Using Signature Files for Querying Time-Series Data

Henrik André-Jönsson[1] and Dushan Z. Badal[2]

[1] Computer and Information Science, Linköping University,
S-581 83 LINKÖPING, SWEDEN
Email: henan@ida.liu.se
[2] Computer Science Department, University of Colorado, Colorado Springs,
CO 80919, USA
Email: badal@sunshine.uccs.edu or dusba@ida.liu.se

Abstract. This paper describes our work on a new automatic indexing technique for large one-dimensional (1D) or time-series data. The principal idea of the proposed time-series data indexing method is to encode the shape of time-series into an alphabet of characters and then to treat them as text. As far as we know this is a novel approach to 1D data indexing. In this paper we report our results in using the proposed indexing method for retrieval of real-life time-series data by its content.

1 Introduction

Digitized 1D data, commonly referred to in the literature as time-series data is a sequence of real values and it can represent voice, sensor reading, history of stock prices, etc. Although we propose to investigate indexing of any sequence of real numbers we use time-series to be consistent with the literature. Time-series data are generated and stored in many applications. Examples include voice, histories of stock prices, histories of product sales, histories of engine testing, seismic data, aircraft flight recordings, weather data, enviroment data (pollution levels for various chemicals), satellite sensor data, astrophysics data, etc. Currently there is no database technology which can retrieve 1D data by their content.

2 Related Work

There has been very little work done so far in the area covered by the proposed project. We found just few recent papers dealing with time-series data indexing [FALO94, FALO95, AGRA95, AGRA95a]. Time-series data indexing described in [FALO94] is based on using first few DFT (Discrete Fourier Transform) coefficients to represent part of time-series within a window moving along the signal one step at a time. At each step the DFT is applied and this generates one point representation in 2D space consisting of the first two DFT coefficients. In this way time-series data is represented by a trail in 2D space. Such trail is then divided into sub-trails which are subsequently represented by their Minimum

Bounding Rectangles in an R*-tree which is used as an index. As pointed out in [AGRA95] the work reported in [FALO94] is not readily applicable as it ignores several problems like amplitude scaling, offset translation, etc. The indexing described in [AGRA95] is based on the shape similarity of atomic subsequencies (windows) of two signals, on stitching similar windows to form pairs of large similar subsequencies (long subsequence matching) and on finding a non-overlapping ordering of subsequence matches having the longest match (sequence matching). Both papers [FALO94, AGRA95] are deficient as they tested their proposed indexing methods on very small collection of stock market histories. Thus, it is not clear whether the indexing they propose will work on large real-life time-series database. Second, neither indexing method is data amplitude, offset or translation, and rotation invariant. This means that in order to use their indexing one has to do amplitude scaling, offset translation, etc. This makes both methods unsuitable for real-life applications.

Related paper dealing with multimedia data indexing [FALO95] avoids the hard problem of feature extraction by assuming that "domain expert" will somehow provide feature vectors or at least the easier to provide dis-similarity/distance of two objects. Given that assumption the paper then describes an algorithm which maps objects into k dimensional space so that distances (dis-similarities) are preserved as well as possible. Another paper reporting related work [AGRA95] presents a shape definition language, called SDL, for retrieving objects based on shapes contained in the histories associated with these objects. A novel feature of the language is its ability to perform blurry matching when the user cares about the overall shape but does not care about specific details.

3 Proposed Approach

Signature files have been shown to work as an index for text files, but no one, to our knowledge, has used them to index a signal. To encode the signal as a text we use the method similar to one in [AGRA95]. We construct the time derivative of the signal by calculating the amplitude difference between two adjacent samples. Depending on the difference that corresponds to the signals time derivative we map the difference to a token. By choosing a suitable alphabet for this mapping we will not only be able to use signature files as an index, but we will also get the ability to perform blurry matching as described in [AGRA95]. Our approach to 1D data indexing was motivated by our recent work on full text indexing where we introduced context signature files [BADA95].

Our approach is as follows. First, we translate the signal into a text-string. Then, we run the string through a signature file generator. The signature file generator creates a signature file from the text by sliding a window along the string and for each window it maps the string into a number of signature bits. We have made experiments letting each window set one, two, four and eight bits. The results presented in this paper are using four bits per window. The signature is stored in the signature file along with a pointer to the text block (representing 1D signal) that has been used to create the signature.

When we wish to query the system we simply calculate the signature of the 1D query and do a linear search of the signature file. In this first implementation we have decided to use a simple signature file. An important part of our preliminary work was to show that by selecting a suitable window size together with a suitable signature size the false drop rate will be minimized.

This method has several advantages over previously suggested methods for doing shape querying. The signature file is relatively small compared to other indexing methods, and as a combination of the size of the signature file and the simplicity in testing for a hit, a search through the signature file is very fast. The signature file search is a linear search in O(n) time and each check is very simple and fast, just a logical AND followed by an equality check.

4 Implementation

We wanted our system to be able to do blurry search and we wanted to see if it was possible to use signature files as an indexing method. All previous work with signature files. to our knowledge, has been done on text files. So the first thing we wanted to do was to transform our signal into some sort of text. We chose to use the alphabet suggested in [AGRA95] (a part of SDL). Even though this alphabet has a great disadvantage by defining absolute values for vague descriptions we decided the advantages with blurry matching, and the fact that it is a very simple alphabet, were greater than the disadvantages. The alphabet was rewritten so that we could easily map the signal to a stream of tokens instead. We call the new alphabet "Shape Description Alphabet", SDA. SDA is only a subset of the alphabet used in SDL but should be sufficient for describing any signal.

Symbol	Description	lvalue	hvalue
a	Highly increasing transition	5	-
u	Slightly increasing transition	2	4.99
s	Stable transition	-1.99	1.99
d	Slightly decreasing transition	-4.99	-2
e	Highly decreasing transition	-	-5

Table 1. SDA

Table 1 shows the different symbols of SDA and how they map to the symbol. The value difference between two adjacent points is calculated and the difference is mapped to the symbol where the difference fits between the symbols lvalue and hvalue. These values are domain dependent and it is necessary to conduct further examination of them in order to find optimal lvalues and hvalues.

The ability to blurry match is an important feature we would like the system to have. Blurry matching allows us to ask the system not only to retrieve all signals that look exactly like the query signal, but also all signals that are similar to the query signal. We get this ability because of the 'fuzziness' in the mapping from the signal time derivative to the SDA alphabet. Assume we have the fol-

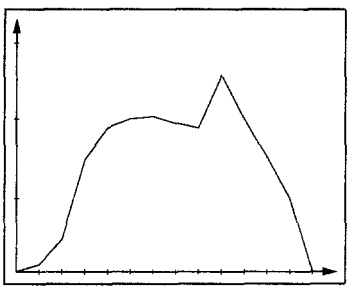

Fig. 1. A Sample Signal.

lowing sampled signal (see Fig. 1): "0 2 6 14 19 20 21 18 17 25 20 15 10 0". This sequence will translate into "uuaussdsaeeee". But there are many similar signals that would translate into the same SDA string. If we make a reverse translation of the SDA string "uuaussdsaeeee", we see that we can get the following signals (see Fig. 2) "4 8 25 28 27 26 22 21 30 20 10 0" or "12 14 19 22 23 24 22 22 28 22 17 12".

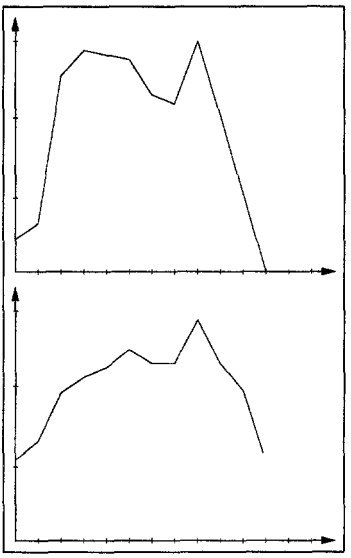

Fig. 2. Two other signals that can be derived from the same SDL string.

SDA is a nonexact approximation of the signal time derivative. This makes the SDA representation of the signal invariant to amplitude. SDA is very easy to use as a query language just because it uses so few symbols. If we want to query the system for a signal, we do not have to remember exactly how the symbol looked like but we can describe the signal very easily like "The signal

was pretty stable for a while then it increased very rapidly before being stable again". We do not have to use any absolute values. The results reported in this paper we were using an SDA alphabet which has seven characters instead of five characters shown in this section.

4.1 Finding Data Across Signature Block Boundaries

One problem we needed to address was the problem of sequences that lie between two signature blocks, ie one part of the query signature can be found in one signature and the other part in the following. The design of signatures files make it easy to find a segment that is a part of one signature, but we must to be able to find sequences that lies between signature blocks.

We have looked at two methods for solving this. The first way is to let the signature blocks overlap each other. The second approach is to define a new 'hit' mechanism when we are searching the signature blocks for a hit.

The first solution we tried was to make the signatures overlap. Whenever we have filled a signature block, we go back in the signal so that the next signature block will start at the middle of the previous signature block. By doing this the new signature block will overlap the previous signature block. There are two major drawbacks with this method. The first is simply that it takes more space to store the extra signatures needed to index the file. The second is a bit more serious as it puts a restriction on how large a query can be. The query length can never exceed the length of a half signature block (the overlap).

The second approach is complex. The basic assumption is that a query will span several signatures, i.e., the query is not a single signature but a signature file. The idea is to introduce a concept of similarity between signature files instead of between signatures. We start by making the observation that a query segment (a string of tokens) can either be found in a signature block or between two signature blocks.

The first thing we do is to check if the query segment is present in the stored signature block S_i. If it is not present we continue by checking if it is present in S_{i+1}. If it can't be found there we logically OR the two stored signatures together, $S = S_i \ OR \ S_{i+1}$ and then we check the combined signature for a hit.

If we get a hit in any of the cases above we just continue to check consecutive blocks and if all signatures in the query match consecutive stored signatures we have a hit. In the last case, the combined signature, the next stored signature that is checked with the second query signature is $S = S_{i+1} \ OR \ S_{i+2}$.

We implemented and tested both approaches. The first method limits the size of query but it is easier to implement. The second method does not limit the query size but it is more complex to implement and it makes the search somewhat slower. The results reported in this paper were achieved using the second method.

4.2 The Signature File Size

Let's examine the signature files size ratio to the signal size. Assume that we have an SDA encoded signal consisting of T data points. If we want to calculate how many signatures will be needed to index this file we can use equation 1.

$$S = \frac{T * b}{N_{sb} * F_g} \tag{1}$$

N_{sb} is the number of bits in the signature, F_g is the fillgrade (how many bits in the signature we use before we consider the signature full), S is the number of signatures needed to index the file (worst case), T is the number of tokens in the file to be indexed and b is the number of bits each window sets in the signature.

The formula gives us a very good approximation of how many signatures will be needed to index the signal, if the window size is small compared to T.

To calculate the size of the signature file we assume that for each signature block we need to store not only the block signature but also a pointer to the part of the signal that the signature represents. The size of each signature block will then be the bit map of the signature plus the pointer to the signal.

By multiplying the result in equation 1 with the size of each signature we get the total size needed for the index and finally if we divide this value with the total size of the data file we get the compression rate using equation 2. The compression is the size of the index compared to the size of the original data file. We then subtract this value from 1 so that a compression of 0% indicates that the index is of the same size as the data file.

$$Compression = 1 - \frac{T * b * (\lceil \frac{N_{sb}}{8} \rceil + P_{size})}{T * R_{size} * N_{sb} * F_g} \tag{2}$$

P_{size} is the size of a pointer in bytes and R_{size} is the size of each element in the data file in bytes. Equation 2 can then be simplified and we get equation 3.

$$Compression = 1 - \frac{b}{R_{size} * F_g} * (\frac{\lceil \frac{N_{sb}}{8} \rceil}{N_{sb}} + \frac{P_{size}}{N_{sb}}) \tag{3}$$

From this formula we can see that the compression rate consists of two parts. The first part isn't influenced by the signature size but is determined by the size of the elements in the data file, the number of bits each window sets in the signature and the fill grade (and if the number of used bits in the signature is a multiple of eight). This can easily be seen if we assume that N_{sb} is divided by 8. Then we can simplify the expression even further. We can also see that the window size doesn't influence the index compression rate.

We also need to make an assumption about the size of the original data file. In this paper we will assume that the data is stored as 32-bit numbers, ie $R_{size} = 4$. We will also assume a pointer size of 32 bits, $P_{size} = 4$, that gives us an adress space of 4 billion points. This can be considered sufficient for many applications.

We can now plot the different compression rates for different combinations of signature sizes, different values of b and F_g. Here we just give the values for

$b = 1$. Table 2 show how the compression varies with the signature size and the fill factor F_g.

$b = 1$	$N_{sb} = 61$	$N_{sb} = 251$	$N_{sb} = 509$
$F_g = 0.3$	83.7%	88%	88.9%
$F_g = 0.4$	87.7%	91%	91.7%
$F_g = 0.5$	90.1%	92.8%	93.3%

Table 2. Compression for $b = 1$.

One thing to remember is that this is the size of the uncompressed signature file. There exists several techniques to compress the signature file but so far we did not investigate this issue.

5 Experiments

In this section we will describe how the system was tested. We have tested the system on both real signals and randomly generated signals. We made several different experiments where we changed three parameters, the size of the signature window, the size of the signature and the number of bits each window set in the signature. We have *not* tested the system time performance. This system was only implemented to see if it were possible to use signature files to index a time series data.

In the first experiments we used a window that was only three characters wide. This leads to problems since such a small window doesn't give the hash function enough information to create a proper signature. There is a threshold at about five characters with the hash function we use. If the window is smaller than five characters the hash function does not operate properly and if the window is more than five characters the hash function generates nice signatures. As the window size increases we get larger and larger segments of data represented by each bit in the signature and as a consequence the signature blocks increase in size as well.

In general, we want the signature blocks to be so large that it is sufficiently faster to search the signatures instead of the signal but at the same time we want the signatures to be so small that we do not have to employ costly indexes to search a signature block to find out if we have a true hit or if we had a false drop.

To test the performance for each window size and signature size we six different queries to the test database. The queries were segments from the time

series signal used to generate the test database. When the result from the system was returned the database was sequentially scanned for the query so we could determine if the signature method had missed an occurrence of the query pattern. Then the false drop rate was calculated by comparing the sequential search result with the signature search result. The false drop rate and the result from the sequential search were used to calculate the precision diagrams in this section.

We used three sizes of signatures, 61 bits, 251 bits and 509 bits, three window sizes (7 characters, 9 characters and 20 characters) and four different settings of b, number of signature bits set per window. In this paper we only report our results with 251 bits signatures, 509 bits signatures and 1 signature bit and 4 signature bits set per window.

6 Test Results

All test diagrams have been normalized so that they can be compared with each other. On the vertical axis we show the precision. The precision is a measurement of the efficiency of the query search and is calculated as the percent equivalent to the number of relevant documents returned divided by the number of documents returned. Another diagram often used together with precision diagrams is a recall diagram. The recall measures the effectiveness of the query search by reporting the percentage equivalent to the number of relevant documents returned divided by the number of relevant documents in the collection. In all experiments we have a recall of 100% so these diagrams are not shown. The horizontal axis shows how many bits we have in each query. It is difficult to change the number of bits in the query since it is the hash function that decides what bit each window is mapped to. Therefore, we show how many windows have been used in generating the query signature.

We just show two extremes here, a 251 bits signature with a 7 character window and a 509 bits signature with a 20 character window. For both cases we show how the precision is influenced by if we let each window set 1 or 4 bits in the signature.

As we can see in Fig. 3 and Fig. 5 we get very good precision even for small queries if we let each window set several bits in the signature. As can be seen in Fig. 4 and Fig. 6 the size of the window and the size of the signature has a much larger influence if we only set 1 bit in the signature for each window.

To summarize our tests we can say that if we set few signature bits with each window we need larger signatures to get better precision for smaller queries. In this case we need larger window sizes to get more uniform behaviour for different queries.

If we use more signature bits per window the need for larger signatures and windows is not that obvious. The big drawback is of course that the compression decreases.

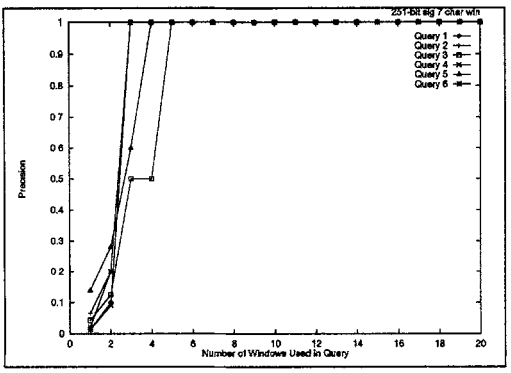

Fig. 3. 251-bit signature and 7 character window, 4 bits per window.

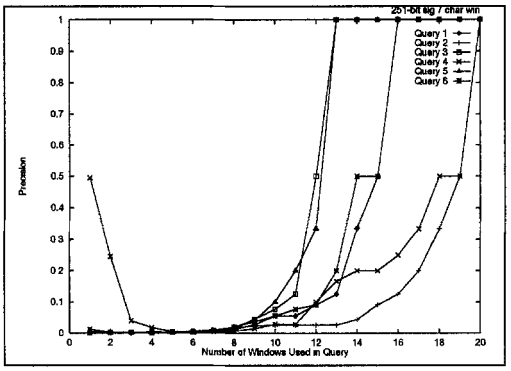

Fig. 4. 251-bit signature and 7 character window, 1 bit per window.

7 Conclusion

We have shown that it is possible to translate a signal to a string in such way that the string can be used to search the signal for behaviour of the original signal. We have chosen a very simple alphabet, SDA, for that mapping. By allowing the mapping to be somewhat blurry we gain the ability to do blurry matching on any given signal. This makes it possible to retrieve all signals that are similar to a given query sequence. We have also shown that by using a signature size of 251-bits or more and a window of at least 20 characters or by setting several bits for each window we can get very good precision performance from the system. The number of bits per set per window has a strong influence on the precision but has also a strong influence on the compression. Thus some compromise has to be found.

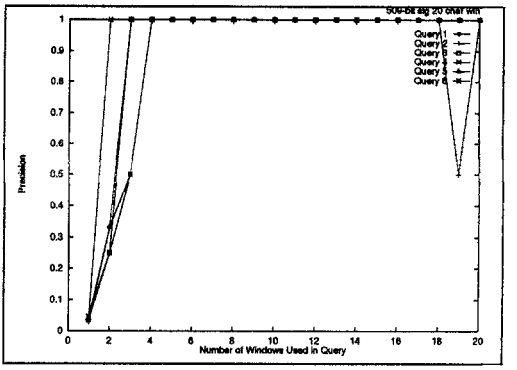

Fig. 5. 509-bit signature and 20 character window, 4 bits per window.

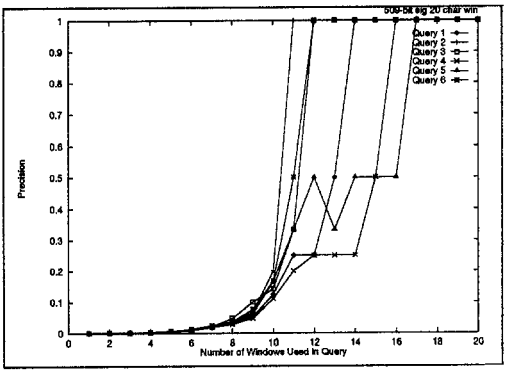

Fig. 6. 509-bit signature and 20 character window, 1 bit per window.

8 References

[**AGRA95**] Agrawal, R., et al., "Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases", Proceedings of the 21st VLDB95 Conference, Zurich, 490-501.

[**AGRA95a**] Agrawal, R., et al., "Querying Shapes of Histories", Proceedings of the 21st VLDB95 Conference, Zurich, 502-514.

[**BADA95**] Badal, D.Z. and Davis, M.L., "Investigation of Unstructured Text Indexing", Proceedings DEXA95 Int Conf on Database and Expert Systems, London, Sept 4-8, 1995, 387-396.

[**FALO94**] Faloutsos, C., et al., "Fast Subsequence Matching in Time-Series Databases" Proceedings of ACM SIGMOD94 Conference, Minneapolis, 419-429.

[**FALO95**] Faloutsos, C., et al., "FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Data", Proceedings of ACM SIGMOD95 Conference, San Jose, 163-174.