DNA Compression Challenge Revisited

Behshad Behzadi and Fabrice Le Fessant

LIX, Ecole Polytechnique, Palaiseau cedex 91128, France
{Behzadi, Lefessan}@lix.polytechnique.fr

Abstract. Standard compression algorithms are not able to compress DNA sequences. Recently, new algorithms have been introduced specifically for this purpose, often using detection of long approximate repeats. In this paper, we present another algorithm, *DNAPack*, based on dynamic programming. In comparison with former existing programs, it compresses DNA slighly better, while the cost of dynamic programming is almost neglectible.

1 Introduction

DNA sequences contain only four bases $\{A, C, T, G\}$. Thus, each base (symbol) can be represented by two bits. However, the standard text compression tools, such as `compress`, `gzip` and `bzip2`, cannot compress these DNA sequences; the size of the files encoded with these tools is larger than two bits per symbol. Consequently, DNA sequences compression has recently become a challenge.

Some characteristics of DNA sequences show that they are not random sequences. If these sequences were totally random, the most efficient and logical way to store them would be using two bits per base. However, DNA is used for the expression of proteins in living organisms, and thus must contain some logical organisation. Moreover, approximate repeats (repeats with mutations) and complementary palindromes (reversed repeats, where A and C are respectively replaced by T and G, and reciprocally) are well-known to frequently appear inside long DNA sequences.

Based on these characteristics, several algorithms have been proposed for the compression of DNA sequences. However, even if these algorithms obtain much better results than standard universal compression algorithms, the compression ratios are not very high. We briefly discuss some of the existing algorithms in the order of their introduction (from the oldest to the most recent ones):

Biocompress [8], and its second version *Biocompress-2* [9], were the first DNA-specific compression algorithms. They are similar to the Ziv-Lampel data compression method. *Biocompress-2* detects exact repeats and complementary palindromes located in the already encoded sequence, and then encodes them by the repeat length and the position of a previous repeat occurrence. When no significant repetition is found, *Biocompress-2* uses order-2 arithmetic coding (Arith-2).

The *Cfact* algorithm [13] looks for the longest exact matching repeat. For this purpose, it uses a suffix tree on the entire sequence. Using two passes, repetitions are encoded this way when the gain is guaranteed, otherwise the two-bits-per-base (2-Bits) encoding is used.

The *GenCompress* algorithm [3, 4, 10] yields to a significantly better compression ratio than the previous algorithms. The idea is to use approximate (and not exact) repetitions. It exists in two variants: *GenCompress-1* uses the Hamming distance (only substitutions) for the repeats while *GenCompress-2* uses the edition distance (deletion, insertion and substitution) for the encoding of the repeats.

CTW+LZ [11] is another algorithm, based on the context tree weighting method. It combines a LZ-77 type method like *GenCompress* and the *CTW* algorithm. Long exact/appoximate repeats are encoded by LZ77-type algorithm, while short repeats are encoded by *CTW*. Although they obtain good compression ratios, its execution time is too high to be used for long sequences.

DNACompress [5] is a DNA compression tool, which employs the Ziv-Lampel compression scheme as *Biocompress-2* and *GenCompress*. It consists of two phases: during the first phase, it finds all approximate repeats including complementary palindromes, using a specific software, *PatternHunter* [12]. Then the approximate repeats and the non-repeat regions are encoded. In practice, the execution time of *DNACompress* is much less than *GenCompress*.

DNAC [7] is another DNA compression tool, working in four phases: during the first phase, it builds a suffix tree to locate exact repeats. During the second phase, all the exact repeats are extended into approximate repeats by dynamic programming. In the third phase, it extracts the optimal non-overlapping repeats from the overlapping ones, and in the last phase, it encodes all the repeats.

DNASequitur is a grammar-based compression algorithm for DNA sequences which infers a context-free grammar to represent the input data. Even if the algorithm is elegant, the practical results show that other methods achieve better compression ratios.

In this paper, we propose a new algorithm, *DNAPack*, which uses Hamming distance for the repeats and complementary palindromes, and either CTW or Arith-2 compression for the non-repeat regions. Unlike the above algorithms, *DNAPack* does not choose the repeats by a greedy algorithm, but uses a dynamic programming approach instead.

The content of the paper is organized as follows: in section 2, we describe the different techniques and concepts used in our algorithm. In section 3, we present the algorithm we use to find a semi-optimal decomposition of the file, and the encoding of the compressed file is presented in section 4. Finally, in section 5, we compare our results on a standard set of DNA sequences with results published for the other algorithms.

2 Useful Techniques

Our algorithm like all of the other DNA-oriented algorithms is based on partitioning the sequence into two kinds of segments: the *repeat* segments (or copied) and the *non-repeat* segments. The repeat segments are either *approximate direct*

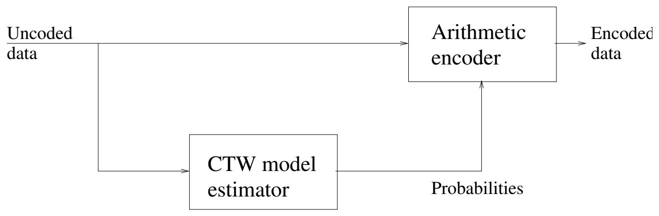

Fig. 1. The schema of a CTW encoder

repeats (repeats with some substitutions) or the *approximate complementary palindrome* repeats (reversed repeats where bases are replaced by their complementary base). In order to achieve the best results in compression, several techniques are used for encoding the different kinds of segments. In this section, we describe a set of techniques and methods which we use in our algorithm.

2.1 Encoding of Non-Repeat Regions

For the encoding of the non-repeat regions, we need to compress arbitrary sequences of bases without long repeats. Two different techniques have been shown to be efficient on these regions:

Order-2 Arithmetic Coding. In comparison to the Huffman Coding algorithm, Arithmetic Coding overcomes the constraint that the symbol to be encoded has to be coded by a round number of bits. This leads to higher efficiency and a better compression ratio in general. The adaptative arithmetic coding of order 2 has the best compression ratio on the DNA sequences; in this arithmetic encoding the adaptative probability of a symbol is computed from the context after which it appears. The best ratio for DNA is obtained for order-2 (contexts are the last two symbols), which seems to correspond to the amino-acid codons, i. e. groups of three bases coding an amino-acid in a protein.

Context Tree Weighting Coding. The probabilities in an arithmetic coding can be computed in different ways. Willems et al. in [14] proposed a universal compression algorithm denoted by Context Tree Weighting (CTW) method which has on average a good compression ratio for an unknown model. The CTW encoder consists of two parts (see Figure 1): a source modeler which is the actual CTW algorithm, which receives the uncompressed data and estimates the probability of the next symbol and an arithmetic encoder which uses the estimated probabilities to compress the data.

One important concept in the the CTW algorithm is the context tree which is built dynamically during encoding/decoding process. All of the already visited substrings of shorter size than a fixed bound (the height of the tree), exist as a path in the tree. Each node of the tree contains a probability. In order to encode a given bit, the following steps are performed: the path in the context tree which coincides with the current context is searched and if needed extended. For every node in this context path, an estimated probability of the next symbol is computed using the data stored in the node. Then a weighted probability is computed using a weighting function on all the estimated probability values. The idea here is that if good coding distributions for two different texts are weighted then the weighted distribution is a good coding distribution for both sources. Finally the weighted probability is sent to the arithmetic encoder which encodes the symbol, and the encoder goes to the next symbol.

2.2 Encoding of Numbers

Different integer numbers have to be encoded in our algorithm. For example, the segments have not a fixed length, so this length has to be encoded. For any repeat segment, the position of the reference substring of the input, from which we copy this segment, should be encoded. When the copies are approximate (and not exact) the positions of the modifications should be encoded. There is no bound on any of these numbers, so these integers should be encoded in a self-delimited way rather than being encoded in a fix number of bits. For encoding the reference position, encoding the relative difference of position of the reference and the copy itself is preferable.

Fibonacci Encoding. An efficient self-delimited representation of arbitrary numbers is the Fibonacci encoding [1]. The Fibonacci encoding is based on the fact that any positive integer n can be uniquely expressed as the sum of distinct Fibonacci numbers, so that no two consecutive Fibonacci numbers are used in the representation. This means that if we use binary representation of a Fibonacci encoding of a number, there are no two consecutive 1 bits. So by adding a 1 after the 1 corresponding to the biggest Fibonacci number in the sum, the representation becomes self-delimited. The Fibonacci representation of some numbers are given in the Table 1.

Shifted Encoding. Although Fibonacci encoding is a good coder for an unknown set of numbers, one can construct better codes if we have some information about the numbers. For example, if there are many small numbers and not a lot of large numbers to encode, Fibonacci encoding can be improved by using a *shifted* version. We define a *k-shifted Fibonacci encoding* as a coding where all the numbers in the range $[1..2^k - 1]$ into their normal binary representation and codes all the other numbres 0^k followed by the Fibonacci encoding of $n-(2^k-1)$. In Table 1, k-shifted representation of some numbers for $k = 1$ and $k = 3$ are given.

	1	2	3	4	8	18
Fibonacci	11	011	0011	1011	000011	0001011
1-Shifted Fibonacci	1	011	0011	00011	001011	01010011
3-Shifted Fibonacci	001	010	011	100	00011	00001011

Table 1. Fibonacci and shifted fibonacci representation of some numbers; depending on the distribution of the numbers to be encoded one method can be prefered to the others.

2.3 Hamming-based Transcription

Suppose a substring v is an approximate repeat of substring u of the same size. To encode v, we first encode the relative position of its already visited repeat u (the pattern to be copied). Then we need to encode the *edit transcription* which transforms u to v. We use three types of instructions: the first instruction is *Copy*(l) which indicates that the next l symbols of the two substrings are the same. *Replace*(x) indicates that the symbol in the corrent position of u should be replaced by x in order to generate v. The last instruction is the $Finish$ instruction which indicates that v is completely generated. Using this termination instruction is a way to prevent encoding the size of the string v.

One important remark is that we use *Hamming* distance, i.e. approximation is done only by substitutions, whereas some other compressors use the *Edit* distance, where approximation can also be done by deletions and insertions. Although we didn't experiment with it, previously published results have shown that the benefit in compression ratio [3] is not worth the increased complexity and computation time.

3 The Algorithm

3.1 Dynamic Programming vs. Greedy Algorithms

DNACompress, *GenCompress* and *CTWLZ* obtain the best results among the existing algorithms. Both *GenCompress* and *DNACompress* use the greedy approach for selection of the repeat segments. *GenCompress* selects the best prefix of the region which is not yet encoded to be coded at the next step. As shown in figure 2(a), the greedy selection of the segment A, prevents the possibility of selecting the longer segment B. *DNACompress* has a different greedy scheme. In each step, it chooses the most profitable segment which does not intersect with the already chosen segments. Figure 2(b) shows how, by this greedy selection, the algorithm may be prevented from choosing the best set of segments.

CTW+LZ tries to solve the problem of *GenCompress* by using some heuristics which are unfortunately very time-consuming without yielding to real improvements. Moreover, they cannot be applied to long sequences, because of their time consumption.

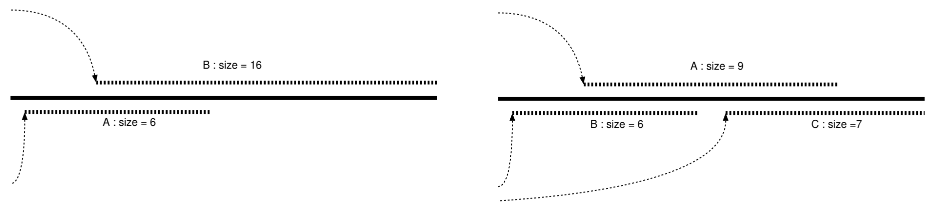

Fig. 2. (a) On the left-hand side, choosing the first segment as gencompress does is not optimal. (b) On the right-hand side, choosing the biggest segment and discarding overlapping segments is also non-optimal.

In our algorithm, *DNAPack*, we use a dynamic programming approach for selection of the segments, therefore solving the problem of greedy selection. We use a set of optimizations which make the running time of our algorithm, reasonably small, so it can be applied to very long sequences.

Let s be the input DNA sequence. Let $BestComp[i]$ be the smallest compressed size of the prefix $s[1..i]$. The following simple recurrence is the general scheme of our dynamic programming.

In this section firstly we explain our method which is based on dynamic programming. Then we comment about the optimizations which makes our algorithm works in a reasonably good time.

Initialization:

$$BestComp[0] = 0$$

Recurrence:

$$\forall i > 0 \qquad BestComp[i] = \min \begin{cases} BestComp[j] + CopyCost(j,i,k) & \forall k \ \forall\, 0 < j < i \\ BestComp[j] + PalinCopyCost(j,i,k) & \forall k \ \forall\, 0 < j < i \\ BestCopy[j] + MinCost(j+1,i) & \forall\, 0 < j < i \end{cases}$$

Fig. 3. Dynamic Programming scheme for finding the best compression

$CopyCost(j,i,k)$ is the number of bits needed to encode the substring of size k starting at position i if it is an approximate repeat of the substring of size k starting at j. The $PalinCost$ is similarly defined for reverse complementary substrings. The function $MinCost(j+1,i)$ is the number of bits needed for compression of the segment $s[j+1,i]$. It depends on the size of the substring (for the size of the Fibonacci encoding) as well as the compression ratio obtained for the algorithm by arithmetic coding or CTW. $MinCost$ allows us to create a repeat segment only if it would yield a benefit in the compression ratio. These three functions are estimations of the real cost, since the efficiency of some optimizations done during the encoding cannot be computed at this point.

3.2 Reducing Execution Time

A direct implementation of this algorithm has a complexity of $O(n^3)$, which is much too high for long DNA sequences. Therefore, we use several techniques to reduce the execution time in pratice. First, we authorize only the repeats which have a common *seed*. The seed is a small string of size l (a parameter of our program), whose already found positions are stored in a hash table. To find a repeat, we need only to find the positions of its seed in the hash table, and try to increase their sizes.

In the third line of the recurrence, we do not really need to examine all the j's. A careful observation shows that we only need to compute $MinCost(j+1, i)$ for a j which is the end of a repeat segment, since there is no gain in creating two consecutive non-repeat segments: each non-repeat segment contains its size in Fibonacci encoding, and the size of two small encoded numbers is greater than the size of one big encoded number. In fact, we can even narrow this search among all of the j's which $BestComp[j+1]$ is not optimized by copying a segment from the same position as $BestComp[j]$.

In the case of repeats, if $s[i-1] = s[j-1]$ then there is no need to check the different values of k One can verify that $BestComp[j-1] + CopyCost(j-1, i, k+1) \leq BestComp[j] + CopyCost(j, i, k)$.

Similar observation can be made for the case of the reverse repeat (2^{nd} line of the recurrence). As a result of these optimizations, the number of possible j and k to search in our dynamic programming will be reduced enormously such that it is possible to execute the algorithm on large sequences in a few seconds (in our experimentations for example, $MinCost(j+1, i)$ is only computed for 2 or 3 different j for every i).

4 Practical Encoding

As written in the introduction, the encoding of bases on two bits (A with 00, T with 01, C with 10 and G with 11 for example) already gives a good compression ratio, which can be hard to beat if we don't pay enough intention to the encoding scheme used for the approximate repeats. Indeed, previous compression algorithms for DNA in the litterature mainly focus on the algorithm used to find the repeats, unfortunately forgetting to discuss the various ways of encoding them, and thus, leading the not so good compressors. As a consequence, we discuss here the choice we did in our implementation to efficiently encode the various compression operations.

The encoding function takes as input a sequence of segments, where each segment is either a sequence of bases, or a repeat of a preceding sequence of bases. It outputs a file containing the same data in a compact representation.

4.1 The structure of the compressed file

The compressed file consists of three different regions: HEADER, CODE and BASES.

The HEADER contains all the information that must be known to decode the CODE and BASES regions. For example, it contains:

– The type of compression used for sequences of bases (it is either Arithmetic-2 Coding, CTW or None), on 2 bits.
– The number of segments in the CODE part.
– The minimum size of the first Copy operation in the repeats.
– The most frequent base substited for another base in a repeat substitution.

The CODE region consists of two different types of segments: repeats and non-repeats. For the non-repeats segments, the CODE region only contains the length of the segment, encoded in Fibonacci encoding, whereas all the bases are put in the BASES region. When all the non-repeats have been processed, the complete BASES region is compressed using either Arithmetic-2 Coding, Context-Tree Weighted Coding or 2-bits Coding, whichever gives the best compression ratio.

Since $|FibEncode(a + b)| \leq |FibEncode(a)| + |FibEncode(b)|$, the CODE region never contains two consecutive non-repeats segments, as they would more efficiently be encoded as a single segment. Consequently, we use the following encoding to describe the type of the segments:

– - an empty code for the first segment of the gene, which is always a non-repeat.
– 0 for a non-repeat segment after a repeat segment.
– 1 for a repeat segment after a repeat segment.
– - an empty code for a repeat segment after a non-repeat segment.

4.2 The encoding of repeats

A repeat segment must contain the following information:

– The type of repeat: direct repeat or complement-palindrome repeat. We only need one bit for this information.
– The offset to the origin of the repeat (increased by one for complement-palindrome repeats to avoid zeroes). We simply encode this offset in Fibonacci encoding.
– The sequence of operations to repeat the segment, containing either copies or mutations.

In this first implementation, we force the sequence of operations to always finish with a Copy. Thanks to this simplification, a single bit can be used per operation to distinguish between copies and Replaces:

– - an empty code for the first operation, which is always a Copy, since our algorithm to find repeats looks for repeats with at least l characters in common.
– 0 for the end of the repeat after a Copy.
– 1 for a Replace after a Copy.
– 0 for a Copy after a Replace.
– 1 for a Replace after a Replace.

4.3 The encoding of operations arguments

A Copy operation requires only one argument, the number of bases to be copied from the original segment. Although we cannot optimize the representation of this length in the general case, our algorithm guarantees that at least the first l bases (called seed) will be similar between the two substrings. Thus, we compute the minimal length of the first Copy of each repeat-segment, and we always encode the first Copy of a segment after removing this mimimum. With hundreds of repeats in each gene, 4 or 5 bits saved per repeat, this simple optimization finally saves a few thousands bits.

For each Replace operation, we need to supply the base to replace the former base of the original segment. However, since we know that former base, the new base can only be one of the 3 other bases. Therefore, we can represent it more efficiently:

- 0 for the most probable base.
- 10 and 11 for the two other bases.

Instead of using probabilities, our first implementation simply computes the most frequent substitution for every former base, and store it in the HEADER region of the file.

5 Experimental Results

sequence	length	BioCompress-2	GenCompress	CTW-LZ	DNACompress	DNAPack
CHMPXX	121024	1.6848	1.6730	1.6690	1.6716	**1.6602**
CHNTXX	155844	1.6172	1.6146	1.6120	1.6127	**1.6103**
HEHCMVCG	229354	1.8480	1.8470	1.8414	1.8492	**1.8346**
HUMDYSTROP	33770	1.9262	1.9231	1.9175	1.9116	**1.9088**
HUMGHCSA	66495	1.3074	1.0969	1.0972	**1.0272**	1.039
HUMHBB	73308	1.8800	1.8204	1.8082	1.7897	**1.7771**
HUMHDABCD	58864	1.8770	1.8192	1.8218	1.7951	**1.7394**
HUMHPRTB	56737	1.9066	1.8466	1.8433	1.8165	**1.7886**
MPOMTCG	186609	1.9378	1.9058	1.9000	**1.8920**	1.8932
PANMTPACGA	100314	1.8752	1.8624	1.8555	1.8556	**1.8535**
VACCG	191737	1.7614	1.7614	1.7616	**1.7580**	1.7583
Average	—	1.7837	1.7428	1.7389	1.7254	**1.7148**

Table 2. Comparison of compression ratios for different algorithms (bits/base)

To experiment our algorithm, we tried to compress a standard set of DNA sequences with our algorithm, and we compare with results published for other efficient DNA compressors. The results are displayed on table 2. The table shows that our program, DNAPack, performs slightly better than other programs, except DNACompress in a few cases.

During our experiments, we tried to compress all the sequences while varying a few parameters used by our algorithm:

– l, the size of the exact prefix that two repeats must have in common to be compared.
– compressions, the set of compressions algorithms used to compress BASES, ranging in $\{Arith-2, CTW, 2-Bits\}$.
– approx, the estimation of the compression ratio that will probably be obtained on the BASES region, used during the dynamic programming phase.

Due to space constraints, we cannot display all these results, but we can briefly summarize them: a big l gives very short execution time, without decreasing too much the compression ratio (in the previous table, the three first genes are compressed with l = 30, and all results are obtained in less than 15 seconds, except for MPOMTCG and VACCG which were produced in around 1 minutes). The two different compressions of BASES are also important: among the 11 genes compressed, 7 were compressed using CTW, 4 using Arith-2. For some genes, 2-Bits was more efficient than Arith-2. Finally, having the most accurate value of approx also impacts a lot on performances, especially for small values of l, where a lot of choices between small repeats and non-repeats must be done.

6 Conclusion

We have presented a new algorithm to compress DNA sequences. As most other DNA compressors, our algorithm works by finding approximate repeats and trying to optimally encode them. The first version of our implementation has results which on average are slightly better than former algorithms. It is mainly due to the benefits of dynamic programming, and the careful choice of the encoding of the repeats. We are now working on different aspects of our compressor: first, we are tuning the different parameters to improve the compression ratio while decreasing the computation time; we are experimenting it on larger genes (chromosomes contain at least twenty million bases on average), and finally, we are trying to find a better compression method for some sequences, where approximate repeats are unfrequent (HUMDYSTROP for example), who failed to be compress efficiently by all the known compressors.

References

1. Apostolico A. and Fraenkel A.S.: Robust transmission of unbounded strings using Fibonacci representations. IEEE trans. inform., 33(2), pp 238-245, 1987.
2. Apostolico A. and Lonardi S.: Compression of Biological Sequences by Greedy Off-line Textual Substitution. In proc. Data Compression Conference, IEEE Computer Society Press, pp 143-152, 2000.

3. Chen, X., Kwong, S., Li, M.: A compression Algorithm for DNA sequences and its applications in genome comparison. The 10th workshop on Genome Informtics (GIW'99), pp 51-61, Tokyo, Japan, 1999.
4. Chen, X., Kwong, S., Li, M.: A compression Algorithm for DNA sequences. IEEE Engineering in Medicine and Biolgoy Magazine, 20(4), 61-66, Jul/Aug 2001.
5. Chen, X., Li, M., Ma, B. and Tromp, J.: DNACompress: fast and effective DNA sequence compression. Bioinformatics, 18:1696-1698, 2002.
6. Cherniavski N. and Lander R.: Grammar-based Compression of DNA sequences. 2004
7. Chang C.-H.: DNAC: A Compression Algorithm for DNA Sequences by Non-overlapping Approximate Repeats. Master Thesis, 2004.
8. Grumbach S. and Tahi F.: Compression of DNA Sequences. In Data compression conference, pp 340-350. IEEE Computer Society Press, 1993.
9. Grumbach S. and Tahi F.: A new Challenge for compression algorithms: genetic sequences. Journal of Information Processing and Management, 30, 875-866, 1994.
10. Li M., Badger J. H., Chen X., Kwong S., Kearney P., Zhang H.: An information based sequences distance and its application to whole motochondrial genome phylogeny," Bioinformatics, 17(2): 149-154, 2001.
11. Matsumuto T., Sadakane K.,Imai H.: Biological sequence compression algorithms. Genome Inform. Ser. Wokrshop Genome Inform. 11:43-52, 2000
12. Ma B., Tromp J. and Li M.: PatternHunter–faster and more sensitive homology search. Bioinformatics, 18, 440-445, 2002.
13. Rivals E., Delahaye J.-P., Dauchet M., Delgrange O.: A Guaranteed Compression Scheme for Repetitive DNA Sequences. Data Compression Conference, 1996.
14. Willems F. M. J., Shtrakov Y. M. and Tjalkens T. J.: The Context Tree Weighting Method: Basic Properties. IEEE Trans. Inform. Theory, IT-41(3), pp 653-664, 1995.