From Bases to Bits: An Analysis of Early DNA Compression Algorithms

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

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

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

***Abstract*—Today, with increased accessibility to genomic sequencing, and the corresponding increase in genomic data, 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-reference, and reference-based compression algorithms (Huffman, Biocompress 1, and DNAzip, respectively). We implemented and applied each algorithm with optimal parameters to three human genomic sequences. In doing so, we identified DNAzip to have the smallest compression ratio across the three genomes (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.**

***Access*—Code and supplementary materials are available at https://github.com/Rawleo/genezippers_comps**

I. Introduction

Since the beginning of the 21st century, the cost of sequencing genomic data has vastly decreased. In 2004, sequencing a human genome cost a staggering \$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, economic resources no longer impede the field of genomic sequencing. Thus there have been multiple ambitious sequencing projects, such as the 1000 genomes project (a catalog of common human genetic variation), which has caused what we can only describe as a “data flood” [3] [4].

Such large scales require specialized compression algorithms to efficiently handle this data, many of which make use of the structural complexity of DNA. DNA is composed of four nucleotide bases (Adenine, Cytosine, Guanine, and Thymine), and therefore can be represented as a string over a four-letter alphabet (A, C, T, G). These sequences are not random, but rather filled with features such as repeats, motifs, and periodicities [5]–[7]. As a result, human genomes can be viewed as three-billion character sequences with complex internal structures.

The first known DNA-specialized compression algorithm is Biocompress 1, which was released in 1993 [8]. The algorithm utilizes the presence of repeated substrings within DNA sequences to compress genomes, which allowed for greater space savings than generic 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 bounded [9] [10]. A breakthrough in genomic data compression came in 2009 with Christley et al’s DNAzip [11]. Given that humans share nearly 99.5-99.8% of their genetic makeup, Christley’s group focused only on the variation across the remaining 0.2-0.5%, immensely shrinking the size of data needed to be stored [12]. To do so, the Christley group encoded differences between a target genome and a reference genome, a single representative genomic sequence for a given species, creating what we know today as reference-based compression algorithms [13].

Since 2009, the field of genomic data compression has seen significant advancements [14] [15]. 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. We have also chosen to investigate Biocompress 1 due to its historical importance and classification as a non-reference-based compression algorithm, compressing genomic sequences using only the information contained within the sequence itself rather than relying on additional genomic context. Along with these two, we implemented Huffman coding, an entropy-based compression algorithm, as a baseline text compression algorithm for the proposed algorithms.

In this paper, we analyze and evaluate three distinct approaches to genomic data compression: a general text-based algorithm and two DNA-specialized compressors (reference and non-reference-based). We begin by optimizing each algorithm across various metrics, such as runtime, compression ratios, and memory usage. With these optimized algorithms, we then compare each algorithm's effectiveness in maximizing space savings and viability of storing large quantities of genomic data. By comparing all three algorithms, we hope to better understand the benefits of using DNA-specialized compression algorithms, and the specific use cases for each.

II. Methods

A. *Biocompress 1*

Biocompress 1 exploits the limited bases that form DNA and its repetitive structure to compress genomic data more efficiently [8]. While the traditional ASCII encoding utilizes eight bits to represent each base, Biocompress 1 makes use of DNA's limited alphabet, encoding each base with two bits as shown in Table I.

TABLE I: ASCII and 2-bit encodings for A, C, G, T

Base	ASCII Encoding	2-Bit Encoding
A	`01000001`	`00`
C	`01000011`	`01`
G	`01000111`	`10`
T	`01010100`	`11`

Along with utilizing two-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 encodes the position and length of the sequence's previous appearance. Such repetitions can be broken down into two categories: factors and palindromes. A factor is an identical copy of a sequence, while palindromes are the reverse complement of a sequence. The reverse complement is obtained by mapping each A to T and each each C to G (and vice versa), followed by reversing the resulting sequence For instance, given the sequence $AACGTA$, the factor would be:

$$AACGTA \tag{1}$$

and the palindrome would be:

$$TACGTT. \tag{2}$$

These repetitions are encoded as a pointer in the form (p, t, l), where p is the position to copy from, t is the type of repetition, 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 $(3, \textit{factor}, 8)$. Palindromes tend to appear near each other, so unlike factors which encode an absolute position, palindromes encode a relative position, the distance from the current sequence to the palindrome. For example, if the sequence $AACGGGAT$ had appeared 5 positions prior, the encoding would be $(5, \textit{palindrome}, 8)$.

To aid in decoding, all encoded sequences of bases or repetitions are prefixed by the quantity that appears. So the following sequence:

$$AACTGTTGTTGTTCAGTT \tag{3}$$

can be divided as such:

$$AACTGT \mid TGTTGTT \mid CAGTT \tag{4}$$

and encoded as:

$$6\,AACTGT \;\; 2\,(4, \textit{factor}, 7)(13, \textit{palindrome}, 5). \tag{5}$$

The 6 and the 2 indicate the number of bases and repetitions to expect, so this can be understood as 6 bases followed by 2 repetitions. The first repetition 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 because most integers are encoded using a Fibonacci encoding, which is unable to represent 0. While each bit from right to left in binary corresponds to powers of two $(1, 2, 4, 8, ...)$, in Fibonacci encoding, each bit from left to right corresponds to a Fibonacci number $(1, 2, 3, 5, ...)$. To indicate the end of an integer, a `1` is appended to the end, to create `11`, which flags the integer as finished, as shown in bold below.

$$\text{Fibonacci Encoding: } 17 \rightarrow \texttt{101001}\mathbf{\texttt{1}}. \tag{6}$$

$$\text{Binary Encoding: } 17 \rightarrow \texttt{10001}. \tag{7}$$

While most integers are encoded with Fibonacci encoding, in the case where encoding positions values with binary encoding uses fewer bits, Biocompress 1 will do so. To differentiate the two, the algorithm will append an additional 1 after the first adjacent `1`'s in the binary encoding, while Fibonacci encoding will append an additional `10` at the end, both shown in bold below.

$$\text{Fibonacci Position Encoding: } 13 \rightarrow \texttt{000001}\mathbf{\texttt{10}}. \tag{8}$$

$$\text{Binary Position Encoding: } 13 \rightarrow \texttt{11}\mathbf{\texttt{1}}\texttt{01}. \tag{9}$$

To find repetitions, Biocompress 1 initializes a 4-ary tree, a tree where each node has four branches, of which the height, n, is a manually set, predetermined parameter. 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. 1.

When a sequence of n bases is read, the algorithm traces through the the tree accordingly, storing position values as needed. When a leaf node is reached, in the case that the leaf node already stores a position, an extended search is then performed. The extended search directly compares bases sequentially between the current sequence and previous appearance for each position stored in the leaf.

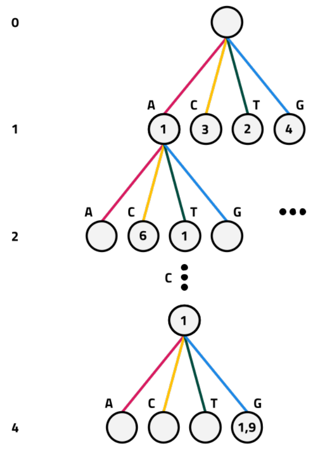


Fig. 1: Example tree of height $n = 4$. Position values have been filled in for sequence: $ATCGTACTATCG$. Note that the subsequence $ATCG$ appears at positions 1 and 9.

To find repetitions, for a base at position i, the algorithm reads in a sequence from i to $i + n$. To find the longest factor, it searches the tree for repetition, and returns the longest match found. It then repeats the search with the reverse complement of the sequence 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 the sequence as individual bases.

B. 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 [16]. Entropy-based compression algorithms seek to reduce the amount of information required to represent each symbol within a given source of data.

The algorithm first sorts the symbols by their respective frequencies. This frequency mapping is then used to construct the binary Huffman tree as a set of nodes from the bottom up, as showin in Fig. 2a. Each symbol becomes a leaf node in the tree, and nodes with the lowest frequencies are combined sequentially until a single root node remains. Once the Huffman tree is built, the algorithm generates minimum-redundancy encodings by traversing the tree from the root to each leaf. The path, where left and right branches are denoted by 0 and 1 respectively, defines the bitstring for that symbol, as depicted in Fig. 2a. This guarantees that the encoding assigned to each symbol is unique and never appears as a prefix of another symbol's encoding. Higher frequency symbols end up closer to the root, receiving shorter bitstrings, while infrequent symbols receive longer ones as shown in Fig.2b. This ensures that the most common symbols are represented with fewer bits, achieving efficient compression.

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 bases provide very little opportunity for Huffman coding to produce any additional space savings outside of encoding each individual base as two bits seen in Table I, as this is already the least amount of bits required to encode the bases for any given frequency of each base.

To address this limitation, the genome is divided into k-mers, substrings of length k. As k increases, the set of possible symbols expands to 4^k, and the k-mer frequency distribution becomes increasingly uneven due to genomic characteristics like repetition. This unevenness introduces statistical redundancy that is not visible when $k = 1$. If the genome is not evenly divisible by k, the remaining bases are encoded using their respective two-bit encoding shown in Table I.

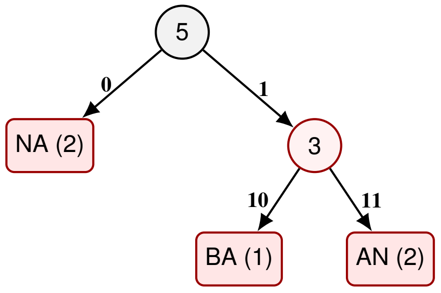
As shown in Fig. 2c, increasing k allows us to reduce the number of unique symbols, and by constructing a Huffman tree based on the frequency of these pairs, we assign shorter bitstrings to more frequent combinations, reducing the storage size.

The compressor generates a minimum-redundancy code based on the observed k-mer frequencies, assigning shorter bitstrings to frequently occurring k-mers and longer bitstrings to those that are less common. This reduces the average number of bits per encoded k-mer.

C. DNAzip

As a reference-based compression algorithm, DNAzip only looks to encode variations on a target sequence, with respect to a reference sequence [11]. For any two given DNA sequences, there exist three types of variation: Single Nucleotide Variants (SNVs), deletions, and insertions. As Fig. 3 illustrates, SNVs are base mismatches at a single position, insertions represent the addition of one or more bases, and deletions denote the removal of one or more bases [17]. Fig. 5 depicts how DNAzip uniquely handles each of the variant types. SNVs are stored as a position value and a two-bit encoding of its single base, deletions as a position value and a deletion length, and insertions as a position value and a sequence of bases.

(a) **Huffman Tree** ($k = 2$)



(b) **Huffman Encoding Map**

Encoding	**Symbol**	**Freq**	**Bit Code**
***k*-mer** ($k = 1$)	A	3	`0`
	N	2	`11`
	B	1	`10`
***k*-mer** ($k = 2$)	NA	2	`0`
	AN	2	`11`
	BA	1	`10`

(c) **Encoding Stream Visualization**

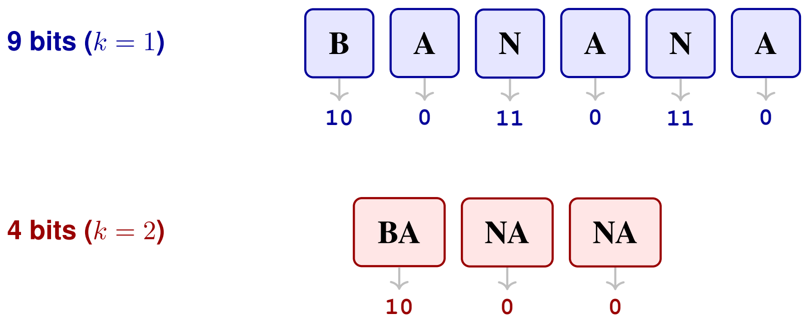


Fig. 2: Comparative analysis of *k*-mer Huffman coding for the sequence "BANANA." (a) The Huffman tree constructed for *k*-mers, ($k = 2$). Each node represents the sum of the frequencies of its children. (b) A lookup table contrasting symbol frequencies and assigned binary codes for $k = 1$ vs. $k = 2$ encodings. (c) Visualization of the encoding stream, demonstrating the compression improvement from 9 bits using $k = 1$ to 4 bits using $k = 2$.

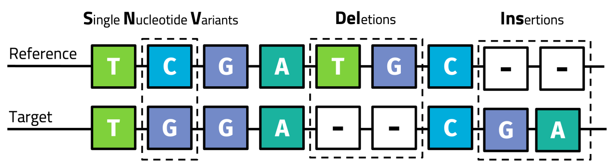


Fig. 3: Alignment of a target sequence (bottom) to a reference genome (top), highlighting the three types of genetic variation: a single nucleotide variant (SNV), a deletion, and an insertion.

Within the algorithm, any integer present is encoded as a VINT (variable-length integer) as seen in Fig. 5. Rather than assigning each integer a set number of bytes, the use of VINTs allows for integer values to be stored within only the necessary amount of bytes to represent such an integer as shown in Fig. 4. To do so, VINTs utilize a singular bit as an end-of-integer flag. This allows for n bytes to encode up to 2^{7n} integers. Such that one byte can encode values 0-127, two bytes can encode 128-16383, 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 generally occupy considerably less space than their absolute positions.

DNAzip also utilizes the Single Nucleotide Polymorphism Database (dbSNP) to increase space savings. The dbSNP stores Single Nucleotide Polymorphisms (SNPs), which are common SNVs found in more than 1% of the human population [18]. Although the Christley group makes use of dbSNP, they caution that its effectiveness may be limited. This occurs because the full dbSNP must be stored regardless of how closely it mirrors the target genome. When the SNVs in the target genome fail to match those cataloged in the dbSNP, the resulting space savings become significantly reduced. Any SNVs not mapped to the dbSNP are then encoded with their respective two-bit representation as shown in Fig. 5.

The Christley group also chose to implement Huffman coding of the insertion sequences with *k*-mers to further improve end compression.

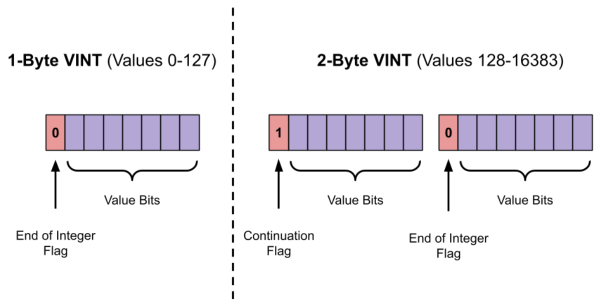


Fig. 4: Example of variable-length integer (VINT) encoding. Each byte uses its leading bit as an end-of-integer flag, while the remaining seven bits store integer information.

III. Results

A. Data Preprocessing

Genomic sequences GRCh38, PAN027_mat_v1.0, Ash1_v2.2, Han1, and T2T-CHM13v2.0 were retrieved from the National Center for Biotechnology Information (NCBI), allowing the construction of our own genomic library, for additional information about each genome, see supplemental table S1 [19]. The common dbSNP build 155 was retrieved from the University of California Santa Cruz (UCSC) Genome Browser and served as an external reference set of common SNP variants required by DNAzip, see supporting information in Appendix A [20]. In the sequences, N represents an ambiguous base indicating that the base at that position is uncertain. These ambiguous bases were cleaned from sequences to allow for compression by Biocompress 1 and Huffman coding.

Reference assembly GRCh38 served as the primary reference genome for all alignments for DNAzip. Target genomes were aligned against the GRCh38 reference by using a multi-step pipeline involving MUMmer4, all2vcf, and Bash scripts illustrated in Fig. 6, creating variant call format (VCF) files; see supporting information in Appendix B [21] [22]. The other four genome assemblies were selected as targets for alignment with GRCh38 and comparison between our algorithms.

B. Optimization

While the original implementations of our algorithms detail specific parameters to utilize, we found it important to trace how these parameters affect memory usage, runtime, and overall space savings provided by each algorithm. In doing so, we tuned each algorithms' components to provide the most optimal results.

To carry out these optimizations, we utilized our aligned VCF, full genomic sequence, and chromosome 21 from Ash1_v2.2. We selected Ash1_v2.2 as it is the assembly of the HG002 genome, one of the target genomes initially tested in Appendix B, sourced from the Genome in a Bottle (GIAB) consortium [23]. By applying our algorithms to both the full genomic sequence and chromosome 21, we aimed to gain deeper insights into both memory usage and runtime.

1) Biocompress 1: When running Biocompress 1, the 4-ary tree height is irrelevant to the compression ratio achieved, as any repetition will be encoded as the longest possible match. However, the height does impact both the runtime and the memory usage of the algorithm significantly. When creating the 4-ary tree, the number of nodes created for a tree of height n is:

$$\text{number of nodes} = \sum_{i=0}^{n} 4^i. \tag{10}$$

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. While each node requires a limited amount of memory, at such scales the total footprint becomes substantial, creating significant runtime and memory considerations. Fig. 7 highlights the relationship between increasing tree heights and the exponential increases in tree creation time and memory required. As each subsequent node created requires a fixed amount of memory and time to create, the increases in tree creation time and tree memory are proportional.

While larger trees take longer to create, the increased number of leaf nodes results in fewer positions being stored in each leaf. This results in fewer extended searches to find the longest repetition and reduces the amount of time required for compression. This creates a tradeoff: larger trees take longer to build but compress faster, whereas smaller trees are built quickly but compress more slowly.

In compressing chromosome 21 of Ash_v2.2, this tradeoff is significant, as larger trees can actually slow down the total runtime. For our implementation, Fig. 8 shows that the optimal tree height was found to be 11 for smaller sequences, resulting in a total runtime of just under five minutes. At larger heights, the tree creation time outweighed the reduced compression time, climbing up to almost 20 minutes for a height of 14.

For longer sequences, the compression takes so long in our implementation that a larger tree height was always justified, with the tree creation time only becoming relevant at large heights. Fig. 9 shows the compression time decreasing for each subsequent tree height, up to

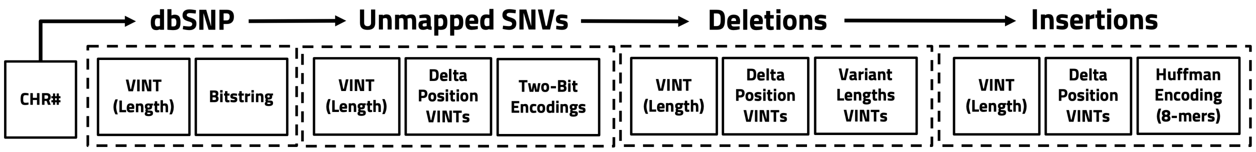


Fig. 5: Per chromosome construction of a DNAzip encoded bitstring for a compressed file containing each variant. Delta positions VINTs, variant length VINTs, and two-bit encodings are concatenated.

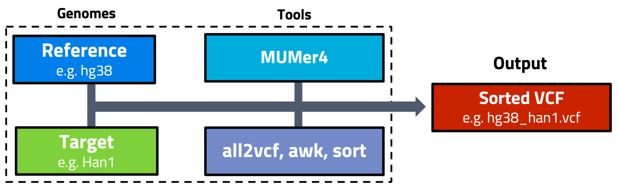


Fig. 6: Creation of the variant call format (VCF) utilized MUMmer4 (genome alignment tool), a seed-extend algorithm that finds reference/target base matches and then dynamically identifies the DNA variations and all2vcf, a VCF conversion tool. Command-line tools AWK and SORT were used to further process the VCF.

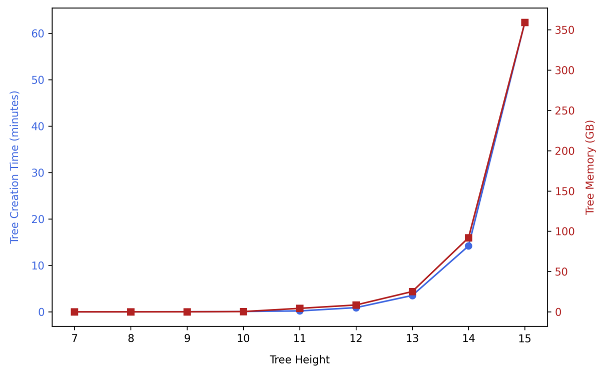


Fig. 7: The time to create the tree and the amount of memory required to store it for 4-ary tree heights from 7 to 15.

a height of 15. Tree heights larger than 15 required enormous amounts of memory, over a terabyte, and therefore could not be tested. Running trees with heights less than 11 for the full genome took a significant amount of time and did not result in useful data.

We found that there are different optimal heights for various DNA sequence lengths, with shorter sequences working optimally with smaller 4-ary trees and longer sequences working optimally with larger 4-ary trees. A height $n = 15$ used all available memory, and so in order to run Biocompress 1 on multiple full human genomes, a height $n = 14$ was selected as it provided the next fastest runtimes.

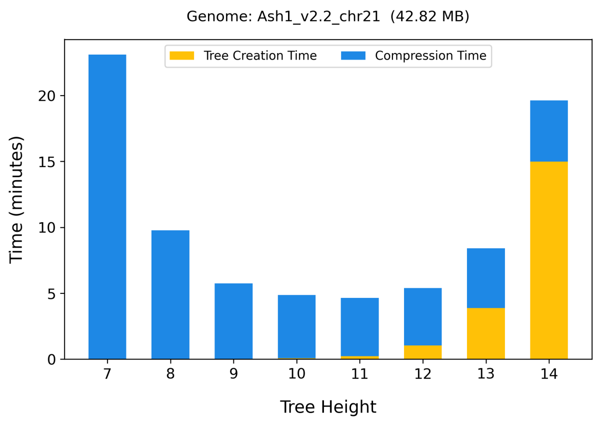


Fig. 8: Total runtime of Biocompress 1 on chromosome 21 (42.82 MB) with tree heights 7 to 14, divided into tree creation (yellow) and compression time (blue).

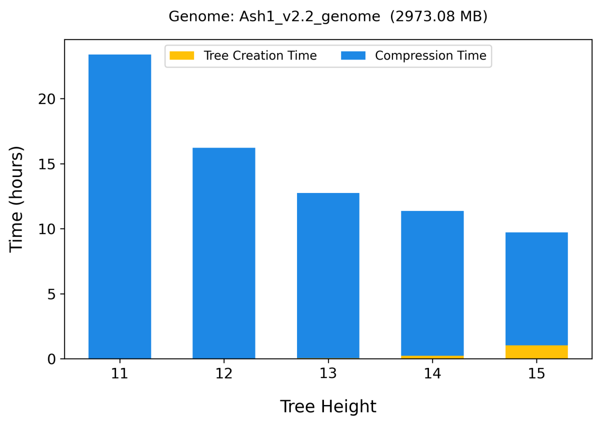


Fig. 9: Total runtime of Biocompress 1 on Ash1_v2.2 genome (2973.08 MB) with tree heights 11 to 15, divided into tree creation (yellow) and compression time (blue).

2) Huffman: Since applying Huffman coding directly to individual bases offers limited benefit, we investigated the potential advantages of using *k*-mers. To assess the effect of different *k*-mer lengths, we measured the resulting space savings on chromosome 21 and the complete Ash1_v2.2 genome, as illustrated in Fig. 10. Increasing the *k*-mer size showed a clear trend of im-

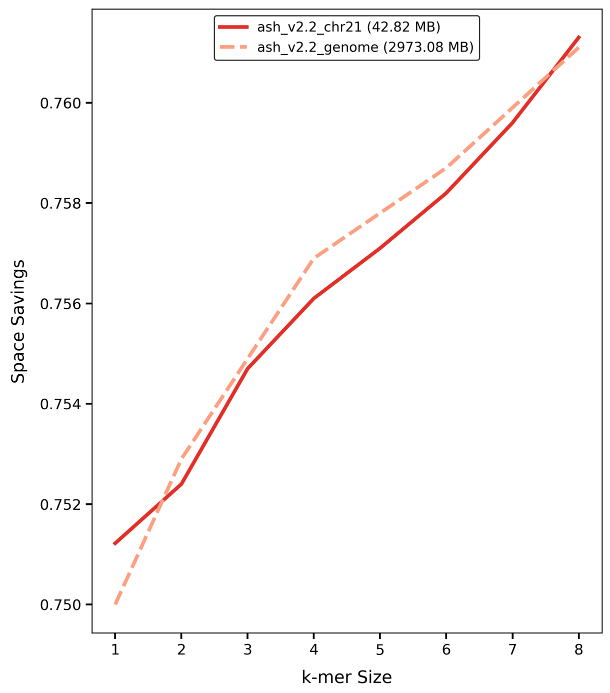


Fig. 10: Increasing k-mer sizes for Ash1_v2.2 genome (2973.08 MB) and chromosome 21 (42.82 MB). Sizes $k \geq 9$ could not be analyzed due to excessive runtime.

proved space savings, as larger k-mers enhanced the variation in symbol frequencies that Huffman coding can exploit. 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 [24]. Consequently, a k-mer size of $k = 8$ was used for subsequent Huffman coding compression and DNAzip encoding analyses, as testing k-mer sizes larger than 9 was not feasible due to the excessive runtime and memory requirements associated with the 4^k time and space complexity when analyzing whole genomes.

3) DNAzip: In reviewing each component of DNAzip's original implementation (VINTs, DELTA, Huffman coding, and dbSNP), we measured 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 [25]. 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. 11 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 0.16 MB, yet the introduction of the dbSNP increased the compressed file size to 0.17 MB. Applying these components to the full genomic sequence revealed similar trends. Fig. 13 reveals a more distinct gap between VINTs and DELTA (8.97 MB), as to when Huffman coding is introduced (8.87 MB). Finally, the addition of the dbSNP remained detrimental by increasing the compressed file size to 10.06 MB.

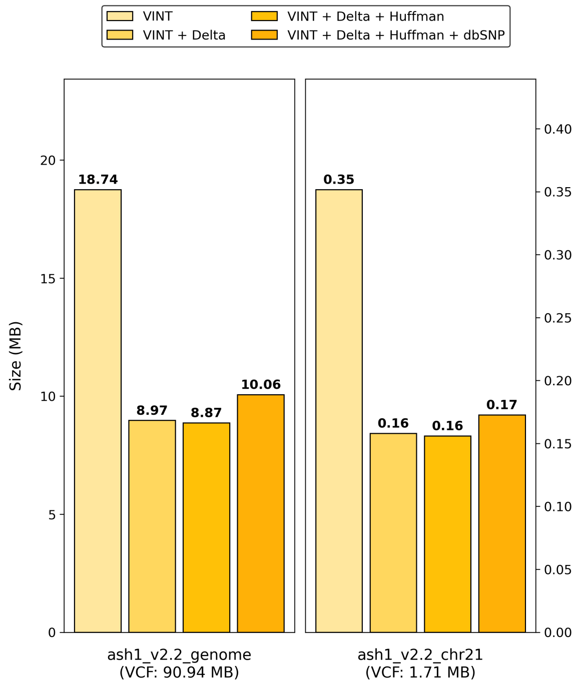


Fig. 11: DNAzip compression performance for the aligned genome (90.94 MB) and chromosome 21 (1.71 MB) variant call format (VCF) files of Ash1_v2.2 is shown for each additional DNAzip component, starting from VINT, with appropriate y-axis scaling.

C. Compression

1) Biocompress 1: Biocompress 1 employs two methods to encode DNA sequences: two-bit encoding and repetition encoding. The compression performance of the algorithm was analyzed by comparing how the two-bit encoding compared to the full Biocompress 1 algorithm in Fig. 12.

2) DNAzip: DNAzip compression was analyzed by assessing the storage requirements of both the prepro-

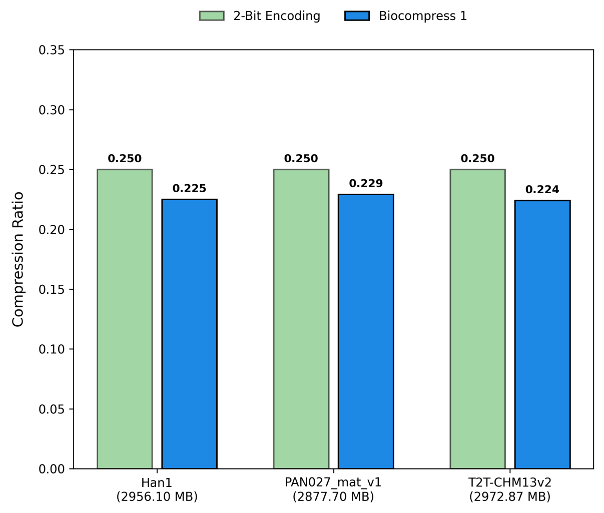


Fig. 12: Comparison of compression ratios of two-bit encoding and the full Biocompress 1 algorithm. Evaluation was performed on Pan027_mat_v1, T2T-CHM13v2 and Han1.

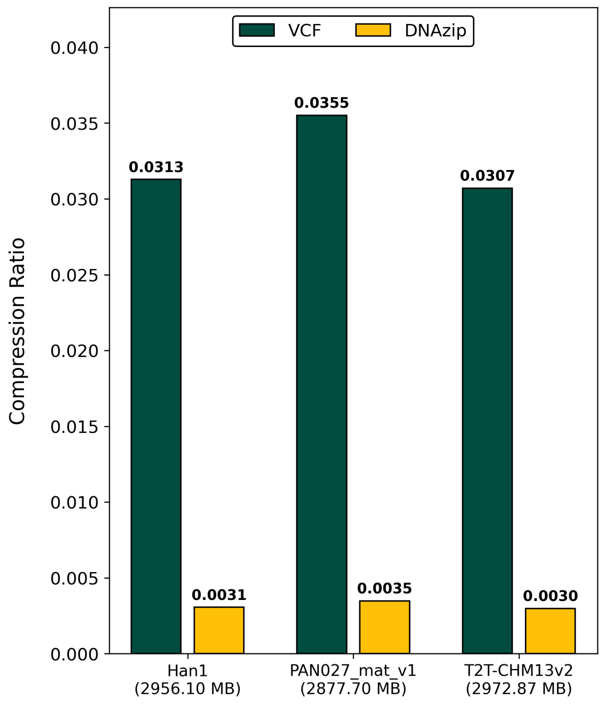


Fig. 13: Comparison of end compression performance between variant call format (VCF) files (green) and DNAzip (yellow) across three variant datasets, with GRCh38 serving as the reference. DNAzip optimizations applied for this comparison included VINT, DELTA, and Huffman coding.

cessed variants in the VCF after target-reference alignment and the compressed files after compressing the each respective VCF with DNAzip, illustrated in Fig. 13. 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 II.

TABLE II: Variant call format (VCF) file structure showing each variant detected in the target genome relative to the reference. Each line represents a single variant, including its type, chromosome, position, and corresponding reference and alternate bases.

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

3) Comparison: To understand how well different compression methods perform on real genomic data, we compared our three algorithms (Biocompress 1, DNAzip, and Huffman coding) across three complete human genomes: Pan027_mat_v1, T2T-CHM13v2, and Han1, as shown in Fig. 14. This allowed us to evaluate how each method handles differing structures of human genomes. In addition, we analyzed how the algorithms behave when compressing multiple genomes, including the effect of DNAzip's requirement for a full reference genome in Fig. 15.

IV. Analysis

A. Biocompress 1

Biocompress 1 was able to achieve a compression ratio of 0.225 ± 0.002 for the three full human genomes shown in Fig. 12, that more than four compressed genomes can be stored in the same amount of space as one uncompressed genome. The slight variations in the compression ratios are likely due to minor differences in the length and number of repetitions found in the DNA sequences.

By comparing the file sizes after performing two-bit encoding and after running Biocompress 1 in Fig. 12, we find that two-bit encoding was responsible for an average of 96.87% of the compression of the algorithm. The repetition encoding was only responsible for the remaining 3.13% of the compression beyond what two-bit encoding achieved. Given that the vast majority of the algorithm's runtime stems from creating the 4-ary tree and searching for repetitions, the viability of Biocompress 1 is questionable. If time or memory usage are constrained, Biocompress 1 is likely not the optimal

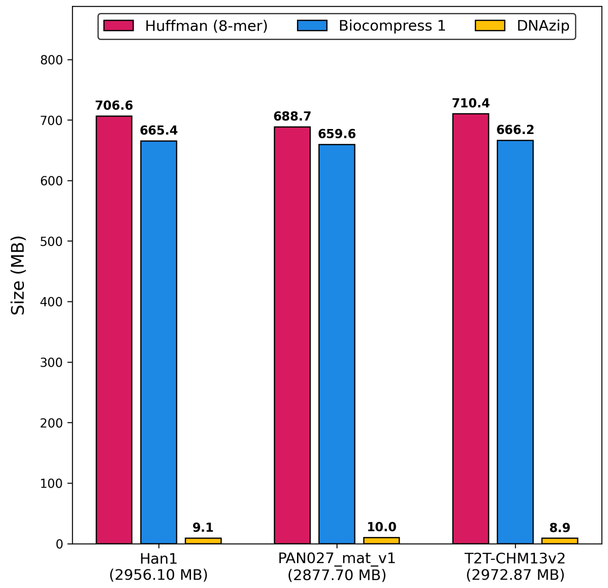


Fig. 14: Comparison of compression performance on Han1, Pan027_mat_v1, and T2T-CHM13v2 by Huffman coding, Biocompress 1, and DNAzip with GRCh38 serving as a reference for DNAzip.

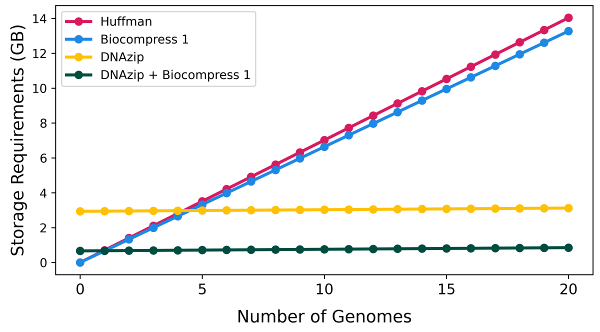


Fig. 15: Estimated storage size required by increasing number of genomes for Huffman coding, Biocompress 1, and DNAzip using the each algorithm's respective average file size per additional genome. DNAzip has an initial 3 GB storage size cost due to the requirement of the reference genome GRCh38. The combination of DNAzip and Biocompress 1 compressing GRCh38 has an initial cost of 0.66 GB.

choice, as the significant runtime and memory usage account for only a small improvement when compared to two-bit encoding.

B. DNAzip

Fig. 13 illustrates the substantial decrease in storage size by compressing the VCF using DNAzip. By employing VINTs, DELTA, and Huffman coding of the insertions, the VCF saw a 90.2% decrease in storage size, extending the VCF's already substantial space savings.

Aligning the target and reference genomes created VCF files that achieved an average compression 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. 13 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. 13. 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. Comparison

In comparing all three algorithms in Fig. 14, 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, Huffman and Biocompress 1 achieve similar results (688.7 MB and 659.6 MB, respectively), however, a drastic increase is seen in DNAzip's compression (10.0 MB). These results were comparable when applied to genomic files T2T-CHM13v2 and Han1, with DNAzip achieving compressed file sizes of 8.9 and 9.1 MB, respectively.

While it is evident that the space savings provided by DNAzip are greater than those provided by Huffman and Biocompress 1, DNAzip requires an additional reference genome. 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. 15, it is shown that if storing less than 5 genomes, the cost of storing the reference is too high. 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 reduce the storage necessary to store the reference genome, we explored how combining DNAzip with Biocompress 1 would influence overall compression efficiency in Fig. 14. Specifically, we first compressed the reference genome using Biocompress 1 and then encoded all subsequent genomes with DNAzip. Our analysis revealed that applying Biocompress 1 to the reference genome significantly reduces baseline storage costs, compressing the reference genome to a size of 665 MB. When combined with DNAzip for additional genomes, it also enhances per-genome compression as the number of genomes increases. Furthermore, using Biocompress 1 for the reference genome makes DNAzip more efficient after just one genome, compared to five when the reference is not compressed.

V. Conclusion

In this paper, we demonstrated the advantages of DNA-specialized compression algorithms over general text compression methods. Between our two DNA-specialized algorithms, we further showed the benefits of reference-based approaches. In addition, we established a cost-effectiveness threshold that indicates when DNAzip is preferable to Biocompress 1. Finally, by combining these insights, we achieved the greatest overall space savings by applying Biocompress 1 to the reference genome and DNAzip to the remaining target genomes.

Overall, reference-based compression algorithms like DNAzip represent a crucial advancement toward scalable, cost-effective genomic data management in this era of high-throughput sequencing. Despite reference-based compression's superior space savings, it also introduces computational complexity by requiring reference-target genome alignment, preprocessing of the variation data after alignment, and storage of reference data.

Therefore, to gain a more comprehensive understanding of DNA compression, additional implementation and in-depth analysis of more advanced iterations of these specialized algorithms would be beneficial. This would explore the techniques used by recent compression algorithms, such as the non-reference-based GReEn and the reference-based GenCoder, which utilizes neural networks, as examples of modern approaches for more efficient compression of DNA sequences [15] [26]. Similarly, in the realm of reference-based compression, recent studies have focused on creating a pangenome, a new reference genome created by merging multiple genome assemblies, carefully chosen to better reflect global genetic diversity [27].

Acknowledgement

We would like to thank our advisor, Professor Layla Oesper for her invaluable guidance and support. We would also like to thank Mike Tie for his technical expertise and assistance.

Appendix A
Download dbSNP

To construct a dbSNP compatible with DNAzip, we obtained the CommonSNPs155 track from the UCSC Genome Browser in bigBed (.bb) format [28]. 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 seen in Table II. This normalized and genome-matched dbSNP set ensured that variant lookup, encoding, and compression steps within DNAzip were accurate.

Appendix B
Genome Alignment

For preliminary testing of the DNAzip implementation, initial variant analysis was done using high-confidence VCF files for HG002/HG003/HG004 [23]. 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 detailed in Fig. 6.

The alignment pipeline begins with nucmer (MUMmer4), which performs whole-genome, base-level alignment to identify matching blocks of bases and variations from Fig. 3 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 II and serve as the foundational input enabling DNAzip's reference-based compression workflow.

References

[1] M. L. Metzker, “Sequencing technologies — the next generation,” Nat Rev Genet, vol. 11, no. 1, pp. 31–46, Jan. 2010, doi: https://doi.org/10.1038/nrg2626.

[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] R. M. Durbin et al., “A map of human genome variation from population-scale sequencing,” Nature, vol. 467, no. 7319, pp. 1061–1073, Oct. 2010, doi: https://doi.org/10.1038/nature09534.

[4] 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/

[5] H. Herzel, W. Ebeling, and A. O. Schmitt, “Entropies of biosequences: The role of repeats,” Phys. Rev. E, vol. 50, no. 6, pp. 5061–5071, Dec. 1994, doi: https://doi.org/10.1103/PhysRevE.50.5061.

[6] [1] G. Nagy and L. Nagy, “Motif grammar: The basis of the language of gene expression,” Comput Struct Biotechnol J, vol. 18, pp. 2026–2032, July 2020, doi: https://doi.org/10.1016/j.csbj.2020.07.007.

[7] H. Herzel, O. Weiss, and E. N. Trifonov, “10-11 bp periodicities in complete genomes reflect protein structure and DNA folding,” Bioinformatics, vol. 15, no. 3, pp. 187–193, Mar. 1999, doi: https://doi.org/10.1093/bioinformatics/15.3.187.

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

[9] S. Grumbach and F. Tahi, “A new challenge for compression algorithms: Genetic sequences,” *Information Processing & Management*, vol. 30, no. 6, pp. 875–886, Nov. 1994, doi: https://doi.org/10.1016/0306-4573(94)90014-0.

[10] Xin Chen, S. Kwong, and Ming Li, “A compression algorithm for DNA sequences,” IEEE Eng. Med. Biol. Mag., vol. 20, no. 4, pp. 61–66, Aug. 2001, doi: https://doi.org/10.1109/51.940049.

[11] 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: https://doi.org/10.1093/bioinformatics/btn582.

[12] K. K. Kidd, A. J. Pakstis, W. C. Speed, and J. R. Kidd, “Understanding Human DNA Sequence Variation,” Journal of Heredity, vol. 95, no. 5, pp. 406–420, Sept. 2004, doi: https://doi.org/10.1093/jhered/esh060.

[13] “Knowledge article · NLM Customer Support Center,” U.S. National Library of Medicine, https://support.nlm.nih.gov/kbArticle/?pn=KA-03578 (accessed Nov. 19, 2025).

[14] Pavlichin DS, Weissman T, Yona G. The human genome contracts again. Bioinformatics. 2013 Sep 1;29(17):2199-202. doi: https://doi.org/10.1093/bioinformatics/btt362. Epub 2013 Jun 22. PMID: 23793748.

[15] 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: https://doi.org/10.1093/nar/gkr1124.

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

[17] National Human Genome Research Institute, “Talking Glossary of Genetic Terms — NHGRI,” Genome.gov, 2019. https://www.genome.gov/genetics-glossary (accessed Oct. 9, 2025).

[18] E. Pajarillo, A. Rizor, J. Lee, M. Aschner, and E. Lee, “The role of astrocytic glutamate transporters GLT-1 and GLAST in neurological disorders: Potential targets for neurotherapeutics,” Neuropharmacology, vol. 161, p. 107559, 2019, doi: https://doi.org/10.1016/j.neuropharm.2019.03.002.

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

[20] Karolchik D, Hinrichs AS, Furey TS, Roskin KM, Sugnet CW, Haussler D, Kent WJ. The UCSC Table Browser data retrieval tool. Nucleic Acids Res. 2004 Jan 1;32(Database issue):D493-6.

[21] 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: https://doi.org/10.1371/journal.pcbi.1005944.

[22] 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).

[23] “Genome in a Bottle,” NIST, Aug. 2012, Available: https://www.nist.gov/programs-projects/genome-bottle (accessed Oct. 5, 2025).

[24] A. Moffat, “Huffman Coding,” ACM Comput. Surv., vol. 52, no. 4, p. 85:1-85:35, Aug. 2019, doi: https://doi.org/10.1145/3342555.

[25] L. Phan et al., “The evolution of dbSNP: 25 years of impact in genomic research,” Nucleic Acids Res, vol. 53, no. D1, pp. D925–D931, Jan. 2025, doi: https://doi.org/10.1093/nar/gkae977.

[26] 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: https://doi.org/10.1109/TCBB.2024.3366240.

[27] W.-W. Liao et al., “A draft human pangenome reference,” Nature, vol. 617, no. 7960, pp. 312–324, May 2023, doi: https://doi.org/10.1038/s41586-023-05896-x.

[28] W. J. Kent, A. S. Zweig, G. Barber, A. S. Hinrichs, and D. Karolchik, “BigWig and BigBed: enabling browsing of large distributed datasets,” Bioinformatics, vol. 26, no. 17, pp. 2204–2207, Sept. 2010, doi: https://doi.org/10.1093/bioinformatics/btq351.

TABLE S1: Genomic file details for each sample [19].

	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