Fast Algorithm for the Analysis of the Presence of Short Oligonucleotide Subsequences in Genomic Sequences

V. Fofanov, C. Putonti, S. Chumakov, B.M. Pettitt, Y. Fofanov

Department of Computer Science
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-05-11

May 3, 2005

Keywords: motif, exact string matching, genome-wide analysis.

Abstract

Statistical analysis of the appearance of short subsequences in different DNA sequences, from individual genes to full genomes, is important for various reasons. Applications include PCR primers and microarray probes design. Moreover, the distribution of short subsequences (*n*-mers) in a genome can be used to distinguish between species with relatively short genome sizes (e.g., viruses and microbes). To be able to perform such an analysis, a group of algorithms were developed to specifically deal with the problem of finding the appearance of all possible patterns of size *n* (*n*-mers) in a sequence or text of size *m*. The concept of a counting array allows us to map our problem for large subsequences onto a useful data structure. The run-time operation count estimation $O(4^n+m)$ makes it computationally convenient to accomplish the calculation of the statistics of the presence/absence of all possible 7-20-mers in more than 250 genomes including the human genome.

FAST ALGORITHM FOR THE ANALYSIS OF THE PRESENCE OF SHORT OLIGONUCLEOTIDE SUBSEQUENCES IN GENOMIC SEQUENCES

V. Fofanov, C. Putonti, S. Chumakov, B.M. Pettitt, Y. Fofanov

Abstract

Statistical analysis of the appearance of short subsequences in different DNA sequences, from individual genes to full genomes, is important for various reasons. Applications include PCR primers and microarray probes design. Moreover, the distribution of short subsequences (*n*-mers) in a genome can be used to distinguish between species with relatively short genome sizes (e.g., viruses and microbes). To be able to perform such an analysis, a group of algorithms were developed to specifically deal with the problem of finding the appearance of all possible patterns of size *n* (*n*-mers) in a sequence or text of size *m*. The concept of a counting array allows us to map our problem for large subsequences onto a useful data structure. The run-time operation count estimation $O(4^n+m)$ makes it computationally convenient to accomplish the calculation of the statistics of the presence/absence of all possible 7-20-mers in more than 250 genomes including the human genome.

Index Terms

motif, exact string matching, genome-wide analysis

I. Introduction

Statistical analysis of the appearance of short subsequences in different DNA sequences, from individual genes to full genomes, attracts attention for various reasons. An incomplete list of its applications includes PCR primer [1,2] and microarray probe design [3]. In literature, several previous attempts have also been made to employ the frequency distribution of short subsequences (*n*-mers or motifs) to identify species for relatively short genome sizes (e.g., viruses and microbes). In such an approach, the shape of the frequency distribution for certain short subsequences, 2-4-mers [4-8] and 8-9-mers [9,10], was proposed to be used to decide what microbial genome is being considered based on a given random piece of genome or the entire genome. Algorithmically, such types of analyses employ a repeatable search for the short patterns in genomes, also known as the exact string matching problem.

Exact string matching is a well-developed area in computer science. The traditional definition of this problem is the following: *Given a string P of size n called the pattern and the longer string T of size m called the text, the exact matching problem is to find all occurrences, if any, of pattern P in text T.* Many algorithms have been previously developed (overviews provided in [11,12]) based on the idea of precomputing.

Some algorithms, such as Rabin-Karp [13], Boyer-Moore [14], and Knuth-Morris-Pratt [15] apply precomputing to the pattern. The memory usage in such approaches is not very extensive, and if $n<<m$ the time is proportional to the length of the text or sequence: $O(m)$. Other approaches are based on the idea of precomputing the text [16-18]. While such algorithms are more memory-expensive and the estimation of time required for precomputing is $O(m)$, string matching can be done extremely fast and depends only of the pattern size: $O(n)$. Here it is necessary to note that there is some variation in the problem definition. In some cases one needs to find any occurrence instead of all occurrences of the pattern in the text; nevertheless the problem has the same notation.

To make an optimal choice as to which algorithm is better, one has to take into account additional parameters for the problem under consideration. In fact, it is rare to match just one pattern against one sequence. In many cases a finite set of k sequences and a finite set of l patterns are considered necessitating kl searches to be performed to find the occurrence of all patterns in all sequences.

Another important parameter to consider is the average ratio $r=n/m$. Assume one needs to compare the Knuth-Morris-Pratt [15] and the Suffix Tree [16-18] algorithms. Taking into account that the precomputing time will be $O(nl)$ for Knuth-Morris-Pratt and $O(km)$ for Suffix Tree, the running time estimations are $O(nl+lkm)$ and $O(km+knl)$ respectively. If memory usage (which in some cases can be critical for the Suffix Trees) is not a concern, one can easily estimate which algorithm would be preferable based on the parameters of a real given problem. The estimates above are typical for these two classes of algorithms. In general, one needs linear time for precomputing; this allows performing the search in linear time.

In the problem of finding all possible n-mers, $n=10^1$-10^2 (length typical for microarray probes and PCR primers), in a sequences on the order of 10^3-10^9 (length typical for a complete genome), a very specific kind of the exact matching problem is encountered. Such a problem needs to be solved if, for example, one needs to perform a comparative statistical analysis of the presence of all possible n-mers of the length from 7 to 25 “characters” (bases, b) in genomes of more than 250 microbial, viral and multicellular organisms varying from 0.32Kb (*Cereal yellow dwarf virus-RPV satellite RNA* NC_003533) to 2.87 Gb (human). Such studies are necessary not only to explore statistical characteristics of genomes, but also to estimate limitations of such promising technologies as microarray analysis for microbial, viral, and even human recognition. According to our knowledge, no such studies have been performed for $n>11$ due to the rapid increase in computational difficulties that appear because the total number of different n-mers grows exponentially fast when n increases, 4^n. The operation count estimation $O(4^n km)$ for existing algorithms which precompute patterns and $O(4^n kn)$ for algorithms which precompute sequence is simply unacceptable.

II. Results

A. *Calculation of the presence of all possible n-mers in a given text*

There are four specific points that characterize our problem versus previous algorithmic studies:

1. Relatively short pattern lengths: $max(N)=25$;
2. Only 4 characters in the alphabet. All DNA sequences contain only 4 nucleotides A (*adenine*), T (*thymine*), C (*cytosine*), and G (*guanine*);
3. For each value of n, the search has to be performed simultaneously for all possible (4^n) n-mers.
4. Regarding each n-mer (pattern), we are interested only in its presence/absence in each genome (text), or in some rare cases how many times it appears in genome.

To take advantage of the specifics of our problem, in particular the fact that we can perform the calculation for all n-mers simultaneously for each given value of n (which is relatively small), we decided to employ an approach similar to the one used in the well-known counting-sort algorithm (for example see [12]). The basic idea is to set in correspondence to each of 4^n n-mers a particular element of counting array A and define a procedure to convert the n-mer character sequence to the index of an element in such an array. One could view such an array as an extreme case of the hash table and the procedure to convert the sequence to the index as the hash function. Assume we need to calculate how many different n-mers are present in the text T.

ALGORITHM 1.

```
NUMBER-OF-N-MERS-IN-TEXT(T,n)
1   for i ← 0 to 4^n-1
2       do A[i] ← 0
3   sum ← 0
4   for j ← 0 to length [T]-n
5       do index=CONVERT_TO_INTEGER_VALUE(T(j, j+n-1))
6           if A[index]=0
7           then sum ← sum+1
8                   A[index]=1
9   return sum
```

In this algorithm, $T(j_1, j_2)$ stands for a substring of the string T, starting in position j_1 and ending in position j_2. The function CONVERT_TO_INTEGER_VALUE(s) is needed to convert the string s of length n, which in our case is created using only a 4 character alphabet, to a unique integer value corresponding to an index in the array A. In fact, if we assign to each character of our alphabet values from 0 to 3, each string can be interpreted as an integer value in a base-4 number system. Using a naïve algorithm this function can be implemented to have a running time $O(n)$. We can utilize the fact that only 2 bits are needed for each character and that we read the text T sequentially, so that each string $T(j_1, j_2)$ already contains n-1 elements of the next one $T(j_1+1, j_2+1)$. Thus, the function CONVERT_TO_INTEGER_VALUE() can be implemented using simple binary shift operations such that execution can be in $O(1)$ time.

The overall running time estimation for ALGORITHM 1 is then $O(4^n+m)$. If both text T and the counting array A can be placed in memory, it takes only seconds on a regular PC (1 GHz clock) to calculate, for example, how many 15-mers are present in both complementary DNA sequences of the *Mycobacterium tuberculosis* H37Rv (NC_000962), a 4,411,529 bp genome.

The biggest concern with this algorithm is memory usage, especially because of the size of the counting array. The necessary size of A can be defined based on the value of n and the characteristics of the problem. If we are interested only in the presence or absence on n-mers in a text, we need only 1 *bit* for each element to keep track of whether the corresponding pattern has already appeared in the text (A[…]=1) or not (A[…]=0). For example, 17-mers will require 4^{17} = 17,179,869,184 *bits* = 2 GB (gigabyte) of memory. This size of RAM can be placed in practically any PC or workstation. This is, however, a limiting factor for larger n-mers, say up to 20 in length.

If the required memory is not available or is inconvenient, we can decompose the counting array A. The idea is a simple divide and conquer strategy; we divide our n-mer into two parts: a *prefix* of size n_1 and a *suffix* of size n-n_1. One then creates the array A to track the appearance of (n-n_1)-mers in the suffixes and summarizes the results for all prefixes. This is shown in ALGORITHM 2.

ALGORITHM 2.

NUMBER-OF-N-MERS-IN-TEXT-2*(T, n, n_1)*

```
1   sum ← 0
2   for prefix ←0 to $4^{n_1}$ -1
3       do for i ← 0 to $4^{n-n_1}$ -1
4           do A[i] ← 0
5       for j ← 0 to length [T]-n
6           if (prefix=CONVERT_TO_INTEGER_VALUE(T(j, j+$n_1$-1))
7           then index=CONVERT_TO_INTEGER_VALUE(T(j+$n_1$, j+n-1))
8               if A[index]=0
9               then sum ← sum+1
10                  A[index]=1
11  return sum
```

The operation count of ALGORITHM 2 is $O(4^{n_1}(4^{n-n_1}+m))= O(4^n+4^{n_1}m)$. In fact, in the case mentioned above, if the available computer has more than 2.5 GB RAM, which means it can handle an array A necessary to keep track of 17-mers, it takes only about 4 times longer to count all 18-mers and 64 times longer to count all 20-mers. Using ALGORITHM 2, the number of 20-mers in the above mentioned *Mycobacterium tuberculosis* H37Rv genome can be calculated in less than a minute on a 1 GHz CPU. Similar calculations were also performed in less than one hour for the 20-mers in all chromosomes of the human genome (available from NCBI at http://www.ncbi.nih.gov/). It is important to note that increasing the prefix size n_1 in this algorithm causes an exponential increase of the run time. At the same time, because the algorithm can be easily parallelized (using for example the "**for** *prefix*" loop in line 2), the required run time can be essentially linearly decreased by increasing the number of processors. Such scaling makes the large-scale application of this algorithm very attractive.

For our problem area, another interesting task was to find how many *n*-mers are present in only one of two given texts T_1 and T_2 and how many of these *n*-mers belong to both texts. Three different approaches to this problem were implemented and tested in our calculations:

1. Run ALGORITHM 1 or ALGORITHM 2 three times to calculate the value of *sum* for text T_1 (sum_1), text T_2 (sum_2), and for both texts simultaneously (sum_{12}). Then sum_1-sum_{12} will be number of *n*-mers present only in T_1, sum_2-sum_{12} will be number of *n*-mers present only in T_2, and of course sum_{12} will be number of *n*-mers present in both texts;
2. Use two arrays, A_1 and A_2, to keep track of the information regarding the presence/absence of *n*-mers in each text separately;
3. Use 2 bits for each element of the array *A* and use them separately to track the presence/absence of *n*-mers in each text.

B. *Calculation of the frequency of the presence of all possible n-mers*

Our approach, similar to that described in ALGORITHMS 1 and 2 can also be used to calculate the actual number of *n*-mers present in the text. A rough or naïve algorithm can be created by employing the counting array *A* to store the number of appearances of each *n*-mer.

ALGORITHM 3.
FREQUENCY-OF-N-MERS-IN-TEXT-2*(T, n)*

```
1   for i ← 0 to $4^n$-1
2       do A[i] ← 0
3   for j ← 0 to length [T]-n
4       do index=CONVERT_TO_INTEGER_VALUE(T(j, j+n-1))
5           A[index]= A[index]+1
6   return A
```

As a result, ALGORITHM 3 produces array *A* with a run time of $O(4^n+m)$. In contrast to the cases discussed earlier, the array *A* contains integer values for the number of times each *n*-mer is present in the text or sequence. The required memory in this case is much larger: instead of one *bit* we will need from 1 to 8 *bytes* to keep integer values. For example, if 4 *bytes* are used to store each integer value, the necessary memory for the case of 14-mers will be 1 GB.

To make this algorithm more efficient it is necessary to take a more careful look at the structure of the original data and the produced results. Tables I and II list how many *n*-mers appear only once, more than once, and never appear in the genome of *Mycobacterium tuberculosis* H37Rv and in the human genome. As one can see, if the text size *m* is less than 4^n, practically all *n*-mers are found to be present. However, for $4^n >> m$ a very different situation arises: (1) the majority of *n*-mers are simply absent in the text and (2) the number of *n*-mers present in that text just once far exceeds the number of *n*-mers present more than once. Although the memory for n>12 was previously of concern, Tables I and II lead us to conclude that for such numbers the majority of array *A* will be occupied by zeros (sparse). In fact, the number of different *n*-mers cannot be larger than m-n. (The number of *n*-mers which appear more than once is not expected to be larger than $(m$-$n)/2$.) Thus, to conveniently keep information about all present *n*-mers we need two arrays: one (*R*) to keep the "sequence" of *n*-mers and another (*Q*) to keep the integer number of times of appearance. Of course it would be even easier to place both the *n*-mer and the number of its appearances in one data structure. The estimation of the memory usage is straightforward. For the array *R* it is $r_n(m\text{-}n) \cong r_n m$, where r_n is the size reserved for the *n*-mer. For the array *Q* it is $r_{integer}(m\text{-}n) \cong r_{integer}m$, where $r_{integer}$ is the size reserved for the integer variable. Such a size is manageable given the parameters of our problem area. For example using 5 bytes for integers and 2 bits for all 20-mers in the sequence on the order of 5,000,000 (approximate size of the complete genome of *Mycobacterium tuberculosis* H37Rv), we will need in the worst-case only 52.5MB RAM (size of *R* + size of *Q*). In this estimation, both strands of the genomic sequence are considered: *R*= (2bits/byte)∗(5Mb*2)=2.5MB and *Q*= (5Mb*2) ∗5bytes=50MB. From Tables I and II, one can see that the number of *n*-mers present in real genomes is much less than the hypothetical worst case.

The following algorithm can be introduced to generate arrays *R* and *Q*:

ALGORITHM 4.
FREQUENCY-OF-N-MERS-IN-TEXT-2*(T, n)*

```
1   for $i \leftarrow 0$ to $4^n-1$
2       do $A[i] \leftarrow 0$
3   $sum \leftarrow 0$
3   for $j \leftarrow 0$ to length $[T]-n$
4       do $index$=CONVERT_TO_INTEGER_VALUE($T(j, j+n-1)$)
5           $A[index]= A[index]+1$
7           if $A[index]= 1$
8           then $sum \leftarrow sum +1$
9   Use value of $sum$ to reserve memory for arrays $R$ and $Q$ of size $sum$
10  $counter \leftarrow 0$
11  for $i \leftarrow 0$ to $4^n-1$
12  if $A[i]=1$
13  then    $R[counter]= i$
14          $Q[i]= A[i]$
15          $counter \leftarrow counter + 1$
16  return $sum$
```

The ALGORITHM 4 produces two synchronized arrays *R* and *Q* of dynamically defined size *sum*. The total run time estimation of ALGORITHM 4 is $O(4^n+m+4^n) = O(4^n+m)$. Because the array *R* of the present *n*-mers is created sorted, the time estimation to check the presence of any *n*-mer in such an array requires only logarithmic time $O(log(sum))$ such that the worst case would be $O(ln(m))$.

It is important to mention that because these two arrays (*R* and *Q*) represent the set of all *n*-mers present in the original sequence, it is reasonable to store them into a data structure which can then be used for future analysis. Traditional set operations, e.g. union, intersection, and subtraction, can be introduced and implemented on such objects with linear run time $O(sum_1+sum_2)$, where sum_1 and sum_2 are the sizes of the arrays in the two data structures under consideration.

TABLE I
PRESENCE/ABSENCE STATISTICS FOR MYCOBACTERIUM TUBERCULOSIS H37RV (NC_000962)

***n*-mer size**	**Number of different *n*-mers (4^n)**	**Number of absent *n*-mers**	**Number of *n*-mers present just once**	**Number of *n*-mers present 1+ times**
7	16,384	0	2	16,382
8	65,536	152	318	65,066
9	262,144	9,234	11,826	241,084
10	1,048,576	186,370	147,776	714,430
11	4,194,304	1,918,866	862,428	1,413,011
12	16,777,216	12,415,707	2,624,671	1,736,838
13	67,108,864	60,778,599	4,938,895	1,391,370
14	268,435,456	260,837,795	6,777,575	820,086
15	1,073,741,824	1,065,526,021	7,798,705	417,098
16	4,294,967,296	4,286,498,557	8,245,983	222,756
17	17,179,869,184	17,171,299,773	8,424,979	144,432
18	68,719,476,736	68,710,863,745	8,499,559	113,432
19	274,877,906,944	274,869,272,195	8,534,649	100,100
20	1,099,511,627,776	1,099,502,979,652	8,555,193	92,931

Calculation was performed using both (original and complementary strands, each 4,411,529 b) DNA sequences.

TABLE II
PRESENCE/ABSENCE STATISTICS FOR THE HUMAN GENOME

***n*-mer size**	**Number of different *n*-mers (4^n)**	**Number of absent *n*-mers**	**Number of *n*-mers present just once**	**Number of *n*-mers present 1+ times**
7	16,384	0	0	16,384
8	65,536	0	0	65,536
9	262,144	0	0	262,144
10	1,048,576	0	0	1,048,576
11	4,194,304	42	324	4,193,938
12	16,777,216	42,501	91,146	16,643,569
13	67,108,864	2,382,096	2,642,582	62,084,186
14	268,435,456	41,634,971	30,411,367	196,389,118
15	1,073,741,824	410,828,287	166,998,278	495,915,259
16	4,294,967,296	2,717,880,983	671,192,253	905,894,060
17	17,179,869,184	14,452,040,667	1,790,043,813	937,784,704
18	68,719,476,736	65,147,397,575	2,881,849,256	690,229,905
19	274,877,906,944	270,850,664,602	3,538,156,028	489,086,314
20	1,099,511,627,776	1,095,257,688,530	3,866,031,543	387,907,703

Calculation was performed using both (original and complementary strands, each 2,874,736,094 b) DNA sequences.

III. CONCLUSIONS

Herein we have presented a novel group of algorithms for the problem of finding the appearances of all possible patterns of size *n* in a text or sequence of size *m*. The operation count estimation $O(4^n+m)$ makes it possible to accomplish the calculation for the statistics of the presence of all possible 7-20-mers in more than 250 genomes, including the human genome. By using the concept of a counting array and parallel processing, the length of *n*-mers considered can be increased.

REFERENCES

[1] R. Frislage, M. Berceanu, Y. Humboldt, M. Wendt, and H. Oberender, "Primer design for a prokaryotic differential display RT-PCR," *Nucleic Acids Res.*, vol. 25, no. 9, pp. 1845-1850, 1997.

[2] R. Fislage, "Differential display approach to quantitation of environmental stimuli on bacterial gene expression," Electrophoresis, vol. 19, no. 4, pp. 1845-1850, 1998.

[3] E.M. Southern, "DNA microarrays. History and overview," *Methods Mol. Biol.*, vol. 170, 2001.

[4] R. Nussinov, "Doublet frequencies in evolutionary distinct groups," *Nucleic Acids Res.*, vol. 12, no. 3, pp. 1749-1763, 1984.

[5] S. Karlin and I. Ladunga, "Comparisons of eukaryotic genomic sequences," *Proc. Natl. Acad. Sci. USA*, vol. 91, no. 26, pp. 12823-12836, 1994.

[6] S. Karlin, J. Marazek, and A.M. Campbell, "Compositional Biases of Bacterial Genomes and Evolutionary Implications," J. Bacteriol., vol. 179, no. 12, pp. 3899-3913, 1997.

[7] H. Nakashima, K. Nishikawa, and T. Ooi, "Differences in Dinucleotide Frequencies of Human, Yeast and Escherichia coli Genes," *DNA Res.*, vol. 4, no. 3, pp. 186-192, 1997.

[8] H. Nakashima, K. Nishikawa, and T. Ooi, "Genes from Nine Genomes Are Separated into Their Organisms in the Dinucleotide Composition Space," *DNA Res.*, vol. **5**, no. 5, pp. 251-259, 1998.

[9] P.J. Deschavanne, A. Giron, J. Vilain, G. Fagot, and B. Fertil, "Genomic Signature: Characterization and Classification of Species Assessed by Chaos Game Representation of Sequences," *Mol. Biol. Evol.*, vol. 16, no. 10, pp. 1391-1399, 1999.

[10] R. Sandberg, G. Winberg, C.I. Bränden, A. Kaske, I. Ernberg, and J. Cöster, "Capturing whole-genome characteristics in short sequences using a naïve Bayesian classifier," *Genome Res.*, vol. 11, no. 8, pp. 1404-1409, 2001.

[11] D. Gusfield, *Algorithms on strings, trees, and sequences*, Cambridge University Press, New York, NY, 1997.

[12] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 1990.

[13] R.M. Karp and M.O. Rabin, "Efficient randomized pattern-matching algorithms," Aiken Computation Laboratory, Harvard University, Technical Report TR-31-81, 1981.

[14] R.S. Boyer and J.S. Moore, "A fast string-searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762-772, 1977.

[15] D.E. Knuth, J.H. Morris, and V.R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323-350, 1977.

[16] P. Weiner, "Linear pattern matching algorithms," in *Proceedings of the 14th IEEE Symp. on Switching and Automata Theory*, University of Iowa, Iowa, 1973, pp. 1-11.

[17] E.M. McCreight, "A space-economical suffix tree construction algorithm, *J. ACM.*, 23, no. 2, pp. 262-72, 1976.

[18] E. Ukkonen, "On-line construction of suffix-trees," *Algorithmica*, vol. 14, no. 3, pp. 249-60, 1995.