Chapter 5
Pattern Detection and Analysis in Financial Time Series Using Suffix Arrays

Konstantinos F. Xylogiannopoulos, Panagiotis Karampelas, and Reda Alhajj

Abstract The current chapter focuses on data-mining techniques in exploring time series of financial data and more specifically of foreign exchange currency rates' fluctuations. The data-mining techniques used attempt to analyze time series and extract, if possible, valuable information about pattern periodicity that might be hidden behind huge amount of unformatted and vague information. Such information is of great importance because it might be used to interpret correlations among different events regarding markets or even to forecast future behavior. In the present chapter a new methodology has been introduced to take advantage of suffix arrays in data mining instead of the commonly used data structure suffix trees. Although suffix arrays require high-storage capacity, in the proposed algorithm they can be constructed in linear time $O(n)$ or $O(n\log n)$ using an external database management system which allows better and faster results during analysis process. The proposed methodology is also extended to detect repeated patterns in time series with time complexity of $O(n\log n)$. This along with the capability of external storage creates a critical advantage for an overall efficient data-mining analysis regarding construction of time series data structure and periodicity detection.

5.1 Introduction

The current chapter proposes a method for detecting and analyzing patterns in financial time series using a novel approach. It utilizes the data structure suffix array which is constructed by the time series and is stored in an external database

K.F. Xylogiannopoulos • P. Karampelas (✉)
Hellenic American University, Manchester, NH, USA
e-mail: kostasfx@yahoo.gr; pkarampelas@hauniv.us

R. Alhajj
University of Calgary, Calgary, AB, Canada
e-mail: alhajj@ucalgary.ca

management system. The method searches the suffix array to identify all repeated patterns and finally analyzes the outcome of the search to detect all patterns with specific periodicity. The proposed methodology has been tested by analyzing real financial data for the US Dollar Euro currency exchange rate's values from December 2001 to December 2011 and it has produced interesting results showing that there is scarce periodicity, even in the foreign currency exchange market which is a chaotic system.

A time series is a set of data values representing a variable over a specific time period. The variable can be of any kind such as weather conditions, traffic, banking transactions, stocks or index prices, earthquakes or other geological events, etc. In each distinct fragment of the specific time period examined one value of the variable is assigned. Time period fragmentation can range from nanoseconds in case nuclear phenomena are observed to days for stock markets or even millions of years for geological events. Time series tend to become very important tools in data mining because they can help in discovering periodicities in the events they represent. Periodicities on the other hand can play an important role in decision making, e.g., in profit maximization in financial markets, in cost minimization in operations management in organizations such as banks or supermarkets, in quality of life improvement such as the traffic in large cities, etc.

The first step in the time series processing is to discretize it by assigning a letter of a predefined alphabet to each value or range [3, 16]. Such a process is important before the analysis because:

(a) Ranges of values are more influential than discrete ones since they represent a wide range of values
(b) Ranges can absorb and eliminate noise and errors of the data collection process
(c) Trying to analyze absolute values instead of ranges might be difficult and produce inconsistent results

For example, let $T = e_0, e_1, e_2, \ldots, e_{n-1}$ be a time series of n events, where e_i is the event occurred at time i. Time series T can be discretized by creating an alphabet of m characters to describe each value region. If T is defined from the stock market index daily changes, since there are two significant decimal digits for the change, it means that for a positive change between 0 and 5% there are 501 distinct values. In a time period of 5 years there are approximately 1,260 different values with each value having a probability of almost 0.024 to occur, which if analyzed might not lead to any result. Therefore, the time series can be discretized using a predefined alphabet, denoted by Σ, e.g., by using the following ranges:

- a for change between [0–0.99%]
- b for change between [1.00–1.99%]
- c for change between [2.00–2.99%]
- d for change between [3.00–3.99%]
- e for change between [4.00–5.00%]

Thus, the time series $T = 0.45\%, 2.12\%, 2.44\%, 1.67\%, 3.09\%, 0.87\%$ can be discretized into $T' = accbda$.

Following the discretization process, periodicity detection is applied in two steps by representing data values in memory or other storage means such as a hard disk and analyzing the data. For the representation of data, the most common approach so far is using suffix trees, which is a representation of all suffix strings in a tree data structure [16,22]. A suffix string is a substring of the original string which represents the time series from which a part of the beginning of the string has been removed. For example, let $\tau = \tau_0, \tau_1, \tau_2, \ldots, \tau_n$ be a string with n characters. A suffix string of τ can be $\tau' = \tau_m, \tau_{m+1}, \tau_{m+2}, \ldots, \tau_n$, where $0 \leq m \leq n$. A string of length n can have exactly $n-1$ suffix strings. An alternative approach to achieve the same results is by using, instead of suffix trees, suffix arrays, which is another powerful data structure. A suffix array is a sorted list of all the suffixes of a string [11]. After the creation of the suffix array the analysis of the time series can be done using already developed algorithms for periodicity detection [3, 4, 8, 16].

The rest of the chapter is organized as follows: Section 5.2 reviews the related work. In Sect. 5.3 the problem to be solved is defined. Section 5.4 describes the proposed method. Section 5.5 discusses the findings from applying the proposed method on historical financial data and finally, Sect. 5.6 presents the conclusions and anticipated future work.

5.2 Related Work

The suffix tree of a string is a tree including all the suffixes of the string [17] which is a very powerful data structure [2, 3, 15] heavily used for data-mining analysis because of its flexibility in string processing [6]. Many algorithms have been developed in the past decades to create suffix trees, like Weiner [23] and McCreight [12], with $O(n^2)$ and $O(n \log n)$ complexity, respectively. A suffix tree can also be created in linear time using Ukkonen's algorithm [22] with the assumptions that it can be stored in main memory [2, 6]. Then several other algorithms can be used to traverse suffix trees such as the non-recursive algorithm for binary search tree traversal [1]. In parallel, due to the size of the suffix trees especially in large time series, many techniques have been developed to store them on disks [2, 6]. Nevertheless, performance problems might occur even in the case of the $O(n)$ method of Ukkonen algorithm when traversing the tree especially if the part that is processed is not loaded entirely on memory. In any case, such methods are very useful for processing large time series and their strings [2] such as DNA analysis in bioinformatics and traffic control systems.

Many of the methods for the construction of suffix trees are very time consuming, especially if the time series to be analyzed is very long [2, 21]. Moreover, for very long time series, in which the whole structure has to be stored in a disk instead of memory, significant issues could occur. The constant disk access for writing and reading data can reduce the performance of the algorithms to a great extent [21]. However, the linear time construction and the lesser space consumption compared to other data structures have established suffix trees as the preferable data structure for many string matching analysis tasks [2,15,16,21]. Despite that, many researchers have shown that it is unfeasible to construct a suffix tree that exceeds the available main memory [2, 13, 14]. To overcome this problem some researchers, like Cheung et al. [2], have developed methods to combine on memory and disk data storage and access. Such techniques have a significant improvement of the performance of suffix trees and turn them into a powerful data structure.

Another data structure that has been developed relatively lately is suffix array [9, 17]. It is an array of all the suffixes of a string. The simpler way to construct a suffix array is by using a sorting method such as the merge-sort with complexity of $O(n \log n)$ [9, 11, 17]. Although a suffix array of a string can be constructed in linear time and then it can be lexicographically sorted, Ko and Aluru [10] have developed a method to construct directly the sorted suffix array in $O(n)$ time. The most important disadvantage compared to suffix trees, except construction time, is the storage space required [10, 11]. The elements of a suffix array are $n(n+1)/2$, based on the Gauss proof for the summation of the series $\sum_{k=1}^{n} k$, which is the summation of the elements of all the $n-1$ suffix strings including the original time series string of length n. On the other hand, the advantage of the suffix array is that it does not require memory storing, but instead it can be directly stored on a disk and accessed whenever needed. Although the amount of data to be stored is very large for long strings, since it has space capacity requirement $O(n^2)$, it is more efficient because by taking advantage of the structure of the array it is easier for storing and accessing a massive number of data on a disk instead of the main memory of a computer. Moreover, a database management system can be used for storage, sorting, and accessing and other data operations. Despite the fact that data operations on disk are usually processed slower than in memory, such methods can overcome the usual difficulty to address the problem of the limited memory size by converting it to a more manageable time delay problem due to disk storage as we will present.

For the computation of all the repeating substrings in a time series, methods and algorithms based either on suffix trees or suffix arrays can be used. There are methods that have linear time efficiency such as those described in [6, 20]. These methods can be implemented on suffix trees or suffix arrays and although they might be linear in time consumption and space capacity, the detection of repetitions can be significantly time and space consuming [16,20]. Franek and Smyth have developed an algorithm that can be used on suffix trees and a variation of it can be used on suffix arrays [5] for the computation of all the repeated patterns in a time series.

There are many algorithms that can be used for the analysis of the time series and the detection of any periodicities [3, 15, 16]. One of the earliest was developed by Elfeky et al. [3]. In their work the authors proposed two distinct algorithms for symbol and segment periodicity with complexity $O(n \log n)$ and $O(n^2)$, respectively. Their main difference is that the former does not work properly in the presence of insertion or deletion of noise while the latter does [16]. Many other algorithms have been developed lately using several techniques [4, 8] such as the algorithm of Han et al. [7] for partial periodicity and multiple period data mining in time series. Based on the work of Han et al. [7], Sheng et al. [18, 19] developed an algorithm to detect periodic patterns in a section of a time series [16]. Moreover, Huang and Chang [8] have also developed an algorithm for asynchronous periodic patterns.

Recently, a new approach [15, 16] has been developed which will be used in the current chapter for detecting the periodicity in time series. The specific methods have more efficient results by combining different techniques to report all types of periods even in the presence of noise. In their research, Rasheed et al. [16] have developed periodicity detection algorithms that are very efficient with average complexity $O(n^2)$. Moreover, their algorithms can work in the presence of insertion or deletion of noise and they can also be used for periodicity detection in a subsection of a time series.

5.3 Problem Definition

The data-mining analysis in time series can solve the problem of the detection of all the repeated patterns with a specific periodicity. This problem can be divided into two phases: (a) the detection of all the repeating patterns in the time series and (b) the filtering of the repeated patterns to discover which of them have specific periodicity.

To address these two challenges several techniques have been introduced in the literature. For the identification of all the repeating patterns, suffix trees have been used so far [2, 3, 15, 16] although some methods using suffix arrays have been introduced lately [9, 17]. Both cases require the construction of the data structure first and then the identification of all the existing patterns that have some kind of repetition in time series follows. Subsequent to the retrieval of all the positions of the repeating patterns in the time series, potential periodicities are detected. To detect periodicities, occurrence vectors need to be created first for storing the positions of all repeated patterns. Occurrence vectors are very important because they are required by periodicity detection algorithms [16] in order to analyze the time series and check if these patterns occur with a specific periodicity.

In the current work, suffix arrays will be used instead of suffix trees, for finding all the occurrences of repeated patterns and creating the occurrence vector in the time series. The proposed approach to solve the problem is to first construct the suffix array using a novel methodology and then detect all the repetitions of substrings in it. Subsequently the introduction of an algorithm will follow, which uses the suffix array to produce the occurrence vector of all the patterns.

5.4 Our Approach

The developed methodology is based on the following mathematical definitions and theorems that have been developed and proved for the scope of the chapter. First the definition and calculation of perfect periodicity is introduced. Such a calculation is important because perfect periodicity is needed to introduce confidence limits that will allow us later to detect periodicities that are highly confident and, therefore, valuable for data analysis.

5.4.1 Theorem for the Calculation of Perfect Periodicity of a Subset in a Time Series

Definition 5.1 (Perfect Periodicity). Let a time series $T = \{e_0 e_1 \dots e_{n-1}\}$ of $n \in N^*$ elements and length $|T| = n$ and a subset $S = \{e_i e_{i+1} \dots e_{i+k-1}\}$, of the time series T, of $k \in N^*$ elements and length $|S| = k$ that occurs in position i where $0 \leq i \leq i+k-1 \leq n-1$. We define as perfect periodicity PP of the subset S, with period $p \in N$, $p \geq 1$, the maximum number of repetitions that the S can have, with period p, in the time series T.

Theorem 5.1 (Calculation of perfect periodicity). *Let a time series $T = \{e_0 e_1 \dots e_{n-1}\}$ of $n \in N^*$ elements and length $|T| = n$ and a subset $S = \{e_i e_{i+1} \dots e_{i+k-1}\}$, of the time series T, of $k \in N^*$ elements and length $|S| = k$. The perfect periodicity PP of the subset S, with period $p \in N$, $p > 1$, can be derived from the formula*

$$PP = \left\lfloor \frac{|T| + (p - |S|)}{p} \right\rfloor.$$

Proof. From the definition of the perfect periodicity we have that PP is the maximum number of repetitions that a subset can have in a time series. Therefore we can write

$$|T| = p \times PP. \tag{5.1}$$

However, the formula is not complete because there are cases in which we can have a residual that we have to add to the multiplication to get the precise number of the length of the time series. Therefore, we can write

$$|T| = p \times PP + R, \tag{5.2}$$

where R is the residual.

From the definition of division between two natural numbers D and d, where D is the dividend and d the divisor, we have

$$D = dq + r. \tag{5.3}$$

What we can notice from comparing (5.2) and (5.3) is that they are analogous. If we change variables D, d, q, and r as follows:

$$D \equiv |T|, \quad d \equiv p, \quad q \equiv PP, \quad r \equiv R$$

then we will have as a result equation (5.2). From this result we can claim that perfect periodicity derives from the algorithm of division since the divisor has exactly the same definition as periodicity, while quotient is the same outcome as perfect periodicity. Indeed with the divisor d we divide the dividend D and get a quotient q, while if we divide $|T|$ with the period p (the divisor) we will get PP (the quotient), plus any remainder.

In the case of perfect periodicity it is important to note that perfect periodicity PP is larger than quotient q by one when the remainder of the division is not 0 (if $r \neq 0$, we do not have perfect division), so, we have to change the equation $PP = q$ to $PP = q + 1 \Rightarrow q = PP - 1$. That happens because while in the division between two natural numbers we get as a result the quotient that gives us how many times D is greater than d (plus a remainder, if any), in periodicity we have to count also the first occurrence of the subset S. Namely, we analyze subsets with a length of at least 1 and therefore the subset itself takes space inside the time series, which is not the case of division between natural numbers. For example, in the division 9/5, we will get as quotient 1 and remainder 4, while in the time series $T_1 = \{a * * * * a * * *\}$ with length 9 we can have two occurrences of subset $S = \{a\}$, in positions 0 and 5, with period 5 and a residual of three elements at the end of the time series after the last occurrence of the subset. So we have the same dividend and time series length ($D = |T| = 9$), divisor and period ($d = p = 5$) but we get as a result perfect periodicity $PP = 2$ instead of the quotient $q = 1$ and residual $R = 3$ instead of remainder $r = 4$. The perfect periodicity is greater than quotient by 1 as we expected to be. In the division 10/5 we will get as quotient 2 and remainder 0, while in the time series $T_2 = \{a * * * * a * * * *\}$ with length 10 we have again two occurrences of subset $S = \{a\}$, in positions 0 and 5, with period 5 and a residual of 4 elements. In this case we have perfect division and that is why perfect periodicity is equal to quotient. If we expand the time series T_2 by adding one more element and create a new time series $T_3 = \{a * * * * a * * * * a\}$ then we will have three occurrences, at positions 0, 5, and 10, with no residuals, while from the division 11/5 we will get as quotient 2 and remainder 1. Again the perfect periodicity will be greater than quotient by 1 as we have described previously.

Moreover, the above example implies that since the smallest length of the subset we can have is $k = |S| = 1$, the residual R can be $1 \leq R \leq p - 1$, where p is the period. Indeed, although the remainder of the division r is $0 \leq r \leq d - 1$, in the case of perfect periodicity, the residual R can only be $k \leq R \leq p - 1$, $k > 0$ in general or $1 \leq R \leq p - 1$, $k = 1$, which is the smallest value that k can take since $k = |S|$ represents the length of the subset, which cannot be 0. That is because if the residual becomes $R = k - 1$, smaller than k, then the last element of the last occurrence of the subset S will be outside of the time series' boundaries. In that case we will have a reduced perfect periodicity by 1 and the new residual will be $R = p - 1$. Therefore, we can claim that $R \in [k, p - 1]$, $0 < k < p$. For the calculation of perfect periodicity we have to choose the smallest possible R; therefore, we will choose the $R = \min\{k, p - 1\} = k = |S|$ which includes all cases we want to examine. If we choose the greatest possible R, $\max\{k, p - 1\} = p - 1$, we fall into the category of normal division with length of subset $|S| = 1$, which is not the general case since we want to cover all the cases with subsets' length greater than one. Furthermore, if $R = k$ then we have no residual, which is the optimum case we can have for perfect periodicity before it is downgraded to the next smaller integer.

According to the above-mentioned analysis, where PP should be changed to $PP - 1$ and R should be replaced with $|S|$, (5.2) can be transformed as follows:

$$|T| = p(PP-1) + |S| \Longrightarrow |T| = pPP - p + |S| \Longrightarrow PP = \frac{|T| + (p - |S|)}{p}. \quad (5.4)$$

Since we care about perfect periodicity, which is a natural number, we can transform (5.4), in order to get the integral part of the outcome, as follows:

$$PP = \left[\frac{|T| + (p - |S|)}{p}\right]$$

which is the biggest natural number smaller than PP, $[PP] \leq PP < [PP] + 1$. □

Furthermore, the decimal part that is truncated is R/p and represents the percentage of the remaining elements in the time series following the last occurrence of the substring. It is also showing how much more elements we need in time series to have one more occurrence of the subset since $1 - R/p$ represent the percentage of elements we need to have in a full period. So, the actual number of the elements to complete another period will be $(1 - R/p)p = p - R$.

Example 5.1. Let us take a simple example of calculating perfect periodicity that will demonstrate, why $k \leq R \leq p - 1$ and why we have to choose as R the smallest $R = \min\{k, p-1\} = k = |S|$. Let us assume that we have a time series T_1 as it is presented in Fig. 5.1 with $|T_1| = 20$ and the subset ab that starts at position 0 and is repeated with period $p = 5$.

In this case we have perfect periodicity $PP_1 = 4$:

$$PP_1 = \left[\frac{|T_1| + (p - |S|)}{p}\right] = \left[\frac{20 + (5-2)}{5}\right] = [4.6] = 4.$$

Suppose that we truncate the time series to have $|T_2| = 17$. Then we have again perfect periodicity $PP_2 = 4$:

$$PP_2 = \left[\frac{|T_2| + (p - |S|)}{p}\right] = \left[\frac{17 + (5-2)}{5}\right] = [4] = 4.$$

We have to remember that perfect periodicity is not just the quotient but it is by 1 greater since $PP = q + 1$. So, if we do the division and express the perfect periodicity as the quotient plus 1 then we will have the following analysis $17/5 = 5 \times 3 + 2 \Rightarrow PP = q + 1 = 3 + 1 = 4$, and, moreover, we have no other values after the subset

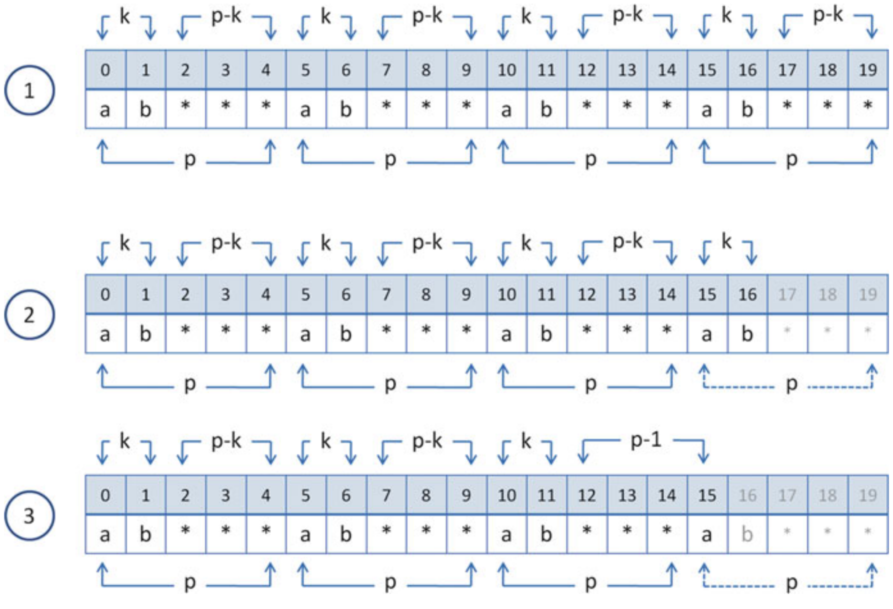


Fig. 5.1 Different time series for the calculation of perfect periodicity

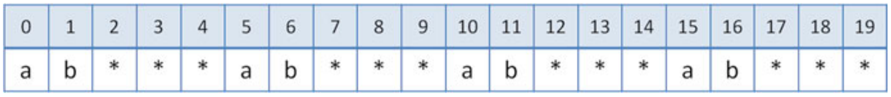
since the remainder of the division $17/5$ is $2 = k$, which is the length of the subset $k = |S|$ and as we have proved it is the smallest value the residual r could take since $k \leq R \leq p-1, k > 0$.

If we truncate the time series by one more element to have $|T_3| = 16$, then the perfect periodicity will be $PP_3 = 3$

$$PP_3 = \left[\frac{|T_2| + (p - |S|)}{p}\right] = \left[\frac{16 + (5-2)}{5}\right] = [3.8] = 3$$

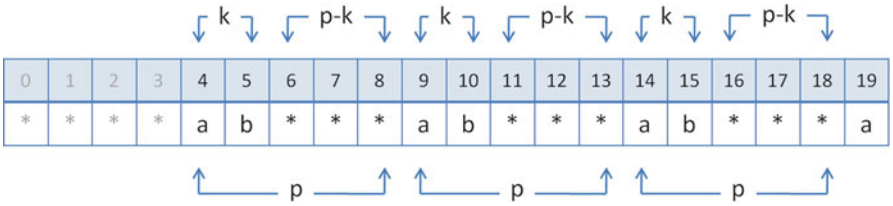
instead of 4 because one character of the subset will be outside of the boundaries of the time series T_3. Now we will have four remaining values (including a at position 15, which is not an occurrence anymore), which is actually the largest value the residual R could take since $k \leq R \leq p-1, k > 0$ and in this case $p-1 = 5-1 = 4$, while the remainder of the division $16/5$ is $1 < k$. Therefore, we conclude that the optimum case for the perfect periodicity, before we fall into smaller number, is when we have no remaining elements as in the second case in which $R = k = |S|$. □

Lemma 5.1 (Calculation of perfect periodicity starting at a position greater than 0). *Let a time series $T = \{e_0 e_1 \ldots e_{n-1}\}$ of $n \in N^*$ elements and length $|T| = n$ and a subset $S = \{e_i e_{i+1} \ldots e_{i+k-1}\}$ of the time series T of $k \in N^*$ elements and length $|S| = k$ where $0 \leq i < i+k-1 \leq n-1$. If we want to calculate perfect periodicity for the subset S from the position of first occurrence e_i, where*



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	b	*	*	*	a	b	*	*	*	a	b	*	*	*	a	b	*	*	*

Fig. 5.2 Original time series for calculation of perfect periodicity



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
*	*	*	*	a	b	*	*	*	a	b	*	*	*	a	b	*	*	*	a

Fig. 5.3 The new time series for calculation of perfect periodicity using Lemma 5.1

$i = StartingPosition$, then the perfect periodicity PP of the subset S, with period $p \in N$, $p \geq 1$, can be derived from the formula

$$PP = \left[\frac{(|T| - i) + (p - |S|)}{p}\right].$$

Proof. If we truncate from the time series T, i elements from the beginning then we have a new time series T_1 with length:

$$|T_1| = |T| - StartingPosition = |T| - i$$

which it will give as a result the formula of Lemma 5.1 for the perfect periodicity if we change the new value of $|T_1|$ with its equivalent in the formula of the theorem:

$$PP = \left[\frac{|T_1| + (p - |S|)}{p}\right] \Longrightarrow |T_1| = |T| - iPP = \left[\frac{(|T| - i) + (p - |S|)}{p}\right]$$

and we get the formula of Lemma 5.1. □

Example 5.2. Let us examine again the first case of Example 5.1 calculating perfect periodicity from a different starting element than e_0 at position 0. We have the time series as presented in Fig. 5.2 in which we have calculated that the perfect periodicity is $PP = 4$. However, assuming that we want to calculate the perfect periodicity starting from position $i = 4$ as it is illustrated in Fig. 5.3 we will have

$$PP = \left[\frac{(|T| - i) + (p - |S|)}{p}\right] = \left[\frac{(20 - 4) + (5 - 2)}{5}\right] = [3.8] = 3$$

and we get the expected result because starting from position 4 we have excluded the first occurrence of the subset and we have four free cells, one at the begging and three at the end, which are the most we can have $1+3=4=p-1$. It is like creating a new time series T_1 with $|T_1|=|T|-4=16$, in which the subset starts from position 4 of the first time series T (position 0 of the new time series T_1) with four remaining elements at the end, including a at the position 19 which is not an occurrence any more as it is represented in Fig. 5.3. □

Lemma 5.2 (Calculation of perfect periodicity starting at a position greater than 0 and ending at position less than n − 1). *Let a time series* $T=\{e_0e_1\dots e_{n-1}\}$ *of* $n\in N^*$ *elements and length* $|T|=n$ *and a subset* $S=\{e_ie_{i+1}\dots e_{i+k-1}\}$ *of the time series* T *of* $k\in N^*$ *elements and length* $|S|=k$ *where* $0\le i<i+k-1\le n-1$. *If we want to calculate the perfect periodicity for the subset* S *from the position of first occurrence* e_i, *where* $i=StartingPosition$, *till another position* e_m, *where* $m=EndingPosition$ *and moreover we have that* $0\le i<i+k-1<m\le n-1$, *then the perfect periodicity* PP *of the subset* S, *with period* $p\in N$, $p\ge 1$, *can be derived from the formula*

$$PP=\left\lceil\frac{(|T|-i-(|T|-(m+1)))+(p-|S|)}{p}\right\rceil \Longleftrightarrow PP=\left\lceil\frac{(m+1-i)+(p-|S|)}{p}\right\rceil.$$

Proof. If we truncate from the time series T i elements from the beginning and $(|T|-(m+1))$ elements from the end, a new time series T_1 is created with length

$$|T_1|=|T|-StartingPosition-(|T|-(EndingPosition+1))\Longleftrightarrow$$
$$|T_1|=|T|-i-(|T|-(m+1))=m+1-i$$

which it will give as a result the formula of Lemma 5.2 for the perfect periodicity if we change the new value of $|T_1|$ with its equivalent in the formula of the theorem:

$$PP=\left\lceil\frac{|T_1|+(p-|S|)}{p}\right\rceil \Longleftrightarrow |T_1|=m+1-iPP=\left\lceil\frac{(m+1-i)+(p-|S|)}{p}\right\rceil$$

and we get the formula of Lemma 5.2. □

In the second lemma we get the general formula that represents the perfect periodicity of a subset in a time series because if we set $m+1=|T|$ and $i=0$ we fall in the formula of the theorem, while with $m+1=|T|$ and $i\neq 0$ we fall in the formula of the first lemma in which we start from a position greater than 0.

Example 5.3. Let us use again Example 5.2, calculating perfect periodicity from element e_i at position $i=4$ but for a different ending element e_m at the position $m=14$. Then we will have as it is presented in Fig. 5.4 a new time series T_1 starting at position 4 and ending at position 14 of the original time series, in which perfect periodicity will be

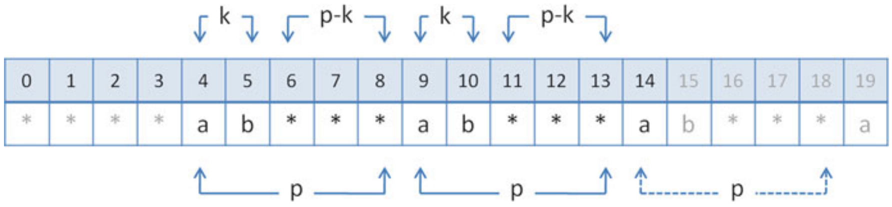


Fig. 5.4 Time series for calculation of perfect periodicity using Lemma 5.2

$$PP = \left\lfloor \frac{((|T| - i - ((|T| - 1) - m)) + (p - |S|)}{p} \right\rfloor \Longleftrightarrow$$

$$PP = \left\lfloor \frac{(m + 1 - i) + (p - |S|)}{p} \right\rfloor = \left\lfloor \frac{(14 + 1 - 4) + (5 - 2)}{5} \right\rfloor = \lfloor 2.8 \rfloor = 2$$

and we get the expected result because starting from position 4 and ending at position 14 of the original time series, we can have only two occurrences of the subset with four remaining elements at the end, including a at the position 14 of the original time series or position 10 of the new time series, which is not an occurrence any more. □

5.4.2 *Algorithms*

5.4.2.1 Suffix Array Construction

Many different approaches for the construction of the suffix array can be used. The most common is the use of a for-loop structure, which each time removes the first letter of the string. The new substring, which is a suffix string of the initial string of the time series, can be stored either on memory as an array (suffix array) or on the disk. Alternatively, many other algorithms can be used to construct the suffix array in linear time and lexicographically sorted [9, 17]. However, it is important to take into consideration memory storage limitations and constrains. When stored in memory, processing is significantly faster, but, for very long strings, the size of the array might be a drawback because it might exceed the size of the available memory or leave a small amount of free memory for algorithms' operation. In this work in order to calculate occurrence vectors faster and easier, a storage method that uses an external database management system has been selected.

The algorithm Suffix Array Construction (SAC) is used, in which a for-loop calculates each time the substring from the position i till the end of the string of the time series. Then the substring is inserted into database with the respective T-SQL insert command. With the selected algorithm we avoid the construction of a huge array on memory before inserting it into database.

Algorithm 1. Suffix Array Construction
Input: string X of time series
Output: an array of all suffix strings or nothing in case of direct insertion into database

```
1      SAC(string X)
2.1    for i := 0; i<X.length; i++
2.2       subString :=X.Substring(i, X.length - i)
2.3       insert substring into database
2.4    end for
3      end SAC
```

Fig. 5.5 The suffix array and the sorted suffix array of string *abcabbabb*

a		**b**	
0	a b c a b b a b b	6	a b b
1	b c a b b a b b	3	a b b a b b
2	c a b b a b b	0	a b c a b b a b b
3	a b b a b b	8	b
4	b b a b b	5	b a b b
5	b a b b	7	b b
6	a b b	4	b b a b b
7	b b	1	b c a b b a b b
8	b	2	c a b b a b b

Algorithm 1 has only one for-loop and, therefore, it has time complexity $O(n)$. Time complexity cannot be defined with accuracy because the time needed for the insertion of the string into the database might vary on different database management systems and on different software and hardware configurations. In general, the process can be counted as one instruction. In case memory storage is used the time complexity will be $\Theta(4n)$ or generally $O(n)$.

5.4.2.2 Repeated Patterns Detection

Let the time series be represented by the string *abcabbabb*. The length of the time series is $n = 9$. The alphabet of the specific sample is $Alphabet = \{a, b, c\}$ of length $m = 3$. The suffix array of the specific string is represented in Fig. 5.5a.

Irrespectively of the selected type of storage two different types of information for each row of the suffix array table are needed. The first is the position of the suffix string in the time series string and the second is the suffix string. Figure 5.5b represents the lexicographically sorted rows of the table by the suffix string column.

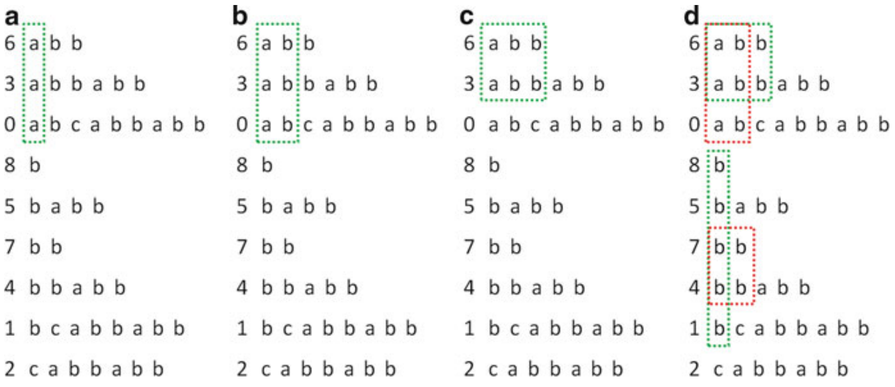


Fig. 5.6 The suffix strings of string *abcabbabb* using a sorted suffix array

In order to calculate the occurrence vectors that will be needed in the periodicity detection algorithms the following process has to be executed:

1. For all the letters of the alphabet count suffix strings that start with the specific letter.
2. If no suffix strings found or only one is found, proceed to the next letter (periodicity cannot be defined with just one occurrence).
3. In case the same number of substrings is found as the total number of the suffix strings, proceed to step 4 and the specific letter is not considered as occurrence because a longer hyper-string will occur.
4. If more than one string and less than the total number of the suffix strings is found, then for the letter used and counted already and for all letters of the alphabet add a letter at the end and construct a new hyper-string. Then do the following checks:

   (a) If none or one suffix string is found that starts with the new hyper-string consider the previous substring as an occurrence and proceed with the next letter of the alphabet.
   (b) If the same number of substrings is found as previously then proceed to step 4. However, the specific substring is not considered as occurrence because a longer hyper-string will occur.
   (c) If more than one and less than the number of occurrences of the previous substring is found, consider the previous substring as a new occurrence and continue the process from step 4.

In the case of the string *abcabbabb* the following process can be used: Starting with first letter of the alphabet, *a*, three substrings can be found that start with a as depicted in Fig. 5.6a. Since more than one and less than the total number of the substrings have been found starting with a, the process should continue to search deeper by adding each letter of the alphabet to a and construct each time a new string.

The first hyper-string is *aa*. Since there is no substring starting with *aa* the process should continue with the next letter and create a new hyper-string, *ab*. Counting the substrings that start with the new string *ab* the result is exactly as many as the previous string (with only one letter, *a*) as illustrated in Fig. 5.6b. In this case, the first string the process started, *a*, is definitely not an important occurrence, because the hyper-string *ab* has occurred exactly the same times in the time series. Since *ab* is longer than *a*, the process uses only the longer *ab*. Continuing to search deeper by adding again each one of the alphabet letters to the new string *ab*, the process starts with the letter *a* and founds no substrings starting with *aba*. It proceeds to the next letter *b*. With the new string *abb* two substrings can be found as presented in Fig. 5.6c. Since the number of the substrings is less than the previous *ab*, definitely *ab* is an occurrence. However, the process should check if *abb* is an occurrence too, or there is a longer string that starts with *abb* is an occurrence. The process continues and finds that there is no string that starts with *abb* and can be counted more than two times. Therefore, the process goes back to step 4 and checks for the string *ac*. Since there are no substrings starting with *ac*, the process has finished with all the substrings starting with *a*. So far the occurrences that are important are *ab* and *abb*. By continuing the process and moving back to the first step and proceeding with the next letter of the alphabet, *b*, the process will find the occurrences *b* and *bb*. For the letter *c* there are no occurrences. So, the whole process has been concluded and produced the findings depicted in Fig. 5.6d.

The whole process can be described by the algorithm Calculate Occurrences' Vectors (COV) "Algorithm 2". The execution of the algorithm should be done by passing an empty string and the length *n* of the time series: COV("", *n*).

In the algorithm there are two external calls: (a) the first one is "how many strings start with newX," which is a T-SQL statement that queries the database and returns the number of the strings that start with the specific substring and (b) the second "find positions of string *X*," which is again a T-SQL statement that gets the positions of the suffix strings in the time series. These positions are the numbers in front of each suffix string. In the specific example with the string *abcabbabb* the occurrences *ab*, *abb*, *b*, *bb* have been found and the equivalent occurrence vectors are: $ab(0,3,6)$, $abb(3,6)$, $b(1,4,5,7,8)$, and $bb(4,7)$.

In the case of memory storage, the appropriate algorithms for sorting and querying the suffix array should be produced to get the respective results, instead of using the two T-SQL statements.

The equivalent suffix tree for the specific sample string will be as it is presented in Fig. 5.7, where $ is the terminal symbol that is used for each suffix string. The relative occurrence vectors have also the substring positions in parenthesis.

5.4.2.3 Periodicity Detection Algorithms

In order to search and detect if there are any periodicity patterns in the time series, we use the periodicity detection algorithms described thoroughly in the

Algorithm 2. Calculate Occurrences' Vectors
Input: string of pattern we want to check, a counter of string length
Output: an array of all occurrence vectors

```
1         COV(string X, int count)
2         isXcalculated := false
3.1       for each letter l in alphabet
3.2          newX := X + l
3.3          newCount := how many strings start with newX
3.4.1        if newCount = count
3.4.2           COV(newX, newCount)
3.4.3        end if
3.5.1        if (newCount = 1) AND (isXcalculated = false)
                   AND (X NOT null)
3.5.2           find positions of string X
3.5.3           isXcalculated := true
3.5.4        end if
3.6.1        if newCount> 1 AND newCount< count
3.6.2.1         if (isXcalculated = false) AND (X NOTnull)
3.6.2.2            find positions of string X
3.6.2.3            isXcalculated := true
3.6.2.4         end if
3.6.3           COV(newX, newCount)
3.6.4        end if
3.7       end for
4         end COV
```

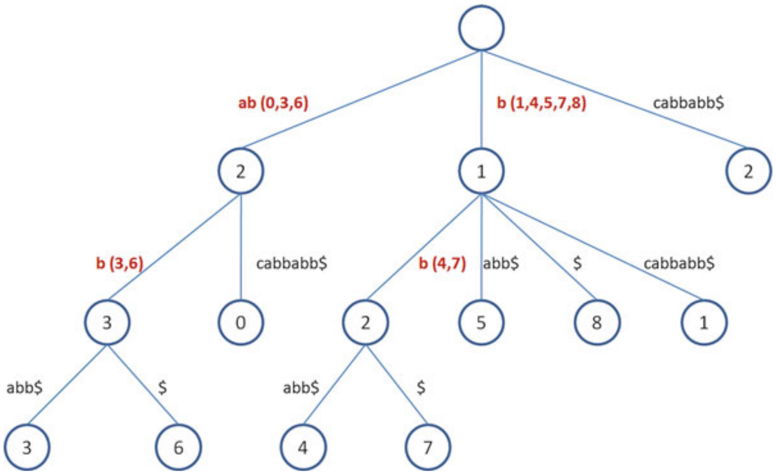


Fig. 5.7 The suffix tree of *abcabbabb* and its occurrences' vectors

research paper [16]. In those algorithms minor modifications have been done using the above-mentioned theorem to calculate confidence and some other minor improvements regarding variable initialization.

Algorithm 3. Periodicity Detection Algorithm
Input: the occurrence vector of time series repeated patterns
Output: positions of periodic patterns

```
1           PDA(string X)
2.1         for each occurrence vector occurVec of size
                    k for pattern X
2.2.1           for j = 0; j < k; j++
2.2.2               p = occurVec[j+1] - occurVec[j]
2.2.3               startPos = occurVec[j]
2.2.4               count = 0
2.2.5.1             for i = j; i< k; i++
2.2.5.2.1               if ((startPos mod p) == (occurVec[i] mod p))
2.2.5.2.2               count++
2.2.5.2.3               end if
2.2.5.3             end for
2.2.6               confidence(p)= count(p) / PP(p,startPos,X)
2.2.7.1             if (confidence(p) >= threshold)
2.2.7.2                 add p to the period list
2.2.7.3             end if
2.2.8           end for
2.3         end for
3           end PDA
```

Algorithm 3 searches all the positions for each repeated pattern Algorithm 2 has found to detect periodicity. First it creates the difference vector which is position $i+1$ minus position i of the repeated pattern and calculates the differences (periodicities) p for each pair. Then it checks if the modulo of the division between starting position and p is equal with the modulo of position i and p. If it is the same, it means that the specific position i is a repetition with the specific periodicity p and the algorithm increments the count of the specific periodicity. When finished it calculates the confidence which is the number that periodicity p has been found valid divided by the perfect periodicity.

Perfect periodicity can be calculated using Theorem 5.1. If confidence is equal or larger than the user's specified threshold then the periodicity is added to the list of all valid periodicities. The threshold is the percentage of the lower limit of the confidence. When the process is finished, Algorithm 3 continues to the next occurrence vector.

Algorithm 4 works very similar to Algorithm 3 but allows noise resilience in the time series. The main and very important difference is the introduction of time tolerance value. More specifically, the algorithm searches for periodicities not only in the specific positions that periodicity is expected but also within the limits of a time tolerance. The algorithm namely searches the elements before and after the actual period within a time tolerance value. The new variables that include represent (a) the distance between the current occurrence and the reference starting position

Algorithm 4. Noise Resilient Periodicity Detection Algorithm
Input: the occurrence vector of time series repeated patterns, time tolerance value tt
Output: positions of periodic patterns

```
1           NRPDA(string X, tt)
2.1         for each occurrence vector occurVec
                              of size k for pattern X
2.2.1          for j = 0; j < k; j++
2.2.2             p = occurVec[j+1] - occurVec[j]
2.2.3             startPos = occurVec[j]
2.2.4             currStartPos = StartPos
2.2.5             preOccur = -1
2.2.6             count = 0
2.2.7.1           for i = j; i< k; i++
2.2.7.2              A = occurVec[i] - currStartPos
2.2.7.3              B = Truncate(A/p)
2.2.7.4              C = A - (p*B)
2.2.7.5.1            if ((-tt<= C <= tt) AND
                     (Truncate((preOccur - currStartPos)*p) <> B))
2.2.7.5.2                 currStartPos = occurVec[i]
2.2.7.5.3                 preOccur = occurVec[i]
2.2.7.5.4                 count++
2.2.7.5.5            end if
2.2.7.6           end for
2.2.8             confidence(p)=count(p)/PP(p,startPos,X)
2.2.9.1           if (confidence(p) >= threshold)
2.2.9.2              add p to the period list
2.2.9.3           end if
2.2.10         end for
2.3         end for
3           end NRPDA
```

(variable A), (b) the number of periodic values that must be passed from the current reference starting position to reach the current occurrence (variable B), and (c) the distance between the current occurrence and the expected occurrence (variable C). The preOccur variable holds the value of the current occurrence. If the pattern that is under examination will be found in between the limits of the expected position and the time tolerance (minus or plus) and the current occurrence is not a repetition of an already counted periodic value then the algorithm increments the count that periodicity *p* has been found [16]. When finished it calculates the confidence which is the number that periodicity *p* has been found valid divided by the perfect periodicity. Perfect periodicity again can be calculated using Theorem 5.1. If confidence is equal or larger than the user's specified threshold then the periodicity is added to the list of valid periodicities. The threshold is the percentage of the lower limit of the confidence. When the process is finished, Algorithm 4 continues to the next occurrence vector.

Table 5.1 Suffix construction space capacity and time complexity results (real-case scenarios A)

Case	1	2	3	4	5	6	7
Alphabet size (m)	9	9	9	9	9	9	9
String length (n)	100	200	400	800	1,600	3,200	6,400
S.A. construction instructions	698	1,398	2,798	5,598	11,198	22,398	44,798
S.A. space	5,150	20,300	80,600	321,200	1,282,400	5,124,800	20,489,600

5.4.3 *Algorithm Analysis*

For the data analysis in the current chapter a typical personal computer has been used. The main disadvantage of the suffix array is the storage allocation on memory or on disk. The size of the required storage space is at least $n(n+1)/2$ or $O(n^2)$. For a string of length 100,000 elements the approximate needed disk space is 12 GB in order for the database management system to create the appropriate database file. The need for approximately 2.5 times more space than what the formula $\sum_{k=1}^{n} k = n(n+1)/2$ computes is because of the metadata and other information that the database management system stores in the file. Furthermore, when a field is declared for example, as char(50) in the database management system, even if only one character is stored in the data field, this occupies 50 characters in the file. Database management systems have many techniques to compact database files to their actual size either when the database is constructed or after the process has been completed. Taking this into consideration and the fact that in database management systems the necessary operations for sorting and querying data are available, it is more efficient to store the suffix array in a database than in memory. In addition, the system memory will not be used apart from performing the necessary calculations. Regarding the complexity of the algorithms for the creation of the lexicographically sorted suffix array various results can be produced. The best could be $O(n \log n)$ [9, 17]. However, a simple for-loop method to create the array first can be used and then the database management system can use its internal procedures to lexicographically sort the array. In this case the time complexity will be O(n) for the creation and $O(n \log n)$ for sorting the array, if the database management system uses merge-sort algorithm, which is the most efficient. The overall complexity will be $O(n + n \log n)$ or generally $O(n \log n)$. The relevant calculated results can be found in Table 5.1.

For the Calculate Occurrence Vectors Algorithm, several tests regarding maximum complexity (worst case) and average complexity have been run based on real financial data of Dow Jones Industrial Average Index 30, 1971–2011. Since COV Algorithm uses recursion, it is very difficult to calculate the exact theoretical worst complexity which can be estimated to be $O(10 \times n \times m \times 16 \times \log n)$ or generally $O(n \log n)$. However, so far the experimental findings have shown a time complexity for the worst-case scenario of $O(10 \times n \times m \times 16 \times 2 \times \log m)$ which is almost linear because it can be simplified as $O(n)$, where m is the length of the alphabet and n the length of the string with $m \ll n$. It is very important also to be mentioned that based

Table 5.2 Repeated pattern detection for real-case scenarios of DJIA 30 index (real-case scenarios B)

Case	1	2	3	4	5	6	7
Alphabet size (m)	9	9	9	9	9	9	9
String length (n)	100	200	400	800	1,600	3,200	6,400
Recursions (R)	57	109	204	398	770	1,522	3,154
Occurrences (O)	51	97	182	373	721	1,419	2,924
Instructions (I)	8,063	15,417	28,857	56,390	109,094	215,608	446,717

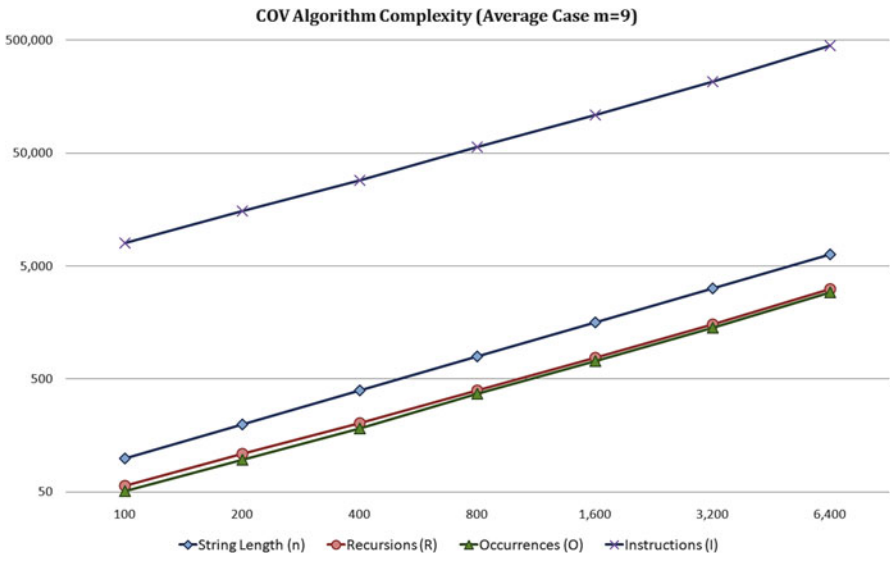


Fig. 5.8 Algorithm's 2 complexity diagram for real data (logarithmic scale)

on real-case scenarios of time series from financial data, the average complexity is $O(\frac{1}{2}mn)$ or generally linear $O(n)$. In both cases the complexity is depending on the alphabet, which is expected since the algorithm uses recursion based on the letters of the alphabet.

Thus, the overall complexity for the creation of the lexicographically sorted suffix array and the execution of the COV algorithm for the detection of the repeated patterns will be of class $\mathrm{O}(n + n\log n + n\log n)$ or in general $O(n\log n)$ while for the average case scenario with real data it will be $O(n + n\log n + n)$ or again in general $O(n\log n)$.

Table 5.2 shows some examples from the Dow Jones Industrial Average 30 index for the period 1971–1995 for different time classes from 100 days to almost 25 years data or 6,400 working days. Recursions and of course occurrences are linear analogue to the length of the string, which is important because it shows that the complexity is of type $O(n)$ as illustrated in Fig. 5.8.

Table 5.3 Repeated pattern detection results of mock data for worst-case scenario

Case	1	2	3	4	5
Alphabet size (m)	26	26	26	26	26
String length (n)	208	416	832	1,664	3,328
Recursions (R)	5,356	26,985	70,249	156,777	329,833
Occurrences (O)	136	364	780	1,612	3,276
Instructions (I)	2,100,331	10,579,939	27,541,507	61,432,440	129,226,590

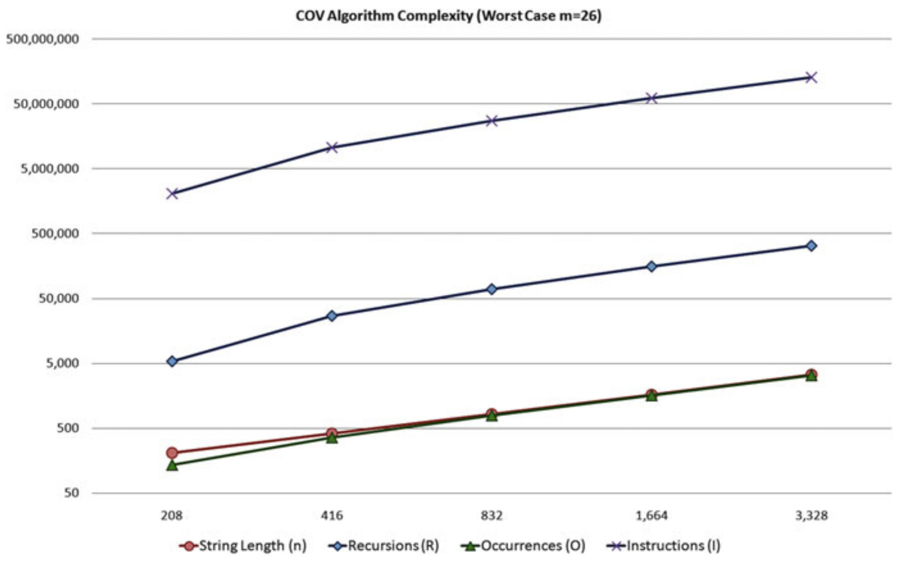


Fig. 5.9 Algorithm's 3 complexity diagram for mock data worst-case scenario (logarithmic scale)

In case of mock data that represent the worst case of a time series for Algorithm 2 the results presented in Table 5.3 can be found. What can be observed from the results is that the instructions and, therefore, the time complexity of Algorithm 2 are almost linear despite the theoretical approach described earlier regarding $O(n \log n)$ complexity. However, the theoretical worst complexity is introduced to avoid underestimation of the time complexity. These results have been illustrated in Fig. 5.9.

The major disadvantage of the described method is the allocation of large storage space on the disk and as a result significant delay in the process because of the slower data access than in memory. However, it can be considered as an advantage since the suffix array is created and stored once in the database and then it can be used in many different ways, e.g., selecting specific time regions to analyze without the need to recalculate each time the respective array. Furthermore, database management systems provide many tools (such as multiprocessing) that can significantly improve the performance of the algorithm. In addition, the alternative method to store the suffix array in memory has a major disadvantage. When computer's memory reaches its limits due to the large amount of data stored and the recursion of the algorithm, then the operating system automatically uses the virtual memory on disk which is significantly slower than the database storing approach.

5.5 Experiments with Financial Data

Suffix arrays and periodicity algorithms have shown some very interesting findings regarding financial data. The most common analysis that can be done is to search if there are specific patterns that occur in currency rate values per day change, by collecting the appropriate time series and then calculating the percentage change of each day from its previous day. Then the percentage daily data change has to be classified by taking into consideration some financial factors. Another type of analysis is to examine also the equivalent weekly data time series instead of daily. In this case, although we lose information, the analysis is more flexible because we can avoid significant noise from the data and its fluctuation.

5.5.1 Data Classification and Classes Construction

Something very important in statistical and financial analysis is the definition of the alphabet. The length of the alphabet (how many discrete ranges exist) and the way it is defined (what are the limits of its range) are very important in order to have credible results.

The most common way to define value regions and therefore the alphabet are, first of all, estimation of the quartiles, deciles, or percentiles. After deciding how many classes are needed or in other words the alphabet used, the calculation of the width of each class follows, in order to discretize the sample and start the analysis.

5.5.2 Financial Aspects

Many statistical tools and methods in stock market analysis, known also as technical analysis, have shown that there are patterns that occur periodically in financial time series and especially major currency rates such as US Dollar and Euro. The most common results fit with the major economic cycles. Each economic cycle can have several growth and recession periods, something that stock markets tend to follow. Moreover, regarding specific major stocks, we can have again results based not only on economic factors but also on sector and market factors.

Periodicity algorithms can detect periodic patterns that might be useful in technical analysis. However, time periods and time ranges are hardly the same, which makes extremely difficult for periodicity detection algorithms to scan and reveal periodicities in standard time series. Furthermore, it has to be mentioned that

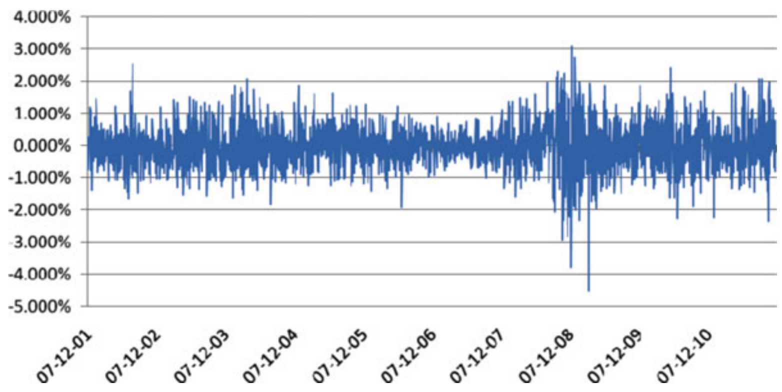


Fig. 5.10 Daily fluctuation of currency rate between US Dollar and Euro

foreign exchange markets are chaotic systems based on many factors that are not necessarily financial (like human behavioral) and it is very difficult to find distinct and clear periodicities like in traffic or other models. More complex models should be defined in order to have credible results. For example, extreme noise might exist in time series because of holiday seasons, different calendar days each Central Bank decides every quarter to report financial facts, figures, interest rate changes and currency policies, even natural disasters like earthquakes, hurricanes or even power failures, which may distort data that otherwise could be periodic. Such an example is the case of the recent earthquake in Japan (summer of 2011) which significantly influenced the US Dollar vs. Japanese Yen currency exchange rate. Moreover, in the case of currency rates, they are directly influenced from many other factors such as inflation and interest rate. Therefore, the development of new approaches, models, and solutions that could eliminate such distortion is needed.

5.5.3 Experimental Results

In the current chapter, data from the US Dollar Euro currency exchange rate will be examined. The specific exchange rate is very important since it represents the two major currencies of the world today. Moreover, because of the financial and state debt crisis of the past 3 years and its further political and economic aspects, the values of the exchange rate have been heavily influenced and present major uncertainty and high volatility in their prices. Currency markets represent transactions of trillion dollars and are very important because they can have significantly high impact on state economic policies, trade, imports, exports, etc. Therefore, it is of great importance to check if there are any patterns that might have periodicity with great confidence.

In the particular research, two different time series will be used for daily changes (2,508 observations) as it is presented in Fig. 5.10 and weekly changes

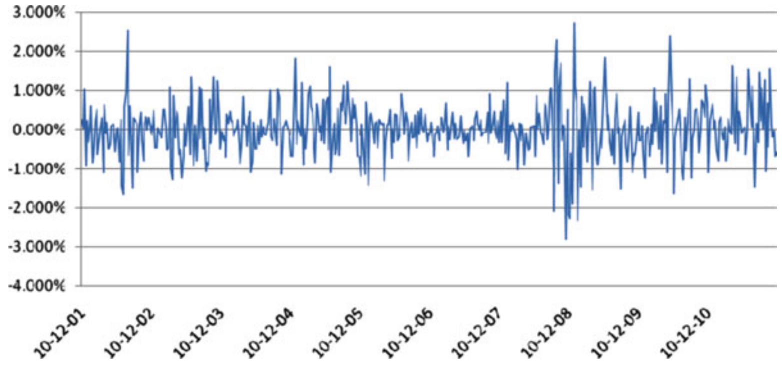


Fig. 5.11 Weekly fluctuation of currency rate between US Dollar and Euro

(518 observations) as depicted in Fig. 5.11. The alphabet used consists of four $\{ABCD\}$ letters for quartiles for each time series. It is also important to mention that in the case of weekly data (instead of more commonly daily), the existence of small patterns of even three or four letters long is very important since they are describing overall longer time periods. A four-letter pattern in weekly data can be considered very important since the underneath time region length is actually 1 month. The data values for the US Dollar Euro currency exchange rate are based on records taken from the Canadian Central Bank's website[1] and time horizon from December 2001 till December 2011.

The first time series of daily data, as it is presented in Fig. 5.10, has an equivalent time series string after applying the alphabet as it is illustrated in Fig. 5.12. The value ranges that each letter represents are: $A = [-4.51, -0.39]$, $B = (-0.39, -0.01]$, $C = (-0.01, 0.34]$, and $D = (0.34, 3.09]$. The second time series of weekly data, represented in Fig. 5.11, has an equivalent time series string after applying the alphabet as it is illustrated in Fig. 5.13. The value ranges that each letter represents are: $A = [-2.81, 0.37]$, $B = (-0.37, 0.02]$, $C = (0.02, 0.36]$, and $D = (0.36, 2.72]$.

In both cases the value of 0.67 has been used as a confidence limit in order to have meaningful results. For example, if a pattern has perfect periodicity 20 and only 2 occurrences, Algorithm 3 will report it as an occurrence; however, for statistical purposes it is not important since it will have confidence 10%. For this reason the confidence has been set to equal 0.67 or greater than 0.67 in order to find more reliable outcomes. It is very important to mention that periodicity detection algorithms have an important effect on analysis. Since they calculate confidence based on the theorem described earlier in the chapter, they tend to lose potentially important results because repetitions used in calculating the confidence start counting from the moment the first pattern occurrence happens till the end. If a pattern starts very early in the time series and also stops very early (in the middle for example) the confidence will be very low. That is not in general bad for the pattern

[1] www.bankofcanada.ca/rates/exchange/10-year-converter.

```
CCBABACCCCDDACAADDCCDACBCDCBCBCDDDABDBACBCAACDCBCADBADDDBCACABDBCBABBCBCDCBDCACBCBDACA
DBBABCCBBACACACABCDABCDABACADCCABADAADACCCBDBCAAAAADAACCCDDAADBCABCCADDBADDADABDADAABA
DBDBBDBBABBCACCDCDCBCCDABBCCCDBACCBDCCACCDBDCBACCBABCAACBCAACDCABBDDCCCCACCABCABCBABCA
CDADAAABDAADBBACBBCABBABABCBDCBACCCDCCACDCBADCBBCBAAABBACCDDDBCBDABBBAABDCCDABADBBABDA
CACDBAACAACAAADCDAABCAACDADCCBADADACABCDCDDDADCABCCDDAADCDCDAACADACCDDBABBDBDBDBDCCDDB
CBCADCAAAABCACDABAABCBCAAABDAABDADDBCBBACBDDCBDDBCDABAABBABCBDBACACABCAABCDABBACBBCABB
ACAADAADDCDDCABADDACDACACAACAACDBACDDBADDCCDDAACADBDABDADADDDDAAAADDAADCDBCACDDBDAADAB
CAADDDCADAACADAACADCBBDAADDBDAADBACCCACBDAACBABCBDACACDDBDBDCBCBBCACBDCACCCADDDCCDBACC
DBABACBDBCCACBCACBACDBCCADDAABCACBACCBCCDAAACCCBCBADACBAABCCBCABBDBCADADBACBABBADDBDDB
BADDCCCDABDADBBCBDCDBBACACBACCACCCADDCACABBADCADDDBDDDBBBDDBBBBADBDAACBBCDCCCBDBBCDBBD
DDCBBCDBBDADDDBCBBBDDDCACADCCDBADBCDBBBCAADCDBDBACCDCABAABDBCCACDCDDCACBABDCAAADCDBDCB
DCDBBDDCDBBBDBAACDBDACDBCCBACADDBADDCCCCDBCBBCBACDACBDBACDACADADBCDCACBCCAABBDCADABDBB
CABCCDCADBDDBCBDCCCCBBCCBDCACABCDBBCAABABCDBDACADABCABDDCBCCABACBBBAAACBCBBCBAABDCDADA
DBCCACCABDCDCDCACBDCADDBBDCABDBADBDCCDCAABDDAABBBBCADBDCCAACDADCCBBCACBCCCDBCBBDBBBBAD
DBCCACCCDDBCCBBDABDBBAACACBDBACABDCCBCBABABBCAABCCCDBBCCDCACBDCBBADDDBCDCBBCBCBCADBDBB
BBDCBBBCCAABCBCCBBCACDBDBBCCABBBACCBADBBCBBCCCADACBBBBBBCCBBDBBDCCBBDBCBBADCBDBDBBBDBC
BBCCDBCCBABBBCBBBCBBABCBCABBBCCBBBCBDBDBBCBACADCDCDDACBBBABCBCCCABABBACCBBAACBBCBABDCC
BDAADBDBACDACAABBBCBCAABCDACBBABBCBCDBDCADCBBCCDDDBDCBBAADACBDBDABACDBDDACADACBABCDBDB
BACABCADABCAABBBBAACBCAABABDDCAABCBDBBACCADBBCACDAADDCBCCDCABDBBACCBAAADACCDDBCDCAACDA
DCABCBADACABCADBDBABABBDBCBDDDBBDCBCDDDDDDCCDBBCADBDBBCDCDDDBDDABCCABABDCCDDCDCDABCDAD
DBDDDBDDBACDBABDBCDCACCDDBAADADDADADACAAAAAADDABCADDBDDACDDDCDADDDADABADDCADCBADDDADDC
ABACCBDDCACACABBAACAADBCBBDDACACDDBDADDBDDABBADDADBABDAAABCCDAABACCDBAAADCDDACADDACABD
ACDABCCCDBDADBCACBACBBDCDDBAACCDCCAADDBACABBDBACDBBCAACBBBABCCACDCCDADABADADAABDBCACBD
DDBDADABCACDBADADCACADCBACBDDDCCDBDADCBDABCCBACBDBACCCDBDDBBCDCDDBBDDDAADDCCADDCBDBDAD
BADBBCABADACDDCBDCAADABDDCCAACADDCCDDACDDAADDDDBADDDDDACAADDACDCDCDCBAACAACBCBDDACCDAA
DADACDABABBDDACACBABADCACDDCDAACCDCBCABDBABADBCBAACACBAACACABCADAACBDAADCDABDADDACCACA
DDDDCDDBABCDCBDDDAAADBDCBABDDDCBCBBAAADDDDCBAACCAADAACBCDAACDDCABDDCBBAABCABCBBAABCDCD
BACCAABBDADCCBABBBACABCBCCDAAACBBABBBCDDDADCDADACDDCCACABCABBDDDCADDACABDBAAAACDDBDDCA
BCDABACCADDADADCABDBBACBDACBCCABDCDDDADDCBBADDABDACABBDDBBAAAADADDADAACDACDDADBCBDBBDC
BCBCDCDABABDAD
```

Fig. 5.12 Time series string for daily data

```
CCDACBCDAACDABBCADBCAACCCABCACAADDDADBACCCACDBACCBCCBCDAACBBBDDCAADAADBDDAAAACABDADCAC
ABDBDACAAADBDDBBDCABBBADCDCCBBCCAAADCCACDAACAACACBCBBCCDBBCCADDBACCCCCBAADACBBDABACDDD
DAADDBCBDBDDADABCABDADDDDCDDCBDCDCAAACADBACDCBCCACCCCABCCDABBDDBBDDDBDCABCBBDADBDCBDCB
BBCABBCBCCBCDACBBDBCBBBCBBBBBACCCBCDCBCBBBBDBDBCDBBCBDBDDADACBCBBBACBCBABAAACCCADBDBBA
DDBBDDBADDADDBCCADAAAADDDABADADCABDDADDAAABADDDBDBABACDBCABCCAAADAAAABDBBCAABCBACADBDB
ADACCDADDDCABCCDBAACACDDABBDDCACDDCDDACDBCAACCBBCABCBBDCBDADCBBCACDDCDAACBDCDBDADADCBB
AA
```

Fig. 5.13 Time series string for weekly data

since it might be considered as an interesting pattern in the time series that might occur again in the future e.g., after a period in time, however, periodicity detection algorithms will discard it if confidence is set to high because of the time gap.

In the first case (Table 5.4) 1,557 distinct patterns have been found with length 1–11. There are some results with zero time tolerance and for patterns with length 3 and 4 characters that occur with great confidence 3 or 4 times in the time series. If we change time tolerance from zero to up to three, more long patterns can be found. That is expected since in this case the time tolerance reduces the noise between the pattern's strings. It is very important to be mentioned though that in the case of daily data, significant findings are not expected due to the high randomness of the data. After all, as being already described and analyzed, foreign exchange markets and financial markets in general are chaotic systems.

Although exchange rates are difficult to be predicted with daily data, several patterns have been found which have some kind of periodicity in which their number is significantly smaller than the total number of occurrences found. It is very important though to analyze the second time series which represents weekly

Table 5.4 Indicative daily data results

Pattern	Starting position	Period	Occurrences	Confidence level	Tolerance
AD	2,387	29	4	0.80	0
AD	2,421	24	3	0.75	0
ADD	1,269	329	3	1.00	0
BD	2,329	53	3	1.00	0
CCA	2,089	142	3	1.00	0
DAD	2,366	52	3	1.00	0
DCCA	293	879	3	1.00	0
D	2,321	9	15	0.71	1
ADD	1,269	329	3	0.75	1
BDA	2,431	24	3	0.75	1
CCA	2,089	142	3	1.00	1
DAD	2,366	52	3	1.00	1
DDA	2,251	110	3	1.00	1
ACBD	1,520	454	3	1.00	1
ADBC	1,903	290	3	1.00	1
CDCC	292	554	3	0.75	1
DCCA	293	879	3	1.00	1
DCDD	2,039	200	3	1.00	1
A	1,826	8	58	0.67	2
B	2,225	18	12	0.75	2
C	1,975	11	33	0.67	2
D	1,313	14	58	0.67	2
DA	2,368	21	6	0.86	2
CDA	2,365	44	3	0.75	2
ACAD	1,534	348	3	1.00	3
BAAA	1,615	259	3	0.75	3
BCBB	1,392	436	3	1.00	3
DDAD	2,417	26	3	0.75	3
DDCC	980	694	3	1.00	3

data, a more compact version of the first case (Table 5.5), as it is mentioned before, weekly data have the ability to normalize the daily data and reduce in a great degree the noise and any abnormal movements because of external factors. Therefore, despite the fact that the COV algorithm has found only 329 distinct patterns in the weekly data, the results from their analysis are much better.

Even without the use of time tolerance, several patterns exist with length up to four characters. It is also important to be mentioned in this case that since each character represents a week, a four-character length pattern represents a month, which is a very long time for exchange rates or stock markets in general. Beside the single character patterns, which are also important since they represent a week, two patterns for three and four characters have been found, and more specifically, ADB and CDA with confidence 0.75, and ADBD and DACB with confidence

Table 5.5 Indicative weekly data results

Pattern	Starting position	Period	Occurrences	Confidence level	Tolerance
A	501	8	3	1.00	0
B	492	11	3	1.00	0
D	469	15	3	0.75	0
DA	466	21	3	1.00	0
DC	374	65	3	1.00	0
DD	238	75	3	0.75	0
CDA	125	103	3	0.75	0
ADBD	249	88	3	0.75	0
DACB	2	157	3	0.75	0
A	441	6	9	0.69	1
B	503	4	3	0.75	1
D	398	10	9	0.75	1
AC	333	63	3	1.00	1
BC	357	42	3	0.75	1
BD	427	29	3	0.75	1
DC	497	7	3	1.00	1
BBC	142	116	3	0.75	1
A	396	5	19	0.76	2
B	115	20	15	0.71	2
C	461	3	14	0.74	2
A	243	6	32	0.70	3
A	324	5	28	0.72	3
B	70	9	34	0.68	3
C	63	14	23	0.70	3
D	36	8	41	0.67	3
D	44	10	33	0.69	3
AB	329	40	4	0.80	3
BC	357	42	3	0.75	3
AAC	447	25	3	1.00	3

0.75, respectively. However, ADB is encapsulated inside the ADBD with different occurrences since the first starts from very early in the time series. Yet, the fact that patterns which represent months in data can be found it is very important and has to be further analyzed in terms of the financial point of view for correlations with other factors, e.g., political.

In the case that time tolerance is used, more interesting results have been identified. More specifically, significant number of occurrences has been found for one-character long patterns. Patterns like A, B, C, and D can be found tenths of times with time tolerance 1, 2, or 3. Furthermore, patterns with two letters have also been found to occur with the time tolerance. Table 5.5 includes some of the findings for the weekly data; however, many more have been found but not included to keep both tables readable.

5.6 Conclusion and Future Work

The current chapter has introduced a new methodology for periodicity detection in time series by using suffix arrays instead of the most commonly used suffix trees. The methodology proposes an algorithm for the construction of the suffix array and another algorithm that searches the sorted suffix array and returns all the repeated patterns. For the calculation of the confidence of the results from the algorithm, perfect periodicity has been introduced. Perfect periodicity calculation is based on the relative theorem that has been proven in this chapter.

Regarding the algorithm complexity, the process of creating a lexicographically sorted suffix array and calculating the repeated patterns (occurrences' vectors) has been computed to be $O(n \log n)$. Especially Algorithm 2, which searches the suffix array to find all the repeated patterns in the time series, has an average complexity of $O(n)$. By using suffix arrays, extremely large time series can be analyzed since they can be stored in a database management system which also has sorting and querying facilities.

The proposed methodology has been applied in US Dollar Euro currency exchange rate which is governed by chaotic models that can be compared more to random walks and therefore there were no great expectations from periodicity detection. The data analysis though has shown that there is some kind of periodicity in several cases which depends on the selected time interval and the alphabet chosen. Moreover, the improvement of the Periodicity Detection Algorithms [16] can lead to more sophisticated and important results, which can trigger off future research for further improvement of both Calculate Occurrences' Vector Algorithm and Periodicity Detection Algorithms.

In future work, new storing approaches will be sought to improve the need for storage space of suffix arrays as it is identified as its main drawback. By achieving less space capacity and in combination of the repeated detection algorithm introduced in this chapter, suffix arrays could be transformed to a powerful tool in time series data mining.

References

1. A. Al-Rawi, A. Lansari, F. Bouslama, A new non-recursive algorithm for binary search tree traversal, in *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems* (IEEE Computer Society, Washington, DC, 2003), pp. 770–773
2. C.-F. Cheung, J.X. Yu, H. Lu, Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Trans. Knowl. Data Eng. **17**(1), 90–105 (2005)
3. M.G. Elfeky, W.G. Aref, A.K. Elmagarmid, Periodicity detection in time series databases. IEEE Trans. Knowl. Data Eng. **17**(7), 875–887 (2005)
4. M.G. Elfeky, W.G. Aref, A.K. Elmagarmid, WARP: time warping for periodicity detection, in *Proceedings of the 5th IEEE International Conference on Data Mining* (IEEE Computer Society, Washington, DC, 2005), pp. 138–145

5. F. Franek, W.F. Smyth, Y. Tang, Computing all repeats using suffix arrays. J. Automata Languages Combinatorics **8**(4), 579–591 (2003)
6. D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (Cambridge University Press, New York, 1997)
7. J. Han, Y. Yin, G. Dong, Efficient mining of partial periodic patterns in time series database, in *Proceedings of the 15th International Conference on Data Engineering, ICDE '99* (IEEE Computer Society, Washington, DC, 1999), p. 106
8. K.-Y. Huang, C.-H. Chang, SMCA: A general model for mining asynchronous periodic patterns in temporal databases. IEEE Trans. Knowl. Data Eng. **17**(6), 774–785 (2005)
9. J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction. J. ACM **53**, 918–936 (2006)
10. P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays. J. Discrete Algorithm **3**, 143–156 (2005)
11. U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (Society for Industrial and Applied Mathematics, Philadelphia, 1990), pp. 319–327
12. E.M. McCreight, A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976)
13. G. Navarro, R. Baeza-Yates, A new indexing method for approximate string matching, in *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, ed. by G. Goos, J. Hartmanis, J. van Leeuwen, vol. 1645 of Lecture Notes in Computer Science (Springer, Berlin, 1999), pp. 163–185
14. G. Navarro, R. Baeza-Yates, A hybrid indexing method for approximate string matching. J. Discrete Algorithm **1**(1), 205–239 (2000)
15. F. Rasheed, R. Alhajj, Using suffix trees for periodicity detection in time series databases, in *Proceedings of the 4th IEEE International Conference on Intelligent Systems*, vol. 2, pp. 11/8–11/13, Varna, Bulgaria, 2008 Sept. 6–8
16. F. Rasheed, M. Alshalfa, R. Alhajj, Efficient periodicity mining in time series databases using suffix trees. IEEE Trans. Knowl. Data Eng. **22**(20), 1–16 (2010)
17. K.B. Schürmann, J. Stoye, An incomplex algorithm for fast suffix array construction. Software Pract. Ex. **37**(3), 309–329 (2007)
18. C. Sheng, W. Hsu, M.-L. Lee, Efficient mining of dense periodic patterns in time series. Technical report, National University of Singapore, 2005. Technical report TR20/05
19. C. Sheng, W. Hsu, M.-L. Lee, Mining dense periodic patterns in time series data, in *Proceedings of the 22nd International Conference on Data Engineering* (IEEE Computer Society, Washington, DC, 2006), p. 115
20. W.F. Smyth, Computing periodicity in strings – a new approach, in *Proceedings of the 16th Australasian Workshop on Combinatorial Algorithms*, pp. 263–268, Victoria, Australia, 18–21 Sept 2005
21. Y. Tian, S. Tata, R.A. Hankins, J.M. Patel, Practical methods for constructing suffix trees. VLDB J. **14**(3), 281–299 (2005)
22. E. Ukkonen, Online construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)
23. P. Weiner, Linear pattern matching algorithms, in *Proceedings of the 14th Annual Symposium on Switching and Automata Theory* (IEEE Computer Society, Washington, DC, 1973), pp. 1–11