AIMS Medical Science

AIMS Medical Science, 4 (4): 426–440
DOI: 10.3934/medsci.2017.4.426
Received 29 April 2017,
Accepted 18 October 2017,
Published 24 October 2017

http://www.aimspress.com/journal/medicalScience

Research article

Space Efficient Data Structures for N-gram Retrieval

Fotios Kounelis and Christos Makris *

Department of Computer Engineering and Informatics, University of Patras, 26500 Greece

*** Correspondence:** Email: makri@ceid.upatras.gr; kounelis@ceid.upatras.gr; Tel: 30-2610-996-968.

Abstract: A significant problem in computer science is the management of large data strings and a great number of works dealing with the specific problem has been published in the scientific literature. In this article, we use a technique to store efficiently biological sequences, making use of data structures like suffix trees and inverted files and also employing techniques like n-grams, in order to improve previous constructions. In our attempt, we drastically reduce the space needed to store the inverted indexes, by representing the substrings that appear more frequently in a more compact inverted index. Our technique is based on n-gram indexing, providing us the extra advantage of indexing sequences that cannot be separated in words. Moreover, our technique combines classical one level with two-level n-gram inverted file indexing. Our results suggest that the new proposed algorithm can compress the data more efficiently than previous attempts.

Keywords: suffix trees; n-gram indexing; data compression; biological sequences; inverted files; weighted sequences

1. Introduction

Efficient string indexing is the main purpose of a large number of algorithms, like those published in [1–6] which are a small but representative amount of the great work that has been done in the fields of compressed indexing. In our research, we have not only focused on handling general strings but we can also handle weighted sequences. There is a slight difference between these two

kinds of strings. Weighted sequences are strings that in some positions may take more than one of the characters of the alphabet with some probability. More specifically, a weighted string $S = x1x2 \ldots xn$ has in some positions (e.g. x_i) a set of couples. Each one of these couples (c, $p_i(c)$) has a character c of the alphabet and a certain probability $p_i(c)$ that the corresponding character appears at position i, instead of some other character in the alphabet. As understood, the sum of the probabilities of all characters at a specific position i should be equal to one ($\Sigma p\iota(c) = 1$). As a result, from a string with one weighted position, there are $|\Sigma|$ different strings generated, where $|\Sigma|$ is the length of the alphabet Σ, assuming there are no zero probabilities. Generally, each generated string has a certain probability that comes from the probability of each character, appearing in a position. However, each string may have more than one weighted positions and in this case, things are getting trickier.

In the sequel, we need an efficient algorithm to handle both regular strings and weighted sequences. The most common way to do such thing is by using suffix trees [7,8]. However, suffix trees are data structures able to manage only regular sequences; in order to handle weighted strings, usually, an extension of it tailored to handle weighted sequence, called Weighted Suffix Tree and presented in [9] is used, and we also implicitly use it in our construction.

Our approach is mainly based on the attempt of Diamanti et al. [10] where they presented an algorithm for compressing inverted files when used to index strings. In essence, this approach employs a coalescence of inverted files and the n-grams techniques. N-grams are a convenient way to split a sequence into small parts that are easy to handle and store efficiently. Instead of using Weighted Suffix Trees, with the n-gram technique we can also handle weighted sequences. In our algorithm, we combine suffix trees and inverted files. The conjunction of suffix trees and inverted files in this algorithm is not a widely-used way to manage weighted or regular strings but it is an elegant one. As a result, no classical pattern matching algorithms are used in our approach, but it is the data structures and the operations on them, that we develop, in order to manage the strings and store them efficiently. Our attempt mostly focuses on biological sequences and DNA strings, but it can be easily used in natural language strings due to the nature of the data structures.

The main attribute of our algorithm that allows it to be implemented on both kinds of texts is the n-gram technique. This technique is a way to split a string and has a lot of advantages, as mentioned above. Although n-gram indexing is not the only tool to solve such a problem, this is the one that is used due to its advantages. Firstly, and the most important advantage of this kind of indexing, is that it works in all kinds of strings; when n-grams are taken from a text, they do not need to have any practical meaning, which means that the extracted subwords will be stored even if they are not natural language words. Another major advantage of the n-gram approach, resulting from the above is that it can be used in all kind of texts. Strings can be either biological data or natural language text or language texts with ideograms. However, there is always a possibility for a text to be transferred with some errors on some symbols of the sequence. N-grams, depending on the technique that is used to be created a sequence, can avoid such errors. Using one-sliding technique for the indexing

construction we can avoid all errors that may be generated during the building of the index. Like almost any algorithm or indexing technique, n-grams have disadvantages too. The main drawback is that the amount of size that is used to store the text, especially in the one-sliding technique, is very large and the retrieval of the queries is not efficient. Mostly Kim et al. [11], and other researchers have proposed some ways to store n-grams efficiently; our work moves to the same direction.

Our algorithm and the one in Diamanti et al. [10], follow the approach that Kim et al. [11] proposed which is one of the most efficient for n-gram indexing. This technique is an admixture of the advantages of the inverted indexing and the n-grams, avoiding the drawbacks of each one. To exploit these advantages, the two-level n-gram indexing needs substrings of a certain length to be extracted from the text and after that, n-grams are extracted from the substrings. As a result, substrings are extracted from the sequence and are stored in an index, subsequently, the classical n-gram technique is used on these substrings and a new index is constructed. By employing a framework of two inverted indexes instead of one, when the substrings are highly repetitive depending on the number of times it occurs and the length of the string, the data compression that is achieved is very high; that is the main idea behind the approach of Diamanti et al. [10].

Although the compaction the previous algorithm has achieved was high, it has some drawbacks. In this research, we propose a novelty, that with the use of extra formulas and extra developments on existing data structures, we can make a much more efficient data compression. As mentioned above, to better exploit the use of the two-level n-gram technique, the substrings that are stored in this index should be highly repetitive. The modification that we propose, is an efficient way to avoid non-highly repetitive substrings and store them in the classic n-gram level indexing instead of the two-level one. In order to achieve that we employ *a more efficient threshold* in the frequency of appearance of each substring, that does not require any *user based* determined parameters but it is determined by inherent properties of the strings that we have to store. This novelty is not only applicable on regular strings, but we also propose an easy way to achieve such an efficient compaction in weighted sequences too. On both applications, we implement our technique and depict the large improvement in compaction between the previous and the proposed algorithm, in order to validate our proposal's performance.

The organization of the paper is as follows. In this section, we make a brief introduction of the algorithms and the techniques we propose. In section 2, related work that has been made in this area is analyzed. In section 3, we analyze the way substrings are produced and the occurrences of each substring so as to manage it as a highly repetitive or not. In section 4, we give the way to compute the efficient thresholds using formulas for both regular sequences and weighted strings. In section 5, we describe the experiments that have been made using our implementation and the comparison between the data compression that has been achieved by the previous algorithm and our technique. In section 6, we provide the conclusion of our work and how this could improve the existing situation. Also, we mention the future work that can be made in order for our algorithm to have a wide range of usage in more applications.

2. Related Work

Efficient data compression on storing strings on large sequences, weighted or not, is a scientific field that has been the active research area of a lot of researchers. In weighted sequences, a great number of algorithms have been developed to handle pattern matching queries. In Christodoulakis et al. [12], there are presented algorithms that handle pattern matching in weighted sequences by adopting techniques from handling regular strings. In Zhu et al. [13] is provided a framework for retrieval in regular strings that is efficient and effective. This is achieved in real time and not only in a certain sequence but in a whole database with a lot of different data. Another challenge is the approximate matching in the regular or weighted sequences. An approach was presented in Amir et al. [14], that employs Hamming and edit distance techniques with probability estimation in order to perform efficiently approximately pattern matching. Later, three types of research were proposed, the first from Atalabi et al. [15] and the other two from Zhang et al. (a) [16] (b) [17], that are using the Equivalence Class Tree a structure resembling the Weighted Suffix Tree. In particular, the research of Zhang et al. (a) generalizes an approach of Iliopoulos et al. [9] to develop an efficient approximate pattern matching algorithm. Furthermore, in Li et al. [18] an algorithm for approximate pattern matching was proposed. This algorithm uses a lot of similarity and dissimilarity functions producing a general threshold that has a certain tolerance, along with stacks and pruning methods. An approach for pattern matching in large data that are not stored in main memory is made from Mouza et al. [19]. In this algorithm, B-trees are used for pattern matching in secondary memory, together with algebraic signatures and n-gram inverted indexing.

Moreover, stochastic models are used in probabilistic suffix trees. In Marsan and Sagot [20] and Sun et al. [21], this connection model is used to represent compactly the conditional distribution of probabilities for a set of sequences. In this probabilistic tree, the nodes are associated with a vector that stores the probabilities for the distribution of the next symbol to appear.

Weighted strings are mainly used in computational molecular biology for handling the possible occurrence of each element in a weighted point in the string; moreover, Cryptanalysis systems also use weighted sequences and intermediate strings. Intermediate strings work at some point with the same way as the weighted sequences, these strings have at some position the probability to take a symbol from a set of symbols. In Holub and Smyth [22] and Holub et al. [23], there are symbols that can be modeled as a set of letters from a certain alphabet with some probabilities for each one.

The data compression using n-grams, used in this paper, is able to handle multilingual texts in web search engines in information retrieval applications. These applications are used in every language, especially in Chinese, Japanese and Korean, because it cannot be separated into words with meaning. Kim et al. [11] approach, with the two-level n-gram inverted index, provides an excellent way to manage these languages especially with the use of bi-grams (n = 2). A deeper analysis in these languages from Manning et al. [24] mentions that each ideogram is a syllable that means is a set of characters.

3. Substring Algorithms and Data Compression with Inverted Files

3.1. Substring and n-gram identification

In this subsection, we will describe the Kim et al. [11] algorithm for efficient handling regular strings. The way to export substrings from the string and n-gram from the substrings, that are exported, is provided. This approach can be easily adopted for weighted sequences too. Essentially, n-grams and substrings are produced and stored in an index; the main difference with the classical approach is that in this technique there is no more one index but two, are connected. Firstly, to produce the first index that is consisted of the substrings we are working on the text. The main purpose of this stage is to produce all the substrings from the text, avoiding to losing any possible extracted n-grams from each substring. For this reason, all the substrings are extracted from the text where two continuous substrings do not overlap in more than n-1 characters, where n is the length of n-gram. With this extraction technique, we avoid losing some n-grams. Secondly, for these substrings that are extracted, an inverted index is built, which is called *back-end* index and each substring is treated as a word. Then, for each one of the above substrings, the n-grams that consist it, are extracted. Lastly, another inverted index for the n-grams is built, that is called *front-end* index that treats the substrings above as documents and the n-grams as words. This two-level scheme is able to make an impressive reduction when the substrings extracted in the first stage are highly repetitive and this is the point that the technique that is proposed in this paper is much more efficient than the previous one in Diamanti et al. [10].

In regular strings, the way to export the substrings is trivial. We extract the substrings of a certain length from the sequence and calculate the occurrences of each one. If the number of occurrences is over a threshold k then it is more efficient to be stored in the two-level scheme, otherwise, this substring is better to be stored in the classic inverted index. The way the threshold is produced for the efficient characterization of its repeatability is widely discussed and analyzed in the next section.

In weighted strings, the substrings extraction is a bit more complicated because it produces much more substrings than on the regular strings. The way to apply this technique on these strings is by extracting all the possible substrings that can be generated and store them in a suffix tree (thus creating a weighted suffix tree). In more detail, the number of different substrings that can be produced in a weighted string depends on the different weighted points. Assuming that in a substring there is only one weighted point and all the characters that are able to appear have a corresponding probability more than zero, then the different substrings that are produced equal the number of the symbols of the alphabet. Respectively, in a substring with more than one weighted points, all the possible substrings are extracted and defined as separated entries in the back-end index. Then, for each one, the possible n-grams are produced and stored in the certain position of the front-end index.

3.2. Inverted file implementation using suffix trees

Both schemes for n-gram inverted indexing, have their own advantages and drawbacks. The classical approach is based on extracting the n-grams from all the string and then storing them in the inverted index. That is a technique that has some main drawbacks that were analyzed previously. On the other hand, the two-level approach extracts and stores substrings and for each one extracts the n-grams. This technique demands the appearance of highly repetitive substrings in order to be space efficient. By the terms of highly repetitive substrings we mean substrings that are frequent enough so that it is more efficient to use the two-level n-gram indexing than the classic n-gram indexing.

For this reason, the best choice is a hybrid algorithm that can efficiently unify both techniques. In this approach, the algorithm locates all the substrings and stores them in the classical inverted index or in the two-level approach when some frequency of appearance criterion is satisfied. For the technique to be effective the criterion should be based on the number of occurrences of each substring, so the substrings that are repeated more than a threshold are stored in the two-level scheme otherwise they will be stored in the classical index. The way that the threshold is calculated will be thoroughly analyzed in the next section and *is our main contribution*.

Moreover, we propose to avoid the continuous runs of the algorithm as proposed in Diamanti et al. [10], and the experimental results show that this not only time efficient but also provides better results in data compression. The way/formula to determine if a substring is being repeated in the string enough, so as to store it in the two-level scheme is the point that previous algorithm could not handle efficiently but in this paper, we try to present a way in order to achieve best compaction results.

4. Two-level Inverted Index Thresholds

4.1. Threshold for regular strings

In order to determine a suitable way to store the substrings efficiently in the inverted indexes, we introduce some effective triggers. The previous algorithm of Diamanti et al. [10] was handling the inverted indexes to make and effective data compression, but it was careless in the true efficiency that it could possibly achieve by inserting effective thresholds. Their approach was based on a user based trigger number of occurrences. All the substrings that have a number of occurrences more than this trigger, mean that are highly repetitive and are stored inside the two-level scheme. Otherwise, it is stored in the classical n-gram inverted index.

This effort has nested drawbacks that can affect the efficiency of the algorithm. First of all, by using a *user-based* parameter, the main difficulty is that knowledge about the entry data ought to be had by the user. When it comes to sequences on the bioinformatics field (and not only in this field), where the entry data are enormous and there is no linguistic meaning in the text so as to be understood, it is very difficult for the user to define an effective threshold. As a result, the way to

hook the best threshold may need several times to run the program and this is inefficient.

Moreover, besides the time inefficiency for finding the best threshold, setting a trigger for all the substrings is inefficient due to the differences each substring has depending on its length. Let us assume that a substring with length m has a number of occurrences e; it is logical to think that as long as the length m of the substring increases, the number of occurrences is decreasing. This provides us the hint that maybe m and e are inversely proportional amounts. For this reason, setting a global threshold for all the substrings resulting from a sequence is wrong. In large strings, there is a significant difference in the times a substring is repeated, between two consecutive lengths. In more details, the number of occurrences e, for a substring with length m and for a substring with length m+1 may have a great difference. Hence, it is inefficient to categorize the substrings by the number of occurrences when it is global. For making the algorithm as efficient as it could be, we propose a way to determine the best threshold to split the substrings to the two-level scheme and the classical n-gram inverted index.

4.1.1. Length approach

In our first approach, the main purpose is to focus more on the length of a substring than on the number of occurrences. That is why we not only want to have substrings that are highly repeated, but we do care about bigger substrings too. Thinking that in random strings, as the length of the substring increases the number of occurrences decreases, in our technique a limit threshold *lim* for each substring is produced by multiplying the square of its length and the number of occurrences.

$$lim = len^2 * s$$

where *len* is the length of the substring, *s* is the times it occurs in the string and *lim* is the limit of the substring. The mathematic equation above describes the effort to focus on the length of the substring, way more than the time of occurrences. This attempt is achieved by taking the product of the square of the length *len* and the number of occurrences *s* of each substring. By this way, we can see the great importance of a substring that has a greater length than other substrings although it occurs fewer times. The square of the length is used due to its better performance on strings that have a small number of symbols; this fact has been deduced by our experiments. As the number of the symbols is increased, this formula becomes inefficient if applied on its own which is a misleading point that will be analyzed and be improved in the next subsection, where we present a more efficient formula.

After that, each substring is mapped to a certain limit threshold *lim*. That limit is compared with a general limit threshold given by the user for all the substrings. Using that technique, which is counting a limit that defines the importance of a substring more generally, we have a more efficient way to judge if a substring is repeated enough, in comparison to the technique used in Diamanti et al. [10]. As a result, all the substrings that have a limit bigger than the user defined threshold are stored in the two-level index due to its efficiency because of their repetition.

The above technique has a good performance, but there are two misleading points that in practice could not make the algorithm very efficient as expected in theory. From the created formula, we are based on the square of the length of each substring, which is a way that it works mainly in strings with small total length and not strings of general length. The second problem, as mentioned above, is the user defined parameter. Since the length of the string is raising, the knowledge we have about its substrings and occurrences is more difficult to be defined or pre-processed. Hence, this attempt is efficient only for short strings. A mathematic formula needs to be found for larger strings and this effort is analyzed in the next subsection.

4.1.2. Average approach

In this subsection, is analyzed the effort to confront the problems of the user-defined parameter and the difference in the occurrences of the substrings of different lengths. To achieve that, we introduce a mathematic formula, which is based on the length of the string and the length of its substrings. By calculating the average time, a substring of a certain length may be occurred in the string, given the length of the alphabet, we can take the average number of occurrences of a substring. This average number counted, is being applied in the formula of the previous subsection and the average limit threshold ML for each possible length of substrings is exported.

$$ML = len^2 * \frac{N - len + 1}{k^{len}}$$

In the mathematic formula above, there are two distinguishable parts. The first part is based on the calculation of the average time a substring occurs in the string. This calculation is based on the difference of the total length of the string *N* and the length of the substring *len* increased by one; this difference is divided by the size of the alphabet $k = |\Sigma|$ (Σ is the alphabet) raised to the length of the substring. In essence, this is the average number of occurrences, so we multiply it with the square of the length of each substring we want to compare it. An average limit threshold *ML* is then exported by that calculation and is compared with the limit *lim* of each substring. If *lim* is greater than *ML* then this substring is stored in the two-level index because it is often repeated, else it is stored in the classic inverted index.

The above formula is used to calculate the average times a substring with a certain length *len* is occurred in a string. The second part of the formula is calculating the occurrences based on the length of the string *N*, the length of the substring and the size of the alphabet *k*. This part of the formula of the average limit *ML* is corresponding to the number of occurrences a substring occurs for the limit threshold formula *lim.* For the formula of the average limit *ML* to be compared exactly with the limit threshold *lim*, we need the first part which is the square of the length of the substring that is compared.

Applying that threshold in the algorithm gives an efficient way to be recognized automatically if a substring is repeated often, so as to be stored in the two-level index. This threshold overcomes the

problems of the user defined parameter imprecision, by taking the exact number of occurrences that is efficient for a substring to be stored. Using a different threshold for each substring length, a more efficient way to judge if a substring is repeated efficiently is achieved.

4.2. Limit Threshold for weighted sequences

Weighted sequences have a slightly different behavior and treatment to calculate if the number of its occurrences is large enough in order for the substring to be stored in the two-level scheme. The difference lies in the positions of the string that are able to take more than one characters of the alphabet with a certain probability. In the sequel, we describe how the technique that is used for regular strings can be transferred in the weighted sequences and how it can be made quite efficient.

Firstly, the limit threshold of every substring has to be found. The formula to compute the limit threshold *lim* for each substring depends on its length and the number of occurrences. In more details, we manage the limit threshold in the same way as in regular strings, by getting the product of the square of a substring's length and its occurrences. This formula gives us the limit threshold for each substring to compare it with the average limit threshold ML.

Secondly, we should define the limit threshold ML. To do so, the weighted points from the string should be analyzed. For a certain length, we extract all the substrings from the string. All the substrings that have at least one weighted point, should be extracted to the corresponding regular substrings. In figure 1 we can see the regular substrings that are exported from a substring with one weighted position. At this time, we can easily compute the limit threshold ML for a certain length. So, we have to define the way a substring with a weighted point is extracted into regular strings.

For example, for a given alphabet $\Sigma = \{A, G, C, T\}$ and w_1 a weighting position, as we can see in Figure 1, all the characters with positive probabilities to appear in w_1 are exported from the weighted sequence. Afterward for the substrings that are exported we calculate its limit and compare it with the ML so as to decide in which scheme, it should be stored.

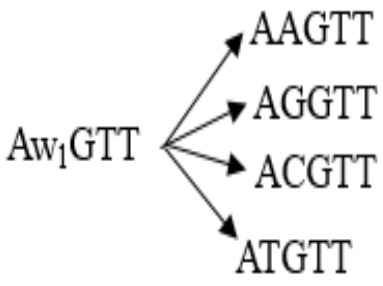

Figure 1. Exporting regular strings from a weighted sequence.

Applying the formula of the regular strings in the weighted strings gives us the advantage to overcome all the problems that existed with the previous limit threshold. Both disadvantages, of the

user, defined and a generalized limit threshold for all the substrings are eliminated with the new limit lim for each substring and the average limit threshold ML, ending up in a much more efficient structure for compaction as it turns out in the next section where we describe the experimental results.

5. Experiments

5.1. Experimental Settings

In order to prove the efficiency of our algorithm, the algorithm with the new thresholds that added was implemented. Data of 10MB, for the experiments, were downloaded from NCBA database (ftp://ftp.ncbi.nih.gov/genomes/) from the genetic material of a living organism. Comparing the results of these data with the one of the previous algorithm fully reflects the bigger efficiency of our algorithm. The length of the substrings and the n-grams that are exported are the same as the length of the previous algorithm of Diamanti et al. [10], in order for the experimental comparison to be exact and as a result, we export the size in bytes of the two-level inverted index.

The computer system that the experiments took place, was an online server of Okeanos systems (https://okeanos.grnet.gr/home) using Windows Server 2012 R2 Datacenter operating system, with a QEMU Virtual CPU version 2.1.2 2.10 GHz processor with two cores and 6GB RAM. The technique implemented, was the determination of the substrings from the string and the storage in the two-level scheme, so as to export the figures of the space used. For the weighted sequences, the weighted positions are with the same probability as in the previous paper to make the comparison more valid, also the figures have the same structure to make the comparison with the naked eye look clear.

5.2. Strings Results

In Figure 2, we can see two lines in each subfigure, the green one which represents the limit threshold technique of this paper and the blue line which is the previous threshold that has been implemented from Diamanti et al. [10] for the two-level n-gram.

With the use of that threshold, there is a stable performance in the space needed, in contrast with the previous algorithm, where the space needed is an order of magnitude greater for each increase in the n-grams. The greater order of magnitude as long as n rises is shown under each subfigure. This notation (x1E+1) represents that for the blue line the space needed is from 0.5E+9 to 1.5E+9 while for the notation (x1E+2) the space needed for the previous algorithm for Diamanti et al. [10] is 0.5E+10 to 1.5E+10. So, by succeeding to prove the better performance in the bigrams, we automatically show the best performance for all n-grams, as long as with our technique for each n in n-grams space is about the same.

Firstly, the two-level scheme implementation of the previous algorithm is not as stable as ours and that is because of the general threshold, in a small length of substrings (four to five), almost all of them are stored because of the great number of occurrences. The performance of our algorithm is

stable in this substring length, in contrast to the approach of Diamanti et al. [10]. After that, for the length of substrings six to ten, the previous algorithm achieves a very good compaction. Our approach, due to its stability, as we can see from the first green graph, needs the same amount of space to be stored. As the length of the subsequence is getting bigger, especially when the substring length equals to ten, our approach needs some more space to store the data due to the bigger subsequences.

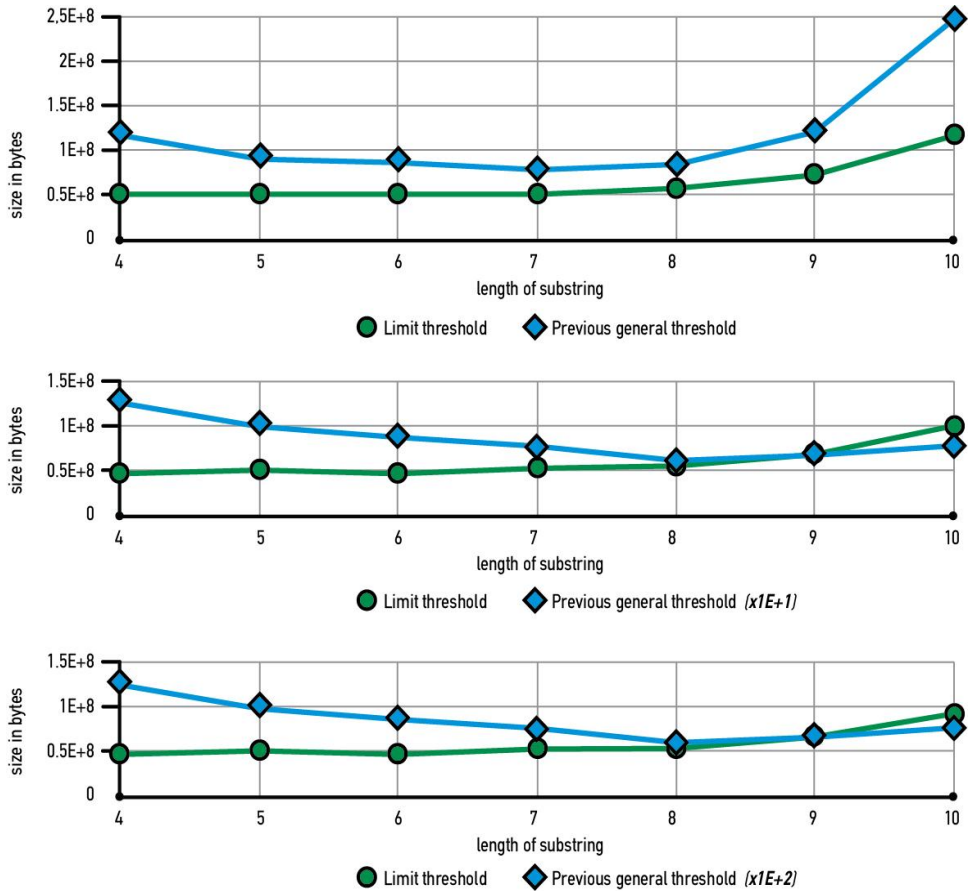

Figure 2. Data 10 MB in regular strings for n = 2, n = 3 and n = 4 top to down respectively.

As the n rises, in our approach the space needed for the data to be stored is slightly less than bigrams. In contrast, in the previous algorithm for each increment of n, space needed raises about an order of magnitude, this is shown under each subfigure next to the description of the blue line. As a result, in trigrams and especially in four-grams our approach needs way less space to be stored, making a very effective data compression.

In the next figure (Figure 3) we show our synthetic data for the creation of the weighted points using the green line. On the other hand, using the blue line we show the results of the previous algorithm of Diamanti et al. [10], an extended version which was taken after private communication with the authors. On the NCBA database, there are no weighting data, as a result, we created

synthetic data using a tool, based on the same synthetic tool that the weighted strings were created on Diamanti et al. [10].

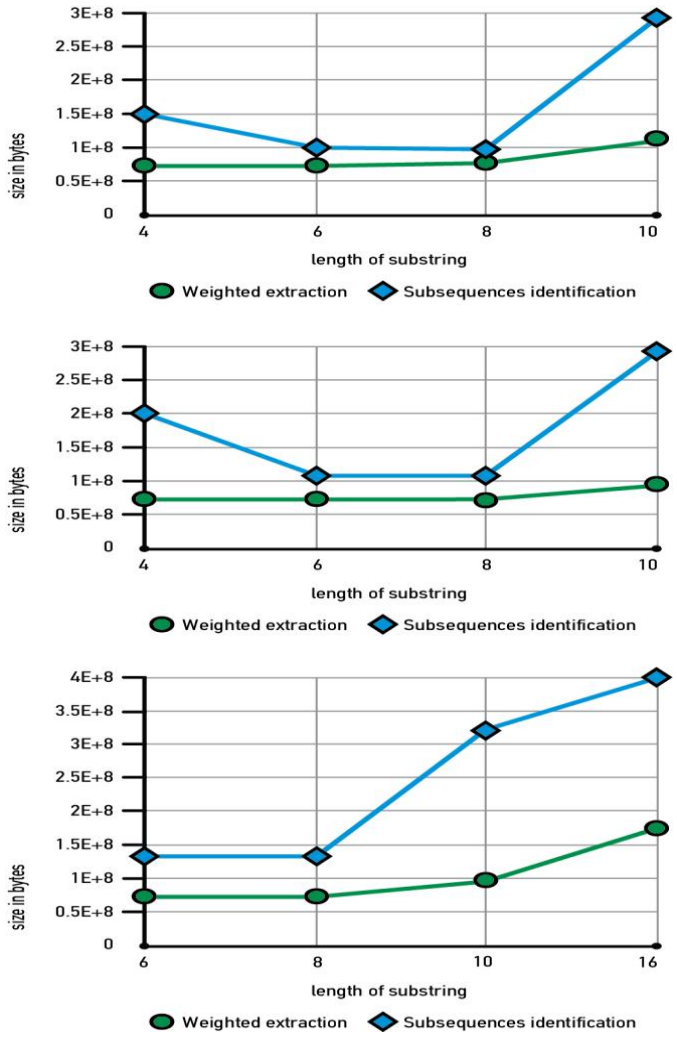

Figure 3. Data 10 MB in weighted sequences for n = 2, n = 3 and n = 4 top to down respectively.

The synthetic tool for the creation of the weighted points in our data was implemented in Python programming language. This tool was processing the string and in some random positions of the strings, it inserted the weighted points. The frequency for the random points to be inserted, was the same as in the previous algorithm of Diamanti et al. [10], so as for the comparison to be equal.

As we can see in this figure too, our approach is more stable. As the length of the substring increases the space that is needed for the data to be stored is not changing roughly. On the other hand, in Diamanti et al. [10], for bigrams, the space needed when the length of the subsequence is 4 is much more than ours because the threshold of their approach was letting almost all of the substrings with this length to be stored in the two-level index. For the length of substrings 6 and 8, there is a better performance for the previous algorithm, but still not as good as our approach, which needs about the same space as in length of substring 4, to be stored. The biggest difference can be shown in

length of substring 10 where our algorithm's space needed is slightly growing due to the bigger sequences that are stored, while in the previous algorithm the space needed for the data to be stored is much bigger. The same behavior is observed in trigrams and four-grams too. Especially in four-grams, we can observe more easily the huge increase of space needed as the length of substring increases in the previous algorithm of Diamanti et al. [10], in comparison with our approach where the space needed for the data to be stored is slightly increasing.

Our formula provides a threshold that is suitable for any length of substrings and for any length of n-grams. By using average weight for every length and calculating the limit of each substring we can efficiently store and compress a large string. Efficient data compression for large data strings is a legitimate achievement. Lastly, it is proven in practice our technique's efficiency.

6. Conclusion—Future Work

The purpose of this paper was to improve the space reduction when handling string using two-level inverted indexes, by employing novel threshold techniques and mathematic formulas. Firstly, the previous attempt that our technique was based on was presented and then our technique based on a novel limit threshold was described and analyzed and its performance was given on both ordinary and weighted strings.

The efficiency of the data compression in this paper is clear and the experimental comparison depicts the improved performance of our limit threshold technique. Our technique, as shown, has a better performance and data compression than the previous algorithm but we have located some points that can broaden the use of our technique, or maybe improve its performance as long as it is possible. In Kim et al. [25], there is an approach to apply the two-level n-gram inverted index in approximate string matching. With the use of this approach, we can try to apply our technique in approximation string matching. Finally, there are some other techniques that can be applied in the weighted sequence threshold to slightly improve the performance based on the possibilities of each character to occur in a weighted position. It could also be interesting to see how our limit thresholds will be improved-changed if we choose different compressed implementations of inverted files.

Conflict of Interest

All authors declare no conflicts of interest in this paper.

References

1. Blandford D, Blelloch G (2002) Index compression through document reordering. In: Storer JA and Cohn M, Eds., Proc. 2002 IEEE Data Compression Conference, April, IEEE Computer Society Press, Los Alamitos, CA. pp. 342-351.
2. Scholer F, Williams HE, Yiannis J, et al. (2002) Compression of inverted indexes for fast query evaluation. In: Beaulieu M, Baeza-Yates R, Myaeng SH and Jarvelin K, Eds., Proc. 25th

Annual International ACMSIGIR Conference on Research and Development in Information Retrieval, August, Tampere, Finland, ACM Press, New York, pp. 222-229.

3. Brandon MC, Wallace DC, Baldi P (2009) Data structures and compression algorithms for genomic sequence data. *Bioinformatics* 25: 1731-1738.
4. Gonzalo N, Veli M (2007) Compressed full-text indexes. *ACM Computing Surveys* 39: 2.
5. Ricardo AB, Berthier AR (2011) Modern Information Retrieval—the concepts and technology behind search, Second edition. Pearson Education Ltd., Harlow, England ISBN 978-0-321-41691-9.
6. Ian HW, Alistair M, Timothy CB (1999) Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. Morgan Kaufmann ISBN 1-55860-570-3.
7. Weiner P (1973) Linear Pattern Matching Algorithms In 14th Annual IEEE Symposium on Switching and Automata Theory.
8. Ukkonen E (1995) On-Line Construction of Suffix Trees. *Algorithmica*: 249-260.
9. Iliopoulos C, Makris C, Panagis Y, et al. (2006) The Weighted Suffix Tree: An Efficient Data Structure for Handling Molecular Weighted Sequences and its Applications. *Fundament Informa* 71: 259-277.
10. Diamanti K, Kanavos A, Makris C, et al. (2014) Handling Weighted Sequences Employing Inverted Files and Suffix Trees. In Web Information Systems and Technologies, pp. 231-238, extended version, private communication with the authors.
11. Kim MS, Whang KY, Lee JG, et al. (2005) N-gram/2l: A space and time efficient two level n-gram inverted index structure. *Int Conference Very Large Databases*: 325-336.
12. Christodoulakis M, Iliopoulos CS, Mouchard L, et al. (2006) Computation of repetitions and regularities of biologically weighted sequences. *J Comput Biol* 13: 1214-1231.
13. Zhu H, Kollios G, Athitsos V (2012) A Generic Framework for Efficient and Effective Subsequence Retrieval. *Proceed Very Large Data Bases*: 1579-1590.
14. Amir A, Iliopoulos CS, Kapah O, et al. (2006). Approximate matching in weighted sequences. *Combinator Pattern Match:* 365376
15. Alatabbi A, Crochemore M, Iliopoulos CS, et al. (2012) Overlapping repetitions in weighted sequence. *Int Informat Technol Conference:* 435-440.
16. Zhang H, Guo Q, Iliopoulos CS (2010) An algorithmic framework for motif discovery problems in weighted sequences. *Int Conference Algorithms Complex:* 335-346.
17. Zhang H, Guo Q, Iliopoulos CS (2010) Varieties of regularities in weighted sequences. *Algorithmic Aspect Informa Manage*: 271-280.
18. Li G, Deng D, Feng J (2011) Faerie: Efficient Filtering Algorithm for Approximate Dictionary-based Entity Extraction. *Special Interest Group Manage Data*: 529-540.
19. Mouza C, Litwin W, Rigaux P, et al. (2009) AS-Index: A Structure for String Search Using n-grams and Algebraic Signatures. *Confer Informat Knowledge*: 295-304.

20. Marsan L, Sagot MF (2000) Extracting structured motifs using a suffix tree—algorithms and application to promoter consensus identification. *Int Conference Res Comput Mol Biol:* 210-219.
21. Sun Z, Yang J, Deogun JS (2004) Misae: A new approach for regulatory motif extraction. *Computa System Bioinforma Conference*: 173-181.
22. Holub J, Smyth WF (2003) Algorithms on indeterminate strings. In Australasian Workshop on Combinatorial Algorithms.
23. Holub J, Smyth WF, Wang S (2008) Fast pattern matching on indeterminate strings. *J Discrete Algorithms* 6: 37-50.
24. Manning CD, Raghavan P, Schutze H (2008) Introduction to Information Retrieval. Cambridge University Press.
25. Kim MS, Whang KY, Lee JG (2007) N-gram/2l-approximation: a two-level n-gram inverted index structure for approximate string matching. *Compu System Sci Engineer* 22: 6.

AIMS Press