1 , its strict (left, right, flat) importance for $dist_2$ is I_2 , and so on. Then, the strict (left, right, flat) importance of this point for the distance function $f(dist_1(a, b), \dots, dist_q(a, b))$ is $f(I_1, \dots, I_q)$.*

In Figure 4.13, we give a fast procedure for calculating the strict, left, right, and flat importances of all minima in a series; the procedure for determining the importances of the maxima is similar; Both procedures run in linear time, and use the memory in proportion to the number of extrema; that is, if the original series includes n points and m of them are extrema, then the procedures run in $O(n)$ time and use $O(m)$ memory.

The procedure in Figure 4.13 computes the importances of all minima in one pass through the series. When it identifies a minimum a_i , it puts its index i on stack S_1 , and adds it to the list L of all minima. The procedure then puts different maximal values corresponding to a_i (Figure 4.14) on stacks S_2, S_3 , and S_3 . The variable mv corresponds to the value a_{lx} (Figure 4.14), and is put on stack S_2 . The values a_{rm} and a_{lm} in (Figure 4.14) are put on stacks S_3 and S_4 respectively. When the procedure later reaches the end of the interval a_i, \dots, a_{ir} (Figure 4.14), it removes the point a_i from the stack and calculates its importances. We denote strict importance of an extremum a_i as *strict-imp_i*, left importance as *left-imp_i*, right importance as *right-imp_i*, and flat importance as *flat-imp_i*. In Figure 4.15, we show the importance of extrema in a derivative series.

If we are interested in the changes of the slope of a given series, we can determine the importance of extrema in the first-derivative and second-derivative series, and assign respective importances to the points of the original series. If a'_i is an extremum in the first-derivative series, then we assign its importance to the two points in the original series used in defining a'_i , that is, to a_i and a_{i+1} . Similarly, if a''_i is an extremum in the second derivative, then we assign its importance to the points a_i , a_{i+1} , and a_{i+2} . We combine the importances of the original series, first derivative, and second derivative by taking the maximum for these importances for each point.

FIND-IMPORTANCE — Top-level function for finding importance of all minima.
The input is a time series a_1, \dots, a_n ; the output is the indices and importances of all minima.

initialize an empty stack S_1 of indices of minima
initialize empty stacks S_2, S_3 , and S_4 to hold different maximal values of minima
initialize an empty linked list L to hold the points in the order of their indices

```

i = 1; left = 1; mv =  $a_1$ 
while i < n do
    while i < n and  $a_i \geq a_{i+1}$  do
        i = i + 1
        if  $a_{left} > a_i$  then left = i
    if left > 1 then PUSH – MINIMUM(left, mv)
    for flat = left + 1 to i – 1
        PUSH – MINIMUM(flat,  $a_{flat}$ )
    if i > left and i < n then PUSH – MINIMUM(i, mv)
    i = i + 1
    while i < n and  $a_i \leq a_{i+1}$  do i = i + 1
    mv =  $a_i$ ; i = i + 1; left = i
while  $S_1$  is not empty do
    mv = POP-MINIMUM(mv)

```

PUSH-MINIMUM(*i*, *mv*) — Push *i*th point onto the stack.

```

while  $S_1$  is not empty and  $a_i < \text{TOP}(S_2)$ 
    mv = POP-MINIMUM(mv)
if  $S_3$  is not empty then  $\text{TOP}(S_3) = mv$ 
PUSH( $S_1, i$ ); PUSH( $S_2, mv$ ); PUSH( $S_3, a_i$ )
if  $S_4$  is not empty
    then PUSH( $S_4, \max(\text{TOP}(S_4), mv)$ )
    else PUSH( $S_4, mv$ );
add  $a_i$  to the end of the list L

```

POP-MINIMUM(*mv*) — Pop the point and set the importances.

```

mv = SET-IMPORTANCES(POP( $S_1$ ), POP( $S_2$ ), POP( $S_3$ ), POP( $S_4$ ), mv)
return mv

```

SET-IMPORTANCE(*i*, *lm*, *rm*, *lms*, *mv*) — Set importances.

```

if  $a_i < lm$  and  $a_i < rm$  then strict-impi = min(dist( $a_i, lm$ ), dist( $a_i, rm$ ))
if  $a_i < lm$  and  $rm < mv$  then left-impi = min(dist( $a_i, lm$ ), dist( $a_i, mv$ ))
if  $a_i < rm$  and  $lm < lms$  then right-impi = min(dist( $a_i, lms$ ), dist( $a_i, rm$ ))
if  $lm < lms$  and  $rm < mv$  then flat-impi = min(dist( $a_i, lms$ ), dist( $a_i, mv$ ))
if  $mv < lm$  then mv = lm
return mv

```

Figure 4.13: Importance calculation procedure. We process a series a_1, \dots, a_n and use a global variable n to represent its size. We use global stack S_1 to hold all the local minima and stacks S_2, S_3, S_4 to hold their respective maximal values. We use a linked list L to hold the points in the order of their indices. The procedure outputs the importances and indices of all minima.

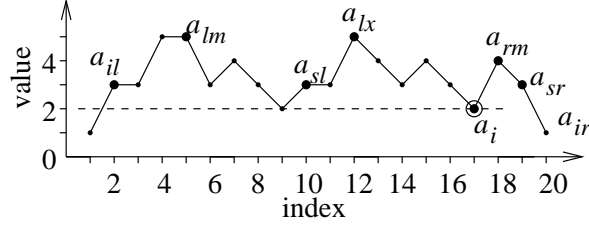


Figure 4.14: Illustration of different maximal values corresponding to a minimum a_i . The point a_{lx} corresponds to the maximum value in the segment a_{sl}, \dots, a_{i-1} , whose values are strictly greater than a_i . Similarly, the point a_{rm} corresponds to the maximum value in the segment a_{i+1}, \dots, a_{sr} , whose values are strictly greater than a_i . The point a_{lm} corresponds to the maximum value in the segment a_{il}, \dots, a_{i-1} , whose values are no smaller than a_i .

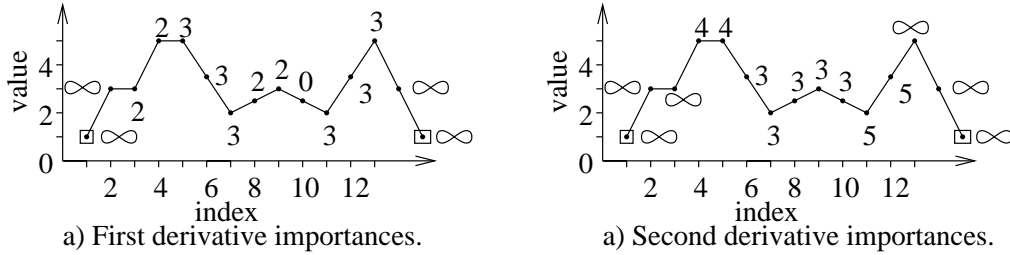


Figure 4.15: Importance of points based on the first-derivative series (a) and second-derivative series (b).

4.4 Compression rate

We next consider the problem of selecting important extrema according to a given *compression rate*, which is the percentage of points removed during the compression. For example, if the series includes hundred points and a given compression rate is 90%, then we select the ten most important extrema and remove the other points. As another example, the compression rate in Figure 1.1 is 60% since we have selected eight out of twenty points.

We assume that the given compression rate is no smaller than the percentage of non-extremal points in a series; for example, if a hundred-point series includes eighty non-extremal points, the compression rate must be at least 80%. If the series includes n points, the number of selected important extrema must be $s = \lfloor n \cdot (1 - \text{rate}) \rfloor$.

We give two algorithms that compress a series at a given rate. The first is a linear-time algorithm that performs three passes through the series. The second algorithm is slower, but it can compress a live series without storing it in memory.

Three-pass algorithm: The linear-time algorithm includes three main steps (Fig-

THREE-PASS — Three-pass algorithm for compression.

The input is a time series a_1, \dots, a_n , and the *rate* of compression; the output is the compressed series.

first-pass: produce extrema and assign importances using the procedure in Figure 4.13

second-pass: calculate s as $\lfloor n \cdot (1 - \text{rate}) \rfloor$ and find s th greatest importance

third-pass: output all the extrema whose importance is no smaller than s th greatest value

Figure 4.16: Three-pass algorithm for compression. The algorithm makes three passes through the series and produces the output in linear time.

ure 4.16). First, it calls the procedure in Figure 4.13, which produces a list of all extrema in a given series and assigns the importance to each extremum. Second, it finds the s th order statistic among the importance values, that is, the s th greatest value. Third, it selects all extrema with importances no smaller than the s th greatest value. For an n -point series with m extrema, the algorithm runs in $O(n)$ time and takes $O(m)$ memory.

Observe that the resulting compression rate may not be exactly equal to the given *rate* value. If the series has multiple extrema with the importance equal to the s th largest value, then the procedure selects all these points, which means that the actual compression rate may be lower than the given rate. For example, if we compress the series in Figure 1.1 and the given rate is 70%, then $s = \lfloor 20 \cdot (1 - 0.7) \rfloor = 6$, the s th order statistic of importances is 3, and the algorithm selects all points with importances no smaller than 3. The series includes eight such points, which means that the resulting compression rate is only 60%.

One-pass algorithm: The one-pass algorithm uses a red-black tree to index the extrema by their importance. The algorithm is a modified version of the procedure in Figure 4.13. First, the procedure in Figure 4.13 is extended to include all extrema; second, the SET-IMPORTANCES function is modified as shown in Figure 4.18. The algorithm makes one pass through the series. When the algorithm determines the importance of a new extremum, it inserts the extremum into the red-black tree. The algorithm keeps only the s most important extrema in the tree; thus, the number of points in the tree is at most s . If the tree includes s points and the algorithm finds

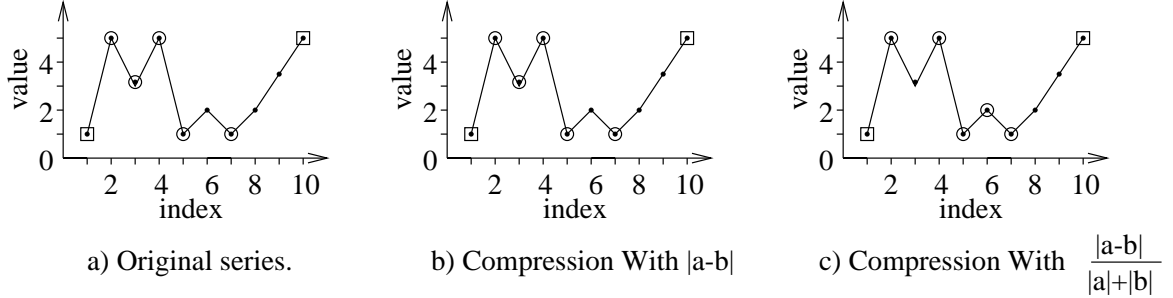


Figure 4.17: Original series and series with 40% compression with different distance functions.

a new extremum, it compares the new importance with the smallest importance in the tree. If the new importance is greater, the algorithm removes the least important point from the tree and from the list L , and then adds the new point to the tree; otherwise, it removes the new point from the list L . After the algorithm has processed the entire series, the tree includes s most important points indexed by their importance, and the list L includes the same important points in the order of their appearance in the original series. For an n -point series with m extrema, the worst-case running time of the algorithm is $O(m \cdot \lg s + n)$, and the required memory is $O(m)$.

Different distance functions: The selection of important extrema depends not only on the compression rate, but also on the distance function; that is, the use of different distances with the same compression rate may lead to different compressed sequences. For example, suppose that we need to compress the series in Figure 4.17(a) with the 40% rate. If the distance function is $|a - b|$, then the selected important extrema are as shown in Figure 4.17(b). On the other hand, if the distance is $\frac{|a-b|}{|a|+|b|}$, then we select the extrema shown in Figure 4.17(c). We next state the condition under which different distance functions lead to the same compression.

Lemma 4.10 *Suppose that $dist_1(a, b)$ is a distance function, $f(d)$ is a strictly monotonically increasing function on non-negative real arguments, and $dist_2(a, b) = f(dist_1(a, b))$. Then, for every series and every compression rate, the selection of important extrema for $dist_1$ is identical to that for $dist_2$.*

Intuitively, a monotonic transformation of a distance function does not affect the selection of important points; for example, the functions $|a - b|$ and $(a - b)^2$ lead

```

SET-IMPORTANCE( $i, lm, rm, lms, mv$ ) — Set importances and insert extremum into the
tree.
if  $a_i < lm$  and  $a_i < rm$  then  $strict-imp_i = \min(dist(a_i, lm), dist(a_i, rm))$ 
if  $a_i < lm$  and  $rm < mv$  then  $left-imp_i = \min(dist(a_i, lm), dist(a_i, mv))$ 
if  $a_i < rm$  and  $lm < lms$  then  $right-imp_i = \min(dist(a_i, lms), dist(a_i, rm))$ 
if  $lm < lms$  and  $rm < mv$  then  $flat-imp_i = \min(dist(a_i, lms), dist(a_i, mv))$ 
 $imp_i = \max(strict-imp_i, left-imp_i, right-imp_i, flat-imp_i)$ 
if  $SIZE(tree) < s$ 
  then  $INSERT(tree, a_i)$ 
  else if  $imp_i$  is greater than the importance of least important point in the tree
    then delete the least important point from the  $tree$  and from the list  $L$ .
    else delete  $a_i$  from the list  $L$ .
if  $mv < lm$  then  $mv = lm$ 
return  $mv$ 

```

Figure 4.18: The modified version of SET-IMPORTANCES in one-pass algorithm. The algorithm keeps only the s most important points in the red-black tree, the other points are deleted from the tree as well as from the list L .

to the same compression.

Chapter 5

Indexing trees

We describe two methods for indexing the extrema in a time series, which allow fast retrieval of important extrema for a given compression rate. The first method is for retrieving all important extrema in a time series, whereas the second allows retrieval of important extrema for any given segment of a series.

Red-black tree: We can use a red-black tree to index extrema in the increasing order of their importance; if several extrema have the same importance, they are sorted in the order of their appearance in the original series. This structure allows the retrieval of s most important extrema in $O(s)$ time. After retrieval of these extrema, we usually need to sort them in the order of their appearance in the original series, which takes $O(s \cdot \lg s)$ time; however, we may avoid the sorting by augmenting the red-black tree with an auxiliary structure that allows retrieval of the important extrema in sorted order.

In Figure 5.2, we illustrate the augmented structure for the series in Figure 5.1. The solid boxes are the extremal points, stored in the red-black tree, in the order of their importances; the dashed arrows form the auxiliary structure. For each extremum, the augmented structure includes a pointer to its *left* and *right superior extrema*. The *left superior* of an extremum is the nearest extremum to the left in the original series with strictly greater importance; the *right superior* is the nearest extremum to the right with equal or greater importance. In Table 5.1, we list the main elements of the structure that represents an extremum in the augmented red-black tree.

In Figure 5.3, we give a procedure that retrieves s most important points from augmented tree in the sorted order of their indices, in $O(s)$ time. It inputs the number of points to be retrieved, and outputs the indices, values, and importances of the selected points. We use a linked list L to maintain the sorted order of the

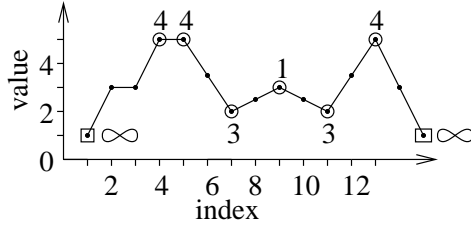


Figure 5.1: Time series with importance of extrema.

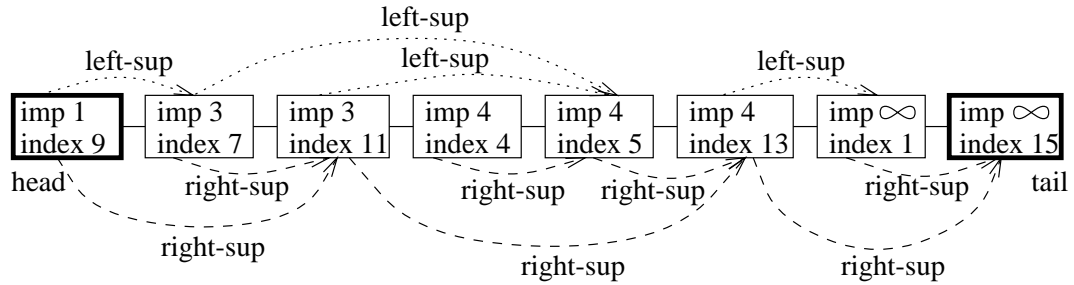


Figure 5.2: Red-black tree with an auxiliary structure for the series in Figure 5.1. The left superior of a point is the nearest left neighbor in the series with strictly greater importance and the right superior is the nearest right neighbor with equal or greater importance.

$index[point]$	index of the point
$value[point]$	value of the point
$imp[point]$	importance of the point
$next[point]$	next important point in the red-black tree
$prev[point]$	previous important point in the red-black tree
$left-sup[point]$	left superior of the point
$right-sup[point]$	right superior of the point

Table 5.1: Main elements of the structure representing a point of a series.

SORTED-RETRIEVAL(*tree*, *s*) — Retrieve *s* most important points.
The input is the indexing tree and the number of points to be retrieved; the output is the sequence of *s* most important points sorted by their indices.

initialize an empty list *L* of points
 $point = head[tree]$; $counter = 0$
while $counter < s$ and $point \neq NIL$ **do**
 if $left-sup[point] = NIL$
 then insert *point* in the beginning of the list *L*
 else insert *point* in *L* immediately after $left-sup[point]$
 $point = next[point]$
 $counter = counter + 1$

Figure 5.3: Procedure for selecting *s* most important points from an importance tree in the sorted order of their indices.

BUILD-AUGMENTED-STRUCTURE — Build augmented structure.
The input is the series with importance of points and the output is the augmented structure.

initialize an empty stack *S* of indices
for $i = 1$ to n **do**
 while *S* is not empty and $imp[points[TOP(S)]] \leq imp[points[i]]$ **do**
 $right-sup[points[TOP(S)]] = points[i]$
 POP(*S*)
 if *S* is not empty **then** $left-sup[points[i]] = points[TOP(S)]$
 PUSH(*S*, *i*)

Figure 5.4: Procedure for building the auxiliary structure. We process the series with known importance of points and use a global variable *n* to represent its size. The procedure sets left superior and right superior for each extremum in the series.

retrieved important points.

In Figure 5.4, we give a procedure that builds the auxiliary structure; for a series with *m* extrema, its time and space complexity is $O(m)$. Note that time for building the red-black tree of extrema is $O(m \cdot \lg m)$; thus the overall time for building the augmented tree is also $O(m \cdot \lg m)$. The size of the augmented tree is proportional to the number of extrema; that is, its space complexity is $O(m)$.

Range tree: We sometimes need to retrieve important extrema of a given segment a_{il}, \dots, a_{ir} , rather than all important extrema in a series. We may have to retrieve all extrema of the segment with importance above a given threshold or,

alternatively, a given number of the most important extrema. We can use the red-black tree to retrieve all important extrema in a series, and then drop the extrema outside the given segment; however, if the segment is much shorter than the series, this solution is inefficient.

A more efficient method is based on the use of a *range tree*, which is a structure for indexing points in the plane by two co-ordinates [Edelsbrunner, 1981; Samet, 1990b]. It allows a fast retrieval of all points with co-ordinates in the given ranges. We use a range tree to index all extrema in a series; the position of an extremum in a series serves as its first co-ordinate, and the importance as its second co-ordinate. For example, the importance of point a_7 in Figure 5.1 is 3; thus, its co-ordinates are 7 and 3.

If a series includes m extrema, then the space complexity of the range tree is $O(m)$, and the time for building it is $O(m \cdot \lg m)$. To identify all extrema with importance at least I in a segment a_{il}, \dots, a_{ir} , we use the range tree to retrieve the extrema with first co-ordinate in the range $[il..ir]$, and second co-ordinate in $[I..\infty]$. If the number of retrieved extrema is s , the retrieval time is $O(s + \lg m)$. We may need to sort these extrema in the order of their appearance in the series, which takes $O(s \cdot \lg s)$ time; the overall time of retrieval and sorting is $O(s \cdot \lg s + \lg m)$. A related open problem is to develop an auxiliary structure that allows retrieval of important extrema in sorted order.

Chapter 6

Distance computation

We next describe a technique for finding an approximate distance between two series based on their compressed versions. Recall that we have defined the distance between two equal-length series as the l_r distance, which uses two parameters: a distance function $dist$ for real values and a positive number r (Definition 3.3).

Since the important-extrema compression is lossy, we cannot determine the exact distance between original series based on their compressed versions; however, we can determine the upper and lower bounds on the exact distance. We describe an algorithm for computing these upper and lower bounds, which provides an approximation of the exact distance, and then show how the approximation depends on the compression parameter R . We then use the result to develop an algorithm for fast computation of approximate distances between series with pre-computed indexing trees.

Distance range: In Figure 6.1, we illustrate the calculation of distance bounds between two compressed series. The monotonicity property (Section 4.1) is used for calculating the bounds. Each leg in the series is bounded by a rectangle as shown. Bounding rectangle represent the bounds of the values of all the points in that segment in the original series. By calculating the distance bounds between bounding rectangles, we can calculate the distance bounds between two series. In Figure 6.2, we give a procedure for calculating distance bounds between two compressed series; it takes constant memory and runs in linear time.

We give two results on the accuracy of the approximate-distance computation. The first result gives a boundary on the width of the distance range, and the second shows that lower compression rate always leads to a narrower range.

Lemma 6.1 *If we compress two equal-length series, first with parameter R_a and second*

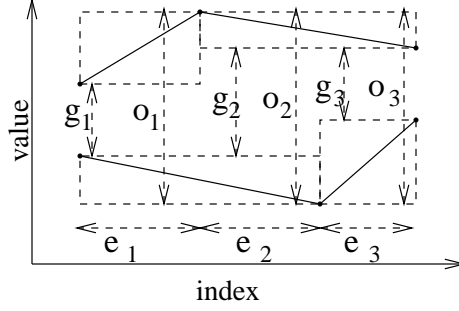


Figure 6.1: Distance bounds between two compressed series. For a real value r , the lower bound of the distance between two compressed series is $\sqrt{\frac{e_1 \cdot g_1^r + e_2 \cdot g_2^r + e_3 \cdot g_3^r}{e_1 + e_2 + e_3}}$, and the upper bound is $\sqrt{\frac{e_1 \cdot o_1^r + e_2 \cdot o_2^r + e_3 \cdot o_3^r}{e_1 + e_2 + e_3}}$

with R_b , and find the distance range $[D_l..D_u]$ for their compressed versions, then $D_u - D_l \leq R_a + R_b$.

Lemma 6.2 Suppose that we compress two equal-length series, first with parameter R_a and second with R_b , and find the distance range $[D_l..D_u]$ for their compressed versions. Suppose further that we compress the same series using different parameters R'_a and R'_b , and find the corresponding distance range $[D'_l..D'_u]$. If $R'_a \leq R_a$ and $R'_b \leq R_b$, then $D'_l \geq D_l$ and $D'_u \leq D_u$.

Approximate distance: We next consider the problem of finding the distance between two equal-length series with pre-computed indexing trees. The computation of the exact distance requires linear time, but we can usually find a good approximation of the distance in much less time.

In Figure 6.3, we give a procedure for computing an approximate distance between two series with a given accuracy Δ ; it finds the distance bounds D_l and D_u such that $D_u - D_l \leq \Delta$. The procedure first generates highly compressed versions of the given series and finds the respective distance range. Then, it repeatedly reduces the compression rate and recomputes the distance range, until the range width becomes smaller than Δ . At each step, it increases the total number of points in the compressed series by a factor of two. If the number of important extrema required for the given accuracy is s , and we use red-black indexing trees, then the time of the approximate-distance computation is $O(s)$. If the use of *all* extremal points does not provide sufficient accuracy, then the algorithm performs the exact distance computation using all points.

CAL-BOUNDS — Top-level function for calculating distance bounds.

The input includes two compressed series, a_{i_1}, \dots, a_{i_s} and b_{j_1}, \dots, b_{j_h} , and distance function $dist$; the outputs are the upper and lower bounds of the distance between series.

```

 $D_l = 0; D_u = 0; x = 1; y = 1$ 
 $D_l, D_u = \text{ADD-SEG-BOUNDS}(x, y, D_l, D_u)$ 
while  $x < s$  or  $y < h$  do
  while  $i_{x+1} < j_{y+1}$  do
     $x = x + 1$ 
     $D_l, D_u = \text{ADD-SEG-BOUNDS}(x, y, D_l, D_u)$ 
  while  $i_{x+1} = j_{y+1}$  do
     $x = x + 1; y = y + 1$ 
     $D_l, D_u = \text{ADD-SEG-BOUNDS}(x, y, D_l, D_u)$ 
  while  $j_{y+1} < i_{x+1}$  do
     $y = y + 1$ 
     $D_l, D_u = \text{ADD-SEG-BOUNDS}(x, y, D_l, D_u)$ 
 $D_l = \sqrt[r]{\frac{D_l + dist(a_{i_s}, b_{j_h})^r}{n}}$ 
 $D_u = \sqrt[r]{\frac{D_u + dist(a_{i_s}, b_{j_h})^r}{n}}$ 
output  $(D_l, D_u)$ 
return

```

ADD-SEG-BOUNDS(x, y, D_l, D_u) — Add bounds between segments.

```

 $xl = x; xm = x + 1; yl = y; ym = y + 1$ 
if  $a_{i_x} > a_{i_{x+1}}$  then  $xl = x + 1; xm = x;$ 
if  $b_{j_y} > b_{j_{y+1}}$  then  $yl = y + 1; ym = y;$ 
 $m = \min(i_{x+1} - j_y, j_{y+1} - i_x)$ 
if  $a_{i_{xl}} > b_{j_{ym}}$  or  $b_{j_{yl}} > a_{i_{xm}}$ 
  then  $D_l = D_l + m \cdot \min(dist(a_{i_{xm}}, b_{j_{yl}})^r, dist(a_{i_{xl}}, b_{j_{ym}})^r)$ 
 $D_u = D_u + m \cdot \max(dist(a_{i_{xm}}, b_{j_{yl}})^r, dist(a_{i_{xl}}, b_{j_{ym}})^r)$ 
return  $D_l, D_u$ 

```

Figure 6.2: Procedure for finding distance bounds between two compressed series. The values D_l and D_u represent the lower and upper boundaries of the distance between two compressed series.

GET-APPROXIMATE-DISTANCE — Get approximate distance between two series. The inputs are two series $a_1, \dots, a_n, b_1, \dots, b_n$, their corresponding indexing trees of extrema $tree_a, tree_b$, and Δ ; the output is the distance range with an accuracy of Δ .

```

initialize an empty list  $L_a$  of points to hold the compression of the first series
initialize an empty list  $L_b$  of points to hold the compression of the second series
 $s_a = 3; s_b = 3$ 
 $point_a = head[tree_a]; counter_a = 0$ 
 $point_b = head[tree_b]; counter_b = 0$ 
while  $s_a \leq size(tree_a)$  or  $s_b \leq size(tree_b)$  do
   $L_a, point_a, counter_a = \text{GET-COMPRESSED-SERIES}(tree_a, point_a, counter_a, L, s)$ 
   $L_b, point_b, counter_b = \text{GET-COMPRESSED-SERIES}(tree_b, point_b, counter_b, L, s)$ 
  find the distance range  $[D_u, D_l]$  between compressed series  $L_a$  and  $L_b$  using the
  procedure in Figure 6.2
  if  $D_u - D_l \leq \Delta$ 
    then output  $(D_u, D_l)$ 
    return
  else  $s_a = 2 * s_a; s_b = 2 * s_b$ 
    if  $s_a > size[tree_a]$  and  $s_a < 2 * size[tree_a]$  then  $s_a = size[tree_a]$ 
    if  $s_b > size[tree_b]$  and  $s_b < 2 * size[tree_b]$  then  $s_b = size[tree_b]$ 
calculate the actual distance  $D$  between  $a_1, \dots, a_n$ , and  $b_1, \dots, b_n$  using all the points
output  $(D)$ 

```

GET-COMPRESSED-SERIES($tree, point, counter, L, s$) — Get the compressed series.

```

while  $counter < s$  and  $point \neq \text{NIL}$  do
  if  $left-sup[point] = \text{NIL}$ 
    then insert  $point$  in the beginning of the list  $L$ 
    else insert  $point$  in  $L$  immediately after  $left-sup[point]$ 
   $point = next[point]$ 
   $counter = counter + 1$ 
return  $L, point, counter$ 

```

Figure 6.3: Procedure for finding the approximate distance between two series using their pre-computed indexing trees. The procedure starts with highly compressed versions of the input series by selecting 3 most important points from each series and finds the distance range between them. If the range is not within the given accuracy Δ , it increases the number of important points from each series by a factor of two in each subsequent step and re-calculates distance range between them. If the approximation can not be found after using all the extrema from both the series, the procedure outputs the actual distance using all the points from both the series.

If we process segments of series and use range trees, then the retrieval of important extrema is slower, which worsens the overall time of finding an approximate distance. If the total number of important points in both series is m , the length of the considered segments is p , and we need s important points for finding an approximate distance, then the overall time is $O(s \cdot \lg s + \lg m)$.

Observe that, if the length p of the given segment is smaller than $s \cdot \lg s + \lg m$, then the linear-time computation of the exact distance is faster than the approximate computation with range trees. When using range trees, we may modify the algorithm to improve its efficiency for small p . The modified algorithm first compares p and $\lg m$; if p is smaller, it computes the exact distance in $O(p)$ time. If not, the algorithm compares p with $s \cdot \lg s + \lg m$ after each increase of s ; if p becomes smaller, the algorithm switches to the computation of the exact distance. The running time of this modified algorithm is $O(\min(p, s \cdot \lg s + \lg m))$.

Threshold test: When searching for series similar to a given pattern (Chapter 7), we may need to determine whether the distance between the pattern series and another given series is smaller than a given threshold T . In other words, we need a boolean function that inputs two equal-length series and a threshold T , and returns `TRUE` if the distance between the series is no greater than T .

We can use a simple procedure that computes the exact distance between given series and the pattern, and compares with T ; its running time is linear in the length of the series. In Figure 6.4, we give more efficient procedure, which may terminate without processing the entire series if it determines that the distance is definitely greater than T ; however, if the distance is smaller than T , it has to process the entire series.

If we have already pre-computed the indexing trees for the given series, then we can usually give the same answer with much less computation by finding a distance range for the two given series. In Figure 6.5, we give an algorithm that uses red-black indexing trees to determine whether the distance is smaller than T .

The principle underlying this procedure is similar to the computation of the distance with given precision. Specifically, it begins with a wide distance range and then narrows it, by reducing the compression rate, until the threshold T falls outside

TEST-THRESHOLD — Procedure for performing the threshold test between two series. The inputs are two series $a_1, \dots, a_n, b_1, \dots, b_n$, and threshold T ; the output is either TRUE or FALSE.

```

i = 1; lr = 0
while i ≤ n do
    lr =  $\sqrt[r]{l_r^r + \frac{\text{dist}(a_i, b_i)}{n}}$ 
    if lr > T then return FALSE
return TRUE

```

Figure 6.4: At each step, the procedure retrieves next points from two series, and adjusts the minimum distance between the series, then it check to see whether the minimum distance is greater than the threshold T .

the distance range. If T is smaller than the lower bound, the procedure returns TRUE; if T is greater than the upper bound, it returns FALSE. If the use of *all* important points does not provide sufficient accuracy, the procedure eventually computes the exact distance.

If the distance between the series is close to T , the procedure may need to process all points, which means that it is no faster than the simple computation in Figure 6.4. On the other hand, if the distance is much smaller or much larger than T , the procedure processes only a small fraction of points. If the required number of important extrema from both series is s , and we use a red-black indexing tree, then the computation time is $O(s)$. If we process segments of series and use range trees, then the time is $O(s \cdot \lg s + \lg m)$. If we use range trees, we can modify the algorithm to improve its speed for small p , in the same way as in the approximate distance-computation. The running time of the modified algorithm is $O(\min(p, s \cdot \lg s + \lg m))$.

TEST-THRESHOLD-TREES — Procedure for performing the threshold test between two series using indexing trees.

The inputs are two series a_1, \dots, a_n and b_1, \dots, b_n , their corresponding indexing trees of extrema $tree_a, tree_b$, and a threshold T ; the output is either TRUE or FALSE.

initialize an empty list L_a of points to hold the compression of the first series

initialize an empty list L_b of points to hold the compression of the second series

$s_a = 3; s_b = 3$

$point_a = head[tree_a]; counter_a = 0$

$point_b = head[tree_b]; counter_b = 0$

while $s_a \leq size(tree_a)$ or $s_b \leq size(tree_b)$ **do**

$L_a, point_a, counter_a = GET-COMPRESSED-SERIES(tree_a, point_a, counter_a, L, s)$

$L_b, point_b, counter_b = GET-COMPRESSED-SERIES(tree_b, point_b, counter_b, L, s)$

find the distance range $[D_u, D_l]$ between compressed series in L_a and L_b using the procedure in Figure 6.2

if $D_u \leq T$ or $D_l > T$

then if $D_u \leq T$

then output TRUE

else output FALSE

return

else $s_a = 2 * s_a; s_b = 2 * s_b$

if $s_a > size[tree_a]$ and $s_a < 2 * size[tree_a]$ **then** $s_a = size[tree_a]$

if $s_b > size[tree_b]$ and $s_b < 2 * size[tree_b]$ **then** $s_b = size[tree_b]$

calculate the actual distance D between a_1, \dots, a_n , and b_1, \dots, b_n using all the points

if $D > T$

then output FALSE

else output TRUE

return

GET-COMPRESSED-SERIES($tree, point, counter, L, s$) — Get the compressed series.

while $counter < s$ and $point \neq \text{NIL}$ **do**

if $left-sup[point] = \text{NIL}$

then insert $point$ in the beginning of the list L

else insert $point$ in L immediately after $left-sup[point]$

$point = next[point]$

$counter = counter + 1$

return $L, point, counter$

Figure 6.5: Procedure for performing the threshold test between two series using their pre-computed indexing trees. At each step the procedure increases the number of points by a factor of two and re-calculate the upper and lower bounds of the distance between series; it then performs the threshold test. If the use of *all* important points does not provide sufficient accuracy, the procedure eventually computes the exact distance and performs the test.

Chapter 7

Pattern retrieval

We next describe algorithms for identifying series in a database that are similar to a given pattern series. We assume that all series in a database have indexing trees for a given distance function, and we consider three standard retrieval problems. First, we describe a procedure for identifying all series whose distance from a given series is no greater than a given threshold. Second, we give an algorithm for finding the series closest to a given pattern. Third, we modify it for identifying a given number of closest series.

Range query: We first consider the retrieval of all series within a given distance T from a given pattern series. In Figure 7.1, we give a simple algorithm that applies the distance-threshold test to every series in the database.

If we are searching for whole series that match a given pattern, and the database includes N candidate series, then we need to apply the similarity test N times. In the best case, the test for each series takes little time, and the overall time is close to $O(N)$. In the worst case, the algorithm may need to find the exact distance for most time series; thus, if the pattern series includes n points, the worst-case time is

RANGE-QUERY — Procedure for selecting all series that are within a given distance T from a given pattern series.

The inputs are N series $a_{11}, \dots, a_{1n}, \dots, a_{N1}, \dots, a_{Nn}$, a pattern series b_1, \dots, b_n , and threshold T ; the outputs are series that fall within the distance T from the pattern series.

```
for  $i = 1$  to  $i \leq N$  do  
    if  $l_r(a_{i1, \dots, in}, b_{1, \dots, n}, dist) \leq T$  then output  $(a_{i1}, \dots, a_{in})$   
return
```

Figure 7.1: Procedure for identifying all series that are within a given distance T from a given pattern series.

NEAREST-SERIES — Procedure for identifying the nearest series from a given pattern series.

The inputs are N series $a_{11}, \dots, a_{1n}, \dots, a_{N1}, \dots, a_{Nn}$ and a pattern series b_1, \dots, b_n ; the output is the series closest to the pattern series.

```
 $j = 0; D = \infty$   
for  $i = 1$  to  $i \leq N$  do  
  if  $l_r(a_{i1, \dots, in}, b_{1, \dots, n}, dist) < D$  then  $D = l_r(a_{i1, \dots, in}, b_{1, \dots, n}, dist); j = i$   
output  $(a_{j1}, \dots, a_{jn})$   
return
```

Figure 7.2: Procedure for identifying the nearest series from a given pattern.

$O(n \cdot N)$, which is linear in the total number of points in the database.

If we are searching for *segments* of length p similar to a given pattern, and each series in the database includes n points, then each series contains $n - p + 1$ candidate segments. In this case, the best-case running time is $O((n - p) \cdot N)$ and the worst case is $O((n - p) \cdot p \cdot N)$.

Nearest neighbor: We next consider the problem of identifying the series closest to a given patten. In Figure 7.2, we give an algorithm that inputs a pattern series and a distance threshold, and returns the closest match within the given threshold. If the database does not include any series within the given distance from the pattern, the algorithm returns no matches.

Multiple neighbors: In Figure 7.3, we give an algorithm for identifying multiple closest neighbors. It inputs a pattern series, a distance threshold, and a required number z of closest series. If the database includes at least z series within the given distance from the pattern, the algorithm identifies the z series closest to the pattern; if not, it retrieves *all* series within the given distance threshold.

MULTIPLE-SERIES — Procedure for selecting z closest series that are within a given distance T from a given pattern series.

The inputs are N series $a_{11}, \dots, a_{1n}, \dots, a_{N1}, \dots, a_{Nn}$, a pattern series b_1, \dots, b_n , and threshold T ; the outputs are z closest series that fall within the distance T from the pattern series.

```

for  $i = 1$  to  $i \leq N$  do
     $D_i = l_r(a_{i1, \dots, in}, b_{1, \dots, n}, dist)$ 
    if  $D_i \leq T$ 
        then if  $size(tree) < k$ 
            then  $INSERT(tree, i)$ 
            else  $last = GET - LAST(tree)$ 
                if  $d_{last} \geq d_i$ 
                    then  $DEL - LAST(tree)$ 
                         $INSERT(tree, i)$ 
output all the series in  $tree$ 
return

```

Figure 7.3: Procedure for identifying z closest series that are within a given distance T from a given pattern series. We use $tree$ to hold the pointers of z closest series. The $tree$ is indexed on the distance of the series from the pattern series.

Chapter 8

Concluding remarks

In this work, we formalized the distance functions used in compression and retrieval algorithms of time series. We defined the distance between real values and distance between time series as monotonic functions that satisfy certain properties. The introduction of functions gave us the flexibility to use different distance functions and their compositions in the time series compression indexing algorithms.

We extended compression techniques developed earlier by Pratt and Fink [Pratt, 2001; Pratt and Fink, 2002; Fink and Pratt, 2003] to account for flat regions and derivatives of time series. We introduced the terms strict, left, right, and flat extrema to classify different types of extrema. We have shown examples of time series where the compression based on derivatives of time series will yield better results.

We introduced the concept of importance level and developed algorithms for assigning importances to extrema and indexing the series based on the assigned importances. We developed several techniques for finding approximate distance between two series using their pre-computed indexing trees. We used these techniques in pattern-search algorithms for improving their performance.

The future work includes investigating the applicability of proposed techniques to the real world data.

a_1, \dots, a_n	original series
b_1, \dots, b_n	original series
a_{i_1}, \dots, a_{i_s}	compressed series of a_1, \dots, a_n with s points
b_{j_1}, \dots, b_{j_h}	compressed series of b_1, \dots, b_n with h points
a'_1, \dots, a'_{n-1}	first derivative series
a''_1, \dots, a''_{n-2}	second derivative series
n	size of the series
i, j	indices of points in a time series
N	number of series in a database
$a_{11}, \dots, a_{1n}, \dots, a_{N1}, \dots, a_{Nn}$	set of series in database
il, ir, lm, rm, rt, lt	indices of different points corresponding to a minimum
r	parameter used to define distance between two sequences
l_r	distance between two equal-length series in multi-dimensi
l_∞	maximum distance between two equal-length sequences
D_l, D_u	lower and upper boundaries of distance between series
R	parameter used for compression
m	number of extrema in a series
s	size of the compressed series
i_c	index in a compressed series
k	k th order statistic
$dist, dist_1, \dots, dist_q$	distance functions
$f(d_1, \dots, d_q)$	distance-composition function
w_1, \dots, w_q	non-zero weights
a, b, c	real values
<i>counter</i>	iterator used in algorithms
<i>left, right, flat</i>	indices of left, right, and flat extrema
S_1, S_2, S_3, S_4	stacks
L	doubly linked list
<i>point</i>	point structure
<i>index</i>	index of a point
<i>value</i>	value of a point
<i>next</i>	pointer to the next point in the compressed series
<i>prev</i>	pointer to the previous point in the compressed series
<i>side</i>	type of the extremum (strict, left, right, or flat)
<i>imp</i>	importance of the extremum (positive real value)
<i>left-imp, right-imp, flat-imp, strict-imp</i>	left, right, flat, and strict importances of an extremum
I, I_1, \dots, I_q	example values of importance
<i>left-sup</i>	left superior of the extrema
<i>right-sup</i>	right superior of the extrema
<i>series</i>	series structure
<i>full – size</i>	number of points in the original series
<i>comp – size</i>	number of points in the compressed series
<i>points</i>	array or red-black tree of point structures in the series
p	length of segment
z	k closest series
T	threshold
Δ	precision

Table 8.1: Notation.

References

- [Aggarwal and Yu, 2000] Charu C. Aggarwal and Philip S. Yu. The IGrid index: Reversing the dimensionality curse for similarity indexing in high-dimensional space. In *Proceedings of the Sixth ACM International Conference on Knowledge Discovery and Data Mining*, pages 119–129, 2000.
- [Agrawal *et al.*, 1995] Rakesh Agrawal, Giuseppe Psaila, Edward L. Wimmers, and Mohamed Zait. Querying shapes of histories. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pages 502–514, 1995.
- [Agrawal *et al.*, 1996] Rakesh Agrawal, Manish Mehta, John C. Shafer, Ramakrishnan Srikant, Andreas Arning, and Toni Bollinger. The Quest data mining system. In *Proceedings of the Second ACM International Conference on Knowledge Discovery and Data Mining*, pages 244–249, 1996.
- [André-Jönsson and Badal, 1997] Henrik André-Jönsson and Dushan Z. Badal. Using signature files for querying time-series data. In *Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 211–220, 1997.
- [Bayer, 1972] R. Bayer. Symmetric binary b-trees. data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.
- [Blum *et al.*, 1973] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.
- [Bollobas *et al.*, 1997] Bela Bollobas, Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. Time-series similarity problems and well-separated geometric sets. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 454–456, 1997.

- [Bozkaya and Özsoyoglu, 1999] Tolga Bozkaya and Z. Meral Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems*, 24(3):361–404, 1999.
- [Bozkaya *et al.*, 1997] Tolga Bozkaya, Nasser Yazdani, and Z. Meral Özsoyoglu. Matching and indexing sequences of different lengths. In *Proceedings of the Sixth International Conference on Information and Knowledge Management*, pages 128–135, 1997.
- [Burrus *et al.*, 1997] C. Sidney Burrus, Ramesh A. Gopinath, and Haitao Guo. *Introduction to Wavelets and Wavelet Transforms: A Primer*. Prentice Hall, Englewood Cliffs, NJ, 1997.
- [Caraca-Valente and Lopez-Chavarrias, 2000] Juan Pedro Caraca-Valente and Ignacio Lopez-Chavarrias. Discovering similar patterns in time series. In *Proceedings of the Sixth ACM International Conference on Knowledge Discovery and Data Mining*, pages 497–505, 2000.
- [Chan and Fu, 1999] Kin-Pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *Proceedings of the Fifteenth International Conference on Data Engineering*, pages 126–133, 1999.
- [Chan *et al.*, 2002] Kin-Pong Chan, Ada Wai-Chee Fu, and C. Yu. Haar wavelets for efficient similarity search of time-series: With and without time warping. *IEEE Transactions on Knowledge and Data Engineering*, 2002. To appear.
- [Cormen *et al.*, 2001] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, New York, NY, second edition, 2001.
- [Das *et al.*, 1997] Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. Finding similar time series. In *Proceedings of the First European Conference on Principles of Data Mining and Knowledge Discovery*, pages 88–100, 1997.
- [Das *et al.*, 1998] Gautam Das, King-IP Lin, Heikki Mannila, Gopal Renganathan, and Padhraic Smyth. Rule discovery from time series. In *Proceedings of the Fourth*

- ACM *International Conference on Knowledge Discovery and Data Mining*, pages 16–22, 1998.
- [Deng, 1998] Kan Deng. *OMEGA: On-Line Memory-Based General Purpose System Classifier*. PhD thesis, Robotics Institute, Carnegie Mellon University, 1998. Technical Report CMU-RI-TR-98-33.
- [Edelsbrunner, 1981] Herbert Edelsbrunner. A note on dynamic range searching. *Bulletin of the European Association for Theoretical Computer Science*, 15:34–40, 1981.
- [Fink and Pratt, 2003] Eugene Fink and Kevin B. Pratt. Indexing of compressed time series. In Mark Last, Abraham Kandel, and Horst Bunke, editors, *Data Mining in Time Series Data Bases*. World Scientific, Singapore, 2003.
- [Floyd and Rivest, 1975] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.
- [Goldin and Kanellakis, 1995] Dina Q. Goldin and Paris C. Kanellakis. On similarity queries for time-series data: Constraint specification and implementation. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 137–153, 1995.
- [Graps, 1995] Amara L. Graps. An introduction to wavelets. *IEEE Computational Science and Engineering*, 2(2):50–61, 1995.
- [Guibas and Sedgewick, 1978] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.
- [Guralnik and Srivastava, 1999] Valery Guralnik and Jaideep Srivastava. Event detection from time series data. In *Proceedings of the Fifth ACM SIGMOD International Conference on Knowledge Discovery and Data Mining*, pages 33–42, 1999.
- [Guralnik et al., 1998] Valery Guralnik, Duminda Wijesekera, and Jaideep Srivastava. Pattern-directed mining of sequence data. In *Proceedings of the Fourth ACM*

- International Conference on Knowledge Discovery and Data Mining*, pages 51–57, 1998.
- [Gusfield, 1997] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, United Kingdom, 1997.
- [Hoare, 1961] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.
- [Huang and Yu, 1999] Yun-Wu Huang and Philip S. Yu. Adaptive query processing for time-series data. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, pages 282–286, 1999.
- [Ikeda *et al.*, 1999] Keita Ikeda, Bradley V. Vaughn, and Stephen R. Quint. Wavelet decomposition of heart period data. In *Proceedings of the IEEE First Joint BMES/EMBS Conference*, pages 3–11, 1999.
- [Keogh and Pazzani, 1998] 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 Fourth ACM International Conference on Knowledge Discovery and Data Mining*, pages 239–243, 1998.
- [Keogh and Pazzani, 2000] Eamonn J. Keogh and Michael J. Pazzani. Scaling up dynamic time warping for data mining applications. In *Proceedings of the Sixth ACM International Conference on Knowledge Discovery and Data Mining*, pages 285–289, 2000.
- [Keogh *et al.*, 2001] 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*, pages 289–296, 2001.
- [Lam and Wong, 1998] Sze Kin Lam and Man Hon Wong. A fast projection algorithm for sequence data searching. *Data and Knowledge Engineering*, 28(3):321–339, 1998.

- [Last *et al.*, 2001] Mark Last, Yaron Klein, and Abraham Kandel. Knowledge discovery in time series databases. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 31(1):160–169, 2001.
- [Lee *et al.*, 2000] Seok-Lyonh Lee, Seok-Ju Chun, Deok-Hwan Kim, Ju-Hong Lee, and Chin-Wan Chung. Similarity search for multidimensional data sequences. In *Proceedings of the Sixteenth International Conference on Data Engineering*, pages 599–608, 2000.
- [Li *et al.*, 1998] Chung-Sheng Li, Philip S. Yu, and Vittorio Castelli. MALM: A framework for mining sequence database at multiple abstraction levels. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, pages 267–272, 1998.
- [Lin and Risch, 1998] Ling Lin and Tore Risch. Querying continuous time sequences. In *Proceedings of the Twenty-Fourth International Conference on Very Large Data Bases*, pages 170–181, 1998.
- [Park *et al.*, 1999] Sanghyun Park, Dongwon Lee, and Wesley W. Chu. Fast retrieval of similar subsequences in long sequence databases. In *Proceedings of the Third IEEE Knowledge and Data Engineering Exchange Workshop*, 1999.
- [Park *et al.*, 2001] Sanghyun Park, Sang-Wook Kim, and Wesley W. Chu. Segment-based approach for subsequence searches in sequence databases. In *Proceedings of the Sixteenth ACM Symposium on Applied Computing*, pages 248–252, 2001.
- [Perng *et al.*, 2000] Chang-Shing Perng, Haixun Wang, Sylvia R. Zhang, and D. Scott Parker. Landmarks: A new model for similarity-based pattern querying in time series databases. In *Proceedings of the Sixteenth International Conference on Data Engineering*, pages 33–42, 2000.
- [Pratt and Fink, 2002] Kevin B. Pratt and Eugene Fink. Search for patterns in compressed time series. *International Journal of Image and Graphics*, 2(1):89–106, 2002.

- [Pratt, 2001] Kevin B. Pratt. Locating patterns in discrete time series. Master's thesis, Computer Science and Engineering, University of South Florida, 2001.
- [Qu *et al.*, 1998] Yunyao Qu, Changzhou Wang, and Xiaoyang Sean Wang. Supporting fast search in time series for movement patterns in multiple scales. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, pages 251–258, 1998.
- [Samet, 1990a] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.
- [Samet, 1990b] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
- [Sheikholeslami *et al.*, 1998] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. WaveCluster: A multi-resolution clustering approach for very large spatial databases. In *Proceedings of the Twenty-Fourth International Conference on Very Large Data Bases*, pages 428–439, 1998.
- [Singh and McAtackney, 1998] Sameer Singh and Paul McAtackney. Dynamic time-series forecasting using local approximation. In *Proceedings of the Tenth IEEE International Conference on Tools with Artificial Intelligence*, pages 392–399, 1998.
- [Stoffer, 1999] David S. Stoffer. Detecting common signals in multiple time series using the spectral envelope. *Journal of the American Statistical Association*, 94:1341–1356, 1999.
- [Yi *et al.*, 2000] Byoung-Kee Yi, Nikolaos D. Sidiropoulos, Theodore Johnson, H. V. Jagadish, Christos Faloutsos, and Alexadros Biliris. Online data mining for co-evolving time series. In *Proceedings of the Sixteenth International Conference on Data Engineering*, pages 13–22, 2000.