Technical Report TR-INF-2003-04-03-UNIPMN

A Simple and Fast DNA Compressor

Giovanni Manzini*   Marcella Rastero†

April 3, 2003

Abstract

In this paper we describe a new DNA compression algorithm. It is well known that one of the main features of DNA sequences is that they contain substrings which are duplicated except for a few random mutations. For this reason most DNA compressors work by searching and encoding approximate repeats. We depart from this strategy by searching and encoding only exact repeats. However, we use an encoding designed to take advantage of the possible presence of approximate repeats. Our approach leads to an algorithm which is an order of magnitude faster than any other algorithm and achieves a compression ratio very close to the best DNA compressors. Another important feature of our algorithm is its small space occupancy which makes it possible to compress sequences hundreds of megabytes long, well beyond the range of any previous DNA compressor.

1 Introduction

The compression of DNA sequences is one of the most challenging task in the field of data compression. Since DNA sequences are "the code of life" we expect them to be non-random and to present some regularities. It is natural to try to take advantage of such regularities in order to compactly store the huge DNA databases which are routinely handled by molecular biologists.

Although the saving in space is important, it is not the only application of DNA compression. DNA compression has been used to distinguish between coding and non-coding regions of a DNA sequence [11], to evaluate the "distance" between DNA sequences and to guess whether two organisms are "close" in the evolutionary tree [5, 12], and in other biological applications [1, 15].

It is well known that the compression of DNA sequences is a very difficult problem. Since DNA sequences only contain the four bases $\{a, c, g, t\}$ they can be stored using two bits per input symbol. The standard compression tools, such as gzip and bzip2, usually fail to achieve any compression since they use more than two bits per symbol. Among the "general purpose" compression algorithms only arithmetic coding [16, 17] is able to consistently use less than two bits per symbol. In [9] was introduced the first DNA-specific compression algorithms and others

---

followed afterwards [2, 5, 14]. These earlier DNA compressors achieve better compression than arithmetic coding, but they are extremely slow with respect to data compression standards. In [2] the authors report a running time of 2–3 minutes for a 80KB sequence. In [14] the authors report a running time of 8 minutes for a sequence of length 38KB, and a running time of some hours for a 224KB sequence.

Recently, a new algorithm called DNACompress [6] has made significant improvements over its predecessors. Thanks to an extremely fast search engine [7], DNACompress compresses better and is much faster than the previous algorithms. These features make DNACompress the current leader in this field. However, since DNA compression is still in its infancy, we can expect further improvements both in running time and compression ratio.

In this paper we describe a new compression algorithm which aims at achieving the same speed of the tools—such as gzip and bzip2—that we use everyday for general purpose compression. For doing so we depart from a strategy that has been used in every other DNA compression algorithm (with the exception of [2]): the encoding of approximate repeats. Searching and encoding approximate repeats is a natural approach in DNA compression since it is well known that DNA sequences contain long strings which are duplicated except for the insertion or deletions of a few bases. Unfortunately, searching for approximate repeats takes a long time and requires a large amount of memory. For these reasons we find it natural to investigate how much compression we can achieve working only with exact matches. Our idea is to search and encode only exact repeats, and to use the knowledge that DNA contains approximate repeats in the design of the coding algorithms. In other words, we only encode exact repeats but in designing the codewords that represent exact repeats we use the knowledge that some of these exact repeats are "fragments" of larger approximate repeats.

This strategy leads to an algorithm which is extremely fast. Our compression speed is roughly 2 microseconds per input byte which is only 2.25 times slower than bzip2. Among the previous DNA compressors the fastest is DNACompress which, for large files, takes roughly 70 microseconds per input byte. In terms of compression ratio our algorithm uses on average 0.066 more bits per symbol than DNACompress: this means an increase in space occupancy of 8.25 bytes every 1000 input bases.

Another important feature of our algorithm is its relatively small space occupancy. In order to detect repetitions which occur far apart in a sequence, DNA compressors usually maintain large data structures. The larger the size of these data structures the smaller is the length of the sequences that can be processed in main memory. Since DNA sequences of hundreds of megabytes are nowadays common, a small working space is definitely an important asset. For a sequence of length N we use $\approx (7N)/5$ bytes of working space which is roughly one fourth of the space used by DNACompress. Thanks to the small working space and to the speed of our algorithm we have been able to compress sequences of length up to 220MB, which is well beyond the range of any other DNA compressor.

2   A framework for DNA compression

In the following we use the term DNA sequence to denote a string over the alphabet $\{a, c, g, t\}$. We say that two sequences α, β are *complementary* if one can be obtained by the other replacing the symbol a with t (and vice-versa) and the symbol c with g (and vice-versa). We say that a sequence α is a *reverse complement* of β, and we write $\alpha = \bar{\beta}$ if α and the reverse of β are complementary.

For example, the reverse complement of *aactgt* is *acagtt* (and vice-versa). Reverse complements are also called *palindromes*.

It is well known that DNA sequences do not present the same regularities which are usually found in linguistic texts. This explains why the standard tools such as `gzip` and `bzip2` fail to compress them. To design a DNA compressor we must take advantage of the regularities which are usually found in this kind of data. The following three properties have been observed in many sequences and have been the basis (so far) of every DNA compressor.

dna-1 DNA sequences contain repeated substrings. Repeated substrings in DNA are often longer than in linguistic texts, but they are much less frequent. Another important difference with linguistic texts is that in DNA sequences repeated substrings may appear far apart in the input sequence, whereas in linguistic texts the occurrences of a substring are often clustered in a small region of the input.

dna-2 DNA sequences contain repeated palindromes, that is, pair of substrings which are the reverse complement of each other. The characteristics of the repeated palindromes are the same of repeated substrings: they are not very frequent, they can be long, and they may appear far apart in the input.

dna-3 DNA sequences contain repeated strings/palindromes with errors. With this term we denote a repetition in which the second occurrence is not identical to the first one (or to the reverse complement of the first one) because a few symbols have been deleted, inserted, or replaced by other symbols.

We now show how to design a fast DNA compressor which takes advantage of the above regularities of DNA sequences. Our algorithm only searches and encodes exact repeat/palindromes; it makes use of property dna-3 implicitly in the design of the codewords which represent the exact matches.

2.1   Finding exact matches

Let $T[1, N]$ denote a string over the alphabet $\{a, c, g, t\}$. Our first building block for the design of a fast DNA compressor is an efficient procedure for finding the occurrences of repeated substrings. We know (property dna-1) that repetitions in DNA sequences are less frequent but usually longer than in linguistic texts. Moreover, repeated substrings may appear far apart in the input sequence.

A technique which naturally matches these features is based on the use of fingerprints. Fingerprints have been introduced in [10] for the problem of finding the occurrences of a pattern P in a text T. The basic idea is to compute a fingerprint of the pattern by interpreting its symbols as the coefficient of a polynomial modulo a large prime. The pattern fingerprint is then compared to the fingerprints of all substrings of T of length $|P|$: this procedure quickly determines a set of positions where the pattern *does not* occur. This is a very efficient technique and fingerprints have become a common tool in the design of text algorithms. Recently, fingerprints have been used also for data compression: Bentley and McIlroy [3] have shown how to use them in order to find long common strings which appear far apart in the input text. Our use of fingerprints will follow very closely the ideas in [3].

We choose a parameter B (in the following we use $B = 20$) and we scan the input sequence T computing the fingerprint of each block of B consecutive symbols, that is, $T[1, B]$, $T[2, B + 1]$, $T[3, B + 2]$ and so on. During this process, the fingerprints of the blocks starting at positions

1, $B + 1$, $2B + 1$ and so on, are stored in a hash table together with their respective starting position; we call these blocks *stored blocks*. When we compute a new fingerprint, say of the block $T[k+1, k+B]$, we check in the hash table whether the same fingerprint has been previously stored. If $T[k + 1, k + B]$ has the same fingerprint of a stored block, say $T[hB + 1, hB + B]$, we check if the two blocks are indeed equal. If there is a match we extend it as far as possible to the left and to the right. As pointed out in [3] this procedure is not able to detect repeated substrings of length smaller than B, it will detect some of the repetitions of length between B and $2B - 2$, and it will detect all repeated substrings of length at least $2B - 1$ no matter how far apart they are in the input. This behavior matches our requirements: for DNA compression we are interested in long repeated substrings which can be far apart in the input sequence.

The above procedure can be easily modified to find also the occurrences of inverted repeats. When we scan the input sequence we compute the fingerprints of the string $T[k + 1, k + B]$ and of its reverse complement $\overline{T[k + 1, k + B]}$. Using the latter fingerprint we can quickly discover the presence of an inverted repeat of length B which can be later extended as far as possible. Note that we do not need to store in the hash table the fingerprints of inverted repeats. Hence, for an input sequence of length N we store N/B items overall.

We implemented the hash table as an integer array of size equal to the smallest prime largest than $2N/B$, and we used double hashing to handle collisions. The hash table and the input sequence T are the only data structures in our algorithm which depend on the size of the input N. Hence, the space occupancy of our algorithm (including the parts discussed in the next sections) is $N + 8N/B$ bytes plus lower order terms. In our tests we set $B = 20$, so the working space of our algorithm is $7N/5$ bytes plus lower order terms.

2.2 Encoding the matches

Having established a procedure for finding repeated strings and palindromes, we now show how to use it for the actual compression of DNA sequences. Suppose we have already compressed an initial portion $T[1, z]$ of the input sequence. Using the procedure outlined in the previous section we check whether the block $T[z + 1, z + B]$ (or its reverse complement) matches a stored block. If not, we try with $T[z + 2, z + B + 1]$, $T[z + 3, z + B + 2]$ and so on, until we find a block $T[z + b, z + b + B - 1]$ which matches. Then we extend this match as far as possible and we get a sequence $\beta = T[z + a + 1, z + a + \ell]$ which is a copy—or the reverse complement—of a substring starting in $T[1, z]$. Our algorithm next encodes the string $T[z + 1, z + a + \ell]$. Notice that this string consists of an *unmatched prefix* $\alpha = T[z + 1, z + a]$, and of a *matched suffix* $\beta = T[z + a + 1, z + a + \ell]$. The string $\alpha\beta = T[z + 1, z + a + \ell]$ is represented by the quintuplet $\langle U, r, P, M, \alpha \rangle$ where:

1. $U = |\alpha|$ is the length of the unmatched prefix;

2. r is a boolean value determining whether β is a plain copy ($r = 0$) or a reverse complement ($r = 1$) of a string in $T[1, z + a]$;

3. P is the starting position in $T[1, z]$ of the copy of β (or of $\bar{\beta}$ if $r = 1$);

4. $M = |\beta|$ is the length of the matched suffix;

5. α is the unmatched prefix.

Our algorithm computes a proper encoding of the quintuplet $\langle U, r, P, M, \alpha \rangle$, then sets $z \leftarrow z + |\alpha| + |\beta|$ and starts over the search for another repeated string or palindrome.

It should be clear that the compression of the input $T[1, N]$ amounts to finding a compact encoding of the sequence of quintuplets generated by the above procedure. In this section we concentrate on the encoding of the values U, P and M; we assume that the values r and α are stored without any compression using 1 bit for r and $2U$ bits for α.

The design of a compact encoding for the values U, P and M is a rather difficult task. The distribution of these parameters for a typical DNA file apparently shows very little regularity. For example, the parameter P appears to be uniformly distributed within its allowable range (recall that P denotes a position within the already compressed portion of the input). However, a closer look at the data shows that some regularity is induced by the presence of *approximate matches* in the input DNA sequence. In other words, although our search procedure only discovers *exact* matches the presence of approximate matches in the input sequence induces some regularity in values of the parameters U, P and M.

To show how this is possible, assume that a string σ of length 1000 appears in $T[2000, 2999]$ and that the same string with one symbol removed appears again in $T[7000, 7998]$. For example assume that $\sigma = \sigma_1 a \sigma_2$ with $|\sigma_1| = 300$ and $|\sigma_2| = 699$ and that $T[7000, 7998] = \sigma_1 \sigma_2$. Since our search procedure only finds exact matches, σ_1 and σ_2 will be encoded in two different quintuplets q_1 and q_2. In q_1 the string σ_1 is encoded as a copy of $T[2000, 2299]$; in q_2 the string σ_2 is encoded as a copy of $T[2301, 2999]$. We observe that the quintuplet q_2 has the following features: *a)* the parameter U is zero, *b* the parameter P is equal to two plus the position where the copy of σ_1 ends. More in general, an approximate match induces in the quintuplet q_2 a small U and a value of P which is very close to the position of the last symbol copied in the previous quintuplet.

These observations suggest that the parameter P be encoded as an offset with respect to the last copied symbol. For example, suppose that the last quintuplet was an encoding of the fact that $T[900, 1000]$ is a copy of $T[200, 300]$. Next we find that $T[1010, 1050]$ is a copy of $T[320, 360]$. Instead of encoding $P = 320$ we encode $\Delta_P = P - 300 = 20$. Note that the offset Δ_P wraps around the last encoded position, in our example $T[1000]$. Hence, if we had $P = 100$ the corresponding offset would be $\Delta_P = (P - 300) \bmod 1000 = 800$. The definition of the offset Δ_P must be modified in the obvious way when we are working with reverse complements. For example, suppose that $T[900, 1000]$ is a reverse complement of $T[200, 300]$ and that $T[1010, 1050]$ is a reverse complement of $T[150, 190]$. In this case the offset Δ_P is defined as $\Delta_P = 200 - 190 = 10$.[1] In the rest of the paper we will make use of the parameter Δ_P defined above, and we will use it—instead of P—when we encode a quintuplet. As we observed in the previous paragraph each approximate match in the input file will generate a quintuplet with a small Δ_P.

3   A simple one pass algorithm

We now describe a first simple DNA compression algorithm, called dna0. This algorithm will be later improved in Sect. 4. dna0 uses the framework described in the previous section, that is, it encodes a sequence of quintuplets generated by our fingerprint-based search procedure. dna0 only encodes the parameters U, Δ_P and M of each quintuplet. The parameters r and α are stored without any compression using 1 bit for r and $2U$ bits for α.

---

[1]Obviously, if a quintuplet with $r = 1$ (reverse complement) follows a quintuplet with $r = 0$ (plain copy), or vice-versa, the above reasoning does not hold and we cannot hope Δ_P be small. However, experiments show that we loss nothing in using Δ_P rather than P even in these cases.

3.1 Encoding of the matched suffix length M

First we notice that by construction the length of the matched suffix is at least equal to the block size $B = 20$ used for computing the fingerprints (see Sect. 2.1). Hence, in our algorithm we encode the difference $M - 20$. We observed the distribution of the parameter M for some DNA files, and we found that the probability that $M = n$ is slowly decreasing as n increases. We have tried several textbook algorithms for the encoding of the integers (Elias, Golumb, etc.) but none of them performed satisfactorily. We have obtained the best results with a technique called *continuation bit encoding*.[2] To encode an integer $n \geq 0$ using this encoding we choose a parameter $k > 1$ and we split the binary representation $(n)_2$ of n into $\ell = \lceil |(n)_2| / (k-1) \rceil$ blocks of $k-1$ bits each. Then we append in front of each block a continuation bit which is zero for all blocks except for the last one. Finally, we concatenate the blocks obtaining an encoding of n of length $k\ell$. In the following we use $C_k(n)$ to denote this encoding. For example, for $k = 4$ we have (continuation bits are in boldface):

$$C_4(3) = \mathbf{1}011, \quad C_4(8) = \mathbf{0}001\,\mathbf{1}000, \quad C_4(15) = \mathbf{0}001\,\mathbf{1}111.$$

After experimenting with k ranging from 3 to 8, we found that the best overall performance is obtained with $k = 4$. Summing up, in the algorithm dna0 we encode the matched suffix length M with the binary string $C_4(M - 20)$.

3.2 Encoding of the unmatched prefix length U

Also for this parameter experiments show that the probability that $U = n$ is slowly decreasing as n increases. However, the distribution of U differs from the distribution of M: the smallest values ($U = 0$ and $U = 1$) are relatively more frequent. This is not surprising since we know that an approximate match in the input file generates a quintuplet with a small U (see Sect. 2.2). In our experiments we found that $U = 0$ with probability ≈ 0.25, $U = 1$ with probability ≈ 0.12 and that U was smaller than 10 with probability ≈ 0.5. This data suggests that we should use (roughly) 2 bits for encoding the event $U = 0$, three bits for $U = 1$, and that half of the code space should be used for the event $U < 10$. Accordingly, we decided to encode the integers in the range $[0, 9]$ with the following codewords:

$$0 \to \mathbf{00}, \quad 1 \to \mathbf{010}, \quad 2 \to \mathbf{011}\,000, \quad 3 \to \mathbf{011}\,001, \quad \cdots \quad 9 \to \mathbf{011}\,111 \tag{1}$$

(the bits in boldface only show the logic of the encoding, they have no other special meaning). Note that the codewords for the integers 0–9 all start by $\mathbf{0}$. For the encoding of the values $U \geq 10$ we use codewords starting with $\mathbf{1}$, in accordance with the experimental observation that $U \geq 10$ with probability ≈ 0.5. After a few tests, we found that the greatest compression is obtained encoding the values $U \geq 10$ with the bit $\mathbf{1}$ followed by the continuation bit encoding of $U - 10$ with $k = 4$. Summing up, in algorithm dna0 the unmatched prefix length U is encoded using the codewords in (1) when $U < 10$, and using the bit $\mathbf{1}$ followed by $C_4(U - 10)$ when $U \geq 10$.

3.3 Encoding of the offset Δ_P.

Recall that this parameter denotes a position in the already compressed portion of the input, say $T[1, z]$. As we observed in Sect. 2.2 each approximate match in T generates a quintuplet in

---

[2]We have not been able to trace the origin of this technique. We know that it has been used many times under many different names.

which Δ_P is a small positive integer. Unfortunately, most of the times we are not encoding an approximate match and in these cases the parameter Δ_P is uniformly distributed in the range $[1, z]$. We are therefore faced with the problem of encoding a parameter whose distribution is uniform with the exception of the smaller values (say $\Delta_P \leq 5$) which are relatively more frequent.

It is well known that an integer uniformly distributed in $[1, z]$ is optimally encoded using for each value a codeword of length $\lceil \log_2 z \rceil$ or $\lfloor \log_2 z \rfloor$ bits. If we assign shorter codewords to the smaller values we are then forced to use more that $\lceil \log_2 z \rceil$ bits for the larger values. We have tested several algorithms for assigning shorter codewords to the smaller values of Δ_P. It turned out that in most cases there was an increase of the overall output size with respect to the simple encoding which uses $\lceil \log_2 z \rceil$ bits for each value. This means that the extra bits required for encoding the larger values more than compensated the saving achieved for the encoding of the smaller values.

After some experiments, we found that the greatest space saving was obtained using a very conservative encoding which never uses more that $\lceil \log_2 z \rceil$ bits for a codeword, and at the same time uses less bits for the smaller values whenever this is possible. To see how this encoding works consider the following example. Suppose we have to encode a parameter t which can assume 76 possible values, say from 0 to 75. We have $\lceil \log_2 76 \rceil = 7$. We use 7-bit codewords for the higher 64 values in the range $[0,75]$. That is, if $t \in [12, 75]$ we encode t with a codeword consisting of the bit **1**, followed by the 6-bit binary representation of $t - 12$ (notice that $t - 12 \in [0, 63]$). If $t \in [0, 11]$ we encode it with a codeword consisting of the bit **0** followed by an encoding of t computed applying the above procedure recursively. That is, since $\lceil \log_2 12 \rceil = 4$, we use a 4 bit codeword for the values in the range $[4, 11]$: the bit **1** followed by the 3-bit binary representation of $t - 4$. If $t \in [0, 3]$ we encode it using the bit **0** followed by the 2-bit binary representation of t. Summing up, we have the following encoding:

$$0 \rightarrow \mathbf{00}00, \qquad 4 \rightarrow \mathbf{01}\,000, \qquad 12 \rightarrow \mathbf{1}000000,$$
$$1 \rightarrow \mathbf{00}01, \qquad 5 \rightarrow \mathbf{01}\,001, \qquad 13 \rightarrow \mathbf{1}000001,$$
$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$$
$$3 \rightarrow \mathbf{00}11, \qquad 11 \rightarrow \mathbf{01}\,111, \qquad 75 \rightarrow \mathbf{1}111111,$$

(the bits in boldface only show the logic of the encoding, they have no other special meaning). Note that we are encoding the values smaller than 12 using less than $\lfloor \log_2 76 \rfloor$ bits. We call this encoding *log-skewed*. Notice that if the initial range is 2^k we use k bits for all codewords. However, if the initial range is between 2^{k-1} and 2^k all codewords have length at most k, but the codewords for the smaller values have length less than k. We call this encoding *log-skewed* and we show a pseudo code computing it in Fig 1. Notice that the recursion stops when the current range is a power of two. In particular if the initial range is 2^k, this algorithm uses k bits for all codewords. However, if the initial range is between 2^{k-1} and 2^k all codewords will have length at most k, but the codewords for the smaller values will have length less than k.[3] Summing up, in algorithm dna0 we encode the offset Δ_P using the log-skewed encoding described in Fig. 1.

3.4   Preliminary experimental results

To test our compression algorithms we have built a collection of DNA sequences using four different organism: yeast (Saccharomyces Cerevisiae), mouse (Mus Musculus), arabidopsis (Arabidopsis

---

[3]We have not found this encoding described elsewhere in the literature, but we expect that it has been used many times by data compression practitioners.

Algorithm ENCODE(t, n)

1. **let** $k \leftarrow \lceil \log_2(n) \rceil$;
2. **if** $(n == 2^k)$
3.   WRITE(t, k);
4. **else**
5.   **let** $\delta \leftarrow n - 2^{k-1}$;
6.   **if** $(t \geq \delta)$
7.   WRITE$(1, 1)$;
8.   WRITE$(t - \delta, k - 1)$;
9.   **else**
10.   WRITE$(0, 1)$;
11.   ENCODE(t, δ);

Figure 1: Pseudo code for log-skewed encoding. The procedure WRITE(a, n) writes to the output stream the binary representation of a using n bits; it must be called with $0 \leq a < 2^n$.

yeast		mouse		arabidopsis		human	
Name	Size	Name	Size	Name	Size	Name	Size
y-4	1,531,929	*m-11*	49,909,125	*at-1*	29,830,437	*h-2*	236,268,154
y-14	784,328	*m-7*	5,114,647	*at-3*	23,465,336	*h-13*	95,206,001
y-1	230,203	*m-19*	703,729	*at-4*	17,550,033	*h-22*	33,821,688
y-mit	85,779	*m-x*	17,430,763			*h-x*	144,793,946
		m-y	711,108			*h-y*	22,668,225

Table 1: Collection of DNA sequences used for testing our compression algorithms.

Thaliana), and human. We have downloaded the complete genome of these organisms from [8], and we have extracted from each chromosome the sequence of bases $\{a, c, g, t\}$ discarding everything else. Then, we have taken the largest, the smallest, and the median chromosome of each organism together with the X and Y chromosomes (which are usually more compressible than the autosomes) and the mitochondrial DNA of yeast. The resulting test suite is summarized in Table 1; note that we have ordered the four organisms according to the average size of their chromosomes.

In Fig. 2 we report the compression ratio and compression speed achieved by gzip, bzip2, order 2 and order 3 arithmetic coding (taken from [16]), and by our algorithm dna0. The compression ratio is expressed in terms of bits per symbol. Since the input files only contains four different symbols, we achieve a space reduction when the number of bits per symbol is less than 2. We can see that the standard compression tools gzip and bzip2 uses more than two bits per symbol for all files except *y-mit*. For what concern their running time (expressed in microseconds per input symbol) we observe the usual behavior: gzip is faster than bzip2 in compression and much faster in decompression. Order 2 and order 3 arithmetic coding both compress to less than two bits per symbol; they are roughly two times slower than gzip in compression and their decompression speed is very close to the compression speed.

The results for dna0 show that it compresses better than arithmetic coding for all files except the smallest ones (the yeast chromosomes and *m-19*). For what concerns the compression speed our algorithm is faster than bzip2 for all files except the largest ones (the human chromosomes). Note however than the time for input byte spent by dna0 grows very slowly with the size of the input; this is a remarkable feature in an algorithm which can find a repetition which occurs anywhere in the already scanned portion of the input. The decompression speed of our algorithm

is roughly two times faster than `bzip2`, and roughly ten times faster than arithmetic coding.

4   Improving the compression

In this section we show how to improve the compression ratio of our algorithm at the cost of a little increase in the running time. Our first improvement consists in doing two passes over the data. In the first one we compute some statistics on the distribution of the parameters M and U and in the second pass we do the actual compression. Consider for example the parameter U. Our idea is to encode the values $U = i$ for $i = 0, 1, \ldots, 254$ using a properly designed Huffman code, and to encode the values $U \geq 255$ using the continuation bit encoding (see Sect. 3.1) implemented as follows: for encoding a value $U \geq 255$ we use the Huffman code for the event $U \geq 255$ followed by codeword $C_k(U - 255)$ for a properly chosen k. More in detail, in the first pass we compute:

1. the number of quintuplets with $U = i$, for $i = 0, 1, \ldots, 254$; and the number of quintuplets with $U \geq 255$;

2. for the quintuplets with $U \geq 255$ the cost (number of bits) of encoding $C_k(U - 255)$ for $k = 3, \ldots, 8$.

At the end of the first pass, we compute a Huffman code for the events $\{U = 0, U = 1, \ldots, U = 254, U \geq 255\}$, and we compute the value of k which minimizes the cost of encoding the values $U \geq 255$. In the second pass we encode the parameter U using the combination of Huffman and continuation bit encoding described above.

For the parameter M we follow the same approach. The only difference is that by construction $M \geq 20$; hence we apply the above procedure to the value $M - 20$. In other words, we encode the values $M = i$ for $i = 20, 21, \ldots, 274$ using a properly designed Huffman code, and we encode the values $M \geq 275$ using the continuation bit encoding with the parameter k which minimizes the overall codeword length (the parameter k used for M is not necessarily the same used for U).

We call `dna1` the algorithm which encodes U and M with this two-pass procedure and encodes the parameter Δ_P using the log-skewed procedure described in Sect. 3.3. The performances of `dna1` are reported in Fig. 2. We can see that `dna1` achieves a better compression than `dna0` for all files except the smallest ones (the yeast sequences). We can also see that the difference in compression ratio is not spectacular: this suggests that the hand-tuned codewords used in `dna0` have been well chosen. The good news is that although `dna1` is a two pass algorithm it is almost as fast as `dna0` both in compression and decompression.

To further improve the compression ratio we observe that so far we considered only the encoding of the parameters U, P and M which appear in each quintuplet $\langle U, r, P, M, \alpha \rangle$. In other words, our algorithms do not compress the bit r and the unmatched prefix α. Since unmatched prefixes can be rather long and we know that arithmetic coding can compress DNA sequences to less than two bits per symbol, it makes sense to compress all unmatched prefixes using arithmetic coding. This idea was already present in the first paper on DNA compression [9] and has been used also in other algorithms. We call `dna2` (resp. `dna3`) the algorithm which encodes the parameters U, Δ_P and M as in `dna1` and compresses the unmatched prefixes using order 2 (resp. order 3) arithmetic coding. In our implementation we have used the arithmetic coding routines described in [16], whose performance are reported in Fig. 2 under the names `ac-o2` and `ac-o3`.

Figure 2 shows the performance of `dna2` and `dna3` on our suite of test files. We can see that the introduction of arithmetic coding greatly improves the compression: on every file of the collection

	y-4	*y-14*	*y-1*	*y-mit*	Ave.	Speed	
gzip	2.313	2.333	2.303	1.920	2.305	0.65	0.03
bzip2	2.163	2.172	2.170	1.787	2.154	0.86	0.34
ac-o2	1.946	1.951	1.953	1.560	1.936	1.28	1.28
ac-o3	1.944	1.950	1.955	1.547	1.934	1.28	1.27
dna0	1.975	1.908	1.935	1.905	1.943	0.72	0.14
dna1	1.975	1.911	1.936	1.912	1.944	0.71	0.14
dna2	1.928	1.869	1.884	1.526	1.884	1.90	1.33
dna3	1.926	1.871	1.881	1.523	1.882	1.91	1.33

	m-11	*m-7*	*m-19*	*m-x*	*m-y*	Ave.	Speed	
gzip	2.257	2.257	2.259	2.248	2.218	2.254	0.62	0.03
bzip2	2.087	2.091	2.095	2.036	2.001	2.075	0.89	0.33
ac-o2	1.920	1.918	1.923	1.911	1.904	1.918	1.27	1.27
ac-o3	1.920	1.919	1.926	1.912	1.908	1.918	1.28	1.27
dna0	1.857	1.907	1.952	1.777	1.782	1.841	0.88	0.14
dna1	1.853	1.905	1.952	1.772	1.781	1.838	0.90	0.14
dna2	1.789	1.834	1.884	1.702	1.704	1.772	1.93	1.18
dna3	1.790	1.835	1.888	1.703	1.707	1.772	1.95	1.18

	at-1	*at-3*	*at-4*	Ave.	Speed	
gzip	2.266	2.266	2.259	2.264	0.63	0.03
bzip2	2.129	2.129	2.129	2.129	0.89	0.33
ac-o2	1.923	1.926	1.923	1.924	1.27	1.27
ac-o3	1.921	1.924	1.922	1.922	1.28	1.27
dna0	1.917	1.913	1.924	1.918	0.87	0.14
dna1	1.916	1.911	1.923	1.916	0.88	0.14
dna2	1.845	1.844	1.853	1.847	2.01	1.28
dna3	1.844	1.843	1.851	1.845	2.02	1.29
DnaC	1.783	1.769	1.787	1.779	71.34	—

	h-2	*h-12*	*h-22*	*h-x*	*h-y*	Ave.	Speed	
gzip	2.260	2.264	2.222	2.252	2.198	2.253	0.62	0.03
bzip2	2.079	2.087	2.017	2.051	2.007	2.066	0.89	0.33
ac-o2	1.912	1.908	1.920	1.910	1.898	1.911	1.27	1.27
ac-o3	1.910	1.906	1.915	1.909	1.895	1.909	1.28	1.27
dna0	1.869	1.904	1.835	1.808	1.478	1.840	0.97	0.14
dna1	1.863	1.900	1.829	1.801	1.472	1.834	0.99	0.15
dna2	1.790	1.818	1.767	1.732	1.411	1.762	1.97	1.13
dna3	1.790	1.818	1.767	1.732	1.411	1.762	1.98	1.13

Figure 2: Compression ratio and speed for the DNA sequences shown in Table 1. From top to bottom we have the results for yeast, mouse, arabidopsis, and human. Each row displays the result for a single algorithm and shows the compression ratio for each sequence and the average compression ratio in bits per symbol. The last two columns show the average compression and decompression speed in microseconds per input byte (average computed over five runs for each sequence). All tests (except the ones for DnaC) have been carried out on a 1GHz Pentium 3 with 1GB of main memory under the GNU/Linux operative system. The compiler was gcc ver. 3.2 with options -O3 -fomit-frame-pointer -march=pentium3.

dna2 and dna3 achieve a better compression than dna1, ac-o2, and ac-o3. On the largest files of the collection (mouse, arabidopsis, and human) the advantage of dna2 and dna3 over ac-o2 and ac-o3 is roughly of 0.15 bits per input symbol and the advantage over dna1 roughly of 0.7 bits per symbol. For what concerns the compression and decompression speed, dna2 and dna3 are slower that the other algorithms, but their performances are still close to the standard compressors of everyday use. For example, dna2 and dna3 are roughly two times slower in compression and four times slower in decompression than the bzip2 tool. Finally, we note that the difference between dna2 and dna3 is very small, both in terms of compression ratio and in terms of compression/decompression speed.

4.1 Comparison with other DNA compressors

The most natural benchmark for our algorithms is a comparison with the other algorithms designed to compress DNA sequences. Unfortunately, to carry out such a comparison turned out to be a rather difficult task. The first reason is that the source or executable code of DNA compressors are usually not available (the only exception being Off-line whose source code is available under GPL [13]). The second reason is that the huge space and time requirements of most DNA compressors make it difficult (or impossible) to test them on sequences of significant length. For these reasons this section is not complete as we would like it to be, but we hope to extend it in the full paper (see below).

Table 2 shows the compression ratios of several algorithms on a set of DNA sequences which have been used for testing in almost every paper on DNA compression. From Table 2 we can see that dna2 and dna3 achieves a compression which is slightly worse that the other DNA compressors. However, the difference is rather small with the only exception of the file *humgh* (sequence HUMGHCSA). Table 2 also tells us that DNACompress [6] is the algorithm with the best compression performances: only for three files it is outperformed by CTW+LZ [14] and only by a very narrow margin.

For what concern the running times we do not have complete data. In [6] the authors report some compression times for the file *hehcm* (229354 bases) on a 700MHz Pentium III. According to [6] the compression of *hehcm* takes a few hours for CTW+LZ, 51 seconds for GenCompress-2 [5], and 4 seconds for DNACompress. According to our tests, on a 1000MHz Pentium III the compression of *hehcm* takes 0.42 seconds for dna2, and 1719.79 seconds for Off-line.

Although this data is certainly incomplete, it shows that the current leader in DNA compression is the algorithm DNACompress. It achieves the best compression ratios and, together with our algorithms, it is the only one with "reasonable" compression times. Unfortunately, we could not test in depth the performance of DNACompress since it is based on the PatternHunter [7] search engine which is not freely available.[4] One of the authors of DNACompress has kindly provided the compression ratios and speed for the arabidopsis sequences [4]. This data is reported in the last row of the third table of Fig. 2. We can see that on average DNACompress uses 0.066 bits per symbol less than dna3. This translates to a smaller space occupancy of 8.25 bytes every 1000 input bytes. This space saving is achieved at the cost of a much greater compression time. On a 2GHz Pentium 4 the average compression speed of DNACompress is $71.34\mu s$ per input byte. This is an order of magnitude larger than the $2.02\mu s$ per input byte achieved by dna3 on a 1GHz Pentium 3. It is interesting to compare also the working space of the two algorithms. For a sequence of

---

[4]We have applied for an academic license of PatternHunter and we hope to be able to give a complete report on the performance of DNACompress in the full paper.

	chmpxx	chntxx	hehcm	humdy	humgh	humhb	humhd	humhp	mpom	mtpa	vaccg
size	121024	155844	229354	38770	66495	73308	58864	56737	186609	100314	191737
gzip	2.2818	2.3345	2.3275	2.3618	2.0648	2.2450	2.2389	2.2662	2.3288	2.2919	2.2518
bzip2	2.1218	2.1845	2.1685	2.1802	1.7289	2.1481	2.0678	2.0944	2.1701	2.1225	2.0949
ac-o2	1.8364	1.9333	1.9647	1.9235	1.9377	1.9176	1.9422	1.9283	1.9654	1.8723	1.9040
ac-o3	1.8425	1.9399	1.9619	1.9446	1.9416	1.9305	1.9466	1.9352	1.9689	1.8761	1.9064
off-line	1.9022	1.9985	2.0157	2.0682	1.5993	1.9697	1.9740	1.9836	1.9867	1.9155	1.9075
BioC	1.6848	1.6172	1.848	1.9262	1.3074	1.88	1.877	1.9066	1.9378	1.8752	1.7614
GenC	1.673	1.6146	1.847	1.9231	1.0969	1.8204	1.8192	1.8466	1.9058	1.8624	1.7614
ctw+lz	1.6690	1.6129	1.8414	1.9175	1.0972	1.8082	1.8218	1.8433	1.9000	1.8555	1.7616
DnaC	1.6716	1.6127	1.8492	1.9116	1.0272	1.7897	1.7951	1.8165	1.8920	1.8556	1.7580
dna0	1.8320	1.6758	1.8815	2.0034	1.3860	1.9398	1.9441	1.9691	1.9567	1.9936	1.8429
dna1	1.8333	1.6765	1.8819	2.0057	1.3946	1.9451	1.9512	1.9760	1.9599	1.9956	1.8440
dna2	1.6733	1.6162	1.8487	1.9326	1.3668	1.8677	1.9036	1.9104	1.9275	1.8696	1.7634
dna3	1.6782	1.6223	1.8463	1.9533	1.3750	1.8807	1.9130	1.9199	1.9312	1.8735	1.7645

Table 2: Compression ratio (bits per input symbol) for gzip, bzip2, arithmetic coding, our algorithms, and for the following DNA compression algorithms: Off-line [2] (off-line), Biocompress2 [9] (BioC), Gencompress-2 [5] (GenC), CTW+LZ [14] (ctw+lz), DNACompress [6] (DnaC).

length N DNACompress uses $6N$ bytes of working space whereas our algorithm only uses $7N/5$ bytes (see Sect. 2.1). This smaller space occupancy makes it possible to handle much larger files.

Finally, we would like to comment on the performance of Off-line which, like our algorithm, only encodes exact repeats. While we use a simple and fast procedure to find repeats, Off-line is designed to search for an "optimal" sequence of textual substitutions. It is therefore not surprising that Off-line is much slower than our algorithm (for the yeast sequences the difference is by a factor 1000 or more). The relatively poor compression achieved by Off-line (see Table 2) is due to the fact that Off-line does not consider palindromes and does not use arithmetic coding. Indeed, rather than a DNA-specific compressor Off-line is a general purpose compressor which works reasonably well also for DNA sequences.

Acknowledgments

References

[1] L. Allison, L. Stern, T. Edgoose, and T. I. Dix. Sequence complexity for biological sequence analysis. *Computers and Chemistry*, 24(1):43–55, 2000.

[2] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In J. A. Storer and M. Cohn, editors, *Proceedings Data Compression Conference*, pages 143–152, Snowbird, UT, 2000. IEEE Computer Society Press.

[3] J. Bentley and D. McIlroy. Data compression with long repeated strings. *Information Sciences*, 135:1–11, 2001.

[4] X. Chen. Personal communication.

[5] X. Chen, S. Kwong, and M. Li. A compression algorithm for DNA sequences and its applications in genome comparison. In *Proceedings of the 4th Annual International Conference on Computational Molecular Biology (RECOMB-00)*. ACM Press, 2000.

[6] X. Chen, M. Li, B. Ma, and J. Tromp. DNACompress: Fast and effective DNA sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.

[7] X. Chen, M. Li, B. Ma, and J. Tromp. PatternHunter—fast and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[8] National Center for Biotechnology Information. `http://www.ncbi.nih.gov`.

[9] S. Grumbach and F. Tahi. A new challenge for compression algorithms: genetic sequences. *Inf. Proc. and Management*, 30(6):875–886, 1994.

[10] R. Karp and M. Rabin. An efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[11] J. K. Lanctot, M. Li, and E. Yang. Estimating DNA sequence entropy. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 409–418, 2000.

[12] M. Li, J. H. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17(2):149–154, February 2001.

[13] S. Lonardi. Off-line home page. `http://www.cs.ucr.edu/ stelo/Offline`.

[14] T. Matsumoto, K. Sadakane, and H. Imai. Biological sequence compression algorithms. In *Proc. Genome Informatics Workshop*, pages 43–52. Universal Academy Press, Tokyo, 2000.

[15] A. Milosavjevic. Discovering by minimal length encoding: A case study in molecular evolution. *Machine Learning*, 12:68–87, 1993.

[16] M. Nelson. Arithmetic coding + statistical modeling = data compression. *Dr. Dobb's Journal of Software Tools*, 16(2):16–18, 1991. `http://dogma.net/markn/articles/arith/part1.htm`.

[17] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.