# A Simple and Fast DNA Compressor

Giovanni Manzini[*] Marcella Rastero[†]

February 17, 2004

**Abstract**

In this paper we consider the problem of DNA compression. 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 tasks 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 [12], to evaluate the "distance" between DNA sequences and to quantify how much two organisms are "close" in the evolutionary tree [7, 13], and in other biological applications [1, 4, 17].

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 bzip, usually fail to achieve any compression since they use more than two bits per symbol. Among the "general purpose" compression algorithms only arithmetic coding [20] is able to consistently use less than two bits per symbol. In [10] was introduced the first DNA-specific compression algorithm and others followed afterwards [2, 7, 16]. 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 [16] the authors report a running time of 8 minutes

[*]Dipartimento di Informatica, Università del Piemonte Orientale, Italy. Email: `manzini@mfn.unipmn.it`. Partially supported by the Italian MIUR project "Algorithmics for Internet and the Web (ALINWEB)".

[†]Dipartimento di Scienze e Tecnologie Avanzate, Università del Piemonte Orientale, Italy.

for a sequence of length 38KB, and a running time of some hours for a 224KB sequence. We are therefore very far from the speed of the tools that we use everyday for general purpose compression (*e.g.* gzip and bzip).

Recently, a new algorithm called DNACompress [8] has made significant improvements over its predecessors. Thanks to an extremely fast search engine [15], 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.

A common feature of the above compressors, with the exception of [2], is that they encode 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 substitution, insertion, or deletion 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.

The strategy of working with exact repeats 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 bzip. 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. However, 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.

The idea of compressing DNA sequences using only exact repeats has been explored also by the algorithm Off-line described in [2, 3]. However, our approach and the one proposed in [2, 3] are completely different. While we use a simple and fast procedure to find repeats, Off-line is designed to search for an "optimal" sequence of textual substitutions. Moreover, our algorithm was designed with DNA compression in mind, whereas Off-line is a general purpose compressor. Given these premises, it is not surprising that our algorithm is much faster and achieves a better compression than Off-line (see Section 4.1 for details).

Finally, we point out that the main objective of this paper is to show the potentiality of using exact repeats for DNA compression. Therefore, we designed our algorithm using known techniques as building blocks, and we did not optimize some parameters that influence speed and compression ratio. We believe that having achieved a very good compression/speed tradeoff without any algorithmic engineering is a further indication of the validity of the "exact repeat" approach for DNA compression.

## 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 $\alpha, \beta$ 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 $\alpha$ is a *reverse complement* of $\beta$, and we write $\alpha = \bar{\beta}$ if $\alpha$ and the reverse of $\beta$ 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 bzip 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 sequences are often longer than in linguistic texts, but they are less frequent.[1] 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, pairs of substrings which are the reverse complement of each other. The characteristics of the repeated palindromes are the same as 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 repeats/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 [11] 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 coefficients of a polynomial modulo a large prime. The pattern fingerprint is then compared

[1]To be more precise, we should say that DNA sequences contain less repetitions whose length is statistically significant. This statement is proven empirically by the fact that traditional compressors which work by succinctly encoding repeated substrings fail to compress DNA sequences.

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 [5] 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 [5].

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*. Each time 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 as 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 [5] 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 larger 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 $T[1, z]$. We are now ready to encode the string $T[z+1, z+a+\ell]$ which consists of an *unmatched prefix* $\alpha = T[z+1, z+a]$, and of a *matched suffix* $\beta = T[z+a+1, z+a+\ell]$.

Note, however, that the current block $T[z+b, z+b+B-1]$ could match more than one stored block. When this happens we extend all matches and we select the one with the *longest* matched suffix $\beta$; in case of ties, we select the match with the *shortest* unmatched prefix $\alpha$. When a match has been finally selected, we represent the string $\alpha\beta = T[z+1, z+a+\ell]$ 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 $\beta$ 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 $\beta$ (or of $\bar{\beta}$ if $r = 1$);
4. $M = |\beta|$ is the length of the matched suffix;
5. $\alpha$ 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 $\alpha$ are stored without any compression using 1 bit for $r$ and $2U$ bits for $\alpha$.

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 $\sigma$ of length 1000 appears as $T[2000, 2999]$ and that the same string with one symbol removed appears again as $T[7000, 7998]$. 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, $\sigma_1$ and $\sigma_2$ will be encoded by two different quintuplets $q_1$ and $q_2$. In $q_1$ the string $\sigma_1$ is encoded as a copy of $T[2000, 2299]$; in $q_2$ the string $\sigma_2$ is encoded as a copy of $T[2301, 2999]$. We observe that the quintuplet $q_2$ has the following features: 1) the parameter $U$ is zero, 2) the parameter $P$ is equal to two plus the position where the copy of $\sigma_1$ 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 has encoded the fact that $T[900, 1000]$ is a copy of $T[200, 300]$. Then we find that $T[1010, 1050]$ is a copy of $T[320, 360]$. Instead of encoding $P = 320$ we encode the offset $\Delta_P = P - 300 = 20$. To be able to address any position in the already scanned input we assume that the offset $\Delta_P$ 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 $\Delta_P$ 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 $\Delta_P$ is defined as $\Delta_P = 200 - 190 = 10$. When working with reverse complements we assume that the offset $\Delta_P$ wraps around the position $T[1]$. In the previous example, if we find that $T[1010, 1050]$ is a reverse complement of $T[450, 490]$ the offset would be $\Delta_P = 200 - 490 \bmod 1000 = 710$.[2]

In the rest of the paper we will make use of the parameter $\Delta_P$ 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 $\Delta_P$.

## 3 A simple one pass algorithm

In this section we 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 the sequence of quintuplets generated by our fingerprint-based search procedure. dna0 only encodes the parameters $U$, $\Delta_P$ and $M$ of each quintuplet. The parameters $r$ and $\alpha$ are stored without any compression using 1 bit for $r$ and $2U$ bits for $\alpha$.

### 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.*[3] To encode an integer $n \geq 0$ using this encoding we choose a parameter $k > 1$ and we split the binary representation $(n)_\mathbf{2}$ of $n$ into $\ell = \lceil |(n)_\mathbf{2}|/(k-1) \rceil$ 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 $\approx 0.25$, $U = 1$ with probability $\approx 0.12$ and that

[2]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 $\Delta_P$ be small. However, experiments show that we lose nothing in using $\Delta_P$ rather than $P$ even in these cases.

[3]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, see for example [18].

$U$ was smaller than 10 with probability $\approx 0.5$. 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 **0**. For the encoding of the values $U \geq 10$ we use codewords starting with **1**, in accordance with the experimental observation that $U \geq 10$ with probability $\approx 0.5$. After a few tests, we found that the greatest compression is obtained encoding the values $U \geq 10$ with the bit **1** followed by the continuation bit encoding of $U - 10$ with $k = 4$. Summing up, in algorithm dna0 we encode the unmatched prefix length $U$ using the codewords in (1) when $U < 10$, and using the bit **1** followed by $C_4(U - 10)$ when $U \geq 10$.

## 3.3 Encoding of the offset $\Delta_P$

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 which $\Delta_P$ is a small positive integer. Unfortunately, most of the times we are not encoding an approximate match and in these cases the parameter $\Delta_P$ 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 $\Delta_P$. 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 saving achieved in the encoding of the smaller values is more than compensated by the extra bits required for the larger 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, 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 the 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:

$$\begin{array}{lll}
0 \to \mathbf{00}00, & 4 \to \mathbf{01}\;000, & 12 \to \mathbf{1}000000, \\
1 \to \mathbf{00}01, & 5 \to \mathbf{01}\;001, & 13 \to \mathbf{1}000001, \\
\quad\vdots & \quad\vdots & \quad\vdots \\
3 \to \mathbf{00}11, & 11 \to \mathbf{01}\;111, & 75 \to \mathbf{1}111111,
\end{array}$$

```
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$(t, \delta)$;
```

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. The complete dataset, including the files used for the tests in Table 2, is available at http://www.mfn.unipmn.it/~manzini/dnacorpus/.

(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* 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.

Summing up, in algorithm dna0 we encode the offset $\Delta_P$ 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 organisms: yeast (Saccharomyces Cerevisiae), mouse (Mus Musculus), arabidopsis (Arabidopsis Thaliana), and human. We have downloaded the complete genome of these organisms from [9], 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. All tests have been carried out on a 1GHz Pentium III with 1GB of main memory running GNU/Linux. The compiler was gcc ver. 3.2 with options -O3 -fomit-frame-pointer -march=pentium3.

In Fig. 2 we report the compression ratio and compression speed achieved by several known compression tools and by the algorithms described in this paper. The compression ratio is expressed in terms of bits per symbol. Since the input files only contain four different symbols, we achieve a real compression when the number of bits per symbol is less than 2.

The first two rows of each table show the performance of the standard compression tools gzip and bzip. We can see that these tools use more than two bits per symbol for all files except *y-mit*. Their running times (expressed in microseconds per input byte) show the usual behavior: gzip is faster than bzip in compression and much faster in decompression.

The next two rows of each table (labelled ac-o2 and ac-o3) report the performance of the order 2 and order 3 arithmetic coding routines described in [20]. We can see that 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 their compression speed.

Since the tools considered so far are general purpose compressors, we can try to "help" them in the compression of DNA sequences with a preprocessing step in which we store four bases in one byte. We then feed the output of this "4-in-1 preprocessing" to gzip, bzip and arithmetic coding. As a result, these tools now operate on input files which have already achieved a two bits per symbol compression. In Fig. 2 we report the performance of gzip and bzip with the 4-in-1 preprocessing under the labels gzip-4 and bzip-4. We can see that gzip-4 is extremely fast and consistently uses less than 2 bits per symbol. Compared to arithmetic coding it is roughly eight times faster and achieves a better average compression ratio for mouse, arabidopsis, and human. The performance of bzip-4 is less spectacular. Note that although bzip compresses significantly better than gzip, gzip-4 outperforms bzip-4 on all files except *m-x*, *h-22*, *h-x*, and *h-y*. We do not report in Fig. 2 the performance of arithmetic coding with the 4-in-1 preprocessing since this combination consistently produced *more* that two bits per symbol. In other words, the 4-in-1 preprocessing degraded the performance of order 2 and order 3 arithmetic coding.

The results for our algorithm dna0 show that it compresses better than arithmetic coding for all files except the smallest ones (the yeast chromosomes and *m-19*). The comparison between dna0 and gzip-4 is less straightforward. If we consider the average compression ratio for each collection we can see that: (1) gzip-4 is marginally better for yeast, (2) the two algorithms have the same performance for arabidopsis, and (3) dna0 is superior for mouse and human. If we look at the compression of the individual sequences we notice that dna0 is usually superior for the more compressible ones, but this rule has the exception of the highly compressible sequence *y-mit* for which gzip-4 outperforms dna0.

The data in Fig. 2 show that dna0 has roughly the same compression speed as gzip and bzip. Note that 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 that can find repetitions occurring anywhere in the already scanned portion of the input. In terms of decompression speed dna0 is the fastest algorithm after gzip and gzip-4.

## 4 Improving the compression

In this section we show how to improve the compression ratio of dna0 at the cost of a small 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 $i = 0, 1, \dots, 254$ 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

| | *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 |
| bzip | 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 |
| gzip-4 | 1.950 | 1.961 | 1.951 | 1.597 | 1.942 | 0.16 | 0.13 |
| bzip-4 | 1.994 | 2.002 | 2.006 | 1.686 | 1.987 | 0.43 | 0.23 |
| 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 |
| bzip | 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 |
| gzip-4 | 1.916 | 1.916 | 1.924 | 1.882 | 1.881 | 1.908 | 0.15 | 0.11 |
| bzip-4 | 1.923 | 1.930 | 1.951 | 1.865 | 1.903 | 1.910 | 0.40 | 0.22 |
| 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 |
| bzip | 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 |
| gzip-4 | 1.917 | 1.921 | 1.916 | 1.918 | 0.15 | 0.11 |
| bzip-4 | 1.975 | 1.970 | 1.973 | 1.973 | 0.41 | 0.22 |
| 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-13* | *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 |
| bzip | 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 |
| gzip-4 | 1.909 | 1.909 | 1.885 | 1.898 | 1.860 | 1.902 | 0.15 | 0.11 |
| bzip-4 | 1.913 | 1.919 | 1.867 | 1.889 | 1.855 | 1.902 | 0.40 | 0.22 |
| 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 showing 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).

follows: for encoding a value $U \geq 255$ we use the Huffman code for the event $U \geq 255$ followed by the 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 the parameter $k$ that minimizes the cost of encoding the values $U \geq 255$. The optimal $k$ and a canonical Huffman tree [19, Sect. 2.3] representing the codewords for the events $\{U = 0,\ \ldots,\ U = 254,\ U \geq 255\}$ are then written to the output file. 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. Again, at the end of the first pass we write to the output file the optimal $k$ and a canonical Huffman tree representing the chosen codewords.

We call dna1 the algorithm which encodes $U$ and $M$ with this two-pass procedure and encodes the parameter $\Delta_P$ using the log-skewed encoding described in Sect. 3.3. The performance of dna1 is 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 $\alpha$. 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 [10] and has been used also in other algorithms. We call dna2 (resp. dna3) the algorithm which encodes the parameters $U, \Delta_P$ 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 [20], whose performance is reported in Fig. 2 under the labels ac-o2 and ac-o3.

Fig. 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: for every file of the collection dna2 and dna3 achieve a better compression than dna1, ac-o2, ac-o3, and gzip-4. The advantage of dna2 and dna3 over ac-o2, ac-o3, and gzip-4 is roughly 0.15 bits per symbol, and the advantage over dna1 is roughly 0.07 bits per symbol. Concerning the compression and decompression speed, we observe that dna2 and dna3 are slower that the other algorithms but their performance is 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 bzip. 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.

|  | 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 |
| bzip | 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 |
| gzip-4 | 1.8635 | 1.9519 | 1.9817 | 1.9473 | 1.7372 | 1.8963 | 1.9141 | 1.9207 | 1.9727 | 1.8827 | 1.8741 |
| bzip-4 | 1.9667 | 2.0090 | 2.0091 | 2.0678 | 1.8697 | 1.9957 | 1.9921 | 2.0045 | 2.0117 | 1.9847 | 1.9520 |
| 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 |
| 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 |

Table 2: Compression ratios (bits per input symbol) for gzip, bzip, arithmetic coding, gzip and bzip with the 4-in-1 preprocessing, our algorithms, and for the following DNA compression algorithms: Off-line [2] (off-line), Biocompress2 [10] (BioC), Gencompress-2 [7] (GenC), CTW+LZ [16] (ctw+lz), DNACompress [8] (DnaC).

## 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, 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 GNU GPL [14]). 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.

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 achieve 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 [8] is the algorithm with the best compression performance: only for three files it is outperformed by CTW+LZ [16] and only by a very narrow margin.

We do not have complete data on the running times. In [8] the authors report some compression times for the file *hehcm* (229354 bases) on a 700MHz Pentium III. According to [8] the compression of *hehcm* takes a few hours for CTW+LZ, 51 seconds for GenCompress-2 [7], and 4 seconds for DNACompress. According to our tests, on a 1GHz Pentium III the compression of *hehcm* takes 1719.79 seconds for Off-line and 0.42 seconds for dna2 which was the slowest of our algorithms.

This data 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 [15] search engine which is not freely available. One of the authors of DNACompress has kindly provided the compression ratios and speed for the arabidopsis sequences [6]. 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 III. It is interesting to compare also the working space of the two algorithms. For a sequence of length $N$ DNACompress uses $6N$ bytes of working space whereas our algorithms only use $7N/5$ bytes (see Sect. 2.1). This smaller space occupancy makes it possible to handle much larger files.

Note that we can further reduce the space occupancy of our algorithms by increasing the parameter $B$ (the size of the blocks stored in the hash table). For example, if we set $B = 24$ the space occupancy drops to $4N/3$ bytes. A larger block size $B$ means that our algorithms run faster (we access less frequently the hash table) but also that they detect a smaller number of repeated substrings. This usually reduces the compression ratio, but not for every sequence: for the human chromosomes we get a better average compression for $B = 24$ rather than for $B = 20$. A possible explanation of this phenomenon is that when we use a larger $B$ we miss some shorter repetitions but we also need less bits to encode the matched suffix length of each quintuplet (see Sect. 3.1).

Finally, we would like to comment on the performance of Off-line which, like our algorithms, 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 algorithms (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, the comparison between our algorithms and Off-line is somewhat unfair since Off-line is not a DNA-specific compressor but rather a general purpose compressor which works reasonably well also for DNA sequences.

## 5 Conclusions and further works

In this paper we have investigated the possibility of compressing DNA sequences working only with exact matches. In other words, our algorithms only search and encode repeated substrings and palindromes with no errors. This is a departure from the common practice in DNA compression of searching and encoding also approximate repeats.

The experimental results show that our algorithms are an order of magnitude faster than the other DNA compressors and that their compression ratio is very close to the one of the best compressors. Thanks to the speed of our algorithms and to their small working space we have been able to compress sequences of length up to 220MB, which are well beyond the range of any other DNA compressor. These results have been obtained using simple and non-optimized algorithms and we believe they prove the validity of the "exact repeat" approach to DNA compression.

In developing our algorithms we had to face several design decisions, especially concerning the encoding of the exact matches (Sect. 3). In most cases our decisions were guided by our intuition and by preliminary experimental results. However, we are aware that other (better) strategies are possible. In particular, we believe that one could improve the compression ratio by using two sets of codes: one for "isolated" repeats, and the other for exact repeats that are likely to belong to a larger approximate repeat. The compressor should estimate, on the basis of the recent history, the probability that it is currently encoding an approximate repeat and use the appropriate code accordingly.

## Acknowledgments

We would like to thank Xin Chen and Stefano Lonardi for their assistance in the testing of their compression algorithms.

## 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] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proceedings of the IEEE*, 88(11):1733–1744, November 2000.

[4] C. Bennet, M. Li, and B. Ma. Chain letters and evolutionary histories. *Scientific American*, pages 32–37, June 2003.

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

[6] X. Chen. Personal communication.

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

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

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

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

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

[18] H. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[19] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

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