Appl Intell (2014) 41:941–955
DOI 10.1007/s10489-014-0553-x

Analyzing very large time series using suffix arrays

Konstantinos F. Xylogiannopoulos · Panagiotis Karampelas · Reda Alhajj

Published online: 23 August 2014

Abstract Suffix arrays form a powerful data structure for pattern detection and matching. In a previous work, we presented a novel algorithm (COV) which is the only algorithm that allows the detection of all repeated patterns in a time series by using the actual suffix array. However, the requirements for storing the actual suffix strings even on external media makes the use of suffix arrays impossible for very large time series. We have already proved that using the concept of Longest Expected Repeated Pattern (LERP) allows the actual suffices to be stored in linear capacity $O(n)$ on external media. The repeated pattern detection using LERP has analogous time complexity, and thus makes the analysis of large time series feasible and limited only to the size of the external media and not memory. Yet, there are cases when hardware limitations might be an obstacle for the analysis of very larger time series of size comparable to hard disk capacity. With the Moving LERP (MLERP) method introduced in this paper, it is possible to analyze very large time series (of size tens or hundreds thousands times larger than what the LERP can analyze) by maximal utilization of the available hardware. Further, when empirical knowledge related to the distribution of repeated pattern's length is available, the proposed method (MLERP) can achieve better time performance compared to the standard LERP method and definitely much better than using any other pattern matching algorithm and applying brute force techniques which are unfeasible in logical (human) time frame. Thus, we may argue that MLERP is a very useful tool for detecting all repeated patterns in a time series regardless of its size and hardware limitations.

Keywords Suffix arrays · Repeated patterns detection · Data mining · Time series · DNA analysis

K. F. Xylogiannopoulos · R. Alhajj (✉)
Department of Computer Science, University of Calgary, Calgary, Alberta, Canada
e-mail: alhajj@cpsc.ucalgary.ca

K. F. Xylogiannopoulos
e-mail: kostasfx@yahoo.gr; kxylogia@ucalgary.ca

R. Alhajj
Wroclaw University of Technology, Institute of Informatics, Wroclaw, Poland

P. Karampelas
Department of Information Technology, Hellenic American University, Manchester, NH, USA
e-mail: pkarampelas@hauniv.edu

R. Alhajj
Department of Computer Science, Global University, Beirut, Lebanon

P. Karampelas
Department of Informatics & Computers, Hellenic Air Force Academy, Attica, Greece

1 Introduction

This paper introduces a new methodology for the analysis of very large time series using suffix arrays. In [1] we introduced a novel, recursive and very efficient algorithm (COV) that has the unique property of detecting all repeated patterns in a time series with $O(n)$ complexity on average. For the past decades, great effort has been applied first on the construction of efficient data structures for pattern matching (i.e., suffix trees and suffix arrays) and, respectively, on the construction of pattern matching algorithms. Yet, although many algorithms exist that take advantage of the previously mentioned data structures, there are no algorithms for the detection of all repeated patterns. Of course, any of the already existing algorithms, e.g., [2–6] and [7] can be used for the detection of all repeated patterns, yet, we will show that this is unfeasible for very long time series and even for patterns with very small length inside any kind of time series. Another feature of our methodology is that although the COV algorithm can be directly executed in memory, we prefer to store the suffix array data structure on an external database management system. This gives us the advantage to reuse the data structure whenever we want for further analysis without the need to reconstruct it and overcome the problem of limited memory which can be of a ratio 1/100 compared to a hard disk.

The tradeoff for the use of an external data storage is that in order for the COV algorithm to detect all repeated patterns in a time series, the actual suffix array has to be constructed. Due to the quadratic space-required capacity of a suffix array, the use of the data structure is limited to very small time series. Trying to solve this problem, we proved in [8] that only suffix strings of length half the length of the original time series need to be stored for the detection of all repeated patterns. Although now the suffix array has smaller size, the quadratic capacity remains and this does not allow the analysis of very large time series. To overcome this problem, in [9] we introduced the concept of the Longest Expected Repeated Pattern (LERP). More specifically, we showed that if we know the length of the longest expected repeated pattern in a time series then we have only to store all suffix strings with length less or equal to this value. However, in [10] we managed to construct and proved the Probabilistic Existence of Longest Expected Repeated Pattern Theorem which allows the pre-calculation of the LERP value without any knowledge of the time series rather than only its length and the assumption that it is random. This technique allows the linear required space capacity storage of the suffix array data structure without any loss of information. Our algorithm is then able to analyze any kind of time series regardless of its size and, therefore, the external media data storage. By expanding our previous work and more specifically the Longest Expected Repeated Pattern (LERP) concept introduced in [9], this paper proposes a new method for the maximal utilization of the external media storage used to store a suffix array in which through a finite loop detects all repeated patterns that exist in a time series. Although with the introduction of LERP it is possible for time series of length up to hundred million or billion characters to be analyzed [11], the methodology proposed in this paper can further expand this size up to tens of billions or even more, depending only on the limitations of the external storage media that will be used to store the suffix array. The experimental results have shown that although the new methodology is slightly slower in comparison with LERP, it is significantly space efficient. This is acceptable since the objective of the new methodology is to analyze very large time series by challenging the main problem of the enormous space needed for storage.

A time series is a representation of a time dependent variable over time (e.g., environmental temperature). Yet, any fixed alphanumeric string based on a predefined and constant alphabet (e.g., chromosomes or proteins) can be also considered as a time series. For the analysis of time series, alphabet categorization and discretization is needed in order to analyze continuous values or values that fluctuate in great ranges which otherwise would be meaningless to analyze. For example, atmospheric temperature in Canada may vary from -35 degrees up to $+35$ degrees Celsius. In a daily analysis, or even worse hourly analysis, the outcome of the measurements will be meaningless for statistical purposes if not categorized, since there are 76 discrete numerical values in the specific range or 7,600 if we use real values with two decimal digits. In order to create a time series of temperatures that can show seasonal patterns, temperatures have to be discretized in ranges and in each range, a symbol (i.e., letter, number, etc.) from a predefined alphabet can be assigned. This categorization helps to extract valuable information of patterns regarding weather conditions, since we are interested in general conditions of how cold or warm the weather can be and not in specific temperature values.

After the process of discretization is completed and the time series is constructed, the data should be stored for further analysis. The most common technique so far is based on a very powerful data structure, the suffix tree [12, 13]. Suffix trees are more preferred because they require low space capacity and they lead to very good time complexity that allows fast analysis of time series. Another data structure used for this purpose is suffix arrays [13–16]. Both data structures hold information about the suffix strings of a time series. A suffix string is a substring that represents part of a time series string after a specific position (suffix). Therefore, a time series can have exactly $n-1$ suffix strings, plus one which is the complete time series itself. Furthermore, a suffix array stores the lexicographically sorted list of all suffixes [14–16]. It has to be mentioned that all current data structure construction methods are temporary, regardless whether they are a combined method of memory and external storage media. When the memory resources are released then the data structure is no longer available.

While the creation of the data structure is the first important step in time series analysis, two equally important processes follow. First, all occurrences of repeated patterns should be detected in the time series and then the outcome should be analyzed for finding potential periodicity among them. This paper builds upon the LERP methodology introduced in [8] and it will not focus on the last process of periodicity detection. It will mainly concentrate on the two first steps in periodicity detection, i.e., the reduction of the required space capacity of a suffix array and then detection of all repeated patterns in the time series. The reported experimental results demonstrate the applicability and effectiveness of the proposed approach.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 defines the problem which will be solved. Section 4 presents the proposed methodology. Section 5 discusses the findings by applying the proposed methodology to DNA time series analysis. Finally, Section 6 is conclusions and future work.

2 Related work

Pattern and periodicity detection have been proved to be problems of great interest. For this purpose, many different algorithms and methods have been introduced to perform pattern, e.g., [2, 5, 7, 17–19] and periodicity detection, e.g., [12, 20–22]. Moreover, the past decades have witnessed the utilization of two powerful data structures for time series analysis, namely suffix trees and suffix arrays. Both are based on the suffix string concept, which is a substring of the initial time series string. Suffix strings can be extracted from a time series by finding each time the string beyond a specific position in the time series.

A suffix tree is a representation of all suffixes of a string in a tree format [13]; it is considered a very powerful data structure [20, 21, 23]. Suffix trees are heavily used for data mining purposes in many scientific and commercial fields (e.g., financial, marketing, frequent item-sets detection, DNA) due to their advantages in string processing [12, 21, 22, 24]. Several methods and algorithms have been introduced for the construction and analysis of a suffix tree, e.g., Weiner [17], McCreight [25] and lately the algorithm of Ukkonen [26] which creates the suffix tree with linear $O(n)$ complexity. One of the main advantages of suffix trees is the small storage space required. This is important because suffix strings have to be stored in memory in order to be further analyzed. Techniques for external storage have been introduced lately in order to analyze larger time series. However, this might be a problem if the suffix tree cannot fit in the main memory and external storage is required, since existing analysis algorithms may crash [23, 24]. That may happen because the current algorithms cannot access from the suffix tree the part which is temporarily saved on external media. To overcome this problem, new techniques have been introduced recently, e.g., [3, 4, 6, 23, 24] that allow for the analysis of the data stored outside the main memory.

Suffix arrays do not suffer from this specific problem since they can be stored and accessed directly from external media such as hard disks. However, the read-write I/O process is very time consuming compared to the direct access in main memory. For very long time series, this can be a significant drawback which forbids sometimes the analysis of time series. For a suffix array, one of the limitations is the required space capacity in case of the full suffix array which is $O(n^2)$ based on Gauss formula for the summation of series $\sum_1^n k = \frac{n*(n+1)}{2}$ and it is very important for our methodology. The latter series represents the summation of the elements of all the $n-$ 1 suffix strings including the original time series. Although the space required to store a suffix array on disk is very large, suffix arrays (as will be proved in this paper) can behave better in the analysis of large time series providing a great advantage compared to the memory limitations of suffix trees. However, Manber and Myers in [16] introduced a very efficient, linear complexity methodology for the construction of the suffix array using only the initial time series and the indexes of the lexicographically sorted suffix strings, which can significantly reduce the required space for the data structure and make it more efficient than suffix trees. Furthermore, great efforts by many researchers over the past years have been done in order to utilize more the suffix array by using a combination of memory and external media storage on the hard disk, e.g., [2, 5, 27].

So far the main efforts of researchers who used suffix arrays and trees have focused on the optimization of the construction time [28–30] and more specifically on constructing the structure with linear time complexity [15, 29]. Especially for suffix arrays the construction complexity depends on the sorting process [15, 31]. Many techniques have been introduced that focus on the Longest Common Prefix (LCP) [14, 15, 29], denoted lcp(a, b), which is the longest common prefix between strings a and b.

The main advantage of suffix arrays compared to suffix trees is the fact that they can be easily constructed and stored to external media, something that cannot be generally achieved for suffix trees [2, 5, 23, 24, 27]. Moreover, suffix arrays may need less storage capacity compared to the size and the expected space by other structures, especially when the alphabet is very large [5, 16]. On the other hand, a disadvantage of suffix arrays compared to suffix trees is the huge amount of data that should be stored when actual suffices have to be stored as well [16, 31].

In [8] the concept of Longest Expected Repeated Pattern (LERP) has been introduced; it can be extremely useful in reducing the size of the actual suffix array to be stored. With the specific methodology, the size of the time series that can be analyzed for pattern and periodicity detection purposes can be expanded. Nevertheless, using this methodology the size of the time series that can be analyzed depends on the size of LERP. More specifically, the number of rows of the suffix array constructed in the database management system is equal to the time series length, while the size of the column that holds the suffix strings is equal to the size of the LERP. Assume the storage limit of the external media, due to hard disk limitations, is equal to one

Terabyte; if we set LERP equal to 1,000 we can approximately analyze a time series of up to 400 MB since the remaining 600 GB of the database are consumed by the metadata and the indexes. With this kind of hardware limitations we cannot analyze the full human genome (which is of size 3.5 GB) and detect all repeated patterns. The only way to detect all repeated patterns in the full human genome is by using as LERP the value 110 which will allow the construction of the actual suffix array. This limitation can form a significant effect on the analysis of the time series since the larger the time series is, the larger the expected occurrences are, and hence a relatively large LERP should be chosen. This happens because with a large LERP we will expect to discover all or at least a large number of the occurrences.

The methodology introduced in this paper can solve this problem by maximal utilization of the hard disk capacity and by storing the suffix array. More specifically, the required space capacity of an actual suffix array is of type $O(n^2)$ [1, 8, 9], where n is the length of the time series. With the LERP methodology, the required space capacity can be reduced to linear $O(l * n)$ (or $O(n)$ since l is a constant) depending on the ratio of the time series and the selected LERP (l) [9]. Yet, with the proposed methodology the required space capacity (as will be proved) is again linear $O(n)$. However, with the new methodology the analysis of very large time series of billions of elements is feasible. As a consequence, for example, the analysis of the whole DNA sequence can be feasible. So far this has been impossible since for a time series of size 3.5 billion characters as in a human DNA chain with selected LERP equal to 5,000, the required space capacity is almost 35 Tb. Besides the enormous amount of the required storage capacity, even the time needed for the construction of such a suffix array could be impossible. However, following the previous example of the one Terabyte storage limitations, we can construct an actual suffix array with LERP 100 and then perform a second phase analysis for substrings longer than 100 based on the results found from the COV algorithm in the first phase. The full process and the methodology of the Moving LERP as we have named it will be thoroughly described in the next section.

3 Our approach

In order to address the enormous required space capacity for analyzing very large time series using suffix arrays, we propose an innovative process which expands the LERP methodology. More specifically, the proposed Moving-LERP (MLERP) methodology is based on finite loops of incrementing LERP values. The MLERP methodology will help in analyzing extremely large time series since the limitation of the analysis is restricted only to the size of the time series and does not depend on the ratio of the time series and LERP size. The proposed methodology is presented in this section.

3.1 Moving LERP (MLERP) process description

Suppose that we have a time series of 40 billion characters that we want to analyze and detect all repeated patterns or perform just a simple single pattern matching. The direct construction of such a time series on the memory is impossible since we need it for a suffix tree at least 480 GB of RAM [5] while for a suffix array (not the actual) we need at least 360 GB of RAM [16] (only for the data structures). If we assume that the limit of a database management system we have can store up to 10 Tb to a hard disk then the maximum LERP that can be used as previously explained is only 100. In the first loop of the MLERP process, the LERP value 100 is assigned. By completing the time series analysis for occurrences detection, a significant amount of substrings with length 100 is expected to appear as occurrences. Of course, all these substrings are not the same because they are very short in length and definitely they are part of longer and different substrings that might be occurrences as well. Depending on the identified number of occurrences with length 100, a new value for the LERP can be assigned so as not to exceed the database size limitation. For example, if the total number of these occurrences is half the size of the original time series, then we can repeat the process of the calculation of occurrences only for these substrings by doubling the LERP. Namely, in this second loop only the occurrences found with LERP 100 from the first loop will be analyzed. Moreover, there is no need to calculate occurrences with length 1 to 99 since in the first loop of the MLERP process we have already found all substrings with length 99 and many with length 100. The substrings with length 99 are confirmed occurrences and there is no purpose to examine them again, since they will give us the same results. Thus, the process in the second loop reconstructs the suffix array and analyzes all substrings that were reported as having length 100 in the first loop. In the second loop, the new value of LERP will be 200, which means a new suffix array is created with, e.g., half the rows and double the size of the substrings. The process continues with the analysis and discovers all patterns that occur at least two times and have length from 100 up to 199. Moreover, occurrences might be discovered with length 200. In this case, the MLERP process starts a new loop to examine if the substrings with length 200 hide longer patterns that occur more than two times. The new loop will start searching for patterns from length 200 (the previous LERP) up to the new LERP, i.e., 400. This process can be continued until there are no substrings with length larger than the current LERP. If in the process of doubling each time the LERP will be used, then after 10 iterations a LERP of size $100 * 2^{10-1} = 100 * 512 = 51{,}200$ will be reached; this is very large and probably adequate for the analysis. The total steps of the process are finite and definitely very few, since it is exponential with base 2, e.g., for 32 loops we have LERP approximately equals to $2^{32-1} = 2{,}147{,}483{,}648$. Moreover, the current process can be stopped sooner if we are interested in patterns with specific characteristics, e.g., patterns with length not longer than a specific value.

With the proposed methodology, it is possible to analyze time series up to 1 trillion characters starting with LERP equal to 4, if we assume the database management system limit for the database's size due to hard disk limitations is 10 Tb. Thus, with LERP equal to. e.g., 4,000 we can analyze time series of size up to 1 GB in order to satisfy the database size limit of 10 Tb as previously mentioned. Using the MLERP methodology, the previous time series size can be expanded by almost 1,000 times even if the final LERP will have a value multiple times larger than 4,000. The process can work when very large occurrences are expected in time series of trillions characters, as long as the time series and LERP size do not exceed the limitation of the database management system in each step. The detailed MLERP process is described in Algorithm 3 after the other required algorithms are introduced.

3.2 The algorithms

The first step in the proposed methodology is the creation of the suffix array. For this purpose, the *OSACLERP* algorithm will be used; this algorithm was introduced in [9].

Optimized Suffix Array Construction with Longest Expected Repeated Pattern Algorithm (OSACLERP) The authors of [8] presented a simple algorithm (OSAC) for the construction of the suffix array in an external DBMS. It was optimized to work using a theorem for the Calculation of Suffix Array's Required Storage Space [8], which proves that approximately only 75 % of the full suffix array is needed to be stored in order to analyze a time series for periodicity. This is possible because if a substring of the time series has length greater than half the size of the time series then the specific substring cannot occur again inside the time series and thus there is no need to store the extra information for such cases. In [9], Algorithm 1 (*OSACLERP*) was presented, which stores substrings with size equal to the value of LERP instead of limiting the size of the substring stored in the database to half the size of the time series as allowed by the theorem for the Calculation of Suffix Array's Required Storage Space [8]. The algorithm is presented again as part of the overall MLERP methodology in order to calculate the overall time complexity. OSACLERP takes as input parameters the string of the time series and the LERP value for which we want to construct the suffix array. Then with a simple for-loop and starting from the first character of the time series' string it starts to construct in each loop a substring of length LERP and adds it to the database.

Algorithm 1 Optimized suffix array construction with LERP (OSACLERP)

Input: string X of time series, LERP value
Output: an array of all suffix strings or nothing in case of direct insertion into the database

```
1       OSACLERP (string X, int LERP)
2.1     for i := 0; i < X.length; i++
2.2.1       if (LERP + i > X.length)
2.2.2           LERP := X.length-i
2.2.3       end if
2.3         subString := X.Substring(i, LERP)
2.4         insert substring into database
2.5     end for
3       end OSACLERP
```

The overall complexity of the *OSACLERP* algorithm is $O(n)$ since it has only one for-loop that browses the whole string of the time series of length $|T| = n$. However, a major part of the overall process described in the algorithm is not only the insertion of the substring into the database but also the sorting of the substrings. Experimental results have shown that database management systems can achieve the sorting with time complexity close to linear. However the well-known Merge-Sort algorithm has complexity of type $O(n * \log n)$. Therefore, we can assume that the overall complexity of the algorithm could be $O(n + n)$ or $O(n)$ as the experimental results show, or in the worst case it is of type $O(n + n * \log n)$ or $O(n * \log n)$ using the merge-sort algorithm.

ARPaD with shorter pattern length algorithm (ARPaD-SPL) In [1], the Calculate Occurrences' Vectors (*COV*) algorithm was introduced for finding the positions of the substrings in the time series. With the introduction of the Shorter Pattern Length (SPL) parameter, we can limit the results that the algorithm returns because if a substring that satisfies the conditions is found it has to be longer or equal to the SPL, in order to proceed and retrieve all positions at which the substring exists. Using this method, we construct a variation of the COV algorithm, which from now on we will name it All Repeated Patterns Detection Algorithm (*ARPaD-SPL*), where very short patterns with length 1 or 2 can be excluded from the results if they are considered insignificant. Moreover, this process can accelerate the algorithm because the retrieval of all positions, especially for short patterns is time consuming. For example, in a time series of length 10 million characters constructed from an alphabet of ten letters, the approximate occurrence of each letter could be one million times. So, the algorithm will have to retrieve all the ten million positions (aggregated) of the ten single character patterns. Those patterns, in most of the cases, are insignificant for further analysis and, therefore, the algorithm can run faster by avoiding detecting them.

ARPaD with longest expected repeated pattern algorithm (ARPaD-LERP) After the introduction of the SPL in the *ARPaD* algorithm, the concept of Longest Expected Repeated Pattern (LERP) is introduced. It works similarly with the SPL but in a reversed way since it eliminates the retrieval of substrings that are longer than LERP. While the SPL can be generally used to search and find all patterns that occur at least twice and have length equal to or longer than SPL, the *ARPaD-LERP* algorithm allows the retrieval of patterns within a specific length limits by using both SPL and LERP parameters as lower and upper bounds for the search and retrieval criteria.

The *ARPaD-LERP* algorithm can be described as follows:

1) For all the letters of the alphabet, count suffix strings that start with the specific letter.
2) If no suffix strings are 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 letters 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 and the length of the previous substring is equal to or larger than SPL and smaller than LERP, consider the previous substring as an occurrence, find previous substrings' positions and proceed with the next letter of the alphabet.
   b) If the same number of substrings is found as previously and the length of the previous substring is smaller than LERP 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 and the length of the previous substring is different than LERP, consider the previous substring as a new occurrence. If the previous substring has not been calculated again and the length of the substring is equal to or longer than SPL then calculate substrings' positions. Continue the process from step 4.

Algorithm 2 All repeated patterns detection with shorter pattern length and longest expected repeated pattern

Input: string of pattern we want to check, a counter of string length, SPL length, LERP length
Output: a list of all occurrence vectors

```
1        ARPaD_LERP (string X, int count, int SPL, int LERP)
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) AND (X.length < LERP)
3.4.2          ARPaD_LERP (newX, newCount, SPL, LERP)
3.4.3       end if
3.5.1       if (count > 1) AND (X.length = LERP) AND (isXcalculated = false)
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 (isXcalculated = false) AND (X NOT null) AND (X.length >= SPL)
3.6.2          find positions of string X
3.6.3          isXcalculated := true
3.6.4       end if
3.7.1       if (newCount > 1) AND (newCount < count) AND (X.length <> LERP)
3.7.2.1        if (isXcalculated = false) AND (X NOT null) AND (X.length >= SPL)
3.7.2.2           find positions of string X
3.7.2.3           isXcalculated := true
3.7.2.4        end if
3.7.3          ARPaD_LERP (newX, newCount, SPL, LERP)
3.7.4       end if
3.8      end for
4        end ARPaD_LERP
```

Moving longest expected repeated pattern algorithm (MLERP) After modifying the algorithms previously introduced in [1, 8, 9, 32], a new algorithm that depicts the process previously described has been constructed. The algorithm first creates the suffix array with the use of *OSACLERP* for a specified LERP. Then the *ARPaD-LERP* algorithm is called to calculate all the new occurrences and then *MLERP* adds them in the list of results. If there are new occurrences then the algorithm continues and assigns to SPL the value of LERP and doubles the value of LERP. Of course, different values can be used for the second phase, depending on the results of the first e.g., if the patterns found with length equal to LERP are not half the initial suffix strings but one tenth of them then we can assign as second LERP value ten times the initial. This will guarantee us that we have occupied total storage space exactly as the initial in the first phase. Then it creates a new suffix array from each substring that was found in the previous step with length LERP. In this case, it uses length for the new substrings the doubled value of the previous LERP. The process repeats until no new occurrences are found. The complexity of the *MLERP* algorithm is completely based on the complexity of

Algorithm 3 Moving longest expected repeated pattern

Input: string of pattern we want to check, MLERP length
Output: a list of all occurrence vectors

```
1        MLERP(int MLERP)
2        OSACLERP(TimeSeries, MLERP)
3        SPL := 1
4        isCompleted := false
5.1      while (isCompleted = false)
5.2         list Temp := ARPaD_LERP("",0,SPL,LERP)
5.3         list Output += Temp
5.4.1       if (Temp.Count = 0)
5.4.2          isCompleted := true
5.4.3       else
5.4.4          Clear Suffix Array Table
5.4.5          SPL := LERP
5.4.6          LELP := 2 * LERP
5.4.7.1        for each element in list Temp
5.4.7.2           insert substring(Temp<Element>.Position, LERP) into database
5.4.7.3        end for
5.4.8       end if
5.5      end while
6        end MLERP
```

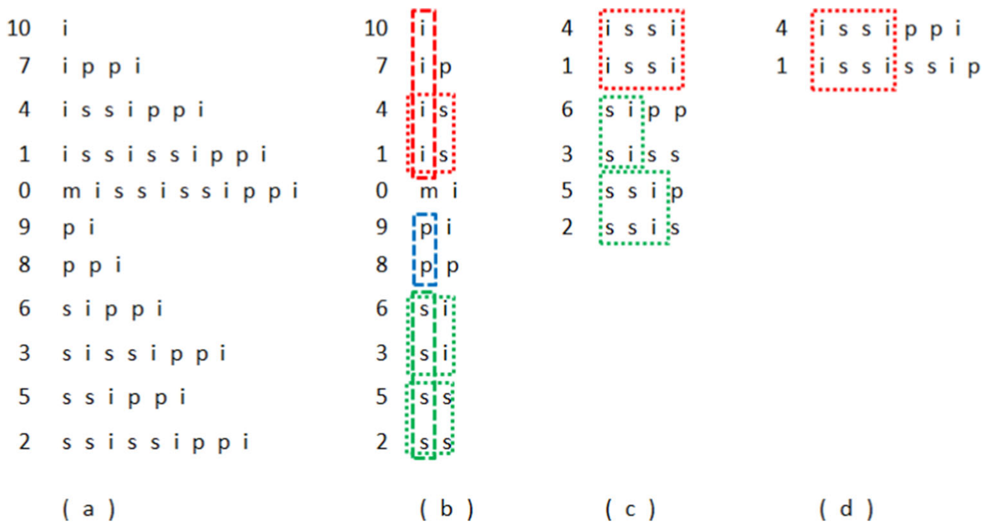
all previously mentioned algorithms, and it is difficult to be theoretically calculated. *MLERP's* complexity is calculated as the sum of the complexities of all the algorithms used in the process.

Example 1 Let us illustrate with an example how the MLERP process works. Suppose that we have for analysis the string *mississippi* and the initial value of LERP is 2. The full lexicographically sorted suffix array as it would be created with the standard process can be found in Fig. 1a and the number to the left of each suffix string denotes the position of the suffix string in the time series. The *MLERP* algorithm will call first the *OSACLERP* algorithm to create the suffix array with parameters the time series' string (*mississippi*) and MLERP value 2. The *OSACLERP* algorithm will create the lexicographically sorted array of Fig. 1b with the same number of records as the standard method; however, the length of the substrings will be from 1 to 2, instead of from 1 to 11, which is the length of the time series.

After the suffix array creation process, the *MLERP* algorithm (Algorithm 3) will call the *ARPaD-LERP* algorithm (Algorithm 2) in order to search for repeated patterns with occurrence equal to or greater than two. It starts with the first letter of the alphabet, i. It founds four substrings that start with the specific letter. It continues to construct a longer substring with again the first letter of the alphabet. The pattern *ii* does not exist so it continues with the next letter m and constructs *im*, which also does not exist. It continues with the letter p and finds one pattern i.e., *ip*. Since pattern *ip* occurs fewer times than the original starting pattern i, then i is definitely a significant occurrence. The certainty comes from the fact that after the appearance of *ip* with fewer occurrences than i, there is no way that a pattern starting with i will occur as many times as i, thus to overcome i as an important occurrence. However, *ip* is not important since it occurs only once and, therefore, does not meet the criteria (of appearing at least twice) to be considered an occurrence. Then the algorithm continues with the next letter, s. It finds two occurrences of the substring *is*. Since *is* has been found twice that might hide a longer pattern with significant occurrence. However, the algorithm cannot decide yet, so substring *is* will be saved for further investigation in the second loop of the *MLERP* algorithm. *ARPaD-LERP* continues with the letter m and finds only one pattern, so it will be discarded for not meeting the occurrence's criteria. The next letter is p. Since p occurs twice and *pi* only once then p is an important occurrence (as we have with i). Substrings *pi* and *pp* are not occurrences since they occur only once. The last letter in the alphabet is the s. There are four patterns starting with s. However, since there are two *si* and two *ss*, then the s is definitely an important occurrence (same as i and p). Moreover, *si* and *ss* have the size of LERP, so the algorithm cannot decide in this loop if there are important occurrences or hidden longer occurrences. They will also be saved for further examination in the second loop of the *MLERP* algorithm. So far the process has discovered as occurrences patterns i, p and s. The next step of the algorithm is to clear the suffix array, assign the value of LERP to SPL (=2), double the size of LERP (=4) and create a new suffix array for the substrings that have been saved (Fig. 1c).

The *ARPaD-LERP* algorithm starts again the process but now instead of searching from the beginning of the substrings it searches from position 2 (SPL) (Fig. 1c). For substring *is*, *ARPaD-LERP* finds first *iss* but because it has the same occurrences as *is* (=2), substring *is* is definitely not an important occurrence (since they have the same number of occurrences). The algorithm continues and finds that there is a pattern with length exactly the same as the size of LERP, i.e., substring *issi*, and the same occurrences as substring *iss*. Thus, *iss* is definitely not an important occurrence (as with *is*) and since the length of *issi* is equal to the length

Fig. 1 The lexicographically sorted suffix array of string *mississippi* and the MLERP process



of LERP, *ARPaD-LERP* cannot decide if the specific string is an important occurrence or hides a longer one. So, it will be saved for further examination in the next loop. For the previously discovered pattern *si*, *ARPaD-LERP* finds that there is only one string with length greater than 2, i.e., *sip*, so string *si* is an important occurrence and the process stops there for the specific pattern. Continuing with *ss*, the algorithm finds the pattern *ssi*, which occurs as many times as the *ss* (=2), so, *ss* is not an important occurrence and the algorithm has to examine the longer string *ssi*. It finds that there is only one pattern with length greater than 3, i.e., *ssip*, so, string *ssi* is also an important occurrence. *ARPaD-LERP* has finished again and has found the new occurrences of *si* and *ssi*.

Since there is one pattern with length equal to LERP, i.e., *issi*, the *MLERP* algorithm continues the process to analyze the specific substring (Fig. 1d). In the new loop the new values of SPL is 4 and LERP is 8. The algorithm finds that there is one pattern with length larger than the length of *issi*, i.e., *issip*, and therefore, *issi* is an important occurrence. The *MLERP* algorithm has finished the analysis of the whole time series and it terminates since there are no saved values for further processing.

At the end of the MLERP process the following occurrences have been discovered with the related positions: *i*(1,4,7,10), *p*(8,9), *s*(2,3,5,6), *si*(3,6), *ssi*(2,5) and *issi*(1,4). The results are exactly the same as in the example presented in [8] with the same time series where the whole suffix array was processed. The difference is that with the methodology introduced in this paper instead of creating from the beginning a large suffix array of all suffixes as in Fig. 1a and of size 66 bytes, with continuous loops the MLERP algorithm created smaller suffix arrays of 21 bytes, 24 bytes and 15 bytes as shown in Fig. 1b, c and d, respectively. This seems to be more time consuming but saves storage space in order to examine larger time series than with the standard procedure presented in [8].

3.3 Algorithm correctness

Although the *COV* algorithm has been thoroughly presented and proved to be correct [1, 8, 32] we will give a strict mathematical proof showing that its variation *ARPaD* Algorithm is also correct. First we will show that the algorithm terminates and then we will use *reduction and absurdum* to prove its correctness.

Proof Termination: First of all the algorithm is finite since the alphabet is finite and it works in a recursive way increasing each time the length of the pattern to detect by one for each one of the alphabet's letters. If the length of the pattern under examination reaches the value of LERP then *ARPaD* Algorithm will terminate at line 3.4.1 or 3.5.1 or will clear the value of LERP and continue from the SPL value with the next alphabet letter (line 3.2) until it will consume all the alphabet (for-loop at line 3.1).

Correction: Let's suppose that the algorithm fails to identify all repeated patterns that have a specific prefix in the suffix strings array. In this case we can identify three different occasions:

1) The length of the pattern is less than SPL. In this case the *ARPaD* Algorithm in lines 3.6.- and 3.7.2.- bypasses the detection of the pattern, which is correct since we have specifically asked to detect patterns that have length larger than or equal to SPL.
2) The length of the pattern is larger than LERP. In this case *ARPaD* Algorithm in lines 3.4.- and 3.5.- stops the detection of the pattern, which is correct since we have specifically asked to detect patterns that have length less than or equal to LERP.
3) The length of the pattern is equal to or larger than SPL and less than or equal to LERP. Since the algorithm failed to detect the repeated patterns, therefore, there is at least one repeated pattern that exists with length k, where $SPL \leq k \leq LERP$ and, therefore, there are at least two suffix strings with the same starting substring that the algorithm failed to identify them and have same length k, where $SPL \leq k \leq LERP$. Therefore, the algorithm in order to fail to detect them stops at position $k-1$. This means that the algorithm has checked the $k-2$ length patterns and found them to be equal and stopped at $k-1$ length because it checked them and found that they were not the equal. However, this is a contradiction because the $k-1$ length substrings are equal and the algorithm cannot stop there and since it has already checked the $k-1$ length patterns (since it stopped there) in lines 3.5.- and 3.6.-, therefore, it will continue in position k and it will detect the repeated patterns of length k. Therefore, a repeated pattern that the algorithm will fail to detect cannot exist and the algorithm has been proven to be correct.

□

3.4 Algorithms' analysis

In [1, 8], the overall theoretical worst case complexity of the *COV* algorithm was very difficult to calculate because of the recursion in the algorithm. Based on experimental observations, it was estimated to be of type $O(10 * n * m * 16 * \log n)$ or generally $(n * \log n)$ [8]. With the use of SPL, the overall worst case complexity of *ARPaD* Algorithm is of type $O(6 * n * m * 9 \log n)$, or generally $O(n * \log n)$. However, so far the experimental findings reported in this paper have shown a time complexity for the average case

Table 1 LERP process time analysis

Time series size	LERP	Max pattern length	Missed occurrences	Create S.A. time (sec)	Occurrence analysis time (sec)	LERP total process time (sec)
125,000	500	79	0	136.60	1,982.83	2,119.43
250,000	500	79	0	286.46	4,535.71	4,822.16
500,000	500	500	49	646.50	12,839.02	13,485.52
1,000,000	500	500	49	1,395.75	28,620.41	30,016.16
2,000,000	500	500	49	2,561.56	71,495.62	74,057.18

scenario of type $O(6 * n * m * 2 * \log m)$ which is almost linear because it can be simplified as $O(n)$, where n is the length of the time series and m is the length of the alphabet which it is considerably smaller than n $(m \ll n)$ and it can be considered a constant. As in the COV algorithm, the theoretical complexity of *ARPaD-LERP* is equally very difficult to be calculated due to the recursion. Based on experimental observations produced in this paper, the complexity is of the same type as of the COV algorithm $O(10 * n * m * 16 * \log n)$ [8] or generally $O(n * \log n)$. With the use of SPL and LERP, however, the worst case complexity is of type $O(4 * n * m * 6 * \log n)$ or generally $O(n * \log n)$. So far the experimental findings in this paper have shown a time complexity for the average case scenario of type $O(4 * n * m * 2 * \log m)$ which is almost linear because it can be simplified as $O(n)$, where m is the length of the alphabet and n is the length of the time series with $m \ll n$.

The overall worst case complexity of the process is very difficult to calculate because of the recursion in the *ARPaD-LERP* algorithm and the fact that there is no indication to how many steps the *MLERP* algorithm will require (but definitely finite steps of order $\log(n-1)$ at the worst case). In general, the complexity of the whole process can be estimated first for the *OSACLERP* algorithm as $O(n + n * \log n)$ if Merge-Sort is used or $O(n)$ if the inherent sorting algorithm of the database management system will be used which seems almost to be linear complexity. Then the *MLERP* algorithm has a while-loop that calls each time the COVLERP algorithm. Although the whole process is definitely finite and can be finished in very few steps ($\log n-1$), it depends on the selection of the MLERP value and the size of the time series. In general the *MLERP* has complexity $O(6 * n * m * 2 * \log m)$ or $O(n)$ generally since $m \ll n$. However, each time the while-loop ends the new n value (in the complexity) is significantly smaller than the original of the time series because it represents the number of occurrences found with length equal to MLERP. Based on experimental results presented in the following section it can be observed that the overall process is of type $O(n)$ in the average case or $O(n \log n)$.

4 Experiments with DNA data

In order to test the effectiveness and reliability of the MLERP process, experiments with DNA data have been conducted. Samples of sizes extending from 125,000 up to 2,000,000 characters long have been used in the experiments. The utilized data represents human chromosome 9. The experimental process covers both the LERP and the MLERP methodologies in order to conclude with comparable analysis data. Moreover, we have used both methods to compare the results and have also an experimental proof of the MLERP process by comparing one to one the results of the two methods, something which have reported that the two sets of repeated patterns from the two methods to be exactly the same. Each time series has been analyzed twice; once for each method. For each process the time has been measured based on a standard personal computer with a double core processor, 4 GB of RAM, 80 GB hard disk

Table 2 MLERP process time analysis

Time series size	Started MLERP	Finished MLERP	MLERP loops	Create S.A. time (sec)	Calculate occurrences time (sec)	MLERP total process time (sec)
125,000	5	80	5	232.25	2,970.13	3,202.38
250,000	5	80	5	536.93	7,256.56	7,793.49
500,000	5	640	8	1,205.88	14,789.21	15,995.09
1,000,000	5	640	8	3,057.06	33,827.98	36,885.04
2,000,000	5	640	8	8,199.18	88,808.80	97,007.98

Table 3 LERP & MLERP required space capacity comparison

Time series size	LERP size	LERP S.A. approximate size (MB)	MLERP initial value	MLERP S.A. approximate peak size (MB)
125,000	500	125.00	5	2.50
250,000	500	250.00	5	5.00
500,000	500	500.00	5	11.78
1,000,000	500	1,000.00	5	30.54
2,000,000	500	2,000.00	5	71.81

and a 32 bit operating system. The results regarding time complexity can be found in Table 1 for the LERP methodology (single *ARPaD* Algorithm execution) and in Table 2 for MLERP methodology (repeated *ARPaD* Algorithm execution). In Table 3, the comparison of the required space capacity between LERP and MLERP has been described. The LERP section of Table 3 describes the approximate required space capacity based on the LERP value, while the MLERP section of Table 3 describes the approximate maximum required space capacity (in the worst, regarding space consumption, loop of the algorithm).

The first thing that has to be mentioned regarding the LERP process is that the experimental results are not directly comparable for each experiment. Although for all the five experiments a relatively large LERP value (=500) has been used, only in the first two experiments it was long enough to discover all occurrences in the time series, while in the next three experiments it was insufficient, since 49 occurrences have been left outside the results since they have been found with length equal to the LERP value. This means in the last three experiments there are 49 occurrences (each one with more than two occurrences) that the method did not manage to further examine and determine if they hide behind them longer patterns with significant occurrences for periodicity.

Regarding time consumption, the creation time of the suffix array for each experiment, including sorting in the database management system, is almost linear based on the experimental findings. Moreover, the occurrence analysis seems to be also linear, yet, not directly, since each time the length of the time series doubles, the analysis needs approximately more time by a factor of 2.3 to 2.5 instead of just 2. This happens because DNA sequences are not random time series as we can see from their occurrences per pattern length and cumulative occurrences graphs shown in Figs. 2, 3, 4, 5, 6, 7 and 8 and their location and dispersion parameters (Table 4). Patterns that have been found are not distributed evenly since most of them have small length as it can be seen from the occurrences per pattern length graphs presented in Figs. 4–8. It can be observed from the location and dispersion parameters (Table 4) that as the length of the DNA sequence doubles in each experiment, the third quartile (75 % of the observations) and the mean change disproportionally.

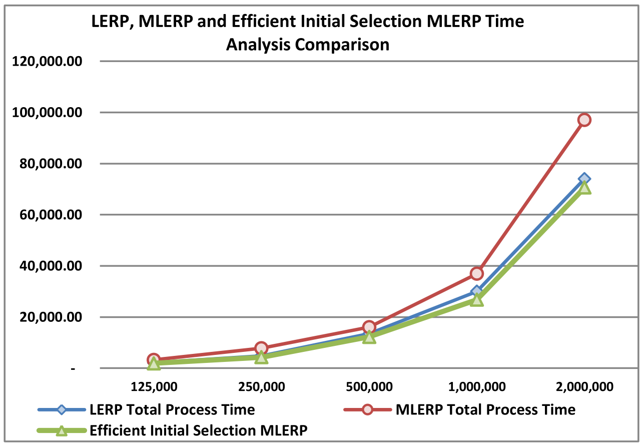


Fig. 2 Time analysis comparison for LERP, MLERP and efficient initial selection MLERP (in seconds)

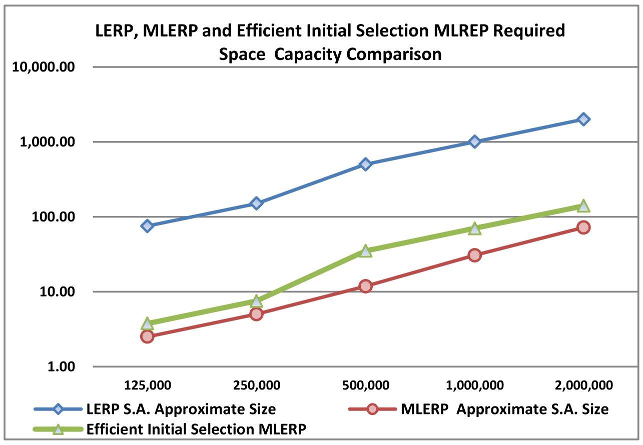


Fig. 3 Required space capacity comparison for LERP, MLERP and efficient initial selection MLERP (in MegaBytes, logarithmic scale)

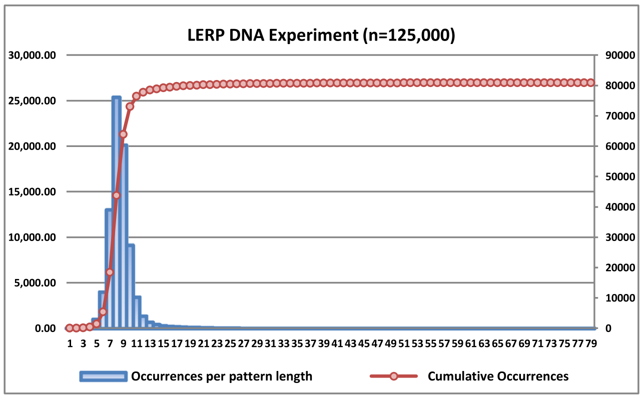


Fig. 4 Occurrences per pattern length and cumulative occurrences for LERP DNA experiment with $n = 125{,}000$

From the first experiment with DNA of length 125,000 characters, 95 % of the found patterns have length less than 12 as it can be seen in Fig. 4. When the length is increased to 2,000,000 characters (15 times more than that in the previous example), 95 % of the found patterns have length less than 25 (twice the length of the patterns found in the previous example) as it can be seen in Fig. 8. That means that nucleotides are distributed with a predefined way in the DNA sequence, something that requires the *ARPaD-LERP* algorithm to search deeper and eventually be a little bit slower than expected. DNA sequences seem not to be random and this is something expected. If every nucleotide had the same probability to occur at any position, DNA sequences with all nucleotide positions occupied by A, C, G or T could exist. However, such an occasion is unlikely due to the nature of a living organization structure, since the DNA sequence is complicated and its nucleotides' subsequences serve a specific role in the DNA. Moreover, the experiments have shown that the number of the appearances of each nucleotide is not the same in the DNA

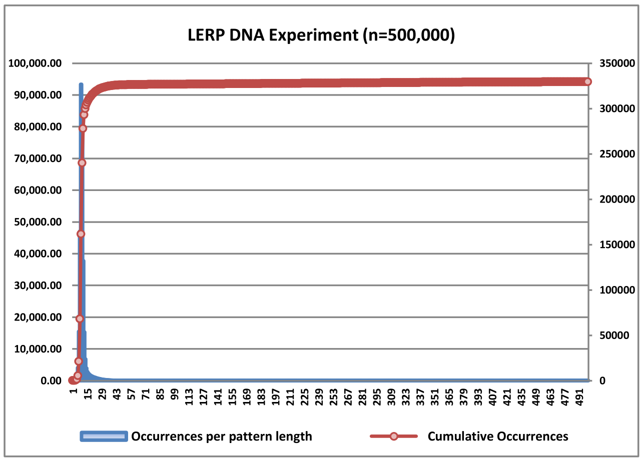


Fig. 6 Occurrences per pattern length and cumulative occurrences for LERP DNA experiment with $n = 500{,}000$

sequence. More specifically, $|S_A| \cong |S_T|$and $|S_C| \cong |S_G|$with $|S_A| \cong 1.8 * |S_C|$, where $|S|$denotes the total number of appearances of a nucleotide (A, C, G or T) in the DNA sequence as it has been found experimentally in chromosome 9.

The MLERP process time analysis shows that time complexity seems to be almost linear, yet, not directly as it can be seen in Fig. 2. That might have occurred because for the two first experiments, in which smaller patterns have occurred, MLERP looped five times while in the next three experiments it looped eight times. However, the most important outcome in comparison to the LERP results is the fact that all occurrences have been discovered by the MLERP process. While MLREP is definitely slower than LERP, as the results have shown, one great advantage over LERP is that MLERP manages to discover all occurrences, despite the initial MLERP value. LERP failed for 49 occurrences

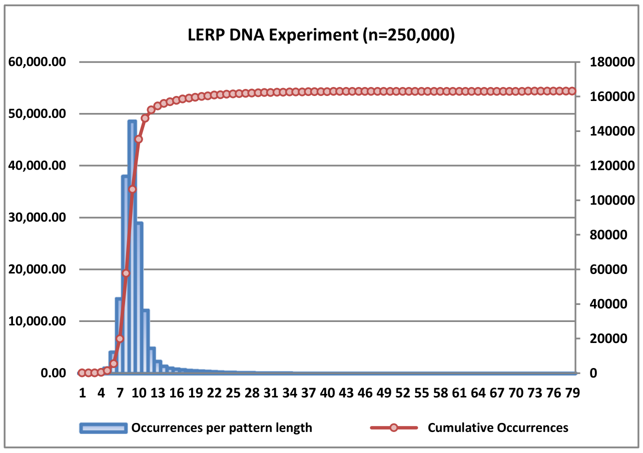


Fig. 5 Occurrences per pattern length and cumulative occurrences for LERP DNA experiment with $n = 250{,}000$

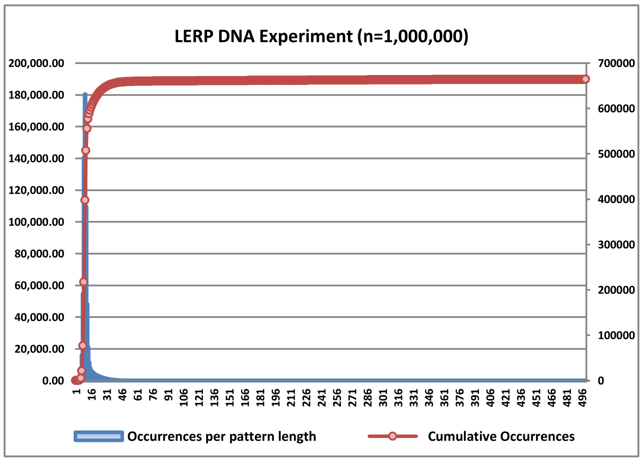


Fig. 7 Occurrences per pattern length and cumulative occurrences for LERP DNA experiment with $n = 1{,}000{,}000$

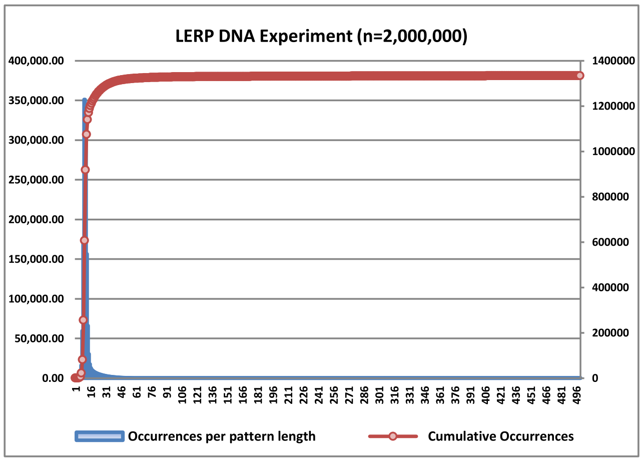


Fig. 8 Occurrences per pattern length and cumulative occurrences for LERP DNA experiment with $n = 2{,}000{,}000$

to do that in the last three experiments (Table 1). Moreover, MLERP's time complexity could have been better if a different initial value would have been chosen.

In Table 5, different initial values for MLERP have been used to present how the MLERP process can be more efficient in time, based on the selection of the initial MLERP. For the first two experiments, the value for MLERP has been chosen to be 30 and for the next three it has been set to 70. In Table 6, there is a comparison between the time needed for the analysis by using as initial MLERP value (=5) and the new MLERP initial values previously selected (30 and 70). It can be seen in Fig. 2 that the MLERP process is faster when the appropriate initial value is selected. It is approximately 30% faster than the experiments using MLERP with initial value 5 because as it can be observed from Table 6 it needs almost half the loops (3 instead of 5 for experiments 1 & 2 and 4 instead of 8 for experiments 3, 4 & 5). However, comparing the space need in the experiment using MLERP with initial value 5 and initial values 30 or 70, significantly more space will be needed for the initial suffix array storage (Fig. 3). What has been earned from the one side has been lost from the other. Furthermore, the algorithm is almost 10 % faster than the LERP process since it has to create and sort a considerably smaller table than LERP (string length 30 instead of 500). The initial MLERP value choice is crucial based on the space/time analogy that has to be met. Faster analysis means more space consumption while less space consumption means slower analysis.

From the required space capacity comparison in Table 3, it is clear how the MLERP process prevails over LERP (Fig. 3), since the maximum storage capacity needed for the MLERP process is significantly smaller 20 or 30 times. By maximum MLERP storage capacity we define the space needed for the largest suffix array in one of the MLERP loops. Obviously this had happened in the first loops since then it is when the process creates the suffix arrays with almost all suffix strings despite the fact of the small length of the suffix strings. As the process continues searching deeper for longer occurrences, the size of the suffix arrays reduces dramatically.

Some very important aspect regarding the *ARPaD* Algorithm for both single execution or with the MLERP process is that it is the only algorithm that can detect all repeated patterns in a sequence. The fact that *ARPaD* uses a recursive approach to detect all repeated patterns without any prior knowledge of the patterns and their characteristics gives it an extreme advantage over any other pattern matching algorithm or method that can be used to detect repeated patterns by applying brute force techniques. We will give a small example that will prove our assertion. In the first experiment with DNA sequence length just 125,000 characters, we have found repeated patterns of size 79 (see Table 1). If we want to use any kind of pattern matching algorithm like the ones published in [2–5, 7, 27], we have to search for all combinations of length 79 we can construct with the 4 nucleotides A, C, G and T. However, this means that we have to check $4^{79} = 2^{158}$ different patterns. Assuming that

Table 4 Location and dispersion parameters for LERP DNA experiments

Time series length	Q$_1$	Q$_2$	Q$_3$	Max	Mean	Std. Dev.	Skewness	Kyrtosis
125,000	8	8	9	79	8.77	2.95	7.39	97.00
250,000	8	9	10	79	9.54	3.34	5.29	45.97
500,000	9	10	11	548	12.55	25.12	12.60	175.80
1,000,000	9	10	11	548	12.81	19.82	14.73	252.93
2,000,000	10	11	12	548	13.26	15.97	15.38	313.24

Table 5 Efficient initial selection MLERP process time analysis

Time series size	Started MLERP	Finished MLERP	MLERP loops	Create S.A. time (sec)	Calculate occurrences time (sec)	MLERP total process time (sec)
125,000	30	120	3	105.58	1,760.51	1,866.09
250,000	30	120	3	209.75	4,036.87	4,246.62
500,000	70	560	4	420.08	11,792.71	12,212.79
1,000,000	70	560	4	884.00	25,886.37	26,770.37
2,000,000	70	560	4	1,873.21	72,504.92	70,631.71

we are very lucky and we found the patterns in the first few steps of our analysis, even then we cannot stop because none of the pattern matching algorithm can know ahead how many patterns of a specific length exist and repeat at least twice (now we know it because of our algorithm and the analysis already performed). Therefore, we have to continue and perform the full analysis. Some of the best pattern matching algorithms, e.g., [2–5, 7, 27] can find a pattern in 10 to 300 milliseconds (however, more time is needed to find all occurrences). In order to simplify our calculations let us assume that they need only 1 nanosecond or 10^{-9} s to find all occurrences of a repeated pattern using some of the best pattern matching algorithms, assuming that they can perform millions of times better than what they actually could achieve. Therefore, the total time needed for those algorithms to detect all repeated patterns of size only 79 is: $2^{158} \times 10^{-9} sec \cong 2^{158} \times \left(3 \times 10^{-17}\right) years \cong 2^{116} years$. Such time is equivalent to almost 6 $\times 10^{24}$ times the age of the Universe in Earth years, if we assume that the age of the Universe is 13.5 billion Earth years. However, our algorithm (even using a personal computer with commonly used hardware with normal performance) for the single step execution needs 2,119 s or almost 36 min while with the MLERP process it needs 3,202 s or almost 53 min to detect all repeated patterns and not just those of a specific length.

5 Conclusion and future work

This paper has introduced a new methodology for the reduction of storage space required for a suffix array while calculating all repeated patterns of a time series. This methodology expands the LERP methodology [9] and allows for the analysis of even larger time series when the required space capacity needed for the suffix array is forbidden due to system, hardware and software limitations. The proposed method can reduce the required space capacity up to hundreds of times depending on the initial values selected for MLERP. The time complexity of MLERP, although almost linear, is more time consuming than the LERP process. Despite this drawback, the MLEPR method allows for the analysis of larger time series. For example, in the case of DNA analysis, the experimental analysis conducted in the context of this paper has shown (Table 4) that usually for very long time series of size one million characters it is not expected to have more than one occurrence with length larger than 500. This means that an initial small LERP value for the LERP process can be selected, e.g., 30 for a DNA sequence of size one million, in order to discover at least 95 % of all occurrences as we can see from the cumulative occurrences diagram (Fig. 8) even without using the MLERP process. However, if the human DNA sequence

Table 6 LERP, MLERP and efficient initial selection MLERP comparison

Time series size	LERP		MLERP		Efficient initial selection MLERP	
	Loops	Total process time (sec)	Loops	Total process time (sec)	Loops	Total process time (sec)
125,000	1	2,119.43	5	3,202.38	3	1,866.09
250,000	1	4,822.16	5	7,793.49	3	4,246.62
500,000	1	13,485.52	8	15,995.09	4	12,212.79
1,000,000	1	30,016.16	8	36,885.04	4	26,770.37
2,000,000	1	74,057.18	8	97,007.98	4	70,631.71

should be analyzed as a whole, this would be impossible with the standard process of the suffix array creation or even with the use of the LERP. The human DNA has 3.5 billion elements. The size of the full suffix array would be approximately 10^{19} or 10 million Terabytes, something which is not supported by common computing systems. Even with the LERP methodology the biggest value that can be used is approximately 50, in order to meet database management possible limitations as described. However, with the use of the MLERP process and 50 as an initial value for the LERP, the size of the suffix array will be approximately 200 GB in the first loop, which is feasible to store in an ordinary computing system and allows the analysis of time series in multiple finite loops.

MLERP is on average 30–50 % slower than standard LERP process because it needs to reconstruct the suffix array for the suffix strings that need further analysis. However, the prior-knowledge of how the pattern's length is distributed can help in significantly improving the required time for analysis. A better choice of the initial value of MLREP can achieve not only the analysis of very large time series by reducing space capacity but also in better actual time length than the standard LERP process. In any case, our *ARPaD* Algorithm with single execution or with the use of MLERP makes the detection of all repeated patterns in a time series a very fast process while no other pattern matching algorithm can do such analysis in feasible time.

Regarding the complexity of the process, the experimental results have shown that it is of type $O(n)$ for average cases, despite the use of a database management system that needs to sort lexicographically the suffix array immediately after its insertion in the system. Improved sorting methods can achieve this in almost linear $O(n)$ time, making the whole process very fast and appropriate for the analysis of large time series like DNA sequences.

As future work, the methodology proposed here could be further enhanced in order to achieve better performance with the use of parallel programming for the parts of the algorithms that can be separated and processed in parallel. This will help in significant reduction of the time consumption needed for the full analysis and all the repeated patterns detection. This will allow the analysis of very large time series in logical time frame, making MLEPR a very powerful process for periodicity data mining and time series analysis. The application domains of MLERP varies from the analysis of DNA chains in bioinformatics, to meteorological data analysis in weather forecasting or power transmission and distribution forecasting over an electrical grid.

References

1. Xylogiannopoulos K, Karampelas P, Alhajj R (2012) Periodicity data mining in time series using suffix arrays. In: Proceedings of the IEEE intelligent systems IS'12
2. Gog S, Moffat A, Culpepper S, Turpin A, Wirth A, 2013 Large-scale pattern search using reduced-space on-disk suffix arrays. arXiv:1303.6481v1
3. Phoophakdee B, Zaki M (2007) Genome-scale disk-based suffix tree indexing. In: Proceeding of the international conference on management of data SIGMOD '07, pp 833–844
4. Phoophakdee B (2007) TRELLIS: genome-scale disk-based suffix tree indexing algorithm, PhD Thesis, Department of Computer Science. Rensselaer Polytechnic Institute, Troy
5. Sinha R, Moffat A, Puglisi S, Turpin A (2008) Improving suffix array locality for fast pattern matching on disk. In: Proceedings of the international conference on management of data SIGMOD '08, pp 661–672
6. Barsky M, Stege U, Thomo A (2011) Suffix trees for inputs larger than main memory. Inform Syst 36(3):644–654
7. Wu Y, Wang L, Ren J, Ding W, Wu X (2014) Mining sequential patterns with periodic wildcard gaps. Applied intelligence
8. Xylogiannopoulos K, Karampelas P, Alhajj R (2012), Exhaustive patterns detection in time series using suffix arrays. Manuscript in submission
9. Xylogiannopoulos K, Karampelas P, Alhajj R (2012) Minimization of suffix array's storage capacity for periodicity detection in time series. In: Proceedings of the IEEE international conference in tools with artificial intelligence
10. Xylogiannopoulos K, Karampelas P, Alhajj R (2013) Probabilistic existence and estimation of longest expected repeated pattern in sequences. Submitted for publication
11. Xylogiannopoulos K, Karampelas P, Alhajj R (2014) Experimental analysis on the normality of pi, e, phi and square root of 2 using advanced data mining techniques. Experimental mathematics, in press
12. Rasheed F, Alshalfa M, Alhajj R (2010) Efficient periodicity mining in time series databases using suffix trees. IEEE Trans Knowl Data Eng 22(20):1–16
13. Schürmann K-B, Stoye J (2005) An incomplex algorithm for fast suffix array construction. In: Proceedings of the 7th workshop on algorithm engineering and experiments and the 2nd workshop on analytic algorithmics and combinatorics (ALENEX/ANALCO 2005), pp 77–85
14. Crauser A, Ferragina P (2002) A theoritical and experimental study on the construction of suffix arrays in external memory. Algorithmica 32(1):1–35
15. Ko P, Aluru S (2005) Space efficient linear time construction of suffix arrays. J Discrete Algorithm 3(2–4):143–156
16. Manber U, Myers G (1990) Suffix arrays: a new method for on-line string searches. In: Proceedings of the first annual ACM-SIAM symposium on discrete algorithms, pp 319–327
17. Weiner P (1973) Linear pattern matching algorithms. In: Proceedings of the 14th annual symposium on switching and automata theory, pp 1–11

18. Chen Y-S, Cheng C-H, Tsai W-L (2014) Modeling fitting-function-based fuzzy time series patterns for evolving stock index forecasting. Applied intelligence
19. Bao D (2008) A generalized model for financial time series representation and prediction. Appl Intell 29(1):1–11
20. Elfeky MG, Aref WG, Elmagarmid AK (2005) Periodicity detection in time series databases. IEEE Trans Knowl Data Eng 17(7):875–887
21. Rasheed F, Alhajj R (2008) Using suffix trees for periodicity detection in time series databases. In: Proceedings of the IEEE international conference on intelligent systems
22. Rasheed F, Alhajj R (2010) STNR: a suffix tree based noise resilient algorithm for periodicity detection in time series databases. Appl Intell 32(3):267–278
23. Cheung C-F, Yu JX, Lu H (2005) Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Trans Knowl Data Eng 17(1):90–105
24. Gusfield D (1997) Algorithms on strings, trees, and sequences. Cambridge Univesity Press, Cambridge
25. Creight EMM (1976) A space-economical suffix tree construction algorithm. J ACM 23(2):262–272
26. Ukkonen E (1995) Online construction of suffix trees. Algorithmica 14(3):249–260
27. Orlandi A, Venturini R (2011) Space-efficient substring occurrence estimation. In: Proceedings of the 30th principles of database systems PODS, pp 95–106
28. Dementiev R, Karkkainen J, Mehnert J, Sanders P (2008) Better external memory suffix array construction. J Exp Algorithmics 12(3.4):24
29. Kim DK, Sim JS, Park H, Park K (2003) Linear-time construction of suffix arrays (Extended Abstract). In: Baeza-Yates R, Chávez E, Crochemore M (eds) Combinatorial pattern matching, pp 186–199
30. Wong SS, Sung WK, Wong L (2007) CPS-tree: a compact partitioned suffix tree for disk-based indexing on large genome sequences. In: Proceedings of the IEEE 2007 international conference on data engineering, pp 1350–1354
31. Han J, Yin Y, Dong G (1999) Efficient mining of partial periodic patterns in time series database. In: Proceedings of the 15th IEEE international conference on data engineering:106
32. Xylogiannopoulos K, Karampelas P, Alhajj R (2012) Pattern detection and analysis in financial time series using suffix arrays. In: Doumpos M, Zopounidis C, Pardalos PM (eds) Financial decision making using computational intelligence. Springer, pp 123–152