Journal of the Nigerian Association of Mathematical Physics
Volume **13 (November, 2008), 281 - 288**

Dynamic hashing algorithm for searching time-series databases

Tola John Odule,
Department of Mathematical Sciences
Olabisi Onabanjo University, Ago-Iwoye, Nigeria

Abstract

Sequence data searching using the Euclidean distance between two sequences as the only criterion for similarity suffers from some deficiencies: It is sensitive to the vertical positions of the two sequences and so is not a good measure of similarity in terms of their shapes. We propose a new definition for similarity that overcomes these deficiencies. A fast searching algorithm based on dynamic hashing, which guarantees that no qualified data subsequence similar to the query sequence will be falsely rejected, is also proposed. The algorithm can also find data subsequences similar to the query sequence with different scaling factors in both amplitude and time dimensions. Several experiments were performed to evaluate the proposed algorithm using both synthetic and real data (stock price movement).

Keywords: *Euclidean distance, hashing, time-series, sequence similarity, statistical noise.*

1.0 Introduction

There is an increasing demand for searching subsequences similar to a given pattern in a time-series database. The problem is stated as follows: Given a collection of data sequences $D_1,D_2,..,D_n$ and a query sequence Q, we want to find all the data subsequences of D_i $(1 \le i \le n)$ that are similar to Q. The technique for sequence data searching can be used in a wide range of scientific and business applications [1, 2, 3]. We cite an example below:

In a stock market, stock prices are recorded in time series. In technical analysis theory [4, 5], the occurrence of a certain pattern in the shape of a sequence such as the one called *head and shoulders* or *double tops and bottoms* is a signal of the reversal of a stock price. Thus, a stock market analyst may ask "find all the companies whose stock prices are similar to a given query sequence in shape" in order to predict the future trend of the stock price. Moreover, **time-series** searching techniques can be used in data mining [6]. Queries such as "find all subsequences from the time-series database which are similar to the query sequence in shape" may help to discover new knowledge.

1.1 Problem definition and formulation

The dissimilarity of sequences is usually measured by the *Euclidean distance* [1, 2]. Two sequences are said to be similar to each other if the Euclidean distance between them is less than or equal to a user-specified error bound. However, the problem of Euclidean distance is that it does not measure the dissimilarity of sequences by their shape directly. If the error bound is not large enough, two sequences that are exactly the same in shape but with different vertical positions may be classified as dissimilar. However, if the error bound is set larger, dissimilar sequences will be reported more easily. It is also mentioned in [3, 7, 8, 9] that other distance measures are needed for different applications.

Consider the query sequence Q: (5, 10, 6, 12, 4) and the two data sequences *A:* (6, 5, 7, 10, 11), B: (15, 20, 16, 22, 14) in Figure 1.1 and assume that the error bound ε is set to 15. Note that B is produced by shifting Q upward for 10 units. Using Euclidean distance as the distance function, we have Distance (Q, A) = 8.9 and Distance (Q, B) = 22.4. Thus, by the similarity definition of Euclidean distance, A is reported as similar

e-mail: tee_johnny@yahoo.com

to *Q*, but *B* is rejected. However, *A* is not similar to *Q* in shape. On the other hand, *B* is definitely similar to *Q* because they are of the same shape. From this example, we notice that Euclidean distance is not a good measure of dissimilarity when shape is the principal consideration. The matching results are easily affected by the vertical positions of the sequences.

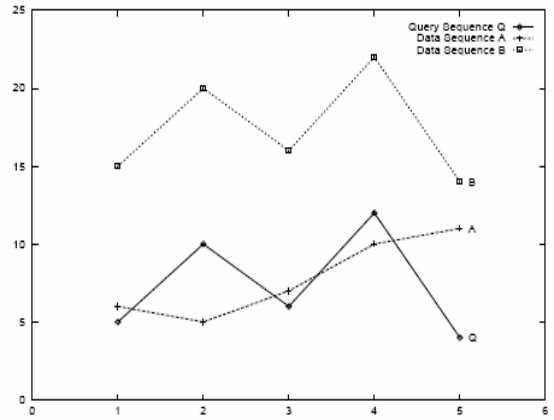


Figure 1.1: Deficiency of the similarity definition by Euclidean distance.

In this paper, we suggest the use of the *gradient* of data sequences as the criterion for similarity. Two sequences are said to be similar if the gradient difference for each pair of corresponding segments is bounded by a predefined threshold. Since the directional information of sequence data is taken into consideration, the dissimilarity between sequences in terms of their shape can be measured. Moreover, the results will not be affected by the vertical positions of the data sequences. Thus, this definition is simple yet powerful enough to meet the requirements of many applications, where the shape of the data sequences is the main concern of the similarity. In addition to the new definition of similarity, a hash-based algorithm for time-series searching is proposed in this paper. Given a query, the algorithm can efficiently search for subsequences which are similar to the query in shape with any scaling factor. The algorithm also guarantees that no qualified subsequence is falsely rejected.

2.0 Sequence similarity

Our definition of sequence similarity is motivated by the properties of similar triangles. The side-angle-side similarity theorem states that two triangles are similar if two pairs of corresponding sides are proportional and the included angle is equal. This theorem can be applied in sequence data searching if we use the slopes of segments to measure the similarity of two sequences.

A data sequence *D* of length n is an ordered set of *n* real numbers. D_i denotes the *i*th data of *D*. $D_i, D_{i+1},\ldots,D_j$, $1 \le i < j \le n$, denotes the subsequence of *D* from data *Di* D_j and the index of this subsequence is defined to be *i*. A segment of a sequence is a line joining two consecutive data points in the sequence. The gradient of a segment formed by data D_i, D_{i+1}, is $(D_{i+1} - D_i)/((i + 1) - i)$ i.e. $D_{i+1} - D_i$. An example of data and query sequences is shown in Figure 2.1.

Two segments D_i, D_{i+1} and Q_i, Q_{i+1} are said to be similar if their gradient difference satisfies the following equation:

$$- \varepsilon \le (D_{i+1} - D_i) - (Q_{j+1} - Q_j) \le \varepsilon \qquad (2.1)$$

where ε is a user-defined threshold. In addition, two sequences with the same number of data points are said to be similar if the gradients of their corresponding segments are bounded by ε. It is formally stated in Definition 2.1.

***Definition* 2.1**

Two sequences $D_1 D_2 \mathrm{K}\ D_m$ *and* $Q_1 Q_2 \mathrm{K}\ Q_m$ *are gradient similar if*

$$-\varepsilon \le (D_i - D_{i-1}) - (Q_i - Q_{i-1}) \le \varepsilon \quad \forall i\colon 2 \le i \le m \qquad (2.2)$$

Moreover, two sequences can also be similar with a scaling factor. The definition of scaling similar is formally stated in Definition 2.2.

***Definition* 2.2**

Two sequences $D_1 D_2 \text{K}\ D_{sm}$ *and* $Q_1 Q_2 \text{K}\ Q_m$ *are linear scale similar with a scaling factor* s *if for all* i, $2 \le i \le sm$, *one of the conditions below is satisfied:*

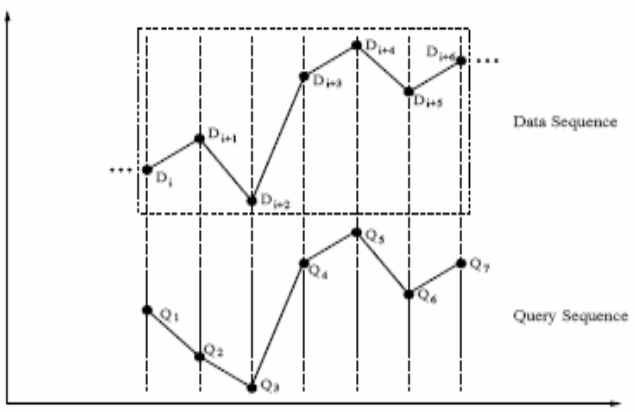


Figure 2.1: Example of a database sequence and a query sequence.

$$-\varepsilon \le (D_i - D_{i-1}) - (Q_{\lceil i/s \rceil} - Q_{\lceil i/s \rceil - 1}) \le \varepsilon, \quad s \ge 1 \text{ (scale up)} \tag{2.3}$$

$$-\varepsilon \le (D_{\lceil i/s \rceil} - D_{\lceil i/s \rceil - 1}) - (Q_i - Q_{i-1}) \le \varepsilon, \quad 0 < s < 1 \text{ } 0 < s < 1 \text{ (scale down)} \tag{2.4}$$

Note that *Definition* 2.1 is only a special case of *Definition* 2.2 in which the scaling factor is one.

3.0 Hashing

When a bucket with *I* indices is searched, in total *I* data subsequences need to be retrieved from the disk to check whether they are similar to the query sequence. It is expensive to access the disk *I* times. Thus, we present a better method to solve this problem in Section 3.4.

3.1 Dynamic hashing

The static hashing algorithm is a good algorithm for databases that are not expanding. However, in financial markets and scientific databases, new data are collected day by day. As the amount of data increases, the number of indices for each bucket increases. Consequently, the number of false alarms increases and the performance is degraded. That is the reason why we extend the static hashing algorithm to an extendable hashing algorithm [10, 11].

For extendable hashing, buckets are created on demand. As in the static hashing algorithm, a subsequence of length $k + 1$ is mapped to a feature vector of k bits. However, instead of using all k bits initially, we use i bits, $0 \le i \le k$, to represent the offset in the bucket address table. Although we are using extendable hashing, it is necessary to choose a maximum value for k. In our implementation, we choose k to be 32. Indeed, 2^{32} buckets means there are over 4 billion buckets, which is already a large number for any database. Using the Nigerian Stock Exchange Index as an example, assume each bucket contains one data point only and that the system collects one data point daily, so we need 11 million years to fill up all the buckets.

For insertion, the first i bits of the 32 bit feature vector are used to locate a bucket. The index is inserted if the bucket is not full. Otherwise, we must split the bucket and redistribute the indices. When the insertion procedure stops, each entry of the bucket address table is using i bits as an offset pointing to a bucket. For searching, the i bits determining vector with minimum unknown state bits is used to locate bucket entries in the bucket address table. Then the bucket(s) can be identified by following the pointers in the bucket entries. Data subsequences represented by the indices in the bucket(s) are retrieved and compared with the query sequence.

In real applications such as the one used to analyse stock data, deletion of data from a sequence is rare. However, it is straightforward. A data sequence can be deleted by removing all the indices of its corresponding subsequences from the buckets. The buckets that become empty are also removed.

Since the hash table only stores the indices which point to the subsequences stored externally in the disk, the size of the hash table is $O(n)$, where n is the length of data sequences. It is smaller than that of an R-tree which requires $O(nk)$ storage as mentioned before, where k is the dimension of the R -tree.

3.2 Noise class

Obviously, the distribution of indices among the buckets will affect the performance of the algorithm. In this section, we argue by the concept of noise class that the hashing algorithm will have a good distribution of indices such that it is efficient for most of the real applications.

It is mentioned in [12] that a lot of real data such as stock movements and exchange rates can be modelled by brown noise which has a skewed energy spectrum in the frequency domain. This class of noise has a property that after the sequences are transformed by discrete Fourier transform, the first few coefficients will contain most of the energy. Therefore, these first few coefficients give a good estimation of the actual Euclidean distance between two sequences [2].

However, when a sequence is represented by the gradient between two consecutive data, it means that we have taken the first derivative on the sequence. By taking the first derivative, the data sequences of brown noise will be transformed to sequences of white noise. Instead of a skewed energy spectrum, white noise has a flat energy spectrum in the frequency domain. Under this situation, the energy of the data sequences will not be concentrated on first few coefficients. Therefore, good performance will not be expected if the methods in [1, 2] are used.

On the other hand, since the energy spectrum of the white noise is flat, the indices of the hashing algorithm will be evenly distributed among the buckets. This means the expected number of data subsequences in every bucket is the same. The hashing algorithm in this case will have an optimal performance. Therefore, for most real applications, our hashing algorithm will have a better performance because of the favourable noise class of the data.

3.3 Hashing with scaling

In this section, we describe how to search for linear scale similar subsequences. First, the user specifies the range of scaling factors. For each scaling factor S within the range, the feature vector of the query sequence is scaled by S and then the k-bit determining vector is extracted from the resulting feature vector with a minimum number of bits in the unknown state. The determining vector is used as an offset to the bucket address table. All indices in the buckets pointed to by the bucket address table entries are collected but only those data subsequences satisfying *Definition* 2.1 are reported.

For instance, suppose the window size, k, is 6 and a query sequence Q with its determining vector V = 1001V is given. If the user wants to find all the similar subsequences with scaling factor 2, V is first scaled up by 2 and the bit stream 11000011 is produced. Then, 110000 is extracted and the bucket pointed to by the entry 110000 (48) is searched.

However, if the range of scaling factors specified by the user is $0 < s \leq 2$, not all scaling factors are possible for Q. For instance, Q cannot be matched with a data subsequence with a scaling factor such as 0.5 or 1.2. Thus, we need a method to calculate the set of possible scaling factors for a query from a range of scaling factors. The method is presented in the following.

Given a determining vector of a query sequence without any unknown state bit, we first construct a list containing the length of each consecutive run of 0's or 1's. For example, the list of a query sequence with determining vector 00111100 is {2; 4; 2}. Then the greatest common divisor (GCD) among the elements of the list is calculated. Assume s_l and s_u are the lower and upper bound of the scaling factors respectively and C is the GCD. The set of possible scaling factors will be

$$\left\{ \frac{i}{c} : s_l c \leq i \leq s_u c,\ i \in N \right\}.$$

For example, the list of a query Q with determining vector 00111100 is {2; 4; 2}. The GCD among the elements is 2. Thus, the set of possible scaling factors of Q is {0:5; 1:0; 1:5; 2} for $0 < s \leq 2$. For those determining vectors with some bits in unknown states, all the possible extensions will be searched. For example, determining vector 110*110* has four extensions which are 11001100, 11001101, 11011100 and 11011101. Thus, 11001100 is searched with the set of scaling factors {0:5; 1:0; 1:5; 2} while the other three are searched with the set of scaling factors {1; 2} for $0 < s \leq 2$.

3.4 Reducing disk accesses

When a bucket with I indices is searched, in total I data subsequences need to be retrieved from the disk to check whether they are similar to the query sequence. We develop a scheme to minimize the number of disk accesses. The scheme consists of *Validation Phase* and *Union Phase*, which are separately described below.

3.4.1 Validation phase

We observe that when more than one determining vector can be extracted from a query, the number of disk accesses can be reduced by comparing the indices with their possible positions at the query. For example, if the query with bit stream **110*001** is searched and k = 3, two determining vectors 110 and 001 can be extracted. Then, two lists of indices in buckets 110 (6) and 001 (1) are collected. Suppose the lists with addresses 6 and 1 are {1; 5} and {9; 13} respectively, since 110 and 001 are at positions 3 and 7 in the original bit stream, their position difference is 4. Thus, if an index in one list cannot match with another index in another list with difference equal to 4, it is invalid and can be removed. In this example, index 1 in the first

list and index 13 in the second list can be removed and only those pages containing subsequences of indices 5 and 9 are retrieved. The steps for the *Validation Phase* to remove invalid indices is as follows.

(1) Given a query Q, it is converted to a bit stream with three states 0, 1 or *. Then, according to the window size k, the set of all the possible determining vectors, V, is extracted.

(2) For each $v_i \in V$, a list of indices l_i is collected from the corresponding bucket. Let $L = \{ l_i: v_i \in V\}$ and assume that each l_i is sorted in ascending order. Then the invalid indices from each l_i are removed. This can be done by comparing the indices in the lists in ascending order similar to merge sort. The worst case complexity of this operation is $O(N)$, where N is the total number of indices in all the lists.

Union Phase. After the invalid indices are removed, we may reduce more disk accesses by the following observations.

(1) Some subsequences pointed to by the indices in the same bucket may be located in the same page. They can be retrieved by just one disk access when the bucket is searched.

(2) When a determining vector with unknown bits is searched, more than one bucket will need to be accessed. Some subsequences pointed to by the indices in different buckets may be located in the same page.

According to the two observations above, the steps for the *Union Phase* to further reduce the disk accesses are as follows.

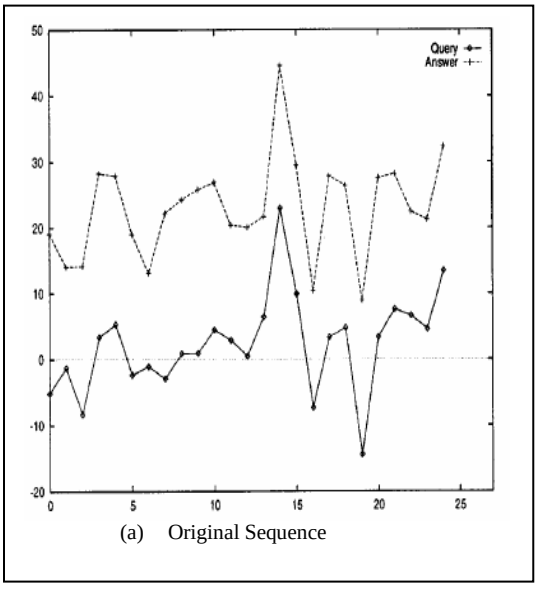
(1) For each l_i from *Validation Phase*, find out the set of pages P_i at which the subsequences of the corresponding indices in l_i are located.

(2) The final set of pages which needs to be accessed is $U_i P_i$.

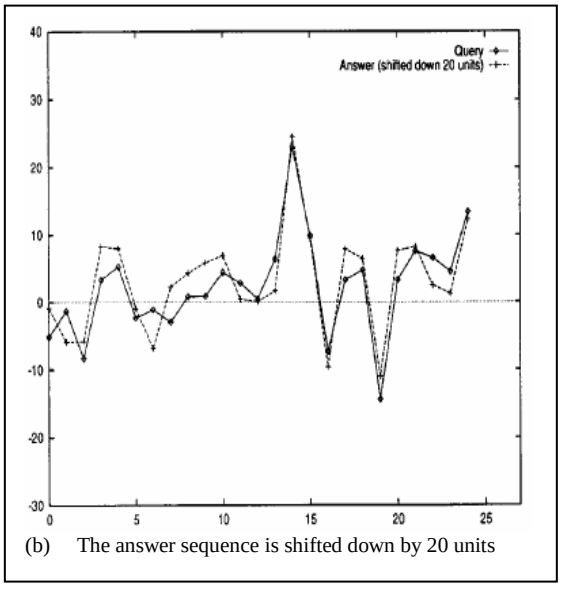
4.0 Experiments

4.1 Similarity definition

In this section, we will show the difference between the traditional similarity definition based on Euclidean distance and the slope (gradient) similarity definition we proposed. Stock data of the Nigerian Stock Exchange Index from January 2, 1986 to December 30, 2005 inclusive are used.



(a) Original Sequence

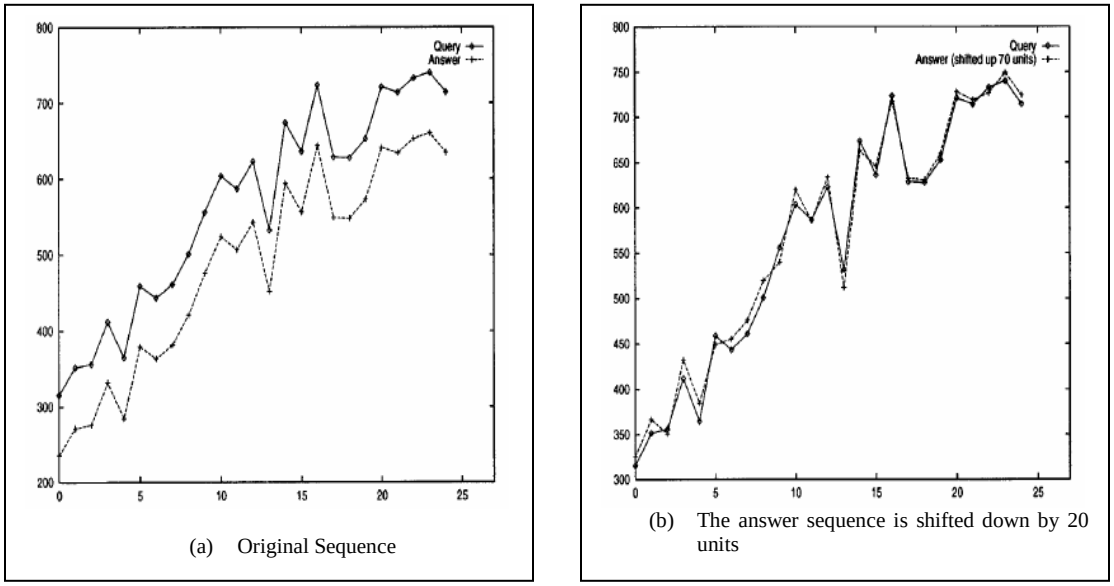


(b) The answer sequence is shifted down by 20 units

Figure 4.1: A sample result.

Suppose ε is 15. In Figure 4.1a, a query sequence and a sample of the answer sequences by slope similarity are shown. If Euclidean distance is used, the answer sequence shown will not be reported. The reason is they have different vertical positions and the Euclidean distance between them is greater than 15. However, it is easier to observe in Figure 4.1b that they are actually similar in shape when the answer sequence is shifted down 20 units. The other example is shown in Figure 4.2.

Suppose *X* and *Y* are sequences of the same length. It is said that a miss happens if *X* is slope similar to *Y* but *X* is not Euclidean similar to *Y*. Then, Figure 4.3 shows the number of misses when the ε is varied from 10 to 20. In each experiment, 100 queries are searched and the average number of misses is recorded.



(a) Original Sequence

(b) The answer sequence is shifted down by 20 units

Figure 4.2 A sample result.

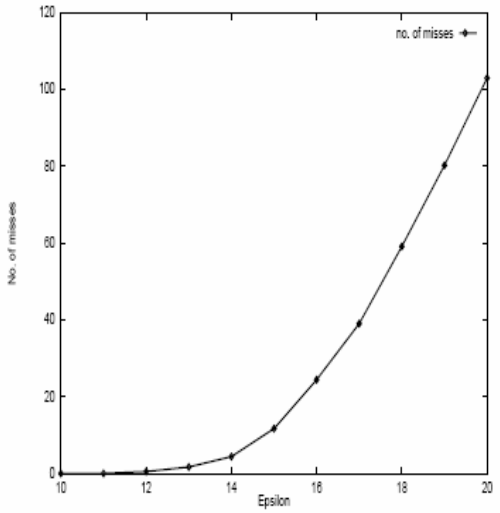


Figure 4.3: The number of misses that is slope similar to but not Euclidean similar to the queries (real data).

4.2 Performance evaluation

We perform several experiments to evaluate our hashing scheme. We compare our hashing scheme to the scheme of sequential search and *R*-tree. The experiments are performed on a Sun SPARCcenter2000 workstation running Solaris 2.5.1 with 512 Mbytes of main memory.

Since the extendable hashing scheme is more general, we exclude the static hashing scheme from the performance study. The lengths of all the queries used are between 10 and 50 and the dimension of the *R*-tree is set to 50 in order to avoid false alarms and extra disk access. The page size for data and directory pages is 4 Kbytes.

Experiments are performed on both synthetic and real data. For the experiments on real data, the Nigerian Stock Exchange Index from January 2, 1986 to December 30, 2005 is used. Synthetic data are generated using a random walk which is widely used to model stock data [12, 13]. The first data point, x_0, of the random walk is chosen randomly. Then each subsequent data point, x_{i+1}, is generated by adding Δx to the previous one, x_i, where Δx is a random variable distributed uniformly in the range between -500 and +500, i.e.

$x_{i+1} = x_i + \Delta x$. In each of the experiments discussed below, 100 queries are searched and the average result is used to evaluate the searching schemes.

First, we evaluate the CPU time of the three schemes on real data by varying ε. The range of the gradient of the real data is from -100 to +100. As we want to limit the error bound to 20% of the possible range, the ε is varied from 0 to 20. The real data set contains 4939 points which is small enough for the three schemes to run on the main memory without disk access. The result is shown in Figure 4.4. We can see that the hashing scheme is faster than the other two schemes. When ε is 0 (exact match), it runs about 100 and 23 times faster than the sequential and *R*-tree schemes, respectively. The *R*-tree searching scheme consumes more CPU time than the hashing scheme because the overlap of the directory of *R*-tree is very high even when the dimension is larger than 10 [14]. For each internal node visited, many rectangles intersect with the query rectangle and a lot of branches need to be traversed. This means the *R*-tree scheme performs more calculations and becomes slower.

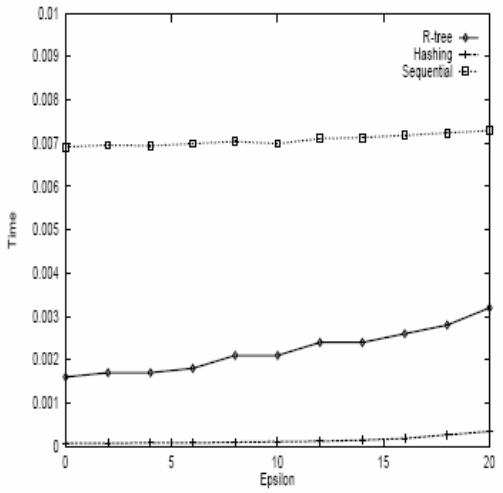


Figure 4.4 CPU-time against epsilon (real data).

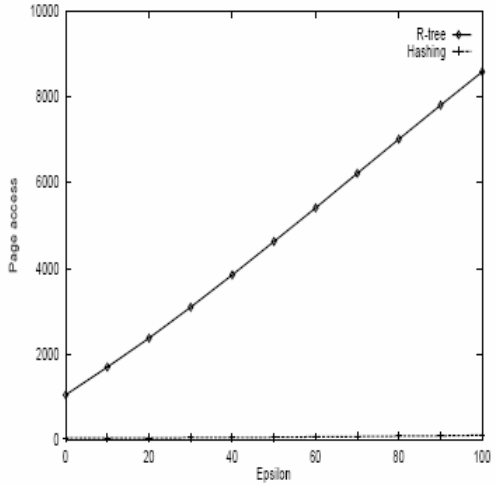


Figure 4.5: Number of page accesses against epsilon (synthetic data).

In real applications, the system usually stores a large amount of time sequences. It is impossible for the applications to load all their data to the main memory. Thus, we carry out an experiment to study the paging behaviour of the schemes. Since the real data set is not large enough, a set of synthetic data (0.3×10^6 points) generated using a random walk is used. Again, the first 20% of the possible error range is used. The value of ε is varied from 0 to 100. For the sequential search, the number of page accesses is constantly equal to ($0.3M \times 4$)/4Kbytes (≈ 293) pages because it has to access all the pages for every query. Figure 4.5 shows the number of page accesses of the hashing and *R*-tree searching schemes. The number of page accesses of the *R*-tree scheme is proportional to the value of ε and is consistently larger than that of the hashing scheme.

The hashing scheme has only 65.71 and 129.91 page accesses at ε = 0 and ε = 100 respectively. This result can also be explained by the high overlap of directories in *R*-tree. We plot the percentage of node accesses to the total number of nodes of the whole *R*-tree in Figure 4.6. When ε = 0 (exact match), it still requires 1.8% node access. This high node access percentage is due to the fact that too many rectangles intersect with the query rectangle during the searching and many branches have to be traversed.

In another experiment, we want to study the effect of the length of queries on the hashing and *R*-tree schemes. We vary the length of the queries and keep ε = 20. The number of page accesses of the hashing and *R*-tree schemes on real data are compared. The result is shown in Figure 4.7.

The numbers of page accesses of the hashing scheme are less than that of the *R* -tree scheme in all lengths examined. Moreover, for both schemes, the number of page accesses decreases when the length of the queries increases. This can be explained as follows. For the hashing scheme, when the query is longer, the probability of constructing a determining vector with a lower number of unknown bits increases. Thus the number of bucket accesses decreases. For the *R* -tree scheme, when the length of queries approaches the dimension of the *R*-tree (it is 50 here), a greater number of dimensions of the rectangle in each internal node visited are compared. Therefore, more branches can be pruned out and the number of page accesses decreases

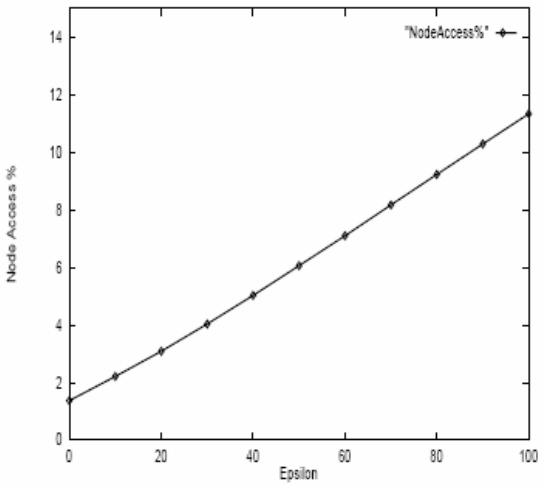


Figure 4.6: Percentage of node accesses with respect to the total number of nodes (synthetic data).

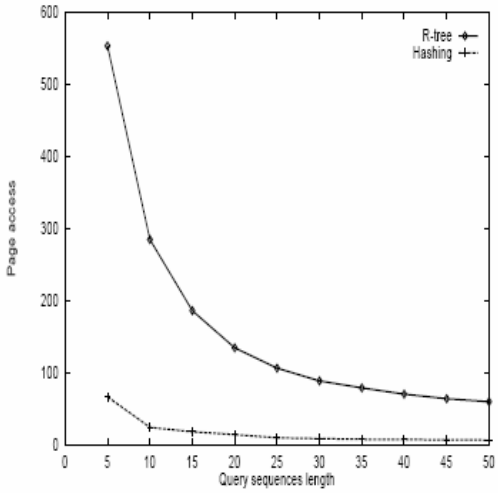


Figure 4.7: Number of page accesses against the length of query sequence (real data).

5.0 Conclusion

Sequence data searching algorithms are in high demand. The Euclidean distance is sensitive to the vertical positions of the two sequences. It is also not a good measure of similarity in terms of their shapes. Thus, we propose a new definition for similarity. The process of comparing the gradients is insensitive to the vertical positions of the sequences and all similar sequences found are similar in shape. In addition to the new definition, we also propose a fast searching algorithm based on dynamic hashing. The algorithm guarantees that no qualified data subsequence similar to the query sequence will be falsely rejected. The proposed algorithm can also find data subsequences similar to the query sequence with different scaling factors. Several experiments were also performed to evaluate the proposed algorithm using both synthetic and real data. The advantage of the hashing algorithm is simple, yet efficient enough to search for time-series sequences.

References

[1] Agrawal, R., Faloutsos, C. and Swami, A. (1993) Efficient similarity search in sequence databases. In *Proc. 4th Int. Conf. on Foundations of Data Organization and Algorithms,* pp. 69–84.

[2] Faloutsos, C., Ranganathan., M. andManolopoulos, Y. (1994) Fast subsequence matching in time-series databases. In *Proc. ACM SIGMOD Conf. on Management of Data,* pp. 419–429.

[3] Agrawal, R., Lin, K. I., Sawhney, H. S. and Shim, K. (1995) Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proc. 21st VLDB Conf.*, pp. 490–501.

[4] Pring, M. J. (1991) *Technical Analysis Explained.* McGraw- Hill.

[5] DeMark, T. R. (1994) *The New Science of Technical Analysis*. John Wiley & Sons.

[6] Berndt, D. J. and Clifford, J. (1995) Finding patterns in time series: a dynamic programming approach. In Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P. and Uthurusamy, R. (eds), *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press.

[7] Goldin, D. Q. and Kanellakis, P. C. (1995) On similarity queries for time-series data: constraint specification and implementation. In *1st Int. Conf. on the Principles and Practice of Constraint Programming, September,* **976**, *137-153.. (1997)*

[8] Bollobas, B., Das, G., Gunopulos, D. and Mannila, H. (1997) Time-series similarity problems and well-separated geometric sets. In *13th Annual ACMSymp. on Computational Geometry,* pp. 454–456.

[9] Das, G., Gunopulos, D. and Mannila, H. (1997) Finding similar time series. In *1st Eur. Symp. on Principles of Data Mining and Knowledge Discovery,* pp. 88–100.

[10] Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H. R. (1979) Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.,* 4, 315–344.

[11] Silberschatz, K. (1991) *Database System Concepts*. McGraw- Hill.

[12] Chatfield, C. (1991) *The Analysis of Time Series: An Introduction.* McGraw-Hill.

[13] Mandelbrot, B. (1977) *Fractal Geometry of Nature*. W. H. Freeman.

[14] Berchtold, S., Keim, D. A. and Kriegel, H.-P. (1996) The Xtree: An index structure for high-dimensional data. In *Proc. 22nd VLDB Conf.,* pp. 28–39.