Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases

Rakesh Agrawal *King-Ip Lin** *Harpreet S. Sawhney* *Kyuseok Shim*

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We introduce a new model of similarity of time sequences that captures the intuitive notion that two sequences should be considered similar if they have enough non-overlapping time-ordered pairs of subsequences thar are similar. The model allows the amplitude of one of the two sequences to be scaled by any suitable amount and its offset adjusted appropriately. Two subsequences are considered similar if one can be enclosed within an envelope of a specified width drawn around the other. The model also allows non-matching gaps in the matching subsequences. The matching subsequences need not be aligned along the time axis.

Given this model of similarity, we present fast search techniques for discovering all similar sequences in a set of sequences. These techniques can also be used to find all (sub)sequences similar to a given sequence. We applied this matching system to the U.S. mutual funds data and discovered interesting matches.

1 Introduction

Time-series databases naturally arise in business as well as scientific decision-support applications. The capability to find time-sequences (or subsequences) that are "similar" to a given sequence or to be able to find all pairs of similar sequences has several applications, including [1] [10]:

- Identify companies with similar pattern of growth.
- Determine products with similar selling patterns.
- Discover stocks with similar price movements.
- Find portions of seismic waves that are not similar to spot geological irregularities.

In [1], an indexing structure was proposed for fast similarity searches over time-series databases, assuming that the data as well as query sequences were of the same length. They use the Discrete Fourier Transform (DFT) to map a time sequence to the frequency domain, drop all but the first few frequencies, and then use the remaining ones to index the sequence using a R^*-tree [3] structure. This work was generalized in [10] to allow subsequence matching. Data sequences could now be of different lengths and the query sequence could be smaller than any of the data sequences. They use a sliding window over the data sequence, map each window to the frequency domain, and save first few frequencies. Thus, a data sequence is mapped into a trail in the feature space. The trails are divided into subtrails, which are then represented by their minimum bounding rectangles, which in turn are stored in a R^*-tree to answer queries.

This earlier work, while pioneering, has the following limitations for employing it in practical applications:

- The similarity measure used is the Euclidean distance between the subject sequences. This distance measure can be quite sensitive to a few outliers.
- The problems of amplitude scaling and offset translation have not been addressed. Consider

*Current Address: Department of Computer Science, University of Maryland, College Park, Maryland.

Proceedings of the 21st VLDB Conference
Zürich, Switzerland 1995

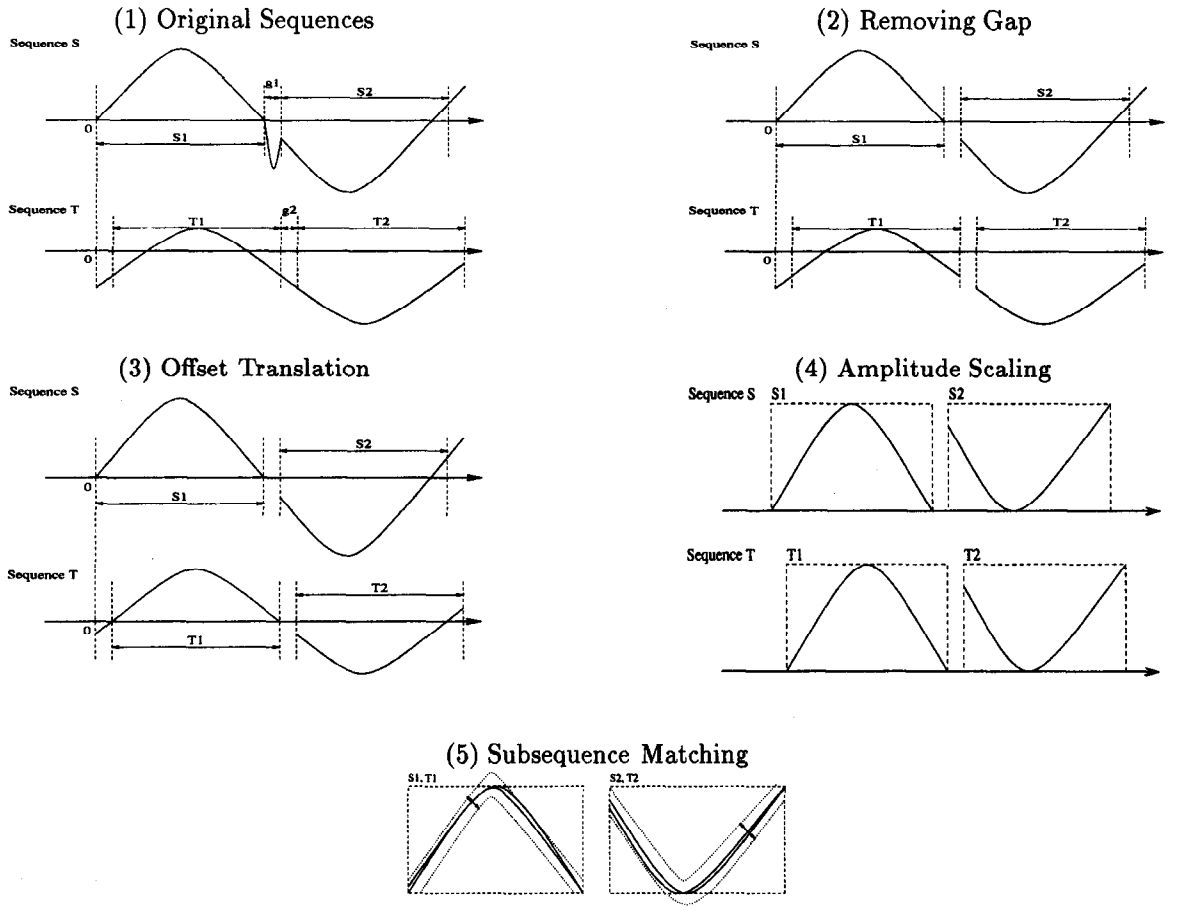

Figure 1: Illustration of sequence matching

the price history of two stocks: one fluctuating around \$10 and the other around \$75. Proper amplitude scaling and offset translation is necessary before determining if the two sequences are similar. A straightforward global scaling will make the method very sensitive to the scale points used, particularly if they happen to be outliers.

- The problem of ignoring unspecified portions of sequences while matching sequences is not addressed.

Our contribution We propose a new model of similarity of time sequences that addresses the above concerns and present fast search techniques for discovering similar sequences. Informally, we consider two sequences to be similar if they have enough non-overlapping time-ordered pairs of subsequences thar are similar. We allow the amplitude of one of the two sequences to be scaled by any suitable amount and its offset adjusted appropriately. For testing the similarity of two subsequences, we check if one lies within an envelope of a specified width around the other, ignoring outliers. The matching subsequences need not be aligned along the time axis.

Figure 1 captures the intuition underlying our similarity model. To determine whether two sequences S and T (1) are similar, we ignore small non-matching regions (called gaps) in S and T (2), translate the offset of T to align it vertically with S (3), scale the amplitude of T (4) so that each of the two subsequences of T lie within an envelope of a specified width around the two corresponding subsequences of S. Since, this condition holds (5) and the total length of gaps is small compared to the total length of sequences, S and T are considered

similar.

We wish to handle a large number of long (say, 5 years of daily data) sequences. Our primary focus is a data mining environment [2] in which the user wishes to find all similar time sequences in a given set of sequences. We would also like to be able to find all similar subsequences that match a given sequence. The user should be able to vary at run-time various similarity parameters such as the width of the envelope, tolerance to outliers, etc., while maintaining efficiency of matching.

Our matching system consists of three main parts: (i) "atomic" subsequence matching, (ii) long subsequence matching, and (iii) sequence matching. The basic idea is to create a fast, indexable data structure using small, atomic subsequences that represents all the sequences up to amplitude scaling and offset. We have chosen the R-tree [12] family of structures for this representation because arbitrary precision can be maintained for the sequence values while still allowing for similarities to be defined with respect to a user-defined ϵ distance in L_∞ norm[1] between the atomic subsequences. Therefore, all atomic subsequences matches within a distance ϵ can be efficiently computed. The second stage employs a fast algorithm for stitching atomic matches to form long subsequence matches, allowing non-matching gaps to exist between the atomic matches. The third stage linearly orders the subsequence matches found in the second stage to determine if enough similar pieces exist in the two sequences. In every stage, the system allows for the flexibility of user/system-defined dynamic parameters without sacrificing efficiency.

Related Work There has been work on finding text subsequences that approximately match a given string [7] [17] [20] [21] [22]. Text sequences normally consist of a few discrete symbols as opposed to continuous numbers that makes the similarity measures and the search methods quite different.

Efficient indexing based matching of two and three dimensional (2D/3D) models to their views in images has been addressed in computer vision and pattern recognition. Geometric hashing [14] has been proposed as a technique for fast indexing. Two key features of this technique are matching that is invariant with respect to certain geometric transformations, and indexing to generate initial hypothesis. In classical geometric hashing, typically affine invariant coordinates for each feature in a model shape/pattern are generated for each possible base coordinate system. The invariant coordinates are represented as a cell in a 2D/3D index table. Matching repeats this process of index generation and is based on collecting high votes for matching indices between patterns.

However, for the application of sequence matching, this classical hashing scheme is not appropriate. Consider a typical case of 1000 sequences each of length 1000–10,000. In order to allow for any global offset and scale changes a similarity invariant index is chosen. For M sequences each with N points, this leads to $O(MN^2)$ indices. If each dimension is quantized into τ buckets, the number of indices per cell is of the order of $\frac{MN^2}{\tau^2}$ which in our example could be anywhere between 1000–100,000 for a τ of 1000. In real situations with relatively smooth sequences, the indices will cluster even more in each cell. Therefore, the matching efficiency suffers because a lot of potentially false matches may be generated. Furthermore, the quantization needs to be fixed at the time of index generation, thus making it hard to vary the tolerance allowed in the definition of similarity at match time. Alternatively, if the index table is populated to take into account the variance allowed in the definition of similarity, the influence of each index is even more spread out in the index table, thus further decreasing the efficiency [11].

In [6] and [7], multidimensional indexing has been proposed as an alternative to the classical 2D/3D geometric hashing in the context of DNA sequence matching and visual shape matching, respectively. The problem of inefficiency due to high saturation of index table and false matches are alleviated to some extent. However, the need for a fixed quantization at table compile time is a drawback that makes this technique inappropriate for our application. Furthermore, the multidimensional indexing scheme works well when the alphabet size is small, for instance only 4 in the context of DNA sequence matching, but the index table becomes prohibitively large even for moderate alphabet size. We expect large alphabet sizes in our application. For instance, sequences with amplitude ranges from 0 to 500 will result in an alphabet size of 50 even with a coarse quantization of 10 levels per alphabet

[1] $L_\infty = \max |p_i - q_i|$ for vectors p and q.

symbol.

Dynamic time warping based matching has been another popular technique in the context of speech processing [18], sequence comparison [9], and shape matching [15]. This method has been used in [4] to match a given pattern in time-series data. The essential idea is to match one dimensional patterns while allowing for local stretching of the time parameterization. However, the matching process is compute intensive at match time and cannot be speeded up by indexing; the complexity of matching is $O(MN)$ given two sequences of lengths M and N.

Organization of the Paper The organization of the rest of the paper is as follows. In Section 2, we formally present our model of similarity of time-sequences. We give our overall approach to finding similar time-sequences using this model in Section 3. We give detailed algorithms in Section 4. In Section 5, we present some sample results of similarity matches from applying the proposed model and algorithms on the U.S. mutual funds data. We conclude with a summary in Section 6.

2 Similarity Model

Two time sequences are said to be similar if they have enough non-overlapping time-ordered pairs of subsequences thar are similar. One of the two sequences can be scaled by any suitable amount and translated appropriately before determining its subsequences that match the subsequences in the other sequence. Two subsequences are considered similar if one lies within an envelope of ϵ width around the other, ignoring outliers. These notions are formalized below.

Notation A time sequence is an ordered set of real values. The ith element of a sequence S is $S[i]$, and a subsequence of S consisting of elements i through j is $S[i,j]$. The first element of S is referred to as *first(S)* and the last element as *last(S)*. The length of the sequence $S[i,j]$ is equal to $j-i+1$. The relationship $<$ defines a total order on the elements of S with $S[i] < S[j]$ iff $i < j$. Two subsequences S and T overlap iff either $\text{first}(S) \leq \text{first}(T) \leq \text{last}(S)$ or $\text{first}(T) \leq \text{first}(S) \leq \text{last}(T)$. We use throughout the L_∞ norm as the distance measure. We assume that the unit of time is the same across all sequences in the database.

Sequence Similarity Time sequences S and T are said to be ξ-similar if they contain non-overlapping subsequences $S_1 \ldots S_m$ and $T_1 \ldots T_m$ respectively such that:

1. $S_i < S_j$ and $T_i < T_j$, $1 \leq i < j \leq m$.
2. $\exists$ *some* scale λ and *some* translation θ so that
$$\forall_{i=1}^{m}\ \theta(\lambda(S_i)) \simeq T_i$$
where $\simeq$ is the *subsequence similarity* operator defined below. $\theta(\lambda(S_i))$ represents a scaled (λ) and translated (by θ) version of the subsequence S_i.
3. $\frac{\sum_{i=1}^{m} \text{length}(S_i) + \sum_{i=1}^{m} \text{length}(T_i))}{\text{length}(S)+\text{length}(T)} \geq \xi$.
That is, the fraction of *match length* to the *total length* of the two sequences is above the specified threshold ξ.

Depending upon the application, some changes can be made in the above definition of similarity. For example:

- If the sequences in the database are of widely varying lengths, one may use only the length of the smaller sequence to test if the fractional match length constraint is satisfied. That is, change the condition 3 above to:
$$\frac{\sum_{i=1}^{m} \text{length}(S_i) + \sum_{i=1}^{m} \text{length}(T_i))}{2 \times \min(\text{length}(S), \text{length}(T))} \geq \xi.$$
- Additional constraints may be placed on the subsequence pairs that can contribute to the match length of the two sequences. An example of such a constraint could be that $\forall_{i=1}^{m}\ S_i$ and T_i must overlap.

Subsequence Similarity The subsequence similarity operator satisfies the following desiderata:

- Two subsequences are similar if one lies within an envelope of a specified width ϵ drawn around the other.
- We should be able to ignore noise (outliers). The atomic unit for matching is a subsequence of length ω (a *window*) in which no outlier is allowed. After matching a window, however, a subsequence of length up to γ (maximum *gap* size) may be ignored.

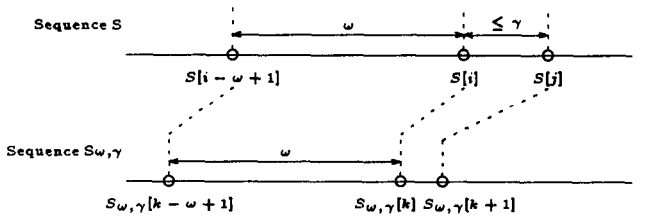

Figure 2: (ω, γ)-projection

We say that S_γ is a γ-projection of a sequence S if it satisfies the following two conditions: i) all the elements in S_γ are also in S and they are in the same order; and ii) if $S[i]$ and $S[j]$ are the two elements in S corresponding to the two consecutive elements $S_\gamma[k]$ and $S_\gamma[k+1]$ in S_γ, then $j - i \leq \gamma$.

We say that $S_{\omega,\gamma}$ is a (ω, γ)-projection of a sequence S if it is a γ-projection of S and additionally if $S_{\omega,\gamma}[i]$ and $S_{\omega,\gamma}[i+1]$ are such that their corresponding elements in S are not consecutive, then the elements of S corresponding to $S_{\omega,\gamma}[i - w + 1] \ldots S_{\omega,\gamma}[i-1]$ are indeed consecutive. Figure 2 shows graphically the concept of (ω, γ)-projection.

We say that two subsequences S and T are $(\epsilon, \omega, \gamma)$-similar if there exist *some* (ω, γ)-projections π_1, π_2 such that

$$\forall i, \ |\pi_1 S[i] - \pi_2 T[i]| \ \leq \ \epsilon$$

and we write $S \simeq T$.

It is easy to add further application-dependent constraints to subsequence similarity defined above. For example, we may require that the corresponding gaps of outliers be of equal size. It can be accommodated by changing somewhat the definition of the γ-projection.

3 Approach

Our overall approach to the problem of determining similarity of two sequences S and T is to decompose the problem into three subproblems:

1. **Atomic Matching**: Find *all* pairs of gap-free subsequences of length ω, called *windows*, in S and T that are similar.

   To account for amplitude scaling and offset translation, we normalize the sequence values within each window W to a range $(-1, +1)$, and form a new window $\tilde{W}$ using the formula:

   $$\tilde{W}[i] = \frac{W[i] - \frac{Wmin + Wmax}{2}}{\frac{Wmax - Wmin}{2}}$$

   where $Wmin$ and $Wmax$ are the minimum and maximum values in the window W. Now two normalized windows $\tilde{W}_1, \tilde{W}_2$ are ϵ-similar if

   $$\forall i, |\tilde{W}_1[i] - \tilde{W}_2[i]| \leq \epsilon$$

   We give in Section 4.1 a fast algorithm for this subproblem.

2. **Window stitching**: Stitch similar windows to form pairs of large similar subsequences.

   Let $\tilde{S}_1 \ldots \tilde{S}_m$ and $\tilde{T}_1 \ldots \tilde{T}_m$ be m normalized windows of two sequences S and T, such that i) $\forall i$, $\tilde{S}_i$ and $\tilde{T}_i$ are similar; and ii) $\forall j > i$, the starting point of window j is later than that of window i.

   We can *stitch* $\tilde{S}_1 \ldots \tilde{S}_m$ and $\tilde{T}_1 \ldots \tilde{T}_m$ into a pair of similar subsequences if the following two conditions are satisfied:

   - For all windows $i > 1$, one of the following is true:
     - $\tilde{S}_i$ does not overlap $\tilde{S}_{i-1}$ and gap between them in S is $\leq \gamma$. The same also holds for $\tilde{T}_i$.
     - $\tilde{S}_i$ overlaps $\tilde{S}_{i-1}$ with the same length d as $\tilde{T}_i$ overlaps $\tilde{T}_{i-1}$.
   - For all S windows, the normalization scale is roughly equal[2]. The same also holds for all windows in T.

   Figure 3 shows the stitching possibilities, assuming that the scaling constraint is satisfied. A match is denoted by two bold horizontal lines connected with a dotted line. The top diagram shows two pairs of windows with the same overlap length. The middle diagram shows two pairs of windows having gaps less than γ. The bottom diagram shows a stitched pair of similar subsequences formed by combining the two conditions.

   Section 4.2 gives a fast stitching algorithm.

[2]This condition is somewhat weaker than requiring one global scale for the whole sequence, but goes well with the spirit of similarity. Moreover, it makes it possible to have a fast window-stitching algorithm.

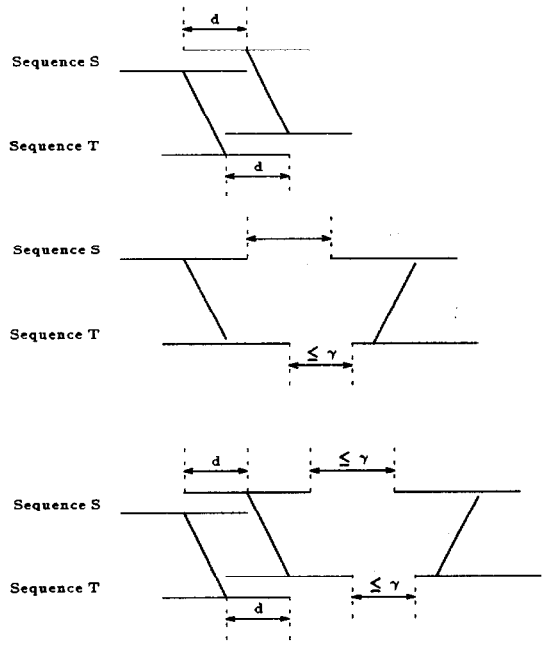

Figure 3: Illustration of the stitching possibilities

3. **Subsequence Ordering**: Find a non-overlapping ordering of subsequence matches having the longest match length.

   Let $\mathcal{S} = (S_1, T_1) \ldots (S_k, T_k)$ be k pairs of subsequences of S and T determined similar in the previous step. We find a subset $(S_{l1}, T_{l1}) \ldots (S_{lm}, T_{lm})$ of $\mathcal{S}$ such that

   - $S_{li} < S_{lj}$ and $T_{li} < T_{lj}$, $1 \leq i < j \leq m$.
   - The scaling used in the matching of each of S_{li} is roughly equal. The same also holds for each of T_{li}.
   - The total match length of this subset is maximal in $\mathcal{S}$. That is, $\sum_{i=1}^{m} \text{length}(S_{li}) + \sum_{i=1}^{m} \text{length}(T_{li})$ is $\geq$ the match length for any other subset of $\mathcal{S}$.

   Section 4.3 gives a fast algorithm for this task.

4 Algorithms

We now give algorithms for the three subproblems identified in the previous section.

4.1 Atomic Matching

In this step, we need to find all pairs of gap-free subsequences of length ω, *windows*, that are similar. A straightforward brute-force approach that compares a window with all other windows to determine similarity will take $O(N^2 l^2)$ time, where N is the number of sequences and l is the length of each one. We present a better solution.

We can consider each window as a point in a ω-dimensional space and reformulate this problem as:

> Given a set of points in a ω-dimensional space, find all pairs of points within a distance of ϵ from each other, where the distance is defined as L_∞ norm.

We can now use a multi-dimensional indexing structure to store the points, and then use a *self-join* algorithm to retrieve all pairs of matching windows.

For building this index, we scan each sequence from beginning to end, extracting and normalizing the ω-dimension point corresponding to each window, and insert the normalized point in the the index. Attached with each point are i) its coordinates, ii) the sequence-id of the corresponding sequence, iii) the starting point of the window, and iv) the scale and translation used to arrive at the coordinate (needed at the time of window-stitching).

Considerations in choosing the index structure The following characteristics of our problem influenced the choice of the index structure:

- *Dimensionality.* The window sizes are typically 5-20. The index structure should be capable of handling dimensions in this range.
- *Self-join.* We want to be able to primarily do self-joins over this structure, as opposed to join between two different structures.
- *Data values.* Since all windows are normalized to a range (-1, +1), any point will always have a -1 and a +1 value for two of its coordinates. Thus, many points will lie on the same hyperplane.

We first considered using hashing, borrowing ideas from the geometric hashing techniques for recognizing shapes [6] [7]. Unfortunately, a static hashing scheme, where all the hash table boundaries have been pre-set, has the following disadvantages:

- Hashing means quantization, which implies errors in precision.

- No matter what the interval is, for each hash region the join algorithm will have to look into "adjacent" hash table entries to avoid false dismissals.

- The number of hash table entries can become very large, making the hash table unmanageable.

We also considered using a grid-based index structure (such as grid-file [13]), but decided against it. Since our points typically have a high dimension, the growth in the size of such a structure can become intolerable. Moreover, because the grid-based methods partition the space into adjacent regions and we are doing a self-join with a non-zero value for distance, many joins of adjacent grids will have to be performed.

We finally settled on the R-tree[3] [12] family of multi-dimensional structures because they tend to be more resilient to higher dimensionalities [16]. Moreover, since the R-tree based methods do not store "dead space", regions can have a larger separation. This can result in fewer pages to be joined, speeding up the join-time.

Specifically, we implemented two R-tree variants: R^*-tree [3] and R^+-tree [19], and specialized them to better fit our problem. We discuss them next, emphasizing the customizations we made in their implementation.

R^*-tree The R^*-tree [3] enhanced the original R-tree in two major ways. First, it added the perimeter of the bounding regions as an important factor to the heuristics for node splitting. Second, it introduced the notion of forced reinsert to make the shape of the tree less dependent on the order of the insertion. When a node becomes full, it is not slitted immediately, but a portion of the node is reinserted from the top level again.

Because of our definition of a window, many data points will be lying on lower dimension hyperplanes and these hyperplanes will have zero volume in ω-dimension. In our R^*-tree implementation, therefore, we defined a new measure for deciding which branch to take during the insertion and for determining splits. This measure takes into account volumes of lower dimension hyper-surfaces. For each ω-dimensional region in the tree, the *measure* of the region is defined to be a ω-dimensional vector with the following values:

> (ω-dimension volume, sum of all ω-1 dimensional regions' volume, sum of all ω-2 dimensional regions' volume, ..., perimeter)

Lexicographical ordering is used to order the measures. Components of the measure are computed on a when-needed basis.

R^+-tree The R^+-tree [19] imposes the constraint that no two bounding regions of a non-leaf node can overlap. Thus, except for the boundary surfaces, there will be only one path to every leaf region, which can reduce search and join costs. However, the drawback is that when splitting an internal node, no split axis may be found that completely divides the bounding regions into two non-overlapping regions, causing the split to be propagated downwards as well as upwards. Thus, no minimum space utilization can be guaranteed as a downward split has to be made on a certain coordinate, leading to uneven distribution. This in turn leads to under-filled internal and leaf nodes, and the tree grows faster. In range searching, this problem may not be too significant as the tree height grow logarithmically. However, in the case of the similarity self-join, this problem can be troubling as more nodes will lead to more pairs of nodes getting joined.

We attack this problem by adopting the R^*-tree reinsertion idea. Whenever a downward split results in an under-filled leaf node (40% of the leaf), the node is released from the tree and all the data points are reinserted from its *immediate parent*. We need not reinsert from the root as in the case of R^*-tree, as the no-overlap rule guarantees that the insertion algorithm will traverse down the tree back to the parent.

A problem that arises sometimes when inserting a point in R^+-tree is illustrated in Figure 4. No matter which branch is taken, the enlargement will introduce overlap between the regions in this case, unless the nodes are restructured. We store the problem points in a temporary structure and reinsert them at a later time, with the hope that future splitting and restructure will allow the insertion of the problem points smoothly.

[3]The R-tree [12] can be viewed as an extension of the B-tree to multi-dimensions. The R-tree is a balanced tree, in which each node represents a region in the space. For each parent-child pair in the tree, all the children's regions are within that of parent's. The tree achieves its balance by splitting and propagating the split upwards.

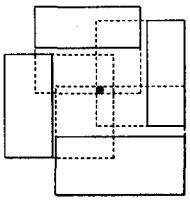

Figure 4: Insertion problem in R^+-tree

```
procedure SelfJoin(node, path, ε)
Input: A node, a path from the root and ε.
Output: A set of pairs of points which are
        within ε distance.
begin
1. if (node.type = non-leaf) then {
2.    forall child ∈ node.children do
3.       output := SelfJoin(child, node ∪ path, ε);
4. }
5. else {
6.    output := output ∪ εJoin(node, node, ε);
7.    forall leaf ∈ intersect(node, path, ε) do
8.       output := output ∪ εJoin(node, leaf, ε);
9. }
10. return(output);
end
```

Figure 5: Self-join algorithm

Self-Join algorithm The previous work on join algorithms for the R-tree variants [5] has been focused on the join between two different index tress. In contrast, our application requires a self-join that lends the opportunity to traverse the tree more intelligently.

Figure 5 shows the the self-join algorithm we have used in our implementation. The algorithm calls itself recursively when the node is a non-leaf node. At a leaf node, we join the node with any leaf node that has an overlapping region. We use the function *ϵJoin()* to compute the joins within a distance ϵ between points in two leaf nodes. The function *intersect()* determines the other leaf nodes whose regions overlap with the given *node* by traversing the index.

The self-join algorithm must ensure that the same two leaf nodes are not joined more than once. This is accomplished by using an ancestor list rather than root in the recursive call of the algorithm and imposing an arbitrary order on the children of each node. The function *intersect()* returns only those leaf nodes that come later in this ordering.

CPU cost is an important factor in spatial-joins [5]. To reduce CPU cost for redundant comparisons between points in an any two nodes, we first screen points which lie within ϵ-distance from the boundary surface of other node and use sort-merge join for those screened points.

The experiments that we performed with our datasets showed that the performance of R^+-tree was better than R^*-tree for our application.

4.2 Window Stitching

We formulate window stitching as a problem of finding longest path in an acyclic graph.

The output of the window-matching step is the pairs of matching windows for every pair of sequences S and T. Construct a match graph $\mathcal{G}$ as follows for each pair of S and T:

- Represent each pair of matching windows as a vertex.
- Draw an arc from a vertex corresponding to match $M_i = (S_i, T_i)$ to a vertex corresponding to match $M_j = (S_j, T_j)$ iff
  - The starting points of both the windows in M_j are later than the starting points of their corresponding windows in M_i. That is, $\text{first}(S_i) < \text{first}(S_j) \wedge \text{first}(T_i) < \text{first}(T_j)$
  - Either one of the following is true:
    - The corresponding windows in the two matches do not overlap and the gap between them is $\leq \gamma$. That is,
      $$(S_i \cap S_j = T_i \cap T_j = \emptyset) \wedge (\text{first}(S_j) - \text{last}(S_i) \leq \gamma) \wedge (\text{first}(T_j) - \text{last}(T_i) \leq \gamma)$$
    - The amount of overlap between S_j and S_i in S is the same as the amount of overlap between T_j and T_i in T.
- Assign label $< l_{ij}, fs_i, ls_j, ft_i, lt_j >$ to arc $M_i \rightarrow M_j$, where
  $$fs_i = \text{first}(S_i),\ ls_j = \text{last}(S_j),$$
  $$ft_i = \text{first}(T_i),\ lt_j = \text{last}(T_j),\ \text{and}$$
  $$l_{ij} = (ls_j - fs_i) + (lt_j - ft_j)$$

where the length of the arc l_{ij} represents the total match length (including gaps)[4].

Figure 6 shows the pairs of window matches named $A \ldots E$, and the corresponding match graph for it. There is no edge $A \rightarrow E$ in the graph because the maximum gap constraint is not satisfied. Similarly, there is no edge $B \rightarrow F$ because there are overlapping windows with unequal overlap.

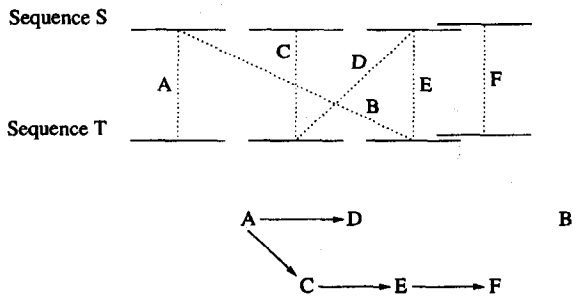

Figure 6: A match graph

Consider a path $P + A$ in $\mathcal{G}$ obtained by composing a path P with arc A. Let the labels of P and A be $< l_{ij}, fs_i, ls_j, ft_i, lt_j >$ and $< l'_{ij}, fs'_i, ls'_j, ft'_i, lt'_j >$ respectively. We define the label of $P + A$ to be:

$$< ((ls'_j - fs_i) + (lt'_j - ft_j)), fs_i, ls'_j, ft_i, lt'_j >$$

With this definition of path composition, $\mathcal{G}$ has the property:

> If for two paths P, Q in $\mathcal{G}$, length(P) < length(Q) and first(P) = first(Q), then for any arc R in $\mathcal{G}$, we have that length($P + R$) < length($Q + R$).

We can therefore traverse $\mathcal{G}$ in reverse topological sort order and find the longest path [8], which will correspond to the longest match.

One final detail concerns ensuring that the normalization scale used is roughly the same for all the windows in a stitched subsequence. It can be incorporated in the graph traversal algorithm by checking that the scales for the windows corresponding to the arc with which a path is being extended is consistent with the scales for the windows already in the path.

[4]Depending on the application, the definition of the length l_{ij} can be changed to to exclude gaps in the match length.

4.3 Subsequence Ordering

Having found pairs of similar subsequences, we can determine the maximal length match in two sequences using a minor variation of the window stitching algorithm.

We again form a match graph and find the longest path in it. The difference is that the subsequence matches now contribute to vertices and arcs are created using a somewhat different constraint. In a match graph for sequences S and T, an arc from match $M_i = (S_i, T_i)$ to match $M_j = (S_j, T_j)$ is created iff last(S_i) < first(S_j) and last(T_i) < first(T_j); i.e., if the corresponding subsequences in M_j do not overlap with those in M_i and come later. The length of an arc is the sum of the lengths of the four subsequences, S_i, S_j, T_i, and T_j.

In fact, this subsequence ordering step can be combined with the window stitching step. We have presented them as separate steps for clarity. In addition, there are applications in which we are interested in finding subsequences that are similar to a given sequence. In that case, we only require window stitching.

5 Experiments

To get the feel for the kinds of similarity matches found by our algorithm, we experimented with the time-series database of the closing prices of U.S. mutual funds. The data is available from the MIT AI Laboratories' Experimental Stock Market Data Server (http://www.ai.mit.edu/stocks/mf.html).

We used $\omega = 8$, $\epsilon = 0.2$, and $\gamma = 4$ in our test. Figure 7 and Figure 8 show two of the several pairs of similar mutual funds discovered by our algorithm. The y-axis is the closing price of the fund in US dollars, and x-axis gives the date for the fund price. The data for the Harbor International Fund, Ivy International Fund, and Fidelity Selective Precious Metal & Mineral Fund is for the period from July 27, 1993 to February, 3 1995, excluding holidays and weekends (385 data points). The data for the VanEck International Investor Fund is from January 4, 1993 to February 3, 1995 (525 points). The solid lines in the graphs represent the portions of the sequences found similar by our algorithm. The dotted lines represent the non-matching part of the sequences.

Even if some funds are in the same group, they do not generally perform similarly because the fund managers maintain different portfolios. The two

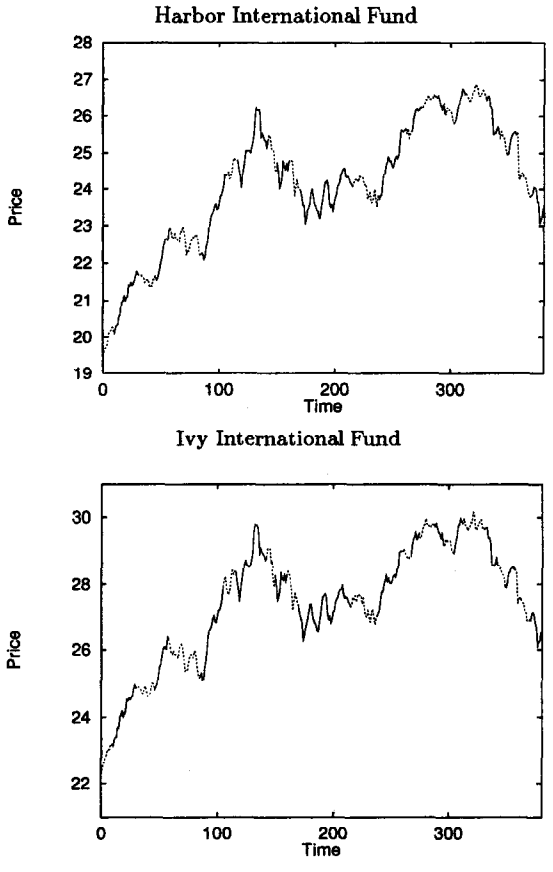

Figure 7: Two similar mutual funds in the same fund group

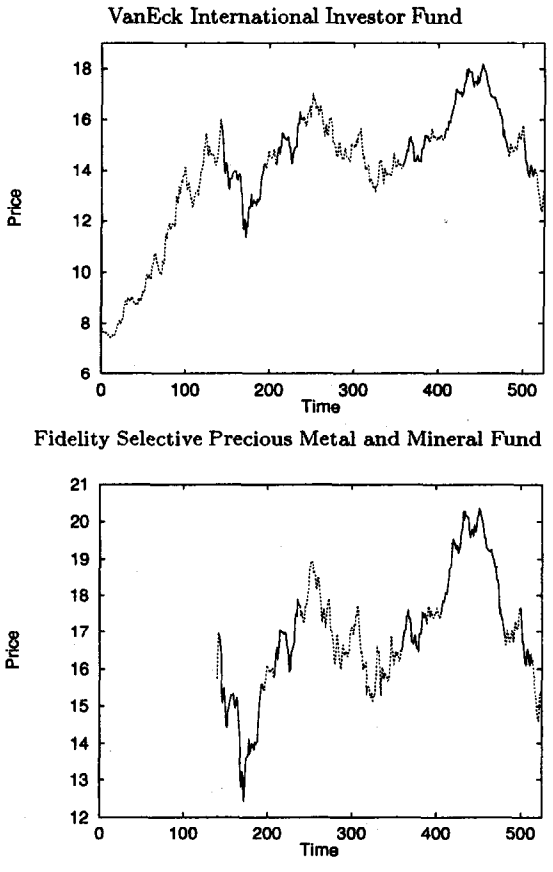

Figure 8: Two similar mutual funds in different fund groups

funds in Figure 7 are both international funds, but managed by different fund managers. The funds in Figure 8 are even more interesting. They belong to two different groups—one is an international fund and the other a precious metal and mineral fund.

6 Summary

We addressed the problem of sequence similarity for applications involving one dimensional time series data. We introduced an intuitive notion of sequence similarity whose parameters a user can vary at runtime, while maintaining efficiency of matching. It is a robust measure that allows non-matching gaps, amplitude scaling, and offset translation.

Given this similarity model, we presented fast search techniques for discovering all similar sequences in a set of sequences. These techniques can also be used to find all (sub)sequences similar to a given sequence. Our matching system consists of three main parts: (i) "atomic" subsequence matching, (ii) long subsequence matching, and (iii) sequence matching. We use the R-tree [12] family of structures (specifically, the R^+-tree) to create a fast, indexable data structure using small, atomic subsequences that represents all the sequences up to amplitude scaling and offset. Therefore, all atomic subsequence matches within a user-specified distance ϵ can be efficiently computed by doing an ϵ self-join on this structure. The second stage employs a graph algorithm for stitching atomic matches to form long subsequence matches, allowing non-matching gaps to exist between the atomic matches. The third stage linearly orders the subsequence matches found in the second stage to determine if enough similar pieces exist in the two sequences.

We applied our matching techniques to the U.S. mutual funds data and discovered several

interesting matches. For example, we could find funds belonging to the same category of funds that had similar price behavior. More interestingly, we could identify funds belonging to different fund categories whose price movements were similar.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proc. of the Fourth International Conference on Foundations of Data Organization and Algorithms*, Chicago, October 1993. Also in Lecture Notes in Computer Science 730, Springer Verlag, 1993, 69–84.

[2] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993. Special Issue on Learning and Discovery in Knowledge-Based Databases.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD*, pages 322–331, Atlantic City, NJ, May 1990.

[4] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, pages 359–370, Seattle, Washington, July 1994.

[5] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. of ACM SIGMOD*, pages 237–246, Washington, D.C., May 1993.

[6] A. Califano and R. Mohan. Multidimensional indexing for recognizing visual shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(4):373–392, 1994.

[7] A. Califano and I. Rigoutsos. FLASH: A fast look-up algorithm for string homology. In *Proc. of the 1st International Conference on Intelligent Systems for Molecular Biology*, pages 353–359, Bethesda, MD, July 1993.

[8] B. Carre. *Graphs and Networks*. Claredon Press, Oxford, 1978.

[9] B. W. Erickson and P. H. Sellers. Recognition of patterns in genetic sequences. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison Wesley, MA, 1983.

[10] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1994.

[11] W. E. L. Grimson and D. P. Huttenlocher. On the sensitivity of geometric hashing. In *Proc. 3rd Intl. Conf. on Computer Vision*, pages 334–338, 1990.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pages 47–57, Boston, Mass, June 1984.

[13] K. Hinrichs and J. Nievergelt. The grid file: a data structure to support proximity queries on spatial objects. In M. Nagl and J. Perl, editors, *Proc. of the WG'83 (Intern. Workshop on Graph Theoretic Concepts in Computer Science)*, pages 100–113, Linz, Austria, 1983.

[14] Y. Lamdan and H. J. Wolfson. Geometric hashing: A general and efficient model-based recognition scheme. In *Proc. 2nd Intl. Conf. on Computer Vision*, pages 238–249, 1988.

[15] R. McConnell et al. Ψ-S Correlation and dynamic time warping: Two methods for tracking ice floes in SAR images. *IEEE Transactions on Geoscience and Remote Sensing*, 29(6):1004–1012, 1991.

[16] M. Otterman. Approximate matching with high dimensionality R-trees. M.sc. scholarly paper, Dept. of Computer Science, Univ. of Maryland, College Park, Maryland, 1992.

[17] M. Roytberg. A search for common patterns in many sequences. *Computer Applications in the Biosciences*, 8(1):57–64, 1992.

[18] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26:43–49, 1978.

[19] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+ tree: a dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on VLDB*, pages 507–518, England, 1987.

[20] M. Vingron and P. Argos. A fast and sensitive multiple sequence alignment algorithm. *Computer Applications in the Biosciences*, 5:115–122, 1989.

[21] J. T.-L. Wang, G.-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Minneapolis, May 1994.

[22] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.