Viral Genome Compression

Lucian Ilie[1,*,**], Liviu Tinta[1],
Cristian Popescu[1], and Kathleen A. Hill[2]

[1] Department of Computer Science, University of Western Ontario
London, Ontario, N6A 5B7, Canada
ilie@csd.uwo.ca
[2] Department of Biology, University of Western Ontario
London, Ontario, N6A 5B7, Canada

Abstract. Viruses compress their genome to reduce space. One of the main techniques is overlapping genes. We model this process by the shortest common superstring problem, that is, we look for the shortest genome which still contains all genes. We give an algorithm for computing optimal solutions which is slow in the number of strings but fast (linear) in their total length. This algorithm is used for a number of viruses with relatively few genes. When the number of genes is larger, we compute approximate solutions using the greedy algorithm which gives an upper bound for the optimal solution. We give also a lower bound for the shortest common superstring problem. The results obtained are then compared with what happens in nature. Remarkably, the compression obtained by viruses is quite high and also very close to the one achieved by modern computers.

Keywords: viruses, viral genomes, genome compression, overlapping genes, shortest common superstring problem, exact algorithms, approximate solutions, lower bounds.

1 Introduction

According to [5], all virus genomes experience pressure to minimize their size. For example, those with prokaryotic hosts must be able to replicate quickly to keep up with their host cells. In the case of viruses with eukaryotic hosts, the pressure on the genome size comes from the small size of the virus, that is, from the amount of nucleic acid that can be incorporated.

One way to reduce the size of their genome is by overlapping genes. Some viruses show tremendous compression of genetic information when compared with the low density of information in the genomes of eukaryotic cells. As claimed in [5], overlapping genes are common and "the maximum genetic capacity is compressed into the minimum genome size." This property looks very interesting from mathematical point of view and we found it surprising that it was not much investigated. Daley and McQuillan [9] introduces and investigates a number

* Corresponding author.
** Research partially supported by NSERC.

C. Mao and T. Yokomori (Eds.): DNA12, LNCS 4287, pp. 111–126, 2006.
© Springer-Verlag Berlin Heidelberg 2006

of formal language theory operations motivated by the biological phenomenon. Krakauer [12] discusses genomic compression in general as achieved through reduced redundancy, overlapping genes, or translational coupling.

In this paper, we investigate this property by naturally modelling it as the shortest common superstring problem (SCS). The genes are seen as strings and we look for the shortest superstring that contains them all. A variation is also considered due to the retrograde overlaps which may be present in some viruses.

The SCS problem is known to be NP-hard. We give an algorithm to compute optimal solutions which works well when the number of strings is not too high. The algorithm is conceptually very simple and also very fast with respect to the total length of all strings. We used this algorithm for those viral genomes whose number of genes is not very high.

When the number of strings increases, we are no longer able to find optimal solutions and use a greedy algorithm for an approximation. This gives an upper bound for the length of a shortest superstring and, for a better estimate, we provide also a lower bound.

Finally, our results are compared with those obtained by viruses. The amount of compression using gene overlapping achieved by the viruses is remarkable; in all examples considered, it is the same or very close to the one obtained by modern computers. The biological significance of these results is to be investigated. Aside from the compression achieved in nature, any solution (or lower bound) for the corresponding SCS problem provides a limitation on the size of a viral genome which contains a given set of genes. Again, the biological relevance of such results remains to be clarified.

2 Basic Definitions

Let Σ be an alphabet, that is, a finite non-empty set. Such an alphabet can be the set of four nucleotides $\{\mathrm{A}, \mathrm{T}, \mathrm{C}, \mathrm{G}\}$. We denote by Σ^* the set of all finite strings over Σ. The empty word is denoted ε. Given a string $w \in \Sigma^*$, $w = a_1 a_2 \cdots a_n$, $a_i \in \Sigma$, the length of w is $|w| = n$; the length of ε is 0. We also denote $w[i] = a_i$ and $w[i..j] = a_i a_{i+1} \cdots a_j$, for all $1 \leq i \leq j \leq n$. The *reversal* of w is $a_n a_{n-1} \cdots a_1$.

If $w = xyz$, for some $w, x, y, z \in \Sigma^*$, then x, y, and z are a *prefix*, *factor* (or *substring*), and *suffix* of w, resp. The prefix (suffix) of length n of w is denoted $\mathsf{pref}_n(w)$ ($\mathsf{suff}_n(w)$).

For further notions and results on string combinatorics and algorithms we refer to [14] and [7].

3 The Shortest Common Superstring Problem

The formal definition of the shortest common superstring problem (SCS) is: given k strings $w_1, w_2, \ldots, w_k$, find a shortest string w which contains all w_is as factors; such a w is usually called a shortest common superstring. Any superstring will be called a solution, whereas a shortest one is an optimal solution.

Example 1. Consider the strings $w_1 = \mathsf{baac}$, $w_2 = \mathsf{aacc}$, and $w_3 = \mathsf{acaa}$. A shortest superstring has length 8; it is baacaacc.

The SCS problem has many applications. Data compression is one of the fields where the SCS problem is very useful because data may be stored very efficiently as a superstring; see [10], [15]. This superstring contains all the information in a compressed form. Computational biology is another field where SCS can be applied; see [13].

The SCS problem was proved to be NP-hard in [10] and then MAX SNP-hard in [3]. Therefore, it is unlikely to have polynomial time exact algorithms and research focussed mainly on approximation algorithms [17,8,11,1,2,4]. The best approximation algorithm to date is due to Sweedyk [16] and can reach an approximation ratio of $2\frac{1}{2}$.

Still, in practice the very simple greedy algorithm is used with very good results. Blum et al. [3] proved that greedy is a 4-approximation algorithm. The still open conjecture is that the approximation factor is 2, which would be optimal as there are examples for which greedy produces no better approximations.

4 Viral Genome Compression

As already mentioned in the introduction, viruses can overlap their genes. There are several types of overlaps. First we need to recall the DNA complementarity: the two strands of DNA are complementary and have opposite direction. The complementarity is such that whenever an A occurs on one strand, a T must appear on the other; we say that A and T are complementary. Similarly, C and G are complementary. We denote the complement of a nucleotide N by $\overline{N}$. That is, we have $\overline{\mathrm{A}} = \mathrm{T}$, $\overline{\mathrm{C}} = \mathrm{G}$, and vice versa. Also, $\overline{\overline{\mathrm{A}}} = \mathrm{A}$. Complementarity is needed to understand retrograde overlapping.

For a string $w = a_1 a_2 \cdots a_{|w|}$, we construct the *complemented reversal* of w, $\overline{w} = \overline{a}_{|w|}\overline{a}_{|w|-1} \cdots \overline{a}_1$. When w appears in one strand, $\overline{w}$ occurs opposite it in the other strand.

Example 2. Let x = ACCGTGTAC and y = GTGTACCGTAC be two hypothetical genes. The complemented reversal of x is $\overline{x}$ = GTACACGGT. The possible overlaps between x and y are shown in Figs. 1 and 2. In Fig. 1 we have overlaps on the same strand, that is, direct overlaps; one is called suffix overlap and the other prefix overlap but such a difference is irrelevant for us.

In Fig. 2 we have retrograde overlaps (as can appear, for instance, in double stranded DNA viral genomes). As seen in the figure, each x in the upper strand

Fig. 1. Direct overlaps (same strand)

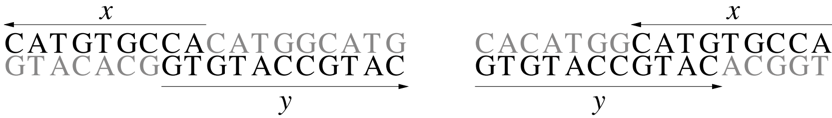

Fig. 2. Retrograde overlaps (opposite strands)

correspond to an $\overline{x}$ in the lower strand. Again, one is called head-on overlap, the other end-on overlap, without relevance for our purpose.

5 Computing Overlaps

In order to give some algorithms for optimal or approximate solutions for the SCS problem, we need to compute overlaps between strings. Also, we need to eliminate those strings which are factors of others.

An *overlap* between two given strings u and v is any suffix of u that is also a prefix of v. We shall need only the longest overlaps but our algorithm computes them all in the same optimal time. The set $\textsc{overlaps}(u, v)$ contains the lengths of all suffixes of u that are prefixes of v. We denote by $\mathsf{overlap}(u, v)$ the length of the longest overlap. Here is an example.

Example 3. For the strings $u = \mathsf{abaababa}$ and $v = \mathsf{abababb}$ we have

$$\begin{array}{ll} \textsc{overlaps}(u, v) = \{1, 3, 5\}, & \mathsf{overlap}(u, v) = 5, \\ \textsc{overlaps}(v, u) = \emptyset, & \mathsf{overlap}(v, u) = 0. \end{array}$$

To compute overlaps, we shall use a classical notion in pattern matching: a *border* of a string w is any string which is both a prefix and a suffix of w; *the border* of w, denoted $\mathsf{border}(w)$, is the longest non-trivial border of w, that is, different from w itself. Notice that all borders of w are: $\mathsf{border}(w), \mathsf{border}^2(w) = \mathsf{border}(\mathsf{border}(w)), \mathsf{border}^3(w), \ldots, \varepsilon$.

Denote $|w| = n$ and consider the array $\textsc{border}_w[0..n]$, where, for all $1 \leq i \leq n$, $\textsc{border}_w[i] = |\,\mathsf{border}(w[1..i])|$; $\textsc{border}_w[0] = -1$ for technical purposes.

Example 4. For the string $w = \mathsf{abaababaaba}$ we have

$$\textsc{border}_w = [-1, 0, 0, 1, 1, 2, 3, 2, 3, 4, 5, 6]$$

and all borders of w are

$$\begin{array}{ll} \textsc{border}_w(|w|) = 6, & \mathsf{border}(w) = \mathsf{abaaba}, \\ \textsc{border}_w^2(|w|) = 3, & \mathsf{border}^2(w) = \mathsf{aba}, \\ \textsc{border}_w^3(|w|) = 1, & \mathsf{border}^3(w) = \mathsf{a}. \end{array}$$

The array $\textsc{border}_w$ can be computed in time linear in $|w|$ by a classical algorithm. The idea is to compute the elements from first to last. Then, when computing $\textsc{border}_w[i]$, all previous elements are known. The border of $w[1..i]$ is either an extension of a border of $w[1..i-1]$ or empty if this is not possible.

We use borders to solve our problem. Assume we are given two strings u and v. Consider a new letters # (which does not appear in u or v) and construct the string $w = v\#u$. It is clear that any border of w gives an overlap of u and v and vice versa. Therefore, using borders, we obtain an algorithm for computing overlaps which is linear in terms of $|u| + |v|$. Notice, however, that if one of the strings is much longer than the other, then we do not need the whole long string but just a short piece of it. An algorithm which works in linear time in the size of the shorter string would simply consider the string $\mathsf{pref}_s(v)\#\,\mathsf{suff}_s(u)$, where $s = \min(|u| - 1, |v| - 1)$.

We can also do it all at once. For the SCS problem, we always exclude from calculations the strings which are included as factors in others. This is pattern searching and there are many linear time algorithms for it. We can also use the borders as above to give a simple algorithm to both identify factors and compute overlaps. We consider $w = v\#u$. Assuming $|v| \le |u|$, v is a factor of u if and only if there is i such that $\text{BORDER}_w(i) = |v|$.

```
OVERLAPS-AND-FACTORS(u, v)
 1.  w ← v#u
 2.  n ← |w|
 3.  BORDER_w[0] ← −1
 4.  b ← −1
 5.  for i from 1 to n do
 6.      while b ≥ 0 and w[b + 1] ≠ w[i] do
 7.          b ← BORDER_w[b]
 8.      b ← b + 1
 9.      BORDER_w[i] ← b
10.      if BORDER_w[i] = |v| and |v| ≤ |u| then
11.          return overlap(u, v) = −1          [ v is a factor of u ]
12.  return overlap(u, v) = BORDER_w[|w|]
```

This algorithm is linear in $|u| + |v|$; this is optimal since it is the minimum required for searching.

Lemma 1. *The algorithm* OVERLAPS-AND-FACTORS(u, v) *returns* -1 *iff* v *is a factor of* u *and otherwise computes the longest overlap of* u *and* v*. It runs in time* $\mathcal{O}(|u| + |v|)$*.*

6 Optimal Solutions of SCS

We may assume that none of the strings w_i appears as factor of another one. (We check this in the algorithm.) Therefore, for any solution w of SCS, there is a permutation σ on k elements such that w contains each w_i as a factor starting at position p_i and

$$p_{\sigma(1)} < p_{\sigma(2)} < \cdots < p_{\sigma(k)}.$$

Example 5. For the strings in Example 1, the optimal solution is given by the permutation $(1, 3, 2)$.

Therefore, our brute-force algorithm to compute an optimal solution of SCS will try all such permutations σ; the set of all permutations on k elements is the symmetric group $\mathcal{S}_k$. For each permutation, we need the maximum overlap between $w_{\sigma(i)}$ and $w_{\sigma(i+1)}$. No other overlaps are needed. Assuming that $w_{\sigma(i)}$ and $w_{\sigma(i+1)}$ overlap each other on a length less than their maximal overlap. Then we can simply overlap them more to obtain a shorter superstring.

We shall need one more definition. For two strings u and v which are not factors of each other, we denote by $\mathsf{merge}(u, v)$ the string obtained by overlapping them as much as possible, that is, $\mathsf{merge}(u, v) = u\,\mathsf{suff}_{|v|-\mathsf{overlap}(u,v)}(v) = \mathsf{pref}_{|u|-\mathsf{overlap}(u,v)}(u)v$.

Example 6. For the strings $u = \mathsf{abaababa}$, $v = \mathsf{abababb}$ we have $\mathsf{merge}(u, v) = \mathsf{abaabababb}$.

Here is the algorithm.

```
SCS-OPTIMAL(w_1, w_2, ..., w_k)
 1. for i from 1 to k do
 2.       for j from 1 to k do
 3.             if i ≠ j then
 4.                   overlap(w_i, w_j) ← OVERLAPS-AND-FACTORS(w_i, w_j)
 5.                   if overlap(w_i, w_j) = −1 then eliminate w_i
 6. scs ← Σ_{i=0}^{k} |w_i|            [ we use the same k but it may be smaller ]
 7. for all σ ∈ S_k do
 8.       w ← w_σ(1)
 9.       for i from 2 to k do
10.             w ← merge(w, w_σ(i))
11.       if scs > |w| then
12.             scs ← |w|
13. return scs
```

Proposition 1. *The algorithm* SCS-OPTIMAL$(w_1, w_2, \ldots, w_k)$ *computes an optimal solution for SCS and runs in time* $\mathcal{O}(k!\ell)$, *where* $\ell = \sum_{i=1}^{k} |w_i|$.

Proof. The correctness follows from the fact that we try all permutations. As explained above, after eliminating strings which appear as factors of others, it it enough to consider only longest overlaps.

The time complexity for the preprocessing steps 1-5 is $\mathcal{O}(k^2\ell)$, because of Lemma 1. In the main processing part, steps 7-12, we repeat $k!$ times something linear in ℓ. This is the dominant order. □

7 Approximate Solutions of SCS

As the SCS problem is NP-hard, in practice approximation algorithms are often used to find a superstring which may not be shortest but hopefully close to optimal. The most common such algorithm for SCS is the greedy algorithm, which we describe below. It uses the natural idea of considering the longer overlaps

first. It may not produce an optimal solution but it cannot be too far away. Here is an example when the greedy algorithm does not give an optimal solution.

Example 7. Consider again the strings in Example 1, $w_1 = \mathsf{baac}$, $w_2 = \mathsf{aacc}$, and $w_3 = \mathsf{acaa}$. The overlaps are shown below:

$\mathsf{overlap}(w_i, w_j)$	w_1	w_2	w_3
w_1		3	2
w_2	0		0
w_3	0	2	

The greedy algorithm chooses first the longest overlap, that is, $\mathsf{overlap}(w_1, w_2)$, and obtains the string baaccacaa of length 9, since $\mathsf{merge}(w_1, w_2)$ and w_3 have no overlap. But there is a shorter one, given by the permutation $(1, 3, 2)$, of length 8, that is baacaacc.

It is conjectured that the greedy solution is always at most twice longer than optimal; see [16] and the references therein for approximation algorithms for the SCS problem. In practice, the greedy algorithm works pretty well, as we shall see also in our experiments.

```
SCS-GREEDY(w_1, w_2, ..., w_k)
1.  compute overlaps and eliminate factors as before
2.  greedy_scs ← Σ_{i=0}^{k} |w_i|
3.  for all (i, j) with overlap(w_i, w_j) = max_{(s,t)} overlap(w_s, w_t) do
4.        eliminate w_i and w_j from the list
5.        add w = merge(w_i, w_j) to the list
6.             denote the new list w'_1, ..., w'_{k-1}
7.             the overlaps of w are given by w_i for prefix and by w_j for suffix
8.        ℓ ← SCS-GREEDY(w'_1, w'_2, ..., w'_{k-1})
9.        if greedy_scs > ℓ then
10.            greedy_scs ← ℓ
11. return greedy_scs
```

The greedy algorithm gives an upper bound for the shortest length of a common superstring.

8 Lower Bounds

We give in this section a lower bound for the length of the shortest superstring. It is computed using also a greedy approach but without checking if it is possible to actually find a superstring which uses the considered overlaps. (When this is possible, we have an optimal solution of SCS.)

Any superstring w is defined by a permutation σ on k elements which gives $k-1$ overlaps. Also, the length of the superstring is the total length of all strings minus the total length of overlaps, that is,

$$|w| = \sum_{i=1}^{k} |w_i| - \sum_{i=1}^{k-1} \mathsf{overlap}(w_{\sigma(i)}, w_{\sigma(i+1)}).$$

For our estimate, we consider the matrix of overlaps, $(\mathsf{overlap}(w_i, w_j))_{1\le i\ne j\le k}$. A permutation σ as above gives $k-1$ overlaps such that no two are in the same row or column. We relax this condition by considering only rows or only columns. Choosing $k-1$ longest overlaps such that no two are on the same row gives a lower bound. Similarly for columns.

The algorithm below computes the first one. The second is computed analogously. We assume the matrix of overlaps has already been computed.

LOWER-BOUND-ROW$(w_1, w_2, \ldots, w_k)$

1. sort all elements of the matrix $(\mathsf{overlap}(w_i, w_j))_{1\le i\ne j\le k}$ decreasingly
2.   to obtain $\mathsf{overlap}(w_{i_1}, w_{j_1}), \ldots, \mathsf{overlap}(w_{i_{n^2-n}}, w_{j_{n^2-n}})$
3. $lower_bound_row \leftarrow 0$
4. $rows_used \leftarrow 0$
5. $t \leftarrow 1$
6. **while** $rows_used < k-1$ **do**
7.   **if** row i_t not used **then**
8. $lower_bound_row \leftarrow lower_bound_row + |w_{i_t}| - \mathsf{overlap}(w_{i_t}, w_{j_t})$
9.   mark row i_t as used
10. $rows_used \leftarrow rows_used + 1$
11. $t \leftarrow t + 1$
12. $lower_bound_row \leftarrow lower_bound_row + |w_{j_{t-1}}|$
13. **return** $lower_bound_row$

Proposition 2. *The above algorithm computes a lower bound for the length of the shortest superstring in time $\mathcal{O}(k^2 \log k)$.*

Proof. The time required by the algorithm is $\mathcal{O}(k^2 \log k)$ because of sorting. The **while** cycle takes only $\mathcal{O}(k^2)$ time as it traverses the list of $k^2 - k$ elements at most once and spends constant time for each element.

For correctness, it is enough to prove that the sum of the overlaps chosen by the algorithm is larger than the sum of overlaps corresponding to an optimal solution. In both cases, we have $k-1$ overlaps involved, no two in the same row. Assume that an optimal solution chooses all rows except for the ith whereas our algorithm for the lower bound misses only the jth row. In all rows chosen by both, the overlap included for the lower bound is at least as large. If $i = j$, this proves that we obtain indeed a lower bound. If $i \ne j$, then the overlap chosen for the lower bound from row i is larger than the one for the optimal solution in row j as the former appear first in the sorted list from step 2. □

As already mentioned, another lower bound is obtained similarly, by choosing $k-1$ elements from different columns in the overlap matrix; denote this lower bound by $lower_bound_col$. We have then the following lower bound:

$$lower_bound_scs = \max(lower_bound_row, lower_bound_col).$$

The next result, which summarizes the above discussed bounds, is clear.

Proposition 3. *We always have*

$$lower_bound_scs \le scs \le greedy_scs.$$

Example 8. For the strings in Example 1, we have:

$lower_bound_row = 7$, because of $\mathsf{overlap}(w_1, w_2)$ and $\mathsf{overlap}(w_3, w_2)$,
$lower_bound_col = 7$, because of $\mathsf{overlap}(w_1, w_2)$ and $\mathsf{overlap}(w_1, w_3)$,
$lower_bound_scs = 7$,
$scs = 8$,
$greedy_scs = 9$.

The lower bound cannot be achieved however, as it involves the beginning of w_2 (or the end of w_1) twice. Also, it happened that the lower bounds corresponding to rows and columns are the same; this is not true in general.

9 Retrograde Overlaps

The possibility of retrograde overlaps (see Fig. 2) further complicates the search for solutions, optimal or approximate. Each string may appear in a superstring as it is or as its complemented reversal.

Therefore, we need first to compute more overlaps. The following equalities help computing only half of all possible ones:

(i) $\mathsf{merge}(x, y) = \overline{\mathsf{merge}(\overline{y}, \overline{x})}$,
(ii) $\mathsf{merge}(x, \overline{y}) = \overline{\mathsf{merge}(y, \overline{x})}$.

For the exact algorithm, we need to consider, for each string w_i, whether w_i or $\overline{w}_i$ appears at position $p_{\sigma(i)}$, which makes the algorithm even slower in the number of strings.

The greedy algorithm works rather similarly. Only the overlaps for the merged strings need to be set a bit differently. For instance, if the overlap between w_i and $\overline{w}_j$ is chosen, then the string $\mathsf{merge}(w_i, \overline{w}_j)$ is added and its overlaps are taken from those given by prefixes of w_i and w_j.

The lower bound is computed similarly. When choosing a certain overlap, the proper rows or columns need to be discarded for further consideration. For instance, in case of *lower_bound_row*, if the overlap between w_i and $\overline{w}_j$ is chosen, then all overlaps involving the suffix of of w_i must be discarded, that is, all pairs (w_i, w_s), $(w_i, \overline{w}_s)$, $(w_s, \overline{w}_i)$ and $(\overline{w}_s, \overline{w}_i)$.

10 Viral Compression Versus Computer Compression

We show in this section our computations for a number of viral genomes which were obtained from “The National Center for Biotechnology Information,” (web site www.ncbi.nlm.nih.gov). We start with a set of strings which are the genes and try to find a short superstring. Then we compare our result with the one

Table 1. Viral genome compression - optimal solutions

Family	Name	Total length	Viral	SCS
Paramyxoviridae	Human respiratory syncytial virus	13641	13609	13602
Rhabdoviridae	Bovine ephemeral fever virus	15029	14662	14650
Rhabdoviridae	Northern cereal mosaic virus	11922	11922	11917
Togaviridae	Sleeping disease virus	11745	11745	11738
Coronaviridae	SARS coronavirus	29974	29046	29040
Retroviridae	HIV-1 isolate 01IN565.11 from India	14125	8647	8646
Retroviridae	HIV-2 isolate ALI from Guinea-Bissau	14466	8809	8809

Table 2. Viral genome compression - approximate solutions

Family	Name	Total length	Viral	Greedy	Lower bound
Baculoviridae	Choristoneura fumiferana MNPV	119168	118319	117414	117228
Poxviridae	Vaccinia Virus strain Ankara	152029	150885	150588	150329
Herpesviridae	Bovine Herpesvirus 1	124819	119378	119276	119137
Adenoviridae	Human adenovirus type 5	36576	34342	34328	34322
Adenoviridae	Hemorrhagic enteritis virus	25158	23433	23414	23402
Iridoviridae	Frog virus 3	85593	84443	84248	84174

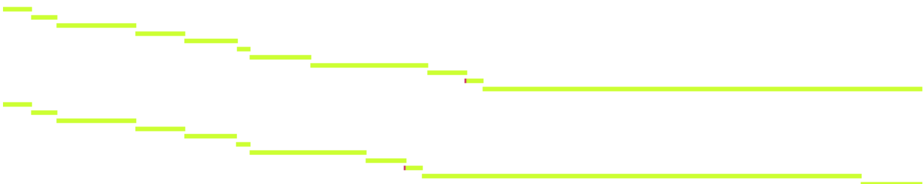

Fig. 3. Human respiratory syncytial virus

Fig. 4. Bovine ephemeral fever virus

Fig. 5. Northern cereal mosaic virus

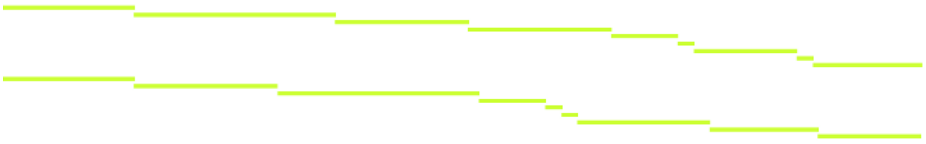

Fig. 6. Sleeping disease virus

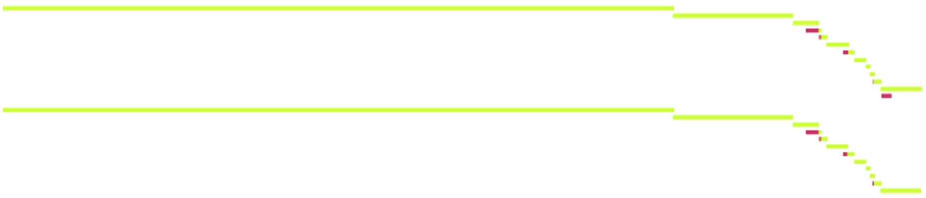

Fig. 7. SARS coronavirus

Fig. 8. HIV-1 isolate 01IN565.11 from India

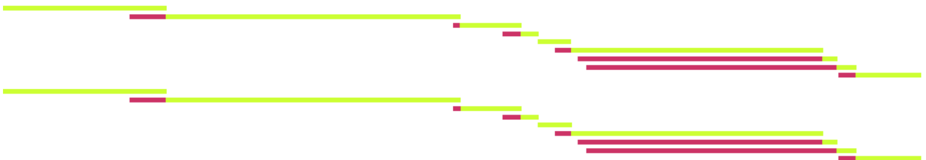

Fig. 9. HIV-2 isolate ALI from Guinea-Bissau

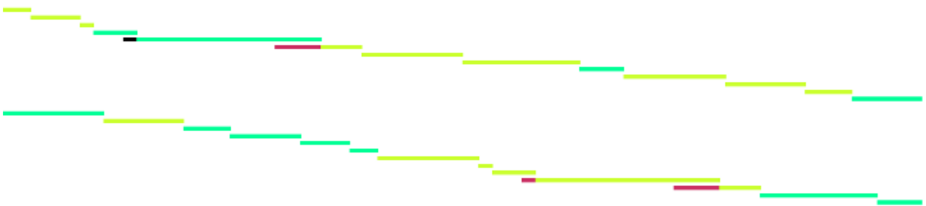

Fig. 10. Human adenovirus type 5

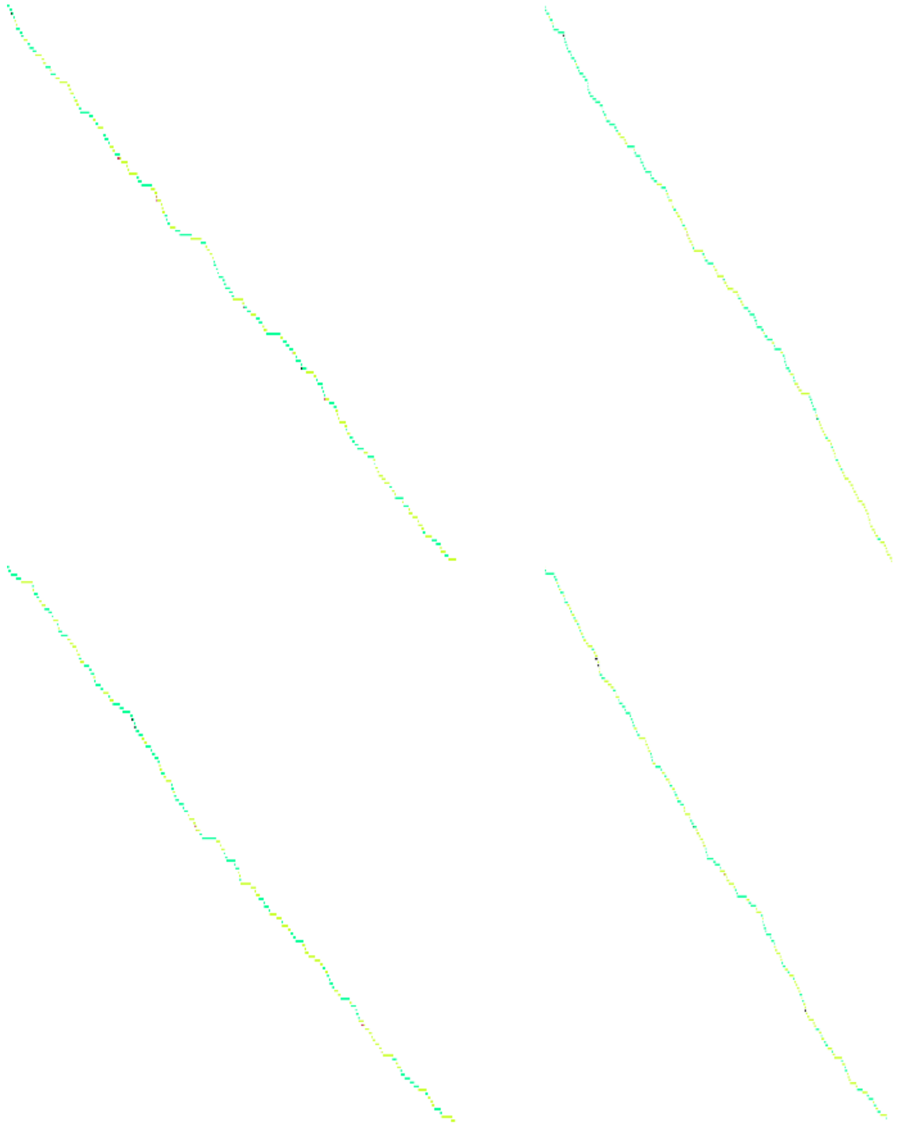

Fig. 11. Choristoneura fumiferana MNPV (left) and Vaccinia Virus strain Ankara (right)

achieved by the viruses. Notice that the time complexity of our exact algorithm grows very fast with the number of genes, but is linear in the total length.

We managed to obtain exact solutions in Table 1 for a number of single stranded RNA viral genomes with relatively few genes. The columns give, in order, the family, the name of the virus, the total length of all genes, the compression achieved by the virus (total length of coding regions), and the shortest common superstring. All lengths are given in number of nucleotides.

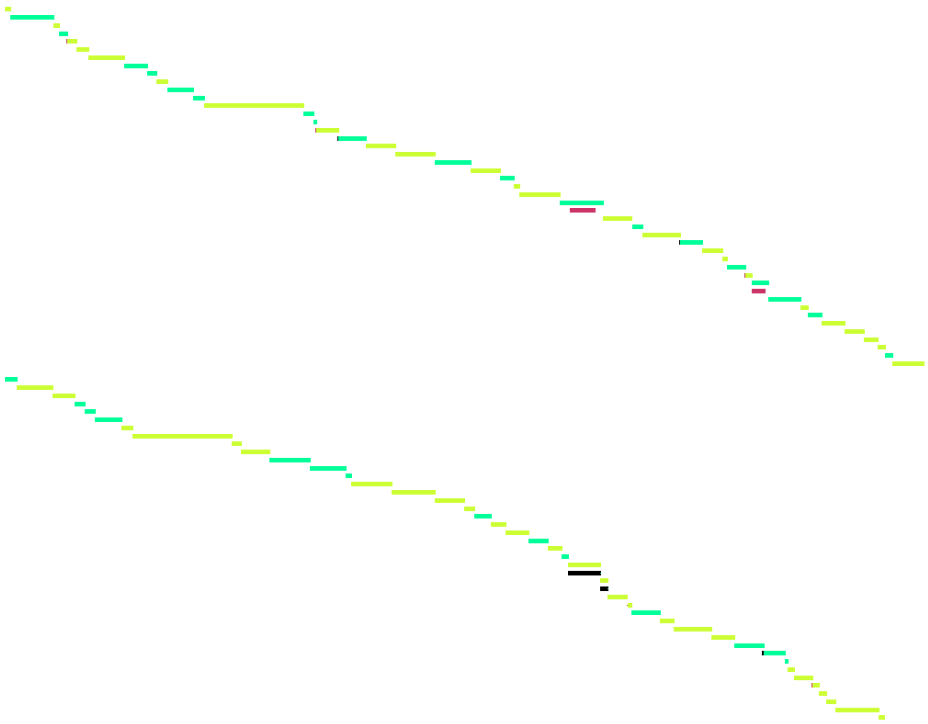

Fig. 12. Bovine Herpesvirus 1

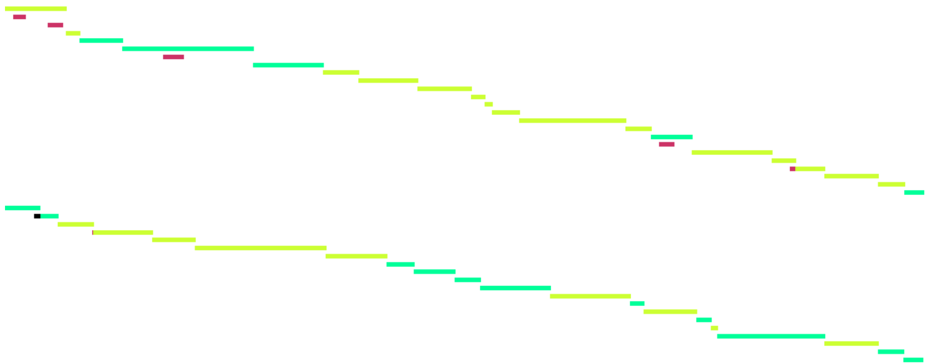

Fig. 13. Hemorrhagic enteritis virus

For genomes with more genes, we had to use the approximation algorithms. The results for a number of double stranded DNA viral genomes are shown in Table 2. The columns have similar meaning, except that the one for the shortest common superstring is replaced by two: greedy and lower bound. All lengths are given in number of base pairs.

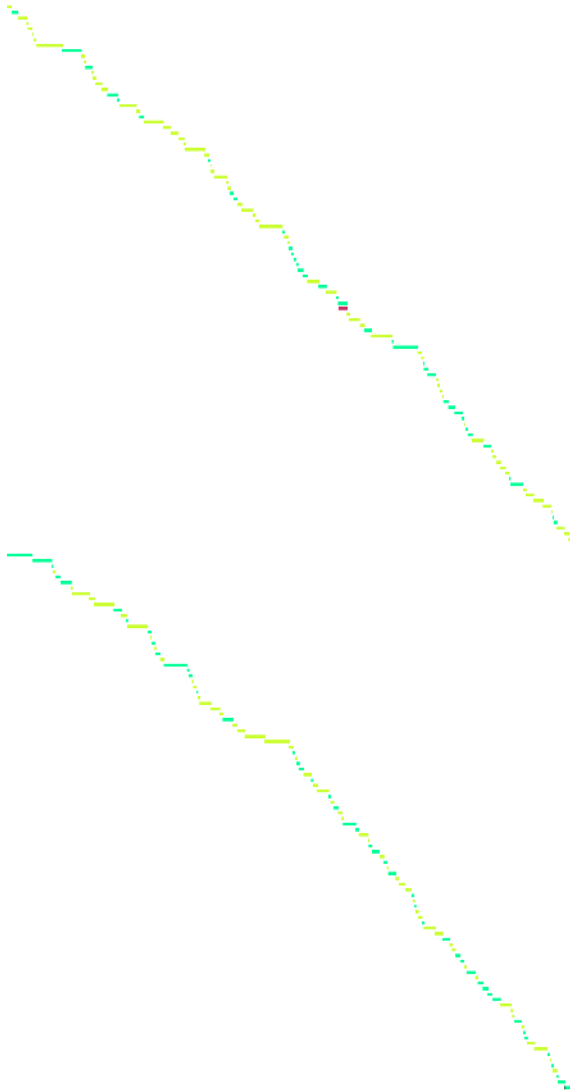

Fig. 14. Frog virus 3

The compression achieved by the viruses is, on average, 7.98%, that is, the (average) ratio between the reduction in size (total length of all genes minus viral coding) and the initial size (total length of genes). For the viruses in the first table, the ratio is higher, 11.95%, whereas for the second table it is 3.36%. The average compression ratio is remarkably high if we keep in mind that DNA molecules (seen as strings) are very difficult to compress in general. Commercial file-compression programs achieve usually no compression at all and the best especially designed algorithms, see [6], can achieve something like 13.73%

(that is the average for DNACompress from [6], the best such algorithm to date).

Also, the compression achieved by viruses is very close to what we can do (using overlapping only) by computers. The above averages, for all viruses considered, single stranded RNA, and double stranded DNA viruses are 8.11% (only 0.13% better than viruses), 11.99%, and 3.59%, resp. For the second table we used the greedy compression; it should also be noticed that our lower bound behaves pretty well.

To give a better idea of the overlaps, Figs. 3–14 at the end show all genomes considered above as they appear in nature with the non-coding regions removed (top) and then as computed by our programs (bottom). The overlaps and different strands are shown in different color. (The figures are most useful in the electronic version of the paper.)

References

1. C. Armen and C. Stein, Improved length bounds for the shortest superstring problem, *Proc. 5th Internat. Workshop on Algorithms and Data Structures*, Lecture Notes in Comput. Sci. **955**, Springer-Verlag, Berlin, 1995, 494 – 505.
2. C. Armen and C. Stein, A $2\frac{2}{3}$ approximation algorithm for the shortest superstring problem, *Proc. Combinatorial Pattern Matching*, Lecture Notes in Comput. Sci. **1075**, Springer-Verlag, Berlin, 1996, 87 – 101.
3. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, Linear approximation of shortest superstrings, *J. Assoc. Comput. Mach.* **41**(4) (1994) 630 – 647.
4. D. Breslauer, T. Jiang, and Z. Jiang, Rotations of periodic strings and short superstrings, *J. Algorithms* **24** (1997) 340 – 353.
5. A.J. Cann, *Principles of Molecular Virology*, 3rd ed. Elsevier Academic Press, London, San Diego, 2001.
6. X. Chen, M. Li, B. Ma, and J. Tromp, DNACompress: fast and effective DNA sequence compression, *Bioinformatics* **18** 2002 1696 – 1698.
7. M. Crochemore and W. Rytter, *Jewels of Stringology*, World Sci. Pub., 2003.
8. A. Czumaj, L. Gasieniec, M. Piotrow, and W. Rytter, Parallel and sequential approximations of shortest superstrings, *Proc. First Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Comput. Sci. **824**, Springer-Verlag, Berlin, 1994, 95 – 106.
9. M. Daley and I. McQuillan, Viral gene compression: complexity and verification, *Proc. of CIAA'04*, Lecture Notes in Comput. Sci. **3317**, Springer, Berlin, 2005, 102–112.
10. J. Gallant, D. Maier, and J. Storer, On finding minimal length superstrings, *Journal of Comput. and Syst. Sci.* **20**(1) (1980) 50 – 58.
11. R. Kosaraju, J. Park, and C. Stein, Long tours and short superstrings, *Proc. 35th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Piscataway, NJ, 1994, 166 – 177.
12. D.C. Krakauer, Evolutionary principles of genomic compression, *Comments on Theor. Biol.* **7** (2002) 215 – 236.
13. A. Lesk, *Introduction to Bioinformatics*, Oxford University Press, Oxford, 2002.

14. M. Lothaire, *Algebraic Combinatorics on Words*, Cambridge Univ. Press, 2002.
15. J. Storer, *Data Compression: Methods and Theory*, Computer Science Press, 1988.
16. Z. Sweedyk, A $2\frac{1}{2}$-approximation algorithms for shortest superstring, *SIAM J. Comput.* **29**(3) (1999) 954 – 986.
17. S. Teng and F. Yao, Approximating shortest superstrings, *Proc. 34th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Piscataway, NJ, 1993, 158 – 165.