Efficient Algorithms for Counting and Reporting Segregating Sites in Genomic Sequences

Manolis Christodoulakis[1]

G. Brian Golding[2]

Costas S. Iliopoulos[1]

Yoan José Pinzón Ardila[3]

William F. Smyth[4]

[1]King's College London, Department of Computer Science
Strand, London WC2R 2LS, UK
{manolis,csi}@dcs.kcl.ac.uk

[2]Department of Biology, McMaster University
Hamilton L8S 4K1, Canada
golding@mcmaster.ca

[3]Pontificia Bolivariana University, Faculty of Informatics
Km 7 Via Piedecuesta, Colombia
yoan@pinzon.co.uk

[4]Department of Computing & Software, McMaster University
Hamilton L8S 4K1, Canada
Department of Computing, Curtin University
Perth WA 6845, Australia
smyth@mcmaster.ca

Abstract

The number of segregating sites provides an indicator of the degree of DNA sequence variation that is present in a sample, and has been of great interest to the biological, pharmaceutical and medical professions. In this paper we first provide linear- and expected-sublinear-time algorithms for finding all the segregating sites of a given set of DNA sequences. We also describe a data structure for tracking segregating sites in a set of sequences, such that every time the set is updated with the insertion of a new sequence or removal of an existing one, the segregating sites are updated accordingly without the need to re-scan the entire set of sequences.

Keywords: segregating sites, single nucleotide polymorphisms (SNPs).

1 Introduction

The number of segregating sites is defined as the number of homologous DNA positions that differ in a sample of m sequences. This number therefore, provides an indicator of the degree of DNA sequence variation that is present in a sample.

This quantity has long been of theoretical population genetic interest. It is directly modelled using what has been termed an *infinite sites* model where every mutation is created at a new unique site. These models date back to 1967 (Karlin and McGregor, 1967) and have since been extensively developed to estimate population size, mutation rate, migration rates, natural selection, and ages of polymorphisms (Innan *et al.*, 2005; Klein *et al.*, 1999; Perlitz and Stephan, 1997; Fu, 1996).

Segregating sites are also of interest to the pharmaceutical and medical professions. The number and the location of polymorphisms within humans and other groups of organisms are critical to determining the potential differences in reactions of individuals to medicines and treatments. Since polymorphisms occur every 100 to 300 bases along the three billion base human genome, they are crucial markers to map human genetic diseases. As a result, single nucleotide polymorphisms (SNPs) are searched for in a systematic manner (*cf.* the SNP consortium[1]).

With the promise of entire genomic complements being determined within a few hours (Margulies *et al.*, 2005) and with entire genome scans desired for SNPs (Syvanen, 2005), the need to quickly determine the number of segregating sites will become ever more urgent. In future, it can be anticipated that these calculations will be required over hundreds of sequences whose length is in the tens of thousands.

Segregating sites are formally defined for *within population* level phenomena but

[1] see `http://snp.cshl.org` for more information

the infinite sites model and its utility to measure genetic variation are useful in other contexts as well. The recent completion of a large number of organelle genomes chosen in a systematic fashion is just one such example. This data records the pattern of sequence variants within a hundred mitochondrial genomes (Miya *et al.*, 2003). In this case, the data consists of approximately 16,000bp from different teleostean fish. To repeatedly query the level and pattern of variation in different subgroups requires efficient methodology. Another large scale biological project that will require an efficient algorithm is the barcode of life project (*e.g.* (Hebert *et al.*, 2003)). This project aims to sequence the *cytochrome oxidase subunit 1* gene from millions of animal species. Again, the variation within a species, within a genus, within different groups of genera will provide useful biological insights into the evolution of these taxa. A tool to permit the addition and removal of sequences in these queries will allow an interactive ability to perform exploratory analyses.

In this paper we address the problem of locating the positions that constitute segregating sites in a given set $\mathcal{S}$ of equal-length sequences. In addition, we create a data structure that stores adequate information about the segregating sites of $\mathcal{S}$, such that if one adds new sequences to $\mathcal{S}$ or removes sequences from $\mathcal{S}$, the number of segregating sites is efficiently resolved accordingly. In this way, one avoids the need to recompute (from scratch) the number of segregating sites in $\mathcal{S}$, every time $\mathcal{S}$ is updated.

The paper is organised as follows. In Section 2 we introduce some necessary notation and state the problems of finding segregating sites within a set of genomic sequences. Section 3 describes (two variants of) a very fast algorithm to locate all segregating sites positions. Section 4 contains an alternative algorithm to identify segregating sites with the enhanced advantage that whenever the set of sequences is updated, the segregating sites are updated accordingly. In Section 5, experimental results are presented and some implementation issues are briefly discussed. Finally, Section 6 contains our concluding remarks.

2 Definitions

A *string* is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The length of a string s is denoted by $|s|$. The *empty string*, the string of length zero, is denoted by ε. The i-th symbol of a string s is denoted by $s[i]$. A string y is a *substring* of s if $s = uyv$, where $u, v \in \Sigma^*$. We denote by $s[i..j]$ the substring of s that starts at position i and ends at position j.

Consider a set of strings $\mathcal{S} = \{s_1, \dots, s_m\}$, where $|s_i| = n \; \forall\, i \in [1..m]$. A position $1 \leq j \leq n$ is then called a *segregating site* iff there exists at least one pair of indices, (i, i'), such that $s_i[j] \neq s_{i'}[j]$.

In Fig. 1 we give a simplistic example of a set $\mathcal{S}$ containing 5 DNA sequences, each of length 14 ($m = 5, n = 14$). To find the segregating sites we look for those sites which are *variable* in the data in terms of different nucleotides; we find them at positions 4, 8 and 10. For instance, site 4 is a segregating site since $s_1[4] \neq s_2[4]$. In contrast, positions 1–3, 5–7, 9 and 11–14 are not segregating sites because all of the nucleotides are the same at each of those sites.

Figure 1

The problems we address in this paper are formally defined as follows.

Problem 1. Given a set of m strings $\mathcal{S} = \{s_1, \dots, s_m\}$, where $|s_i| = n \; \forall\, i \in [1..m]$, locate all the positions $1 \leq j \leq n$ that constitute segregating sites in $\mathcal{S}$.

Problem 2. Given a set of m strings $\mathcal{S} = \{s_1, \dots, s_m\}$, where $|s_i| = n \; \forall\, i \in [1..m]$, whose segregating sites have already been computed, and a set of ℓ strings $\mathcal{S}' = \{s'_1, \dots, s'_\ell\}$, where $|s'_i| = n \; \forall\, i \in [1..\ell]$, efficiently find the positions $1 \leq j \leq n$ that constitute segregating sites in $\mathcal{S} \cup \mathcal{S}'$ or $\mathcal{S} \backslash \mathcal{S}'$.

3 Locating Segregating Sites

In this section we consider Problem 1. First we present a fairly simple and fast algorithm that identifies all the segregating sites in a given set of sequences in time linear with respect to the input size. We then show a variant of this algorithm, whose expected running time is sublinear for sets of sequences that contain many segregating sites (*i.e.* for the *dense* case).

3.1 A Simple and Fast Algorithm

Let $\mathcal{S} = \{s_1, \ldots, s_m\}$ be the set of strings to be analysed, where $|s_i| = n \; \forall \, i \in [1..m]$. We will use a binary vector ss of length n to indicate the positions where there is a segregating site, *i.e.* $ss[j] = 1$ iff site located at j is segregating, and $ss[j] = 0$ otherwise. At the beginning this vector should contain only zeros (no segregating sites have been found). The algorithm works as follows. All (but the first) sequences are scanned from left to right. During the scan of the i-th sequence, if a position j has not already been marked as a segregating site, then the symbol $s_i[j]$ is checked for discrepancies with the corresponding symbol in the first sequence $s_1[j]$. If they are different, a new segregating site has been discovered and we set $ss[j] = 1$, otherwise we leave everything unchanged. The complete algorithm is summarized in Algorithm 1.

Note that this algorithm first checks if $ss[j] = 0$ to know if position j is a segregating site already (*see* line 5). If it is ($ss[j] = 1$), then there is no need to check what type of symbol $s_i[j]$ is, because it wouldn't make a difference to the current status of site j. If it is not ($ss[j] = 0$), then that implies that all previous sequences $\{s_1, \ldots, s_{i-1}\}$ contain the same symbol at position j, and therefore the new symbol $s_i[j]$ needs to be checked against position j of any of the sequences already scanned, *e.g.* $s_1[j]$. To understand

Algorithm 1 Locate all the segregating sites in a set of strings $\mathcal{S}$

Input: Set of strings $\mathcal{S} = \{s_1, \ldots, s_m\}$, each of size n

Output: Positions of segregating sites in $\mathcal{S}$

```
procedure SS-Alg1($\mathcal{S}$)
    $ss[1..n] \leftarrow \{0, 0, \ldots, 0\}$
    for $i = 2$ to $m$ do
        for $j = 1$ to $n$ do
            if $ss[j] = 0$ and $s_i[j] \neq s_1[j]$ then
                $ss[j] \leftarrow 1$
    return the positions of 1's in $ss$
```

this issue better, we present the following simple example (*see* Fig. 2). Consider a set of m sequences $\mathcal{S}$ whose first two sequences s_1 and s_2 have already been processed, and a third sequence s_3 is now being processed. Note that, for every position that has not already been marked as a segregating site, we need to compare the current nucleotide against the corresponding nucleotide in s_1. For instance, at position 1 the symbol $s_3[1]$ is compared against $s_1[1]$; since they are equal, position 1 remains unmarked. At position 6, on the contrary, $s_3[6] \neq s_1[6]$, thus position 6 is marked as a segregating site ($ss[6] = 1$). Notice though that when column 3 is processed, there is no need to check $s_3[3]$ because that position has already been marked as a segregating site.

Figure 2

The running time of Algorithm 1 is clearly $\Theta(mn)$: all the sequences are scanned only once, spending a constant amount of time at each position of every sequence. The space requirements of the algorithm are limited to $\Theta(n)$, the space required by vector ss.

3.2 An Expected-sublinear-time Algorithm

From close observations of the way Algorithm 1 works, we know that the larger the number of segregating sites and the earlier the sites are identified as segregating, the more the symbols that do not need to be checked. For instance, in the set of sequences depicted in Fig. 3, approximately 19% of the symbols are not checked (the symbols displayed in gray).

Figure 3

Although Algorithm 1 does not check the symbols at those positions that have been identified as segregating sites, it would nonetheless check vector *ss* to decide whether the positions are segregating sites or not, as a result, the algorithm performs precisely $(m - 1)n$ iterations. However, for those cases when a large number of segregating sites is present (*i.e.* the *dense* case), it seems to be beneficial to completely skip those positions and only process non-segregating-site positions.

To this end, a simple linked list can be used to maintain the list of positions to be processed. Every time a new segregating site is discovered, the corresponding position will be removed from the list. Then for every subsequent sequence, only the positions included in this list will ever be processed. Algorithm 2 describes these ideas in more detail.

In the worst case, Algorithm 2 will yet again need $O(mn)$ time to process a set of m sequences each of length n. The expected running time though will be sublinear with respect to the input size, since as soon as a position is identified as a segregating site it will no longer be visited again. In practice (as we shall see in Section 5) only when the number of segregating sites is very small, the extra time consumed in maintaining the list of positions, exceeds the time saved by the skipped positions, confirming that this algorithm is good for samples containing more than a small number of segregating sites.

Algorithm 2 Locate all the segregating sites in a set of strings $\mathcal{S}$. Algorithm for dense segregating sites.

Input: Set of strings $\mathcal{S} = \{s_1, \ldots, s_m\}$, each of size n

Output: Positions of segregating sites in $\mathcal{S}$

```
procedure SS-Alg2($\mathcal{S}$)
    $ss[1..n] \leftarrow \{0, 0, \ldots, 0\}$
    $P = \{1, 2, \ldots, n\}$
    for $i = 1$ to $m$ do
        for all $j \in P$ do
            if $s_i[j] \neq s_1[j]$ then
                $ss[j] \leftarrow 1$
                $P \leftarrow P \backslash \{j\}$
    return the positions not in $P$
```

4 Locating and Updating Segregating Sites

In this section we extend our techniques to tackle both Problems 1 and 2 (as described in Section 2). We start by describing a data structure capable of holding enough information about the segregating sites of a set of sequences $\mathcal{S}$, such that any update to the set of sequences $\mathcal{S}$ can be incorporated, avoiding thus the need to recompute from scratch the positions of segregating sites of the updated set. We then present the algorithms that handle this new data structure.

4.1 The SSCounter Data Structure

Let $\mathcal{S} = \{s_1, \ldots, s_m\}$ be the set of strings to be analysed, where $|s_i| = n \; \forall \, i \in [1..m]$. Our data structure, which we will hereafter call the `SSCounter`, will maintain the following information:

1) An integer k representing the number of strings that occur at any time in $\mathcal{S}$.

2) The (common) length n of each string in $\mathcal{S}$.

3) A binary vector ss of length n that will indicate the positions that are segregating sites; that is, $ss[j] = 1$ iff position j is a segregating site, and $ss[j] = 0$ otherwise, for all $1 \leq j \leq n$.

4) For every distinct symbol $\sigma \in \Sigma$, a vector $c_\sigma = c_\sigma[1..n]$ will hold counters that keep the number of strings $s_i \in \mathcal{S}$ that contain the symbol σ at each position $1 \leq j \leq n$; *i.e.* $c_\sigma[j] = \nu$ iff the number of strings s_i for which $s_i[j] = \sigma$ is precisely ν.

From the above definitions, the following fact becomes immediately apparent. Any position $1 \leq j \leq n$ will be a segregating site iff there exists at least one symbol

$\sigma \in \Sigma$ such that $0 < c_\sigma[j] < m$, or equivalently, there is no symbol $\sigma \in \Sigma$ such that $c_\sigma[j] = m$. Naively, the evaluation of any position j requires the evaluation of (at most) $|\Sigma|$ counters for that position. However, this step can be efficiently computed by storing one extra piece of information:

5) A vector of counters $ac[1..n]$, which for each position j indicates the number of non-zero (active) counters; that is, $ac[j] = \ell$ iff there exist precisely ℓ symbols $\sigma_1, \sigma_2, \ldots, \sigma_\ell$ such that $c_\sigma > 0$ for all $\sigma \in \{\sigma_1, \sigma_2, \ldots, \sigma_\ell\}$.

Figure 4

An example of an SSCounter data structure is shown in Fig. 4(b), which corresponds to the set of strings shown in Fig. 4(a). Positions 1, 3, 6 and 7 are segregating sites, since in each of those positions there are more than one symbols σ for which the counter $c_\sigma > 0$.

Clearly, the space requirements of an SSCounter is $\Theta(|\Sigma|n)$. This will normally be significantly less than the space required by the set $\mathcal{S}$ itself, since usually the number of strings m is much larger than the size of the alphabet Σ.

4.2 Inserting New Sequences into $\mathcal{S}$

Procedure Insert (*see* Algorithm 3) handles insertions. For each new sequence s' inserted into $\mathcal{S}$, the algorithm works as follows. s' is scanned from left to right, and for every position j, the counter of the symbol occurring at $s'[j]$ is increased; in other words, if $s'[j] = \sigma$, then the counter $c_\sigma[j]$ is increased by 1 to denote that one additional string contains the symbol σ at position j. Next, the value of counter $c_\sigma[j]$ is checked and if it equals 1 then this corresponds to a new non-zero counter and thus $ac[j]$ is increased by 1. Finally, if $ac[j] = 2$, the position has just become a segregating site and has to be marked as such.

Algorithm 3 Insert a string s' into $\mathcal{S}$ and update the `SSCounter`

Input: The `SSCounter` of a set of strings $\mathcal{S}$, and a string s' of size n

Output: The `SSCounter` of $\mathcal{S} \cup \{s'\}$

```
procedure INSERT(the SSCounter of $\mathcal{S}$, string $s'$)
    $k \leftarrow k + 1$
    for $j = 1$ to $n$ do
        $\sigma \leftarrow s'[j]$
        $c_\sigma[j] \leftarrow c_\sigma[j] + 1$
        if $c_\sigma[j] = 1$ then
            $ac[j] \leftarrow ac[j] + 1$
            if $ac[j] = 2$ then
                $ss[j] \leftarrow 1$
    return the positions of 1's in $ss$
```

Algorithm 3 performs a single scan through the newly inserted sequence s'. The amount of time spent at each position j depends on the type of alphabet:

— for *indexed* alphabets, the counter c_σ that corresponds to the current symbol σ, can be identified in $O(1)$, thus the time spent at each position j is $O(1)$.

— for general ordered alphabets, a binary search is required to identify c_σ, thus the time spent at each position is $O(\log |\Sigma|)$.

Therefore, the overall running time of Algorithm 3 is $\Theta(n)$ for indexed alphabets, and $\Theta(n \log |\Sigma|)$ for ordered alphabets. In biological applications, of course, the DNA alphabet or the amino-acid alphabet is used, which will be indexed.

Algorithm 4 Remove a string from an `SSCounter`

Input: The `SSCounter` of a set of strings $\mathcal{S}$, and a string s' of size n

Output: The `SSCounter` of $\mathcal{S}\backslash\{s'\}$

```
procedure REMOVE(the SSCounter of $\mathcal{S}$, string $s'$)
    $k \leftarrow k - 1$
    for $j = 1$ to $n$ do
        $\sigma \leftarrow s'[j]$
        $c_\sigma[j] \leftarrow c_\sigma[j] - 1$
        if $c_\sigma[j] = 0$ then
            $ac[j] \leftarrow ac[j] - 1$
            if $ac[j] = 1$ then
                $ss[j] \leftarrow 0$
    return the number of 1's in $ss$
```

4.3 Removing Sequences from $\mathcal{S}$

Similarly, procedure Remove (*see* Algorithm 4) handles removals of strings from $\mathcal{S}$. When a sequence s' is removed from $\mathcal{S}$, s' is scanned from left to right, and for every position j, the counter $c_\sigma[j]$ is decreased by one, where $\sigma = s'[j]$. Next, we have to check whether that counter equals 0. If it does, and also position j was marked as a segregating site, then one needs to re-evaluate whether position j is still a segregating site. It will still be 1 iff at least two symbols occur at column j; *i.e.* iff $ac[j] > 1$.

Likewise, as in procedure Insert, the time complexity of Remove is also $\Theta(n)$ for indexed alphabets, and $\Theta(n \log |\Sigma|)$ for ordered alphabets.

5 Experimental Results

We implemented all of the algorithms presented in the paper using C++ and conducted a series of experiments in order to test their performance. The datasets we used consisted of both randomly generated sequences, tailored to the needs of the specific experiments we run, and biological sequences[2].

First, we tested how the running time of the algorithms scales with respect to the number of the segregating sites present in the dataset. In Fig. 5(a) we considered a varying number of sequences ($10 \leq m \leq 100$) of length $n = 10000$, which contained only a small number of segregating sites, while in Fig. 5(b) again a varying number of sequences of length $n = 10000$ was considered but this time the number of segregating sites was considerably larger.

Figure 5

From the graph in Fig. 5(a) we can see that the running times of procedures SS-Alg1 and SS-Alg2 are practically identical when the number of segregating sites is small, while Fig. 5(b) shows that as the number of segregating sites increases, the running time of SS-Alg2 is reduced, by a small fraction in this case. The amount of this reduction depends exclusively on the number of comparisons avoided by SS-Alg2, which itself depends on both the number of segregating sites and how early these segregating sites are identified; the earlier the better. Finally, we observe that the running time of Insert is practically independent of the number of segregating sites. This is true, given that this algorithm performs almost the same amount of work for each position of any input sequence, regardless of whether it is a segregating site or not.

Secondly, we compared the running time of procedure Insert against that of Re-

[2] All the sequences were given in the FASTA format.

move. An existing `SSCounter`, `ssc`, was passed to the two procedures, and then the same sets of sequences were inserted to/removed from `ssc`, using Insert and Remove, respectively. As can be seen in Fig. 6, the two algorithms perform identically in terms of running time for the values chosen in this experiment. Figure 6

Finally, we run our procedures on sets of biological sequences. The first set consisted of 106 sequences of intron 4 from the licorne gene of *Drosophila simulans* and *Drosophila melanogaster* strains taken from Haddrill et al (**?**). The *Anoplura* data set are sequences of cytochrome oxidase subunit 1, downloaded from GenBank[3]. The third set contained various *Drosophila* complete mitochondrial genomes, also downloaded from GenBank. And the last data set is 54 newly sequenced complete mtDNA genomes from teleost fish from Miya et al (Miya *et al.*, 2003). The results are shown in Table 1, where n is the common length of all the sequences in each set, and m is the number of sequences in the set. Table 1

[3] http://www.ncbi.nlm.nih.gov

6 Conclusions

In this paper we presented algorithms for finding all the segregating sites of a given set of strings $\mathcal{S}$. The first two algorithms are very fast and are useful when $\mathcal{S}$ is a new, not already processed, set of sequences. Both algorithms operate in $O(mn)$ time, where m is the number of input sequences and n is their length, however the second algorithm's expected running time will be on average less than $O(mn)$.

We also presented two more algorithms, which not only can compute the segregating sites of a new set of strings $\mathcal{S}$, but also can update the number and positions of the segregating sites of $\mathcal{S}$, whenever the latter is updated with an insertion of a new string or the removal of an existing one.

7 Acknowledgements

We thank J. Holub and G. Valiente for their useful comments and suggestions.

References

Fu, Y. X., 1996. Estimating the age of the common ancestor of a DNA sample using the number of segregating sites. *Genetics* 144, 829–838.

Hebert, P. D., Cywinska, A., Ball, S. L., and deWaard, J. R., 2003. Biological identifications through DNA barcodes. *Proc. Biol. Sci.* 270, 313–321.

Innan, H., Zhang, K., Marjoram, P., Tavare, S., and Rosenberg, N. A., 2005. Statistical tests of the coalescent model based on the haplotype frequency distribution and the number of segregating sites. *Genetics* 169, 1763–1777.

Karlin, S. and McGregor, J. L., 1967. The number of mutant forms maintained in a population. *Proc. Fifth Berkeley Symp. Math. Statist. Prob.* 4, 415–438.

Klein, E. K., Austerlitz, F., and Laredo, C., 1999. Some statistical improvements for estimating population size and mutation rate from segregating sites in DNA sequences. *Theor Popul Biol* 55, 235–247.

Margulies, M., Egholm, M., Altman, W. E., Attiya, S., Bader, J. S., Bemben, L. A., Berka, J., Braverman, M. S., Chen, Y. J., Chen, Z., Dewell, S. B., Du, L., Fierro, J. M., Gomes, X. V., Godwin, B. C., He, W., Helgesen, S., Ho, C. H., Irzyk, G. P., Jando, S. C., Alenquer, M. L., Jarvie, T. P., Jirage, K. B., Kim, J. B., Knight, J. R., Lanza, J. R., Leamon, J. H., Lefkowitz, S. M., Lei, M., Li, J., Lohman, K. L., Lu, H., Makhijani, V. B., McDade, K. E., McKenna, M. P., Myers, E. W., Nickerson, E., Nobile, J. R., Plant, R., Puc, B. P., Ronan, M. T., Roth, G. T., Sarkis, G. J., Simons, J. F., Simpson, J. W., Srinivasan, M., Tartaro, K. R., Tomasz, A., Vogt, K. A., Volkmer, G. A., Wang, S. H., Wang, Y., Weiner, M. P., Yu, P., Begley, R. F., and Rothberg, J. M., 2005. Genome sequencing in microfabricated high-density picolitre reactors. *Nature* 10.1038/nature03959 [doi].

Miya, M., Takeshima, H., Endo, H., Ishiguro, N. B., Inoue, J. G., Mukai, T., Satoh, T. P., Yamaguchi, M., Kawaguchi, A., Mabuchi, K., Shirai, S. M., and Nishida, M., 2003. Major patterns of higher teleostean phylogenies: a new perspective based on 100 complete mitochondrial DNA sequences. *Mol Phylogenet Evol.* 26, 121–138.

Perlitz, M. and Stephan, W., 1997. The mean and variance of the number of segregating sites since the last hitchhiking event. *J Math Biol* 36, 1–23.

Syvanen, A. C., 2005. Toward genome-wide SNP genotyping. *Nat Genet* 37s, S5–10.

List of Figures

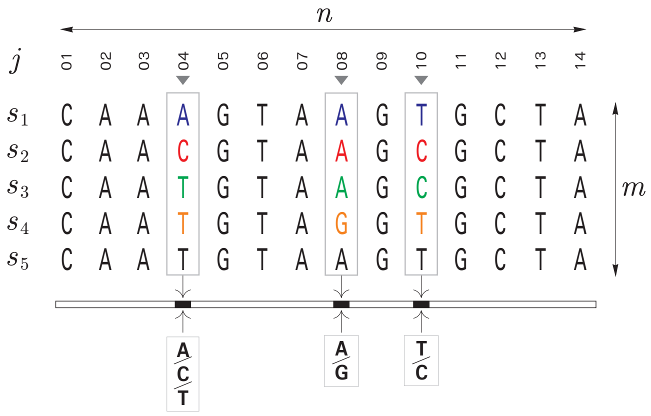

n
j 01 02 03 04 05 06 07 08 09 10 11 12 13 14
s_1 C A A A G T A A G T G C T A
s_2 C A A C G T A A G C G C T A
s_3 C A A T G T A A G C G C T A
s_4 C A A T G T A G G T G C T A
s_5 C A A T G T A A G T G C T A
m
A/C/T
A/G
T/C

Figure 1: A sample and the three segregating sites.

⇑

Manolis Christodoulakis

Figure 1 (of 6)

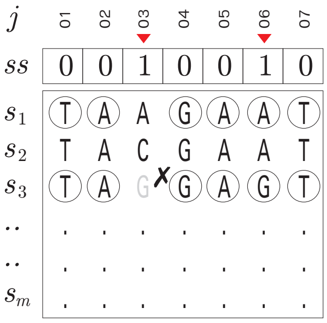
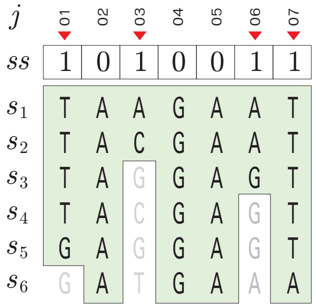


Figure 2: Illustration of how a new string (s_3) is processed by Algorithm 1. Note that only those nucleotides marked with circles were compared.

$\Uparrow$

Manolis Christodoulakis

Figure 2 (of 6)



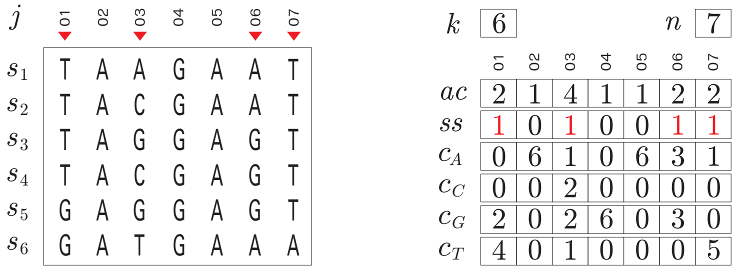
j	01	02	03	04	05	06	07
ss	1	0	1	0	0	1	1
s_1	T	A	A	G	A	A	T
s_2	T	A	C	G	A	A	T
s_3	T	A	G	G	A	G	T
s_4	T	A	C	G	A	G	T
s_5	G	A	G	G	A	G	T
s_6	G	A	T	G	A	A	A

Figure 3: Illustration of the fact that not all symbols in a sample need to be checked. Here 8/42 (19%) of the symbols are not checked — greyed symbols.

⇑

Manolis Christodoulakis

Figure 3 (of 6)



j	01	02	03	04	05	06	07
	▼		▼			▼	▼
s_1	T	A	A	G	A	A	T
s_2	T	A	C	G	A	A	T
s_3	T	A	G	G	A	G	T
s_4	T	A	C	G	A	G	T
s_5	G	A	G	G	A	G	T
s_6	G	A	T	G	A	A	A

(a) Sample $\mathcal{S}$

k 6 n 7

	01	02	03	04	05	06	07
ac	2	1	4	1	1	2	2
ss	1	0	1	0	0	1	1
c_A	0	6	1	0	6	3	1
c_C	0	0	2	0	0	0	0
c_G	2	0	2	6	0	3	0
c_T	4	0	1	0	0	0	5

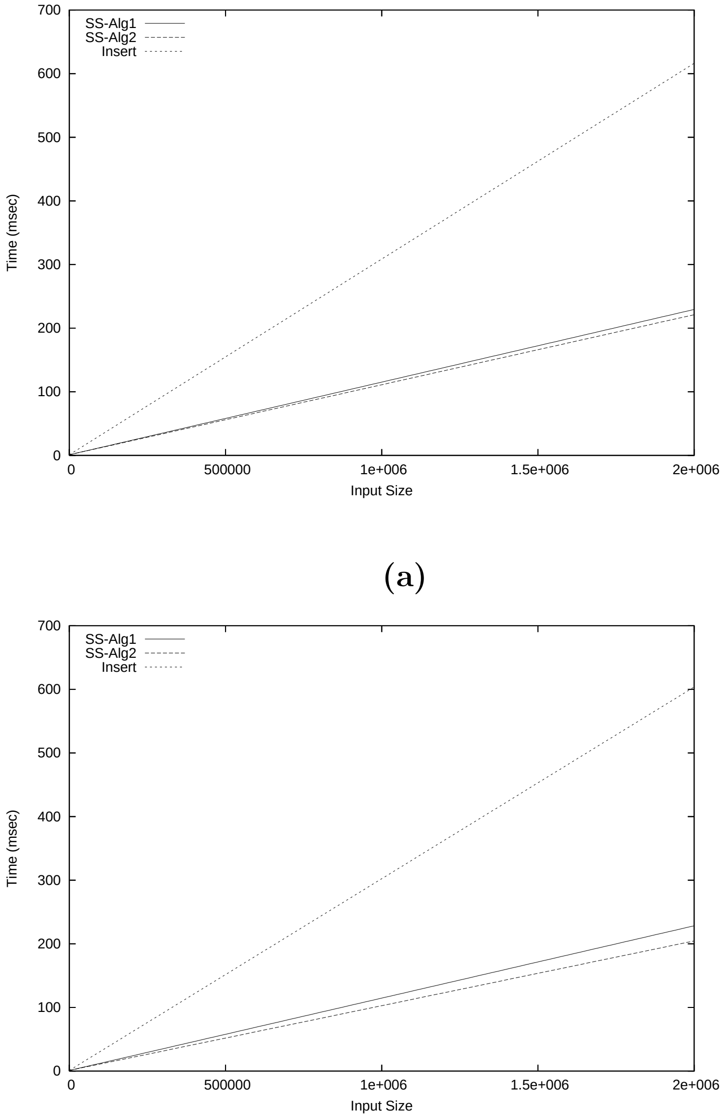
(b) `SSCounter` of $\mathcal{S}$

Figure 4: Example of an `SSCounter`.

⇑

Manolis Christodoulakis

Figure 4 (of 6)



(a)

(b)

Figure 5: Running time for **(a)** 1000, and **(b)** 9000 segregating sites.

⇑

Manolis Christodoulakis

Figure 5 (of 6)

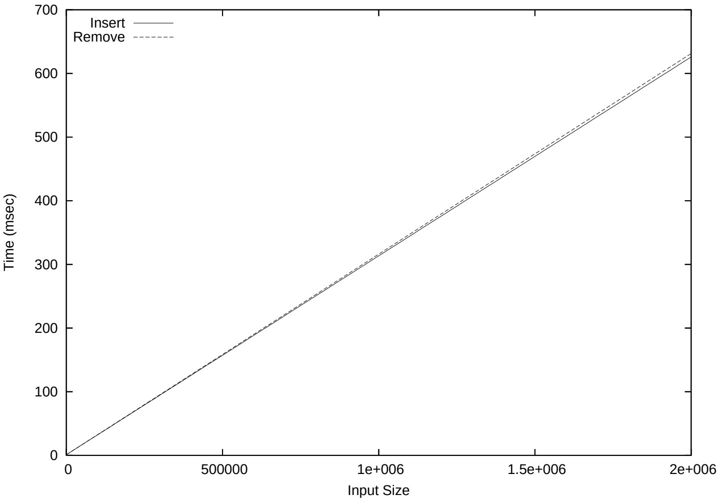


Figure 6: Running time for updating an `SSCounter`.

$\Uparrow$

Manolis Christodoulakis

Figure 6 (of 6)

List of Tables

				Procedures		
Set of Sequences	n	m	Seg. Sites	SS-ALG1	SS-ALG2	INSERT
Drosophila 1	546	106	225	17ms	16ms	22ms
Anoplura	882	184	822	38ms	29ms	61ms
Drosophila 2	16111	35	2632	66ms	88ms	181ms
Fish	25619	54	22492	207ms	224ms	370ms

Table 1: Performance of the algorithms for biological sequences.

⇑

Manolis Christodoulakis

Table 1 (of 1)