Approximate Queries and Representations for Large Data Sequences

Hagit Shatkay

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-03
March 1995

1 Introduction

Domains such as Medicine, Music, Seismology and experimental sciences in general, all require non-traditional database support. They are characterized by large data sequences such as time series. Also, users of this data are typically searching for certain *patterns* of behavior that they *approximately* visualize, rather than for specific values.

Previous work such as [Mo88, SW94, WZ94, ChS94], regards *vague* and *approximate* queries, as queries to which the answers are not exactly what was asked for. The query defines an exact result, in terms of *specific values*, which is the "best" we can expect. The actual results, however, are within some measurable distance, (often expressed by a metric function), from the desired one. Hence, the *queries* are actually *precise*, while the *results* are *approximate*.

Figure 1.1 demonstrates this notion of approximation with respect to time series. The user looks for sequences similar to the (exact) sequence represented by a solid curve – which is the *query*, and will get as a result all the stored (approximate) sequences in the database that are within the boundary defined by the dashed curves. A stored sequence is an *exact match* if it is identical to the solid curve, (i.e. *distance* = 0).

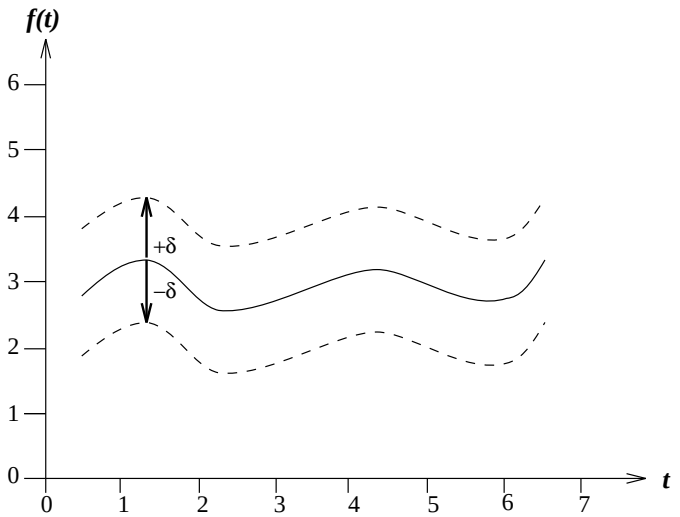

Figure 1.1: The sequence represented by a solid curve specifies a query. The result consists of all stored sequences within the region between the dashed curves. All those sequences are within distance δ from the desired sequence.

However, when dealing with long time series and scientific data, a more general notion of approximation is required. The actual values occurring in the sequences are usually not of interest and are not examined individually. It is the relationships between values, the *patterns* that the sequences follow which are of interest. For instance, in a seismic database we want to look for a sudden vigorous seismic activity, in a stock market database we look at rises and drops of stock values, in a music database we look for a melody regardless of key and tempo.

Another aspect of many real applications is that sequences are typically very long. In many

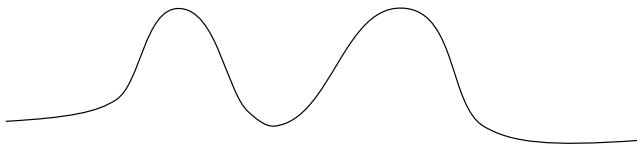

Figure 2.2: Temperature pattern with exactly two peaks.

cases they must be archived, and retrieving this data is time consuming. The retrieved data is then analyzed by domain experts to examine its features. For example, once a requested earthquake data is obtained from a central archive (a matter of days) [Fi93], examining the specific features is done by the geochemist in the lab. If this data does not contain the relevant features, another query must be issued. Since the exact data points are not necessarily interesting – we can exploit approximate representation to accomplish a significant space reduction, and have the sequences stored locally and available faster. Moreover, if search by features is supported, there will be no need for repeated retrieval.

In this paper we present a new notion of approximation that is appropriate for long data sequences and time series. We discuss approximation both in terms of queries and in terms of representation. We generalize the previous notion of approximate queries, and suggest a strategy for handling this kind of approximation, without committing ourselves to any specific query language or form.

The rest of the paper proceeds as follows: In section 2 we motivate and provide a definition of generalized approximate queries. Section 3 provides a survey of related work. Section 4 presents our "divide and conquer" strategy to handling sequences, and demonstrates how we can apply it to generalized approximate queries. Section 5 presents the algorithms we developed, examined and implemented, and provides experimental results of applying them to electrocardiograms. Section 6 concludes the paper and provides an outline of future work.

2 Generalized Approximate Queries

In current application domains that use sequential data there is a need to find patterns in data rather than explicit values. That is – individual *values* are usually not important but the *relationships* between them are. In what follows we present an example and analyze it, to clarify the issues.

2.1 GoalPost Fever – an Example

One of the symptoms of Hodgkin's disease is a temperature pattern, known as "goalpost fever", that peaks exactly twice within 24 hours. A physician looking for all patients with "goalpost fever" is interested in finding all temperature graphs that look roughly like the curve plotted in Figure 2.2. Suppose we have a specific exemplar of a sequence with 2 peaks. For instance – the one depicted in Figure 2.2, fixed on a cartesian system,

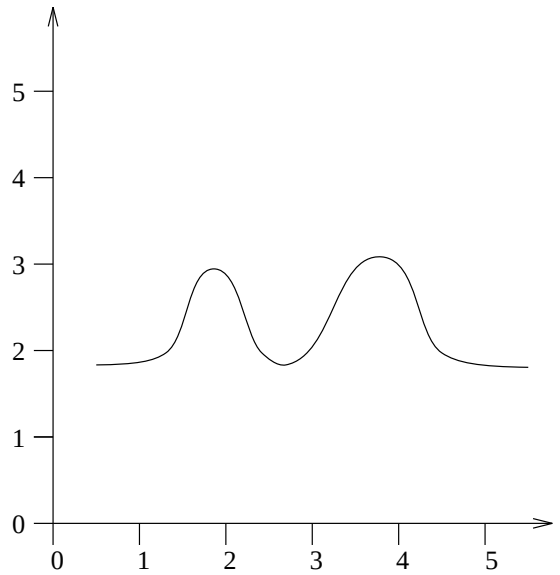

Figure 2.3: A fixed 2-peaks pattern

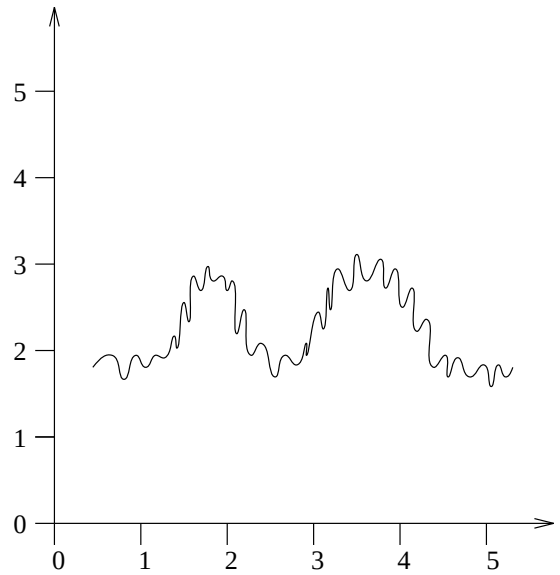

Figure 2.4: A fixed 2-peaks pattern with pointwise fluctuations within some tolerable distance.

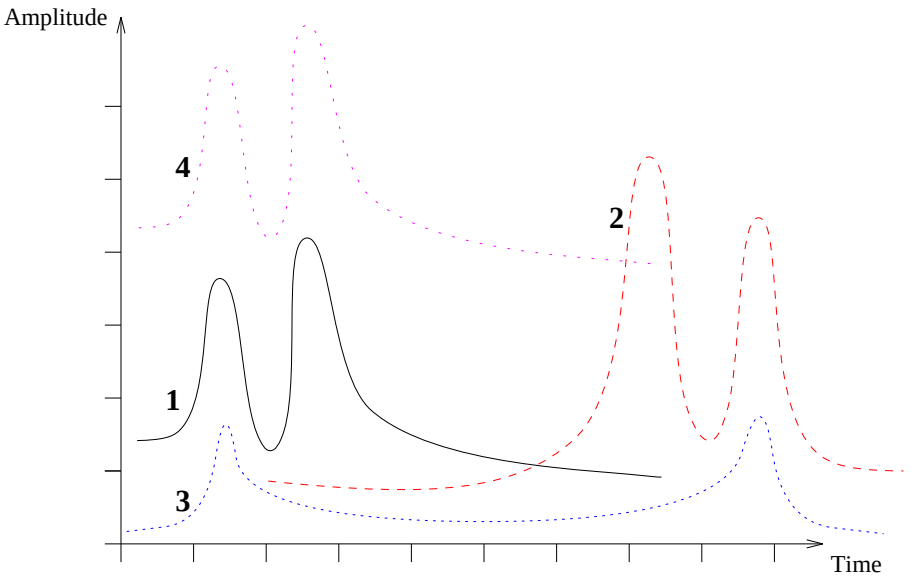

Figure 2.5: Various sequences with two peaks that are not within a value-based distance δ from the sequence depicted in Figure 3

as shown in Figure 2.3. Using this sequence as a query, fixing the distance to be δ and looking for all sequences whose values are within δ distance, allows for the matching of sequences like the one in Figure 2.4.

However, it does not cover sequences like those shown on Figure 2.5. This is because the latter demonstrate not only a δ change in amplitude with respect to the values of the query sequence, but also *scaling* (1,2,3,4), *contraction* (1,2,4), *dilation* [1](3), *shift in time* (2), *shift in amplitude* (4).

All the sequences of Figure 2.5 are results of *transformations* that maintain the "two peaks" property. Hence, the query denotes an *equivalence class* of sequences under the

[1]Dilation and contraction are both forms of changes in frequency

relation "having two peaks", which is closed under transformations that preserve this property. All these sequences are members of the class, and therefore constitute an *exact match* for the query, rather than an approximation. This is despite the fact that none of them is within *value-based* distance 0 from the others.

We should note that the query itself is *approximate* in the sense that it ignores specific values, and looks for "something with two peaks". The following section formalizes this notion of approximation.

2.2 Generalized Approximate Queries - Characterization

We define *generalized approximate queries* as queries with the following characteristics:

1. There is some general, value independent, pattern that characterizes the desired results. Defining the pattern constitutes the query. (The query could be an exemplar or an expression denoting a pattern).

2. The query denotes a (possibly uncountable) *set* $\boldsymbol{S}$ of sequences, rather than a single sequence, to be matched.

3. S is closed under any *behavior-preserving transformations*, and not merely under *identity* of specific values. Taking any pattern $r \in S$, any other $r' \in S$ can be obtained from r, through the application of transformations that preserve the significant features of r. The actual features and transformations are domain dependent. Examples of such transformations include:
   - Translation in time and amplitude.
   - Dilation and Contraction (frequency changes)
   - Deviations in time, amplitude and frequency.
   - Any combination of the above.

4. A result is an *exact match* if it is a member of S. Hence, the expected result is a *set* of matches that can all be exact.

   A result is *approximate*, if it deviates from any of the specified features within a domain-dependent error tolerance, which is a metric function over the features. This means that it is obtained from some $r \in S$ by a transformation that is not completely feature-preserving.

The above definition describes an *approximation* notion since it abstracts away from particular values and allows us to talk about how things "approximately look". It generalizes the standard notion of approximation ([Mo88, ChS94]) in the following ways:

- We generalize what the *query* denotes, from a single sequence (or a set closed under identity of values), to a set of similar sequences, which can be obtained from an exemplar through *similarity-preserving* transformations. Thus we define *approximate queries* and not just *approximate results*.

- An *approximate result* can deviate from the exact one in various dimensions (each dimension corresponds to some feature), as opposed to deviation within a fixed distance from the specific values.

2.3 Approximate Data Representation

Sequences such as time series are characterized by their large volume. Data is constantly generated, sampled, gathered and analyzed. The observation that data is *sampled* already hints that the actual values just "happened to be" what they are. A slight delay in the starting point of sampling would have resulted in a different sequence of values.

Moreover, people who use this data are rarely interested in some particular value. Efficient access is required only to locations or subsequences of the data, that satisfies some domain-dependent properties, rather than to arbitrary points. Hence, it is not necessary (nor is it feasible) to store all the data in a manner that supports immediate access to any part of it at all times. It is worthwhile to characterize the "interesting" features in the data, and store characterizations rather than the whole sequences.

Examples of such characterizations are differential equations, approximating polynomials, compressions, or main frequencies of DFT [FRM94]. We are looking for an *alphabet* of interesting features to take the place of the alphabet of numbers in the original sequence. For example, in order to find a word in a database of recorded text, it is a good strategy to transform the sequence of sampled audio signals into a sequence of words, represented as strings of ascii characters. Those ascii strings constitute the features alphabet.

Our goal is to have a transformation, from the actual data to some other alphabet for representing sequences. The target alphabet should have the following characteristics:

- The representation should be significantly more space efficient than the original.
- If a, b are two sequences, *Features(a), Features(b)* are their respective interesting features, and *Rep(a), Rep(b)* their respective representations in the new alphabet, then: *Rep(a) = Rep(b)* $\iff$ *Features(a) = Features(b)*[2]

  Namely, similar representation should correspond to similar features.
- Preserve important features of the sequence. For instance, using Lempel-Ziv compression does not preserve magnitude of data, while representing data using approximating polynomials does.
- The original data can be approximately reconstructed from the representation, with the help of some additional stored information. (Such as the start and end points of each section of the data which is represented by an approximating polynomial – if a piecewise polynomial representation is used). This means that enough information is stored to allow reconstruction of the original sequence up to some error tolerance.

[2] $\iff$ is a strong requirement, and can be relaxed by requiring it to hold with high probability. Probabilities assigned to each direction need not be the same. Sometimes false hits are acceptable but false dismissals are not, (thus right to left should have a higher probability), or vice versa.

Not all the reconstruction data needs to be highly accessible. It just has to be available.

- The representation is conveniently indexable.
- The representation supports queries on general patterns of sequences rather than digitized concrete values, as discussed in the previous subsection.
- The representation can be used to predict or deduce information about unsampled points in time. (This is a "nice to have" feature. Not absolutely necessary for supporting generalized approximate queries.)

We don't propose discarding the accurate sequences. They can be stored archivally and be used when finer resolution is needed.

3 Related Work

No existing systems or models that we know of, support the notions of approximation we have introduced. Recent work on databases for temporal and sequential data, [GuS93, Ri92, SLR94, SZ93] does not address approximation. Very interesting work was done on notions of similarity and on similarity and approximation-based search for data other than sequences [Mu87, Ja91, SW94], but it does not generalize well to the approximation required over sequences.

Faloutsos et al [AFS93, FRM94] presented recent work on similarity search on sequential data. A method is given for representing sequences, by mapping *all* subsequences of fixed length to K-dimensional points, that are K coefficients of the DFT (Discrete Fourier Transform), and using minimal bounding rectangles for storage and indexing. Queries are sequences that are matched against stored data up to some error tolerance, measured using Euclidean distance. This approach meets part of our goal, namely that of finding an efficient approximate representation for time series. The same representation is also applicable to queries, and a suitable indexing technique is provided.

However, the supported notion of similarity is limited to similar behavior in the frequency domain, and would not detect similarity under transformations such as dilation (frequency reduction) or contraction (frequency increase). For instance, looking at the goalpost fever example, none of the sequences given in Figure 2.5 would match the sequence given in Figure 2.3, if main frequencies are compared. Moreover, their approach is to have an index over all fixed-length subsequences of each sequence. We claim that not all subsequences are of interest, and therefore there is no need to facilitate efficient access to all subsequence of the data.

Other recent work presented in [AB94, ABK94, ABV94] deals with recognition, matching and indexing handwritten text using Hidden Markov Models (HMM's). Such text consists of a stream that denotes the same words even under slight variations in size and shape. Thus, the problem of realizing that two streams denote the same word is related to the problem of recognizing that sequences follow similar patterns. Their solutions consists of

breaking text into words and letters, and incorporating HMM's that recognize letters into a lexicographic index structure to support search. This approach doesn't generalize well to other sequential data which is not as structured and predictable as handwritten text.

Another current research area that is relevant to our work is Constraint Logic Programming (CLP). In order to express generalized approximate queries a supporting query language is required. The idea of using CLP as a database query language is discussed in [KaG94, KKR90, BJM93]. By using constraints which contain arithmetic predicates, we can express queries that are not supported in traditional query languages. For instance, in order to express the *Goalpost fever* query, we can define predicates for *monotonically_increasing* and *monotonically_decreasing* sequences. We can then use these predicates to define what *peak* is and then define an *exactly_two_peaks* predicate using the *peak* predicate. Further research is required in order to determine how general the constraints can get (for example, constraints involving trigonometric functions over real numbers are not handled in the current literature). We should note that even if we have an appropriate query language, we still have to provide efficient data representation and index structures to support time-effective search.

4 Our Approach – Divide and Conquer

In this section we introduce and demonstrate our idea of representing sequences in a way that facilitates generalized approximate queries. It consists of breaking sequences into meaningful subsequences and representing them using well behaved real-valued functions. No other system that we know of takes this approach.

4.1 A General Approach

Generally speaking, in order to facilitate the generalized approximation discussed in section 2, given any application domain, the following concerns must be addressed:

- Identify the important features of the data in the domain.
- Find a feature-preserving representation.
- Find a transformation from the time-series to the chosen representation.

Thus the method we pursue for supporting approximation is always tightly coupled with the application domain. Still, we believe there are techniques that are general enough to be useful for more than a single domain. In the rest of this section we propose such a technique and demonstrate how it can be applied.

4.2 Function Sequences

Our technique consists of mapping digitized sequences to sequences of real-valued functions. We claim that sequences can be broken into meaningful subsequences, and each

subsequence can be represented as a continuous and differentiable function. Thus, each numerical sequence can be transformed into a sequence of functions.

Functions have the following desirable features:

1. Significant compression can be achieved. The exact compression rate depends, of course, on the nature of the sequences, on the loss of information that is tolerated and on the chosen functions.

2. Simple lexicographic ordering/indexing exists within a single family of functions.[3] Some examples are:

   Polynomials – By degrees and coefficients (where degrees are more significant)

   $$3x^2 + 2x + 1 < x^4 + 2 < 2x^4$$

   Sinusoids – By amplitude, frequency, phase.

3. Being continuous, they allow prediction and deduction of unsampled points.

4. Behavior of functions is captured by derivatives, inflection points, extremums, etc.

The last of those features is the most important for supporting generalized approximate queries. The behavior of sequences is represented through the well-understood behavior of functions.

To facilitate ordering and indexing, we will not map each subsequence to some arbitrary approximating function, but rather associate with each application domain, the specific family of functions that is best suited for it.

To achieve the desired representation, several issues must be addressed. These include breaking the sequences, choosing functions, storing them and indexing them. The first issue is crucial for addressing the others. The following subsection elaborates on it.

4.3 Breaking Up Sequences

In order to get a good representation of a sequence using our technique, it is important that the places where a sequence is broken are carefully picked. A *breaking algorithm* determines where each subsequence starts and ends, and a *break point* is a point in the sequence on which a new subsequence starts, or a previous subsequence ends. The breaking algorithm must decide to which subsequence the break point belongs, according to the behavior of the resulting subsequences. The following list provides properties that a breaking algorithm must satisfy to be beneficial:

Consistent – Sequences with similar features (where 'similarity' is domain-dependent) are broken at corresponding break points, with respect to those features.

[3] Each subsequence has to be shifted and regarded as if starting from time 0 to allow comparison of representing functions.

Thus, the features by which subsequences are determined (for instance – minima and maxima points) are detected even when the sequence is modified by any form of feature-preserving transformations.

Therefore, if A and B are two sequences such that A can be obtained from B by a feature-preserving transformation, then when A and B are broken into subsequences, each subsequence of A can be obtained through a transformation of the corresponding subsequence of B.

Robust – Minor changes to a sequence do not affect the respective break points. Putting it formally:

Let S denote the sequence $< s_1, ..., s_n >$. Let S' denote the sequence obtained from S by adding a new element s' between s_l, s_{l+1} $(0 \leq l \leq n)$[4].

1. Suppose $< s_i, ..., s_{l-1}, s_l, s_{l+1}, ... s_j >$ is one of the subsequences obtained from applying the breaking algorithm to S, and $F(t)$ is the representing function (within error tolerance ϵ) of $< s_i, ..., s_j >$. If there exists t, $l < t < l+1$, s.t. $F(t) - s' < \epsilon$, then the algorithm must break S' s.t. $< s_i, ..., s_{l-1}, s_l, s', s_{l+1}, ... s_j >$ is one of the subsequences. The rest of the subsequences are the same as those of S.
2. Suppose $< s_i, ..., s_l >$ and $< s_{l+1}, ..., s_j >$ are two consecutive subsequences of S, and $F_1(t), F_2(t)$ are their respective representing functions. If there exists t, $l < t < l+1$, s.t. $F_1(t) - s' < \epsilon$, (or $F_2(t) - s' < \epsilon$), then the algorithm must break S' s.t. $< s_i, ..., s_l, s' >$ (or $< s', s_{l+1}, ..., s_j >$ respectively) is one of the subsequences. The rest of the subsequences are the same as those of S. (If the condition holds for *both* F_1 and F_2, any of the two associations of s' preserves robustness).

Therefore, adding or deleting "behavior preserving" elements to the sequence, where the behavior is captured by the representing function, does no more than shift the break points by at most the number of elements added/deleted.

Avoids Fragmentation – Most resulting subsequences should be of length $>> 1$. This condition is necessary for achieving a substantial compression.

We have implemented and experimented with several breaking algorithms, which are demonstrated and discussed in Sections 4.4 and 5

4.4 GoalPost Fever – An Example (cont.)

To illustrate how the divide and conquer approach handles generalized approximate queries, we show how we can use it to solve the Goalpost fever query, discussed in section 2.1. The query looks for all 24 hours sequences with *exactly two peaks.*

[4] If $l = 0$, s' is added right before s_1. If $l = n$, s' is added right after s_n.

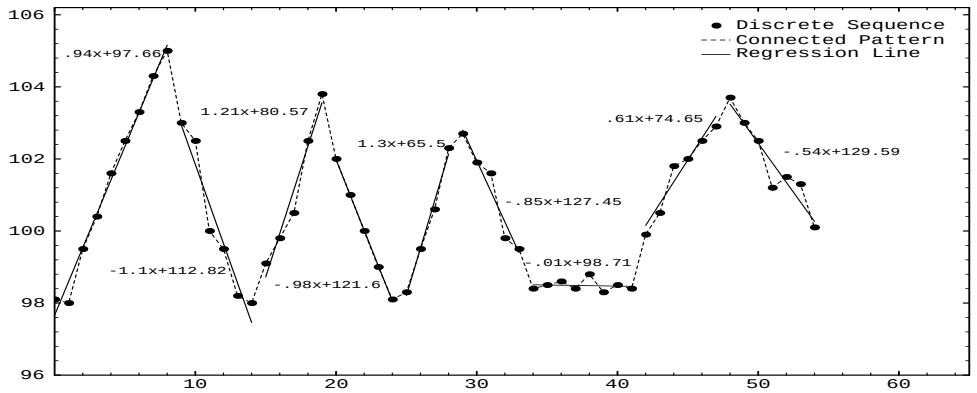

Figure 4.6: Breaking a sequence at extremums and representing it by functions. The regression function is specified near each line.

To keep things simple we make the following assumptions:

1. Each original sequence of 24 hour temperature logs is broken at extremum points, where little local extremums are ignored – up to some error tolerance ϵ. Since *peaks* are features of interest in the medicine domain, that's a very reasonable strategy.

   One of the algorithms we developed and implemented uses linear interpolation to break sequences (see section 5), and satisfies the above assumption – as demonstrated in Figure 4.6.

2. The resulting subsequences are represented by linear approximations. This is easily achieved by either using the interpolation line, which is a by-product of the breaking algorithm or by calculating the linear regression line through each subsequence. (Other information included in the representation, like start/end points of subsequences is of no interest for this example).

   Figure 4.6 shows the results of running our breaking program on a sequence, where the program calculates the approximating regression line of each resulting subsequence.

3. An index structure that supports pattern matching (such as Trie [Fr60], Position Tree [AHU74] or the one described in [Su94]) is maintained on the "positiveness" of the functions' slopes. For a fixed small number ϕ, there are 3 possible index values: $+\phi$ (slope $> \phi$), $-\phi$ (slope $< -\phi$), or 0 (slope is between $-\phi$ and ϕ). We take $\phi = 0.3$. For example, given the sequence $0(+\phi)0(-\phi)0(+\phi)(-\phi)$ (over the alphabet $\{+\phi, -\phi, 0\}$), by using the index we get the positions of the first point of all stored sequences that match that pattern. (Other index structures aren't required for this specific example).

Our approach works as follows:

The database – The stored sequences are already represented as sequences of linear functions. Each function is an approximation of a subsequence of the original sequence, as demonstrated in Figure 4.6.

The Query – The most naive way to pose the query is as a regular expression over the alphabet $\{+\phi, -\phi, 0\}$, as defined in the index assumption:

$$0^*(+\phi)(-\phi)0^*(+\phi)(-\phi)0^*$$

It is important to point out that our approach does not depend on this particular choice of pattern language.

The Result – Following the index structure, we can find all sequences matching the required pattern. The sequence depicted in Figure 4.6, does not match the query pattern, while those depicted in Figure 4.7 are all exact matches.

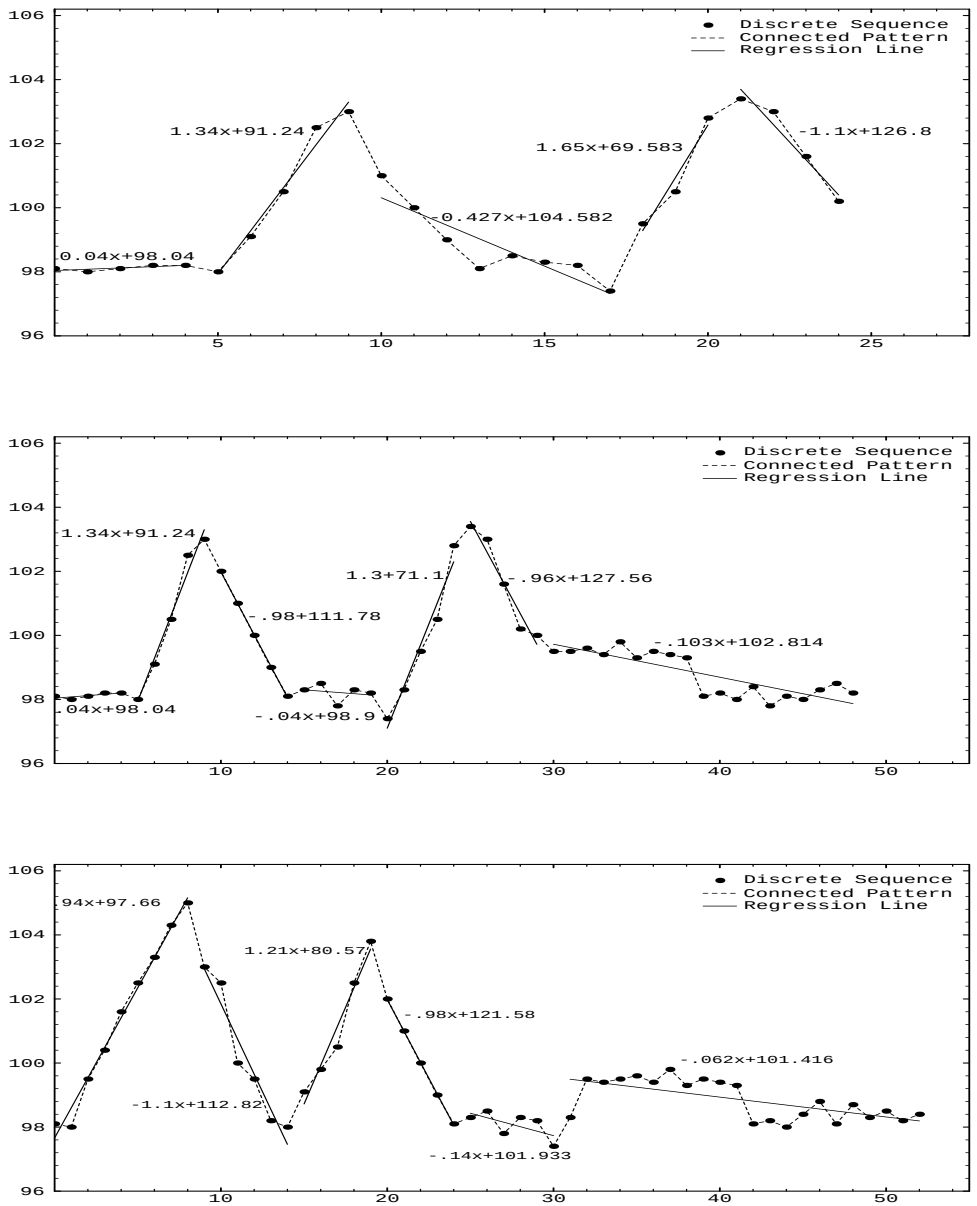

Figure 4.7: Three different two-peaks sequences, broken at extremums by our algorithm, and approximated by regression lines.

Note that the correctness of the results depends on ϕ (the steepness of the slopes) and the distance tolerated between the linear approximation and the subsequences. Our program was parameterized with distance 2.00 for the above example[5]. These values are application dependent.

5 Breaking Algorithms and Results

The goal of the breaking algorithms is to obtain break points in places where the behavior of the sequence changes significantly. There are two classes of algorithms we have studied and implemented:

On-line algorithms, determine break points while data is being gathered, based on the data seen so far, with no overall view of the sequence [Ka94]. Their main merit is the fact that while data is gathered, it is processed and stored directly in an approximated format. Thus an additional step of post-processing gathered sequences, to transform the explicit digitized version into an approximated one, is not required. Their obvious deficiency is possible lack of accuracy. Hence, it is difficult to come up with an on-line algorithm that satisfies all our requirements for a wide variety of sequences. We implemented and studied one family of on-line algorithms based on sliding a window, interpolating a polynomial through it, and breaking the sequence whenever it deviates significantly from the polynomial. The experiments are discussed in Section 5.1.

Off-line algorithms, are applied to whole sequences, and break them based on complete knowledge of the sequence. We experimented with three off-line algorithms, based on Bézier curve fitting, linear regression and linear interpolation. These are discussed in Sections 5.2, 5.3.

Section 5.4 provides a comprehensive example of applying such an algorithm to digitized electrocardiograms (ECGs).

5.1 Polynomial Interpolation on a Sliding Window

Any sequence of n points, can be interpolated by a $(n-1)^{th}$ degree polynomial. However, the interpolating polynomial representation is not more economical spacewise than the full sequence. Thus, the goal of the following algorithms is to find polynomials that interpolate several points of each subsequence, while approximating the rest. We have experimented with several variants of the basic algorithm of Figure 5.1. Basically, the algorithm slides a window over the sequence, interpolates a polynomial through it, and breaks the sequence if the mean absolute error of the sequence values seen so far, (with respect to the corresponding values of the polynomial) is larger than a given ϵ.

We should note that sliding the window and generating a *new* polynomial for each window is not the same as fixing the polynomial at the beginning of the subsequence, and trying to extend the subsequence until the error is too large. The difference is that adjusting the polynomial at each iteration can account for little divergences in the sequence, thus

[5]This is the ϵ in Figure 5.7.

Sliding Window Interpolation Algorithm

begin

Global Variables:

F	A file containing the sequence.
W_Size	The window size (and the degree of the polynomial).
$Window$	A window containing W_Size elements of the sequence.
ϵ	Error tolerance
S	The current subsequence
P	The current polynomial

1. Read the first W_Size numbers from F into S and $Window$.
2. Slide $Window$ one position to the right, reading one additional element into S.
3. Interpolate a polynomial P through the window. ($(W_Size-1)^{th}$ degree.)
4. Find mean absolute error between P and S.
5. If the mean absolute error $\leq \epsilon$ Return to step 2.
   Else {
   (a) Output S (with its *Start* and *End* point).
   (b) Reinitialize S and $Window$ starting from current position in F.
   (c) Return to step 2.
   }

end

Figure 5.1: The general algorithm for interpolating a polynomial on a sliding window.

recognizing that a subsequence still continues rather than breaking it up. For instance, the sequence: *[0, 2, 4, 9, 15, 24]* which is approximately X^2 might have been initially matched by $2X$ and broken as *[0, 2, 4, 9]* and *[15, 24]*, if the polynomial was not constantly updated, and changed to X^2 .

The algorithm of figure 5.1 suffers from the following disadvantages:

- It cuts the subsequence *exactly* at the point where the mean absolute error exceeds the threshold ϵ . It does not observe gradual change in behavior, which indicates a need to break the sequence before that point is actually reached.
- In order for the algorithm to be sensitive of the sequence behavior, it must have a wide enough window, that captures large segments of the whole sequence. This implies a high degree polynomial, which fluctuates a lot, and magnifies even small changes in the sequence behavior. On the other hand, a low degree polynomial corresponds to a narrow window, which does not reflect enough of the sequence behavior.

  For the two above reasons, breaking done by the algorithm is neither consistent nor

even robust.

- Generating a new polynomial for each window shift is time consuming, particularly for large windows.

To overcome the inability to observe gradual change and breaking at the wrong places, the following variants were tried:

5.1.1 Variant 1

The algorithm for this variant is given in Figure 5.2. Rather than breaking at the point

Sliding Window Interpolation Algorithm

begin

Global Variables: Same as in Figure 5.1

1. Read the first *W_Size* numbers from F into S and into $Window$.
2. Interpolate a polynomial P through $Window$.
3. Slide $Window$ one item to the right, reading one additional element into S.
4. Find mean absolute error between P and S.
5. If the mean absolute error $\leq \epsilon$ Return to step 3.
   Else If the mean absolute error $\geq 2\epsilon$ {
   - (a.1) Output S (with its *Start* and *End* point).
   - (a.2) Reinitialize S and $Window$ starting from current position in F.
   - (a.3) Return to step 2.

   }
   Else {
   - (b.1) Let P be a new polynomial through $Window$.
   - (b.2) If the mean absolute error with respect to P is reduced, Return to step 3 using P.

     Else{
     - (c.1) Step back on S until reaching a position i, s.t. $|S[i] - P(i)| > \epsilon$.
     - (c.2) Output S up to position $i-1$.
     - (c.3) Reinitialize $Window$ and S, from position i.
     - (c.4) Return to step 2.

     }

   }

end

Figure 5.2: Sliding Window interpolation algorithm adjusted for gradual breaking.

where the deviation exceeds ϵ, breaking is deferred until deviation is twice as big as ϵ.

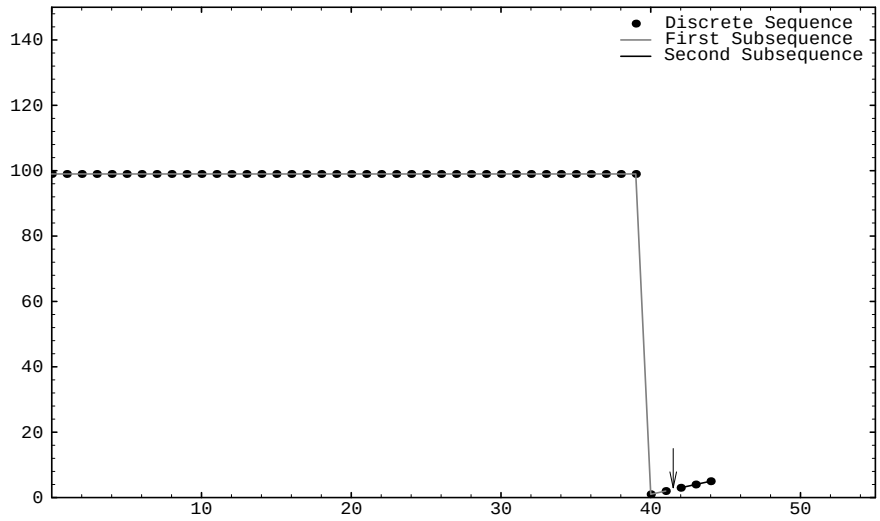

Figure 5.3: A sequence consisting of many 99's and ending by $1, 2, 3, 4, 5$, broken (where the arrow is pointing) into two subsequences using Variant 1.

If the error is between ϵ and 2ϵ, adjusting the polynomial is tried first. Only if this doesn't reduce the error, breaking occurs at a preceding point where the error is greater then ϵ (with respect to the adjusted polynomial).

This variant, as opposed to the former, is more sensitive to gradual changes. It better detects the place where a change in the nature of the sequence starts, and breaks the sequence there. Since it replaces polynomials only when the error indicates that it is required, it is also more efficient.

The main drawback of this variant is its insensitivity to changes if they are preceded by long sequences that fit well to one polynomial, as illustrated in Figure 5.3 ($window_size = 5$, $\epsilon = 5$). This is a result of using *mean* error together with anchoring the polynomial at the beginning of the subsequence, rather than sliding it along with the window. The polynomial $y = 99$ fits perfectly for such a long part of the depicted sequence, that the error when 1 first occurs is divided by a large factor, (the whole length of the subsequence until the value 1 occurs), thus reducing the effect of the deviation, and breaking too late (where the arrow points to in the figure). This problem does not occur when we interpolate a new polynomial for each shift of the window, since a polynomial that fits several 99's and 1, would not fit perfectly for the preceding 99's, thus increasing the mean error to better reflect the diversion.

5.1.2 Variant 2

This variant is similar to the former, in breaking up the sequence when the mean error is *much* greater the ϵ . However, instead of using just the mean error, when it is not much larger than ϵ, the algorithm also uses the *ratio* between the former error and current error. The decision of whether to break the sequence or to generate a new polynomial is based on the size of this ratio. When it exceeds a certain threshold, a new polynomial is tried even

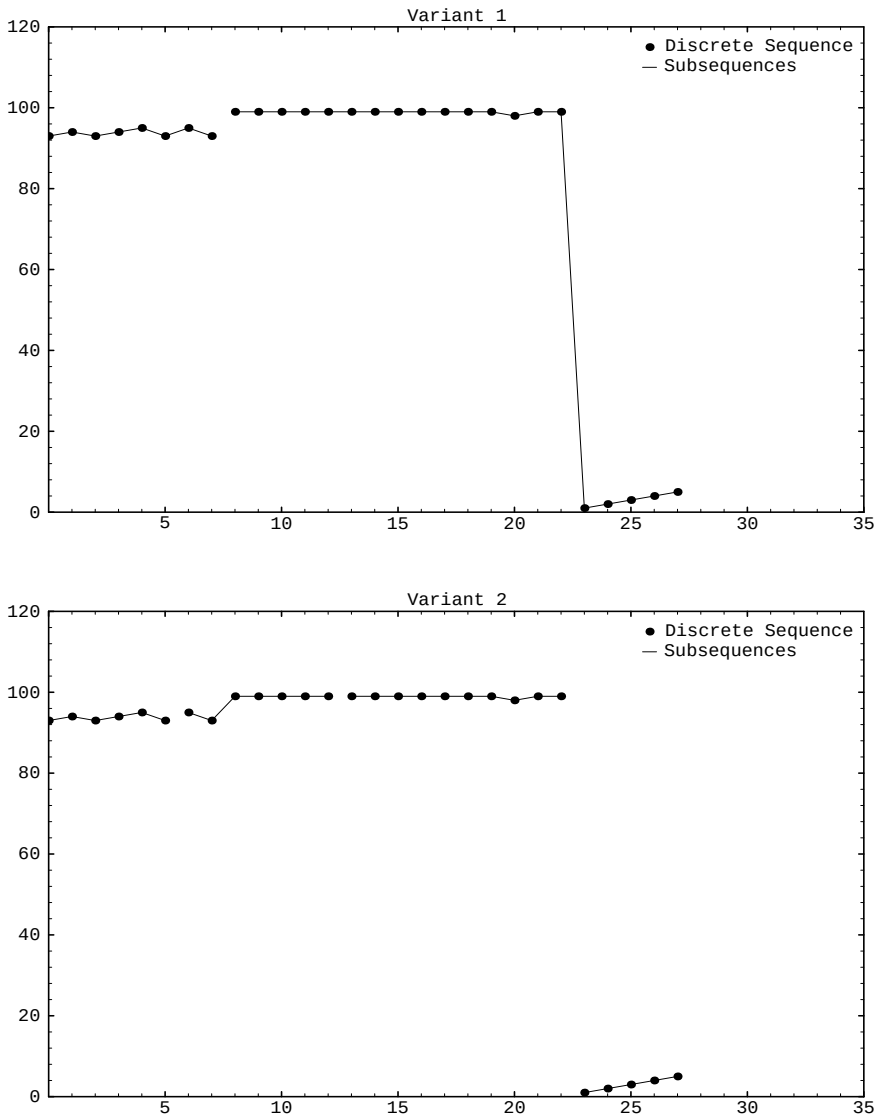

Figure 5.4: The top sequence was broken using Variant 1. The bottom sequence, using Variant 2.

if the mean error did not exceed ϵ. Figure 5.4 illustrates the difference between variants 1 and 2 in breaking the same sequence using the same *window size* (5) and ϵ (25). Variant 2 is more sensitive to changes, but the trade-off is unnecessary fragmentation. Moreover, it is not sensitive enough to changes following long perfect fits, and gives the same results as Variant 1 for the sequence of Figure 5.3. Moreover, if the mean error at a certain point is close to 0, the ratio at the next point would become close to infinity even when the deviation is small, resulting in fragmentation.

5.1.3 Variant 3

Like the former algorithm it uses an additional indication (on top of the mean error) to check if the sequence has to be broken. The indication used is the number of standard

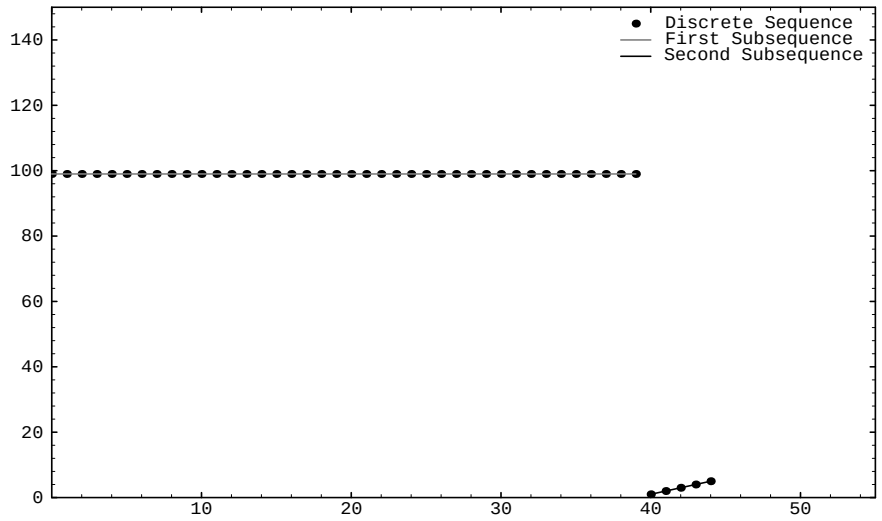

Figure 5.5: A sequence consisting of many 99's and ending by $1, 2, 3, 4, 5$, broken into two subsequences using Variant 3.

deviations of the next element from the mean of the current subsequence.

Thus, the mean and standard deviation is calculated for the subsequence after each new element is appended to it. If the next element is more than *factor*(standard deviation)* away from the mean of the previous elements (and the mean error with respect to the current polynomial is not much greater than ϵ), a new polynomial is tried, to test if the mean error reduces. If it does not – the sequence is broken at that point.

This variant improves on previous because it actually traces the behavior of the sequence so far (reflected by the mean and the standard deviation). Figure 5.5 illustrates how the algorithm breaks the sequence shown in Figure 5.3, using the same *window size* (5) and ϵ (5) as used by Variant 1, with better results.

There is still a problem when the current standard deviation, is close to 0 (or 0). In this case, every little change in the sequence's pattern is more than *factor*(standard deviation)* away from the mean (for any reasonable *factor*), and we end up breaking up the sequence unnecessarily, thus causing fragmentation and inconsistent break points.

5.1.4 Conclusion

The algorithms described so far break some sequences (such as step functions) consistently well, while not doing as well with others.

The basic weakness of the above algorithms, is that they all take a polynomial that *exactly* interpolates the points in the window, and try using it to approximate points outside the window. A low degree polynomial corresponds to a narrow window, that doesn't capture enough of the sequence behavior. This results in a large error for points outside the window, which leads to unnecessary breaking and fragmentation.

A wider window, corresponds to a high degree polynomial, that fluctuates a lot. Therefore

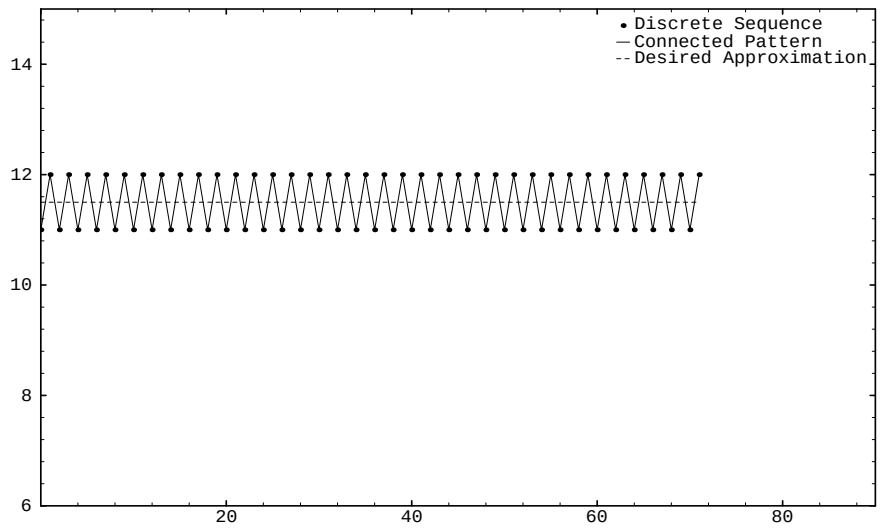

Figure 5.6: An oscillating sequence, should be approximated by the *non-interpolating* dashed line.

it deviates a lot from the sequence outside the window, which results again in breaking and fragmentation.

The oscillating sequence of Figure 5.6 is the ultimate example for the unwanted effects of interpolation. For any window size, unless the error tolerance is very large (relative to the sequence length), the sequence would be broken into fragments whose size depends on the size of the window and the error tolerance – rather than on the behavior of the sequence. The sequence behavior is the same throughout (assuming that the oscilations are smaller than the error tolerance), and would be best approximated – if we wish to avoid breaking – by a constant function averaging the oscillating values (the dashed line in the figure).

The conclusion is that we need an algorithm to generate polynomials that approximate the subsequence values, rather than match them exactly at initial points and hopefully approximate them for the rest of the points.

Part of the future work is to develop on-line algorithms to find approximating polynomials, using non-linear regression over the sliding window. The rest of this section deals with finding approximations, using *off-line* algorithms.

5.2 Bézier Curve Fitting

Bézier Curves are used in computer graphics, for representing digitized curves by parametric piecewise-cubic curves [Fo90]. Given a two-dimensional digitized curve $((x_1, y_1), ..., (x_n, y_n))$, a parametric curve representation uses an additional parameter t and represents both x and y as polynomials $x(t)$ and $y(t)$. (Rather than representing y as a function of x).

Computer-graphics techniques match our interest in queries based on "the way sequences look" (as demonstrated in section 2.1). They also generalize well to sequences other than

Bézier Curve Fitting Algorithm

begin

Global Variables:
S Sequence of points $((x_1, y_1), ..., (x_n, y_n))$
ϵ Error tolerance

1. Parameterize S
2. Fit a Bézier curve to S
3. Find point (x_i, y_i) in S with maximum deviation from curve.
4. If deviation $< \epsilon$ $return(S)$.
5. Else {
   (a) Fit a Bézier curve to the subsequence ending at (x_{i-1}, y_{i-1}), S'.
   (b) Fit a Bézier curve to the subsequence starting at (x_{i+1}, y_{i+1}), S''
   (c) If (x_i, y_i) is closer to the curve obtained in (a) make (x_i, y_i) the last element of S'
   Else make (x_i, y_i) the first element of S''.
   (d) Recursively apply the algorithm to S' and S''.

   }

end

Figure 5.7: Bézier curve fitting algorithm, adjusted for breaking purposes.

time-series (not *functions* of time), and to multi-dimensional sequences. Furthermore, having a graphics-oriented representation of sequences, supports the use of methods from computer graphics, such as the multi-resolution analysis [FiS94], for further analysis of data. However, unlike computer graphics applications, we have no indication of where curves start/end (no "mouse clicks"), nor do we allow user interference in the breaking process. The algorithm by P. J. Schneider, adjusted for breaking instead of curve fitting, is outlined in figure 5.7[6]. It supports *fully automated* curve fitting [Sc90], and therefore is useful for our purposes.

The original algorithm imposed continuity between curves, thus associating the break point (found in step 3 of Figure 5.7) with both resulting subsequences. This leads to inaccurate breaking points when continuity is not desirable, as demonstrated in Figure 5.8. Figure 5.8a, shows the step sequence which, intuitively, should be broken into the two distinct subsequences shown. Since the algorithm attempts to make the two continuous, the result is three subsequence rather than two, the second of which is merely for the sake

[6]Implementation was available through ftp wuarchive.wustl.edu at graphics/graphics/books/graphics-gems/Gems

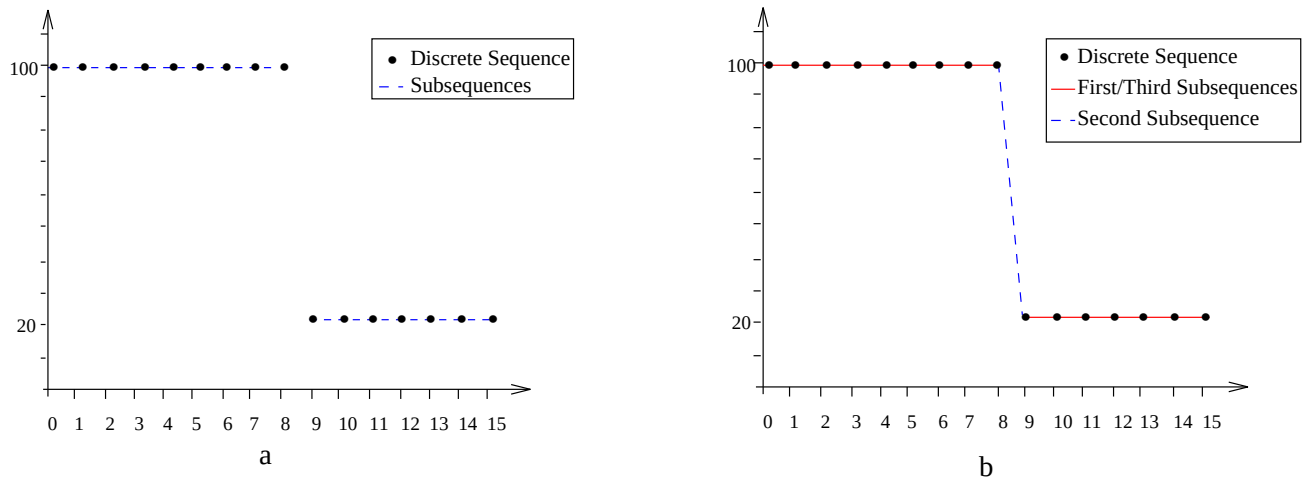

Figure 5.8: Figure *a* shows a step sequence and the desired two subsequences. Figure *b* shows how Schneider's original algorithm breaks the sequence into 3 subsequences, enforcing continuity.

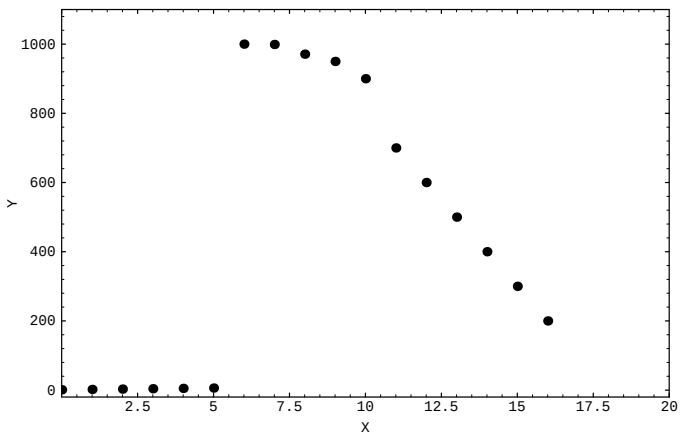

Figure 5.9: An abruptly changing sequence.

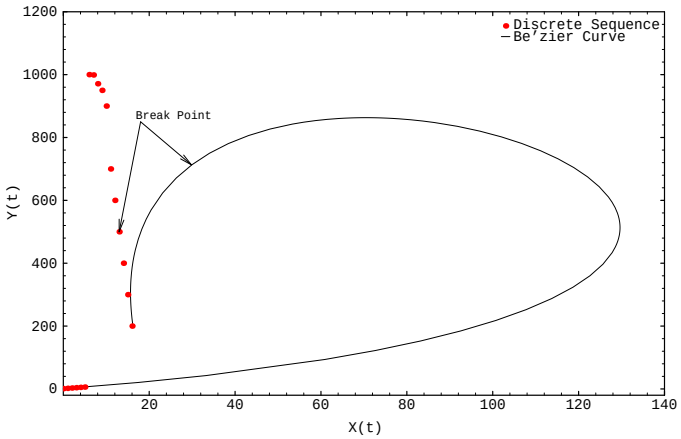

Figure 5.10: The curve for $Y(t)$ is not a function of $X(t)$. The original sequence is shown as a function of $X(t)$.

of continuity, as shown in Figure 5.8b.

A related problem is that whenever the algorithm breaks a sequence in two, it associates the breaking point with each of the two resulting subsequences, thus practically duplicating the breaking point. Our application doesn't require continuity, and we want to avoid the fragmentation resulting from subsequences formed for continuity purposes. Steps 5(a)-5(c) of Figure 5.7 are our adjustment of the original algorithm, to decide with which subsequence the break point is associated.

The main problem of the algorithm is that it doesn't operate well if a sequence changes abruptly, like the one shown in Figure 5.9.

The problem manifests itself in what we call "the balloon effect". The plotted dots in figure 5.10 denote a scaled version of the same sequence, while the solid curve in figure 5.10

(whose shape motivated the name for this effect), is the initial Bézier curve fitted to the sequence. It does interpolate the start and end points of the original sequence (for which $t = 0$ and $t = 1$, respectively). However, the parameterization is not uniform, and is done so that $t \in [0, 1]$ and the distance between t's values is proportional to the distances between the sequence points. Hence, when the sequence changes abruptly, there are large intervals of t's values for which the original sequence does not have corresponding points (x, y). Thus the resulting curve (defined for all the values of t), is far from approximating the sequence. Therefore, the maximal distance between the sequence value and the curve value, (shown by arrows to the curve and to the sequence[7]), is obtained at some arbitrary position with respect to the sequence.

Thus, the break points in the initial iterations of the algorithm can be arbitrary, causing fragmentation and lack of consistency.

A solution is to detect obvious "discontinuities", and do a preliminary breaking of the sequence before using the above algorithm for finer breaking. "Discontinuity" is not well defined for discrete sequences, and we developed a method to detect abrupt "jumps" in sequences, based on slope ratios. This improves the performance. Still, setting an exact ratio of slopes that justifies breaking, is not simple and varies a lot with the specific values of the sequences.

The benefit of early detection of major "jumps" is that it turns the algorithm into an on-line algorithm, that can be applied to subsequences, rather than to the whole sequence. This technique is beneficial for any other off-line algorithm as well.

We have also tried to simplify the algorithm by reducing it to a single dimension. The motivation was to better support time series, which are one dimensional given that the sampling rate is constant. The one-dimensional algorithm, though simpler, gives rise to

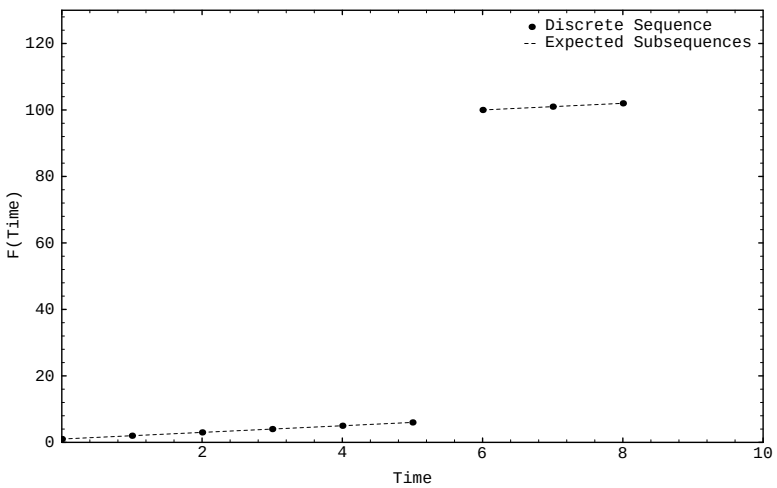

Figure 5.11: The sequence: *[1, 2, 3, 4, 5, 6, 100, 101, 102]* should be broken as shown.

another problem. Since the t parameterization, mentioned previously, is non-uniform on the interval $[0, 1]$, it can be done so that a single polynomial would still fit diverse values of the sequence, thus avoiding the need to break the sequence into subsequences. Consider,

[7]Note that the scale for *X(t)* is not the same as for *Y(t)*.

for instance, the sequence depicted in Figure 5.11. Using a very dense parameterization that is very close to 0 for the first 6 points and very close to 1 for the last 3, as shown in Table 5.1, results in fitting the (1-dimensional) polynomial: $101t + 1$ to the whole

Time	0	1	2	3	4	5	6	7	8
F(Time)	1	2	3	4	5	6	100	101	102
$\boldsymbol{t}$	0.0000	0.0099	0.0198	0.0297	0.0396	0.0495	0.9802	0.9901	1.0000
$\mathbf{101}\boldsymbol{t} + \mathbf{1}$	1.0000	1.9999	2.9998	3.9997	4.9996	5.9995	100.0002	101.0001	102.0000

Table 5.1: The parameterization t and Bèzier polynomial values for the sequence of Figure 5.11.

sequence. Thus, the need to break the sequence is avoided. This contradicts our intuition that the sequence should be broken into two distinct subsequences.

To conclude the above discussion, using the two-dimensional algorithm is preferable to the one dimensional. In order to use two-dimensional representation of time series, we view time as $X(t)$, while the sequence value at the time corresponds to $Y(t)$. To avoid undesirable breaking points, and to accommodate online processing, we can apply pre-processing for the preliminary detection of discontinuities. Filtering the sequence before applying the curve fitting algorithm would also improve the results.

5.3 Linear Curve Fitting

A simpler version of curve fitting, which does not require parameterization (step 1 of Figure 5.7), is the use of linear functions as curves. We use the same basic algorithm as the one given in Figure 5.7, but rather than using Bézier curves in steps 2 and $5a, b$ we use the linear regression line or linear interpolation through the endpoints.

The **Linear Regression** line, $Y = aX + b$, is obtained for a sequence S of n elements (x_i, y_i) using the following formula [Sp61]:

$$a = \frac{n\sum_{i=1}^{n}(x_i y_i) - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$$

$$b = \frac{(\sum_{i=1}^{n} y_i)(\sum_{i=1}^{n} x_i^2) - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} x_i y_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$$

This is the *least square error* line for the sequence S. Using the regression line as the fitted curve, can be problematic for breaking purposes. It causes fragmentation when there is a very long subsequence with values that are very close to each other (averaging around some value v), and a much shorter sequence with values which are very far from v, as illustrated in Figure 5.12, where $v = 5$. The long subsequence is denoted by V, while the short one is denoted by W. Since the regression line is the least square line, it runs very close to the values around 5 (denoted by V). Thus, the leftmost values of W are farthest from it, and this is were the first break point occurs. This is true for each of the successively formed regression lines, causing fragmentation of the subsequence W. The unwanted break points are pointed to by arrows in the figure.

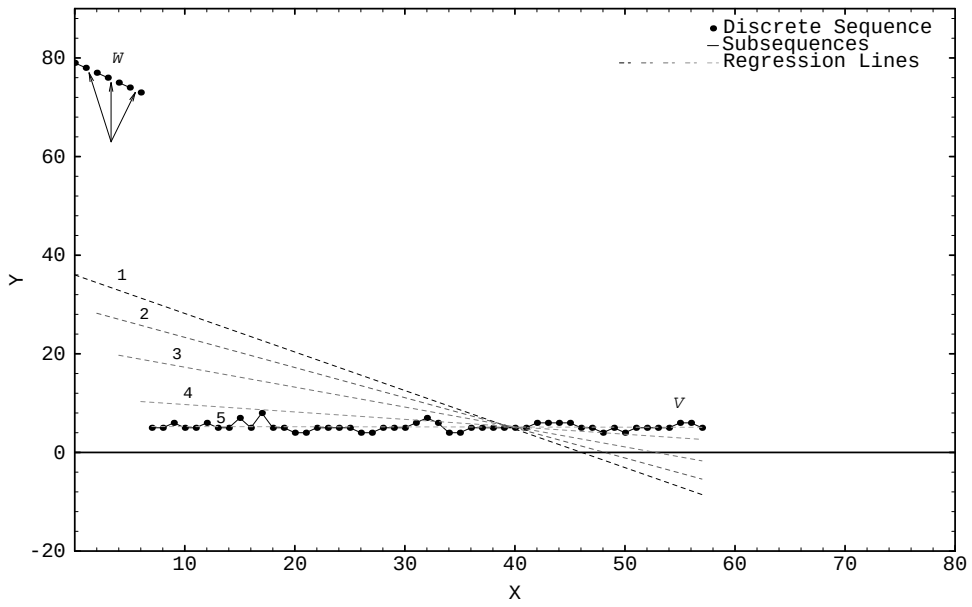

Figure 5.12: Fragmentation of the subsequence denoted by W, caused by regression lines being close to the long subsequence denoted by V.

Linear Interpolation through the endpoints of the sequence, is simpler and produces much better results. The *linear interpolation* algorithm takes as its curve the line interpolating the endpoints of the sequence, and effectively breaks sequences at extremum points. The intuitive explanation is that by passing a non-vertical line through the sequence, extremum points are further away from it than others. Thus the point most distant from the line is either some maxima point above it or a minima point below it. Due to the recursion step (Figure 5.7, step 5(d)), the extremum point becomes one of the endpoints in the next iteration, thus points close to it would be close to the interpolation line, and therefore there is *no fragmentation*, unless it is justified by extremely abrupt changes in the sequence's value. Hence the algorithm is *robust, consistent* and *avoids fragmentation.* Another advantage of the algorithm is that finding an interpolation line through two points does not require complicated processing of the whole sequence. Only end points need to be considered in order to generate the line. The next section demonstrates the effectiveness of this technique.

5.4 Linear Interpolation on ECGs

We already demonstrated the applicability of our linear interpolation program for breaking sequences in the context of the goalpost fever query, on data we have generated ourselves. We also tested our program on actual digitized segments of electrocardiograms[8] and the results are demonstrated in Figure 5.13.

Such breaking is useful for real queries of the form "Find all ECG's with distance $n \pm \delta$ between every two consecutive peaks", where δ is the error tolerance on the distance. It

[8]The segments of ECGs are available through WWW, http://avnode.wustl.edu

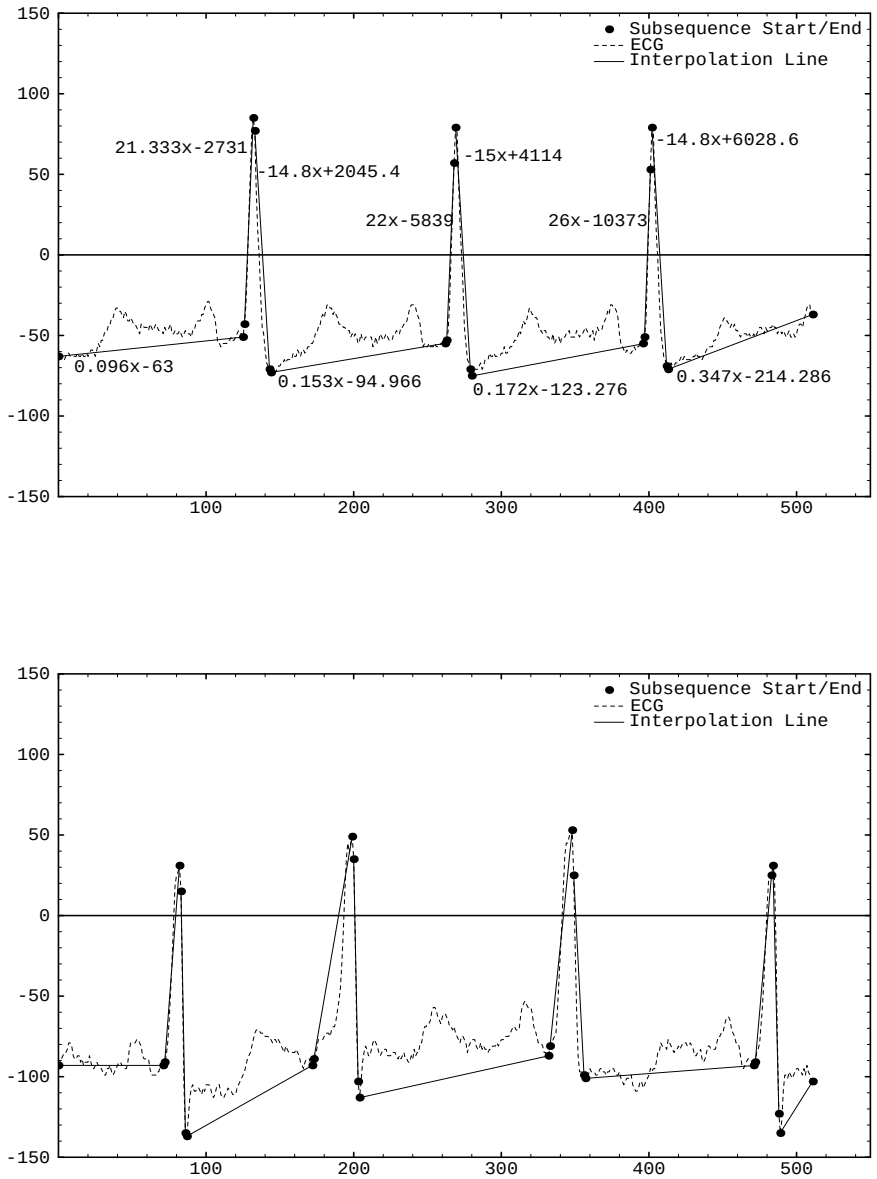

Figure 5.13: Two ECG segments of 512 points each, broken by our algorithm. The distance parameter (ϵ of Figure 8) was set to 60.

Peaks	Rising Function	RStart	REnd	Descending Function	DStart	DEnd
1	21.333x-2731	(126,-43)	(132,85)	-14.8x+2045.4	(133,77)	(143, -71)
2	22x-5839	(263,-53)	(268,57)	-15x+4114	(269,79)	(279, -71)
3	26x-10373	(397,-51)	(401,53)	-14.8x+6028.6	(402,79)	(412, -69)

Table 5.2: Peaks information for the top ECG on Figure 5.13

can be answered easily, if we have an index based on the time elapsed between peaks. Building such index is reasonable if this kind of queries is common, and can be done as follows:

1. Find the peaks in the sequences. This can be done while storing the data as part of the preprocessing – by examining the slopes of the representing functions. If data is already stored, an (already existing) index structure such as a position tree [AHU74], that finds all subsequences of the form $(+\phi)(-\phi)$ (as defined in Section 4.4) and returns their positions, can be used.

2. Start and end points of subsequences are part of the information obtained from the breaking algorithm, and are maintained with any representation of the sequence. Hence, a table like Table 5.2 can be constructed for each sequence, in which peaks are found. The table contains for each peak the approximating functions with the positive and negative slopes, and the start and end points of the respective subsequences approximated by those functions. Each point is a pair of the form $(time, amplitude)$.

3. For each peak, compare the amplitude at the end point of the rising subsequence (REnd) with that of the start point of the descending subsequence (DStart). The one with the larger *amplitude* is where the peak actually occurred. Keep *time* for that point.

4. For each pair of successive peaks, find the difference in time between them. The result is a sequence of distances between peaks. Enter this sequence into an index structure supporting sequences, (like the ones mentioned in Section 4.4).

Figure 5.13 illustrates the efficiency of representation that is obtained by our technique. 512 points sequences are represented by about 12 function segments. Assuming each representation requires 4-6 parameters (such as function coefficients and break points), we get about a factor of 10 reduction in space.

The "peak distance" query is a *generalized approximate query* by the criteria set in section 2.2:

1. The pattern characterizing the desired results is: "having distance exactly n between peaks".

2. The query defines a set S of all electrocardiograms that have this property, regardless of their explicit values.

3. The set is closed under transformations that preserve the distance between peaks, such as shift in time or in amplitude of the whole sequence.

4. A result is an *exact match* if the distance between its peaks is *exactly* n.

5. A result is an *approximate match* if the distance between its peaks is within δ distance from n. If we allow the slopes for what's considered a peak to be within some error around ϕ then we would get another dimension of approximation – with respect to "peakness".

Thus, the example demonstrates the ability of our technique to facilitate queries unsupported within any other framework, to reduce significantly the length of scanned sequences and to support search by using indices that support sequences.

It is important to note that our breaking algorithms produce as a by-product functions that approximate the subsequences. In some cases, the function is a good representation of the subsequence, while in other cases some other representation is required. For instance, in the example given in the previous section, the by-product functions were interpolation lines, but the ones used for representation were regression lines. It is possible that with some adaptation (see section 6) by-product functions can be a good representation.

It should be pointed out that since functions representation is quite compact, it would be possible to compute and store multiple representations and indices for the same data. This would be useful for simultaneously supporting several common query forms.

6 Conclusions and Future Work

We have presented a new notion of approximation, which is required in application domains that are new to the database world, such as medicine, experimental sciences, music and many others. The data in these domains consist of very long sequences. We address two basic needs of these new domains. The need for queries that are based on patterns of behavior rather than on specific values, and the need to reduce the amount of stored and scanned data while increasing the speed of access.

We analyzed the meaning of the first of those needs, and formally introduce generalized approximate queries. Such queries are not facilitated within any other framework.

We introduced a general approach that addresses the two needs simultaneously. Our "divide and conquer" approach, is based on breaking sequences into meaningful subsequences and storing an approximate representation of each subsequence as a mathematical function. Since a function's behavior can be captured by properties of the derivatives, indexing can be done according to such properties. Queries that specify sequences with certain behavior can be transformed in the same way, and can be matched to the appropriate sequences using such indices.

We presented several algorithms for breaking sequences, and demonstrated the applicability of our new approach for solving real problems in medical applications. Our method also reduces the amount of data to be scanned for answering such queries.

Future work includes:

- Continue applying our approach to real problem domains, and test how well we can perform given a variety of features and required query forms.
- Preprocessing the sequences before and after breaking, using filtering, compression, multiresolution analysis and normalization. For instance, normalizing sequences to have mean 0 and variance 1, eliminates differences between sequences that are linear transformations (scaling and amplitude shifting) of each other.
- Defining a query language that supports generalized approximate queries. One of the options is to use a visual query language in which the user draws the shape of the sequence he/she is looking for, points out the important dimensions for comparison, and specifies error tolerance in each dimension. For an underlying query language, constraints logic programming, (as discussed in Section 3), may prove to be a good choice. Constraints involving arithmetic predicates can express queries that are not supported in traditional query languages, and can describe general properties of sequences. Therefore we may be able to use them for expressing generalized approximate queries.
- Address efficiency considerations. The (off-line) transformations, from sequences to function representation, need not be very efficient as long as they can be done within a reasonable time. More important is how efficient access and storage can get once the data is stored in the approximated format.

Acknowledgments

I wish to thank my advisor Stan Zdonik for his guidance, John Hughes for the introduction to Bézier curves and other graphics techniques, Leslie Kaelbling for pointing us in the direction of on-line breaking algorithms, Jon Elion for background on medicine related data problems, Karen Fischer and John Weeks for sharing the data problems of seismology with us, and Swarup Acharya, Catriel Beeri, Vasiliki Chatzi, Dimitris Michalidis, Christopher Raphael, and Bharathi Subramanian for their help and advice.

References

[AB94] W. G. Aref, D. Barbará, *Indexing Multimedia Data Streams*, Proceedings of the ACM Multimedia '94, Conference Workshop on Multimedia DBMS, San Francisco, California, Oct. 1994, pp. 73-77.

[ABK94] W. G. Aref, D. Barbará and H. F. Korth, *Handwritten Databases: Ink as a first-class database type (Extended Abstract)*, Technical report, Matsushita Information Technology Laboratory, Princeton, NJ, October 1994.

[ABV94] W. G. Aref, D. Barbará and P. Vallabhaneni, *The Handwritten Trie: Indexing Electronic Ink*, Technical report, Matsushita Information Technology Laboratory, Princeton, NJ, October 1994.

[AFS93] R. Agrawal, C. Faloutsos, A. Swami, *Efficient Similarity Search In Sequence Databases*, FODO Conference, Evanston, Illinois, October 1993.

[AHU74] A. V. Aho, J. E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison & Wesley, 1974.

[BJM93] A. Brodsky, J. Jaffar, M.J. Maher, *Toward Practical Constraint Databases*, Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993, pp. 567-580.

[ChS94] A. Chatterjee, A. Segev, *Approximate Matching in Scientific Databases*, Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, 1994, pp. 448-457.

[FiS94] A. Finkelstein, D. H. Salesin, *Multiresolution Curves*, Computer Graphics Proceedings, Annual Conference Series, SIGGRAPH 94, Conference Proceedings, 1994, pp. 261-268.

[Fi93] K. Fischer, Dept. of Geochemistry, Brown University, Private conversation, October, 1993.

[Fo90] J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics Principles and Practice*, The Systems Programming Series, Addison and Wesley, 1990, pp. 478-507. .

[Fr60] E. Fredkin, *Trie Memory*, Communications of the ACM, Vol. 3, No. 9, September 1960, pp. 490-500.

[FRM94] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, *Fast Subsequence Matching in Time-Series Databases*, SIGMOD - Proceedings of Annual Conference, Minneapolis, May 1994.

[GuS93] H. Gunadhi, A. Segev, *Efficient Indexing Methods for Temporal Relations*, IEEE Transactions on Knowledge and Data Engineering, Vol. 5, No. 3, June 1993, pp. 496-509.

[Ka94] Leslie P. Kaelbling, Private conversation, June 1994.

[KaG94] P. C. Kanellakis and D. Q. Goldin, *Constraint Programming and Database Query Languages*, Technical report, CS-94-31, Brown University , June 1994.

[KKR90] P. C. Kanellakis, G. M. Kuper and P. Z. Revesz, *Constraint Query Languages*, Technical report, CS-90-31, Brown University , November 1990.

[Ja91] H.V. Jagadish, *A Retrieval Technique for Similar Shapes*, ACM SIGMOD, Proceedings of the International Conference on Management of Data, Denver, May 1991, pp.208-217.

[Mo88] A. Motro, *VAGUE: A User Interface to Relational Databases that Permits Vague Queries*, ACM Transactions on Office Information Systems, Vol. 6, No. 3, July 1988, pp. 187-214.

[Mu87] D. Mumford, *The Problem of Robust Shape Descriptors*, Center of Intelligent Control Systems, Report CICS-P-40, Harvard University, Cambridge, Mass., Dec. 1987.

[Ri92] J. Richardson, *Supporting Lists in a Data Model (A Timely Approach)*, Proceedings of the 18th VLDB Conference, Vancouver, Canada, 1992, pp. 127-138.

[Sc90] P. J. Schneider, *An Algorithm for Automatically Fitting Digitized Curves*, in Graphic Gems, A. S. Glassner Ed., Academic Press Edition, 1990, pp. 612-626.

[SLR94] P. Seshadri, M. Livny, R. Ramakrishnan, *Sequence Query Processing*, SIGMOD - Proceedings of Annual Conference, Minneapolis, May 1994, pp. 430-441.

[SW94] D. Shasha, J.T. Wang, K.Zhang, F.Y. Shih, *Exact and Approximate Algorithms for Unordered Tree Matching*, IEEE Transactions on Systems, Man and Cybernetics, Vol. 24, No. 2, March 1994.

[Sp61] M. R. Spiegel, *Theory and Problems of Statistics*, Schaum Publishing Co., New York, 1961, pp. 217-242.

[Su94] B. Subramanian, PhD Thesis Proposal, Technical report, CS-94-47, Brown University , December 1994.

[SZ93] B. Subramanian, S.B. Zdonik, T.W. Leung, S.L. Vandenberg, *Ordered Types in the Aqua Data Model*, The 4th International Workshop on Database Programming Languages, New York, Septempber 1993.

[WZ94] J.T. Wang, K.Zhang, K. Jeong, D. Shasha, *A System for Approximate Tree Matching*, IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 2, April 1994.