From Bases to Bits: An Analysis of DNA Compression Algorithms

Gavin Saxer '26
saxerg@carleton.edu
Carleton College
Northfield, MN, USA

Jared Arroyo Ruiz '26
arroyoruizj@carleton.edu
Carleton College
Northfield, MN, USA

Ryan Son '26
sonr@carleton.edu
Carleton College
Northfield, MN, USA

Abstract—**Today, with increased accessibility to genomic sequencing, there is a necessity to create specialized compression algorithms to more efficiently store such data. This paper seeks to compare the space savings achieved by entropy, non-referenece, and reference based compression algorithms (Huffman, Biocompress 1, and DNAzip, respectively). To do so, we implement and apply each algorithm with optimal parameters to human genomic sequences. In doing so, we identified DNAzip to have the smallest compression ratio (0.00318 $\pm$ 0.0002), followed by Biocompress 1 (0.225 $\pm$ 0.002), and lastly Huffman coding (0.239 $\pm$ 0.0002). Given that the superior efficiency of DNAzip comes with the caveat of requiring a reference genome, our analysis revealed that DNAzip was only more efficient when storing five or more genomes. Simultaneously, Biocompress 1 proved most efficient for genomic datasets containing fewer than five genomes.**

Index Terms—**Data compression, Huffman coding, Compression algorithms, DNA data storage, Reference based compression, Non-reference based compression**

I. Introduction

Since the beginning of the 21st century, the cost of sequencing genomic data has vastly decreased. Previously, sequencing a human genome cost an overwhelming $30 million [1]. Today, thanks to technological advancements, an entire genome can be sequenced for around $80—a 3,750,000-fold decrease in cost [2]. Due to the reduced financial burden of modern sequencing, financial resources no longer impede the field of genomic sequencing. Yet, the data created by ambitious sequencing projects, such as the 1000 genomes project, has caused what we can only describe as a "data flood" [3]. As such, the use of specialized compression algorithms to handle extensive amounts of genomic data has proved to be pivotal in continuing the development of personalized medicine.

The first known DNA-specialized compression algorithm is Biocompress 1, which was released in 1993 [4]. The algorithm utilized the presence of repeated sub-strings within DNA sequences to compress genomes. This allowed for greater space savings than regular text-based compression algorithms. Into the 21st century, scientists continued their attempts to compress the entire human genome. Though their approaches proved to be more successful each time, given the substantial size of the human genome (nearly 3.6 billion base pairs), the space savings provided were limited. A revolution in genomic data compression came in 2009 with Christley et al's DNAZip [5]. Rather than attempting to compress the full human genomic sequence, Christley's group exploited the similarities between genomes, only encoding the differences between the target genome and a reference genome, creating what we know today as reference-based compression algorithms. That is, given that humans share nearly 99.9% of their genetic makeup, Christley's group focused only on the variation across the remaining 0.1%, immensely shrinking the size of data needed to be stored.

Since 2009, the field of genomic data compression has seen significant advancements. Yet, given that the publication of DNAZip was one of the top 20 papers in translational bioinformatics at the time, we chose to produce an implementation of the algorithm. Given the historical importance of Biocompress 1, we have also chosen to investigate it due to its classification as a non-reference-based compression algorithm. Along with these two, we implemented Huffman coding, an entropy-based compression algorithm, as a baseline against the other two algorithms. In this paper, we analyze and evaluate three distinct approaches to genomic data compression: general text-based algorithms and specialized DNA compressors (reference and non-reference based). We begin by optimizing each algorithm across various metrics, such as runtime, compression ratios, and space savings. Given these optimized algorithms, we then compare each algorithm's effectiveness in maximizing space savings and if they can reasonably accommodate this current, generational influx of genomic data. By comparing all three algorithms, we hope to better understand the benefits of using DNA-specialized compression algorithms, and within the set of these algorithms, understand the benefits of reference-based compression algorithms for handling immense volumes of genomic data.

II. Methods

A. Data Preprocessing

Genomic data were sourced from several publicly available repositories including the National Center for Biotechnology Information (NCBI), and the University of California Santa Cruz (UCSC) Genome Browser [6] [7]. Genomic sequences GRCh38, Ash1_v2, PAN027_mat_v1.0, Han1, and T2T-CHM13v2.0 were retrieved from NCBI, allowing the construction of our own genomic library. These files are necessary for evaluating the compression performance of our three algorithms and the creation of variant call format (VCF) files containing the three types of variations in Fig. I for DNAzip. The common dbSNP build 155 was retrieved from the UCSC Genome Brower and served as an external reference set of common SNP variants required by DNAzip.

Reference assembly GRCh38 served as the primary reference genome for all alignments for DNAzip. The other four genome assemblies were selected as targets for alignment with GRCh38 and comparison between our algorithms. Aligning target genomes against the GRCh38 reference created VCF files by using a multi-step pipeline involving MUMmer4, all2vcf, and Bash scripts illustrated in Fig. 2 [8] [9]. Each genome was further split into multiple chromosomes. Compression performance analysis of each algorithm was also done on chromosome 21 of Ash1_v2, due to its minimal size and ease of testing.

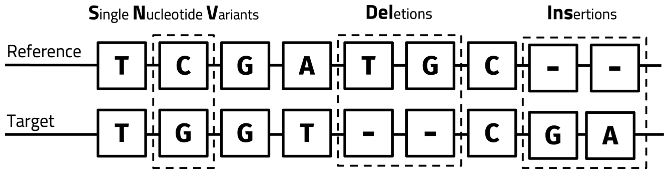

Fig. 1: Alignment of a target sequence (bottom) to a reference genome (top), highlighting three types of genetic variation: a single nucleotide variant (SNV), a deletion, and an insertion.

TABLE I: VCF file structure for each variation of each target genome when aligned against the reference. Each line contains one of three types of variations along with its chromosomal coordinates and the relevant reference/target nucleotide information.

Variation Header	Variation Info
SNV, Chromosome, Position	C/G
DEL, Chromosome, Position	TG/–
INS, Chromosome, Position	–/GA

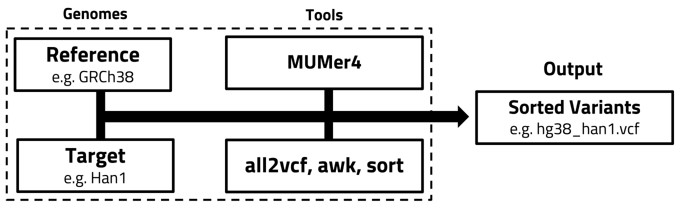

Fig. 2: Creation of the VCF utilized MUMmer4 (genome alignment tool), a seed-extend algorithm that finds reference/target nucleotide matches and then dynamically identifies the DNA variations and all2vcf, a VCF conversion tool.

B. Biocompress 1

Biocompress 1 exploits the limited bases that form DNA and its repetitive structure to compress genomic data more efficiently. Given that DNA is only composed of four nucleotide bases (Adenine, Cytosine, Guanine, and Thymine), Biocompress 1 represents each base with two unique bits (11, 10, 01, and 00, respectively) Such representations allow for greater space savings compared to ASCII encodings, which require eight bits.

$$\text{ASCII Encoding: } A \rightarrow 01000001, C \rightarrow 01000011, \quad T \rightarrow 01010100, G \rightarrow 01000111 \tag{1}$$

$$\text{2-bit Encoding: } A \rightarrow 11, C \rightarrow 10, T \rightarrow 01, G \rightarrow 00 \tag{2}$$

Along with utilizing 2-bit encodings, Biocompress 1 exploits the repetitions found within DNA. To do so, when the algorithm comes upon a sequence it has previously seen, rather than encoding each individual base again, the algorithm will encode a pointer to the sequence's previous appearance. Such repetitions can be broken down into two categories: factors and palindromes. A factor can be considered an identical copy of a sequence, while palindromes represent the complement of a sequence. For instance, given the sequence AACGTT, a factor would be represented by replicating the sequence, while a palindrome would be represented by the sequence TTGCAA.

These repetitions are encoded as (p, t, l), where p is the position to copy from, t is the type of repetition (factor or palindrome), and l is the length of the repetition. As an example, if the sequence $AACGGGAT$ had previously appeared at position 3, the encoding would be $(3, \text{factor}, 8)$. Notably, position values for palindromes are relative positions rather than absolute positions as palindromes tend to appear near each other in sequences.

All sequences of 2-bit encoding and repetition encoding are led by the number of each type. So the following sequence:

$$AACTGTTGTTGTTTTGAC \tag{3}$$

is encoded as:

$$6\,AACTGT\;2\,(4, \text{factor}, 7)(1, \text{palindrome}, 13) \tag{4}$$

This can be translated as 6 bases followed by 2 repetitions, the first is a factor of length 7 from position 4, and the second is a palindrome of length 5 from 13 positions prior.

It should be noted that position values are 1-indexed given that all integers are encoded using a Fibonacci encoding. While each bit in binary corresponds to powers of two (1, 2, 4, 8, etc.), in Fibonacci encoding, each bit corresponds to a Fibonacci number (1, 2, 3, 5, etc.). This prevents adjacent 1's, so an additional 1 is appended so the only adjacent 1's in the integer are at the end. Though, in the case where encoding positions values with a binary encoding is more efficient (a lower number of bits is used), Biocompress 1 will do so. To differentiate the two, the algorithm will append an additional 1 after the first adjacent 1's for binary encoding, while Fibonacci encoding will append an additional 10.

All repetitions are found using a 4-ary tree of height n created as the initial step in the algorithm. Each branch from a node corresponds to a base and positions are stored within the nodes. Non-leaf nodes store only a single position, while leaf nodes store every found position, as shown in Fig. 3. When a sequence of n bases is read in, the algorithm traces through the tree according to the sequence, adding in position values as needed. When it reaches a leaf node, if there is already a position value, it performs an extended search before adding the position to the node. The extended search directly compares bases sequentially between the current sequence and previous appearance for each position stored in the leaf.

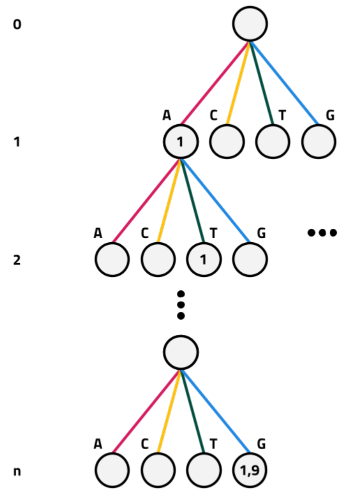

Fig. 3: Example tree of height n where the sequence AT...G was found at positions 1 and 9. Note that no other sequence positions have been added.

To find repetitions, for a base at position i, the algorithm first reads in a sequence from i to $i + n$, searches the tree for repetition, and returns the longest match found. It then takes the complement of that sequence, and repeats the search to find palindromes, returning the longest palindrome found. Given the two, it then compares whether it is most efficient to encode the factor, palindrome, or individual bases. Then increases i accordingly.

C. Huffman Coding

Introduced by a David A. Huffman in 1952, Huffman coding is a lossless entropy-based compression algorithm that assigns shorter bitstrings to symbols that occur more frequently within input data [10]. Entropy-based compression algorithms seek to reduce the amount information required to represent each symbol within a given source of data. Huffman coding does this by integrating the probabilities of a set of unique symbols within an input into a binary tree: the lower the probability that a symbol will be used, the longer the path is to that symbol from the root of the binary tree.

The algorithm first sorts the symbols by their respective frequencies. This frequency mapping is then used to construct the binary Huffman tree from the bottom up. Each symbol becomes a leaf node in the tree, and nodes with the lowest frequencies are combined step by step until a single root node remains. This ensures that each resultant encoding for each leaf is prefix-free, or does not contain the encoding of another symbol within the encoding of it. After the Huffman tree is built, the algorithm generates minimum-redundancy encodings by traversing the tree from the root to each leaf. The path taken, 0s for left branches and 1s for right branches, form the bitstring assigned to that symbol. Higher frequency symbols end up closer to the root, receiving shorter bitstrings, while infrequent symbols receive longer ones. This ensures that the most common symbols are represented with fewer bits, achieving efficient compression.

Encoding of the input data is completed by reading and matching each symbol to its respective Huffman encoding until the end of the input, appending each encoding to the final compressed file. This mapping was implemented by creating a Python dictionary, where each symbol was the key, and the assigned value was their respective Huffman encoding.

Decoding requires the Huffman tree and the compressed file. If the Huffman tree is no longer available, it can be reconstructed with the Huffman encoding dictionary that pertains to the compressed file. Decoding the compressed file is completed by transforming the bytes back into a bitstring The bitstring is then read bit-by-bit to traverse the Huffman tree until an appropriate leaf node is encountered. Once the leaf node is reached, the decoded symbol is concatenated to a decoded output string. The decoded output string is completed once there are no more bits to be read from the bitstring.

Currently, Huffman coding is used in many general-purpose compression algorithms, but there are a few nuances introduced by encoding genomic data that the general Huffman coding algorithm does not account for. With an alphabet size of four, DNA nucleotides (A, C, G, T) provide very little opportunity for Huffman coding to produce any additional space savings outside of encoding each individual base as two bits, as this is already the least amount of bits required to encode the nucleotides for any given frequency of each nucleotide.

TABLE II: Sample k-mers ($k = 8$) from the Huffman encoding map. Each distinct k-mer has a unique bitstring. Each bitstring is encoded into the compressed file when the matching k-mer has been found. Ranked by frequency from top to bottom. The most frequent k-mer has the shortest bitstring.

***k*-mer**	**Bitstring**	**Length**
CTACATAT	000000000000000	15
CCTAAGGT	11111111100001111	17
GCAATCGC	11111111111011110101	20
CGCGTACA	1111111011011100100000	22

To reveal more structure, the genome is divided into k-mers—substrings of length k. As k increases, the effective alphabet expands from 4 symbols to 4^k symbols, and the k-mer frequency distribution becomes increasingly uneven due to genomic factors like repeats and sequence motifs. This unevenness introduces statistical redundancy that is not visible at the single-nucleotide level.

By constructing a minimum-redundancy code over the observed *k*-mer frequencies, the compressor can assign shorter bitstrings to common *k*-mers and longer ones to rare *k*-mers, thereby reducing the average number of bits per encoded *k*-mer. This mirrors other general-purpose entropy-based compression algorithms for text, audio, and image data, where grouping symbols exposes correlations that single-symbol coding cannot. This should enable Huffman coding to achieve better overall compression.

D. DNAzip

As a reference-based compression algorithm, DNAzip only looks to encode variations on a given sequence, with respect to a reference sequence. For any two given DNA sequences, there exist three types of variations: Single Nucleotide Variants (SNVs), deletions, and insertions. SNVs are nucleotide mismatches at a single position, insertions represent the addition of one or more nucleotides, and deletions show the loss of one or more nucleotides [11]. Given that the variants differ from each other, DNAzip handles each of them uniquely. SNVs are stored as a position value and a two-bit encoding of its single nucleotide, deletions as a position value and a deletion length, and insertions as a position value and a sequence of nucleotides.

Position values, along with deletion lengths, are stored as variable integers (VINTs). Rather than assigning each position value, or deletion length, a set number of bytes, the use of VINTs allow for smaller integer values to be stored within only the necessary amount of bytes to represent such an integer. To do so, VINTs utilize a singular bit as an end-of-integer flag. This allows for 1 byte to represent integers 0-127, 2 bytes to represent integers 0-16383, etc. Given that storing smaller position values provides greater space savings, DNAzip applies the use of relative positions (DELTA), as opposed to absolute positions. That is, position values are stored as the difference between the current variant and the previous variant's position. In doing so, position values will always be less than or equal to the absolute positions.

DNAzip also utilizes the Single Nucleotide Polymorphism Database (dbSNP) to increase space savings. The dbSNP stores Single Nucleotide Polymorphisms (SNPs), which are SNVs found in more than 1% of the human population (CITATION). DNAzip begins by mapping each variant within the dbSNP as a 0 in a bit array or bitmap. If an SNV is found within both the given data and the database, it's respective bit in the array will be converted to a 1. Along with this, the variant referenced by the bit will be removed from the given data and only referenced through the dbSNP bit array. Although the Christley group utilizes the dbSNP, they caution that its application may yield lower space savings if the database's full potential is not realized. Any SNV's not mapped to the dbSNP are then encoded with their respective 2-bit representation.

While a majority of the variants can be accounted for by VINTs or two-bit representations, given the variation in insertion sizes, the Christley group chose to implement Huffman coding to further improve their compression ratio. Their implementation of Huffman coding involves breaking insertion sequences into k-mers of size 4, which are then used to create Huffman encodings. Any remaining nucleotides not represented within the k-mers are represented with their original two bit encoding.

III. Optimization

While the original implementations of our algorithms detail specific parameters to utilize, we found it important to trace the effects of such parameters on the overall space savings provided by each algorithm. In doing so, we intended to verify if the application of each respective algorithms' components still provided optimal results.

To perform such optimizations, we utilized the full genomic sequence and chromosome 21 of the genomic file Ash1_v2.2. In applying our algorithms to sequences of varying size, we also hoped to better understand the impacts of scale on our space savings.

A. Biocompress 1

When running Biocompress 1, for any tree height, the compression ratio will always be the same, but the runtime of the algorithm can shift dramatically. When creating the tree, the number of nodes created for a tree of height n is:

$$\text{number of nodes} = \sum_{i=0}^{n} 4^i \tag{5}$$

For trees where $n = 9$, there are around 3.5×10^6 nodes, but for trees where $n = 14$, there are 3.5×10^8 nodes. This creates runtime and memory considerations, as that many nodes take up significant amounts of memory.

As larger trees result in less extended searches for repetition, they are able to compress DNA sequences faster. This creates a tradeoff where larger trees take time to create but compress faster and smaller trees are created quickly but take longer to compress.

For shorter sequences, this tradeoff is significant, as larger trees can actually slow down the total runtime. For this implementation, the optimal tree height was found to be 10 for smaller sequences.

For longer sequences, the compression takes so long that a larger tree height was always justified, with the tree creation time only becoming significant at extremely large heights. Running trees with heights less than 11 for the full genome was unfeasible and took too long to justify running.

We were only able to run heights up to 15, as any larger would require over a terabyte of memory, which we did not have access to.

It is important to note that runtimes are highly dependent on specifics of the implementation and on the hardware used to run the algorithm. Runtimes are meaningful only in comparison to the same implementation run on the same hardware, and should only be understood in reference to each other.

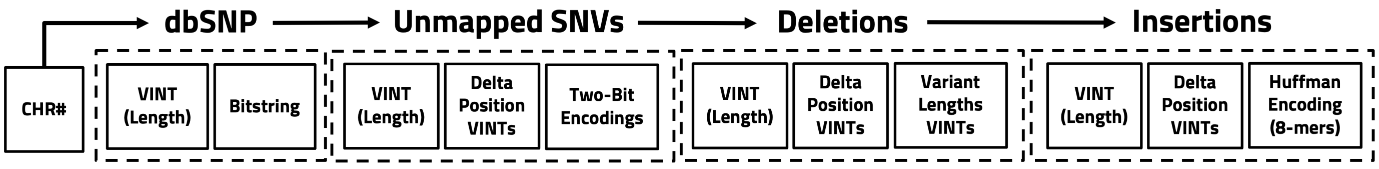

Fig. 4: Per chromosome construction of a DNAzip encoded bitstring for a compressed file containing each variant. Delta positions VINTs, variant length VINTs, and two-bit encodings are concatenated. The VINT (length) allows the decoder to intuitively know how many bits (positions & variant info) to iterate through.

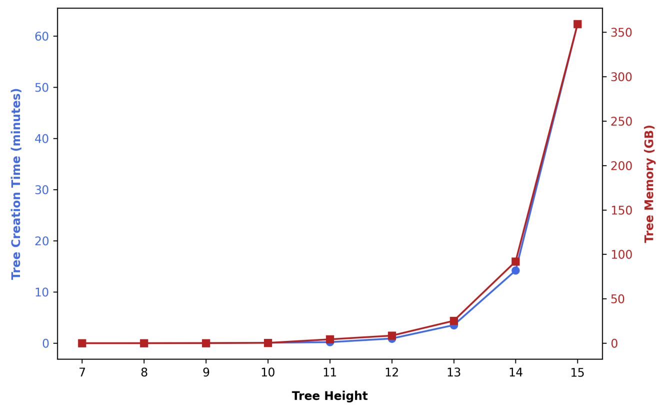

Fig. 5: The time to create the tree and the amount of memory required to store it for heights from 7 to 15.

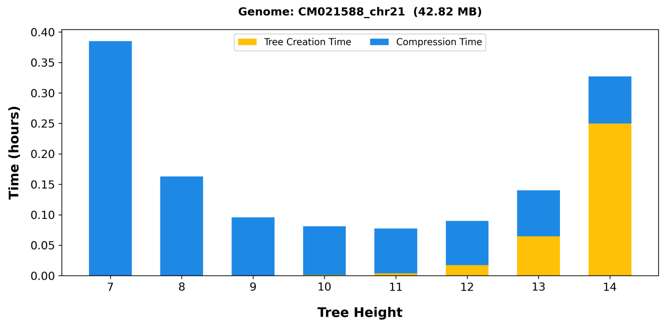

Fig. 6: Total runtime of Biocompress 1 divided into tree creation time and compression time for chromosome 21 with tree heights 7 to 14.

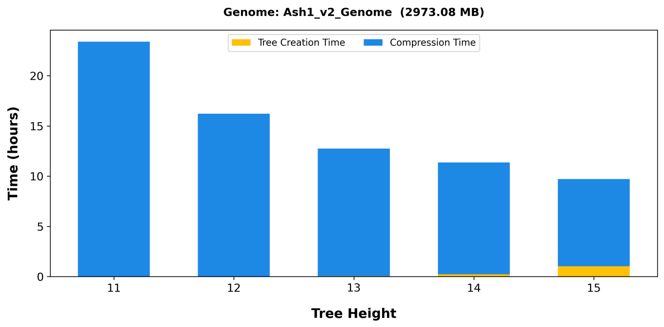

Fig. 7: Total runtime of Biocompress 1 divided into tree creation time and compression time for the full human genome with tree heights 11 to 15.

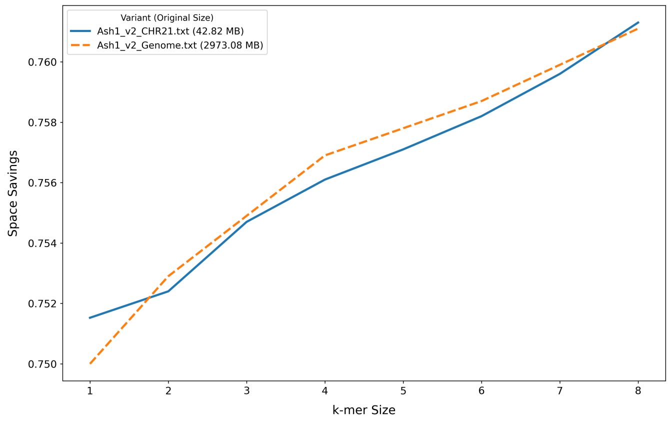

Fig. 8: Increasing k-mer sizes for Ash1_v2 genome (2973.08 MB) and chromosome 21 (42.82 MB) when Huffman coding resulted in higher space savings.

In summary, the algorithm uses a tree that gets so large that it results in significant time and memory usage. This results in different optimal heights for various DNA sequence lengths, as well as considerations on the amount of memory the algorithm has access to.

B. Huffman

Because genomic data consists of only four nucleotides, applying Huffman coding directly to single bases yields limited benefit, as each symbol is only assigned a two-bit code. We can expose additional redundancy within the genome by instead partitioning it into k-mers, and as k increases the effective alphabet grows to 4^k. This produces a far more uneven frequency distribution, allowing Huffman coding to assign shorter codewords to common k-mers and longer ones to rare k-mers. With larger k-mer sizes generally yielding better compression.

To quantify the impact of different k-mer lengths, we measured the space savings across chromosome 21 and the full Ash1_v2.2 genome, as shown in Fig. 8. Increasing the k-mer size produced a clear trend of improved space savings because larger k-mers increased the variation in symbol frequencies that Huffman coding can exploit. This optimization was then used for later Huffman coding compression and DNAzip encoding analyses.

C. DNAzip

In reviewing each component of DNAzip's original implementation (VINTs, DELTA, Huffman coding, and dbSNP), we hoped to measure the impact of each component on space savings. In addition to this, given the advisory of the Christley group, our objective was to provide concrete evidence if the use of the dbSNP still proved to be optimal given its expansion since 2009. To perform the analysis, we introduced each component in a sequential matter, with the application of VINTs as our foundation. We then introduced DELTA, Huffman coding of insertions, and finally the dbSNP.

When applying DNAzip to both chromosome 21 and the full genomic sequence of the Ash1_v2.2 genome, Fig. 9 illustrates that utilization of only VINTs, DELTA, and Huffman coding provided the most optimal results. More specifically, when applied to the chromosome 21 sequence, VINTs, DELTA, and Huffman coding we were able to achieve a compressed file size of 0.16MB, yet the introduction of the dbSNP increased the compressed file size (0.17MB). Applying these components to the full genomic sequence revealed similar trends. Figure 11 reveals a more distinct gap between VINTs and DELTA (8.97 MB), as to when Huffman coding is introduced (8.87MB). Finally, the addition of the dbSNP remained detrimental by increasing the compressed file size 10.06 MB.

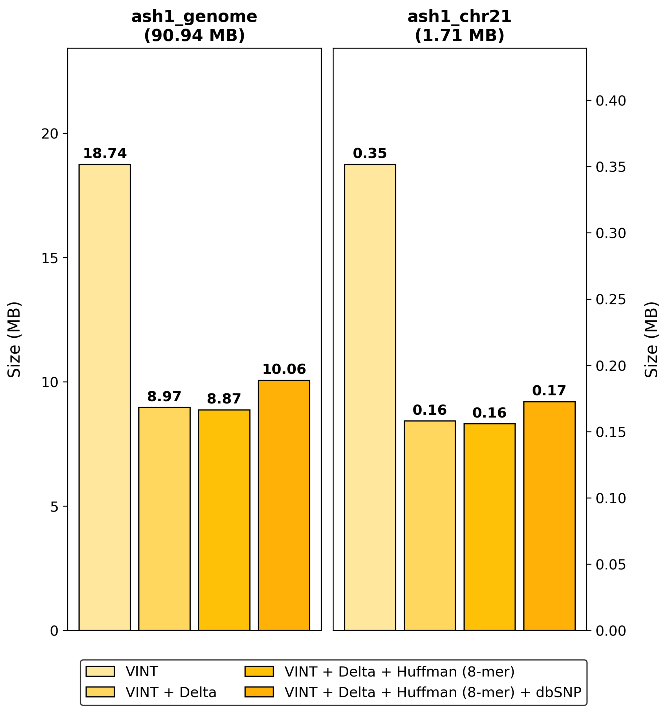

Fig. 9: Evaluation of end compression performance between Ash1_v2 genome VCF (90.94 MB) and chromosome 21 VCF (1.71 MB) with GRCh38 as the reference. Compressed file sizes follow each additional component of DNAzip, starting with VINT, then by DELTA, Huffman (8-mer), and dbSNP.

IV. Results

A. Biocompress 1

Biocompress 1 employs two methods to encode DNA sequences: 2-bit encoding and repetition encoding. The compression performance of the algorithm was analyzed by comparing how the 2-bit encoding compared to the full algorithm.

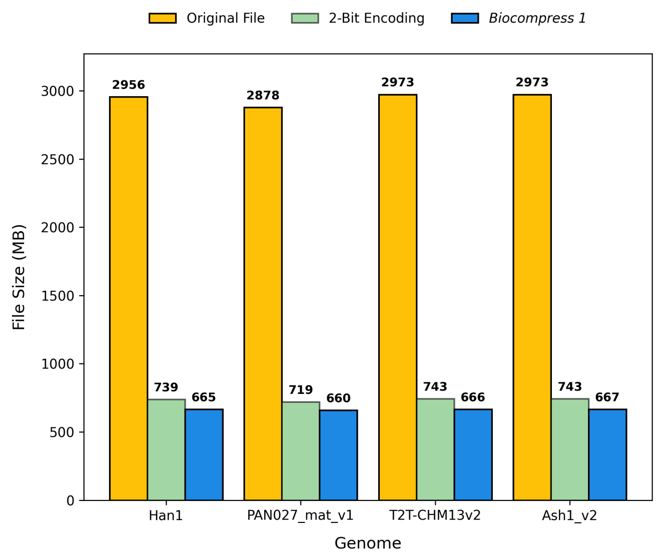

Fig. 10: Compression of four full human genomes with the 2-bit encoding and the full Biocompress 1 algorithm as compared to the original DNA file size.

TABLE III: Compression ratios and space savings for Biocompress 1 on human genomes as shown in Fig. 10

	Han1	PAN027_mat_v1	T2T-CHM13v2	Ash1_v2
Compression Ratio	0.225	0.229	0.224	0.224
Space Savings	0.775	0.771	0.776	0.776

B. DNAzip

DNAzip compression was analyzed by assessing the storage requirements of both the preprocessed variants in the VCF after target-reference alignment and the compressed files after compressing the each respective VCF with DNAzip, illustrated in Fig. 11. The compressed files produced by DNAzip are optimized by only applying the components that improved end compression (VINTs, DELTA, and Huffman coding). Each VCF had been preprocessed to only include the relevant information for each variant by following the format found in Table I.

C. Combined

For three full human genomes, Pan027_mat_v1, T2T-CHM13v2, and Han1, we compared the compression achieved by Biocompress 1, DNAzip, and Huffman coding. We also analyzed how the different algorithms perform when compressing multiple genomes, factoring in DNAzip's need for a full reference genome.

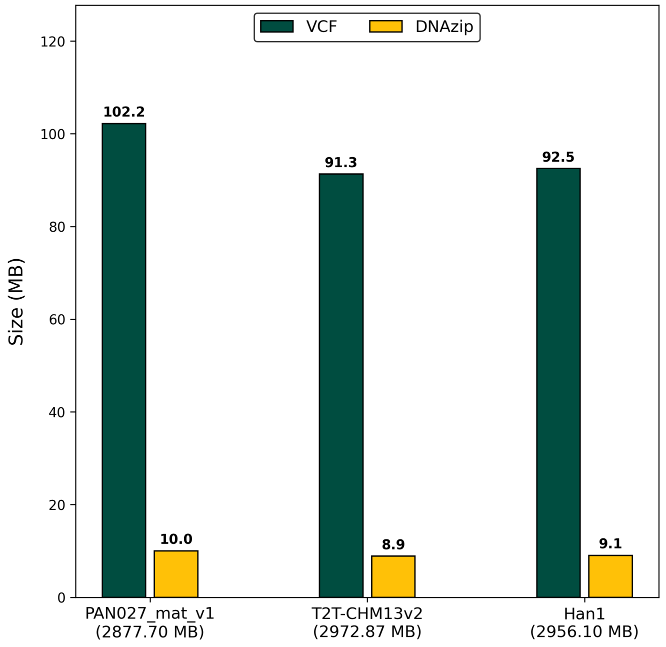

Fig. 11: Comparison of DNAzip end compression performance between three different VCFs and their respective compressed file with GRCh38 serving as the reference. Optimizations to end compression applied only VINTs, DELTA, and Huffman coding (8-mer).

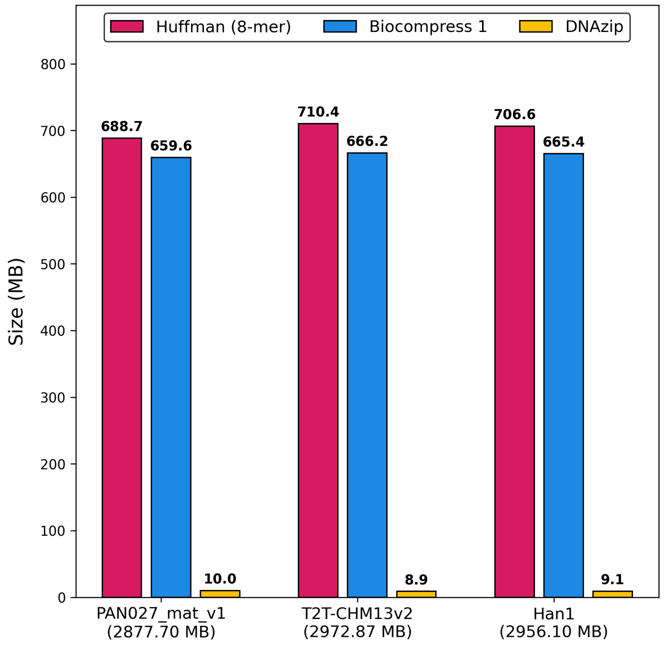

Fig. 12: Compression performance of three full human genomes by Huffman coding (8-mer), Biocompress 1, and DNAzip (8-mer) with GRCh38 serving as a reference for DNAzip.

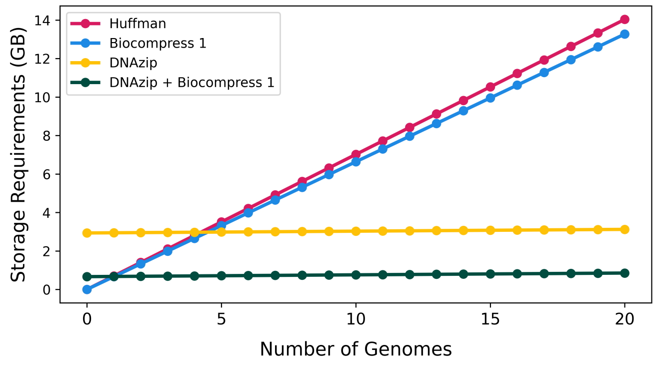

Fig. 13: Storage size required by increasing number of genomes for Huffman coding (8-mer), Biocompress 1, and DNAzip (8-mer). DNAzip has an initial 3 GB storage size cost due to the requirement of the reference genome GRCh38. The combination of DNAzip and Biocompress 1 compressing GRCh38 has an initial cost of 0.66 GB.

V. Analysis

A. Biocompress 1

Biocompress 1 was able to achieve a compression ratio of 0.225 ± 0.002 for the four full human genomes shown in Fig. 10, meaning more than four compressed genomes can be stored in the same amount of space as one uncompressed genome. All four genomes were of similar size, and so the algorithm produced similar encoded sizes, as is to be expected.

By comparing the file sizes after 2-bit encoding and after Biocompress 1 in Fig. 10, we find that 2-bit encoding was responsible for an average of 96.87% of the compression of the algorithm. The repetition encoding was only responsible an average further 9.8% compression beyond what 2-bit encoding achieved.

Variations in the compression ratios are likely due to variations in the length and number of repetitions found in the DNA sequence. However, the compression ratios are all within 0.005 if each other, showing that these variations are minor.

Given that the vast majority of the algorithm's runtime is from tree creation or repetition searching, the viability of Biocompress 1 is questionable. If time or memory usage are constrained while compressing DNA, Biocompress 1 is likely not the optimal choice, as the significant runtime and memory usage account for only a small increase in space savings.

B. DNAzip

Fig. 11 illustrates the substantial decrease in storage size by compressing the VCF using DNAzip. By employing VINTs, DELTA, and Huffman coding of the insertions, the genomes saw a 90% decrease in storage size. Comparison of compressed file sizes shows that DNAzip extends the VCF's already great space savings. When compressing genome Pan027_mat_v1, the VCF is able to achieve a file size of 102.2 MB, DNAzip then achieves a compressed file size of 10.0 MB. Similarly, when applied to genomes T2T-CHM13v2 and Han1, DNAzip is able to achieve compressed file sizes of 8.9 and 9.1 MB.

Aligning the target and reference genomes created VCF files that achieved an average size ratio of 0.0325 ± 0.002 across the three genomes. The compressed file for each VCF with DNAzip achieved an average compression ratio of 0.098 ± 0.0003 and from the original genomes, an average compression ratio of 0.00318 ± 0.0002.

Fig. 11 only shows compression for genomes without taking into account the storage requirements of the reference genome. In real-world scenarios, the algorithm also requires a reference genome to be stored uncompressed and available. So in the case where only a single target genome and reference are stored, an average compression ratio of 0.515 ± 0.0034 is achieved across each genome in Fig. 11. Overall, the single genome performance of DNAzip is subpar, it is only when multiple genomes are stored does it start to show its value in storage space efficiency.

C. Combined

In comparing all three algorithms in Fig. 12, it is clear that DNAzip significantly outperforms the others. In comparing the final compressed file size of each algorithm across the PAN027_mat_v1 genomic file, with an original file size of 2877.70 MB, while Huffman and Biocompress 1 achieve similar results (688.7 and 659.6 MB, respectively), a drastic increase is seen in DNAzip's compression (10.0 MB). These results were mirrored when applied to genomic files Pan027_mat_v1 and Han1, with DNAzip achieving compressed file sizes of 8.9 and 9.1 MB, respectively.

While it becomes evident that the space savings provided by DNAzip are superior to those provided by Huffman and Biocompress 1, without a readily accessible reference, DNAzip cannot be effective. As such, when compressing with DNAzip, one must take into account the cost of storing the human reference genome (approx. 3 GB). As seen in Fig. 13, it is shown that if storing less than 5 genomes, the cost of storing the reference is too taxing. Once this threshold is surpassed, DNAzip's compression capabilities offset this initial cost and compound with each additional genome, thereby reducing the effective total storage per genome.

To further reduce the storage necessary to store the reference genome, we explored how the combination of DNAZip and Biocompress 1 would alter this effect. To do so, we compressed the reference genome with Biocompress 1 and encoded (any next) genomes with DNAzip. Such analysis revealed...

VI. Conclusion

Overall, reference-compression algorithms like DNAzip represent a crucial advancement toward scalable, cost-effective genomic data management in the era of high-throughput sequencing. While DNAzip offers superior space savings compared to its non-reference-based counterparts like *Biocompress* 1, it also introduces additional computational complexity by requiring target genome alignment, preprocessing of the variation data, and storage of reference data.

This paper discuses only the initial algorithms for the two types of DNA compression. For a more complete understanding of DNA compression, further analysis of DNA compression algorithms could be done. Implementation and comparison of more modern compression algorithms would enable a deeper understanding of the techniques used by compression algorithms to more efficiently compress DNA sequences. A modern reference based algorithm such as ones that utilize neural networks [11] and a non-reference based algorithm such as GReEn [12] would be useful to analyze and compare.

References

[1] S. Deorowicz and S. Grabowski, "Data compression for sequencing data," Algorithms Mol. Biol., vol. 8, no. 1, pp. 1–13, Jan. 2013, doi: 10.1186/1748-7188-8-25.

[2] F. L. Genomics and L. Fletcher, "The $100 Genome: Where's the Limit?," Front Line Genomics. Accessed: Oct. 05, 2025. [Online]. Available: https://frontlinegenomics.com/the-100-genome-wheres-the-limit/

[3] D. Keiger, "The DNA data flood," The Hub, Fall 2013. Accessed: Sep. 29, 2025. [Online]. Available: https://hub.jhu.edu/magazine/2013/fall/dna-data-flood/

[4] S. Grumbach and F. Tahi, "Compression of DNA sequences," in Proc. Data Compression Conf., 1993, pp. 340–350, doi: 10.1109/dcc.1993.253115.

[5] S. Christley, Y. Lu, C. Li, and X. Xie, "Human genomes as email attachments," Bioinformatics, vol. 25, no. 2, pp. 274–275, Jan. 2009, doi: 10.1093/bioinformatics/btn582.

[6] National Center for Biotechnology Information, https://www.ncbi.nlm.nih.gov/ (accessed Sep. 30, 2025).

[7] "Index of /goldenPath/hg38/chromosomes," Ucsc.edu, 2025. https://hgdownload.cse.ucsc.edu/goldenPath/hg38/chromosomes/ (accessed Oct. 5, 2025).

[8] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, "MUMmer4: A fast and versatile genome alignment system," PLoS Comput Biol, vol. 14, no. 1, p. e1005944, Jan. 2018, doi: 10.1371/journal.pcbi.1005944.

[9] M. Schiavinato, "Matteoschiavinato/all2vcf: Toolkit to convert the output of common variant calling programs to VCF," GitHub, https://github.com/MatteoSchiavinato/all2vcf (accessed Oct. 9, 2025).

[10] D. A. Huffman, "A method for the construction of minimum-redundancy codes," Reson, vol. 11, no. 2, pp. 91–99, Feb. 2006, doi: 10.1007/BF02837279.

[11] S. K. Sreeram and M. S. Nair, "GenCoder: A novel convolutional neural network based autoencoder for genomic sequence data compression," IEEE/ACM Trans. Comput. Biol. Bioinformatics, vol. 21, no. 3, pp. 405–415, Feb. 2024, doi: 10.1109/TCBB.2024.3366240.

[12] A. J. Pinho, D. Pratas, and S. P. Garcia, "GReEn: a tool for efficient compression of genome resequencing data," Nucleic Acids Res, vol. 40, no. 4, p. e27, Feb. 2012, doi: 10.1093/nar/gkr1124.

VII. Supplemental

For preliminary testing of the DNAzip implementation, initial variant analysis was done using high-confidence VCF files for HG002/HG003/HG004 [giab cit.]. These VCF files were then parsed using Bcftools, which enabled extraction of the variant records into an intermediate, human-readable text format. From these files, we retained only the fields required by DNAzip, including the chromosome, variant position, and both reference and alternate alleles. Following extraction, the text-processing utility Awk was used to systematically iterate through each record and restructure the fields into the format expected by DNAzip. Ensuring that all variant entries were consistently ordered, normalized, and suitable for compression for our DNAzip pipeline.

Breaking each genome assembly into per-chromosome FASTA files allowed for more efficient alignment and preprocessing through job parallelization and reduced memory load when later aligning each genome. Instances of the nucleotide placeholder **N**, were also cleaned from each genome.

The alignment pipeline begins with nucmer (MUMmer4), which performs whole-genome, nucleotide-level alignment to identify matching blocks of nucleotides and variations from Fig. 1 between the reference and target assemblies. The resulting alignment files were filtered and refined using delta-filter (MUMmer4) to remove low-confidence or redundant alignment segments.

These alignment files were then further converted into variant-level differences using all2vcf, which translates them into VCF entries. This process effectively isolates all sequence differences between the reference and target genome.

The VCF outputs were then processed with custom Bash scripts to enforce consistent ordering, normalize variant representation, and conform the same variant-formatting as the original DNAzip implementation. Through this alignment of genomic differences, the initial 3 GB target genome was reduced to a 100 MB variant file. The final variant files follow the structure shown in Table I and serve as the foundational input enabling DNAzip's reference-based compression workflow.

To construct a dbSNP dataset compatible with DNAzip, we obtained the CommonSNPs155 track from the UCSC Genome Browser in bigBed (.bb) format. Using UCSC's bigBedToBed utility, this track was converted into standard BED files, which were then separated into per-chromosome variant lists to match the input requirements of our pipeline. Subsequent cleaning steps included removing non-standard records, ensuring position consistency with GRCh38, and reformatting fields to mimic the dbSNP representation expected by DNAzip. This normalized and genome-matched dbSNP set ensured that variant lookup, encoding, and compression steps within DNAzip were accurate.

	GENOMIC FILES			
	Ash1_v2.2	T2T-CHM13v2.0	PAN027_mat_v1.0	Han1
Sample Details				
Sex	Male	Male	Female	Male
Tissue	B-Lymphocyte	N/A	N/A	N/A
Assembly				
Technology	Illumina HiSeq; Oxford Nanopore; PacBio	N/A	PacBio; Oxford Nanopore	PacBio Sequel; Oxford Nanopore PromethION
Method	MaSuRCA v4.0.7	N/A	Verkko v2	Hifiasm v0.16.1-r375
Contig N50	150.2 Mb	150.6 Mb	135.1 Mb	148 Mb
Contig L50	9	9	9	9
GC Percent	40.5	41	41	41
Coverage	123x	30x	70x	66x