# Compression of Nucleotide Databases for Fast Searching

Hugh Williams* Justin Zobel
Department of Computer Science, RMIT
GPO Box 2476V, Melbourne 3001, Australia
{hugh,jz}@cs.rmit.edu.au

**Abstract**

**Motivation:** International sequencing efforts are creating huge nucleotide databases, which are used in searching applications to locate sequences homologous to a query sequence. In such applications, it is desirable that databases are stored compactly; that sequences can be accessed independently of the order in which they were stored; and that data can be rapidly retrieved from secondary storage, since disk costs are often the bottleneck in searching.
**Results:** We present a purpose-built *direct coding* scheme for fast retrieval and compression of genomic nucleotide data. The scheme is lossless, readily integrated with sequence search tools, and does not require a model. Direct coding gives good compression and allows faster retrieval than with either uncompressed data or data compressed by other methods, thus yielding significant improvements in search times for high-speed homology search tools.
**Availability:** The direct coding scheme (`cino`) is available free of charge by anonymous ftp from `goanna.cs.rmit.edu.au` in the directory `pub/rmit/cino`.
**Contact:** hugh@cs.rmit.edu.au.

**Keywords** Compression, nucleotide data, Huffman coding, integer coding, nucleotide search tools.

## 1 Introduction

Sequencing initiatives are contributing exponentially increasing quantities of nucleotide data to databases such as GenBank (Benson et al., 1993). We propose a new *direct coding* compression scheme for use in homology search applications such as FASTA (Pearson and Lipman, 1988), BLAST (Altschul et al., 1990), and CAFE (Williams and Zobel, 1996a). This scheme yields compact storage, is lossless—nucleotide bases and wildcards are represented—and has extremely fast decompression.

Prior to proposing our scheme we investigate benchmarks for practical compression and high-speed decompression of nucleotide data. We compare our scheme with the entropy, with Huffmann coding, with the utilities *gzip* and *compress*, and with uncompressed data retrieval. All the compression methods closely approach the entropy but direct coding is over 9 times faster than Huffmann coding and requires much less memory; direct coding is also several times faster than the standard compression utilities. Direct coding requires around 25% of the space required to store uncompressed data and, due to savings in disk costs, has significantly lower retrieval times.

## 2 Database compression

Compression consists of two activities, modelling and coding (Rissanen and Langdon, 1981). A model for data to be compressed is a representation of the distinct symbols in the data and includes information such as frequency about each symbol. Coding is the process of producing a compressed representation of data, using the model to determine a code for each symbol. An efficient coding scheme assigns short codes to common symbols and long codes to rare symbols, optimising code length overall.

Adaptive models (which evolve during coding) are currently favoured for general-purpose compression (Bell et al., 1990; Lelewer and Hirschberg, 1987), and are the basis of utilities such as *compress*. However, because databases are divided into records that must be independently decompressible (Zobel and Moffat, 1995), adaptive techniques are generally not effective. Similarly, arithmetic coding is in general the preferred coding technique; but it is slow for database applications (Bell et al., 1993).

For text, Huffman coding with a semi-static model (where modelling and coding are in separate phases) is preferable because it is faster and allows order-independent decompression. Such compression

*To whom correspondence should be addressed. A preliminary version of this work was presented in "Practical Compression of Nucleotide Databases", Proc. Australian Computer Science Conf., Melbourne, Australia, 1996, pp. 184–192.

| Base | A | B | C | D | G | H | K | M |
|---|---|---|---|---|---|---|---|---|
| Probability | 27.483 | ≈ 0 | 22.270 | ≈ 0 | 22.985 | ≈ 0 | 0.002 | 0.003 |
| Base | N | R | S | T | V | W | Y | |
| Probability | 0.737 | 0.002 | 0.003 | 26.508 | ≈ 0 | 0.001 | 0.004 | |

Table 1: Probabilities of each base in GENBANK (percent).

schemes can allow retrieval of data to be faster than with uncompressed data since the computational cost of decompressing data can be offset by reductions in transfer costs from disk.

The compression efficiency of a technique can, for a given data set, be measured by comparison to the information content of data, as represented by the *entropy* determined by Shannon's coding theorem (Shannon, 1951). Entropy is the compression that would be achievable with an ideal coding method using a simple semi-static model. For a set $S$ of symbols in which each symbol $t$ has probability of occurrence $p_t$, the entropy is

$$E(S) = \sum_{t \in S} (-p_t \cdot \log_2 p_t)$$

bits per symbol.

Implicit in this definition is the representation of the data as a set of symbol occurrences, that is, modelling of the data using simple tokens. In some domains, different choices of tokens give vastly varying entropy; for example, in English text compression, choosing characters as tokens gives an entropy of around 5 bits per character, whereas choosing words as tokens gives an entropy of around 2 bits per character (Bell et al., 1990). The cost of having words as tokens is that more distinct tokens must be stored in the model, but for sufficiently large data sets the net size is still much less than with a model based on characters.

## 3 Entropy of nucleotide data

We now consider the entropy of nucleotide data. We first describe our test data.

In this paper we measure the volume of DNA in megabases, that is, units of $2^{20}$ bases. In our nucleotide compression experiments we have extracted sequences from GenBank[1] to give two test collections: VERTE, a collection of 121,624 rodent, mammal, primate, vertebrate and invertebrate sequences containing 168.88 megabases; and GENBANK, the full database of 1,021,211 sequences containing 621.77 megabases. All the experiments in this paper were carried out on a Sun SPARC 20, with the machine otherwise largely idle.

[1]Flat-file Release 97.0, October 1996

A possible choice of symbol for nucleotide data is the distinct non-overlapping *intervals* in the data, where an interval is a string of bases for a fixed length $n$. While this token model may only capture simple patterns and not any semantics of genomic nucleotide data, this simple model is practical for comparison to high-speed compression schemes where complex structure determination is prohibitively computationally expensive.

For sequences divided into intervals, the entropy is

$$E_n^{int}(S) = \frac{1}{n} \sum_{t \in S} (-p_t \cdot \log_2 p_t)$$

bits per base, where $p_t$ is the probability of the occurrence of interval $t$. Note that one would expect a low entropy for short samples and long intervals—it is not a sign of pattern. Long intervals also imply a large model, since the number of distinct symbols to be stored will approach $4^n$ (or exceed it if there are occurrences of wildcards).

Now we consider the entropy of our test collections. Results are shown in Table 2, giving the entropy $E_n^{int}$ and the number of distinct intervals for each collection and interval length. The entropy is almost exactly as expected for random data. (We further discuss estimation of entropy for this data elsewhere (Williams and Zobel, 1996b).) As another estimate of compressibility, we tested PPM predictive compression (Bell et al., 1990), currently the most effective general-purpose lossless compression technique, and found that even with a large model PPM was only able to compress to 2.06 bits per base on the GENBANK collection. (Note that PPM is adaptive—and rather slow—and hence unsuited to nucleotide data.) We therefore conclude that, as is commonly believed for genomic nucleotide sequences, there is little discernible pattern when compressing using simple token-based models and that compression to approximately 2 bits per base is a good result.

Other approaches to modelling can, however, yield better compression. Techniques that use more complex secondary structure to achieve additional compression, such as the palindromic repeats in DNA, are discussed in Section 6.

| Int. length | VERTE $E_n^{int}$ | VERTE Intervals | GENBANK $E_n^{int}$ | GENBANK Intervals |
|---|---|---|---|---|
| 1 | 1.98 | 15 | 2.04 | 15 |
| 5 | 1.97 | 7,487 | 2.02 | 25,981 |
| 8 | 1.96 | 123,036 | 2.00 | 462,422 |
| 10 | 1.94 | 1,117,579 | 1.98 | 2,928,638 |

Table 2: Properties of GenBank, with sequences divided into intervals (entropy in bits per base, distinct intervals in model).

| Property | $n$ | VERTE | GENBANK |
|---|---|---|---|
| Compression rate (Mb/sec) | 1 | 0.08 | 0.08 |
| | 5 | 0.11 | 0.11 |
| | 8 | 0.07 | 0.04 |
| Decompression rate (Mb/sec) | 1 | 0.53 | 0.52 |
| | 5 | 1.05 | 1.03 |
| | 8 | 1.00 | 0.99 |
| Compression (bits/base) | 1 | 2.22 | 2.24 |
| | 5 | 1.99 | 2.04 |
| | 8 | 1.97 | 2.03 |

Table 3: Performance of Huffman coding.

## 4 Huffman coding

Huffman coding is a well-known technique for making an optimal assignment of variable-length codes to a set of symbols of known probabilities (Witten et al., 1994). Although not the best general-purpose coding method, Huffman coding is preferred for text databases in which records need not necessarily be decompressed in the order they were stored (Zobel and Moffat, 1995). We have experimentally applied an array-based efficient implementation of Huffman coding, known as canonical Huffmann coding, to our test collections.[2] As symbols we used non-overlapping intervals of fixed-length $n$ for several choices of $n$. As sequence length is not always an exact multiple of $n$ bases, the model includes, not just strings of length $n$, but also shorter strings from the ends of sequences.

Results of the Huffmann coding scheme for a range of interval lengths are shown in Table 3, with the compression rates including model size. A length of 1 was included to show that direct coding of individual bases is not very efficient; predictably, the scheme of allocating a fixed code to each base and wildcard did not work well. We have also experimented with larger values such as $n = 10$, but performance was poor, presumably due to constraints of hardware and the large model size.

[2]The implementation of canonical Huffman coding used is incorporated into the MG text database system and is due to Moffat (Bell et al., 1995; Witten et al., 1994).

Overall, $n = 5$ has worked best: the model is fairly small and, on our hardware, tends to remain resident in the CPU cache, so that accesses to intervals to be decoded is as fast as possible. The actual decoding process is slightly more efficient for $n = 8$, but decompression is slower overall again because of hardware cache constraints and a large model size.

## 5 Direct coding

We have seen that the frequency of wildcards in our test collections is extremely low; over 99% of all characters are one of the four nucleotides and over 97.8% of the wildcard occurrences are N. Because the data is highly skewed, we investigate a lossless compression scheme where the four nucleotide bases are encoded using two-bit representations and wildcards are stored compactly in a separate structure.

In the encoded sequence, we eliminate each wildcard occurrence by replacing it with a random nucleotide chosen from those represented by the wildcard. First, during decoding it is less computationally efficient to insert the wildcards into the sequence than to recreate the original string by replacing the randomly-chosen nucleotides by the original wildcards. Second, as wildcards are often not needed or used in searching of genomic databases, the random substitution of a base is more appropriate than deleting the wildcard to make a compression saving, as a deletion completely removes any semantic meaning from a sequence. This is an acceptable solution for some practical applications—and indeed it is an option in GenBank search software such as BLAST (Altschul et al., 1990). Having replaced all occurrences of wildcards, we code the sequence using two bits for each nucleotide base.

Sequence length varies from around 10 bases to over 400,000, with an average of around 650 bases. Therefore, use of a fixed-length integer representation of sequence length will be space-inefficient. We chose to use a variable-byte representation in which seven bits in each byte is used to code an integer, with the least significant bit set to 0 if this is the last byte, or to 1 if further bytes follow. In this way, we represent small integers compactly; for example, we represent 135 in two bytes, since it lies in the range $[2^7 \cdots 2^{14})$, as 00000011 00001110; this is read as 00000010000111 by removing the least significant bit from each byte and concatenating the remaining 14 bits.

We then store wildcard data independently, in a separate structure. First, we store in unary the count of different wildcards that occur in the sequence, where a unary integer $n$ is a string of $(n-1)$ 0-bits terminated with a single 1-bit—in most sequences with wildcards, this is a single bit representing the occur-

rence of N. Second, for each different wildcard we store a Huffmann-coded representation of the wildcard (ranging from a single bit for N to 6 bits for the most uncommon wildcards), followed by a count of the number of occurrences, then a series of integer positions or *offsets* within the sequence.

Using this encoding scheme, there are at most eleven tuples of the form

$$(w, count_w : [pos_1, \ldots, pos_p]) ,$$

where $w$ is the Huffmann-coded representation of a wildcard, $count_w$ is the number of occurrences and $pos_1, \ldots, pos_p$ are the offsets at which $w$ occurs.

As offsets may be of the order of $10^6$ and counts of occurrences typically small, we must be careful to ensure that storing wildcard information does not waste space; variable-byte codes, for example, would be highly inefficient. The solution is to use variable-bit integer codings such as the Elias codes (Elias, 1975) and the Golomb codes (Golomb, 1966). We have used the Elias gamma codes to encode each $count_w$ and Golomb codes to represent each sequence of offsets. These techniques are a variation on techniques used for inverted file compression, which has been successfully applied to large text databases (Bell et al., 1993) and to genomic databases (Williams and Zobel, 1996a; Williams and Zobel, 1996b).

Compression with Golomb codes, given the appropriate choice of a pre-calculated parameter, is better than with Elias coding. In particular, using Golomb codes the maximum space required to store a list of positions for a given wildcard arises when that wildcard occupies every position; in this worst case the storage requirement is 1 bit per position.

Instead of storing absolute offsets we store the differences between the offsets, which with Golomb codes can be represented in fewer bits. Thus each tuple is stored in the form

$$\begin{aligned}&(w, number_w + 1 : \\ &[pos_1, \\ &(pos_2 - pos_1), \ldots, (pos_p - pos_{p-1})]) .\end{aligned}$$

To illustrate wildcard storage, consider an example where the wildcard N occurs three times in a sequence, at offsets 253, 496 and 497, and the wildcard B occurs once, at offset 931. The other nine wildcards do not occur. Illustrating our example with the data as integers, the wildcard structure would be

$$[2 : (\mathtt{n}, 3 : [253, 496, 497]), (\mathtt{b}, 1 : [931])] .$$

After taking differences, we have

$$[2 : (\mathtt{n}, 3 : [253, 243, 1]), (\mathtt{b}, 1 : [931])] .$$

To simplify sequence processing when wildcard information is not to be decoded, we store the length of the compressed wildcard data, again using the variable-byte coding scheme. A benefit of this scheme is that, for sequences with no wildcards, a length of zero is stored without any accompanying data structure—an overhead of a single byte.

| Property | VERTE | GENBANK |
|---|---|---|
| *With wildcards* | | |
| Compression (Mb/sec) | 0.36 | 0.51 |
| Decompression (Mb/sec) | 13.67 | 10.81 |
| Compression (bits/base) | 2.02 | 2.09 |
| *Without decoding of wildcards* | | |
| Decompression (Mb/sec) | 14.07 | 13.44 |
| *Without wildcards* | | |
| Compression (Mb/sec) | 0.36 | 0.54 |
| Decompression (Mb/sec) | 14.75 | 14.27 |
| Compression (bits/base) | 2.01 | 2.03 |
| *Retrieval of direct-coded data* | | |
| Sequential (Mb/sec) | 13.67 | 10.81 |
| Random 10% (Mb/sec) | 1.43 | 2.96 |
| *Retrieval of uncompressed data* | | |
| Sequential (Mb/sec) | 4.12 | 2.97 |
| Random 10% (Mb/sec) | 0.38 | 0.59 |

Table 4: Performance of direct coding.

With this representation of sequences, decoding has two phases. In the first phase the bytes representing the sequence, each byte of four 2-bit values, are mapped to four nucleotides through an array. This process is extremely fast; it is an insignificant fraction of disk fetch costs, for example. In the second phase the tuples of wildcard information are decoded, and wildcard characters are overwritten on nucleotides at the indicated offsets.

The first block of Table 4 shows results for this direct coding scheme. For VERTE, compression is around 0.05 bits per base higher than the entropy and slightly higher in the GENBANK collection, because the proportion of sequences containing wildcards increases from around 16% in the VERTE collection to 58% in GENBANK; this also results in a reduction in decompression speed from around 14 Mb/sec for VERTE to around 11 Mb/sec for GENBANK.

Overall, decompression speed is excellent, between 10 and 14 times faster than that given by Huffman coding. We have also shown, in the second block of Table 4, decompression rates without decoding of wildcards—as discussed above, some search tools are used without them—and as can be seen the impact of wildcards on time is small.

The third block of Table 4 shows compression performance with wildcards replaced by random matching nucleotides. This achieves compression of around 2.02 bits per base, as shown. The compressed data

occupies slightly more than 2 bits per base because for each sequence we must store the sequence length and, since we store sequences byte-aligned, the last byte in the compressed sequence is on average only half full. Note that in GENBANK the wildcards contribute disproportionately to decompression costs: they are 0.6% of the compressed data but account for around 25% of the decompression time.

The last two blocks of table 4 compare retrieval times for uncompressed data to those for direct-coded data. The first line in each block is the speed of sequential retrieval of all sequences: by using direct coding, the reduction in disk costs results in a four-fold improvement in overall retrieval time. The second line in each block illustrates the further available improvement when retrieving only a fraction of the sequences: in this case, we retrieved a random 10% of the sequences and averaged the results over 10 such runs. In the case of random access, retrieval of direct-coded data is again over four times faster than with uncompressed data. We therefore expect that use of direct coding in a retrieval system would significantly reduce retrieval times overall.

To further test this hypothesis we incorporated the scheme into CAFE, our genomic database retrieval engine (Williams and Zobel, 1996a), and found that retrieval times fell by over 20%.

In BLAST (Altschul et al., 1990) a simple approach is taken to nucleotide compression. All occurrences of wildcards are replaced by a random choice of any of the four nucleotides. In addition to a count indicating sequence length, there is an indication of whether the sequence originally contained wildcards. BLAST achieves compression of 2.03 bits per base on the GENBANK collection using this scheme; this is a saving of 0.06 bits per base over our direct-coding scheme, but is lossy because wildcard data is discarded. To allow processing of sequences with wildcards, each sequence is also stored uncompressed, giving a total storage requirement of 10.03 bits per base.

With BLAST, a user preference during retrieval is optional wildcard matching by retrieving the original uncompressed data file for sequences with wildcards. As our results show, fetching this data will have a serious impact on query evaluation time because retrieval of uncompressed data is extremely slow.

Tools like BLAST inspect all the sequences in a database in response to a query, either decompressing them or processing them directly in compressed form. We have investigated alternatives based on indexing (Williams and Zobel, 1996a), but even with indexing a significant fraction of the database must be inspected during query evaluation. Fast decompression, or a format that can be processed directly, is thus crucial to efficient query processing.

Table 5 shows the results of using the compression tools *gzip* and *compress* on the VERTE and GENBANK collections. Both are relatively slow in compression and decompression and require more bits per character than the direct coding scheme. Note that both methods are unsuitable for database compression, as both allow only sequential access to sequences.

| Scheme | VERTE | GENBANK |
|---|---|---|
| *gzip* | | |
| Compression (Mb/sec) | 0.23 | 0.41 |
| Decompression (Mb/sec) | 4.12 | 3.84 |
| Compression (bits/base) | 2.07 | 2.14 |
| *compress* | | |
| Compression (Mb/sec) | 0.97 | 1.17 |
| Decompression (Mb/sec) | 2.30 | 2.11 |
| Compression (bits/base) | 2.13 | 2.19 |

Table 5: Performance of standard compression utilities.

## 6 Structure-based coding

A special-purpose compression algorithm for nucleotide data could take advantage of any secondary structure known to be present (Griffiths et al., 1993). For example, Grumbach and Tahi have used the palindromes that are known to commonly occur in DNA strings (without wildcards) to compress to less than 2 bits per base, typically saving 0.2 bits per base and in some cases rather more (Grumbach and Tahi, 1993). The difficulty with such approaches is the cost of recognising the structure: identification of palindromes is an expensive operation, and is complicated by the presence of wildcards. However, palindrome compression would be easy to integrate with our direct coding scheme, as the structure of wildcard information would not be affected.

Another possibility is vertical compression (Grumbach and Tahi, 1993): since sequences in GenBank are grouped, to some extent, by similarity, adjacent sequences may differ in only a few bases; and more frequently may share long common substrings. This similarity could be exploited by a compression technique, and again could easily be integrated with the direct coding, but would violate our principle that records be independently decodable.

## 7 Conclusions

We have considered the problem of practical compression of databases of nucleotide sequences with wildcards, and have identified two lossless compression schemes that work well in practice. Our experimental evaluation of canonical Huffmann coding with a semi-

static model of fixed-length intervals showed that it gives excellent compression, but with the overhead of a large in-memory model and, at decompression rates of around 1 Mb per second, is somewhat slow.

Our compression method, a direct coding designed specifically for nucleotide sequences with wildcard characters, performs rather better. While the compression performance is slightly worse—by around 0.03 bits per base—than for Huffman coding, memory requirements are slight and sequences can be decompressed at up to 14 Mb per second. Such speed is vital to good searching performance, since current searching tools for nucleotide databases inspect a substantial fraction of the database in response to every query. We have shown that compression not only reduces space requirements, but that direct coding results in a fourfold improvement in retrieval time compared with fetching of uncompressed data.

**Acknowledgements**

We are grateful to Alistair Moffat for his implementation of canonical Huffman coding. This work was supported by the Australian Research Council, the Centre for Intelligent Decision Systems, and the Multimedia Database Systems group at RMIT.

## References

Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410.

Bell, T., Cleary, J., and Witten, I. (1990). *Text Compression*. Prentice-Hall, Englewood Cliffs, New Jersey.

Bell, T., Moffat, A., Nevill-Manning, C., Witten, I., and Zobel, J. (1993). Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44(9):508–531.

Bell, T., Moffat, A., Witten, I., and Zobel, J. (1995). The MG retrieval system: Compressing for space and speed. *Communications of the ACM*, 38(4):41–42.

Benson, D., Lipman, D., and Ostell, J. (1993). GenBank. *Nucleic Acids Research*, 21(13):2963–2965.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203.

Golomb, S. (1966). Run-length encodings. *IEEE Transactions on Information Theory*, IT–12(3):399–401.

Griffiths, A., Miller, J., Suzuki, D., Lewontin, R., and Gelbart, W. (1993). *An Introduction to Genetic Analysis*. Freeman, New York, fifth edition.

Grumbach, S. and Tahi, F. (1993). Compression of dna sequences. In Storer, J. and Cohn, M., editors, *Proc. IEEE Data Compression Conference*, pages 340–350, Snowbird, Utah.

Lelewer, D. and Hirschberg, D. (1987). Data compression. *Computing Surveys*, 19(3):261–296.

Pearson, W. and Lipman, D. (1988). Improved tools for biological sequence comparison. *Proc. National Academy of Science*, 85:2444–2448.

Rissanen, J. and Langdon, G. (1981). Universal modeling and coding. *IEEE Transactions on Information Theory*, IT-27(1):12–23.

Shannon, C. (1951). Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:55.

Williams, H. and Zobel, J. (1996a). Indexing nucleotide databases for fast query evaluation. In *Proc. International Conference on Advances in Database Technology (EDBT)*, pages 275–288, Avignon, France. Springer-Verlag. Lecture Notes in Computer Science 1057.

Williams, H. and Zobel, J. (1996b). Practical compression of nucleotide databases. In *Proc. Australasian Computer Science Conference*, pages 184–193, Melbourne, Australia.

Witten, I., Moffat, A., and Bell, T. (1994). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York.

Zobel, J. and Moffat, A. (1995). Adding compression to a full-text retrieval system. *Software—Practice and Experience*, 25(8):891–903.