Fast Discovery of Statistically Interesting Words

Pedro Pereira, Nuno A. Fonseca, Fernando Silva

Technical Report Series: DCC-07-01

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL
Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

Fast Discovery of Statistically Interesting Words

Pedro Pereira[1,2], Nuno A. Fonseca[1,3], and Fernando Silva[1,2]

[1] Artificial Intelligence and Computer Science Laboratory, Rua do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
[2] Department of Computer Science, Faculty of Science, Universidade do Porto, Rua do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
[3] Institute for Molecular and Cell Biology, Rua do Campo Alegre 823, 4150-180 Porto, Portugal
{pdr,nf,fds}@dcc.fc.up.pt

Abstract. In biological sequences, statistically interesting words can lead us to detect subsequences that have a relevant functional significance (e.g. subsequences preserved through evolution or copies of genes). In this paper, we present an efficient method for discovering statistically interesting words in biological sequences. It is based on suffix arrays and for a sequence of size n it has a worst case time of $O(n^2)$ but it is faster in practice. It consumes $5n$ bytes of primary memory in practice but needs $9n$ bytes of secondary memory. The filter to classify statistical interesting words is based on the measure for their overrepresentation in the sequence or sets of sequences where they occur. We implemented the word discovery tool, evaluated its performance and validated its usefulness by running it on large and real biological sequences, and by checking if the words found were known motifs or patterns in the literature.

1 Introduction

In recent years, we have witnessed an exponential increase in the amount of biotech data, namely in genomic [1] and proteomic [3] data. The volume of data has been doubling every three to six months as a result of automation in biochemistry and projects of genome sequencing. Therefore, tools to analyse biosequences should be fast and memory efficient in order to cope with the vast amounts of data available.

Our research goal is to devise an efficient and automatic tool capable of detecting statistically (overrepresented) *interesting* words in biosequences. This is important because statistically *interesting* words in biosequences can lead us to detect subsequences preserved through evolution, copies of genes, transposons, signals, etc. Since we are interested in DNA and aminoacid sequences, we consider a word as being a sequence of symbols, with no spaces, punctuation characters or any other kind of word separators.

In this paper we describe a fast and memory efficient tool that given a sequence or a set of sequences reports all statistically interesting words that occur in the input. The problem of interesting word discovery was divided into two phases: i) the identification of all *interesting* words; ii) and, the filtering of statistically *interesting* words using a statistical filter. Several measures can be used to classify a word as statistically *interesting*. We explored measures that are directly related to the over-representation or under-representation of the words. The tool's performance is evaluated in several biotech data and some conclusions are drawn. The program and respective documentation is available under the GNU Public License at `http://www.ncc.up.pt/~pdr/word-discovery/`.

2 Identifying Interesting Words

A biological sequence can be seen computationally as a string. The problem addressed in this paper can be stated as finding all statistical interesting words (substrings) in a string (biological sequence or set of sequences). Note that a string containing $O(n)$ characters can have up to $O(n^2)$ substrings (repeated).

A substring $x = x_1 x_2 \cdots x_n$ is *interesting* if any character z appended to it, does not create a new substring $y = x_1 x_2 \cdots x_n z$ with the same frequency as x in the main string. For example, let the

string be *abaababa*. The substring *ab* is not *interesting* because *aba* occurs with the same frequency and *aba* has *ab* as a prefix. This classification of *interesting* substrings has an important consequence: there are only $O(n)$ such substrings in a string and all substrings not considered *interesting* can be inferred by the ones that are considered *interesting* [19].

Many problems involving strings can be solved using suffix trees [14]. We looked into them for solving the problem of computing the frequencies of all substrings that occur in a string. In fact, after constructing the suffix tree, the problem is directly solved by performing a depth-first traversal on the tree. This works because the suffixes sharing a prefix have the same prefix-path on the tree. The real advantage of suffix trees lies in the fact that can be constructed in linear time and space. In spite of this characteristic, they are not very practical due to their high memory consumption, which can be as much as 20 bytes per input character [9]. Furthermore, suffix trees have the extra problem of being relatively difficult to construct [9, 14].

2.1 Suffix Arrays

Since we are interested in analyzing DNA or aminoacid sequences with tens of millions base pairs, in general, we cannot perform all computations in main memory. Despite being possible to construct a suffix tree in secondary memory, it would take a prohibitive amount of time. This is due to the awful cache characteristics of the currently known construction algorithms [9]. There is, however, a better alternative for this problem: suffix arrays.

A suffix array [11] is a simple data structure that provides efficient look-up of any substring of a text and identification of repeated substrings. It is more compact than a suffix tree and can be easily stored in secondary memory. A suffix array is a sorted list of the suffixes of a given string. The sorted list is represented as an array of integers, or SA, that identify the suffixes in lexicographic order. An often useful auxiliary data structure is the LCP array, an array containing the length of the longest common prefix between each substring and its predecessor in the suffix array. Since the LCP array is vital for many of the algorithms presented, we will, from now on, interpret "suffix array" as "suffix array with the LCP array". Theoretically, the space requirements of suffix arrays for a string of length n are $O(n \log n)$ bits because $O(n)$ integers with $\log n$ bits capacity are needed. In practice, we limited n to $2^{32} - 1$, or 4 gigabytes, meaning that each integer has exactly 4 bytes. Therefore, a string with length n needs $9n$ bytes in the worst case; n for the string; $4n$ for the sorted array and $4n$ for the longest common prefix array. This is less than half of the space needed for suffix trees [9] in the worst case.

2.2 Suffix Sorting

Sorting efficiently all suffixes in a string is not easy. The obvious solution involving a comparison sort, such as merge-sort or quick-sort, is $O(n^2 \log n)$ in the worst case. With radix sort the worst-case complexity is lowered to $O(n^2)$, however that is still too high for large sequences. In [11], the authors propose an algorithm that is $O(n \log n)$, and more recently (year 2003) some linear time algorithms [8, 7, 4] have been described. These algorithms can only be applicable if the alphabet size is constant (as is the case of biosequences).

After some practical experiments[4] we observed that the trivial solution using a comparison sort, was too slow (as predicted), but has an optimal memory usage - $5n$ bytes; the linear time algorithm from Kärkkäinen and Sanders [4] was quite slow in practice and used too much memory - $18n$ bytes; and the Divsufsort [15] was fast in practice (the worst-case time complexity is $O(n \log n)$) and had a reduced memory usage - $5n$ bytes.

We only evaluated the suffix sorting algorithms that had a sample implementation available [8, 7]. Msufsort [12] was also evaluated but it used $9n$ bytes of memory and was not substantially faster than Divsufsort.

[4] We executed each algorithm three times and averaged the runtime and memory required for sorting the suffixes of *Saccharomyces cerevisiae* and *Drosophila melanogaster* (sequences obtained from the release 38 of the Ensembl project).

Memory usage is critical since all the evaluated algorithms operate solely in primary memory. In a 32-bit architecture the process size is usually restricted to less than 2^{31} bytes of space. This means that if the algorithm uses $5n$ bytes of memory, up to 409 megabytes of input can be processed. If $18n$ bytes are used instead, only inputs up to 113 megabytes can be processed. Therefore, based on the experiments and considering the memory restrictions we selected Divsufsort for the suffix sorting step.

2.3 Longest Common Prefix

Many algorithms for suffix sorting compute the longest common prefix (LCP) as an intermediate step (e.g. [11]). However, it was decided to separate the two computations into two different steps. We modified the worst-case linear time LCP construction algorithm given in [6]. It was not used out-of-the-box mainly because of its memory requirements. Besides the optimal $9n$ bytes of space, it requires an extra $4n$ bytes of memory for the Rank array. In [13] the LCP computation is made using the optimal $9n$ bytes of memory. The accesses made to the arrays are, however, random and not cache-efficient.

```
1: for i := 0 to n − 1 do
2:    Rank[Sort[i]] := i
3: end for
4: h := 0
5: for i := 0 to n − 1 do
6:    if Rank[i] > 0 then
7:       j := Sort[Rank[i] − 1]
8:       while (i + h < n) and (j + h < n) and (T[i +
         h] = T[j + h]) do
9:          h := h + 1
10:      end while
11:      Height[Rank[i]] := h
12:      if h > 0 then
13:         h := h − 1
14:      end if
15:   end if
16: end for
```

Algorithm 1. Original GetHeight.

```
1: for i := 0 to n − 1 do
2:    Height[Sort[i]] := i
3: end for
4: h := 0
5: for i := 0 to n − 1 do
6:    if Height[i] > 0 then
7:       j := Sort[Height[i] − 1]
8:       while (i + h < n) and (j + h < n) and (T[i +
         h] = T[j + h]) do
9:          h := h + 1
10:      end while
11:      Height[i] := h
12:      if h > 0 then
13:         h := h − 1
14:      end if
15:   end if
16: end for
```

Algorithm 2. Modified GetHeight.

It is possible to use $9n$ bytes of memory for computing the inverse LCP, reducing to $5n$ bytes the space needed to be addressed in a random way. The remaining $4n$ bytes are used sequentially. The idea is to reuse the Height array to compute Height[Rank[i]] instead of Height[i] for every i. Although the need to make an extra access to Sort when LCP[i] seems a terrible disadvantage, every access to element Height[i] is coupled with an access to Sort[i]. The algorithm that use the LCP uses its indexes sequentially, so the extra access to Sort is not a problem.

The modified algorithm can be seen in Algorithm 2. The changes are small: Height[i] is rewritten when it is no longer needed; LCP[i] can be accessed by Height[Sort[i]] because LCP[Rank[i]] = Height[i] and LCP[i] = Height[Sort[i]].

2.4 Substrings and Associated Frequencies

It turns out that computing all *interesting* substrings contained in a string of length n can be done using only the LCP array and an extra $4n$ bytes of memory in $O(n)$ time in the worst case.

The problem of listing all the *interesting* substrings and their associated frequencies using the LCP array is equivalent to list all the maximal areas in a histogram. All areas can be described as an interval and a height.

Scanning the LCP array left-to-right, being at position i, if LCP[$i+1$] is smaller than LCP[i] then we know that some maximal areas have the right interval value i. If we report the areas right-to-left,

then their heights and their left interval values will be decreasing, because they all have the same right interval i. The idea is to have a stack which holds the incomplete sub-problems. We process the LCP array left-to-right. If the next LCP value is greater or equal than the current LCP value ($\text{LCP}[i+1] \geq \text{LCP}[i]$), then the next value (which is $\text{LCP}[i+1]$) is pushed onto the stack. If it is smaller, then the stack is popped until the value on top of the stack is smaller or equal than the next LCP value (**while** top() $<$ LCP$[i+1]$) **do** pop() **end while**). The areas are reported when the stack is being popped and the next value on the stack is smaller than the value just popped.

Using the Height array, as computed from Algorithm 1; making the adjustments needed for the characteristics of LCP array instead of a histogram; and using the indexes of the LCP array instead of the actual values, the algorithm described can be seen in Algorithm 3. This algorithm computes all *interesting* substrings and their respective frequencies.

```
 1: push(0)
 2: for i := 0 to n − 1 do
 3:    ReportWord(i, i, Height[i])
 4:    while Height[top()] > Height[i + 1] do
 5:       j := top()
 6:       if Height[top()] < Height[j] then
 7:          ReportWord(top(), i, Height[j])
 8:       end if
 9:    end while
10:    push(i + 1)
11: end for
```

Algorithm 3. Original GetInteresting.

```
 1: push(0)
 2: for i := 0 to n − 1 do
 3:    ReportWord(i, i, Height[i])
 4:    while Height[top()] > Height[i + 1] do
 5:       j := top()
 6:       ReportWord(top(), i, Height[j])
 7:    end while
 8:    j := top()
 9:    if Height[j] < Height[i + 1] then
10:       push(j)
11:    end if
12:    push(i + 1)
13: end for
```

Algorithm 4. Improved GetInteresting.

It is possible to further improve Algorithm 3 by disallowing duplicates on the stack. This helps controlling the stack size, which we want to be as low as possible, since the Height array is a memory hog. Such modification is shown in Algorithm 4. We discovered Algorithm 4 independently, before finding out that a very similar algorithm was already described in [5]. A similar algorithm, but using more stack space, is also presented in [19].

Several Sequences After being able to compute all the *interesting* words and respective frequencies for a single sequence, we proceeded to process more than one sequence at a time, computing the global *interesting* substrings and their frequencies. Assuming that there is a character that cannot appear in any sequence, for example the LF, line delimiter in Unix systems, the sequences can be concatenated forming a single sequence separated by LF. This separator is denoted as \$. The introduction of the separator is important, because simply concatenating the sequences would not give us the vital information about the sequence endings.

Since a substring containing the separator spans through several sequences, the *interesting* substrings cannot have the separator. The easiest way to omit such substrings is to change the **while** condition of Algorithm 2 to:

$$\begin{aligned}&\textbf{while } (i+h<n) \text{ and } (j+h<n) \text{ and } (\text{A}[i+h]\neq \$)\\ &\quad \text{and } (\text{A}[j+h]\neq \$) \text{ and } (T[i+h]=T[j+h]) \textbf{ do}.\end{aligned}$$

Sequence Frequency When dealing with sets of sequences more statistics for each substring can be computed, being the number of sequences where the substring occurs an important one. The algorithm for computing the number of sequences where the substring occurs is described in [19]. This algorithm can be improved using the ideas present in Algorithm 3. The improvement achieves $4k$ less space, being k the stack size. This statistic is computed in $O(n \log n)$ time and $O(n)$ space.

3 Statistical Filter

After finding all *interesting* words, a statistical filter is applied to them. A word is considered statistically interesting if it is overrepresented in the sequence or in the set of sequences where it occurs. To measure the over-representation, the expected number of occurrences and the standard deviation of this value is computed. Equivalently, we need to know how the values are distributed.

3.1 Statistical Distribution

We assume that the random variables are independent and identically distributed. Under these assumptions, it follows a binomial distribution. We consider every character position, that can be a possible place for the word occurrence, as a Bernoulli trial. For example, if we have the sequence `ACGATCAGTACA` and the word we are computing the statistics for, has length 5, there are exactly 8 places where the word can occur. Generalizing, having a sequence and word of length S_n and W_n respectively, there are $S_n - W_n + 1$ places where the word can appear if $S_n \geq W_n$ or zero otherwise.

The binomial distribution gives the discrete probability $b(k; n, p)$ of obtaining exactly k successes (matches) out of n Bernoulli trials (word positions). Each Bernoulli trial is true with probability p and false with probability $q = 1 - p$. For large values of n and small values of p, typically larger than 1000 and smaller than 0.1, respectively, the binomial distribution can be approximated by the Poisson distribution with $\lambda = np$, which has the probability mass function:

$$p(k; \lambda) = e^{-\lambda} \frac{\lambda^k}{k!}. \tag{1}$$

Equivalently $p(k;\lambda) \approx b(k; n, \lambda/n)$ when n is sufficiently large and p sufficiently small [2]. It is advantageous to use the approximation, since the Poisson cumulative distribution function is less computationally expensive.

3.2 Word Frequency in a Set of Sequences

The number of Bernoulli trials for a word can be easily computed for a single sequence with all characters composing the sequence belonging to the alphabet. For example, in DNA sequences, the `N` is a placeholder and does not belong to the DNA alphabet. The characters that do not belong to the alphabet, are not considered in the computation of the Bernoulli trials (word positions) because they are not used for computing the probability of the alphabet symbols.

The idea is to build a table with an entry of each different *run* length that appear in the input. A *run* is a block of consecutive alphabet symbols surrounded by non-alphabet symbols, begin-of-file or end-of-file. For example, `AxAAxAAA` has three *runs* with lengths of 1, 2 and 3. For each entry (indexed by *run* length), we have the *run* length frequency and two more entries. See Table 1 for an example. The entry "Accumulated" is the accumulated frequency computed bottom up; the "Sum" is an accumulated "Accumulated" times "Length" also bottom up.

Length	Frequency	Accumulated	Sum
1	4	9	19
2	2	5	15
3	1	3	11
4	2	2	8

Table 1. Runs table for `ACTGxAxATxA$xTTTxGx$GAxAxTGCA`.

A sequence of size n cannot have a corresponding table with more than $O(\sqrt{n})$ entries. This is achieved because the table has a number of entries corresponding to the number of *runs* with different lengths presented in the input. If we imagine that the input has *runs* with length $1, 2, \ldots, 2\sqrt{n}$, with each *run* of different length appearing only once, we can see that $2n + \sqrt{n}$ characters of the input are covered. Therefore, in the worst case the table has $O(\sqrt{n})$ size which is when each *run* length only appears once in the input and every single *run* length up to $2\sqrt{n}$ is present.

Being k the length of the word we are considering, if we have n runs with lengths $L_1 L_2 \cdots L_n$, $L_i \geq k$ for $1 \leq i \leq n$, the number of Bernoulli trials is

$$(L_1 - k + 1) + (L_2 - k + 1) + \ldots + (L_n - k + 1)$$

simplifying yields

$$n(-k+1) + (L_1 + L_2 + \ldots + L_n) = A_k(-k+1) + S_k$$

where A_k is the "Accumulated", F_k the "Frequency" and S_k the "Sum". It is possible that an entry for k does not exist in the table. In this case, the formula is $A_i(-k+1) + S_i$, where i is the largest tabulated value smaller than k. This result is important because we can use a binary search for searching the *runs* table. The binary search on the table has complexity $O(\log t)$, where t is the number of table entries.

To check if, an *interesting* word, as computed in the previous section, might have any statistically interesting substring, we only need to test if the whole word is statistically interesting, as such word is the largest, or the representative of the class. This works because we know that

$$p(w_1 w_2 \ldots w_k w_{k+1}) = p(w_1)p(w_2)\ldots p(w_k)p(w_{k+1})$$

and because $0 \leq p(w_{k+1}) \leq 1$ (it is a probability)

$$p(w_1)p(w_2)\ldots p(w_k) \geq p(w_1)p(w_2)\ldots p(w_k)p(w_{k+1}).$$

If the table has entries for both k and $k+1$, then, one can prove that

$$N_k = N_{k+1} + A_{k+1} + F_k. \tag{2}$$

We can conclude that $N_k \geq N_{k+1}$, because by definition, for every k, both A_k and F_k are non negative. It can be shown that (CDF is the Poisson cumulative distribution function)

$$\begin{aligned} \mathrm{CDF}(x, N_k p_{w_k}) &\leq \mathrm{CDF}(x, N_{k+1} p_{w_{k+1}}) \\ 1 - \mathrm{CDF}(x, N_k p_{w_k}) &\geq 1 - \mathrm{CDF}(x, N_{k+1} p_{w_{k+1}}). \end{aligned}$$

In fact, we can generalize the result for all possible k. We can extend the table to all values of k without associated runs in the sequences setting the F_k to zero. Therefore, testing the bigger word is sufficient to determine if the word is statistically interesting because if any substring of a word is statistically interesting, the word is also statistically interesting.

A word is considered statistically interesting if the probability of occurring at least the numbers of times than it did occur is smaller than a user specified threshold. Note that we only verify if the word appears much more than the statistically expected value. If we wanted the opposite, or to classify a word as being interesting, because it appeared less than the number of expected occurrences, we would have to test more than one substring per word.

3.3 Interesting Number of Occurrences in Sequences

A word can also be considered interesting because it occurs in large number of sequences, despite not being interesting when considering the number of global occurrences. It is rather easy to see the following:

$$\Pr(\text{"occurring in sequence } i\text{"}) = 1 - q^n \tag{3}$$

where the n is the number of Bernoulli trials that there are in a certain sequence and with a certain word length. The problem with this approach is that each sequence potentially allows a different number of Bernoulli trials for the same word size. Furthermore, we would end up with too many probabilities - one for each sequence. A short-cut was taken due to efficiency reasons; compute the global n and then divide it by the number of sequences. This is possible assuming that the sequences length are approximately the same.

The equation is then seen as (m is the number of sequences and N_k is the number of Bernoulli trials in the whole input for a word of length k):

$$\Pr(\text{“occurring in sequence } i\text{”}) = 1 - q^{N_k/m}. \tag{4}$$

We then look at each sequence as a Bernoulli trial with a success probability equal to $\Pr(\text{“occurring in sequence } i\text{”})$. The statistics we need to compute can be modeled by a Binomial distribution. The number of sequences is typically low; as low as 10, for example. This means that the Binomial cannot be approximated by the Poisson distribution, which is acceptable, since the Binomial cumulative distribution function is reasonably fast to compute for small numbers.

With this statistic, we compute, for each word, two values: the probability of occurring in more sequences than it did, and occurring in less sequences than it did. A word is considered statistically interesting if any of these two values is smaller than a threshold.

4 Implementation

We implemented the word discovery tool using the C language, because we needed to perform a substantial low-level interfacing with the kernel, namely using memory mapped files and having an extreme control on memory allocation. For the suffix sorting, we use the Divsufsort [15] implementation. We also use the R [17] library for computing the Poisson's and Binomial's cumulative distribution functions. This library is linked to the C code. Since R allows the computation of the log(probability), we used this metric instead of probability where it makes sense (ie, where the values are too low).

In spite of being possible to process sets of sequences up to 409 megabytes in 32-bit machines, not having 2 gigabytes of memory will slow down dramatically the computation. So, in practice, the limit for n is less than $k/5$ bytes, where k is $\min(m, 2^{31} - 1)$ and m is the memory available in the system. All values are in bytes. Additionally, $8n$ bytes of external memory are also required, but these days this is not really a hard restriction. For instance, 32-bit systems with 2 GB of memory are capable of handling inputs up to 400 MB, 64-bit systems with 4 GB of memory can handle inputs up to 800 MB.

5 Experiments and Results

We validated the word discovery tool with genomic data and checked if the words found were known motifs and patterns in the literature. We have also performed some benchmarks for measuring the program's performance. The sequences used in the benchmarks are indicated in Table 2 and were obtained from the release 38 of the Ensembl project. The computer used in the experiments had a “Dual core AMD Opteron Processor 250”, with 4 gigabytes of RAM but only 600 megabytes free.

5.1 Validation

Human Gene for Proinsulin One of the chosen sequences was the human gene for proinsulin from chromosome 11 [16] with 4992 base-pairs. Using our word discovery program on this sequence, it produced:

```
A [Z=388.6] [N=15] [L=26] (1/1) GGGGACAGGGGTGTGGGGACAGGGGT
```

as the best ranked word. `A` indicates that the word was considered statistically interesting because it occurs more times than expected; `Z` is the score of the word; `N` is the number of occurrences; `L` is the word length, and finally `1/1` indicates that the word occurs in one sequence on a total of one sequence. The word found is a super-word of a previously reported pattern `ACAGGGGTGTGGGG` [10].

Saccharomyces cerevisiae We searched for statistically interesting words within the *Saccharomyces cerevisiae* genome, also known as baker's yeast for the properties: $-\log(\text{pvalue}) \geq 2500$, having at least 250 base pairs, occurring more than seven times, and appearing in at least six chromosomes.

Sequence	Length	Average	Standard deviation
Saccharomyces cerevisiae whole genome	12156606	102.435	0.052
Anopheles gambiae chromosome 2R	61545105	330.896	2.269
Drosophila melanogaster whole genome	144141726	12141.928	966.574

Table 2. Organisms used for evaluation and runtimes in seconds.

The most significantly interesting word found occurs nine times in six of the seventeen chromosomes and has a length of 853 base pairs. The word was them feed to BLAST[5] which returned the significant ($p(N) < e^{-6}$) results. The extremely low $p(N)$ indicate that the hits were not expected to occur by pure chance. All the hits were reported to occur in zones marked as retrotransposons.

5.2 Scalability

To evaluate the scalability behaviour of the the word discovery program we considered 3 sequences with increased size and searched on each sequence for all statistically interesting words with $-\log(\text{pvalue})$ greater or equal to 128. Each organism sequence were processed three times, and the average times and standard deviation were recorded.

The values collected are presented in Table 2. The great discrepancy between the first two sequences and the last can be explained by the fact that the last sequence needed 685 megabytes of primary memory. However, only 600 megabytes of memory were available. This lead to the use of swap memory during execution, thus explaining the high average and high standard deviation times. Looking at Table 2 we know that the word discovery program discovers all words of all sizes for a sequence size with over 137 megabytes in 3 hours and 30 minutes. With these results, we believe that the program has feasible runtimes.

6 Related Work

The Teresias [18] is a program developed at the IBM Bioinformatics and Pattern Discovery group. It is based on well-organized exhaustive search based on combinations of shorter patterns. The Teiresias algorithm guarantees that all maximal [18] words are reported. The algorithm needs to receive as input the minimum number of literals that a pattern can contain, L. Another required parameter is W that indicates the maximum distance between any consecutive L literals. In the worst case the algorithm is $O(n^3 \log n)$, but it is reported to work very well when the inputs are highly regular and the parameters W and L are small. Teiresias has a more powerful language since it admits don't care characters and classes of equivalency (for example, in the aminoacid alphabet one can group the symbols according the chemical nature or structural character).

Teiresias also discovers exact words with a statistical filter that relies "on Bayes theorem in conjunction with a 2nd order Markov chain" [18].

Beside the worst-case – $O(n^3 \log n)$ vs. $O(n^2)$, a critical difference is the availability of the programs. While our program is freely available with the full source code under the Gnu Public License, Teiresias is available as a binary and only on a reduced set of architectures.

7 Conclusion

We designed, implemented, and validated a tool for discovering statistically interesting words in biosequences. It supports a flexible alphabet that can be composed by any ASCII subset.

The tool finds the words by performing four steps: i) suffix sorting in $O(n \log n)$ time using $5n$ bytes of primary memory and secondary memory; ii) computing the longest common prefix array in $O(n)$ time using $5n$ bytes of primary memory but $9n$ bytes of secondary memory; iii) calculating the *interesting* words and their frequency in $O(n \log n)$ time using $4n + 12k$ ($k \leq n$ is the maximum

[5] Located at `http://www.yeastgenome.org/`.

stack size, in practice $k \approx c \log n$) bytes of primary memory and $8n$ bytes of secondary memory; iv) filtering the *interesting* words using a statistical filter in $O(n^2)$ time in the worst case (assuming that computing the cumulative distribution function for the binomial and Poisson distributions is $O(1)$ time and space) but fast in practice using $5n$ bytes of primary memory and $9n$ bytes of secondary memory.

Therefore in practice, all steps are performed with a maximum memory consumption of $5n$ bytes of primary memory.

Acknowledgements The work presented in this paper has been supported by funds granted to *LIACC* through the *Programa Operacional "Ciência, Tecnologia, Inovacão" (POCTI) e do Programa Operacional Sociedade da Informacão (POSI) do Quadro Comunitário de Apoio III (2000-2006)*. Pedro Pereira is funded by FCT grant SFRH/BD/30628/2006 and Nuno Fonseca by FCT grant SFRH/BPD/26737/2006.

References

1. Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. Genbank. *Nucleic Acids Research*, 33:235–242, 2005.
2. Willian Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, third edition, 1968.
3. H.M.Berman, J.Westbrook, Z.Feng, G.Gilliland, T.N.Bhat, H.Weissig, I.N.Shindyalov, and P.E.Bourne. The protein data bank. *Nucleic Acids Research*, pages 235–242, 2000.
4. Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings 13th International Conference on Automata, Languages and Programming*. Springer-Verlag, 2003.
5. Toru Kasai, Hiroki Arimura, and Setsuo Arikawa. Efficient substring traversal with suffix arrays.
6. Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer-Verlag, 2001.
7. Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *CPM '03: Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, pages 186–199. Springer-Verlag, 2003.
8. Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *CPM '03: Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, pages 200–210, London, UK, 2003. Springer-Verlag.
9. Stefan Kurtz. Reducing the space requirement of suffix trees. *Software - Practice and Experience*, 29(13):1149–1171, 1999.
10. Amy Lew, William J. Rutter, and Giulia C. Kennedy. Unusual dna structure of the diabetes susceptibility locus iddm2 and its effect on transcription by the insulin promoter factor pur-1/maz. *Proceedings of the National Academy of Sciences of the United States of America*, 97(23):12508–12512, November 2000.
11. Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Jornal on Computing*, 22(5):935–948, 1993.
12. Michael Maniscalco. Msufsort. `http://www.michael-maniscalco.com/msufsort.htm`.
13. Giovanni Manzini. Two space saving tricks for linear time lcp array computation. In *SWAT*, pages 372–383, 2004.
14. Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
15. Yuta Mori. Divsufsort. `http://homepage3.nifty.com/wpage/software/libdivsufsort.html`.
16. Rotwein P., Yokoyama S., Didier D. K., and Chirgwin J. M. Genetic analysis of the hypervariable region flanking the human insulin gene. *The American Journal of Human Genetics*, 1986.
17. R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005.
18. Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences: The teiresias algorithm. *Bioinformatics*, 14(1):55–67, 1998.
19. Mikio Yamamoto and Kenneth W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1):1–30, 2001.