In Proceedings, IEEE CS Conference on Bioinformatics, Stanford University, CA, August 2002, pp. 303-313.

DNA Sequence Compression using the Burrows-Wheeler Transform

Don Adjeroh[*], Yong Zhang[*], Amar Mukherjee[#], Matt Powell[$] and Tim Bell[$]

Abstract -- **We investigate off-line dictionary oriented approaches to DNA sequence compression, based on the Burrows-Wheeler Transform (BWT). The preponderance of short repeating patterns is an important phenomenon in biological sequences. Here, we propose off-line methods to compress DNA sequences that exploit the different repetition structures inherent in such sequences. Repetition analysis is performed based on the relationship between the BWT and important pattern matching data structures, such as the suffix tree and suffix array. We discuss how the proposed approach can be incorporated in the BWT compression pipeline.**

***Index terms*-- DNA sequence compression, repetition structures, Burrows-Wheeler Transform, BWT**

1. INTRODUCTION

The availability of the draft sequence of the complete human genome, and the complete sequencing of the genome for various other model organisms represent an important milestone in molecular biology. With the complete genome, it becomes possible to perform genome-wide analysis of entire genomes, and cross-genome analysis with complete genomes, which could be important in complete identification of the genes associated with gene-related diseases, and in the discovery of potential drugs to combat them. The human genome contains about 3.1647 billion deoxyribonucleic acid (DNA) base pairs. Other mammalian genomes are also of a similar order of magnitude. Thus, along with the availability of complete genomes, comes the exponential growth in the volume of the available biological sequence data. It is estimated that the number of available nucleotide bases doubles approximately every 14 months[1], while genomes are being sequenced at a rate of 15 complete genomes per month[2].

An important new challenge facing researchers, therefore, is how to make sense out of this huge mass of data. With single gene sequences, we usually consider sequences with tens of thousands of base pairs. With whole genomes, we now have to deal with millions, or billions of base pairs. In addition to the heightened need for efficient and effective algorithms for the analysis, annotation, interpretation and visualization of the data, there is also the need to device effective techniques for the management, organization, and distribution of this unprecedented mass of biological sequence data [Collins98pjcgw]. In this paper, we consider the problem of compressing biological sequences, which is important in addressing the latter problem. We propose off-line dictionary-based methods for compressing DNA sequences. The approach can handle different types of repeats, such as interspersed, tandem, reversed, complimented repeats, and palindromes. In the next section, we present a background to the problem. In section 3, we describe the BWT, which forms the basis for our approach. Section 4 presents two algorithms for compressing DNA sequences, using offline dictionary-based methods. We present test results in section 5, and discuss some issues related to the proposed methods in section 6.

2. BACKGROUND

The biological activity of every living organism is controlled by billions of individual cells. The control-center of each cell is the deoxyribonucleic acid (DNA) that contains a complete set of instructions needed to direct the functioning of each and every one of the cells. The chemical composition of the DNA is the same for all living organisms. The DNA of every living organism contains four basic nucleotide bases: **adenine, cytosine, guanine,** and **thymine,** ususally abreviated using the symbols **A, C, G** and **T** respectively.

The DNA sequence is usually divided into a number of chromosomes, which in turn contain genes. Genes are sequences of base pairs that contain instructions on how to produce **proteins**. They are also related to heredity. The areas of the DNA that contain genes are thus called **coding areas**, while the remaining parts are called **non-coding** areas. In higher-level eukaryotes, genes are usually spliced up into alternating regions of **exons** and **introns**. The introns are non-

[*] Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV26506-6109

[#] School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816

[$] Department of Computer Science, University of Canterbury, Christchurch, New Zealand

[1] Source: http://www.ncbi.nlm.nih.gov/Database/index.html

[2] Source: http://www.jgi.doe.gov/JGI_microbial/html/index.html

coding DNA and are cut-out before the messenger ribonucleic acid (mRNA) leaves the nucleus for the ribosome - where the protein specified by the mRNA is synthesized. The information in the mRNA thus represents the exons which are then used to make proteins. The non-coding area (sometimes called "junk DNA") contains redundant repeating sequences . They are estimated to make up at least 50% of the human genome.

From a computational viewpoint, a biological sequence can be viewed mainly as a one-dimensional sequence of symbols, for instance with an alphabet of 4 symbols for DNA, or RNA, and 20 symbols for proteins. Biological sequences typically contain different types of repetitions and other hidden regularities. Long runs of tandem repeats and of randomly interspersed repeats are prominent features of DNA sequences. The family of **Alu** repeats (typically about 300 bases in length) is typical of short interspersed repeat sequences (SINEs - short interspersed nuclear elements). These have been estimated to make up about 9% of the human genome, thus out-numbering the proportion of protein coding regions[Herzel94es]. There are also the long interspersed repeat sequences (LINEs - long interspersed nuclear elements) which are usually more than 6000 bases in length. In the human genome, the L1 family is the most common LINEs, with about 60,000 to 100,1000 occurrences. There are also short repeats (sometimes called "random repeats"), attributed to the fact that typical sequences and genomes are orders of magnitude larger than the alphabet size (4 in this case).

The function of the repetitive non-coding areas are not completely understood. Although the major attention has been on the coding areas, it has long been shown that the non-coding areas could be performing some important function [Gatlin72, Mantegna94bghpms], and hence should not be ignored. Repetition structures have been implicated in various diseases and genetic disorders. For instance, the triplet repeats $(CTG)_n/(CAG)_n$ have been associated with the Huntington's disease, while the hairpins formed in $(CGG)_n/(CCG)_n$ repeats have been linked to the Fragile-X mental retardation syndrome [Bat97ka]. While the exact function of each of the different genes so far identified is not completely known, even less is known about the repeats, and this represents a major challenge.

2.2 General Data Compression

Given that we represent the information in DNA sequences using four alphabets (A,C,G,T), if the sequences were to be completely random (that is, completely unpredictable or incompressible [Li93v]), then, we should need two bits to code each nucleotide base pair. Biological sequences are however known to convey important purposeful information between different generations of organisms. Moreover, from the view point of compression and sequence understanding, the repetitions inherent in biological sequences (as described above) imply redundancies which can provide an avenue for a signifcant compaction. The identification of such dependencies is the starting point for biological sequence compression.

There are three general classes of lossless compression schemes: ***symbol-wise substitution***, ***dictionary-based*** and ***context-based*** methods [Bell90mw, Witten99mb]. In symbol-wise substitution methods, each symbol is replaced with a new codeword, such that symbols that occur more frequently are replaced with shorter codewords, thereby achieving an overall compression. Under dictionary-based methods, a vocabulary of frequently occurring symbols or group of symbols is constructed from the input sequence. Compression is achieved by replacing the positions of the symbols with a pointer to their positions in the dictionary. The dictionary could be **off-line** or **on-line**. In on-line dictionary methods, the text itself is used as the dictionary, and symbols that have previously been observed in the sequence are replaced with pointers to the positions of their earlier occurrence. The popular LZ-family of algorithms is based on on-line dictionaries [Ziv77l, Ziv78z]. Off-line dictionary methods compress the input sequence in two passes: first to build the dictionary by identifying the repeating sequences, and the second to encode the repeats with pointers into the dictionary. Essentially, dictionary-based compression is a substitution scheme whereby the substitution is in terms of previously occurring symbols or sequence of symbols, and not necessarily in terms of codeword substitution. Rubin [Rubin76] and Wolff [Wolff77] describe various issues in dictionary-based schemes, including dictionary management. Storer and Szymanski [Storer82s] provide a general framework to describe different substitution-based schemes.

Context-based compression methods make use of the fact that the probability of a symbol could be affected by the nearby symbols. For instance, in an English text, when the symbol "q" appears, we know with a high probability that the next symbol should be a "u". Thus, context-based models are more or less techniques to estimate the symbol probabilities, which are then used by the symbol-wise approaches. The contexts are usually expressed in terms of symbol neighborhoods in the sequence. The PPM (prediction by partial match) family of compression schemes [Cleary97t] uses contexts of different sizes. Symbols are encoded by considering the previous occurrences of their current context and then choosing the longest matching contexts. Block-sorting methods also make use of contexts, but here the contexts are characters permuted in such a way that the contexts appear to be sorted - that is, symbols in lexicographically similar contexts will be clustered together. This clustering can thus be exploited to code the symbols in a compact manner. The block sorting approach is called the Burrows-Wheeler Transform or BWT for short [Burrows94w, Fenwick96]. The LZ-based algorithms are the most popular due mainly to their availability on most commercial computing platforms (utilities such as GZIP, WINZIP, PKZIP, GIF, and PNG are LZ-based schemes). However, in terms of compression performance, the PPM-schemes provide the best compaction, although they are slower. The BWT is second to

the PPM in compaction performance, but faster than the PPM schemes. They are, however, generally slower than the LZ-family[3]. This middle-ground performance has made the BWT an important addition to the long list of text compression algorithms.

2.3 Compression of Biological Sequences

Traditional compression methods as described above that have done very well on text often find it difficult to compress biological sequences, such as DNA or protein sequences. In fact, using the classical compression methods (such as word-based Huffman, arithmetic coding, or LZ-based methods) directly on such sequences often result in ***data expansion***, rather than compression [Grumbach94t, Rivals97ddd]. The major reason is the fact that these methods often use models that were derived for traditional text, and hence fail to consider certain special characteristics of biological sequences. Even when they do, (for instance in dictionary-base schemes), they often fail to produce reasonable compression. Yet, biological sequences are known to contain a lot of regularities which can be exploited for compression.

Most successful methods for compressing biological sequences thus consider the different regularities or repetition structures that are inherent in these sequences. Some repetition structures that can be exploited here include simple (interspersed) repeats (SINEs and LINEs), tandem repeats, palindromes, complemented repeats, and complemented palindromes. In each case, we may need to consider both cases of exact and in-exact (approximate) repetitions. The key problem becomes how we can speedily identify these repetition structures, and how we can make use of their knowledge in compressing the sequence. Different specialized algorithms have thus been proposed for compressing biosequences, with varying degrees of success. The different algorithms differ in the type(s) of repetitions they exploit, and how they do this.

The methods are generally on-line dictionary-based schemes, where the dictionary (or substitution mechanism) is constructed by considering the different types of repetition structures. Grumbach and Tahi [Grumbach94t] were among the first to propose a special purpose compression algorithm for DNA sequences. Their algorithms BIOCOMPRESS1 and BIOCOMPRESS2 [Grumbach94t] factor the input sequence into repetitive structures. Each factor (or repeat) is represented using a pair of integer numbers indicating the size of the repeat and the position where the factor appeared in part of the input already observed. The pair of integers is then coded using the Fibonacci code - a universal code for coding the integers. Before a factor is replaced, a check is made to determine if the replacement will lead to some compression - i.e. if the size of the coded representation of the pair of numbers will be smaller than the size of the original nucleotide bases without coding. The remaining parts of the sequence (i.e. those that could not be factored) are then coded using arithmetic codes. BIOCOMPRESS2 is quite similar to BIOCOMPRESS1, except that it provided mechanisms to handle palindromes.

In a similar approach, Rivals and colleagues proposed the CFACT algorithm [Rivals96ddd, Rivals97ddddo]. They distinguished between random repetition and non-random repeats. Random repeats are those that might have occurred due to the typically large size of the nucleotide sequence relative to the small alphabet size. Non-random repeats are those that should have occurred due to some biologically sound phenomena, such as mutations. Compression is achieved by avoiding the random repeats, which typically results in data expansion. In [Rivals97ddddho], they also proposed methods for compression by exploiting the approximate tandem repeats that are often observed in biological sequences. In [Chen99kl], substitution-based methods were also proposed for compressing DNA sequences. Their algorithm called GENCOMPRESS also handles approximate repeats. In addition, they showed how the results from GENCOMPRESS could be used to construct phylogenetic trees. In [Lanctot2000ly], GTAC (grammar transform analysis and compression) was proposed for estimating the entropy of a nucleotide sequence. They used context free grammars to develop a model that recognizes repeats, reverse repetitions, and complemented repeats in a sequence. Masumoto et al [Matsumoto2000si] proposed a method based on context-tree weighting and LZ77-parsing. Short repeats are encoded using the context tree, while long and inexact repeats are encoded using the LZ method.

The above schemes are on-line dictionary methods, and each used the LZ-parsing scheme at one stage or the other. They mainly differ in the way they make decisions at the time of replacing the factored repeats, in the type of repeat structures that they recognize, and whether they handle approximate repeats or not. Apostolico and Leonadi [Apostolico2000l, Apostilico2001l] took a different approach, by making explicit use of off-line dictionaries, using a simple greedy parsing scheme. The greedy parsing approach implies that decisions on the replacement of a particular repeating sequence are not necessarily optimal. Moreover, they did not explicitly consider the different forms of regularities in a DNA sequence, neither did they consider possible compression of the dictionary. But they showed that, in general, simple off-line schemes outperform the classical algorithms on biological sequences. Loewenstern and Yainilos[Loewenstern99y] proposed an approach to estimate the entropy of DNA sequences by using inexact matches based on a family of distance measures, which are then weighted to produce a single prediction. The distance measure was based on simple Hamming distances, taking *n*-block sequences at a time.

3. THE BURROWS-WHEELER TRANSFORM

The BWT [Burrows94w] performs a permutation of the characters in the sequence, such that characters in lexically similar contexts will be near to each other. The important procedures in BWT-based compression/decompression are

[3] See benchmark results at the Canterbury Corpus: http://www.corpus.canterbury.ac.nz

the forward and inverse BWT, and the subsequent encoding of the permuted sequence.

3.1 The forward transform.

Given an input sequence $T = t_1, t_2, \ldots, t_u$, the forward BWT is composed of three steps: i) Form u permutations of T by cyclic rotations of the characters in T. The permutations form a $u \times u$ matrix **M'**, with each row in **M'** representing one permutation of T; ii) Sort the rows of **M'** lexicographically to form another matrix **M**. **M** includes T as one of its rows; iii) Record ***L***, the last column of the sorted permutation matrix **M**, and ***id***, the row number for the row in **M** that corresponds to the original sequence T. The output of the BWT is the pair, (***L, id***). Generally, the effect is that the contexts that are similar in T are made to be closer together in L. This similarity in nearby contexts can be exploited to achieve compression. As an example, suppose T=ACTAGA. Let F and L denote the array of ***first*** and ***last*** characters respectively. Then, F=AAACGT and L=GATAAC. The output of the transformation will be the pair: (***L, id***)=(GATAAC, 2) - indices are from 1 to u. The rotation matrices for the sequence T=ACTAGA is given below:

	M'			**M**	
			F		***L***
1	A C T A G A $			$ A C T A G A	
2	C T A G A $ A		**A**	**A** $ A C T A G	**G**
3	T A G A $ A C		**A**	**A C T A G A** $	**A**
4	A G A $ A C T	**sort** →	**A**	**A G A** $ A C T	**T**
5	G A $ A C T A		**C**	**C T A G A** $ A	**A**
6	A $ A C T A G		**G**	**G A** $ A C R A	**A**
-	$ A C T A G A		**T**	**T A G A** $ A C	**C**

Figure 1. BWT forward transformation. M' is the matrix of cyclic rotations before sorting. M is the matrix after sorting. We have included the extra symbol $ for consistency in the suffixes.

3.2 The inverse transform.

The BWT is reversible. It is quite striking that given only the (L,id) pair, the original sequence can be recovered exactly. The inverse transformation can be performed using the following steps [Burrows94w]: i) Sort L to produce F, the array of first characters; ii) Compute V, the **transformation vector** that provides a one-to-one mapping between the elements of L and F, such that $F[V[j]] = L[j]$. That is, for a given symbol $\sigma \in \Sigma$, if $L[j]$ is the c-th occurrence of σ in L, then $V[j]$=i, where $F[i]$ is the c-th occurrence of σ in F; iii) Generate the original sequence T, since the rows in **M** are cyclic rotations of each other, the symbol $L[i]$ cyclically precedes the symbol $F[i]$ in T. That is, $L[V[j]]$ cyclically precedes $L[j]$ in T. For the example with T=ACTAGA, we will have V = [**5 1 6 2 3 4**]. Given V and L, we can generate the original text by iterating with V. This is captured by a simple algorithm: $T[u+1-i] = L[V^{i-1}[id]], \quad \forall i = 1,2\ldots,u$, where $V^0[s] = s$, and $V^{i+1}[s] = V^i[s], 1 \le s \le u$. With V, we can use the relationship between L, F, and V to avoid the sorting required to obtain F. Thus, we can compute F in linear time.

3.3 BWT-based compression.

Compression with the BWT is usually accomplished in a four-phase pipeline, viz. *input*→ BWT→ MTF→ RLE→ VLC→ *output*. Here, BWT stands for the forward BWT transform; MTF is move-to-front encoding [Bentley86stw] used to further transform L for better compression (this usually produces runs of the same symbol); RLE is run length encoding of the runs produced by the MTF; and VLC is variable length coding of the RLE output using entropy encoding methods, such as Huffman or arithmetic coding.

3.4 Auxiliary transformation arrays.

In order to provide some form of random access to the transformed BWT output, we introduced auxiliary transformation vectors [Adjeroh2002mpbz, Bell2002pbma]. Define an array ***Hr*** computed from V by the following algorithm: ***x:=id; for i:=1 to u do*** **{*x:=V[x];Hr[u+1-i]:=x*}.** Given the arrays F and Hr, the original text can be retrieved by applying a new *inverse* transformation $T[i]$= $F[Hr[i]], 1 \le i \le u$. Another auxiliary array used is ***Hrs***, defined as the index vector to Hr. That is, $T[Hrs[i]] = F[i] = F[Hr[Hrs[i]]]$. With the example, T=ACTAGA, L=GATAAC, ***id***=2, F=AAACGT, V = [**5 1 6 2 3 4**], Hr=[**2 4 6 3 5 1**] and Hrs=[**6 1 4 2 3 5**]. Figure 2 provides an illustration of the auxiliary arrays and their relation to other vectors.

Idx	**T**	**L**	**F**	**V**	**Hr**	**Hrs**
1	A	G	A	5	2	6
2	C	A	A	1	4	1
3	T	T	A	6	6	4
4	A	A	C	2	3	2
5	G	A	G	3	5	3
6	A	C	T	4	1	5

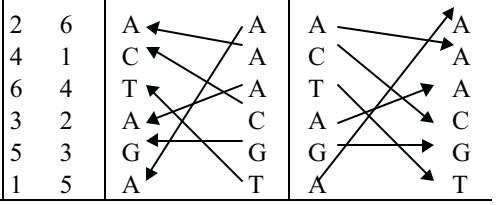

Figure 2: Auxiliary arrays for the sequence T=ACTAGA

3.5 The BWT and repetition structures.

The BWT is very closely related to the suffix tree and suffix array - two important data structures used in pattern matching and in analysis of repetition structures. The major link is the fact the BWT provides a lexicographic sorting of the contexts as part of the permutation of the input sequence. The suffix tree [McCreight76, Giancarlo95a] is a tree that contains all the suffixes of a given sequence, such that each leaf node in the tree corresponds to a suffix in the original sequence. A traversal of the tree from the root to the leaf reads out the corresponding suffix. An internal node in the suffix tree with more that one child leaf node implies that the substring obtained by tracing from the root to the internal node is a repeated substring in the sequence. A suffix array [Mamber93] is obtained by sorting the suffixes (or leaf nodes in the tree). Thus, a suffix array can be seen as a

lexicographically ordered list of the suffices. Given the starting position of each substring in the sorted array of suffixes, the suffix array can be represented as a simple array of these indices. Below we show the list of suffixes and the corresponding sorted suffixes for the example used above. The suffix tree and the suffix array are also included.

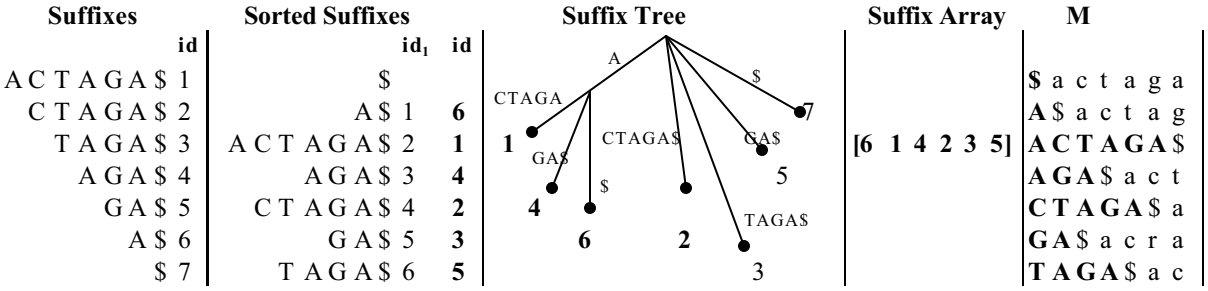

Figure 3. **Suffix tree, suffix array, and matrix of sorted rotations. The numbers on the leaf nodes in the suffix tree correspond to those on the suffixes (id), which indicate the starting position of the suffix in the sequence. The numbers on the sorted suffixes (id_1) indicate the sorted index of the suffixes. The corresponding position in the sequence (id) is also shown. The label on each link corresponds to a substring in the sequence.**

To see the relationship between the BWT and the suffix tree or suffix array, we have also included the final matrix of sorted rotations (**M**) from the BWT (described above). If we ignore the characters after the special symbol \$ in the final results from the BWT rotation and permutation procedures, the sorted suffixes correspond exactly to the results from the BWT. (We have used bold and uppercase letters to highlight these suffixes in the sorted matrix of rotations). Most importantly, the suffix array corresponds exactly to the auxiliary array, *Hrs*, defined previously!

This key relationship is the cornerstone of our approach to the analysis of repetition structures. By constructing the auxiliary arrays from the output of the BWT, we implicitly construct the suffix array from the original sequence. Further, with the availability of the two auxiliary arrays, *Hr* and *Hrs*, and the array of first characters *F*, we can access any part of the original sequence *T,* via the transformed results of the BWT.

3.6 Repetitions with suffix trees.

With the BWT, it becomes possible to use the internal structure of the BWT-transformed output to identify repetitions in the sequence. With the relationship between the BWT and suffix arrays (and hence suffix trees), we can use these known data structures to identify the repeating sequences. Gusfield [Gusfield97] provides a detailed study on the use of suffix trees for identification of repeats.

4. BWT-BASED COMPRESSION WITH REPETITION STRUCTURES

Having identified the various forms of repeats, we can turn to the problem of using these to compress the sequence. This is done by performing substitutions using the discovered repetition structures. Thus, we need to parse the input sequence to indicate the positions of the repeats and where and how they are to be substituted. Three approaches can be taken:

(a) **Off-line dictionary**: Remove the repeats from the original sequence, and move them into an off-line dictionary (or index) of repeats. Code all occurrences of the repeat with reference to the position of the repeat in the dictionary. The size of the pointers is likely to be generally smaller than when we use on-line dictionary (especially with absolute references), since we should expect that the number of repeats should be much smaller than the size of the input sequence.

(b) **Online dictionary:** Use pointers into the sequence itself rather than to an index in an external dictionary. Substitution and on-line dictionary-based approaches using LZ-type parsing are the predominant methods used by current DNA compression schemes [Grubach94t, Rivals96dddd, Matsumoto2000si].

(c) A **combination** of offline and on-line dictionary

The simplest to implement is the offline dictionary. But we have to compress and send the off-line dictionary as part of the compressed string. To guarantee compression, we can enforce a condition that whenever an item is inserted in the dictionary, it should not lead to a significant expansion of the data. We also have to consider whether referencing using the pointers should be relative or absolute references.

Below we present two off-line schemes. The schemes are similar to those used by Apostolico and Leonadi [Apostolico2000l], but with considerations of different repetition types, different dictionary organization schemes, and dictionary compression. We call the algorithm vocabulary-parsing scheme (**vps**). The compression performance will depend on a number of factors, such as: the size of the reference numbers (the pointers), and the way they are coded; the type of referencing used - absolute or relative; the type of dictionary used - on-line or off-line; the size of the dictionary (i.e. number of distinct repetition structures); and the type of referencing used in the dictionary (if any), especially for offline dictionaries.

4.1. Algorithm VPS1:

Off-line dictionary with pointers in the dictionary

In this scheme, we remove each repeated substring from the input sequence, and move it to an external dictionary. In the dictionary, we record the positions in the sequence where each repetition occurred, along with the repetition type. Thus there is no reference or pointer information in the original

sequence. The parsed string is just a concatenation of the remaining subsequences after the repeats have been removed. To capture the case of different repetition types, we include **repetition codes** for each type of repetition in the dictionary. Using the following repetition codes: **1**: repeat; **2** reverse repeat; **3**: palindrome; **4**: compliment; **5**: complimented palindrome; **6**: reverse compliments, we have the following general structure for the dictionary:

index	***r***	***l*(*r*)**	***rT*(*r*)**	**$\eta(r)$**	**positions**
i	r_i	$l(r_i)$	1	$\eta(r_i,1)$	p(i,1,1), p(i,1,2) …
			2	$\eta(r_i,2)$	p(i,2,1), p(i,2,2) …
			3	$\eta(r_i,3)$	p(i,3,1), p(i,3,2) …
			4	$\eta(r_i,4)$	p(i,4,1), p(i,4,2) …
			5	$\eta(r_i,5)$	p(i,5,1), p(i,5,2) …

Where r_i is the *i*-th repetition pattern, $l(r)=|r|$ is the length of repeat pattern *r*, $rT(r)$ is the repetition code (or repeat type) for *r*, $\eta(r)$ is the total number of occurrence of *r*, $\eta(r, j)$ is the number of occurrences of *r* with repetition type *j*, $\left(\eta(r)=\sum_j \eta(r,j)\right)$, and $p(i,j,k)$ is the position in the sequence of the *k*-th instance of repetition pattern r_i, with repetition type *j*.

Example. Consider the following sample sequence, **S**:

	P1		P2		P3		P4		P5		P6		P7
x_1	AACTGTCAA	x_2	AA	x_3	GTCAA	x_4	TG	x_5	AACTG	x_6	TTGACAGT	x_7	AA

When we move the repeated sequences into the dictionary, the remaining parsed sequence will be:

Parse(S) : $x_1x_2x_3x_4x_5x_6x_7$

The x_i's represent some other parts of the sequence that are not included in the repetition structure. Thus, the parsed sequence from **Algorithm VPS1** is very simple. The dictionary however is a little more complicated.

Ignoring the entries with $\eta(r_i, j) = 0$, the dictionary structure for the sample sequence is given below:

index	**r**	**l(r)**	**t(r)**	**$\eta(r)$**	**positions**
1	AACTGTCAA	9	1	1	P_1
			5	1	P_6
2	GTCAA	5	1	1	P_3
			2	1	P_5
3	AA	2	1	2	P_2, P_7
4	TG	2	1	2	P_4

4.1.1 Analysis

The performance of the above scheme depends critically on the internal organization (or representation) used for the dictionary. We use the term ***vocabulary*** to refer to the ensemble of repeat structures without reference to their specific locations in the sequence. For simplicity, we consider only the case with one type of repeat. The analysis can be extended to the general case with different types of repeats. We analyze the expected compression gain for the above offline dictionary scheme for two organizations of the vocabulary. We note that we do not need to explicitly encode $l(r)$ and $\eta(r)$ since these can be computed on the fly. We code the integer numbers with any available instantaneous universal code for the integers, such Elias codes[Elias75] or Fibonacci codes[Apostolico87f]. Thus, we avoid direct coding of the delimiters in the above representations, since the codes are self-delimiting, and uniquely decodable.

We use the following additional notations: $P_{r,j}$ = position of the *j*-th occurrence of repeat pattern *r*, κ = dictionary size (i.e. number of distinct repetitions), $u = |S|$ = size of the sequence, Σ = input alphabet, ($|\Sigma|= 4$ for nucleotide base pairs), $\beta = \lceil \log|\Sigma| \rceil$ = number of bits required to code each base pair in the original sequence[4], $b(n) = \lceil \log n \rceil$ = number of bits required to represent an integer *n*, $C(X)$ cost of coding sequence *X*, $L = \max_i \{l(r_i)\}$ - the maximum length of the repeats, **0** and **1** are flag bits.

Vocabulary Encoding A:

Vocabulary: $r_1\ \mathbf{0}\ r_2\ \mathbf{0} \ldots \mathbf{0}\ r_\kappa\ \mathbf{0}$

Positions:

$$P_{1,1}, P_{1,2}, \ldots P_{1,\eta(1)}\ \mathbf{0}\ P_{2,1}, P_{2,2}, \ldots P_{2,\eta(2)}\ \mathbf{0} \ldots \mathbf{0} P_{\kappa,1}, P_{\kappa,2}, \ldots P_{\kappa,\eta(\kappa)}\ \mathbf{0}$$

From the above representation, we have the following:

Cost of **original sequence**: $C(S) = |S|.\lceil \log \Sigma \rceil = u.\beta$

Cost of **parsed sequence** (i.e. remaining sequence after removing repeats): $C(parse(S)) = u\beta - \beta\sum_{i=1}^{\kappa} l(r_i)\eta(r_i)$

Cost of **vocabulary**:

$C(V_A(S)) = \beta_1 \sum_{i=1}^{\kappa} \left(l(r_i)+1\right) = \kappa\beta_1 + \beta_1 \sum_{i=1}^{\kappa} l(r_i)$,

where $\beta < \beta_1 \le \beta + 1$. Essentially, the addition of the flag bit **0** forms a new alphabet with the original alphabet Σ.

Cost of **positions**:

$$C(Posn(S)) = \sum_{i=1}^{\kappa}\sum_{j=1}^{\eta(i)} b(P_{i,j}) + \sum_{i=1}^{\kappa} 1 = \kappa + \sum_{i=1}^{\kappa}\sum_{j=1}^{\eta(i)} \lceil \log P_{i,j} \rceil$$

The cost of dictionary representation is then the combined cost of the vocabulary and the positions:

$$C(D(S)) = C(V_A(S)) + C(Posn(S))$$

Compression Gain:

$$G(S) = C(S) - \left[C(D(S)) + C(Parse(S))\right]$$

$$= u.\beta - \left[\left(\kappa\beta_1 + \beta_1 \sum_{i=1}^{\kappa} l(r_i) + \kappa + \sum_{i=1}^{\kappa}\sum_{j=1}^{\eta(i)} \lceil \log P_{i,j} \rceil\right) + \left(u\beta - \beta\sum_{i=1}^{\kappa} l(r_i)\eta(r_i)\right)\right]$$

With $\beta_1 = \beta + 1$, we have an underestimation of the gain:

$$G(S) = \beta \sum_{i=1}^{\kappa} l(r_i)\left(\eta(r_i)-1\right) - \kappa\left(\beta+2\right) - \sum_{i=1}^{\kappa} l(r_i) - \sum_{i=1}^{\kappa}\sum_{j=1}^{\eta(i)} \lceil \log P_{i,j} \rceil$$

Vocabulary Encoding B

Here, the vocabulary is reorganized to reduce the number of flag bits used to encode the dictionary. We eliminate the single flag bit between each repeat structure (which leads to an extension of the original alphabet for the vocabulary) by

[4]All logarithms are to **base 2**, unless otherwise noted.

simply concatenating the repeat patterns. We sort the repeats in order of their lengths, and use alternating flag bits to distinguish between patterns of different lengths. By using the lengths $l(r)$, (as given by the sorted values), we can determine the size of the repeat pattern, and hence isolate each repeat in the concatenated vocabulary. The position information is coded as before, but they are now organized in order of $l(r)$.

Vocabulary: $r_1\ r_2\ \ldots r_\kappa$

Lengths $l(r_1), l(r_2), \ldots, l(r_k)$

Repeats sorted according to $l(r)$:

$$\overbrace{r_{2_1}..r_{2_i}}^{l(r)=2}\ \overbrace{r_{3_1}...r_{3_j}}^{l(r)=3}\cdots\ \overbrace{r_{L_1}...r_{L_l}}^{l(r)=L}$$

Encoding: $r_{2_1}...r_{2_i}\ r_{3_1}...r_{3_j}\cdots\ r_{L_1}...r_{L_l}$

Lengths: 2…2 3 …3 … *L*…*L*

Flag bits: 0…0 1 …10 … 01…1

Positions:

$$P_{1,1}, P_{1,2}, \ldots P_{1,\eta(1)}\ \mathbf{0}\ P_{2,1}, P_{2,2}, \ldots P_{2,\eta(2)}\ \mathbf{0}\ldots\mathbf{0} P_{\kappa,1}, P_{\kappa,2}, \ldots P_{\kappa,\eta(\kappa)}\ \mathbf{0}$$

New cost of vocabulary:

$$C(V_B(S)) = \kappa + \beta\sum_{i=1}^{\kappa} l(r_i)$$

This can be reduced further to:

$$C(V_B(S)) = \sum_{i=1}^{\eta_L}\lceil \log \eta_L(i)\rceil + \beta\sum_{i=1}^{\kappa} l(r_i) \le \kappa + \beta\sum_{i=1}^{\kappa} l(r_i)$$

where η_L denotes the number of dictionary items with distinct lengths, and $\eta_L(i)$ denotes the number of dictionary items with length $l(r) = i$.

In both cases the compression performance depends on the length of the repeats, their frequency of occurrence, and the particular positions in the text where they occurred. Thus, with this scheme, we can guarantee compression by making a first pass on the sequence to determine the above parameters, and then make a second pass to replace a given repetition structure only if it will lead to a positive compression gain. The compression however depends critically on the position of the repeat in the original sequence. This could be a large number for very long sequences, such as the human genome. One simple way to reduce the space taken by the positions is to encode each position relative to the previous position. Since the difference is guaranteed to be smaller than the absolute position, some further gain can be achieved. However, there could be a better approach that avoids directly coding the position information in the sequence, by using pointers (or references) from the sequence into the dictionary (rather than vice versa).

4.2. Algorithm VPS2:

Off-line dictionary with pointers in the sequence

In the second approach, we identify the different repetition structures, and replace their positions in the sequence with reference indices indicating their positions in the dictionary, and the type of repeat. The parsed sequence will then contain a combination of integer references and the remaining part of the original sequence (the areas with no repeats). In **Algorithm VPS1**, we had a simple remaining parsed sequence, but a complicated dictionary. One attraction for **Algorithm VPS2** is the simplicity of the dictionary. The dictionary contains just the repetition structures. Although we might need statistics such as $l(r)$, and $\eta(r)$, these need not be explicitly coded in the dictionary. One major issue now is how the original sequence is to be parsed. We distinguish between two parsing strategies, depending on the schema used.

Parsing schema1: <reference index> <rI>

Parsing schema2: <reference index, repeat type > <rI,rT>

Parsing schema3: <index, repeat type, range> <rI, rT, sP, nP>

(sP, nP) define the range, where sP is the starting position in the repeat with index rI, and nP is number of base pairs to copy.

In **fixed-length parsing**, each reference code in the sequence must have the same number of parameters. In **variable-length parsing**, reference codes in the sequence are allowed to have different number of parameters. When the number of parameters are less than the maximum, flag bits are used to indicate the end of the reference code. The type of parsing depends on the parsing schema used. Variable length parsing results from using a combination of parsing schemas.

Example. Consider the sample sequence used previously **S**:

	P1		P2		P3		P4		P5		P6		P7
x_1	AACTGTCAA	x_2	AA	x_3	GTCAA	x_4	TG	x_5	AACTG	x_6	TTGACAGTT	x_7	AA

The dictionary will be:

rI	**repeat, r**
1	AACTGTCAA
2	GTCAA
3	AA
4	TG

Using fixed-length parsing, we can obtain the following parsed sequence:

Parse(S): x_1**<1,1>**x_2 **<3,1>**x_3**<2,1>**x_4**<4,1>**x_5**<2,2>**x_6**<1,5>**x_7**<3,1>**

Using variable-length parsing, we obtain the following:

Parse(S): x_1**<1>**x_2**<3>**x_3**<2>**x_4**<4>**x_5**<1,1,1,5>**x_6**<1,5>**x_7**<3>**

The parsing is not unique. For instance, we can obtain a different parsing with the changes: replace <2> with <1,1,1,5>; replace <1,1,5,5> with <1,2,1,5>; etc. Each modification could lead to a different compression gain. Thus, we could consider the different possibilities at each step and then determine the local compression gain or loss that will result, and then use the best one. Since the computation that may be required for these possibilities might not be trivial, there is a need for a tradeoff between local compression gain and computation time.

Compression with variable length parsing is more complicated than using the fixed-length method. But it has a potential for a much better compression. For instance, better

compression will be achieved if use the case of single-parameter parsing (<rI>) to code the repetition types with the highest frequency of occurrence. We will say a particular parsing will lead to a compression gain if the cost of coding the parsing - the repeat code type, pointers, and references will be less than that of coding the repeat structure without substitution. Depending on the organization of the dictionary, and the replacement strategy used, a local optimal compression gain may or may not necessarily lead to a global optimal compression gain.

4.2.2 Analysis

Unlike the case of pointers from the dictionary to the sequence where performance depended mainly on the design of the dictionary, here the performance depends more on how the parsed sequence is represented (encoded). To encode the parsed sequence, we decompose it into two parts: **ParsePart1** and **ParsePart2**. We remove the reference codes in the parsed sequence and replace each with a flag bit. This part could be coded with an optimal instantaneous code, such as the Huffman or arithmetic code. The second part contains the reference codes, which are concatenated with a flag bit between them. This part can be coded with any universal integer code, such as Elias or Fibonacci codes. For the sequence used previously, we have the following for the parsed sequence:

ParsePart1: $x_1\mathbf{0}x_2\mathbf{0}x_3\mathbf{0}x_4\mathbf{0}x_5\mathbf{0}x_6\mathbf{0}x_7\mathbf{0}x_8\mathbf{0}x_9$

ParsePart2:

<1>**0**<3>**0**<2>**0**<4>**0**<1,1,1:5>**0**<1,5>**0**<3>**0**<4>**0**

We avoid alphabet extension in **ParsePart1** by coding it using the method of vocabulary **encoding B**. That is, we code it using the length of its components as follows:

ParsePart1: $x_1x_2x_3x_4x_5x_6x_7x_8x_9$

ParsePart1 lengths: $sk_1, sk_2, sk_3\ sk_4, sk_5, sk_6\ sk_7, sk_8, sk_9$ where sk_i = length of x_i .

Once again, the delimiters have been used for clarity, but may not be explicitly coded in the final. The major consideration here will be the choice of the reference index *rI*(*r*) for a given repeat pattern, and the choice of the repetition codes *rT*. For the repetition code, the simplest approach will fix the codes before hand, so that there will be no need to transmit this to the decoder. However, assigning the repetition codes based on the probability of each repetition type should lead to more compression in general.

Most existing approaches classify short repeats as being statistically random[5], and hence should not lead to any significant compression gain [Grumabch94t, Rival96ddd]. Yet, these short repeats are often the most predominant. Because of their relatively high frequency of occurrence, an approach that can exploit the redundancy in these apparently random repeats could lead to an improved compression. The choice of *rI*(*r*) has an important influence on the compression gain. This will be the key to exploiting the redundancy in short repeats, even when they could be classified as random.

[5] A repeat pattern *r* is **statistically random** if $l(r) < \log u$, where *u* is the length of the original sequence [Rival96ddd].

We can calculate the cost of replacing each occurrence of a given repeat pattern *r*. With **VPS2**, we have:

Cost of **ParsePart1**: $b(sk)$

Cost of **Parsepart2**: $b(rT) + b(rI)$

Cost of **vocabulary**: $\frac{l(r)\beta}{\eta(r)} + \frac{\log \eta_L(r)}{\eta(r)\eta_L(r)} \le \frac{l(r)\beta}{\eta(r)} + 1$,

where the second component is the *per-occurrence* cost of the repeat pattern in the dictionary.

Total cost: $c(r) \le b(sk) + b(rT) + b(rI) + \frac{l(r)\beta}{\eta(r)} + 1$

In a simple implementation, we used four repeat types (i.e. without palindromes), using $b(rT) = 2$, and used a fixed Huffman code for the decimal digits, with a 2-bit flag as a delimiter between two numbers. With this (and ignoring the inequality), we obtain the overall *per-replacement* cost for a given repeat pattern *r*:

$$c(r) = 2 + b(sk) + 2 + 2 + b(rI) + \frac{2.l(r)}{\eta(r)} + 1, \text{ or}$$

$$c(r) = 7 + b(sk) + b(rI) + 2.l(r)/\eta(r).$$

We can then derive the constraints required on $l(r)$, $\eta(r)$ and $rI(r)$ for a positive compression gain $g(r)$:

$$g(r) = \beta.l(r) - c(r)$$
$$= 2l(r)\left(1 - 1/\eta(r)\right) - \left(7 + b(sk) + b(rI)\right).$$

For $g(r) \ge 0$, we require that:

$$\eta(r) \ge 1\Big/\left(1 - \frac{7 + b(sk) + b(rI)}{2l(r)}\right)$$

Thus, $rI(r) < 2^{\left[2l(r)\left(1 - 1/\eta(r)\right) - b(sk) - 7\right]}$

In the limits, as $n(r) \to \frac{u}{2}$, $sk \to 0$, or $b(sk) = 1$,

Thus, $rI(r) < 2^{2l(r)-8}$.

Since we require that $rI(r) > 0$, we must have $l(r) \ge 5$.

Hence, for a repetition pattern of a given length, we can analytically choose a range of index reference numbers to guarantee compression. The equation shows that we may not be able to use all the short repeat sequences - for instance, if $l(r) = 5$, we can choose only 4 of them. But we can rank the patterns by their frequency, and choose based on this ranking. The above relations also show that the reference index for the repeat could be large, for long repeats. The analysis above is for fixed-length, 2-parameter parsing, (i.e. with <rI, rT>). The analysis can be extended to the more general case of more than 4 repetition types, with variable-length parsing. Here, in addition to the lengths and the repetition probabilities, we also need to consider the probability of each repetition types, so as to assign the smallest repetition code to the most probable repetition class.

4.4. BWT compression with off-line dictionaries

The BWT provides a transformation of the input sequence to make it more suitable for compression. The question now is to

determine at what point on the BWT compression pipeline that we should embed the repetition analysis process. With the BWT compression model: ***input*** → **BWT** → **MTF** → **RLE** → **VLC** → ***output***, we can introduce the idea at different stages of the BWT compression process:

(a) Between input and BWT. That is, we use the discovery of the repetition structures as a kind of pre-processing, and then pass the parsed input sequence along with the dictionary to the BWT for compression. We perform the repetition analysis on the input data (i.e. constructing the suffix tree or suffix array on the original input data), before the BWT stage. Then the results are passed to the BWT for further compression.

(b) Between BWT and MTF. Here, the analysis to discover and code the repetitions will be done on the BWT output - the *L* array. But some natural repetitions in the sequence will be lost, while new repeat structures might be introduced by the BWT.

(c) Combination of (a) and (b). Perform repetition analysis before and after the BWT stage. This might be practically more time consuming, although it will have the same theoretical complexity as with the other two methods.

5. RESULTS

We tested the performance of the proposed methods using real DNA sequences from different types of organisms. The sequences used were: complete genomes of five mitochondria (**MIPACGA**, **MPOMTCG**, **mitoLB** - Loligo bleekeri, **mitoBP** - Balaenoptera physalus, and **mitoGallus** - Gallus gallus); two sequences from human DNA (**HUMGHCSA** - the human growth hormone, and **HUMHDABCD** - sequence of contig comprising three cosmids HDAB, HDAC, HDAD); and complete genomes of two viruses (**VACCG** - vaccinia virus, and **humEBV** - Epstein-Barr virus, also called human herpes virus 4), The sequences can be downloaded from the GENBANK database: http://www.ncbi.nlm.nih.gov:80/entrez/query.fcgi

We were mainly concerned with compaction ability, and hence did not consider the time taken for the compression. To code the integers, we used a simple fixed Huffman code for the decimal numbers, and a two-bit flag as separator between numbers. The results presented are only for **Algorithm VPS2**. We tested for when repetition analysis is performed only before BWT, or only after BWT. Table 1 shows the results when analysis of repetition structures is performed before the BWT, while Table 2 shows the result when the analysis is done after the BWT but before the move-to-front transformation (MTF). Results in bits per character (bpc) are shown in **boldface**, under the corresponding compressed file size. The tables also show the results when the sequences are compressed with two other popular compression schemes - arithmetic coding (**arithmetic**) and LZ-based UNIX program GZIP (**gzip**). The results under the other columns are obtained as follows:

BWT - input sequence is compressed using the traditional BWT algorithm (Mark Nelson's implementation [Nelson96]).

vps2a -- VPS2 parsing is performed on the input sequence, then the parse results are passed to BWT (without encoding the numbers into a binary representation);

vps2aa -- parsing is performed on the input sequence, the parse results are encoded into a binary sequence using the fixed Huffman code, and converted to ASCII. These are ***not*** passed to BWT.

vps2aBWT -- parsing is performed on the input sequence, the parse results are encoded into a binary sequence and converted to ASCII. These are passed to BWT for final compression.

vps2b -- parsing is performed on the BWT output when the original sequence is the input. The parse results are passed to further stages of the BWT compression pipeline - MTF, RLE, and VLC for compression.

vps2ba -- parsing is performed on the BWT output when the original sequence is the input. The parse results are encoded into a binary sequence using the fixed Huffman code, and converted to ASCII. These are ***not*** passed to the remaining stages of the BWT compression pipeline - MTF, etc.

vps2bMTF -- parsing is performed on the BWT output when the original sequence is the input. The parse results are encoded into a binary sequence and then converted to ASCII. These are finally compressed by passing then to further stages of the BWT compression pipeline - MTF, etc.

The tables show that the introduction of repetition analysis and parsing in the BWT compression pipeline generally improves the compression results, even without binary coding for the integers. In essence, the analysis and parsing stage further exposed the hidden regularities in the DNA sequence (such as reverse compliments), which typically will not be discovered by traditional compression algorithms, such as the BWT. This was the case, whether the analysis was done before the BWT, or after BWT but before the MTF. Further, the variable length coding of integers significantly improves the results (There was never a case where the result after coding (last two columns) was more than 2.0 bits per character (bpc)). Hence, the choice of the particular code to use is an important factor. We have used fixed Huffman codes with a flag for its simplicity. We envisage that more compression could be achieved by using higher-order Fibonacci codes for the integers. It can also be observed that repeat analysis and parsing before the BWT generally produced a better result than doing the parsing after BWT. One reasons for this is that the replacement of various types of repetition typically disrupts the original nature of the BWT output, and hence could make the results less suitable for MTF transformation. An analysis of the nature of the resulting sequence after the replacement of repeats will be important in matching the subsequent stages of the BWT compression pipeline to the parse results, and hence will be a key to further compression. It also appears

that **Algorithm VPS1** could be a better option when parsing is done after the BWT, since similar repeats will be clustered close to each other, and hence the difference in position numbers will be small. We are yet to test this.

Table 1. Results for repeat analysis and parsing before BWT

File	size (bytes)	gzip	arith-metic	BWT	vps2a	vps2aa	vps2a BWT
HumEBV	172281	40181	43171	41582	38492	38154	36605
	2.000	**1.866**	**2.005**	**1.931**	**1.787**	**1.772**	**1.700**
MPOMTCG	186609	54332	47121	47482	47357	46018	46084
	2.000	**2.329**	**2.020**	**2.036**	**2.030**	**1.973**	**1.976**
MIPACGA	94192	26997	22635	23723	23734	23395	22902
	2.000	**2.293**	**1.922**	**2.015**	**2.016**	**1.987**	**1.945**
HUMHDABCD	58864	16481	15077	14816	14831	14567	14466
	2.000	**2.240**	**2.049**	**2.014**	**2.016**	**1.980**	**1.966**
mitoBP	16398	4985	4204	4347	4362	4064	4090
	2.000	**2.432**	**2.051**	**2.121**	**2.128**	**1.983**	**1.995**
HUMGHCSA	66495	17168	17053	13369	13354	14169	13950
	2.000	**2.065**	**2.052**	**1.608**	**1.607**	**1.705**	**1.678**
mitoGallus	16775	5039	4291	4441	4456	4152	4173
	2.000	**2.403**	**2.046**	**2.118**	**2.125**	**1.980**	**1.990**
mitoLB	17211	4807	4188	4449	4420	4190	4110
	2.000	**2.234**	**1.947**	**2.068**	**2.055**	**1.948**	**1.910**
VACCG	191737	53973	46889	48120	48037	46950	46470
	2.000	**2.252**	**1.956**	**2.008**	**2.004**	**1.959**	**1.939**
Average(bytes)	**91174**	**24885**	**22737**	**22481**	**22116**	**21740**	**21428**
Average(bpc)	**2.000**	**2.235**	**2.005**	**1.991**	**1.974**	**1.921**	**1.900**

Table 2. Results for parsing after BWT but before MTF

File	size (bytes)	gzip	arith-metic	BWT	vps2b	vps2ba	vps2b MTF
humEBV	172281	40181	43171	41582	41563	42883	40847
	2.000	**1.866**	**2.005**	**1.931**	**1.930**	**1.991**	**1.897**
MPOMTCG	186609	54332	47121	47482	47460	46420	46539
	2.000	**2.329**	**2.020**	**2.036**	**2.035**	**1.990**	**1.995**
MIPACGA	94192	26997	22635	23723	23738	23427	22981
	2.000	**2.293**	**1.922**	**2.015**	**2.016**	**1.990**	**1.952**
HUMHDABCD	58864	16481	15077	14816	14799	14670	14542
	2.000	**2.240**	**2.049**	**2.014**	**2.011**	**1.994**	**1.976**
mitoBP	16398	4985	4204	4347	4358	4070	4100
	2.000	**2.432**	**2.051**	**2.121**	**2.126**	**1.986**	**2.000**
HUMGHCSA	66495	17168	17053	13369	13356	16588	14988
	2.000	**2.065**	**2.052**	**1.608**	**1.607**	**1.996**	**1.803**
mitoGallus	16775	5039	4291	4441	4447	4178	4193
	2.000	**2.403**	**2.046**	**2.118**	**2.121**	**1.992**	**2.000**
mitoLB	17211	4807	4188	4449	4430	4277	4181
	2.000	**2.234**	**1.947**	**2.068**	**2.059**	**1.988**	**1.943**
VACCG	191737	53973	46889	48120	48139	47515	46852
	2.000	**2.252**	**1.956**	**2.008**	**2.009**	**1.983**	**1.955**
Average(bytes)	**91174**	**24885**	**22737**	**22481**	**22477**	**22670**	**22136**
Average(bpc)	**2.000**	**2.235**	**2.005**	**1.991**	**1.990**	**1.990**	**1.947**

Finally, it may be observed that the results from the proposed methods (last three columns) are significantly better than the ones from traditional compression schemes, such as GZIP and arithmetic coding. In fact, as was pointed out earlier, these schemes (including BWT) often produce results with more than 2 bits per character.

6. CONCLUSION AND DISCUSSION

One major idea here is to analyze the DNA sequence in order to expose the various forms of regularities, and hence make the sequence more suitable for traditional text compression schemes. Another is to exploit the little compression in the predominantly short repeats. We have shown that indeed, we can code these short repeats in a way that can produce some compression gain. Because of the sheer number of these short repeats, the local gain will typically accumulate to a significant global compression gain. The described approach can be improved in a lot of ways.

Dictionary Compression. For long sequences, such as complete mammalian genomes, the size of the vocabulary could be quite large and hence could itself become a candidate for compression. Thus, the overall compression could be improved by considering different dictionary organization and compaction strategies, especially, the applicability of recursive decompositions for the compression of repetition dictionaries. Examples here include the hierarchical grammar-based approach [nManing2000w].

Compresison using multiple dictionaries. When compression is for a family of homologous sequences or genomes for species that have the same ancestor, the knowledge that the sequences could be similar could be useful for compression. Hence we can use the dictionaries from some selected members of the family to compress the other members. This will reduce the overall storage required for the dictionary, but could reduce the compression from the parsing stage. An important issue that requires further work is how to determine the criteria for choosing the members to use for dictionary generation.

Variable Length Coding. The overall compression will depend on the matching between the variable length codes and the distribution of source. We have used a simple fixed Huffman code in the experiments. We envisage that there could be some improvements if we use other variable length codes, such as higher order Fibonacci codes, since these may not need to explicitly code the flag bits. A similar comment applies to coding the new results after the MTF when repetition analysis is incorporated at some point(s) in the BWT compression pipeline. Further plans include tests on more DNA sequences, comparative analysis with other biological sequence compression algorithms, and more theoretical studies on the proposed approach

REFERENCES

[Adjeroh2002mtpz] Adjeroh D.A., Mukherjee A., Bell T.C., Powell M., and Zhang N., "Pattern matching in BWT-

transformed text", *Proceedings, IEEE Data Compression Conference*, Snowbird, Utah, April 2 - 4, 2002.

[Apostolico87f] Apostolico A. and Fraenkel A.S., "Robust transmission of unbounded strings using Fibonacci representations", *IEEE Transactions on Information Theory*, **33**(2):238--245, 1987.

[Apostolico200ll] Apostolico A and Lonardi S, "Compression of biological sequences by greedy off-line textual substitution", *Proceedings, IEEE Data Compression Conference*, Snowbird, UT, 2001

[Apostolico2000l] Apostolico A and Lonardi S, "Offline compression by greedy textual substitution", *Proceedings of the IEEE*, **88** (11), 1733--1744, 2000

[Bat97ka] Bat O, Kimmel M., and Axelrod D. E, "Computer simulation of expansions of DNA triplet repeats in the Fragile X Syndrome and Huntington's disease", *Journal of Theoretical Biology*, 188, 53-67,. 1997,

[Bell90cw] Bell T.C., Cleary J.C., and Witten I.H., *Text Compression*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[Bell2002pma] Bell T.C, Powell M., Mukherjee A. and Adjeroh D. A. "Searching BWT compressed text with the Boyer-Moore algorithm and binary search", *Proceedings, IEEE Data Compression Conference*, Snowbird, Utah, April 2 - 4, 2002. Available at:
http://www.cosc.canterbury.ac.nz/tim/tmp/dcc02match.pdf.

[Bentley86stw] Bentley J. L., Sleator D.D., Tarjan R.E., and Wei V., "A locally adaptive data compression scheme", *Communications of the ACM*, **29**(4), 320-330, 1986.

[Burrows94w] Burrows M. and Wheeler D.J., "A block-sorting lossless data compression algorithm", *Technical Report*, Digital Equipment Corporation, Palo Alto, CA, 1994.

[Chen99kl] Chen X., Kwong S.and Li M, "A compression qlgorithm for DNA sequences and its applications in genome comparison", In *Proceedings, 10th Workshop on Genome Informatics* (GIW'99), pp. 52--61, 1999.

[Cleary97t] Cleary J.G. and Teahan W.J., "Unbounded length contexts for PPM", *The Computer Journal*, **40**(2/3), 67-75, 1997

[Collins98pjcgw] Collins F.S., Patrinos A., Jordan E, Chakravarti A, Gesteland, and Walters L, "New goals for the US Human Genome Project: 1998-2003", Sciecne 282(5389):682– 689, 1998.

[Elias75] Elias. P., "Universal codeword sets and the representation of the integers", *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[Fenwick96], Fenwick P., "The Burrows-Wheeler Transform for block sorting text compression", *The Computer Journal*, **39**(9), 731-740, 1996.

[Gatlin72] Gatlin, L, *Information Theory and the Living System*, Columbia University Press, New York, 1972.

[Giancarlo95] Giancarlo R., "A generalization of the suffix tree to square matrices, with applications", *SIAM Journal on Computing*, **24**(3), 520-562, 1995.

[Grumbach94t] Grumbach S and Tahi F., "A new challenge for compression algorithms: genetic sequences", *Info. Processing & Management*, **30**: 875-886, 1994.

[Gusfield97] Gusfield D., *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, UK, 1997.

[Herzel94es] Herzel, H., Ebeling, W., and Schmitt, A, "Entropies of biosequences: The role of repeats", *Physical Review E*, **50**(6):5061--5071, 1994.

[Lanctot2000ly] Lanctot J. K., Li M. and Yang E., "Estimating DNA sequence entropy", *Proceedings,* 11th *Annual ACM-SIAM Symposium on Discrete Algorithms* (*SODA2000*) , pp. 409-418, 2000.

[Li93v] Li M and Vit'anyi P, *An Introduction to Kolmogorov Complexity and its Applications*, Springer--Verlag, 1993.

[Loewenstern99y] Loewenstern, D. and Yainilos, P., "Significantly lower entropy estimates for natural DNA sequences", *Proceedings, IEEE Data Compression Conference*, Snowbird, UT, pp. 151-161, 1997.

[Manber93m], Manber U. and Myers G., "Suffix arrays: A new method for on-line string searches", *SIAM Journal of Computing*, **22**(5), 935-948, 1993.

[Mantegna94bghpms] Mantegna R.N, Buldyrev S.V., Goldberger A.L., Peng C.K., Simons M., and Stanley H.E., "Linguistic features of noncoding DNA sequences", *Physical Review Letters*, 73, 23, 3169-3172, 1994.

[Matsumoto2000si] Matsumoto T., Sadakane K. and Imai H, "Biological sequence compression algorithms", Department of Information Science, University of Tokyo, 2000 . http://citeseer.nj.nec.com/457261.html

[McCreight76] McCreight E. M., "A space-economical suffix tree construction algorithm", *Journal of the ACM*, **23**(2), 262-272, 1976.

[Nelson96] Nelson M., "Data Compression with the Burrows-Wheeler Transform", *Dr. Dobb's Journal*, Sept. 1996. http://dogma.net/markn/articles/bwt/bwt.htm

[nManning2000w] Nevill-Manning C.G and Witten I.H. "On-line and off-line heuristics for infering hierarchies of repetitions in sequences", *Proceedings of the IEEE*, **88** (11), 1745--1755, 2000

[Rivals96ddd] Rivals E., Delahaye, J-P, Dauchet M. and Delgrange O., "A guaranteed compression scheme for repetitive DNA sequences", *Proceedings, IEEE Data Compression Conference*, Snowbird, UT, pp. 453

[Rivals96ddddho] Rival, E., Delgrange, O., Delahaye, J.P., Dauchet, M., Delorme, M. Henaut, A. and Ollivier, E., "Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences", *CABIOS*, **13**: 131-136, 1997

[Rubin76] Rubin F., "Experiments in text file compression", *Communications of the ACM*, **19** (11), 617--623, 1976.

[Storer82s] Storer, J.A. and Szymanski, T.G., "Data compression via textual substitution," *Journal of the ACM ssociation for Computing Machinery*, 29(4), 928-951, 1982.

[Witten99mb] Witten I. H., Moffat A. and Bell T. C., *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufman, 1999.

[Wolff78] Wolff, J. G., 'Recoding of natural language for economy of transmission or storage," *The Computer Journal*. 21 (**1**), 42-44, 1978.

[Ziv77l] Ziv J. and Lempel A., "A universal algorithm for sequential data compression", *IEEE Transactions on Information Theory*, **23**(3) 337-343, 1977.

[Ziv78l] Ziv J. and Lempel A., "Compression of individual sequences via variable rate coding", *IEEE Transactions on Information Theory,* **24**(5), 530-536, 1978.