Overview and Comparison of Genome Compression Algorithms

Tim Snyder
University of Minnesota, Morris
Department of Computer Science
600 East 4th St.
Morris, Minnesota
snyde479@morris.umn.edu

ABSTRACT

Genomic data is being created at a dramatically increasing pace. In the past, researchers were able to rely on the trend of cheaper storage space to store that genomic data. However, the pace at which genomic data is being created is increasing faster than the ability to purchase cheap storage mediums. To help keep storage costs down, compression algorithms must be used to keep the size of the data as small as possible. I will be analyzing and presenting genomic compression algorithms for both single genomes and sets of genomes. The single genome compression algorithms that I will present are the Expert Model by Cao *et al.* and Tabus and Korodi's algorithm and the database compression algorithms are COMRAD by Kuruppu *et al.* and Heath *et al.*'s algorithm. Comparison of algorithms indicates that there is room to make new algorithms to compress sets of genomes and that compressing sets is more effective.

Categories and Subject Descriptors

E.4 [**Data**]: Coding and Information Theory—*Data Compaction and Compression*

General Terms

Algorithms

Keywords

genome, compression, XM, COMRAD, database, single, arithmetic compression

1. INTRODUCTION

Kryder's law states that every twelve months, storage capacity per dollar doubles [13]. Despite this law of computing, genome storage is becoming increasingly more expensive because genomic data is being created at an even greater rate. In the past, researchers were able to rely on the trend of cheaper storage space to store the genomic data being generated. However, certain trends in sequencing, such as the maturation of second generation sequencing technologies, the creation of cheaper sequencing machines and the third generation of DNA sequencing technologies, are responsible for an ever increasing number of genomes being sequenced. These genomes are from both new species and more individual creatures having their genomes sequenced. This increasing rate of sequencing is outpacing Kryder's law even after general purpose compression algorithms are applied [9].

The genome of an organism is the DNA within that organism. DNA is comprised of nucleotides, also refered to as bases, which can be represented by the characters A, C, G and T for Adenine, Cytosine, Guanine and Thymine, respectively. In addition to those four letters specifying specific bases, there are letters which are wildcards and represent an arbitrary base or set of bases such as N for a non-specified string of bases and M for either Adenine or Cytosine. Converting the physical DNA to a data file is called sequencing. The human genome is about 3,000 megabytes of uncompressed data. In comparison, the complete works of William Shakespeare is about 5 megabytes.

This paper will present and compare four algorithms designed to compress genomes. Background information and a pair of non-genome specific algorithms will be presented first. Then, I will present a pair of single genome compression algorithms in Section 3, Expert Model by Cao *et al.* [2] and Tabus and Korodi's algorithm [10]. Following that, I will present a pair of database genome compression algorithms in Section 4, COMRAD by Kuruppu *et al.* [6] and Heath *et al.*'s [7] algorithm. Finally, I will draw some conclusions from the papers presented.

2. BACKGROUND

Before we discuss the algorithms in this paper, it is necessary to understand some general compression algorithms, their performance and Markov models which appear in two of the algorithms in this paper.

2.1 General Compression Algorithms

Compression algorithms are algorithms that will attempt to store data in a manner that takes up less space than the original data set. There are two kinds of compression algorithms; lossy and lossless. Lossy compression loses data from the original data set, which is fine for images and movies, but is not acceptable for genomes. Lossless compression

CSci Senior Seminar '12 Morris, Minnesota USA
.

keeps all of the original data, which is what is required for genome compression since the entire set of data is what is important when looking at genomes. General purpose compression algorithms do not care about what kind of data is being compressed aside from the format they are given for compression. This paper will only look at compression algorithms designed for compressing strings.

As far as efficiency is concerned, compression algorithms can be rated in one of two ways. The first is by dividing the size of the compressed data set by the size of the original data set which gives the percentage of size still used. This is referred to as a compression ratio. So, if a 500 byte file was compressed to 250 bytes, the compression ratio would be 50%. The second way to rate compression is in the average number of bits per character. So, assuming that the 500 byte file was encoded using the standard 8 bits per character with unicode, the 250 byte version would use 4 bits per character.

Before looking at genome specific algorithms, it is necessary to know how effective general purpose algorithms are for compressing genomes. One general purpose compression algorithm that works well on repetitive text is the Limpel-Ziv-Welch (LZW) algorithm. This seems like it would work well on genomes since they are highly repetitive. LZW starts with a dictionary of all possible characters as keys and values of the index in the dictionary, a dictionary being a data structure holding key-value pairs for easy searching.

Algorithm 1 LZW pseudo code

```
Generate dictionary
s = ""
while file has more characters do
    t = s
    c = next character
    s+= next character
    if dictionary does not contain s then
        put s and dictionary size in dictionary as next pair
        output index of t in dictionary
        s = c
    end if
end while
```

It reads in a character and, if the stored string concatenated with this new character is in the dictionary, it repeats this step. Otherwise, it outputs the binary for the index of the stored string and sets the string that it is looking for in the dictionary to the new character. By repeating this process until there is no more input, LZW can reduce the amount of space for a data file that has repeated patterns. Decoding the binary is done with the same starting dictionary of characters and indices and takes the binary data and slowly adds what it finds to the decompression dictionary in the same way that it did when compressing, except it outputs the characters instead of binary. Since LZW recreates the dictionary as it decompresses, there is no need to store the dictionary from the compressing of the file [12].

While this might seem like a good algorithm to use because genomes are highly repetitive, in practice, the LZW algorithm actually makes genomes larger [5, 7]. Clearly, this is not a solution to the problem of genome compression. Fortunately, this is not the only generic compression algorithm. Next up is arithmetic coding.

Arithmetic coding is done by taking the probability that a given character will occur and fitting the probabilities in the range [0,1). Then by reading in data character by character, a decimal is created by taking the number that fits the ranges of each of the characters in the string in order. In the output file before the decimal would be the number of characters that are being represented.

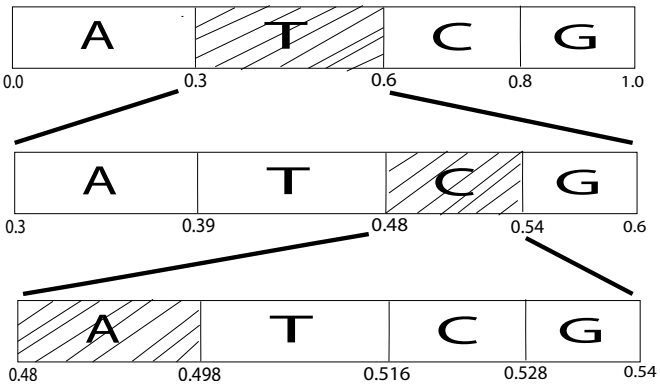

Figure 1: Example of Arithmetic coding with probabilities of 30% for A and T and 20% for C and G. See text for details.

Algorithm 2 Arithmetic coding pseudo code

```
start = 0.0
end = 1.0
numChars = 0
while file has more character do
    c = next character
    newStart = start
    for i = 0;i<index of c in probabilities;i++ do
        start+=probabilities[i]*newStart
    end for
    end = start+newStart*probabilities[c]
    numChars++
end while
output shortest decimal between start and end
output numChars
```

An example of arithmetic coding in action is seen in Figure 1. The probabilities in this example are 30% for A and T and 20% for C and G. The string being compressed is TCA, so we start with our range of [0,1) and fit it into where T would be in that range. This takes our range down to [0.3,0.6). We recreate the distribution in this new range and repeat. This will give us a range of [0.48,0.54) for the string TC and [0.48,0.498) for the full string of TCA. Any decimal in that range will give us a representation for TCA, so we would want to pick the shortest binary decimal in that range which would be 0.011111. [11]

Using arithmetic coding gives us about 2 bits per character for genomic data which should be the baseline for other compression algorithms. 2 bits per character makes sense given that there are only 4 characters that are regularly used in genomes and 2 bits has 4 possible states.

2.2 Markov Models

A Markov model is a model that allows for predictions of the next state of a machine from its current state. In addition, an order-n Markov model will take the current

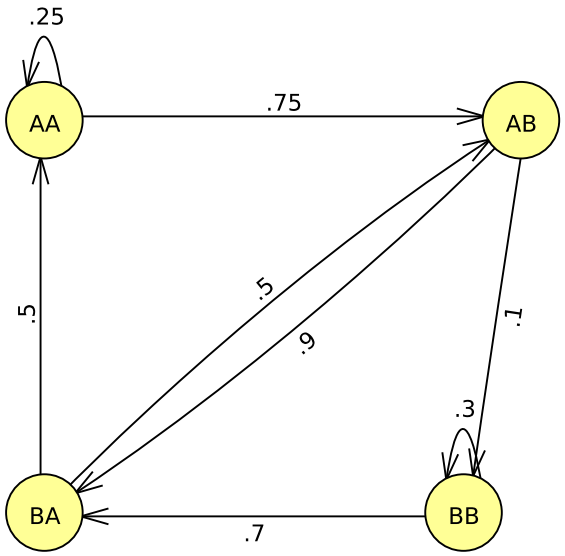

Figure 2: An example of an order-1 Markov model with 2 characters

state and the previous n states to predict the next state. In genomic compression, these are used to create probability distributions for the next character for arithmetic coding. In Figure 2.2, there is a simple order-1 Markov model on the characters A and B. The characters in the state are the current and previous input. Using them, we get the probabilities of moving to a different state with the next input. So, if the last two characters seen are both A, we have a 25% chance of seeing an A next and a 75% chance of seeing a B. We can feed those probabilities into arithmetic coding to generate the decimal for compression.

3. SINGLE GENOME COMPRESSION ALGORITHMS

Single genome compression algorithms take in a single genome and output the compressed genome. These algorithms have the benefit of being able to run on genomes as they are generated as well as having the benefit of being able to decompress the genome on its own without the need for more data. I will explain two single genome compression algorithms. The first is Expert Model by Cao *et al.* and the second is Tabus and Korodi's algorithm.

3.1 Expert Model

This subsection will explain the Expert Model algorithm (XM) as presented by Cao *et al.* [2] and go over the results of testing XM.

XM uses arithmetic coding as described above. The unique portion is how it determines the probabilities for each of the characters. The algorithm starts with a population of experts. An expert is anything that gives a good probability distribution for a position in the sequence. Examples of experts are Markov models, which were described in Subsection 2.2, and a copy expert, which determines if something is likely to be a copy. After obtaining the probabilities from the population of experts, XM will combine the probabilities for each of the characters and feed them into arithmetic coding to generate the final decimal. In addition, XM weights the

Sequence	BioC	GenC	DNAC	DNAP	XM
CHMPXX	1.684	1.673	1.671	1.660	**1.657**
HEHCMVCG	1.848	1.847	1.849	**1.834**	1.842
HUMHBB	1.880	1.820	1.789	1.777	**1.751**
Average	1.783	1.742	1.725	1.714	**1.694**

Table 1: XM's efficiency in bits per character. Table based on data from [2].

experts based on how accurate they have been in the past which is taken into account when finding the probabilities for arithmetic coding.

Cao *et al.* compared XM to other genome specific compression algorithms. The algorithms included BioCompress [5], GenCompress [3], DNACompress [4] and DNAPact [1]. On one of the genomes, DNAPact performs better than XM, but in general, the results in the paper show that XM compresses better on more genomes and better on average than the other algorithms. Considering that the results are also less than the two bits per character provided by arithmetic coding, XM is clearly more efficient at compressing than generic compression algorithms.

There is one rather large flaw with the paper's description of XM. While the paper does explain what some experts are, it does not actually explain what experts were used to get the compressions in Table 1.

3.2 Tabus and Korodi's Algorithm

This subsection will explain the algorithm presented by Tabus and Kordi (TK) [10] and go over the results of TK.

TK starts by breaking the genomic sequence into non-overlapping blocks of equal size. Then it goes block by block compressing them. Compressing the block happens with three different methods and then the most efficient one of those is chosen for storage. The first method uses a Markov model. The second uses arithmetic coding to compress to 2 bits per base. The third and final compression method finds an approximate match to a previously compressed block and compresses based on the differences between them.

The details of Markov models are in Subsection 2.2 and arithmetic coding was described previously in Subsection 2.1. The last coding method starts by finding the previously compressed block that is the closest to the block currently being compressed. Then, it stores the differences between the two blocks as a string, x, of 1's and 0's where a 1 at position i means that they are different at position i and a 0 at position i means that they are the same at position i. Then arithmetic coding is used to store distances for the differences. An example would be if at some index the old block has A and the new block has G, the distance from A to G might be 2. So there would be an arbitrary cycle between A, C, G and T where moving from one to the next increases the distance for the difference by 1. The algorithm then counts up each of the substrings in x of length k used in both prefixes and suffixes. Using this count and the string of differences, a probability distribution is created to create prefixes that would compress the difference string the way arithmetic coding would. Then another probability distribution called a universal code is created from that probability distribution. The universal code is to make prefixes that, while not the best in a specific probability distribution, will be good enough over all probability distributions that could

be expected. Then we store the previous section being referenced, the universal code, the count of substrings and the starting substring of length k.

In order to decompress, the counts and starting substring are used to calculate the ending substring. This is possible because the number of substrings that start with the starting substring is either the same or one more than the number of substrings that end with the starting substring. If the two numbers are equal, then the ending substring must be the same as the starting substring. Otherwise, there must be one more substring ending in the ending substring than there are that start with the ending substring. Then the counts, probability distribution and starting and ending substrings are used to recreate the original string of differences. Then it is just a matter of undoing arithmetic coding and changing the characters from the referenced portion that was decompressed earlier.

So, if we have the strings ACGTGA and ACTGGA, the difference string would be 001100. Then we calculate the distances which will be 1 and 3 if our cycle goes A, C, G, T and A. We count up the substrings of the binary string of a certain length, say 2, and using them and the starting substring, "00", we create a universal code. We can undo this by taking the universal code and generating the starting substring, substring counts and differences. The substring counts are 2 for all substrings: "00", "01", "10" and "11". Since the count for "00" is even, we know that the ending substring must also be "00". If we had "00" odd instead, then there would be another substring with an odd count that would be the ending substring.

Tabus and Korodi ran their algorithm on each chromosome of a human genome. When wildcards are included in the genome, the algorithm does considerably better with an average compression of 1.449 bits per base on the human genome compared to the algorithm without wildcards' 1.616 bits per base. Unfortunately, Tabus and Korodi did not compare their algorithm to any other compression algorithms, so it is impossible to draw any conclusions from this paper alone about the algorithm's effectiveness.

4. GENOME DATABASE COMPRESSION ALGORITHMS

In this section, I will present genome compression algorithms that, instead of taking in a single genome, take in a set of genomes and compresses them all using knowledge combined from the genomes. The algorithms that I will present are COMRAD by Kuruppu *et al.* and an algorithm by Heath *et al.* The additional knowledge from a database compression algorithm allows for increased compression.

4.1 COMRAD

This subsection will go over the COMRAD algorithm and its experimental results as presented by Kuruppu *et al.* [6].

From the genomes given as input, COMRAD generates a dictionary of substrings of length L keeping track of the frequencies of each substring. COMRAD will then go through the strings to be compressed and count the number of places where the most numerous substring can be replaced making sure to not count overlapping instances. COMRAD replaces the substring with the most non-overlapping counts and stores the replacement that it made. COMRAD then repeats the counting of substrings and replacing of them until all of the counts of substrings that it finds fall under a certain threshold. Finally, COMRAD will output the final string and the replacements that were made in order. Decompressing from the file is rather simple. The compressed string is read in followed by the changes made. Then the changes are reversed one by one until the original string is rebuilt.

Input
aabcbcaabcabc
·
Step 1
aa:2 ab:3 bc:4 cb:1 ca:2
·
Step 2
aabcbcaabcabc (no candidate)
a**abc**bcaabcabc (no candidate)
aa**bcb**caabcabc (cand cnt bc:1)
aabc**bca**abcabc (cand cnt bc:2)
aabcbc**aab**cabc (no candidate)
aabcbca**abc**abc (no candidate)
aabcbcaa**bca**bc (cand cnt bc:3)
aabcbcaabc**abc** (no candidate)
aabcbcaabca**bc** (cand cnt bc:4)
·
Step 3
bc → A
aaAAaaAaA
· **Step 4**
aa:2 aA:2 AA:1 Aa:2
...
aA → B
aBAaBB
...
aB → C
CACB

Figure 3: Example of COMRAD in action from [6] with L=2.

Dataset	original	RLCSA	RLZ	COMRAD
Influenza	1.97	0.43	3.10	0.43
Hemoglobin	2.07	2.16	4.13	1.16
Bacteria	2.00	5.13	2.90	2.26
H. sapiens	2.18	2.54	0.50	1.44
Average	2.06	2.22	2.04	1.10

Table 2: COMRAD's efficiency in bits per character. Table based on data from [6].

An example of COMRAD on an arbitrary string is in Figure 3. In the first step, COMRAD counts the occurences of substrings of length $L = 2$. COMRAD selects `bc` since it is substring that occurs most frequently, in step 2, COMRAD recounts the number of substrings of bc making sure to not count overlaps. COMRAD replaces the substring with a character that does not show up in the input string unless a different substring might occur more often after the second counting. An example where the most common substring is not replaced at first can be seen in Figure 4. In this example, the substring `ab` can be replaced in more places than `aa` can. Because of this, `ab` gets replaces instead of `aa` to decrease the size of the string by the largest possible amount.

Table 2 shows some experimental results presented by Ku-

Input
aaaaababab
.
Step 1
aa:4 ab:3 ba:2
.
Step 2
aaaaababab (cand cnt aa:1)
aa**aaa**babab (cand cnt aa:2)
aaaa**aba**bab (no candidate)
aaaaa**bab**ab (no candidate)
aaaaab**aba**b (no candidate)
aaaaaba**bab** (no candidate)
...
ab → A
aaaaAAA
...

Figure 4: An example where COMRAD will replace a substring that is not the most common substring in the original string.

ruppu *et al.* RLZ is a previous algorithm by Kuruppu *et al.* and uses a form of LZW. RLCSA was designed to work well on repetitive text, but RLCSA is not designed specifically to compress DNA [8], so it is unclear why the authors chose that as an algorithm to compare with COMRAD. Particularly troubling, most of the compression rates from RLZ and RLCSA are worse than the 2 bits per character obtained by arithmetic coding. RLCSA will, on average, increase the size of the genome set from the original while RLZ doesn't compress the datasets on average. There is also an instance where COMRAD fails to compress the data. The original size of the bacteria set on which COMRAD fails to efficiently compress is about two bits per base while COMRAD compresses this dataset to 2.26 bits per character. The authors failed to explain this occurrence. However, COMRAD was quite effective on the other datasets and on average.

COMRAD is a simple algorithm which works well on most of the data sets given in the paper, but in certain cases, COMRAD makes the dataset larger. Algorithms that were used in the paper to compare to COMRAD could have been better chosen since one of them was not specific to genome compression and both of them were either worse or not considerably better than the standard 2 bits per character compression scheme.

4.2 Heath's Algorithm

This subsection explains the algorithm presented by Heath *et al.* [7] and goes over their results.

Their algorithm starts by using a sequence alignment application called MUSCLE to align the sequences with a reference genome. The algorithm then determines the most efficient way to transform the reference genome to the current genome with the five operations of insertion, replacement, deletion, insertion after replacement and deletion after replacement, examples of which are in Figure 5. Insertions are noted with an index and the string of bases that is added starting at that index. Replacements are noted with the starting index and the string of bases that replace the string in the original sequence. Deletions are noted with the starting index and how many bases are being deleted. Insertion after replacement and deletion after replacement are replacing the base at the given index and then deleting or inserting the appropriate string following the index. While it is possible to do the algorithm without the last two of those five editing operations, Heath *et al.* argue that without them, there is ambiguity since an insertion at a given index and a replacement at the same index could mean two different things depending on the order that they are resolved in decompression and resolving this problem increases the complexity of the algorithm unnecessarily.

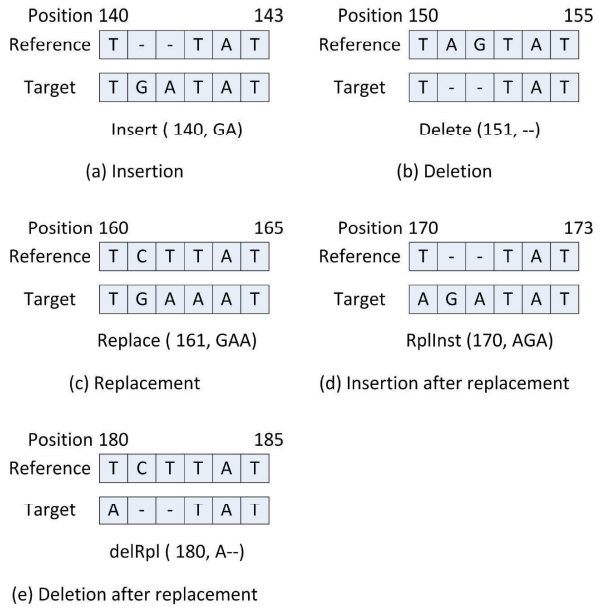

Figure 5: Examples of genome changes. Copied from [7]

	bzip2	DNAC	GenCom	binary	Huffman
Mito	0.966	0.761	0.861	0.975	0.988
H3N2	0.972	0.762	0.987	0.963	0.973

Table 3: Heath *et al.*'s algorithm's efficency in proportion of space saved using Huffman coding (Huffman) and without using Huffman coding (binary). Table based on data from [7].

This algorithm uses a generic compression algorithm called Huffman coding. Huffman coding is a special case of arithmetic coding and allows for a table of the characters and bit strings that would represent them to be created. The instructions and position for differences between the current genome and the reference, but not the bases, are combined into a single 32 bit integer since the maximum size of a human chromosome does not exceed 250 million base pairs which fits into 29 bits. The new ints and the base pair strings are then compressed with Huffman coding and then both the compressed data and the Huffman table are written to disk. However, according to the authors, there are some instances where a base pair string will be shorter using the standard 2 bits than using their own Huffman codes, so the algorithm will check for those cases and note when they would happen before the string being stored as either the standard compressed string or the string from their Huffman code.

As seen in Table 3, Heath *et al.* compared their algorithm

to standard compression software as well as two DNA specific compression algorithms, DNACompress and GenCompress. The algorithms were run on datasets of mitochondrial DNA, labeled "Mito" in the table, and H3N2 DNA, the influenza A virus. In the table, the algorithm labeled binary was their algorithm without the last stage of Huffman coding. The algorithm labeled Huffman is their algorithm including the final stage of Huffman coding. The values in the table are the proportion of the space from the original datasets that was saved, so 0.9728 means that the new files saved 97.28% of the original's size, or that they took up only 2.72% of the room of the original. On the two genomes tested, this algorithm does significantly better than DNACompress but only on the Mitochondrial genome does it do better than GenCompress and does slightly worse on the H3N2 genome.

Unfortunately, this algorithm was not designed for genomes with more base pairs than there are in the human genome. Heath *et al.*'s algorithm could be improved by replacing the Huffman coding step with an arithmetic coding step since Huffman coding is at best arithmetic compression. It is also a bit strange that they do not rate their algorithm's efficiency in bits per character since bits per character is the standard way to measure efficiency for genomic compression.

5. CONCLUSIONS

Most of the algorithms within this paper used bits per character as their measure for compression efficiency. However, Heath *et al.*'s algorithm was measured by the percentage of space saved in comparison to the original and the number of bases in the genomes being compressed is not given, so I could not get the compression rate into bits per character. In addition, Heath *et al.*'s algorithm is the only one which did not compress the human genome. Because of these two things, it is impossible for any conclusions to be drawn from the data for comparing to other algorithms presented in this paper.

The other algorithms are easier to compare, fortunately (see Table 4). XM reduced a human genome to 1.7513 bits per base, TK reduced a human genome down to 1.449 bits per base and COMRAD reduced a set of human genome down to 1.44 bits per base. XM did not compress nearly as effectively as either COMRAD or TK, it is clear that XM will not solve the problem of compressing genomes like TK or, in most cases, COMRAD.

Despite comprable compression rates on the human genome, TK is far more intricate than COMRAD. Unfortunately for COMRAD, there was a genome dataset which was increased when COMRAD tried to compress it. While TK is unable to have genomes larger than 2 bits per character, it does seem like it would be able to get near the compression rates of COMRAD whose average is over 20% less than TK's only run. The future of genome compression algorithms seems to be for databases since the single genome compressing TK is far more complicated than the dataset compressing COMRAD.

Acknowledgements

I would like to thank Peter Dolan for being my advisor, Nic McPhee for teaching the course, and Elena Machkasova, Chad Seibert and Jay Lapham for giving feedback.

	XM	TK	COMRAD
Human	1.7513	1.449	1.44
Average	1.6940	1.449	1.10

Table 4: Algorithms that gave bits per character compression rates on human genomes and averages over all data give. TK was only run on the human genome, so the two numbers are the same.

6. REFERENCES

[1] D. Boulton and C. Wallace. The information content of a multistate distribution. *Journal of Theoretical Biology*, 23(2):269 – 278, 1969.

[2] M. D. Cao, T. Dix, L. Allison, and C. Mears. A simple statistical algorithm for biological sequence compression. In *Data Compression Conference, 2007. DCC '07*, pages 43 –52, march 2007.

[3] X. Chen, S. Kwong, and M. Li. A compression algorithm for DNA sequences and its applications in genome comparison. In *Proceedings of the fourth annual international conference on Computational molecular biology*, RECOMB '00, pages 107–, New York, NY, USA, 2000. ACM.

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

[5] S. Grumbach and F. Tahi. A New Challenge for Compression Algorithms: Genetic Sequences. *Information Processing & Management*, 30, 1994.

[6] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel. Iterative dictionary construction for compression of large DNA data sets. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 9(1):137–149, Jan. 2012.

[7] A.-p. H. Lenwood S. Heath and L. Zhang. A genome compression algorithm supporting manipulation. In *9th Annual International Conference on Computational Systems Bioinformatics (CSB 2010) Proceedings*, volume 9, pages 38 –49, august 2010.

[8] J. Siréns. Run-length compressed suffix array, Nov. 2012.

[9] L. Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[10] I. Tabus and G. Korodi. Genome compression using normalized maximum likelihood models for constrained markov sources. In *Information Theory Workshop, 2008. ITW '08. IEEE*, pages 261 –265, may 2008.

[11] Wikipedia. Arithmetic coding — Wikipedia, the free Encyclopedia, 2012. [Online; accessed 26-September-2012].

[12] Wikipedia. Lempel - Ziv - Welch — Wikipedia, the free Encyclopedia, 2012. [Online; accessed 25-September-2012].

[13] Wikipedia. Mark Kryder — Wikipedia, the free Encyclopedia, 2012. [Online; accessed 20-October-2012].