BIOINFORMATICS ORIGINAL PAPER Vol. 25 no. 14 2009, pages 1746–1753
doi:10.1093/bioinformatics/btp321

Sequence analysis

Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome

Verónica Becher*, Alejandro Deymonnaz and Pablo Heiber

Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina

Received on February 9, 2009; revised on May 11, 2009; accepted on May 12, 2009

Advance Access publication May 18, 2009

Associate Editor: John Quackenbush

ABSTRACT

Motivation: There is a significant ongoing research to identify the number and types of repetitive DNA sequences. As more genomes are sequenced, efficiency and scalability in computational tools become mandatory. Existing tools fail to find distant repeats because they cannot accommodate whole chromosomes, but segments. Also, a quantitative framework for repetitive elements inside a genome or across genomes is still missing.

Results: We present a new efficient algorithm and its implementation as a software tool to compute all perfect repeats in inputs of up to 500 million nucleotide bases, possibly containing many genomes. Our algorithm is based on a suffix array construction and a novel procedure to extract all perfect repeats in the entire input, that can be arbitrarily distant, and with no bound on the repeat length. We tested the software on the *Homo sapiens* DNA genome NCBI 36.49. We computed all perfect repeats of at least 40 bases occurring in any two chromosomes with exact matching. We found that each *H.sapiens* chromosome shares ∼10% of its full sequence with every other human chromosome, distributed more or less evenly among the chromosome surfaces. We give statistics including a quantification of repeats by diversity, length and number of occurrences. We compared the computed repeats against all biological repeats currently obtainable from Ensembl enlarged with the output of the dust program and all elements identified by TRF and RepeatMasker (ftp://ftp.ebi.ac.uk/pub/databases/ensembl/jherrero/.repeats/all_repeats.txt.bz2). We report novel repeats as well as new occurrences of repeats matching with known biological elements.

Availability: The source code, results and visualization of some statistics are accessible from http://kapow.dc.uba.ar/patterns/

Contact: vbecher@dc.uba.ar

1 INTRODUCTION

We present a time and memory efficient algorithm and its implementation as a software tool to exhaustively find all perfect repeats in sequences of up to 500 million nucleotide bases. Thus, the input can be made of possibly many whole chromosomes (a single chromosome, two or more). The search is efficiently performed, with no upper bound on the length of the perfect repeats, and the different occurrences can be arbitrarily distant. The output is ordered by length reporting, for each perfect repeat, its number of occurrences and starting positions in the input data. Our tool also gives a quantification in terms of length, diversity and number of occurrences, in possibly many genomes.

The main contribution of our algorithm is its efficiency and exhaustiveness in extracting all perfect repeats in large inputs, hence its usefulness to find novel repeats and to perform cross comparisons in different genomes. We tested our tool on Human genome, our findings are reported in the second part of this article. We verified the efficiency of our method with respect to the tradeoff in time and memory, granted by its theoretical complexity. Our algorithm is based on the *suffix array* construction of Manber and Myers (1993) and a novel procedure to extract all perfect repeats in the entire input. It is well known that the linear space complexity of the alternative data structure, the *suffix trees*, hides a large constant that makes it impossible to allocate and manipulate multiple entire chromosomes even in 8 GB RAM, the currently largest addressable memory with nowadays workstations. Hence, the attractive linear time operations of suffix trees vanishes for such large inputs needed in comparative genomics. In contrast, the small constant involved in the linear space complexity of suffix arrays, and the order of $n \log n$ worst case time complexity for inputs of size n, have made suffix arrays the standard data structure replacing suffix trees (Gusfield, 1997; Kärkkäinen *et al.*, 2006; Poddar *et al.*, 2007; Puglisi *et al.*, 2007).

2 WHAT IS A PATTERN?

Notation: assume an alphabet $\mathcal{A}$, i.e. a set of symbols. A string is a finite sequence of symbols in $\mathcal{A}$. The length of a string w, denoted by $|w|$, is the number of symbols in w. We address the position of a string w by counting from 1 to $|w|$. The symbol in position i is denoted $w[i]$, and $w[i..j]$ represents the substring that starts in position i and ends in position j of w, inclusive. A prefix of a string w is an initial segment of w, $w[1..\ell]$. A suffix of a string w is a final segment of w, $w[i..|w|]$. We say u is a substring of w if $u = w[i..j]$ for some i, j, and we say u occurs in w at position i if $u = w[i..i+|u|-1]$. When u is a substring of w we call w an extension of u.

Definition of a pattern: biological repeats can be generated by many different processes including tandem repeats, segmental duplication and satellites as well as families of transposable elements. They result from sequence divergence, abortive repair of double-strand DNA breaks, incomplete reverse transcription and recombination or nesting among transposable elements (Bergman and Quesneville, 2007; Caspi and Pachter, 2006; Catasti *et al.*, 1999; Goodier and Kazazian, 2008). Repeat classes, such as segmental

*To whom correspondence should be addressed.

duplications and tandem repeats, may be interleaved. We tackle the fragmental nature of biological repeats with the mathematical definition of a *pattern*. It requires exact matching of the different occurrences, allowing nested and overlapping patterns. Biological repeats could then be found by combining smaller exact matching patterns (this phase is beyond the scope of this article). The plots of the distribution of patterns along the chromosome surfaces, as shown in Figure 6, give some insight of such combinations.

We define a *pattern* as a string that occurs more than once, and each of its extensions occur fewer times (Gusfield, 1997). For example, the set of patterns in $w = abcdeabcdfbcde$ is $\{abcd, bcde, bcd\}$. Clearly *abcd* and *bcde* are patterns occurring twice. But also *bcd* is a pattern because it occurs three times in w, and every extension of *bcd* occurs fewer times. There are no other patterns in w (*bc*, for example, occurs three times but since *bcd* occurs the same number of times, *bc* is not a pattern). The time complexity of our algorithm relies on the fact that there can be at most n patterns in any string of length n (Gusfield, 1997).

To find the patterns in many strings requires more than treating them individually. If s and t are two strings, and \$ is a symbol not occurring either in s or in t, the patterns in $w = s\$t$ may be different from getting the individual patterns and take their union, because a pattern in w may occur only once in each string. For instance, if $s = ACTGC$ and $t = CTGAG$, the individual patterns are just C for s and G for t, but there is a pattern CTG that occurs twice in $w = ACTGC\$CTGAG$.

3 ALGORITHM

Let w be a string of length $n = |w|$. The suffix array (Manber and Myers, 1993) of w is a permutation r of the indices $1...n$ such that for each $i < j$, $w[r[i]..n]$ is lexicographically less than or equal to $w[r[j]..n]$. Thus, a suffix array represents the lexicographic order of all suffixes of the input w. For convenience, we also store the inverse permutation of r and call it p, namely, $p[r[i]] = i$. As a first step of our procedure we use a simplified version of the fast algorithm of Larsson and Sadakane (2007) to build the suffix array of the input. We can think each substring of w as a prefix of a suffix of w. The algorithm has to distinguish which prefixes of the already represented suffixes are patterns.

Suppose a pattern u occurs k times in w; thus, it is a prefix of k different suffixes of w. Since the suffix array r records the lexicographical order of the suffixes of w, the pattern u can be seen as a string of length $|u|$ addressed by k consecutive indexes of r. Namely, there will be an index i such that u occurs in positions $r[i], r[i+1], \ldots, r[i+k-1]$ of w. The algorithm has to identify which strings addressed by consecutive indexes of the suffix array are indeed patterns. We need to find out the longest common prefix of such strings. For this task, we use the linear time algorithm of Kasai *et al.* (2001) that computes the longest common prefix (LCP) array of the lexicographically ordered suffixes of w. For any position $1 \le i < n$, LCP[i] gives the length of the longest common prefix of $w[r[i]..n]$ and $w[r[i+1]..n]$. Figure 1 gives an example of the suffix array and the LCP array for $w = mississippi$.

Recall that the definition of a pattern asks that any extension of a pattern must occur fewer times. In this sense, patterns are maximal. Let us say that a candidate is maximal to the left when all its extensions to the left occur fewer times; similarly for maximality to the right. Of course, a candidate is a pattern exactly when it is maximal to the left and to the right. To identify the patterns, we first identify the candidates that are maximal to the right, and then we filter out those that are not maximal to the left.

LCP[i]	r[i]	w[r[i]..n]
	11	i
1	8	ippi
1	5	issippi
4	2	ississippi
0	1	mississippi
0	10	pi
1	9	ppi
0	7	sippi
2	4	sissippi
1	6	ssippi
3	3	ssissippi

Fig. 1. Suffix array and the LCP array for $w = mississippi$.

Each pattern of length ℓ occurring k times in w will correspond to consecutive indexes $i, \ldots, i+k-1$ of the suffix array such that $\text{LCP}[i-1] < \ell$, $\text{LCP}[i+k-1] < \ell$ and $\text{LCP}[i+j] \ge \ell$ for each j where $0 \le j < k-1$. (Notice that in the first two conditions it can also be the case that $i = 1$ or $i+k-1 = n$). Observe that at least one of the LCP[$i+j$] must be exactly ℓ, otherwise we would have a pattern longer than ℓ. For each such set of strings addressed by consecutive indexes of the suffix array, it is clear that the implied substring of w of length ℓ is maximal to the right. This is because there is some j such that $\text{LCP}[i+j] = \ell$, hence any extension of $w[r[i+j]\ldots r[i+j]+\ell-1]$ to the right occurs fewer times.

If there were an extension to the left, then the respective k extensions $w[r[i]-1..r[i]+\ell-1], \ldots, w[r[i+k-1]-1..r[i+k-1]+\ell-1]$ should all coincide. Hence, they should all correspond to consecutive indexes in the suffix array (in the same order as before). To keep just those candidates that are maximal to the left we check that this situation does not hold. The candidate is not maximal to the left if and only if $w[r[i]-1] = w[r[i+k-1]-1]$ and the difference between $p[r[i]-1]$ and $p[r[i+k-1]-1]$ is exactly k. We omit the formal proof because of its technicality.

In the previous *mississippi* example, the sets of positions $\{11,8,5,2\}$, $\{10,9\}$, $\{7,4,6,3\}$, $\{5,2\}$, $\{7,4\}$ and $\{6,3\}$ imply strings that are maximal to the right. However, $\{6,3\}$ implies the string *ssi* which is not maximal to the left because *issi* also occurs twice at positions $\{5,2\}$. To round up the example, the patterns in *mississippi* are: *issi* with two occurrences, *p* also with two occurrences, and *s* and *i*, both with four occurrences.

We call the algorithm **findpat**. Its pseudocode is described in Algorithm 1. It takes as an extra parameter an integer *ml* which is the minimum length of a pattern to be reported (it can be set to 1 if desired). All patterns of length at least *ml* will be found.

To represent the set S of indexes of LCP, we use a convenient data structure. We need that the insertion operation in S, and the queries for the 'minimum greater than a certain value' and the 'maximum less than a certain value' to be done in $\mathcal{O}(\log n)$ time. Such a structure needs to have the total number of elements specified in advance. In our case this is not a problem because the universe for the set S is the integer interval $[0, n]$. The set S is represented by a binary tree of bits with $n+1$ leaves. Each leaf represents one of the $n+1$ elements of the universe, with its bit set to 1 if the element is in S, and set to 0 otherwise. The internal nodes are set to 0 if and only if all its children are set to 0, otherwise they are set to 1.

Algorithm 1

findpat(input: w, ml)
$n := |w|$
$r :=$ suffix array of w
$p :=$ inverse permutation of r
$LCP :=$ longest common prefix array of w and r
$I :=$ permutation array representing an increasing order of LCP
$S := \{u \mid LCP[u] < ml\} \cup \{0, n\}$
$ini := \min\{t \mid LCP[I[t]] \geq ml\}$
–this is the main loop of the algorithm–
for $t := ini$ to $n-1$ **do**
  $i := I[t]$
  $pi := \max\{j \in S \wedge j < i\} + 1$
  $ni := \min\{j \in S \wedge j > i\}$
  $S := S \cup \{i\}$
  if ($pi = 1$ or $LCP[pi-1] \neq LCP[i]$) and ($ni = n$ or $LCP[ni] \neq LCP[i]$) **then**
    –here we have a substring maximal to the right–
    –check if it is maximal to the left–
    if $r[pi] = 0$ or $r[ni] = 0$ or $w[r[pi]-1] \neq w[r[ni]-1]$ or $|p[r[ni]-1] - p[r[pi]-1]| \neq ni - pi$ **then**
      –here it is both maximal to the right and to the left –
      report $ni - pi + 1$ patterns of size $LCP[i]$ whose list of positions in w are contiguous in r starting at pi.
    end if
  end if
end for

Algorithm 2

Maximum less than(input: t)
repeat
  if t is a left child **then**
    $t :=$ rightmost node to the left of t in its level
  else
    $t :=$ parent(t)
  end if
until node t is set to 1
while t is not a leaf **do**
  if right child(t) is set to 1 **then**
    $t :=$ right child(t)
  else
    $t :=$ left child(t)
  end if
end while
return t

The insertion operation only needs to update the branch of the modified leaf, so it can clearly be done in $\log n$ time. Algorithm 2 finds the maximum less than a given t. Finding the minimum greater than t is analogous.

In the repeat loop of Algorithm 2 there is at most one move to the right for each move up, therefore we have in total $\mathcal{O}(\log n)$ iterations. Then, in the while loop, every move goes down one level, therefore there are also $\mathcal{O}(\log n)$ total moves. If the tree is implemented over a bit array, all moves in the previous algorithm are easily implemented in $\mathcal{O}(1)$, therefore the entire running time of each query is $\mathcal{O}(\log n)$.

Time complexity of the main algorithm **findpat**: all steps before the main for loop clearly take less than $n \log n$ operations. The main loop iterates n times. The most expensive procedure performed in the loop body is the manipulation of the tree for the set S, which requires at most $\mathcal{O}(\log n)$ operations. The overall time complexity of the algorithm is then $\mathcal{O}(n \log n)$. This is achievable since it is possible to code the output in linear time and space in the length of the input.

Space complexity of the main algorithm **findpat**: the whole input w is allocated in memory. Since it contains n symbols, its memory usage is $n \log |\mathcal{A}|$ bits. The data structures r, p, LCP and I are arrays of length n whose elements are between 0 and n; therefore, each of them use $n \log n$ bits. The described tree for the set S has $2n+1$ nodes, implemented with an array of $2n+1$ bits. The total needed space is $n(4 \log n + \log |\mathcal{A}| + 2)$ bits. The array I is only used one element at a time in the main loop of the algorithm, so it can be easily handled by the swap memory without significantly affecting the running time.

4 IMPLEMENTATION

The tool is written in C (ANSI C99), it is platform independent and compiles either in a 64 bits machine or a 32 bits one. The memory space requirement is $n(4 \log n + \log |\mathcal{A}| + 2)$ bits for an input of n bits. For $\mathcal{A}$ the ASCII code and storing indices in 32 bits variables, this becomes a total memory space requirement of $17.25n$ bytes. In a 64 bits processor and 8 GB RAM installed, the tool runs inputs of size up to 474 MB without any swapping. Inputs of size up to half a gigabyte can be run efficiently because a small swapping does not affect the running time. Our tests validate this assertion. Much larger inputs would require swapping on oftenly used memory and increase runtime. In 64 bits environments, our tool has a hard limit of the input size of 4 GB because the indices are stored in 32 bits variables. In 32 bits environments, our tool has an input size limit of 237 MB because processes cannot handle more than 4 GB of RAM, regardless of whether they are installed or not.

The input: the tool receives as arguments the input data, and optional parameters including the minimum length of the patterns to be reported in the output. The algorithm runs faster as this minimum length increases, because it saves the effort of searching the smaller patterns, which are the most abundant in common cases. Although the algorithm (and the software tool) accepts any minimal pattern length, it makes sense to choose a minimal pattern length that ensures that *not* all possible patterns of that length occur in the input sequence (much less with the same frequency). For random inputs (i.e. inputs having a uniform distribution of the alphabet symbols) of length m, all patterns of length up to $\log_{|\mathcal{A}|} m$ will have approximately the same relative frequency. But, necessarily, for all longer lengths there will be absent patterns.

Invocation of the tool is as follows: ./findpat [options] *input1 input2 output_dir min_length*. Among the options, -1 counts position starting at one (instead of zero). To search over *two* genomic sequences, let them be in the files *input1* and *input2*. Internally, they will be concatenated inserting a special symbol in between them, and the result is treated as a single input file. In case of a *single* genomic sequence, let *input1* be its file name, and set *input2* to /dev/null (or its equivalent in your platform). To find patterns in *many* genomic sequences $\{s_1, \ldots, s_n\}$, you should construct a single input file by concatenating all sequences interleaved with different

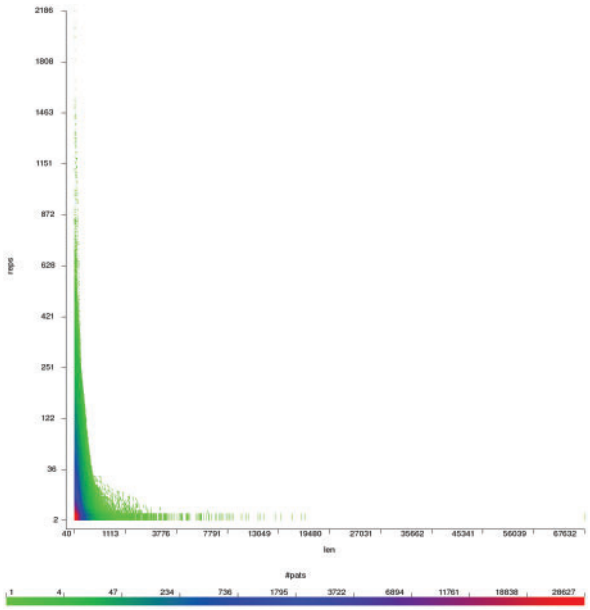

Fig. 2. Patterns of length ≥ 40 in HS1.

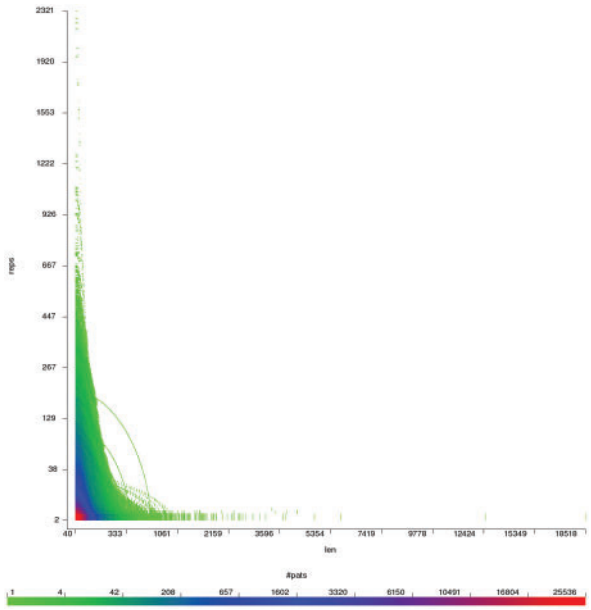

Fig. 3. Patterns of length ≥ 40 in HS2.

special symbols $w{=}s_1\$_1 s_2\$_2 \ldots \$_{n-1}s_n$. Save w in a file *input1*, and set *input2* /dev/null. The fact that each $\$_i$ symbol occurs only once in the string ensures that there will be no patterns containing them.

The output: there are three kinds of output. The *full* output lists all the patterns, each one in two lines. The first line describes the pattern, the number of occurrences of that pattern and the length of that pattern, in a human readable format, for example: TGGATAACTTTTT #5 (13). The second line, which is optional, lists the positions of each occurrence, separated by a space, for example: >14 542 942 <27 416 506 >25 965 497 >25 103 413 <16 670 527. All the positions of the patterns are relative to the beginning of each file, and are preceded with a sign '<' or '>' indicating whether they correspond the first or the second input file. The *statistics* output lists, for each pattern, a single line containing the length of the pattern, the total number of occurrences of that pattern and the number of occurrences in *input1* and *input2*. The *abbreviated statistics* output lists, for each group of patterns of the same length and the same number of occurrences, one line indicating these two values and the number of different patterns in this situation. The three kinds of output can be generated in the same run, directed to three distinct output files. This is considerably faster than running the algorithm three times.

5 PATTERNS IN *H.SAPIENS*

We ran the experiment on Linux operating system with a 64 bits processor with 8 GB RAM. We worked with the FASTA files NCBI 36.49 downloadable from Ensembl. For every two chromosomes, we computed all patterns of length at least 40 bases occurring jointly in both chromosomes, as well as those occurring in individual chromosomes. Our choice of 40 was just to have a manageable output. These sum up 276 runs obtained with ./findpat -p -s -S -2 *input1 input2* ./ 40, and for patterns in single chromosomes, the 24 runs ./findpat -p -s -S *input1* /dev/null ./ 40. We produced the three kinds of output—full, statistics and abbreviated statistics.

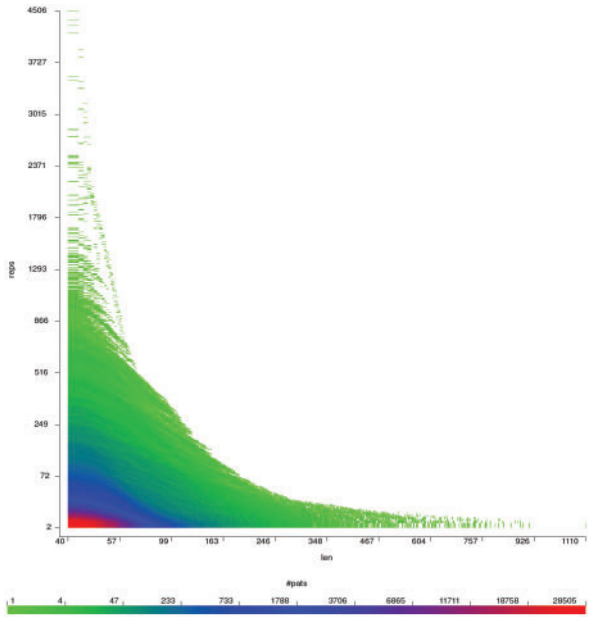

Fig. 4. Patterns of length ≥ 40 jointly in HS1 and HS2.

Statistics: for each abbreviated statistics file, we produced a graph that summarizes its results, as shown in Figures 2–4 for chromosomes 1 and 2 (hereafter HS1 and HS2). For each length and number of occurrences we place a colored point in the *xy* plane which represents the number of found patterns for that length and total number of occurrences. The *x*-axis runs from 40 to the length of the longest found pattern. The *y*-axis runs from 2 to the maximum number of occurrences of a pattern. The colors vary from green to blue to red as the number of different patterns on the same point increases. Observe that in Figure 2, there is a pattern of length 40

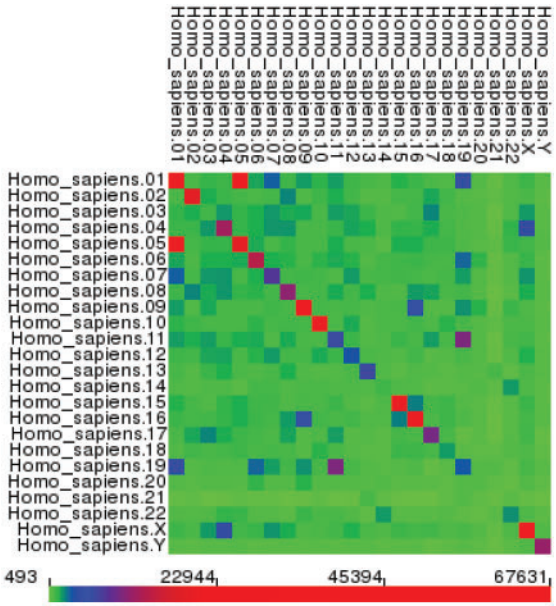

Fig. 5. Length of the longest patterns.

occurring 2186 times; there are 28632 patterns of length 40 having exactly two occurrences. The longest pattern has length 67632, and it occurs twice. The diversity of patterns of length 40 is approximately the same in Figures 2 and 4, but drops to 25536 for the patterns solely in HS2, as shown in Figure 3.

Length of the largest patterns: in the symmetric matrix of Figure 5, we depict in a color scale the length of the longest patterns occurring jointly in each of the two chromosomes addressed by row and column. The diagonal shows that for individual chromosomes. The longest pattern in the whole HS genome has 67631 bases and it occurs inside chromosome HS1 (top left square, the brightest red in the matrix). The longest pattern occurring jointly in two chromosomes has length 21864 and it occurs in HS1 and HS5. The shortest of the longest has 493 bases and belongs to HS17 and HS21 (lightest green square). The longest patterns occur exactly once in each chromosome, except for a few cases, e.g. HS7 and HS20 have exactly four occurrences of their longest joint pattern.

Coverage of chromosomes by patterns: the coverage analysis was done for patterns in single chromosomes, as well as for patterns jointly in two chromosomes. For each chromosome, we counted the total number of bases covered by patterns of length ≥40 bases, and expressed it as a percentage of the size of the chromosome. See Table 1, where the maximum and minimum cases are in boldface. Clearly if one considers patterns starting at a smaller length (or accepts imperfect matching) the coverage increases. So, this quantification is informative as a parameter of the minimum pattern length. We plotted the physical distribution of the patterns along each chromosome, as shown in Figure 6. Here, one pixel represents ∼5000 bases. HS1 has ∼250 million bases, so its map consists in 50 rows of 1000 pixels per row. The color of a pixel indicates the percentage of the bases that has been covered by patterns. There are patterns in single chromosomes that cover long regions in the centromeric area (red spots).

Patterns versus biological repeats: we compared our computed patterns with all repeats currently in Ensembl (2009) in all

Table 1. Coverage of chromosomes by patterns

Chr	Length	%chr	1-crh	%HS1	%chr
1	247249719	12.53			
2	242951149	10.85	**1-2**	**10.07**	9.58
3	199501824	10.35	1-3	9.84	10.09
4	191273063	10.64	1-4	9.84	10.02
5	180857866	11.36	1-5	9.71	10.19
6	170899992	10.78	1-6	9.73	10.32
7	158821424	12.77	1-7	9.81	11.04
8	146274826	10.15	1-8	9.39	9.73
9	**140273252**	**5.46**	1-9	9.16	9.25
10	135374737	12.65	1-10	9.30	10.13
11	134452384	10.79	1-11	9.38	10.38
12	132349534	11.32	1-12	9.43	11.27
13	114142980	6.86	1-13	8.57	7.33
14	106368585	8.59	1-14	8.78	9.01
15	100338915	11.13	1-15	8.63	8.82
16	88827254	13.68	1-16	8.63	10.74
17	78774742	15.81	1-17	8.70	13.55
18	76117153	7.65	1-18	8.24	8.63
19	**63811651**	**17.35**	**1-19**	8.68	**15.84**
20	62435964	8.94	1-20	8.15	9.85
21	46944323	5.56	1-21	7.22	6.56
22	49691432	10.17	1-22	7.54	9.11
X	154913754	13.85	1-X	9.63	12.39
Y	57772954	9.37	**1-Y**	**6.17**	**4.90**

Left, patterns in single chromosomes. Right, patterns jointly in HS1 and another.

strands, enlarged with the output of the dust program (BLAST reference), and all elements output by the TRF program (Benson, 1999), and RepeatMasker (Smit *et al.*, 2009). This database was compiled by Javier Herrero at EMBL-EBI UK and can be downloaded from ftp://ftp.ebi.ac.uk/pub/databases/ensembl/jherrero/.repeats/all_repeats.txt.bz2. It is ordered by chromosome number and starting position of the entries (many of them overlap). We compared the intervals covered by the occurrences of our patterns, against the intervals covered by the entries in the database of biological repeats. This comparison can be done in linear time with respect to the number of entries of the two files and the size or sizes of the chromosomes involved.

For each chromosome, and for each biological class, we computed the total number of instances in the class, how many of these instances have been (partially or totally) covered by our patterns and the percentage of coverage of each instance. Table 2 shows these results for *all* patterns in HS1 of length ≥40. These include not only just the patterns occurring solely in HS1, but also all the patterns occurring jointly in HS1 and some other chromosome(s). Again, the coverage would increase if one considers patterns starting at a smaller length. A detailed analysis on HS1 reveals that, with the exception of LINE/RTE, all biological classes have a greater coverage by patterns occurring jointly in HS1 and another chromosome, than by the patterns occurring solely in HS1. This contrasts with the fact that, disregarding biological classes, chromosomes have a larger coverage by their individual patterns than by patterns jointly in two chromosomes, see Table 1.

Tandem repeats: each green dotted line in Figures 2–4 depicts a tandem repeat. For instance, a tandem repeat of CCCTT yields patterns of length that are multiples of 5. The more instances of

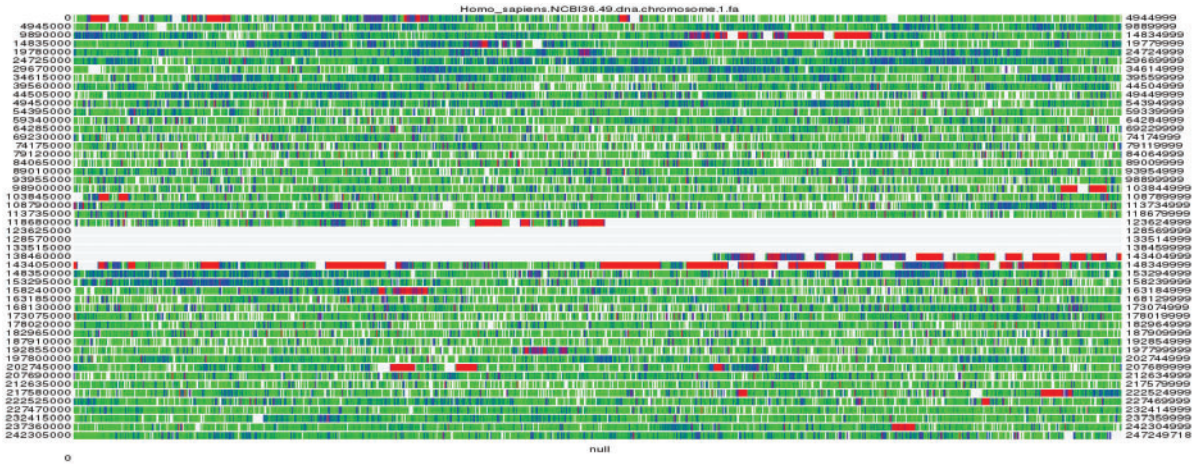

Fig. 6. Coverage of HS1 by its individual patterns of length ≥40 bp. One pixel represents 5000 bases. Red means 100% covered, blue means 0%, green means less than 1%, gray means no base has been covered. Intermediate colors denote intermediate percentages.

Table 2. Biological classes in HS1 and average of the percentage of coverage per instance by patterns of length ≥40

Class	No. of instances in the class	No. of covered instances	Average of the percentage coverage per instance
DNA	1070	87	2.44
DNA/AcHobo	1443	137	2.24
DNA/hAT	225	19	0.57
DNA/Mariner	1163	296	13.57
DNA/MER1_type	15422	2448	5.52
DNA/MER1_type?	147	17	0.21
DNA/MER2_type	6594	2962	12.31
DNA/Merlin	2	0	0.00
DNA/MuDR	158	58	7.00
DNA/PiggyBac	152	83	17.58
DNA/Tc2	610	54	1.62
DNA/Tip100	2275	204	1.26
dust	232100	77096	22.98
LINE/CR1	5932	350	2.39
LINE/L1	64678	24871	28.65
LINE/L2	41411	3066	1.76
LINE/RTE	1161	78	1.08
LTR/ERV	44	1	2.13
LTR/ERV1	13557	7582	28.61
LTR/ERVK	841	788	65.01
LTR/ERVL	9370	1953	10.03
LTR/MaLR	23497	8613	17.44
RNA	54	5	2.75
rRNA	161	37	19.38
Satellite	72	68	61.05
Satellite/acro	3	3	83.55
Satellite/centr	73	67	73.22
Satellite/telo	4	3	45.80
scRNA	133	34	18.66
SINE/Alu	102210	96320	72.16
SINE/MIR	59220	3179	1.93
snRNA	431	116	20.75
srpRNA	82	50	14.49
trf	110344	54064	32.87
tRNA	276	136	47.38
Other	483	483	96.83
Unknown	89	1	0.27
total	695487	285329	49.94

CCCTT in a pattern, the fewer the number of occurrences of such pattern. Table 2 indicates that half of the tandem repeats of HS1 found by the TRF program (Benson, 1999) were covered by our patterns of length ≥40.

Novel repeats: we used the above mentioned database of biological repeats and a database of all genes from Ensembl, also compiled by Javier Herrero at EMBL-EBI UK, available at ftp://ftp.ebi.ac.uk/pub/databases/ensembl/jherrero/.repeats/all_genes.txt.bz2. Given a pattern, if each of its occurrences overlaps with no entries in the database of biological repeats or genes, we consider this pattern to be *novel*. If at least one of its occurrences overlaps with some entry in the database, but some other occurrence overlaps none, we consider this pattern to be a *witness of new occurrences of known repeats*.

The total number of found patterns of length ≥40 in HS1 (occurring solely or jointly in HS1 and another chromosome) was 64449272. About 0.05% of them were novel patterns, and 0.01% were witnessing patterns. The remaining patterns had all their occurrences overlapping some known repeats or genes, in one or more nucleotide bases. About 65% of the novel patterns and about 59% of the witnessing patterns occur in intergenic regions. Novel patterns covered 0.85% of the full chromosome HS1 (actually 2 102 787 covered bases out of 247 249 719). Figure 7 shows their physical distribution.

The largest novel pattern has length 15 79, and it occurs twice in HS1, at positions 143 216 056 and 143 035 804 (we count positions starting at 1), both occurrences are intergenic.

Another example is a pattern of length 1313 that also occurs twice in HS1, in intergenic regions, at positions 146 884 281 and 147 046 309. This novel pattern contains five blocks of TAATTA, which is a recognition sequence for the homeodomain DNA-binding module, highly present in non-exonic ultraconserved elements—these are perfect repeats longer than 200 nt bases occurring jointly in mouse, rat and human (Bejerano *et al.*, 2004; Katzman *et al.*, 2007). It presents a 3.5 rate of occurrence of AT over GC. Higher rates of AT over CG is a property of ultraconserved elements that suggests a biological mechanism (Chiang *et al.*, 2008). None of our novel patterns occur in UCbase, the database of ultraconserved

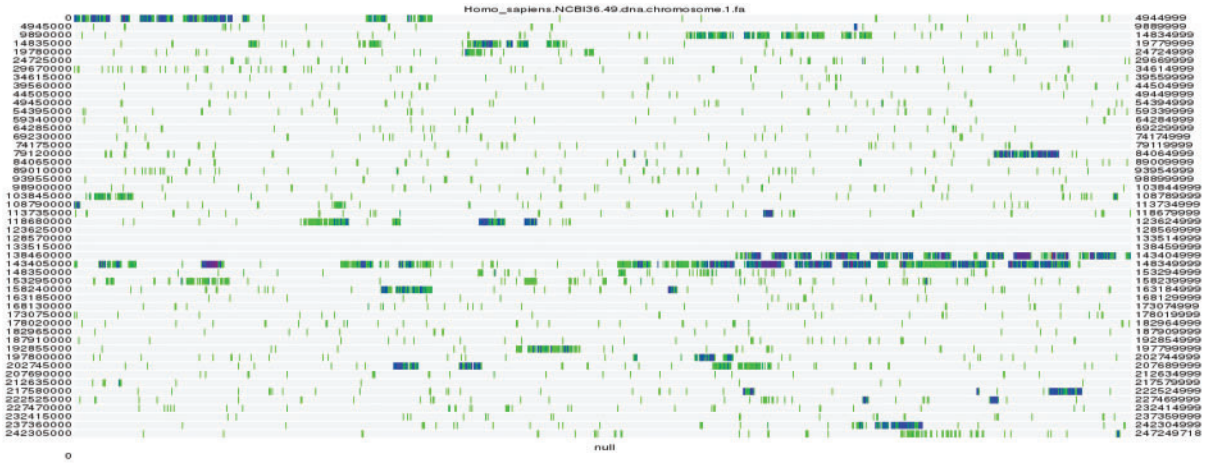

Fig. 7. Coverage of HS1 by the novel patterns.

elements of Taccioli *et al.* (2009). About half of the novel patterns of HS1 (indeed 11 787 out of 25 333) occur just once in HS1 and once or more in some other chromosome(s).

We are currently doing a thorough analysis of the novel patterns, which lays outside the scope of this work. Witnessing patterns in HS1 are reported in http://kapow.dc.uba.ar/_media/homo1-ext.txt.bz2. The format of this file is as follows: pattern, number of *new* occurrences in HS1, length and the list of *new* positions.

Patterns occurring in all chromosomes: there are no patterns occurring jointly in every chromosome having length 600 bases or more; however, there are many of them of length 100, and a lot of length 40. The most frequent pattern of length 100 has 1983 occurrences in the whole HS genome and it is the sequence: TTTTTATGGCTGCATAGTATTCCATGGTGTATATGTGCCACATTTTCTTAATCCAGTCTATCATTGTTGGACATTTGGGTTGGTTCCAAGTCTTTGCTAT. It is part of the reverse transcriptase of L1 retrotransposons.

6 COMPARISON WITH OTHER METHODS

Our algorithm follows the recommendations in the recent survey of Sahal *et al.* (2008a, b) which reviews major repeat exploration methods, devoting special attention to *ab initio* programs as the most promising tools. They emphasize the requirement that inputs be entire genomic sequences, and ask for efficiency as well as for interoperability with other tools. The main feature of our tool is its ability to find all perfect repeats of unbounded length, that can be arbitrary distant in very large inputs. Its efficiency is due to the manipulation of the suffix array data structure in memory, together with an original procedure to extract all patterns. With this in mind, a comparison of our method against the most popular perfect repeat finders follows from the survey of Sahal *et al*. Leave aside heuristic methods such as FORRepeats (Lefebvre *et al.*, 2003), programs performing specific biological identifications as the already classical Sputnik (Abajian, 1994), TRF (Benson, 1999) or TROLL (Castelo *et al.*, 2002) or library-based methods like RepeatMasker (Smit *et al.*, 2009). Existing methods based on the suffix array use the length of repeats as a parameter at each run. All repetitions of the specified length are listed, giving no clue to know when a repeat is maximal. The toolkit of Poddar *et al.* (2007) constructs a suffix array after constructing a suffix tree of the input. For an input of 55 MB, they report 2181 s for the initial suffix tree, while we require 57 s for the actual suffix array, with similar hardware. Then the two methods compute the longest common prefixes in linear time, but afterwards they become incomparable: Poddar *et al*. compute n-grams requiring one pass for each n, while we compute all patterns of arbitrary length in one pass. Lippert (2005) constructs a space efficient suffix array and uses it to find fixed k-mers. His experiment reporting all 20 mers in common between the mouse and human genomes already shows the time increase due to his compressed structure.

The REPuter tool of Kurtz and Schleiermacher (1999), further developed in Kurtz *et al.* (2001), not only looks for exact maximal repeats but also approximate repeats and palindromes. Currently our tool does not support these features. Although it is fairly straightforward to extend it to account for reversed complementary sequences, partial matching requires a vastly different strategy, or a second phase. In the task of finding all exact maximal repeats REPuter has two main drawbacks: first, the input can be up to 135 MB, which is considerably less than 500 MB of our tool. The memory requirement of REPuter depends on the repeat length and the number of occurrences in the input. In the worst case inputs are limited to 1/45 times the RAM size. As output, REPuter gives one entry per each occurrence of a repeat, followed with the list of all the *forward* positions where it occurs. Therefore, the output information is not factorized, and becomes very large, needing $\mathcal{O}(n^2)$ space for inputs of size n; this is not admissible for any practical application. In contrast, our tool produces one entry per pattern with the annotation of *all* the positions where it occurs. Besides, our tool provides some statistics while REPuter does not.

ACKNOWLEDGEMENTS

We thank Joaquín Dopazo for raising this problem to us, and Hernán Dopazo for all his directions towards the open questions. We are highly indebted to Javier Herrero for his compilation at EMBL-EBI UK of the database of all known biological repeats and genes in the Human genome. We also acknowledge the discussions with Alfredo Villasante and Lino Barañao on the biological value of our reported novel repeats.

Funding: Agencia Nacional de Promoción Científica y Tecnológica. Biosidus and IBM Argentina.

Conflict of Interest: none declared.

REFERENCES

Abajian,C. (1994) Sputnik. Available athttp://espressosoftware.com/pages/sputnik.jsp.

Altschul,S. *et al.* (1990) BLAST. Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Bejerano,G. *et al.* (2004) Ultraconserved elements in the human genome. *Science*, **304**, 1321–1325.

Benson,G. (1999) Tandem Repeats Finder: a program to analyze DNA sequences. *Nucleic Acids Res.*, **27**, 573–580.

Bergman,C. and Quesneville,H. (2007)Discovering and detecting transposable elements in genome sequences. *Brief. Bioinform.*, **8**, 382–392.

Caspi,A. and Pachter,L. (2006)Identification of transposable elements using multiple alignments of related genomes. *Genome Res.*, **16**, 260–270.

Castelo,A. *et al.* (2002) TROLL-Tandem Repeat Occurrence Locator. *Bioinformatics*, **18**, 634–636.

Catasti,P. *et al.* (1999) DNA repeats in the human genome. *Genetica*, **106**, 15–36.

Chiang,C. *et al.* (2008) Ultraconserved elements: analyses of dosage sensitivity, motifs and boundaries. *Genetics*, **180**, 2277–2293.

Ensembl (2009) Ensembl. Available athttp://nar.oxfordjournals.org/cgi/content/abstract/gkn828.

Goodier,J.L. and Kazazian,H.H. (2008) Retrotransposons revisited: the restraint and rehabilitation of parasites. *Cell*, **135**, 23–35.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA.

Kärkkäinen,J. *et al.* (2006)Linear work suffix array construction. *J. ACM*, **53**, 918–936.

Kasai,T. *et al.* (2001) Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM '01: Proc.12th Annual Symposium on Combinatorial Pattern Matching.* Springer, London, pp. 181–192.

Katzman,S. *et al.* (2007) Human genome ultraconserved elements are ultraselected. *Science*, **317**, 915.

Kurtz,S. and Schleiermacher,C. (1999)Reputer: fast computation of maximal repeats in complete genomes. *Bioinformatics*, **15**, 426–427.

Kurtz,S. *et al.* (2001). REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Res.*, **29**, 4633–4642.

Larsson,N.J. and Sadakane,K. (2007) Faster suffix sorting. *Theor. Comput. Sci.*, **387**, 258–272.

Lefebvre,A. *et al.* (2003)FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, **19**, 319–326.

Lippert,R. (2005) Space-efficient whole genome comparisons with burrowswheeler transforms. *J. Comput. Biol.*, **12**, 407–415.

Manber,U. and Myers,G. (1993)Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, **22**, 935–948; In *SODA '90: Proceedings of the 1st annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, pp. 319–327, 1990.

Poddar,A. *et al.* (2007) Evolutionary insights from suffix array-based genome sequence analysis. *J. Biosci.*, **32**, 871–881.

Puglisi,S.J. *et al.* (2007) A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, **39**, 4.

Sahal,S. *et al.* (2008a) Computational approaches and tools used in identification of dispersed repetitive dna sequences. *J. Trop. Plant Biol.*, **1**, 85–96.

Sahal,S. *et al.* (2008b) Empirical comparison of ab initio repeat finding programs. *Nucleic Acids Res.*, **36**, 2284–2294.

Smit,A.F. *et al.* (2009) RepeatMasker, open-3.0. 1996-2004. Available at http://repeatmasker.org.

Taccioli,C. (2009) UCbase & miRfunc: a database of ultraconserved sequences and microRNA function. Available at http://microrna.osu.edu/.UCbase4.