Master's Thesis: Mining for Frequent Events in Time Series

by

Zachary Stoecker-Sylvia

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

August 2004

APPROVED:

Professor Carolina Ruiz, Thesis Advisor

Professor Fernando Colon Osorio, Thesis Reader

Professor Michael A. Gennert, Head of Department

Abstract

While much work has been done in mining nominal sequential data much less has been done on mining numeric time series data. This stems primarily from the problems of relating numeric data, which likely contains error or other variations which make directly relating values difficult. To handle this problem, many algorithms first convert data into a sequence of events. In some cases these events are known a priori, but in others they are not. Our work evaluates a set of time series data instances in order to determine likely candidates for unknown underlying events. We use the concept of bounding envelopes to represent the area around a numeric time series in which the unknown noise-free points could exist. We then use an algorithm similar to Apriori to build up sets of envelope intersections. The areas created by these intersections represent common patterns found throughout the data.

Acknowledgements

I would like to thank my advisor, Professor Carolina Ruiz, for all her help and contributions to this thesis and my time in general at Worcester Polytechnic Institute, and my reader, Professor Fernando Colon Osorio, for being very flexible in his scheduling and thus allowing me to meet my deadlines. I would also like to thank Professor Edgar Ramos at the University of Illinois at Urbana-Champaign for his aid in finding an algorithm to find the best intersection of two envelopes.

Additionally I would like to thank the members of WPI's Knowledge Discovery in Databases Research Group for their aid and input throughout this project. Special thanks are due in particular to Keith A. Pray and Dharmesh Thakkar for their great help in both finding resources and as sounding boards. I would also like to thank my Section Leader at work, Robert Hyland, for his support in the form of both suggestions and understanding.

Finally, I would like to thank all of my friends and family for their understanding as I was consumed by my work on this thesis.

Contents

1 Introduction **1**

1.1 Time Series Data 1

1.2 Events 2

1.3 Error 3

1.4 Bounding Envelopes 4

1.5 The Apriori Algorithm 5

1.6 Related Work 7

2 Our Approach **10**

2.1 Overview 10

2.2 Input and Output 12

2.2.1 Input 13

2.2.2 Output 16

2.3 In-Depth Process Description 17

2.3.1 Creating the Initial Envelopes 18

2.3.2 Finding the Best Intersection Between Two Envelopes 20

2.3.3 Building the Envelope Combinations 30

2.3.4 Resulting Output 35

2.4 Pseudo-Code 35

2.4.1 CreateEnvelopes 35
2.4.2 FindBestIntersection 36
2.4.3 CreateFirstTwoLevels 39
2.4.4 CreateFurtherLevels 40
2.4.5 FindTimeSeriesTemplates 41

3 Experimental Results 42

3.1 Synthetic Data 42
3.1.1 Data Generation 42
3.1.2 Effectiveness Relative to Epsilon 43
3.2 Unemployment Data 44
3.2.1 Experiments 45
3.3 Monthly Stock Quotes 50
3.3.1 Experiments 50
3.4 Run-time Experiments 55
3.4.1 Varying Epsilon 56
3.4.2 Varying Sequence Length 58
3.4.3 Varying Number of Sequences 59
3.4.4 Time Spent per Intersection 61

4 Conclusion 63

4.1 Contributions of this Work 63
4.1.1 Contributions to Data Mining at the Worcester Polytechnic Institute 65
4.1.2 Future Work 66

A Unemployment Data 67

B Monthly Stock Data **69**

List of Figures

1.1 An example of time series data 2

1.2 Creating a bounding rectangle 5

1.3 Envelopes are the union of all bounding rectangles 5

2.1 The three time series all have the same temporal distance between the same-valued points, thus they are equivalent within our algorithm. 13

2.2 The middle point has no effect on the envelope created ($\delta = 1$) 14

2.3 The time series envelope above completely encompasses template envelope 17

2.4 Creating a bounding rectangle 18

2.5 Envelopes are the union of all bounding rectangles 18

2.6 Only the vertical values at each relative time points are needed to represent the bounding envelope 20

2.7 The first blocks (representing the first time point) of two short envelopes (shown in the upper left corner) are shifted vertically against each other. As two boxes are shifted the area of their intersection creates a piecewise function. 21

2.8 This figure shows both blocks of the short envelope shown in Fig. 2.7 being intersected individually (the first time point is on the left, the second on the right). Since a shift value is applied to envelopes as a whole the two area functions work on the same shift scale and can be added together to find a composite area function for the total area. . 22

2.9 The area functions created from intersecting the two blocks (from Fig. 2.8) are combined to create a unified area function which is split into subsections with simple linear functions. The purple/dark-thin line represents the intersection of the first point of each envelope, the orange/light line represents the intersection of the second pair of points, and the red/dark-thick line represents the combination of the two. 23

2.10 The individual sections of the functions are added together in a tree structure. The first level is the 0 area root, the second adds in the first box and the third adds in the second (and shows the total area function). 25

2.11 Boxes 1-5 represent a previous node level, while boxes A-C are new ranges being compared to them. 29

2.12 Two different envelope combinations created from the same two envelopes 31

3.1 Table of fully matched templates, partially matched templates, and false templates by number of contributors 44

3.2 Graph of normalized monthly unemployment rate for Middlesex, Worcester, and Barnstable counties 46

3.3 An unemployment template envelope 47

3.4 An unemployment template envelope 48

3.5 An unemployment template envelope 49
3.6 Graph of monthly stock values for Intel, Microsoft, Sun, and IBM . . 50
3.7 A template envelope found by our algorithm and the 'Descending Triangle' stock value template 52
3.8 A template envelope found by our algorithm and the 'Ascending Triangle' stock value template 53
3.9 A template envelope found by our algorithm and the 'Broadening Top' stock value template 54
3.10 Graph of Intersections Performed as Epsilon Varies 56
3.11 Graph of Total Time Spent as Epsilon Varies 57
3.12 Graph of Intersections Performed as Sequence Length Varies 58
3.13 Graph of Total Time Spent as Sequence Length Varies 59
3.14 Graph of Intersections Performed as Number of Sequences Varies . . 60
3.15 Graph of Total Time Spent as Number of Sequences Varies 61
3.16 Graph of Total Time Spent Relative to Intersections Perfromed . . . 62

Chapter 1

Introduction

Our work will evaluate a set of *time series data* instances in order to determine likely candidates for unknown underlying *events*. To do this we will wrap each time series in an *envelope* representing the noise and *error* inherent in numeric measurements. These envelopes will then be intersected using a modified version of the *Apriori algorithm* used commonly to mine association rules. The areas remaining after intersecting two or more envelopes represent common patterns within the sequences that are likely to represent some event these sequences have in common. These resulting areas may then be used to visually learn more about the data set or to transform numerical time series into a sequence of events over time for use in further mining.

1.1 Time Series Data

A time series is a series of numerical measurements related through time, $T = (t_1, y(t_1)), (t_2, y(t_2)), ..., (t_n, y(t_n))$ (see Fig. 2.1). Time series is a very common form for collected data as companies and analysts are often concerned with discovering patterns in time such that they may be capable of predicting future patterns.

Examples of time series include stock prices, periodic temperature readings, and other measurements made over time.

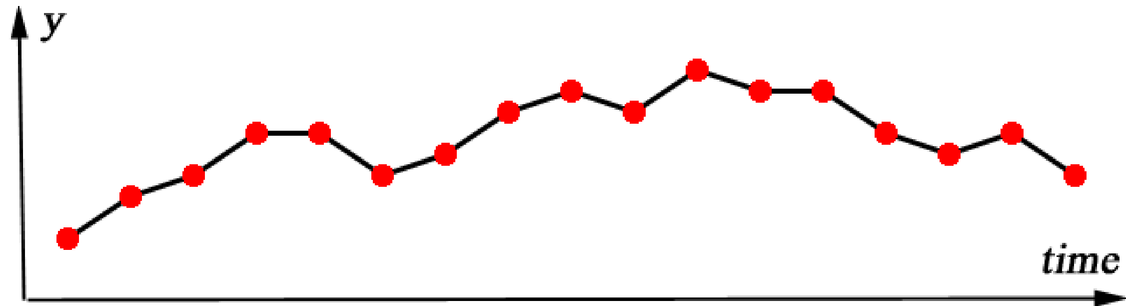


Figure 1.1: An example of time series data

While many algorithms exist to discover patterns in non-temporal data or temporal nominal data few algorithms exist to find patterns within time-based numeric data. This stems from the increased complexity of such data. Not only are pieces of numeric data difficult to relate due to the errors and other noise which make exact matching of values impossible, but the temporal relationships are not only an additional dimension but also have special meaning which must be preserved in mining.

1.2 Events

Since many data mining algorithms are not directly capable of performing mining on numeric data, data must be somehow abstracted before using these algorithms. In the case of time series data, one possibility is to convert the time series into a sequence of events. These events represent an occurrence happening at a point in time and continuing for some duration (which may be 0).

Some examples of events in a generic time series would be when the series reaches its highest value (instantaneous), a period of monotonic increase, or a much more specific pattern that repeats within the data. In some domains, such as the stock

market, events for abstraction of data have already been determined by domain experts [LR91], however in many others no such complex template events exist. Identifying the first two types of events above in a time series is fairly straightforward. In contrast, discovering events of the third type, which is the topic of this thesis, is much more complex.

1.3 Error

A basic assumption when mining data of any sort is that some meaningful underlying pattern exists within. This underlying pattern is often hidden by errors in measurement or simply obscured by noise from other sources. The purpose of data mining is to account for that error and attempt to find the underlying pattern.

Data can be considered to be some absolute pattern, $f(t)$, plus an error function, $\epsilon(t)$, that encompasses all the various errors that are introduced to the pattern. However, not all error exists solely along the y-axis and thus while a single error function can accommodate errors along the time-axis it is much more precise to allow for another term to represent this. This time-based error we label $\delta(t)$ and place within the function. Thus now $y(t) = f(t + \delta(t)) + \epsilon(t)$.

For all error measurements there are two types of error that can be used. Error that changes over time is called **heteroscedastic** error. On the other hand, error that remains relatively constant is called **homoscedastic** error. While homoscedastic error is still variable (otherwise it would not really be error), its maximum and minimum values are generally stable and thus predictable. Since for heteroscedastic error these bounds are variable with time, problems involving it are usually much more difficult to solve and require a domain expert to predict the general function that governs the error. Also, because time variables likely exist in both the pattern

and the error, it is very difficult to extract a representation of the underlying function in these problems. Unless a problem is known to have a heteroscedastic error model, it is reasonable to assume that the error is homoscedastic [GS99].

Our method is capable of handling both types of errors (assuming a formula for the heteroscedastic error is known), but the remaining text will assume homoscedastic error for simplicity, thus the functional aspect of the errors in the equation above can be dropped and it may instead be represented as $y(t) = f(t \pm \delta) \pm \epsilon$.

1.4 Bounding Envelopes

Bounding envelopes are regions around a time series used to represent bounds for error in the series. Since an observed series may contain within it some amount of error, bounding envelopes are used to provide an area within which the actual data (free from the distortions of error) is likely to lie.

Bounding envelopes may be computed in a variety of ways depending on the chosen representation of a series' error. For this work, we use bounding envelopes as presented in Vlachos et al. [VHGK03]. Their bounding envelopes are constructed as the union of bounding rectangles cast by each point.

Each bounding rectangle gives the range within which that point could exist. For a point, $p = (x, y)$, the bounding rectangle covers all points with y-values within $y \pm \epsilon$ (the vertical error) and x-values within $x \pm \delta$ (the time-based error).

Vlachos et al. use bounding envelopes in order to quickly determine whether two sequences are similar enough to warrant further calculation. Any two sequences whose bounding envelopes intersect can match perfectly with a maximum shift of δ horizontally and ϵ vertically to the points in each sequence.

Our process, however, will use bounding envelopes, not as a method of eliminat-

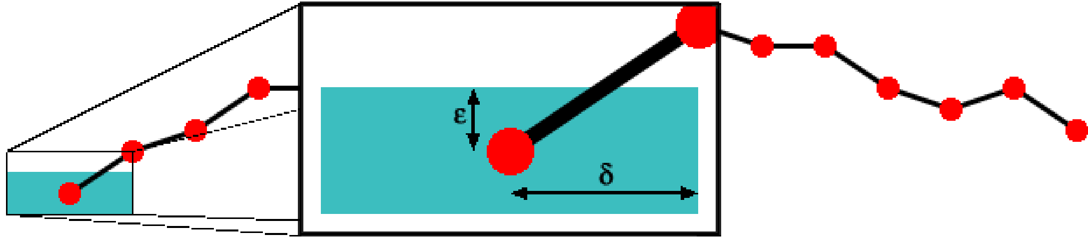


Figure 1.2: Creating a bounding rectangle

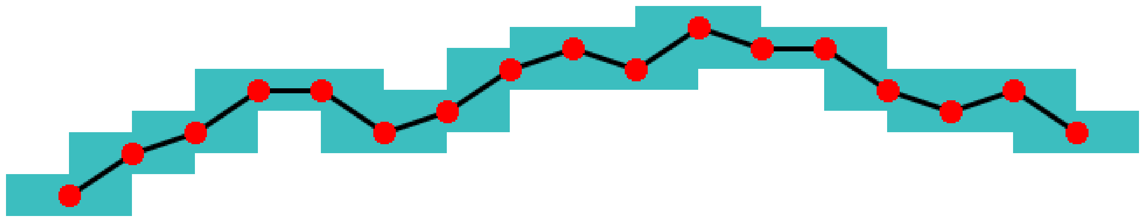


Figure 1.3: Envelopes are the union of all bounding rectangles

ing poor matches before performing more precise measuring, but as a representation for the unknown errors existing within a time series and as a facilitation for the search with uncertainty over an open-ended search space.

1.5 The Apriori Algorithm

Agrawal and Srikant first introduced the Apriori algorithm for creating item sets for mining association rules from a database of transactions in 1994 [AS94]. The Apriori algorithm iteratively builds larger and larger item sets by combining items known from the previous level to occur often enough to satisfy a *minimum support threshold*.

The algorithm begins at a base level with single item sets for each item in the database. Thus, if a database contained items A, B, and C, Apriori would first

generate prospective item sets of $\{A\}$, $\{B\}$, and $\{C\}$. These prospective item sets are then checked to ensure that each set is *frequent*. At this level this amounts to ensuring that each item appears in the database enough to satisfy a minimum *support* value. Any item sets that do not satisfy this minimum metric are discarded from the prospective item sets. All those that do are retained as *frequent* item sets.

At the next level combinations between the three sets are made. Only sets which differ in only one item are valid for combination (as the new level is supposed to consist of item sets containing one more item than the sets of the previous level). As all the sets are only length one this is trivial at this point. Additionally, for convenience, items within a set are ordered to aid in matching. So for the second level, the prospective sets $\{AB\}$, $\{AC\}$, and $\{BC\}$ are created. The sets are then tested to ensure that the two items together appear enough times in the database to satisfy the minimum *support*.

The third level (and beyond) introduces a further level of optimization. Support for an item set can only decrease as more items are added. For instance, if $\{AB\}$ does not appear often enough in the database then adding C to those instance to create $\{ABC\}$ cannot possibly make them any more common. Thus any newly created item set must have all of its subsets be *frequent* or it cannot possibly be so (as opposed to only the two sets which combine to create the new prospective set). This property holds because the value of *support* is monotonically decreasing relative to the number of items in a set.

New prospects are created by choosing two subsets which only differ in the last term (for convenience of ordering). This ensures that the same prospect is only created once. For instance, $\{AB\}$ and $\{AC\}$ can be combined, but $\{AB\}$ and $\{BC\}$ cannot. However, as stated above, those subsets which do not begin with the same $n-1$ values must still be checked to ensure that they are frequent. Thus,

even though $\{BC\}$ is not used directly in determining the new prospects, it is still checked to ensure that it is frequent. After verifying that all subsets are frequent, the new prospect must of course still be checked to see that it itself is also frequent.

This manner of generating new candidates is used with the support metric they put forth for mining association rules, however the property will hold for any measure that is monotonically decreasing with the number of items. In our process we will use this property to create sets of envelopes at time offsets that satisfy the monotonically decreasing property of minimum length.

1.6 Related Work

While much related work has been done in finding patterns within sequences of nominal values, less work has been done by the datamining community in finding patterns within numerical sequences.

Han, Dong, and Yin use the Apriori property to build up partial-periodic sequences of nominal items [HDY99]. Allowing for errors is far less precise in nominal sequences as the sequences are usually considered independent and thus there is no measure for an individual item being close to the expected item. In their approach error and variability are accounted for by allowing wildcard items in the finished sequences (such as $ab * c$).

Wang, et al. construct a generalized suffix tree using regular expressions to create similar wildcard sequences with the added concept of an edit distance [WCM+94]. This edit distance specifies a maximum number of characters that a wildcard may substitute for, thereby increasing the flexibility of the end sequences in allowing for patterns that span unimportant data without being too distant to be considered related.

Related work on sequential nominal data includes that by Shalizi, Shalizi, and Crutchfield [SSC02]. They use Hidden Markov Models to learn patterns within an input string, but unlike many others endeavored to learn the states within the process rather than have them specified a priori. The algorithm was designed to be effective on a continuous stream of characters and thus evaluated only past characters (within a sliding window to limit the scope to only those most influential states). States with similar probability models were merged in order which allowed for the learning of simple automata.

The primary difference between algorithms designed to work on nominal series and those designed to work with numerical series is whether the algorithm worked from the bottom up (as is usual with nominal data) or the top down (with numeric data). The key reason for this difference is that with nominal data there are only finitely many possibilities for each step and thus combining small sequences to form larger ones or even generating sequences and then counting support is entirely feasible. Numeric sequences on the other hand have no real limits to the number of possibilities, but benefit from more easily allowing for imperfect matching.

Guralnik and Srivastava use a model similar to the one presented above in the Introduction section to calculate instantaneous events in a single series [GS99]. They accomplish this by splitting the sequence at a change-point that they slide across the sequence and fitting each piece to a combination of model functions. The point that splits the sequences into two smaller sequences with the least error to the fit is the most likely event point. The process is then repeated over each subsequence until the fitting no longer improves.

Much work has been done in quickly discarding sequences when querying for matches. Vlachos et al. [VHGK03] use bounding envelopes and rectangles to quickly determine sequences whose differences are too great to be considered a close match.

As explained above, each sequence casts a bounding envelope over all points within δ horizontal points and ϵ vertical distance. Sequences which do not fit within these envelopes are known to have at least one point further than δ horizontally and ϵ vertically from any point in the sequence forming the envelope.

Berndt and Clifford [BC96] use dynamic programming and a cumulative distance matrix to determine the distance between two sequences after time warping. The grid is constructed by finding the distance between the first point of the matching sequence and each point of the base sequence. The smallest distance is then added to the distances between the second point and each base point. For each point the lower distance is used to determine the correct time warping and the cumulative distance for this warping is used to determine if the sequences make a good match.

Chapter 2

Our Approach

In this section we first provide a general overview of our process, then outline the inputs and outputs of the process, and finally provide an in-depth description of the process followed by pseudo-code for our algorithm.

2.1 Overview

Any real-world numeric measurement is expected to have some amount of error and other noise obscuring the desired measurement. This prevents the simpler equality matching of values that is often used to mine patterns from data with nominal values. Our algorithm accounts for these uncertainties by casting a bounding envelope around each time series sequence to represent possible positions for the desired measurement. These envelopes provide bounds for the probable locations for the desired measurement. Thus a larger envelope represents a stronger chance of containing the desired measurement.

This concept means that two when two envelopes intersect they both contain some of the same probability space to describe some of their points. Furthermore, since a greater area indicates a greater chance that the actual measurement exists

within the area, the probability that two envelopes are based off of the same desired source measurement increases with the area of intersection.

Since these these measurements and envelopes exist in time and actual events could be shifted both horizontally and vertically relative to each other, envelopes must be intersected over all possible horizontal and vertical shifts. Time coordinates are assumed to be integral and thus it is easy to have complete coverage for comparing the results horizontal shifts. Value coordinates, however, exist on a real-valued scale. Thus to find the optimal vertical shift to provide the greatest area we create a special tree structure covering ranges of shifts in which the same linear function (on shift) is used to determine area. These ranges can then be evaluated to determine which function provides the greatest area.

When combining envelopes we use a modified version of the Apriori algorithm to generate unique *envelope-offset combinations*. These envelope-offset combinations are a set of envelopes, each with a time offset relative to the first envelope in the set. These combinations are intersected to create new *template envelopes* to describe common patterns.

Combinations are only considered *valid* if their intersections yield an envelope that has a continuous set of points that have non-zero area with a length greater than a user-input minimum length. This prevents very small coincidental combinations from being considered valid. Since this envelope length property can only decrease, larger combinations need only be considered if their sub-combinations have already been found to be valid. Envelopes can also be considered invalid if a sub-combination does not exist because it is known that an intersection of sufficient length is impossible.

Thus our algorithm begins by creating combinations of two envelopes at all valid horizontal shifts and proceeds to build larger combinations by merged two lower

level combinations that differ only in one envelope-offset contributor. This allows for all valid combinations to be found without searching through space known to be invalid.

This process yields a large set of all possible valid envelope intersections. These envelopes are then ordered according to a user specified value function. This aids in that even though every possibility is returned certain results that might be more important are filtered to the top. The user can then choose how many of those results are useful.

2.2 Input and Output

The algorithm requires 5 pieces of input data from the user:

- A set of *time series* instances, $TimeSeriesInstances$.
- A horizontal bounding box range, δ.
- A vertical bounding box range, ϵ.
- A minimum valid envelope length, $MinLength$.
- A minimum *time shift* to use when combining an envelope with itself, $MinTimeShift$.
- $Value(envelope, contributors)$, a function determining the relative value of an *envelope* based on the envelope and the *contributing envelopes* that were intersected to create it.

The results of the algorithm are a set of *template envelopes.* These envelopes each represent a common area within the set of envelopes created from the time series instances and thus a description of a common pattern.

2.2.1 Input

The first parameter, $TimeSeriesInstances$, is a set of time series along an integral time axis. Each instance represents a series of datapoints progressing in time. Each datapoint consists of an integer time coordinate and a real-valued measurement coordinate. As these times coordinates are only used relatively, the numbers themselves do not matter as long as they preserve the relative distance between points. Additionally this means that sequences with no explicit time coordinate can be used as well by mapping each successive value to an increasing integer (see Fig. 2.1).

Sample Time Series				
(1, 27.37)	(2, 27.52)	(3, 27.46)	(4, 27.21)	...
27.37	27.52	27.46	27.21	...
(37, 27.37)	(38, 27.52)	(39, 27.46)	(40, 27.21)	...

Figure 2.1: The three time series all have the same temporal distance between the same-valued points, thus they are equivalent within our algorithm.

While nothing in the algorithm innately requires that the times be integral finding a best intersection between two envelopes would be far more computationally expensive and may become somewhat infeasible for longer sequences with non-integral time values. Using integral time values also greatly simplifies storage. However, the principles which form the basis for this algorithm do hold for real-valued time coordinates and thus the algorithm is also applicable for them.

In most cases each of these time series will represent a separate data instance within some greater dataset (i.e., companies in a stock value dataset), but a single instance can be broken down into smaller periods to focus the algorithm on finding patterns within a single sequence (i.e., breaking a single stock price into year long instances).

The second and third input, δ and ϵ, are parameters used for creating the bound-

ing envelopes and thus govern the tolerance of the algorithm. They represent the expected uncertainty due to error and noise within the data. While we present δ and ϵ as arbitrary but fixed here, values that are variable relative to time or specific to each sequence (if some sequences are known to have more of less associated error) are within the capabilities of this algorithm. This is the case because once the envelopes are created it is not necessary to create new envelopes later. Fixed values are used here merely for simplicity of explanation.

δ measures how far the bounding box extends along the time axis to each side of the source point. This allows for leniency along the time axis but should only be used when the dataset is known to have some shrinking and stretching in time. The nature of δ means that it allows points to be entirely subsumed by the points next to them such that the algorithm is only actually guaranteed to use 1 out of every $\delta + 1$ points in performing its matching (see Fig. 2.2). This arises because if for point P there is both a higher and lower measurement within δ time points those values will be used for creating the envelope at P. Unless the sequences call for time warping this value should usually remain at 0.

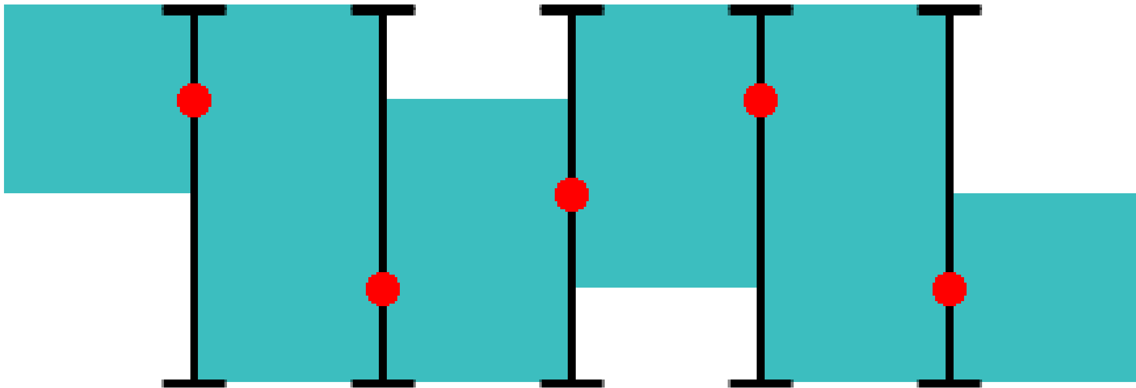


Figure 2.2: The middle point has no effect on the envelope created ($\delta = 1$)

The ϵ value is not nearly so dangerous as it is applied to every point evenly and

small increases in its value do not drastically change the shape of the constructed envelope. Since ϵ is simply applied evenly it will never change which points are used in the envelope creation, merely the height of the envelope. Since this value provides horizontal flexibility it governs how closely two sequences' shapes must correspond in order to be considered a good match. Since some amount of area is required in order to consider two envelopes as overlapping this value must be greater than 0.

The next two inputs control the suitability of envelopes created by the algorithm, the *template envelopes* that represent the common patterns found within input time series. The first value gives a minimum length at which a pattern is considered *valid* while the second is a value function used to determine which envelopes are more desirable. $MinLength$ is the minimum number of points required in any valid envelope. It helps ensure that results are not random coincidence by discarding good matches that are simply the result of chance. $MinLength$ should have a value of at least 2 (since single point envelopes are entirely trivial) and likely even higher. Since the process of intersecting envelopes can only reduce the total length of a template envelope the algorithm terminates when the minimum length can no longer be reached and thus higher values for this input will shorten the algorithm's processing time.

$MinTimeShift$ dictates how far an envelope must be shifted to be considered a possible combination with itself. This helps to prevent an envelope with a sustained pattern from intersecting itself many times without really changing the shape. When in doubt this value should usually be set to be at least equal to the $MinLength$, but only absolutely needs to be at least 1. Higher numbers for this value will reduce the number of items created initially and thus may reduce the total running time by removing uninteresting combinations from the start.

The last parameter is a function designed to rank the resulting template en-

velopes in order to display the most valuable results first. Since a very large number of template envelopes are created it is essential to rank those template envelopes. $Value(envelope, contributors)$ is a function that takes an envelope and the envelopes intersected to create that envelope and returns a relative value for that envelope. It determines the qualities that are most desired by the user in a template envelope. For instance, since an envelope created from only two other envelopes could simply be coincidence, the Value function may return higher values for envelopes with larger numbers of contributors. Similarly envelopes themselves may be evaluated on different criteria. In some situations the length of sequences may be important while in others the area or the average height (area divided by length) would be most desirable.

The function should return a positive value used to rate the envelope, with higher values being more desirable. Generally this function should be monotonically increasing relative to the number of contributors unless there is some reason the user wishes to favor results created with less data.

2.2.2 Output

The results of the algorithm come in the form of a set of *template envelopes* representing common patterns within the time series set. These envelopes are the intersection of envelopes generated from the instances and the other inputs described above. Each template envelope expresses a common shape among the *base envelopes* (the envelopes created directly from the input time series). It is our assumption that each of these shapes is frequent because there is some common event that causes the same output in multiple places. Thus these template envelopes can also be referred to as *event templates*.

In order to determine if a new time series contains the event template the se-

quence must first be converted to an envelope and then intersected with the template envelope. If the new envelope completely encompasses the template then the time series is considered to contain the event template (see Fig. 2.3). Since the envelopes themselves have been shifted horizontally and vertically in order to create these results, their time and measurement values are only relative to each other. Thus the template must also be shifted across the new envelope when trying to determine if the envelope encompasses the template.

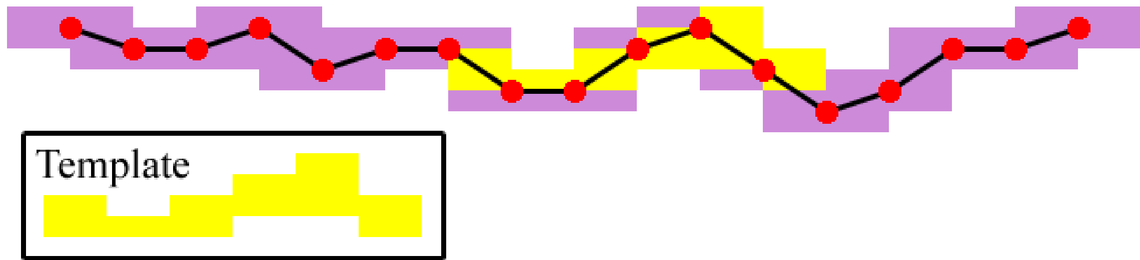


Figure 2.3: The time series envelope above completely encompasses template envelope

2.3 In-Depth Process Description

In this section we provide an in-depth description of our algorithm. We will first describe how the algorithm creates the initial envelopes from the time series data input by the user. Next we provide a description of the special algorithm used to find the best intersection between two envelopes. After this we explain how we use the Apriori algorithm to combine envelopes in order to find template envelopes. Finally we give a description of the result generation and ordering of our algorithm. Additionally, in the section 2.4 we provide pseudo-code for each portion of our algorithm.

2.3.1 Creating the Initial Envelopes

For each input time series, a bounding envelope is created by combining bounding rectangles cast by each point in the series.

Bounding rectangles represent an area capturing the possible locations for a point after accounting for errors. A rectangle is created from all points that are within δ on the timescale and ϵ along the value scale (see Fig. 2.4).

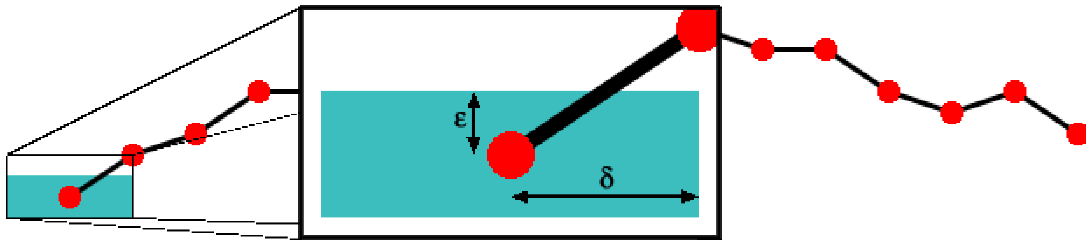


Figure 2.4: Creating a bounding rectangle

Each point is considered to be only an uncertain observation of the true output of some event. The δ value represents the point's possible uncertainty in time and the ϵ value represents the point's possible uncertainty in measurement. Thus after removing these uncertainties, a point (t, y) could actually exist anywhere between $(t - \delta, y - \epsilon)$ and $(t + \delta, y + \epsilon)$. Thus the bounding rectangle represents its possible location without noise and error.

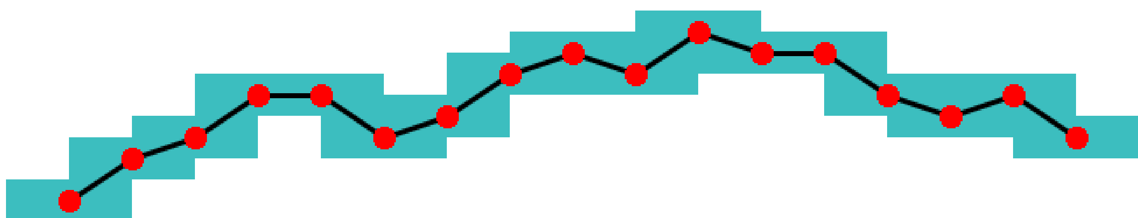


Figure 2.5: Envelopes are the union of all bounding rectangles

Once all of the bounding rectangles have been calculated, they are unioned together to form the bounding envelope (see Fig. 2.5). This bounding envelope ensures that every point within it is at most δ points along the time axis and ϵ along the value axis from a point in the original sequence. Thus, any two sequences whose envelopes intersect at some number of time points can be considered to be potentially equal at those time points. Since the envelopes dictate that, within the bounds of uncertainty, every point within each envelope could be the true location of the source point, if two envelopes intersect then the two time series that they were generated from could both share some number of source points. This forms the basis for our algorithm.

Principle. *Since an observed sequence representing some phenomenon includes some amount of uncertainty and we represent this uncertainty by bounding envelopes, any two sequences whose bounding envelopes intersect at each of their relative time positions have the potential to represent the same base sequence and thus the same phenomenon.*

Since we are only making comparisons at the integral time coordinates the created envelopes can be simplified by taking only high and low values at each time position (see Fig. 2.6). This can be done because envelopes are only ever compared at these time indexes and overlaps occurring between points are defined by the surrounding points. This reduces the problem from a comparison of areas, or even a comparison of linear ranges to a simple comparison between high values and low values. The algorithm will function without this assumption and optimization, but such a change drastically increases the amount of processing required and similarly increases the storage for each envelope.

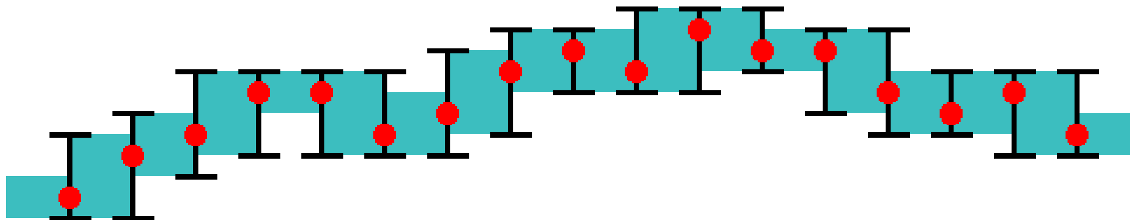


Figure 2.6: Only the vertical values at each relative time points are needed to represent the bounding envelope

2.3.2 Finding the Best Intersection Between Two Envelopes

Intersecting two envelopes is not a trivial process. Because relative positions are all that matters when finding similar shapes within sequences each intersection must consider the full scope of horizontal and vertical shifting to align the two envelopes in every possible position. Since horizontal shifting is based on an integral scale simply looping through those shifts is sufficient to determine the best intersection. The vertical scale however is real-valued and thus not so simple. Since the function governing the overlapping area for just a single pair of ranges is discrete, simple mathematical maximization is not possible.

The Intersection Area Function for Two Blocks

As shown in Figure 2.7, there are four basic functions making up the overall area vs. vertical shift function: no area (at each end), linearly increasing area, constant area, and linearly decreasing area. While the shift is such that the envelope being shifted (the dark/blue envelope) is entirely below or above the stationary (light/green) envelope (sections 1 and 5) the area is constant at 0. Similarly while one box completely encompasses the other box (section 3) the area is equal to the maximum intersection, which is a constant equal to the smaller box's area (which does not

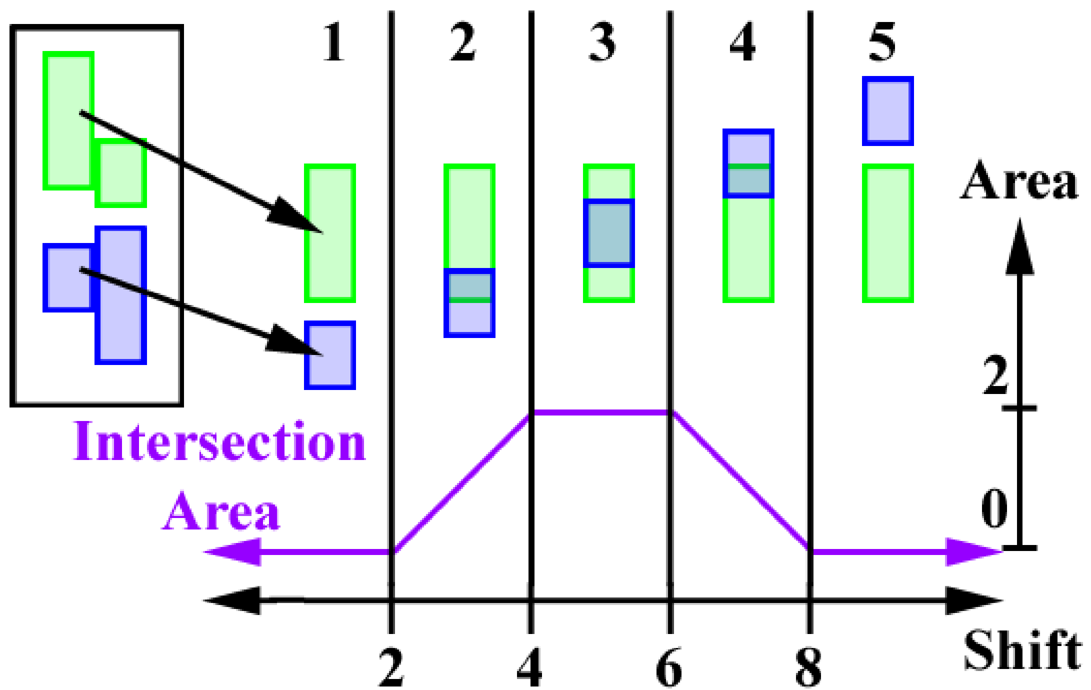


Figure 2.7: The first blocks (representing the first time point) of two short envelopes (shown in the upper left corner) are shifted vertically against each other. As two boxes are shifted the area of their intersection creates a piecewise function.

change throughout the optimization).

The more complex cases occur when the two boxes only partially overlap each other. While the moving box is entering the stationary box but still only partially matching (section 2), the area is a determined by a linear function starting at 0 and ending at the known maximum intersection. This function has a slope (the coefficient to the variable shift value) of +1. Lastly, while the moving box is partially matching the stationary box and leaving it (section 4), there is a similar but decreasing linear function which has a slope of -1 instead of the above +1.

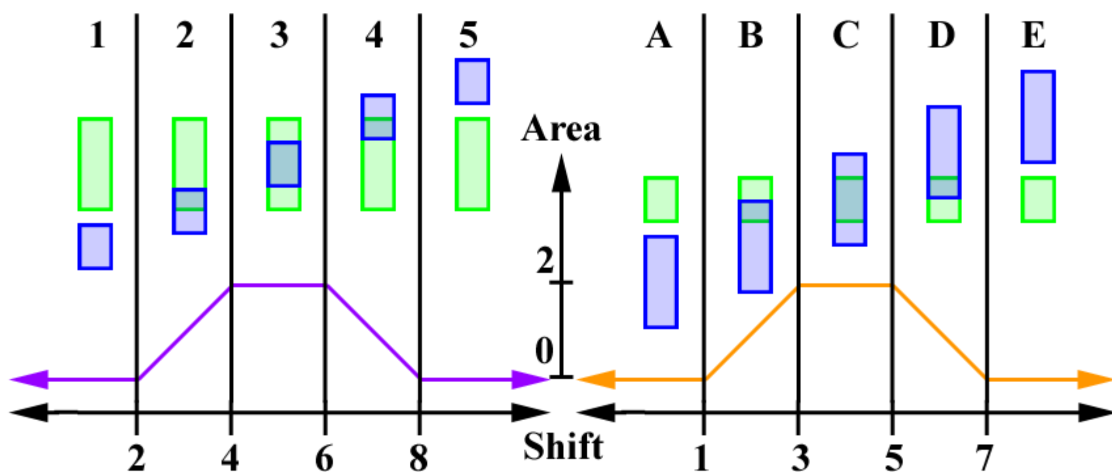


Figure 2.8: This figure shows both blocks of the short envelope shown in Fig. 2.7 being intersected individually (the first time point is on the left, the second on the right). Since a shift value is applied to envelopes as a whole the two area functions work on the same shift scale and can be added together to find a composite area function for the total area.

The ranges and values for the above functions can be directly calculated given two boxes. This allows us to split each overlap into 5 separate sections, each with a continuous function governing its area (see Fig. 2.8. Since the sections for each point of comparison within the two envelopes all exist on the same time scale these sections can be combined together to find the total area of a series of point ranges. In combining sections their shift ranges may need to be split into small ranges so that

the functions remain continuous (see Fig. 2.9. The final result will be some number of sections, each with a continuous linear function governing the total area within that shift range. Given a continuous linear function a determining the maximum shift is a simple matter.

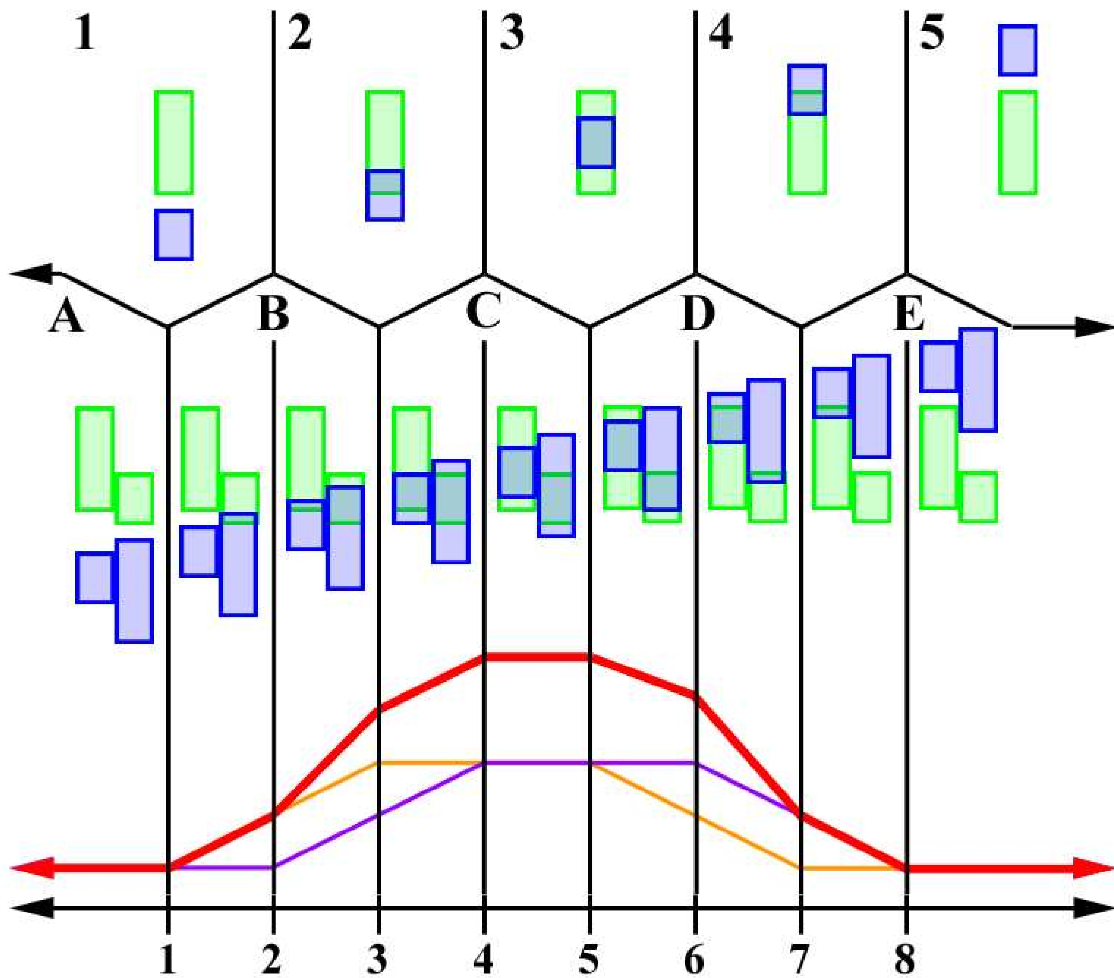


Figure 2.9: The area functions created from intersecting the two blocks (from Fig. 2.8) are combined to create a unified area function which is split into subsections with simple linear functions. The purple/dark-thin line represents the intersection of the first point of each envelope, the orange/light line represents the intersection of the second pair of points, and the red/dark-thick line represents the combination of the two.

Creating the Area Function Tree

In order to efficiently calculate the functions which dictate the total intersection of two envelopes we created a special tree structure with a number of beneficial properties. We describe the below the creation of the area function tree, then explain the valuable properties that this structure gives to us, and then finally the time complexity required to build our tree.

The root of the tree is a zero area space that extends from $-\infty$ to $+\infty$ (see Fig. 2.10). When the first box of each sequence is evaluated that node will have 3 children added (since those ranges with 0 area are unimportant). When the next pair of boxes is added, if one of its sections overlaps two sections from the previous boxes the new section is divided up into two smaller sections and the matching pieces are added to each of the previous sections it overlapped. Thus the very bottom of the tree will likely contain very small sections, but the area function for each section will contain no discontinuities.

The one special case in this tree is that if a section does not exist within any of the shift ranges of the previous level (even if it does match a range from a level before that) then a new node off of the root is created (as the root node serves to indicate that within its children's shift ranges the previous boxes did not overlap rather than that it is the start of the sequence). This allows us to find only contiguous overlapping sections. If discontiguous overlaps are desired then simply adding the 0 area end sections will accomplish this. We chose to only use contiguous areas because they are more likely to represent a single pattern. Separate patterns that appear in multiple sequences are more likely to represent a relation between two events rather than a single event with a break in time.

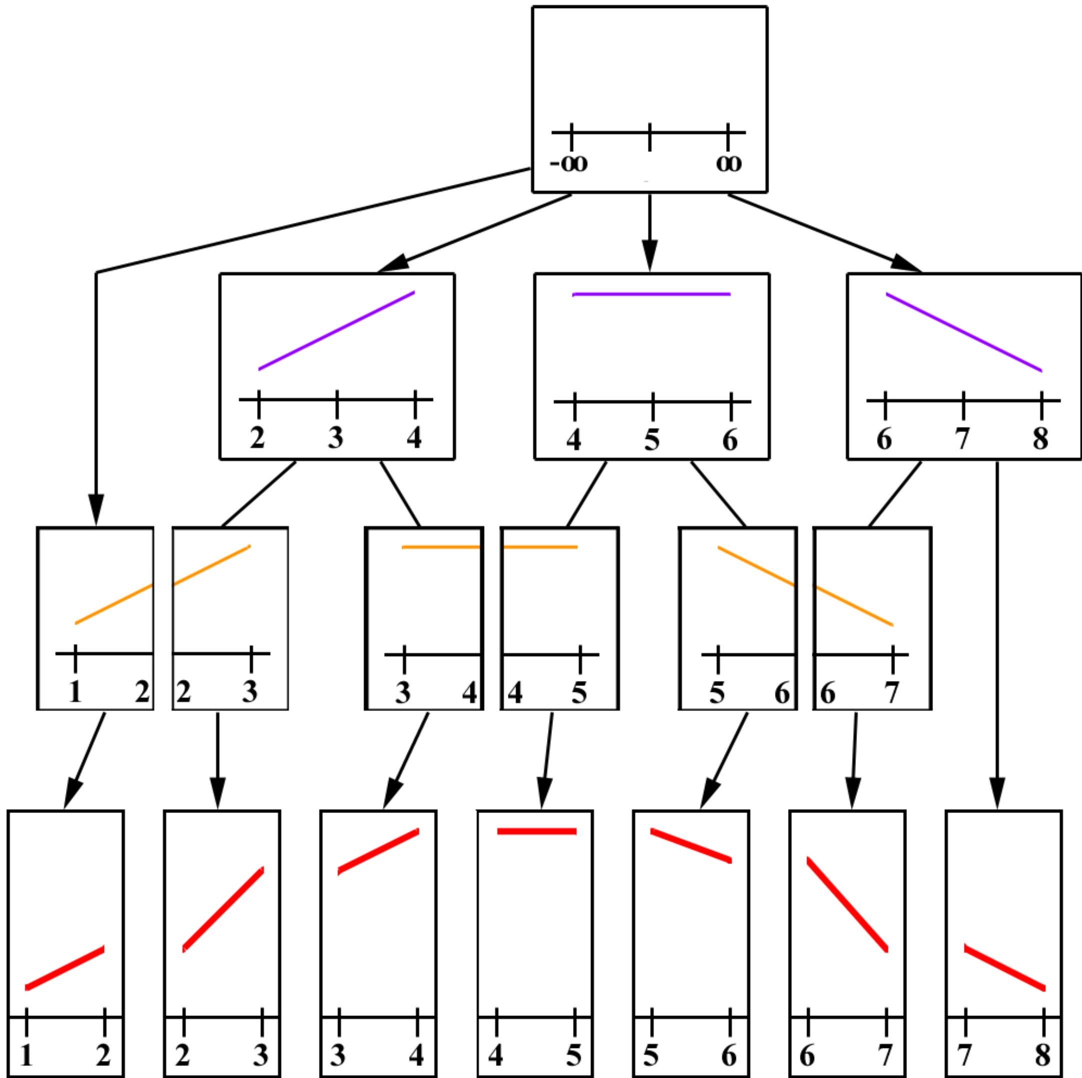


Figure 2.10: The individual sections of the functions are added together in a tree structure. The first level is the 0 area root, the second adds in the first box and the third adds in the second (and shows the total area function).

Properties of the Tree

The tree that is generated has several useful properties both for storage and searching. First, since the value ranges represented by each node within the tree get progressively more specific, in order to find a node (or nodes) that covers (or is covered) by a specific range the tree can be searched from its root downward, thus decreasing the number of nodes that need to be searched (relative to iteratively searching all leaves). Conversely all the information for previous blocks (which is used when creating an envelope from a node) is stored in previous nodes rather than repeating information for many leaf nodes. The data storage and searching is thus very efficient.

Additionally, since each node represents adding a block to the existing area, the area can only increase. Thus, to find the greatest area one needs only evaluate the area at the leaf nodes. Additionally the length of the envelope represented by a node is equal to its depth within the tree.

The ordering of the child nodes also provides another interesting property. Nodes with no children will only appear at each edge of a level. This occurs because each level is created by adding 3 consecutive shift ranges (with no internal gaps) with nothing to either side. Thus any node with no children is guaranteed to have nodes also with no children to one direction or the other. Also, if searching a level to add new ordered ranges (like a new level), those nodes from the previous level which occurred before the first range added will automatically be ineligible for any subsequent ranges and any range that falls after a range already past the end of the previous level will also fall after it.

Properties of the Area Functions

The linear area functions themselves also have some convenient properties. Since the functions for a single block intersection have either -1, 0, or 1 as slope all further slopes will also be integral. Furthermore an increasing and decreasing slope will cancel each other out to 0. Since a slope of zero within a range indicates that the actual shift within that range is inconsequential to the actual area, this means that an area combining an equal number of increases and decreases is also independent of what shift (within the range) is used.

Determining the maximum area within a range is a simple matter. If the slope is positive then the upper bound yields the greatest area. If the slope is negative then the lower bound instead yields the greatest area. If the slope is 0, as stated above, the actual shift does not affect the resulting area (obviously only so long as it is within the bounds of the range). In this case we select the mid point as the best shift since it is known to be furthest from other possibly decreasing segments and best shift only matters for the range with the highest area, so this will only move it away from decreasing areas. The middle point also makes for the most equal contribution from both the increasing and decreasing area functions so that neither is favored in further matching.

The Best Intersection

Thus, after the tree is completely generated, the leaves of the tree (having the greatest possible area) are then individually evaluated based on their area function and bounds to determine which contains the largest possible area. This node can then be used to generate the best intersection.

Time Complexity of Tree Creation

Theorem: Complexity of area tree creation algorithm. *Given two envelopes of length n, our area function tree contains at most $2n^2 + n + 1$ nodes, and its construction requires at most $2n^2 - n - 1$ comparisons.*

Proof: The number of comparisons and the number of nodes in an area function tree are innately connected. The total number of nodes is the sum of the total nodes at each level, while, in the worst case, the number of comparisons used to create each level is related to the number of nodes in the previous level. Thus, our first goal is to determine the upper bound for the number of nodes at a given level of our tree.

Consider a level of our area function tree, l, and call the number of nodes in the level $s(l)$. At this new level, 3 ranges are being used to create the new nodes. Unless a range is split across two nodes from the previous level, each range will create a single node and 3 is therefore the best case value. More nodes are created when the range must be split so as to fall beneath multiple nodes from level $l - 1$ (or from the root node if a range falls partially outside of the previous level's nodes). When a range spans across the border between two nodes of the previous level it is split from 1 range into 2 ranges. Thus, every border spanned adds 1 to the number of nodes created in the new level. If there are $s(l - 1)$ nodes in level $l - 1$, then there are $s(l - 1) + 1$ borders that can be spanned by one of the three new ranges being added. Therefore, the size of the new level, $s(l)$, is equal to $3 + s(l - 1) + 1$ or $s(l - 1) + 4$.

This recurrence simplifies down to $s(l) = 4(l - 1) + s(1)$ and given that $s(1) = 3$, $s(l) = 4l - 1$.

$$s(l) = 4l - 1 \tag{2.1}$$

This result can then be used to determine the maximum number of nodes in a tree with n levels by summing the number of nodes in each level.

$$\sum_{l=1}^{n} s(l) = \sum_{l=1}^{n} 4l - 1 = 2n^2 + n \tag{2.2}$$

The number of comparisons required to create each level is also based on the number of nodes in the previous level. In order to create level l we must compare each of the 3 new ranges to the nodes of the previous level. Fortunately the method in which we store our data helps to prevent unnecessary comparisons through the ordering of nodes. Since both the previous nodes and the new ranges are sequentially ordered we know that we can skip any nodes that fall completely before the end of the previous range.

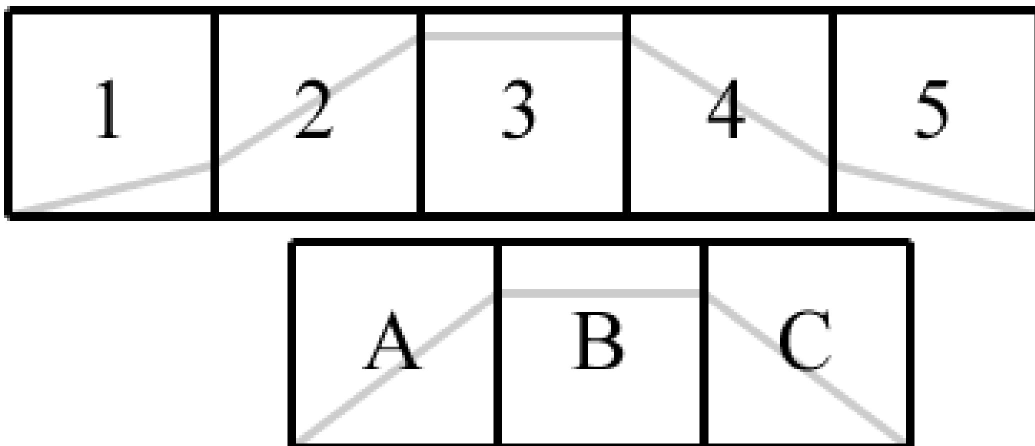


Figure 2.11: Boxes 1-5 represent a previous node level, while boxes A-C are new ranges being compared to them.

For instance, given the three ranges A-C in Fig. 2.11, range A will first compare to nodes 1, 2, and 3 and then B will skip right to 3 since nodes 1 and 2 have already been passed (while 3 has only been partially covered). The complete process goes

as follows:

1. A compares to 1, no match
2. A compares to 2, partial match
3. A compares to 3, partial match, A finished
4. B compares to 3, partial match
5. B compares to 4, partial match, B finished
6. C compares to 4, partial match
7. C compares to 5, partial match, C finished, no more comparisons needed

In a more general sense, a node from the previous level will only be compared to multiple new ranges if the node spans across a border between ranges. Therefore, since 3 ranges will have 2 borders between the ranges, the number of comparisons at any level, $c(l)$, is equal to the number of nodes in level $l-1$, $s(l-1)$, plus 2.

$$c(l) = s(l-1) + 2 = 4l - 3 \tag{2.3}$$

The total number of comparisons can thus be determined by summing the totals for each level.

$$\sum_{l=2}^{n} c(l) = \sum_{l=2}^{n} 4l - 3 = 2n^2 - n - 1 \tag{2.4}$$

2.3.3 Building the Envelope Combinations

Once the envelopes have been prepared the true mining process can begin. To do this we used a variation of the Apriori algorithm developed for finding association rules. While the purpose of the standard Apriori algorithm is to create item sets such that

those sets all meet a certain minimum support metric, our goal is to create sets of envelopes that when intersected maintain a minimum length. *Envelope combinations* are sets of envelopes, each with a relative time offset that are intersected with each other to form a representative template envelope. Envelope combinations act as the item sets in our version of Apriori.

Additionally these envelopes can be intersected at a number of different relative time offsets such that intersecting envelope A with envelope B at time offset 1 is different from intersecting it with time offset 3 (see Fig. 2.12). Items are thus more closely associated with such an *envelope-offset pair* rather than just an envelope.

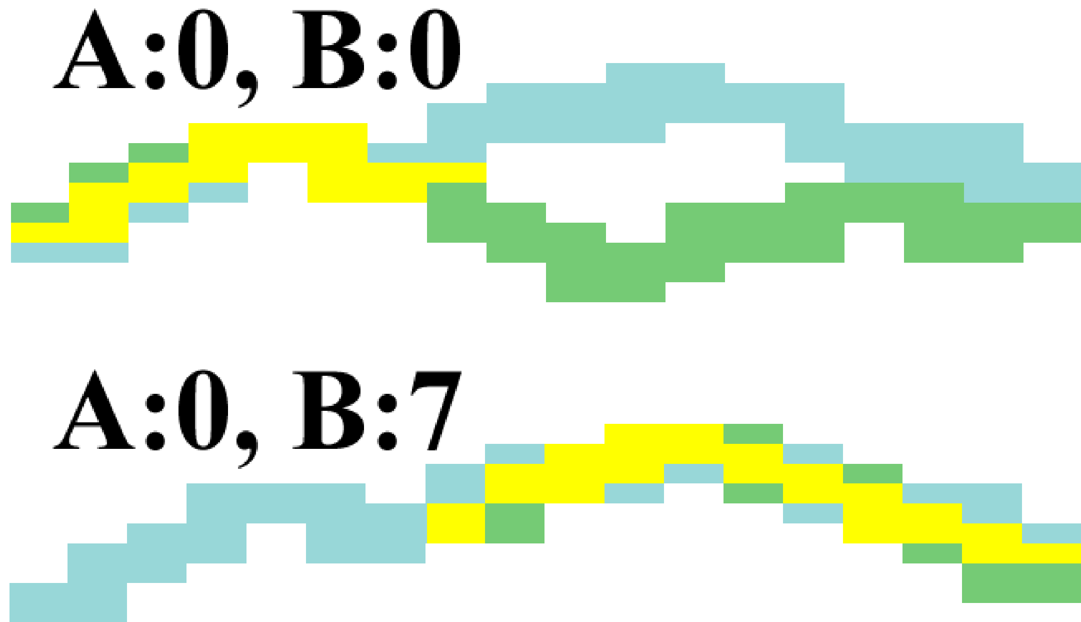


Figure 2.12: Two different envelope combinations created from the same two envelopes

As with standard Apriori, these envelope-offset pairs are given an ordering to simplify the combination process. Envelope-offset pairs are first ranked according to the (arbitrary) ordering in which the time series appear in the dataset. Then

for those created with the same envelope the numeric ordering of the offset is used. Thus, if envelopes are in the dataset in the order A, B, C, then B with offset 5 (designated $B : 5$ hereafter) is ordered after any A envelope-offset or B envelope-offset with an offset less than 5 and before any envelope-offset pair with an envelope of C or an envelope of B and an offset greater than 5.

The first level

The process begins with a first level consisting of envelope combinations containing only a single envelope. Each envelope is given a fixed offset of 0. Since all offsets within an envelope combination set are only relative to each, a single value can be fixed without changing the relative value. We do this with the initial envelopes to aid in later matching.

Since we wish to find only envelopes that meet a certain minimum length we prune out at this level any envelopes that do no meet that minimum length. For the rest of this description we consider envelopes to be *valid* when they meet the user specified minimum length.

Creating the second level

Once the first level has been pruned of invalid envelopes, the second level can be created. First each fixed envelope is combined with both itself and every other envelope of higher order (thus A combines with A, B and C, and B combines with B and C) to create a set of potential combinations. Except for the special case of an envelope combining with itself, every possible time offset is given to the added envelope. If A and B are both length 6 and the minimum valid length is 3 then A is combined with $B : -3$ (to compare positions 1-3 in A with 4-6 in B) through $B : 3$.

In the special case of an envelope combining with itself two additional conditions

apply. First the offset of the second instance must be positive since $\{A : 0, A : 3\}$ is no different from $\{A : 0, A : -3\}$ and second the shift must be greater than or equal to the $MinTimeShift$ parameter. This helps to ensure that envelopes do not combine too closely with themselves (since if that were the case straight lines would intersect themselves at many points when it is not really a new shape to take note of). Using a $MinTimeShift$ equal to the example $MinLength = 3$ from above, A would be combined with $A : 3$. It must still be remembered that the number of positions that can potentially intersect must still meet the $MinLength$ requirement. Thus if A were only length 5 a shift of 3 would leave only 2 positions to consider and no combination would be possible.

After these potential combinations have been generated they must still be intersected to determine if a long enough intersection is possible. Since the horizontal offset has already been determined by iterating over possible values, only finding the vertical offset for each of those remains. However finding the ideal vertical shift is not as simple as it may seem. The process used to determine such was described above in section 2.3.2.

Creating levels beyond the second

Creating the first two levels is a special process. Creating the levels beyond two is all done with the same algorithm. Each combination is extended by finding other combinations that differ in only the very last term. Envelopes are no longer added directly from the base set, but are instead selected from the previous level's combinations. Thus both the envelope to be added and the time offset are already known. No worry needs to be given to whether it is adding the same envelope or whether the offset is actually acceptable because it is already known to be a valid potential item.

For the two combinations to match all except the last term must be the same. This includes both the envelope and the offset. Thus the combination $\{A:0, B:3, C:-1\}$ can be combined with the combination $\{A:0, B:3, D:4\}$, but not the combinations $\{B:3, C:-1, D:4\}$ or $\{A:0, B:0, D:4\}$. Two valid combinations combine together by adding the last term of the second on to the first (and since the combination sets themselves should be ordered this will preserve the correct ordering within the new set). For instance, from the example above, $\{A:0, B:3, C:-1\}$ can combine with $\{A:0, B:3, D:4\}$ to create $\{A:0, B:3, C:-1, D:4\}$.

Once this *potential envelope combination* has been created it must be tested to ensure that it is a valid combination or sets before intersecting the envelopes to measure the intersection length. This is done by making sure that all subsets of the new potential combination also exist as valid sets. $\{A:0, B:3, C:-1\}$ and $\{A:0, B:3, D:4\}$ are known to be valid (since they were taken from the valid combination set), but the other two subsets of cardinality 3 ($\{A:0, C:-1, D:4\}$ and $\{B:3, C:-1, D:4\}$) must also exist. If they do not, it is known that they were not valid and thus the new set cannot possibly be so.

One special operation must be performed in the case where the first term does not have an offset of 0. Since envelopes that created the first level were given a fixed offset of 0 and new envelope-offset pairs are only added to the end due to the ordering within envelope combination sets, every envelope combination set begins with an envelope-offset pair with an offset of 0. All other envelopes are relative to this fixed offset and the other relative offsets.

Thus the set $\{B:3, C:-1, D:4\}$ is made up entirely of relative offsets and since it does not begin with an offset of 0, it could not possibly exist as is. Therefore, to check the validity of the subset, all of the offset values must be simultaneously shifted up or down so as to set the first offset to the fixed 0. $\{B:3, C:-1, D:4\}$ becomes

$\{B : 0, C : -4, D : 1\}$ and can then be found within the existing sets beginning with B (which all have offset 0). The terms all still have the same relative distance, but the axis is changed so as to fit with the existing sets that have a fixed first term.

Once these potential envelope combination sets have been created and their subsets have been checked the new envelope-offset pair (the one at the end) is intersected with the existing intersection. This means the intersection of $\{A : 0, B : 3, C : -1\}$ is intersected with envelope D at an offset of 4 to create the new intersection of $\{A : 0, B : 3, C : -1, D : 4\}$. If this intersection is long enough to meet the minimum length requirement then the potential combination is valid and is added to the next level of combinations. The process concludes when no new combinations are created for a level and all of the combinations are then returned.

2.3.4 Resulting Output

When the initial step can no longer create a new level of valid envelope-offset combinations the process finishes. The results are the intersected envelopes created with each envelope-offset combination set as described above. These envelopes are then rated according to the value function and returned in order. Since every combination and subset is found, lower level envelopes may end up having a better value and be returned higher in the list.

2.4 Pseudo-Code

2.4.1 CreateEnvelopes

```
CreateEnvelopes(time_series_instances, delta, epsilon)
{
```

```
  envelope_set = {}

  for each (time_series in time_series_instances)
  {
    envelope = null

    for each (time_point in time_series)
    {
      high_value = -infinity
      low_value = +infinity

      for (i = -delta to delta)
      {
        if (high_value < time_series[time_point + i].Value + epsilon)
        {
          high_value = time_series[time_point + i].Value + epsilon
        }
        if (low_value < time_series[time_point + i].Value - epsilon)
        {
          low_value = time_series[time_point + i].Value - epsilon
        }
      }

      envelope[time_point] = (low_value, high_value)
    }

    envelope_set.add(envelope)
  }

  return envelope_set
}
```

2.4.2 FindBestIntersection

```
FindBestIntersection(stationary_envelope, moving_envelope,
                                       horizontal_offset) -> Envelope
{
  best_area = 0
  best_vertical_offset = 0

  temp_envelope = moving_envelope.shift(horizontal_offest)

  area_function_tree = new area_function_node(low_shift = -infinity,
```

```
                                          high_shift = +infinity,
                                          area_function =
                                                   0 * SHIFT + 0)
previous_leaves = area_function_tree

for each (time_point shared between
          stationary_envelope and temp_envelope)
{
  range1 = stationary_envelope[time_point]
  range2 = temp_envelope[time_point]

  new_leaves = {}
  remaining_nodes = {}

  max_intersect_area = Min(range1.area, range2.area)
  lowest_bound = range1.low - range2.high
  highest_bound = range1.high - range2.low

  remaining_nodes.add(
          new area_function_node(lowest_bound,
                                 lowest_bound + max_intersect_area,
                                 +1 * SHIFT - lowest_bound))
  remaining_nodes.add(
          new area_function_node(lowest_bound + max_intersect_area,
                                 highest_bound - max_intersect_area,
                                 0 * SHIFT + max_intersect_area))
  remaining_nodes.add(
          new area_function_node(highest_bound - max_intersect_area,
                                 highest_bound,
                                 -1 * SHIFT + highest_bound))

  while (remaining_nodes is not empty)
  {
    node = remaining_nodes[0]
    remaining_nodes.remove(node)
    parent_node = FindNodeContaining(previous_leaves, node.low_shift)

    if (parent_node exists)
    {
      node.area_function = node.area_function +
                                      parent_node.area_function

      if (parent_node.high_shift >= node.high_shift)
```

```
      {
        parent_node.add(node)
      }
      else
      {
        temp_node = new area_function_node(parent_node.high_shift,
                                           node.high_shift,
                                           node.area_function)
        node.high_shift = parent_node.high_shift
        parent_node.add(node)
        remaining_nodes.add(temp_node)
      }
    }
    else
    {
      area_function_tree.add(node)
    }

    new_leaves.add(node)
  }
}

all_leaves = FindAllLeaves(area_function_tree)

for each (node in all_leaves)
{
  if (BestArea(node) > best_area)
  {
    best_area = BestArea(node)

    if (node.area_function.slope > 0)
    {
      best_vertical_offset = node.high_shift
    }
    else if (node.area_function.slope < 0)
    {
      best_vertical_offset = node.low_shift
    }
    else
    {
      best_vertical_offset = (node.low_shift + node.high_shift)/2
    }
  }
```

```
  }
  result_envelope = intersect(stationary_envelope, temp_envelope,
                              best_vertical_offset)

  return result_envelope
}
```

2.4.3 CreateFirstTwoLevels

```
CreateFirstTwoLevels(base_envelope_set, min_length, min_time_shift)
            -> EnvelopeCombinationSets
{
  for each (envelope in base_envelope_set)
  {
    if (! envelope.valid)
    {
      base_envelope_set.remove(envelope)
    }
  }

  result_envelope_combinations = {}
  potentials = {}

  for each (envelope in base_envelope_set)
  {
    for (offset = min_time_shift to envelope.length - min_length)
    {
      potentials.add({envelope:0, envelope:offset})
    }

    for each (new > envelope in base_envelope_set)
    {
      for (offset = -(new.length - min_length) to
                    (envelope.length - min_length))
      {
        potentials.add(envelope:0 + new:offset)
      }
    }
  }

  for each (potential in potentials)
  {
    potential.envelope = FindBestIntersection(potential.head.envelope,
```

```
                                              potential.tail.envelope,
                                              potential.tail.offset)
    if (! potential.envelope.valid)
    {
      potentials.remove(potential)
    }
  }

  return potentials
}
```

2.4.4 CreateFurtherLevels

```
CreateFurtherLevels(envelope_combinations) -> EnvelopeCombinationSets
{
  potentials = {}
  for each (combination in envelope_combinations)
  {
    for each (new > combination in envelope_combinations)
    {
      // combination.head = first n-1 items from combination
      // combination.tail = last item of combination
      if (combination.head = new.head)
      {
        potentials.add(combination + new.tail)
      }
    }
  }

  for each (potential in potentials)
  {
    for each (length-1 subset of potential)
    {
      if (subset.first.offset != 0)
      {
        subset = subset.shift(-subset.first.offset)
      }
      if (! envelope_combinations.contains(subset))
      {
        potentials.remove(potential)
        break
      }
    }
```

```
  }

  for each (potential in potentials)
  {
    potential.envelope = FindBestIntersection(potential.head.envelope,
                                              potential.tail.envelope,
                                              0)
    if (! potential.envelope.valid)
    {
      potentials.remove(potential)
    }
  }

  return potentials
}
```

2.4.5 FindTimeSeriesTemplates

```
FindTimeSeriesTemplates(time_series_instances, delta, epsilon,
                        min_length, min_time_shift, value_function)
{
  base_envelope_set = CreateEnvelopes(time_series_instances, delta,
                                      epsilon)
  result_combination_set = CreateFirstTwoLevels(base_envelope_set,
                                                min_length,
                                                min_time_shift)

  new_combination_set = CreateFurtherLevels(result_envelope_set)

  while (new_combination_set not empty)
  {
    result_combination_set.add(new_combination_set)
    new_combination_set = CreateFurtherLevels(new_combination_set)
  }

  result_combination_set.orderUsing(value_function)
  print(result_combination_set)
}
```

Chapter 3

Experimental Results

3.1 Synthetic Data

The synthetic data is most capable of determining the algorithms capability to function under the assumptions as to what representations an event has within a time series. Synthetic data provides us data where we know the events that exist within each time series.

3.1.1 Data Generation

The data was generated with the basis of a dataset ranging from 0 to 99, with each successive instance actually being the average of the previous instance and a random value. This represents that data usually progresses through time rather than simply jumping to random values. The dataset consists of 20 sequences each with 40 values. Within these sequences 4 events of length 10 are inserted. The template event sequences are created in the same manner as the normal data but just repeated throughout the dataset. Each point that isn't already representing an event has a 5% chance of beginning a new instance of a template. Additionally the

template is shifted upwards or downwards by a random amount, but never enough so that the maximum would be greater than 99 or less than 1. After this is done every value is shifted up or down randomly by up to ϵ (which is specified at run time).

3.1.2 Effectiveness Relative to Epsilon

For this test we generated 6 versions of the same dataset, each with a different value for ϵ, ranging from 5 to 10. These values are applied to a single randomly generated number to avoid different random generations influencing the result. Thus a point modified upward by 5 when using an ϵ of 5 will be modified up by 10 for the $\epsilon = 10$ version. The template sequences had lengths of 4, 5, 6, and 8 and include a repeated single envelope within a sequence.

Results were filtered to include only envelopes that were not used to create other envelopes. This is done to easily increase the diversity of results, but could be replaced by a better value function to favor coverage. Without such the biggest template's subsets would appear above other good results.

The results consistently show the templates being found (see Fig. 3.1). 3 of the 4 templates were consistently found completely in the results. The last was only expressed partially (two results with 5 of 6 expected instances). This highlights one of the only ways in which the algorithm isn't guaranteed to find every valid intersection. Since the algorithm to determine best vertical shift can only be performed on pairs of envelopes being intersected the early intersections can influence later intersections. This arises because A, B, and C cannot be intersected together so C must intersect with the intersection of A and B rather than with the individual envelopes. Thus when A and B are intersected without C the best area may be dependent on other coincidental matching pieces and affect which envelope is created

even though they are stripped later.

Since the valid results all existed in 4 or more sequences they true results were always above the false positives that showed up as ϵ increased. Thus many of them could be ignored simply if it was known that intersections between 2-3 sequences were too infrequent to be valuable.

	Epsilon					
	5	6	7	8	9	10
Full	3	3	3	3	3	3
Partial	1	1	1	1	1	1
False (4)	0	0	0	0	1	3
False (3)	0	0	0	1	4	9
False (2)	0	1	3	14	10	3
False (all)	0	1	3	15	15	15

Figure 3.1: Table of fully matched templates, partially matched templates, and false templates by number of contributors

3.2 Unemployment Data

For the first test a set of unemployment data for three counties in Massachusetts was mined for frequent patterns. The data contained monthly rate measurements covering the years between 1990 and 2003. Rates varied from between 1.9% and 15.1% and had no missing values.

The counties of Barnstable, Middlesex, and Worcester were chosen because the author has a degree of familiarity with each county and their commerce patterns. Barnstable County's economy is heavily influenced by summer tourists, while Middlesex County is heavily influenced by the large city of Boston, and Worcester County surrounds the large but less active city of Worcester.

The data began as a single long stream for each county but was split into yearly sequences to avoid forcing the mining process to begin by matching patterns across

counties. It was hoped that the process would discover similarities between the sequences within each county to show that it could identify known patterns (such as employment patterns unique to each county).

3.2.1 Experiments

Our first step in experimenting with this data was to normalize the unemployment data. We did this in order to place each year on an equal scale and to minimize the effect of the overall economy so that good years can compare shape to bad years.

The normalized data is much more shape dependent than the non-normalized data and this causes the Barnstable County sequences to be even more tightly tied together (see Fig. 3.2). Worcester and Middlesex County can also be seen to be very similar though normalization causes sequences with high points at the beginning and end of the year to be further differentiated from others from the those counties. This reveals a secondary trend among dates for the two city-influenced counties in that some years (presumably while the economy was increasing) caused the overall unemployment to drop. The holidays were usually either very good or very bad for unemployment.

We ran tests over a variety of ϵ and *minLength* values. The values of 0.08 for ϵ and 8 for *minLength* provided some of the best result. 3 of the most representative envelopes discovered can be seen below.

Figures 3.3 and 3.4 show the very strong grouping of Barnstable county unemployment patterns. These envelopes not only combine together to build up nearly the entire Barnstable year pattern, but also encompass a large number of sequences, thus showing that the pattern is very frequent. The first begins at position 5 (May) while the bottom starts at 1 (January). Barnstable data grouped very tightly with other Barnstable data and only very rarely was combined with either of the other

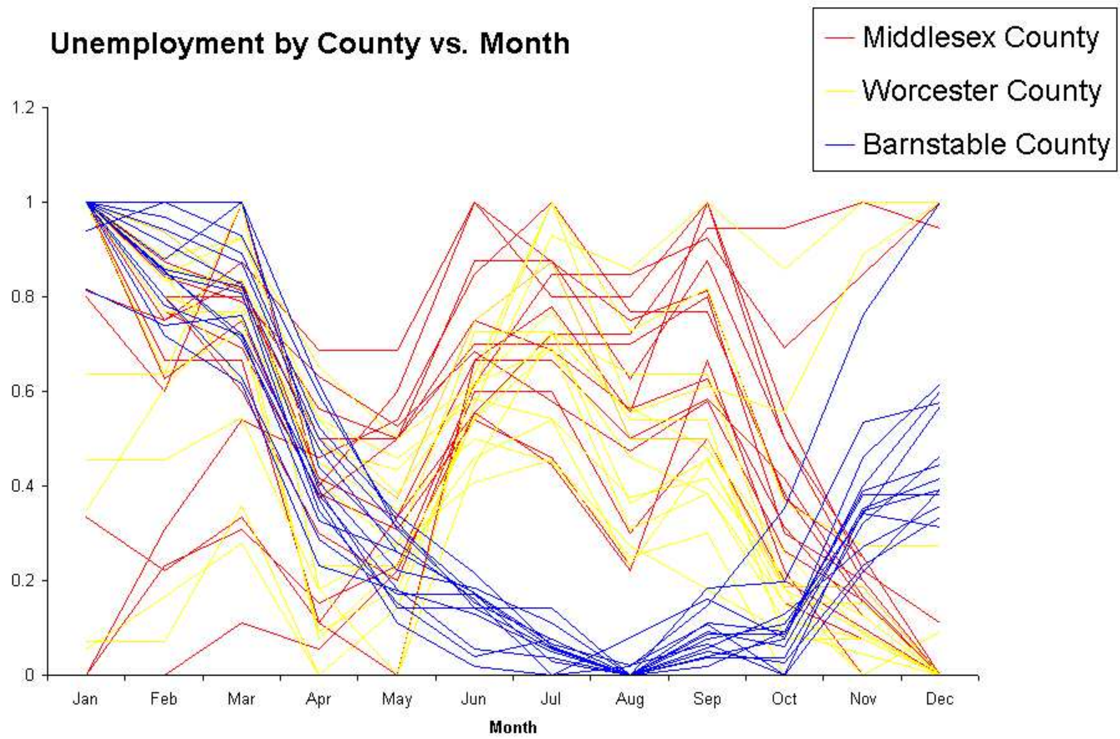
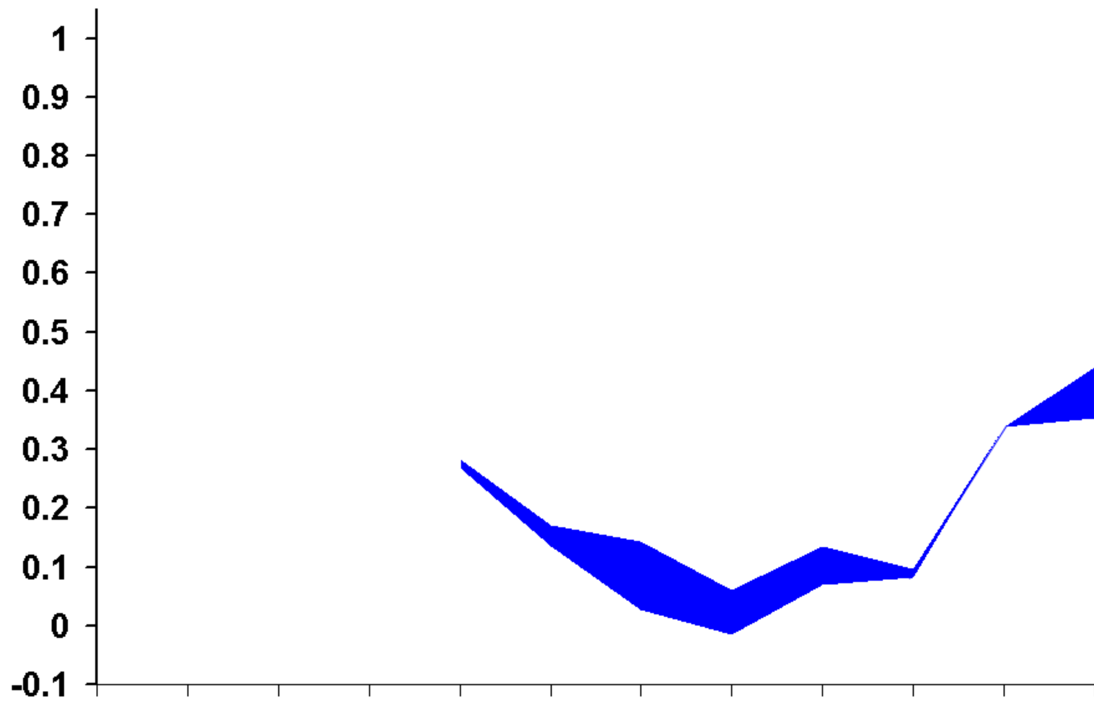
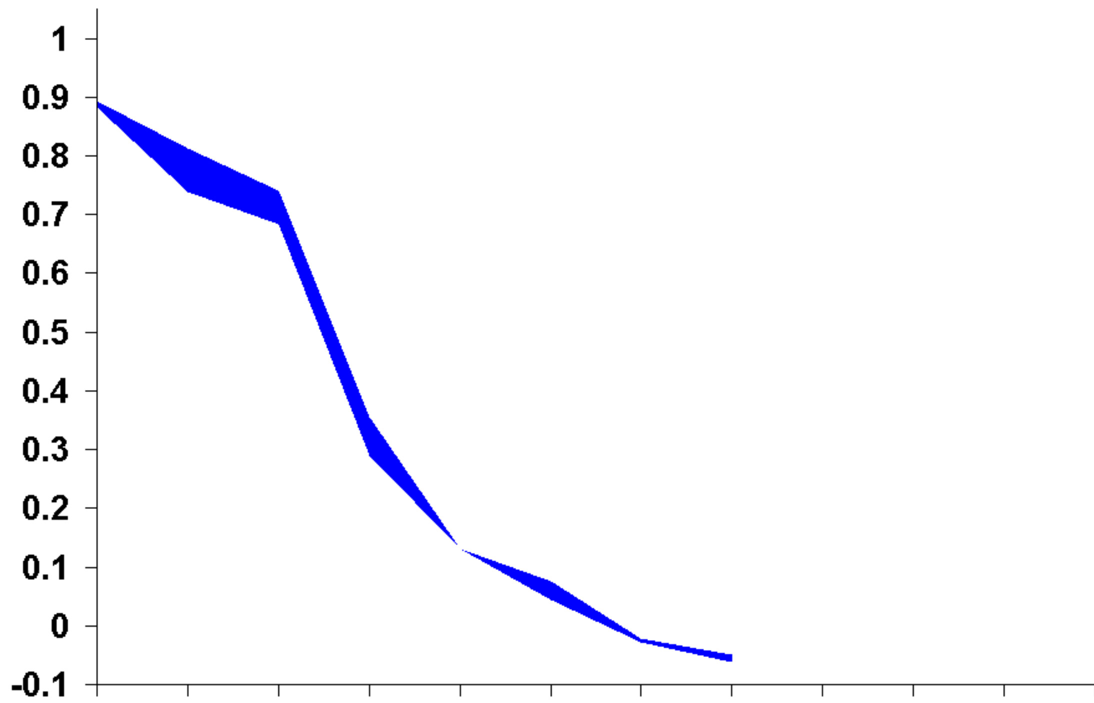


Figure 3.2: Graph of normalized monthly unemployment rate for Middlesex, Worcester, and Barnstable counties



Contributing Envelopes and Offsets:
B1991:0, B1994:0, B1995:0, B1996:0, B1997:0, B1998:0, B1999:0, B2000:0

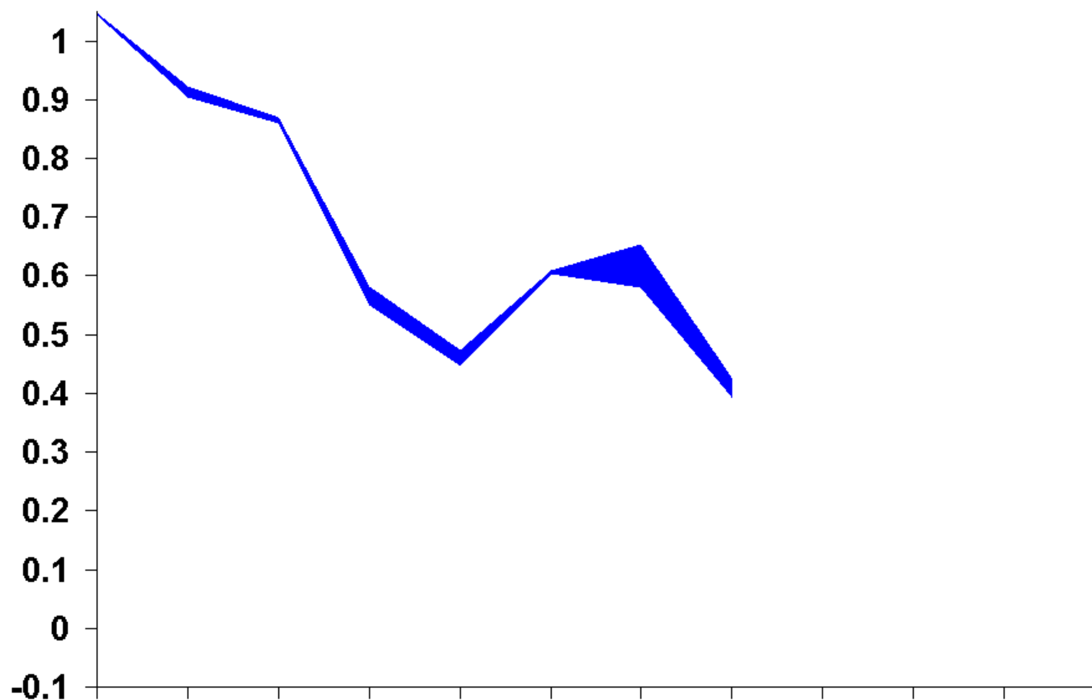
Figure 3.3: An unemployment template envelope



Contributing Envelopes and Offsets:
B1990:0, B1994:0, B1995:0, B1996:0, B1997:0, B1998:0, B2001:0, B2002:0

Figure 3.4: An unemployment template envelope

counties.



Contributing Envelopes and Offsets:
M1993:0, M1996:0, W1992:0, W1993:0, W1994:0, W1996:0

Figure 3.5: An unemployment template envelope

Figure 3.5 displays the largest grouping of non-Barnstable data instances. As opposed to Barnstable, Worcester and Middlesex county intermixed often with each other. This sequence displays the most frequent pattern found for Worcester and Middlesex counties at this ϵ value. This pattern displays another quality of the data in that it Worcester and Middlesex often compare well at similar time periods rather than throughout. This likely is indicative of an economy that is more greatly affected by the general economy than by the change of seasons when compared to Barnstable.

One of the last general points to note about all three results is that sequences in this dataset combined all with an offset of 0. This shows that the patterns evicted

are likely rather predictable and very cyclic on a yearly basis.

3.3 Monthly Stock Quotes

The last dataset we used in testing our algorithm is a 5 year monthly quote of stock prices for 4 large technology companies: Microsoft, Sun Microsystems, Intel, and IBM. We retrieved these listings from the Yahoo Finance Stock History search. Each quote was taken monthly and the timespan covered was January 1998 through December 2003 (see Fig. 3.6). For our evaluation we used the adjusted close column and normalized each time series.

3.3.1 Experiments

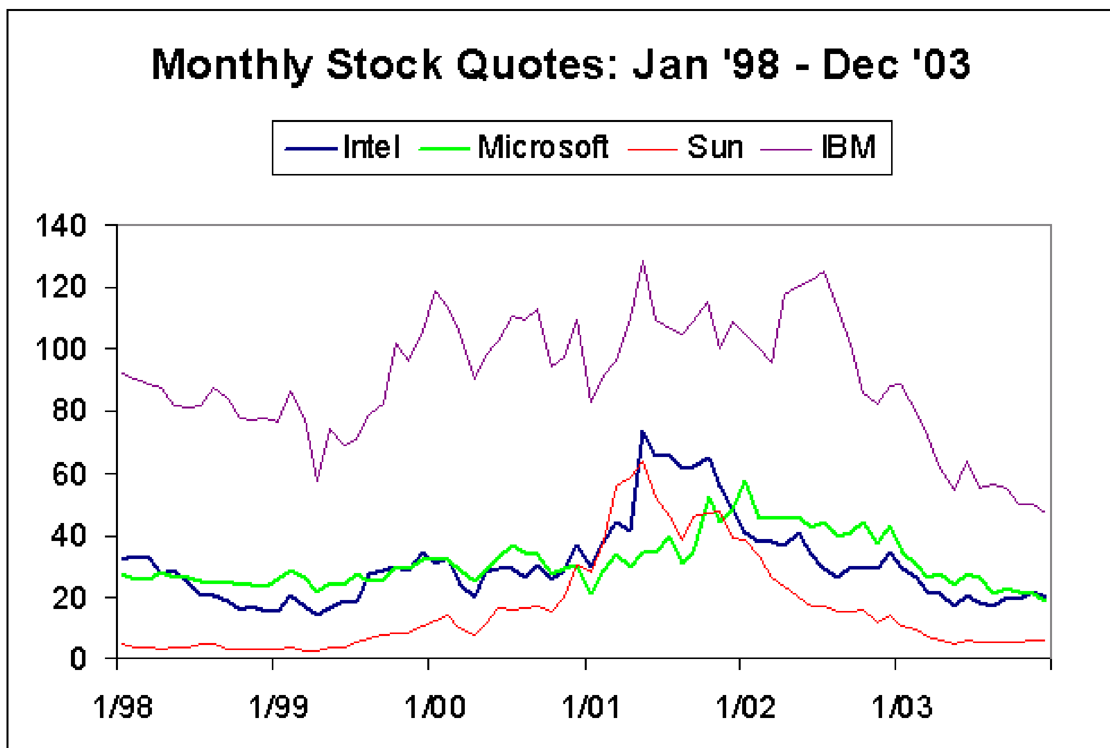


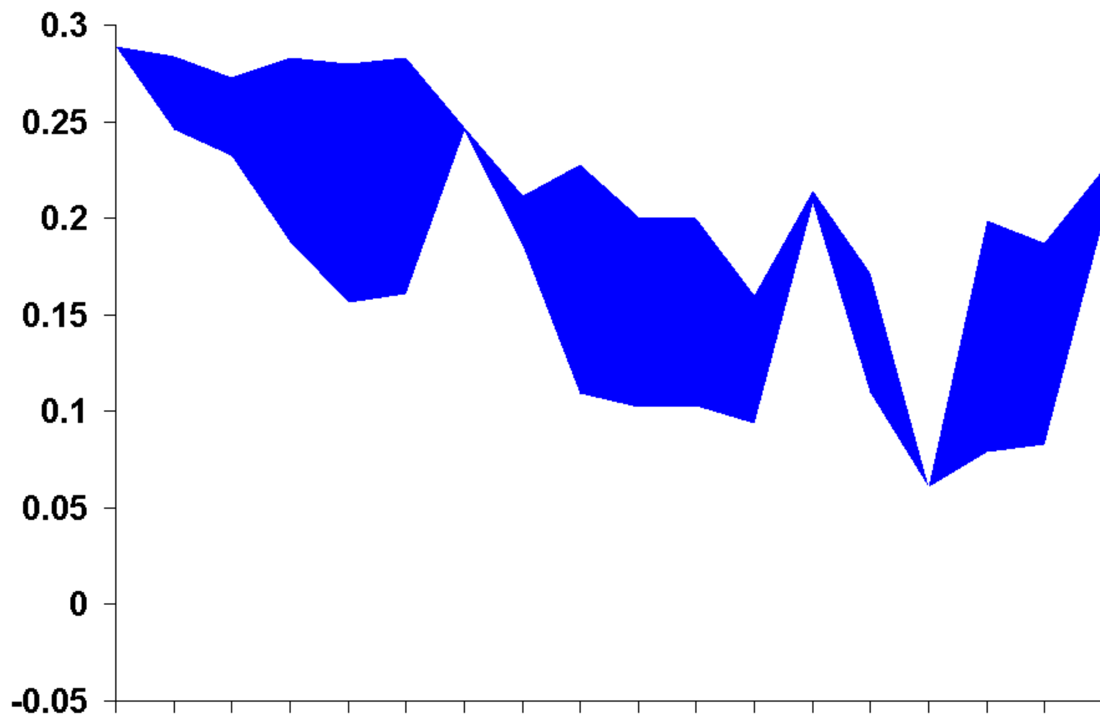
Figure 3.6: Graph of monthly stock values for Intel, Microsoft, Sun, and IBM

We again performed mining over a variety of ϵ and $MinLength$ inputs and selected one of those trials as the best result by comparing the uniqueness, length, and number of the resulting envelopes. The results below are for $\epsilon = 0.1$ and $MinLength = 18$. Thus the results below show trends over one and one half years.

Figures 3.7, 3.8, and 3.9 show the three most unique results found from mining using the parameters above, along with a possible match from the predetermined financial templates used by stock analysts. Many other results similar to the ones above were also discovered, primarily with different offsets from the same contributors, which indicates that the data does not change very dramatically (at least compared to the 10% ϵ value). The primary purpose in using this dataset was to determine if any of the pre-established templates used in stock values could be discovered by our algorithm. Each of the figures shows both a graph of the template found through our algorithm and a possible match to a stock template. Figures 3.7 and 3.9 both show reasonably recognizable correspondence to the templates given. Figure 3.8 has a small amount of correspondence to the 'Ascending Triangle' template, but did not fit well with any of the predetermined templates.

However, even though comparisons can be drawn between the shapes of the templates we found and the predetermined templates it is difficult to state definitively that our templates are truly representations of the predetermined templates. Knowing when to apply those templates is necessarily an inexact science performed by domain experts and thus an expert eye would be needed to state anything conclusively.

Contributing Envelopes and Offsets:
INTC:0, INTC:51, MSFT:52, SUNW:-1, SUNW:52, IBM:52



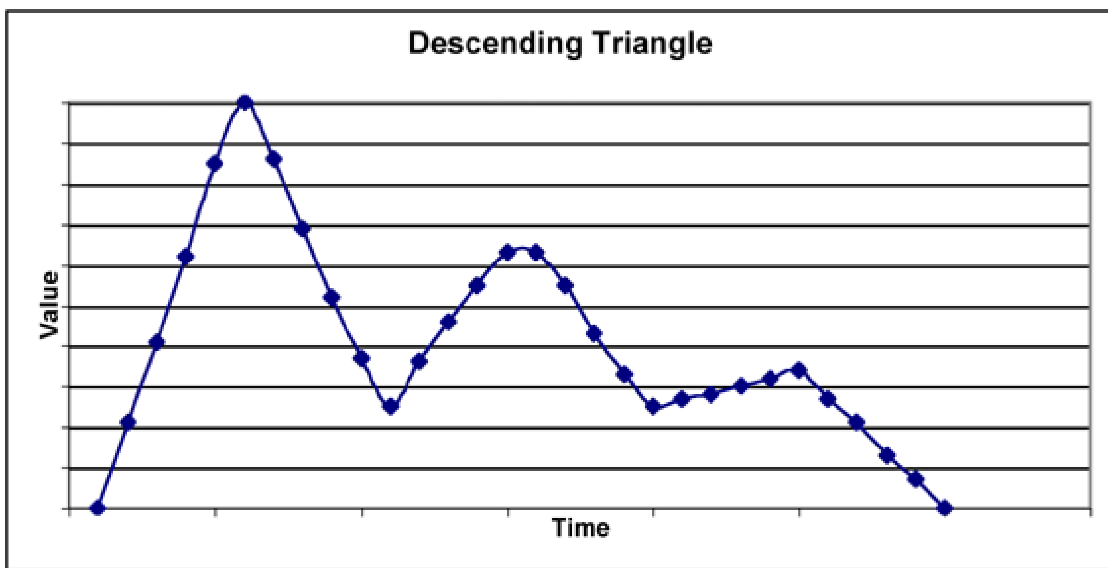
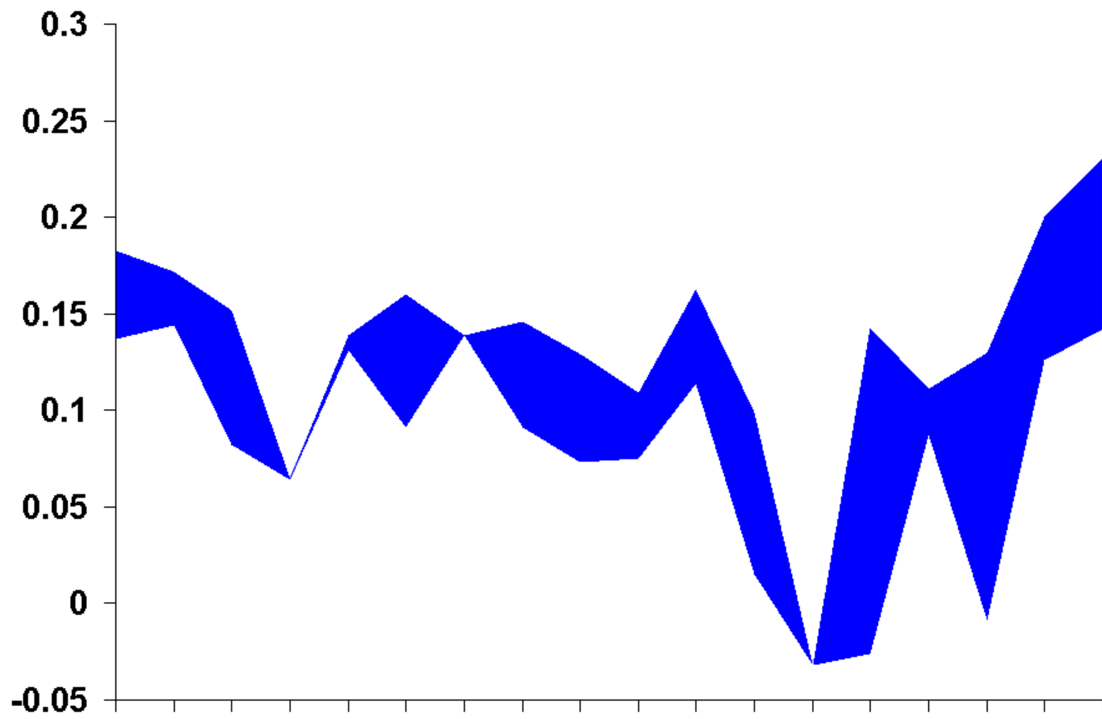


Figure 3.7: A template envelope found by our algorithm and the 'Descending Triangle' stock value template

Contributing Envelopes and Offsets:
INTC:0, INTC:34, MSFT:46, SUNW:0, SUNW:47

Figure 3.8: A template envelope found by our algorithm and the 'Ascending Triangle' stock value template

Contributing Envelopes and Offsets:
INTC:0, MSFT:-21, MSFT:-1, SUNW:1, IBM:0



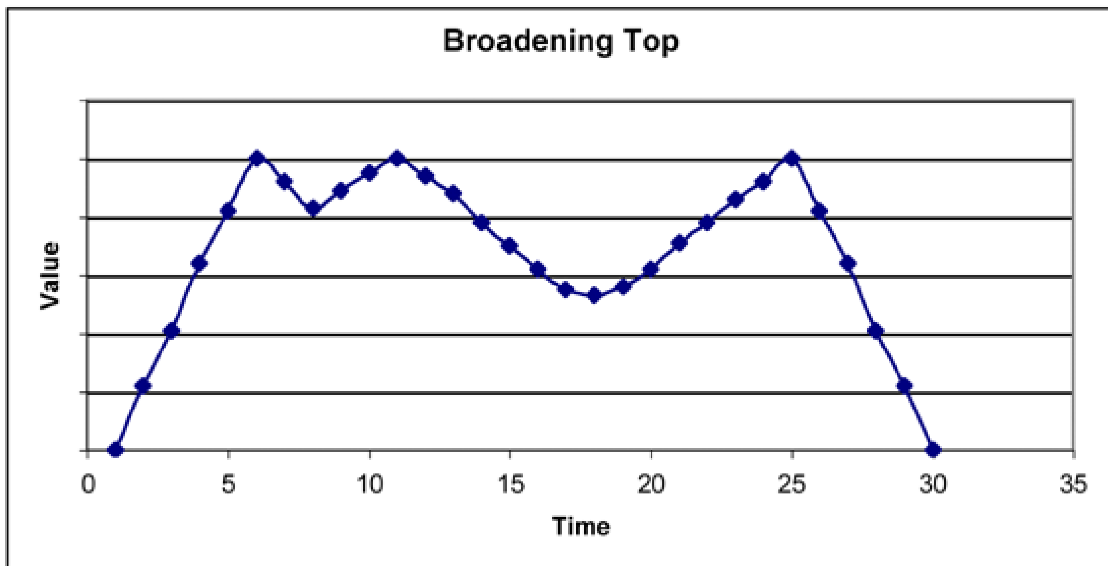


Figure 3.9: A template envelope found by our algorithm and the 'Broadening Top' stock value template

3.4 Run-time Experiments

The experiments below measure the effect on intersections performed and run-time as properties of the experimental dataset change. Our first experiment varies the epsilon value (both within the dataset and as a parameter to the algorithm), the second varies the length of the data sequences, and the third varies the total number of sequences. We chose these values to vary because they are the most likely to affect total run time and the most commonly varied properties.

Except for the properties that vary in each experiment, the experiments use the following properties:

- $\delta = 0$
- $\epsilon = 5$
- sequence length = 40
- number of sequences = 40
- value range = 0..99

Sequences were generated in the same manner as the artificial experiments in Section 3.1.

The experiments were run on a machine with dual AMD Opteron 246 CPU's, 6 GB of RAM, 4 GB of swap, an 80 GB UDMA166 hard drive, and running SuSE Linux Enterprise Server 8.0 for x86-64.

Each experiment measures the intersections and time used during both the initial pairing of envelopes created from the input sequences (see Section 2.3.3) and during the further intersections performed among those pairing results (see Section 2.3.3. The results for the initial pairing are labelled "Base Envelope Pairing" and the

results for the further intersections are labelled "Further Combination" in the graphs below. The graph areas are additive, thus the top line indicates total intersections or time. All three experiments use the same base datasets, thus while individual properties may alter the data the underlying patterns themselves do not change and thus will not affect the results.

3.4.1 Varying Epsilon

The variation of ϵ is performed as in Section 3.1 above. In this experiment, ϵ varies from 5 to 10 (relative to values between 0 and 99) and each of the 6 ϵ varying datasets are created with the same relative variance. Thus if a value is modified upwards by 5 in the $\epsilon = 5$ dataset the same value will be modified upwards by 10 in the $\epsilon = 10$ version. This prevents the random generation of noise from providing different sets for each ϵ value (except for the expected difference in ϵ).

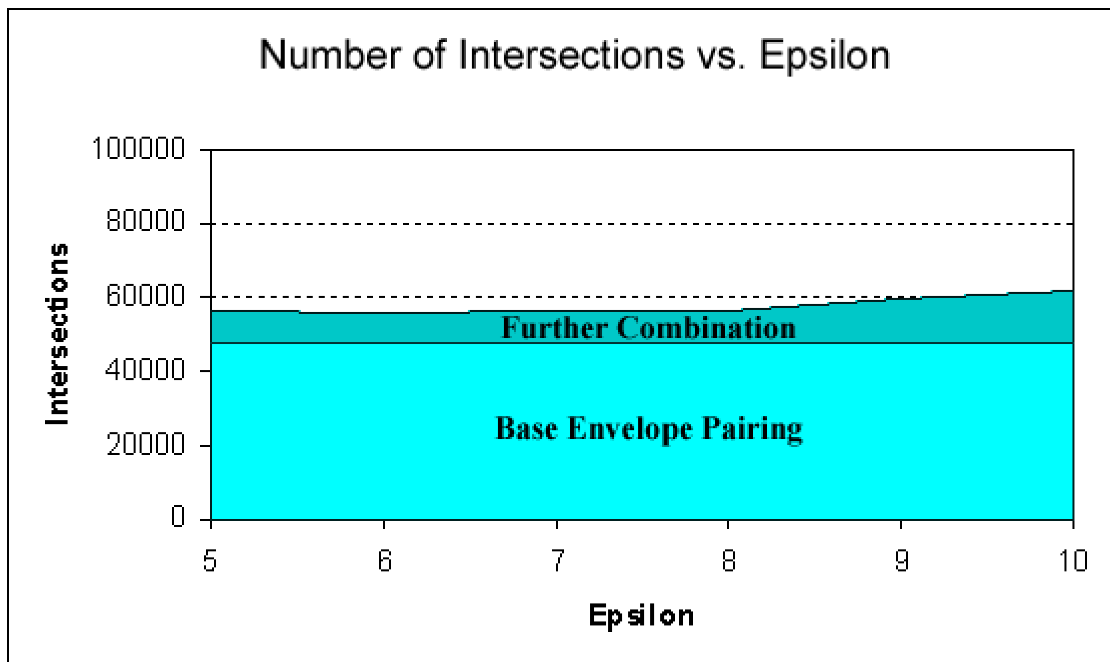


Figure 3.10: Graph of Intersections Performed as Epsilon Varies

Figure 3.10 shows the number of intersections performed as the datasets' ϵ-value increases. Since intersections of all pairings and offsets must be tried regardless of their ϵ-value, varying ϵ has no effect on the number of initial combinations. On the other hand, since a larger ϵ increases the likelihood of further combinations, the total number of intersections increases with ϵ.

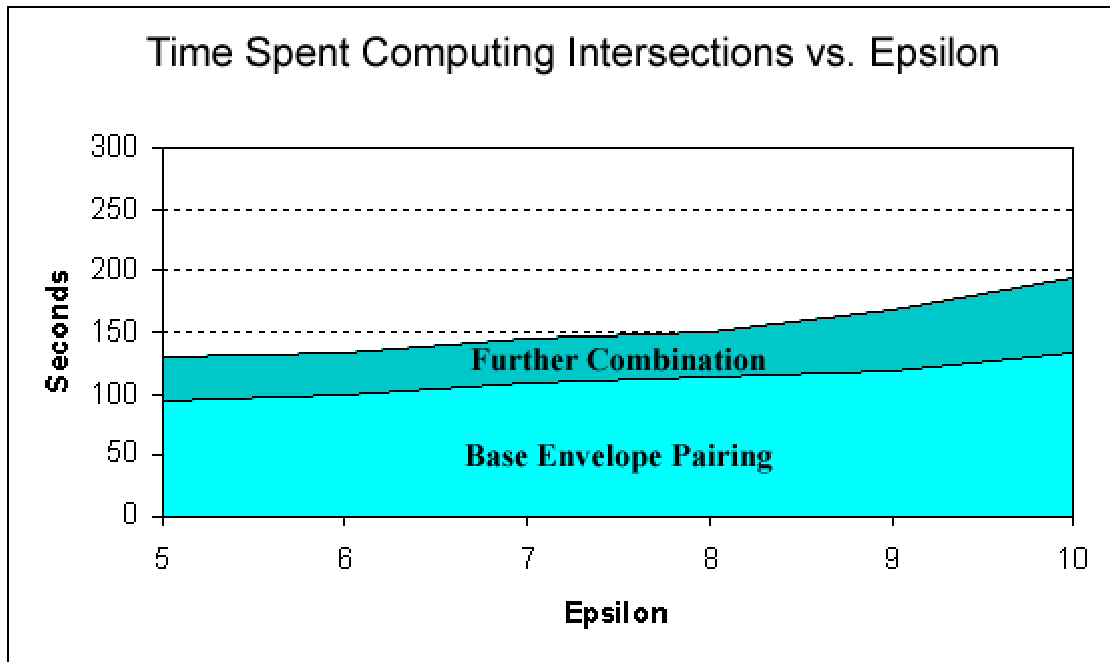


Figure 3.11: Graph of Total Time Spent as Epsilon Varies

While intersections and time are closely related values, this experiment shows that their relation is not perfectly linear. Even though Fig. 3.10 shows a perfectly constant number of initial intersections, 3.11 shows that the time required for those intersections increased with ϵ. This result comes form the intersection algorithm. A larger ϵ creates larger intersections between sequences. Since the tree that is created when intersecting envelopes grows larger with these larger intersections this leads to more time spent as ϵ grows.

3.4.2 Varying Sequence Length

Our next experiment measured the intersections performed and time spent as sequence length varied from 40 to 60. To create the varied datasets for these tests each of the sequences was extended by adding random values to the end of the existing sequences. Since these values are random, the number of template envelopes found by the algorithm is not expected to vary (as the number of template sequences does not increase).

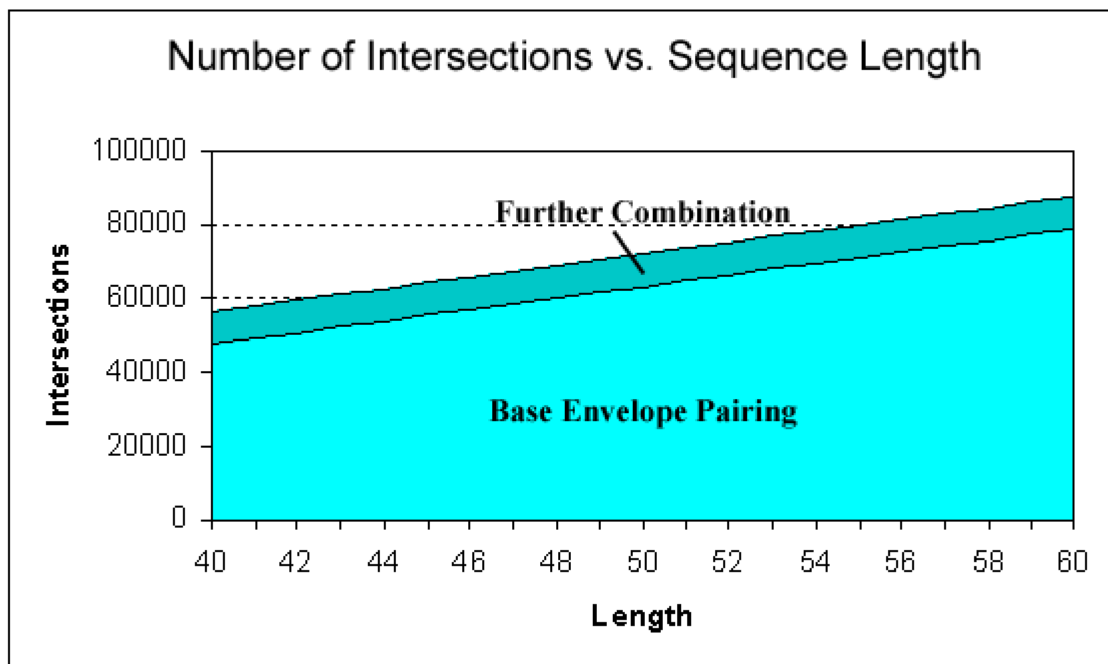


Figure 3.12: Graph of Intersections Performed as Sequence Length Varies

Figure 3.12 shows the number of intersections performed as the length increases. The number of intersections performed during the pairing of the initial envelopes increases due to the longer length of sequences, but the number of intersections needed to refine those pairings into template envelopes stays constant. This is because the additional initial pairings are unlikely to create envelopes of valid size and thus are discarded before the further refinements take place. Even should the

random values create a valid envelope in the initial pairings, it would still be unlikely to match other and thus would be filtered out after the first level.

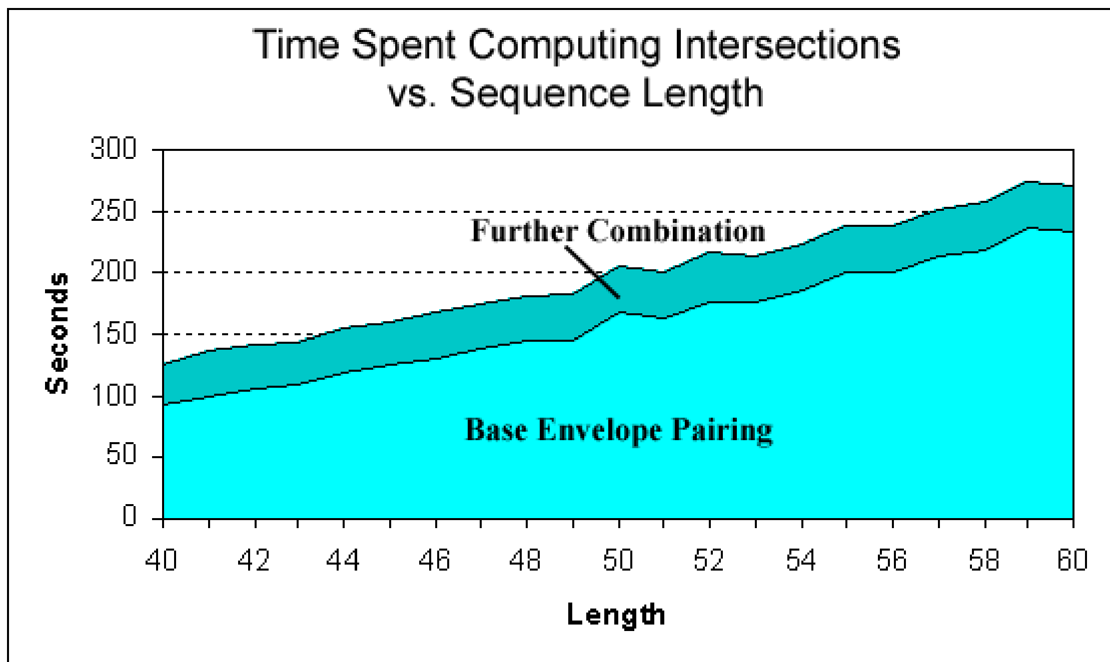


Figure 3.13: Graph of Total Time Spent as Sequence Length Varies

Figure 3.13 shows the time spent as the length of the sequences increases. While the longer sequences cause more total intersections these intersections do not create larger or longer envelopes. Thus, unlike in the first experiment, do not require more time per intersections. Thus the two graphs are roughly linearly proportionate.

3.4.3 Varying Number of Sequences

We next examined the effect on number of intersections performed and processing time as the number of sequences in a dataset varies. Varying the number of sequences is a simple matter of including of discarding complete instances from the dataset. Our experiments run from 20 sequences up to 40 sequences.

Figure 3.14 shows the number of intersections vs. the total number of sequences

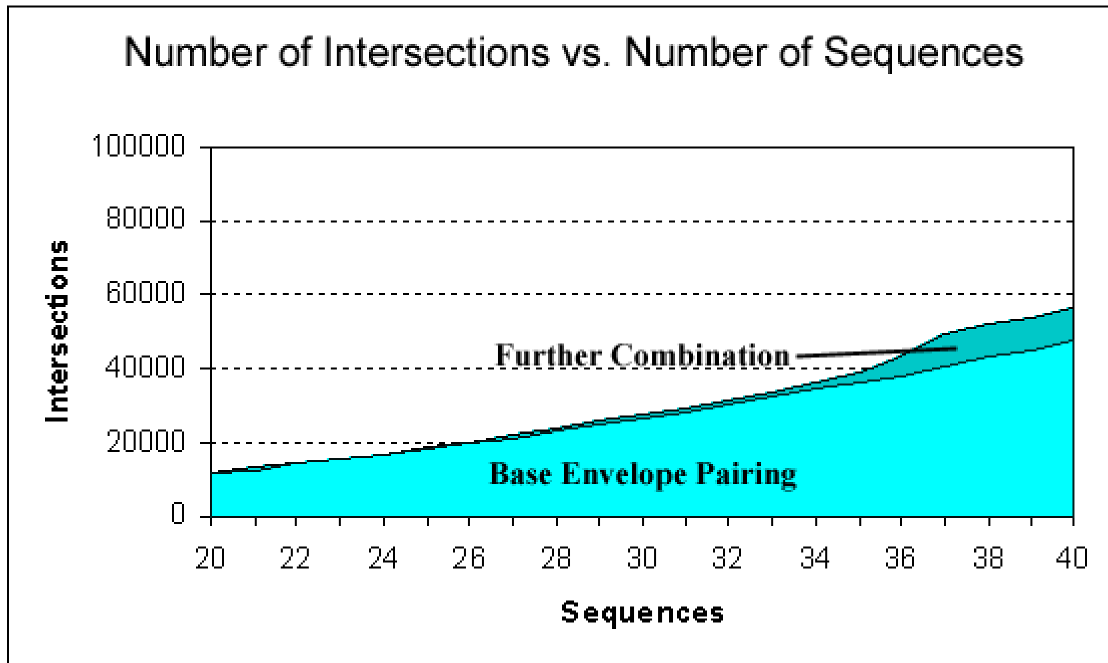


Figure 3.14: Graph of Intersections Performed as Number of Sequences Varies

in the dataset. The somewhat strange shape of the further combinations section is due to a simple quirk of the data. Since the datasets were created by dropping sequences from the basic 40 sequence dataset used in the other experiments the flattening that occurs beyond the 37 sequence dataset is actually due to sequences 38 through 40 not containing any expressions of the template sequences. Additionally, due to the nature of the algorithm, every new sequence containing a template must be matched to every other sequence also containing that template and thus each additional sequence expressing that template increases the number of intersections by a greater amount than the previous one. Thus, a roughly exponential increase can be seen up until 38 sequences after which the further combinations levels out and the increase is simply linear.

Figure 3.15 shows the corresponding time graph for varying sequences. There are no surprises in this graph as it is roughly proportional to the intersection graph.

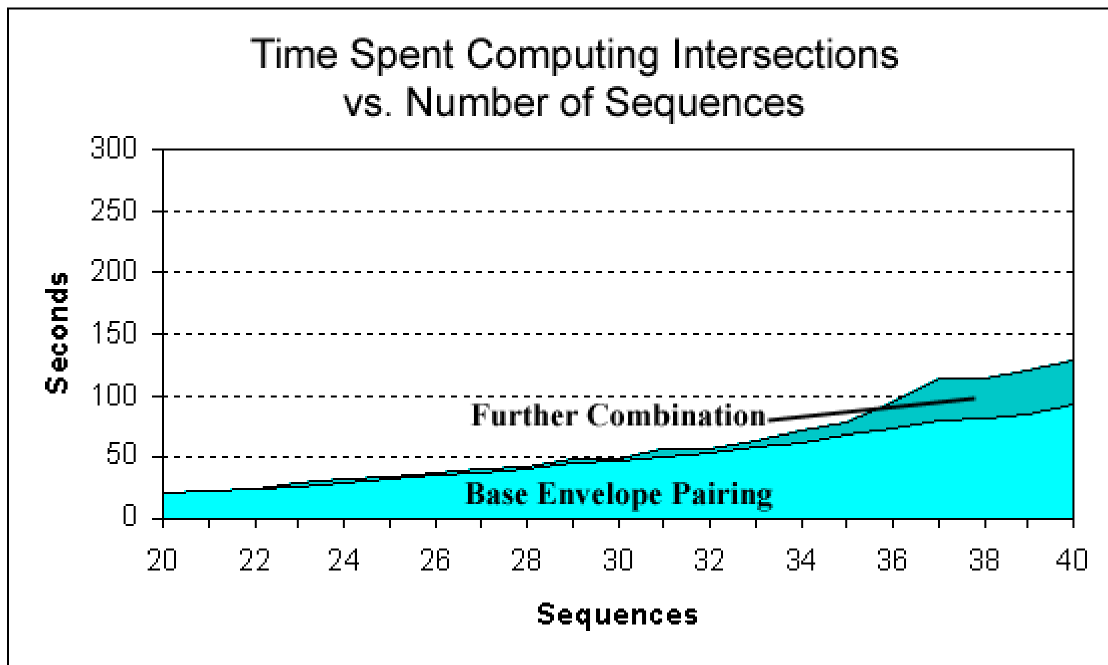


Figure 3.15: Graph of Total Time Spent as Number of Sequences Varies

3.4.4 Time Spent per Intersection

Finally, figure 3.16 shows a composite graph of time vs. intersections points for each of the experiments above. Each point represents a time vs. intersections combination for one of our three experiments above. While the lines for the three experiments seem to connect at approximately 56,000 intersections by 128 seconds this is not an indication of connection between results but rather just the point where each experiment used the same input values.

The nearly linear nature of the results from experiments on sequence length and number of sequences shows that our intersection algorithm reacts well to changes in those values. While the number of intersections increases the time spent per intersection increases very little. ϵ, on the other hand, creates larger intersection trees and thus increases the time spent per intersection at a much faster rate.

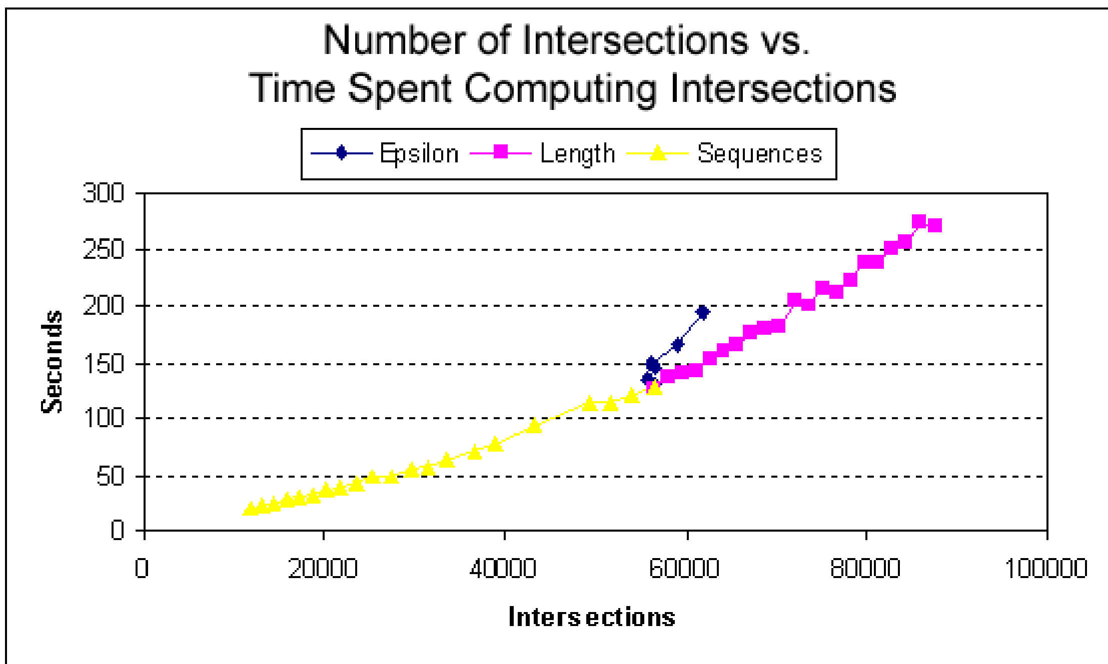


Figure 3.16: Graph of Total Time Spent Relative to Intersections Perfromed

Chapter 4

Conclusion

In this thesis we created an algorithm for use in finding patterns shared between instances in a set of time series data. To do this we used the concept of bounding envelopes to cast an area of probability around each time series. We then use a variation of the Apriori algorithm to build sets of envelopes that are intersected together to create template envelopes. Intersection is performed using a specially developed algorithm that uses a tree structure to organize the piecewise area functions that result from the intersection of each block of the envelopes. The template envelopes created from the intersection represent a pattern shared among all of the envelopes in the set used to create it. Finally, these template envelopes are ranked according to a user specified value function and returned as the results of the algorithm.

4.1 Contributions of this Work

- **We have created an algorithm to find common patterns within a set of time series data**

  - Our algorithm finds all combinations of envelopes at all offsets of sufficient length - no combinations are missed.
  - Our algorithm does so in an efficient manner that minimizes unneeded searching.
  - Our algorithm is flexible enough to work with many different datasets without requiring extensive knowledge of the user.

- **We developed a special tree structure to find the best vertical shift for intersecting two envelopes**

  The tree has a number of valuable properties:

  - The tree structure allows us to find only contiguous intersections.
  - Ranges get progressively more precise such that searching out nodes for a point or area is very efficient.
  - Information for previous intersections that are shared among multiple nodes is stored in common ancestor nodes so that space is conserved.
  - Total area always increases as nodes are added.
  - Length the length of an intersection is equal to the depth of its last node.
  - Children of a node cover one continuous range such that only two end-point nodes exist.

- **Our algorithm provides a valuable contribution to data mining research at Worcester Polytechnic Institute (see Section 4.1.1)**

  - As a preprocessing tool for further data mining.
  - As a tool to visualize common patterns within time series data.

4.1.1 Contributions to Data Mining at the Worcester Polytechnic Institute

For the past two years students at Worcester Polytechnic Institute have been working to improve the association rule mining capabilities of an open source data mining project called WEKA [WF99]. WEKA is an open source data mining system created at the University of Waikato in New Zealand. WEKA provides support for many different data mining techniques but initially focussed mostly on classification methods.

Students at WPI have been working to expand WEKA's association rule mining methods. Initially support for mining over data containing sets of items was added [SS02]. More recently this work has been greatly extended to mine for association rules over data containing sequences of temporal events [Prass]. Pray's work assumes that domain-dependent events of interest have been predefined.

Our work will provide a valuable preprocessing tool to use in the mining of time series association rules by finding more complex and data dependent events. By discovering data dependent events we expect to allow for more in-depth and meaningful sequence mining and to make the WPI WEKA association rule mining approach suitable for a wider range of application domains.

Additionally the finding of events is useful in and of itself. Our algorithm will identify common patterns from any time series data. Whether those patterns will then be used to simplify the series to a more manageable form or used more directly depends on the goals of the user.

4.1.2 Future Work

There are some pieces of our algorithm that could be expanded and improved in future projects.

- Upgrade the intersection algorithm so that it supports simultaneous intersection of multiple sequences rather than only two at a time. This would make the algorithm as a whole complete as it would avoid the small greedy aspect caused by only being able to intersect two envelopes at a time.
- Create a set of common value functions so that a novice user could simply select one rather than requiring them to create their own. Further evaluation of the properties of "good" results would also be valuable.
- Perform further optimization on the algorithm to reduce overall run time.

Appendix A

Unemployment Data

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
M1990	4.2	4.5	4.6	4.4	4.5	5	5.3	5.3	5.4	5.1	5.3	5.5
M1991	6.7	7.1	7.4	7.3	7.4	7.8	8	7.7	7.7	7.2	6.9	6.7
M1992	7.5	7.4	7.6	7.3	7.3	7.8	7.6	7.4	7.5	6.8	6.6	6.2
M1993	6.6	6.3	6.2	5.9	5.7	6	5.8	5.6	5.8	5.2	5	4.7
M1994	5.5	5.2	5.3	4.8	4.7	5.1	5.1	4.9	5	4.8	4.5	4.3
M1995	5	4.8	4.7	4.3	4.2	4.6	4.5	4.3	4.4	3.9	3.6	3.4
M1996	4	3.7	3.6	3.2	3.1	3.4	3.3	3	3.3	2.9	2.8	2.7
M1997	3.4	3.2	3.2	2.9	2.9	3.1	3.1	3.1	3.2	2.7	2.6	2.4
M1998	3	2.7	2.7	2.2	2.4	2.6	2.5	2.3	2.7	2.3	2.1	2.1
M1999	2.8	2.5	2.6	2.3	2.4	2.7	2.7	2.5	2.7	2.4	2.2	2
M2000	2.5	2.3	2.1	1.8	1.7	2.1	2.1	1.8	2	1.7	1.6	1.5
M2001	2.2	2.2	2.4	2.3	2.6	3.2	3.5	3.5	3.9	3.9	4	3.9
M2002	4.8	4.7	4.8	4.6	4.5	5.1	5.2	5	5.4	5	4.7	4.6
M2003	5	4.8	5.2	4.6	4.8	5.2	5	5	5.2	4.7	4.4	4.2
W1990	6	6.2	6.4	5.9	5.9	6.7	7.2	6.9	7	6.9	7.5	7.7
W1991	9.7	10.3	10.7	10	9.9	10.3	11.2	10.2	10	9.5	9.1	8.9
W1992	10.1	9.8	9.9	9.2	8.8	9.1	9.3	8.7	8.5	7.8	7.7	7.5
W1993	8	7.6	7.6	6.9	6.7	7	6.9	6.5	6.6	6.1	5.9	5.6
W1994	6.9	6.6	6.5	5.6	5.3	5.6	5.7	5.3	5.1	4.9	4.8	4.7
W1995	6	5.9	5.7	5.2	5	5.6	5.8	5.2	5.2	4.7	4.7	4.4
W1996	5.5	5.1	5	4.4	4.2	4.5	4.4	4	4.1	3.5	3.5	3.5
W1997	4.6	4.4	4.5	4	3.8	4.1	4.3	4	4	3.4	3.4	3.3
W1998	4.2	3.9	3.9	3	3.2	3.5	3.6	3.2	3.5	3.1	2.9	2.9
W1999	4	3.6	3.7	3	3.1	3.5	3.7	3.3	3.4	3.1	3	2.9
W2000	3.6	3.4	3.1	2.6	2.6	3	3.2	2.7	2.8	2.5	2.5	2.3
W2001	3.4	3.4	3.8	3.3	3.5	4.1	4.6	4.5	4.7	4.5	4.7	4.7
W2002	6.1	6.1	6.2	5.8	5.6	6.3	6.7	6.4	6.5	6	5.9	5.9
W2003	6.7	6.7	7.1	6.2	6.3	6.8	6.8	6.7	6.7	6.2	6	6.1

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
B1990	10.9	10.5	10.6	8.5	7.1	6.6	6.5	6.6	7.3	8.4	10.6	11.9
B1991	14.7	15.1	15.1	12.6	10.6	9.6	9.1	8.6	9.2	9	10.9	11.3
B1992	14.2	14.2	13.7	11.3	9.4	8.1	7.5	7.1	8.4	8.5	10.9	11.2
B1993	12.4	12.2	11.7	9.3	8.1	7.3	6.3	5.9	6.4	6.6	8.1	9.6
B1994	12.2	11.8	11.3	8.7	7.4	6.4	5.4	5.2	5.6	6.1	7.1	7.7
B1995	9.8	9	8.2	6.4	5.3	4.6	4.5	4.3	4.4	4.8	6.4	6.4
B1996	8.7	7.9	7.2	5.5	4.5	4.3	3.9	3.3	3.9	3.8	5.4	5.7
B1997	7.9	7.5	7.1	5.5	4.5	4.1	3.6	3.4	3.6	3.3	4.9	5.1
B1998	7.1	6.1	5.8	4	3.7	3.3	2.8	2.6	3	2.5	3.5	4.3
B1999	6.3	5.2	4.8	3.3	3.1	2.9	2.6	2.4	2.6	2.5	3.3	3.7
B2000	5.4	4.7	4.1	2.9	2.5	2.5	2.1	1.9	2.2	2.2	3.1	3
B2001	5	4.6	4.5	3.3	2.8	2.5	2.6	2.4	2.5	2.6	3.6	4
B2002	5.9	5.5	5.4	4.2	3.5	3.5	3.5	3.1	3.2	3.2	4	4.4
B2003	5.8	5.5	5.8	4.4	4	3.7	3.3	3.5	3.7	3.5	4.3	4.8

Appendix B

Monthly Stock Data

INTC	31.96	33.45	32.84	27.43	28.49	24.79	20.72	20.73	18.3
	16.19	17.16	15.56	15.47	20.74	17.17	13.79	16.54	18.63
	18.11	27.38	28.34	30.12	28.28	34.69	31.13	32.33	24.16
	20.22	27.66	29.47	28.92	26.7	30.54	25.99	28.22	36.53
	29.68	37.58	44.41	41.02	73.9	65.86	65.95	61.51	62.53
	65.06	55.72	48.76	40.56	37.79	38.13	36.59	40.47	33.95
	29.27	26.6	30.08	29.21	29.48	34.58	29.09	26.4	21.84
	21	17.43	20.65	18.13	17.47	19.76	19.06	21.9	19.78
MSFT	27.37	25.71	26.14	27.64	26.37	26.26	25.5	24.47	25.43
	24.07	23.57	23.52	25.62	28.59	26.5	21.68	24.32	23.78
	27.11	25.23	25.9	29.89	28.91	31.58	32.83	31.82	28.82
	25.36	28.27	32.8	36.18	34.29	33.58	27.11	29.24	30.26
	21.5	28.44	34.13	29.89	34.6	34.6	39.65	31.01	34.57
	52.66	44.29	48.51	57.86	45.13	45.87	44.88	45.87	42.53
	44.7	39.99	40.3	44.42	37.2	43.37	34.37	30.23	26.24
	27.27	23.77	27.24	26.86	21.02	22.33	22.18	21	18.49
SUNW	4.47	4.26	3.95	3.31	3.9	3.76	4.65	4.35	3.31
	3.26	3.44	3.09	3.11	4.29	2.96	2.59	3.69	3.92
	5.01	6.89	8.18	8.82	8.51	10.76	12.3	14.24	10.15
	8.27	11.45	16.29	15.72	16.47	17.12	15.37	19.87	30.56
	27.87	38.03	55.44	58.38	63.47	52.72	45.47	38.31	45.97
	46.85	47.62	39.28	38.72	33.06	26.45	23.25	19.88	16.97
	17.22	14.94	14.95	15.63	12.16	13.97	10.7	9.26	7.28
	6.23	4.95	5.91	5.43	5.01	5.15	5.22	5.95	5.99

IBM	92.34	90.21	88.99	87.85	81.56	80.65	81.89	87.39	84.11
	77.7	77.23	77.33	76.63	85.95	77.91	57.55	74.4	69.33
	70.91	79.23	82.33	102.22	96.44	105.91	118.74	113.46	105.95
	89.92	97.99	103.01	111.13	109.46	112.6	94.06	97.69	109.4
	83.03	91.33	96.09	109.87	128.79	109.39	106.76	104.57	108.52
	115.21	100.01	109.14	104.88	100.2	95.4	117.49	120.95	121.92
	125.38	112.52	101.23	85.78	82.15	88.44	88.99	79.7	71.46
	61.84	54.19	63.54	55.06	56.35	55.36	49.63	49.9	46.99

Bibliography

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[BC96] D.J. Berndt and J. Clifford. Finding patterns in time series: A dynamic programming approach. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (eds.), editors, *Advances in Knowledge Discovery and Data Mining*, pages 229–248. AAAI Press / MIT Press, 1996.

[CLR89] T.H. Cormen, C.E Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[GS99] Valery Guralnik and Jaideep Srivastava. Event detection from time series data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 33–42. ACM Press, 1999.

[HDY99] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Fifteenth International Conference on Data Engineering*, pages 106–115, Sydney, Australia, 1999. IEEE Computer Society.

[Kue94] Robert O. Kuehl. *Statistical Principles of Research Design and Analysis*. Duxbury Press, Belmont, CA, second edition, 1994.

[LR91] Jeffrey Little and Lucien Rhodes. *Understanding Wall Street*. Liberty Hall Press and McGraw-Hill Trade, 3rd edition, 1991.

[Prass] K. Pray. Mining association rules from time sequence attributes. Master's thesis, Department of Computer Science, Worcester Polytechnic Institute, In progress.

[SS02] Zachary Stoecker-Sylvia. Merging the association rule mining modules of the weka and arminer data mining systems. Undergraduate Graduation Project (MQP). Worcester Polytechnic Institute, April 2002.

[SSC02] Cosma Rohilla Shalizi, Kristina Lisa Shalizi, and James P. Crutchfield. Pattern discovery in time series, part i: Theory, algorithm, analysis, and convergence. 2002.

[VHGK03] Michail Vlachos, Marios Hadjieleftheriou, Dimitrios Gunopulos, and Eamonn Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 216–225. ACM Press, 2003.

[WCM+94] J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 115–125. ACM Press, 1994.

[WF99] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann Publishers, October 1999.