Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

HiDAC: A Hierarchical Dictionary-Aided Compression Framework for Genomic Sequences

Manthan Shah
Imperial College London
Rahul Semwal
rsemwal@iiitn.ac.in

Indian Institute of Information Technology Nagpur
Imlimaong Aier
National Institute of Technology, Durgapur
Prabhat Tripathi
Indian Institute of Information Technology Allahabad
Pritish Kumar Vardwaj
pritish@iiita.ac.in

Indian institute of information technology Allahaba

Research Article

Keywords: DNA compression, Dictionary-based compression, Aho-Corasick algorithm, Arithmetic encoding, Compressed-domain sequence analysis

Posted Date: September 25th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11151040/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

HiDAC: A Hierarchical Dictionary-Aided Compression Framework for Genomic Sequences

Manthan Shah* Rahul Semwal† Imlimaong Aier‡ Prabhat Tripathi§
Pritish Varadwaj¶

September 24, 2026

Abstract

The rapid increase in genomic data generated by next-generation sequencing (NGS) has created a strong demand for efficient DNA compression methods that can reduce storage requirements and support downstream sequence analysis. Existing approaches such as statistical, reference-based, and dictionary-based techniques provide good compression performance, but most of them are primarily designed for storage and require full decompression before any analytical operation can be carried out. In this work, a hierarchical dictionary-based DNA compression framework, HiDAC (Hierarchical Dictionary-Aided Compression) is proposed that can perform efficient compression and compressed-domain analysis of genomic sequences. The method identifies recurring patterns of variable length through an iterative cost-guided substitution strategy and represents them using reusable tokens. For efficient encoding, the generated dictionary is integrated with an Aho–Corasick based multi-pattern matching scheme followed by context-modelled arithmetic coding of the tokenised sequence. The proposed framework was evaluated on multiple genomic datasets, including *Homo sapiens*, *Escherichia coli*, and *Mycobacterium tuberculosis*, and was compared with general-purpose compressors as well as specialized genomic compression methods such as NAF and HMG. Experimental results showed that the proposed method achieved superior or competitive compression performance compared with the other methods across the tested datasets. In addition, substring-frequency analysis performed directly on the tokenised representation was on average 2.6–2.8× faster than the same analysis performed on the original sequence representation, across query lengths ranging from 13 to 466 bases and across two genomes of differing size and repeat structure. These results indicate that the proposed framework is effective not only for DNA compression but also for enabling computation directly on semi-compressed (tokenised) genomic data.

Keywords: DNA compression, Dictionary-based compression, Aho-Corasick algorithm, Arithmetic encoding, Compressed-domain sequence analysis

1 Introduction

Deoxyribonucleic acid (DNA) is the basic unit of hereditary information in living organisms. A large amount of biological information is stored in DNA using only four nucleotide bases. With the development of next-generation sequencing (NGS) technologies [16], the amount of genomic data has increased rapidly. This has created difficulties in storage, transmission, and analysis of genomic datasets. Conventional lossless compression tools such as Gzip [33] and Bzip2 [45] were developed for general text data and are not suitable for DNA sequences. These tools do not make proper use of the small alphabet of DNA and the repetitive patterns present in the sequence [18]. As a result, several DNA-specific compression methods have been reported. These methods can broadly be divided into three groups:

*Department of Computing, Imperial College London, London, United Kingdom; Department of Computer Science and Engineering, Indian Institute of Information Technology Nagpur, Maharashtra, India. E-mail: manthanjigar.shah26@imperial.ac.uk

†Department of Computer Science and Engineering, Indian Institute of Information Technology Nagpur, Maharashtra, India. E-mail: rsemwal@iiitn.ac.in

‡National Institute of Technology, Durgapur, West Bengal, India. E-mail: iaier.bt@nitdgp.ac.in

§Indian Institute of Information Technology Allahabad, Prayagraj, Uttar Pradesh, India. E-mail: pmb2022001@iiita.ac.in

¶Indian Institute of Information Technology Allahabad, Prayagraj, Uttar Pradesh, India. E-mail: pritish@iiita.ac.in

- **Statistical Compression**: These methods use probabilistic models to estimate the next symbol in the sequence. Methods such as GenCompress [7] and XM [32] use Markov models and context-based schemes to encode probable symbols with fewer bits. These methods can provide good compression, but they usually depend on complex models and require high computation. Also, such methods are not always effective in capturing the structural regularities present in DNA sequences.
- **Reference-Based Compression**: These methods [12, 31, 44, 53] encode a target genome with respect to a known reference genome. Techniques and tools such as SAM[29] and HiRGC [31] represent only the differences between the target and reference sequences. This often gives high compression for related genomes. However, these methods depend on the availability of a suitable reference genome.
- **Dictionary-Based Compression**: These methods [4, 38] replace recurring subsequences by shorter tokens and store the mapping in a dictionary. Such methods do not require an external reference and can utilize repeated patterns from the sequence itself. However, most of these methods process the sequence in a sequential manner. Therefore, they have limited ability to capture nested or hierarchical repeats. Further, the dictionary usually needs to be rebuilt for every new dataset, which increases preprocessing cost.

Although these DNA compression tools [2, 17, 26, 41] reduce storage and transmission cost, they are mainly meant for compression only. The compressed sequence produced by these methods is generally not useful for direct analysis. Even simple tasks such as substring search, comparison, and repeat analysis need full decompression of the sequence before processing. This increases computational overhead and reduces the practical use of compression for large genomic data.

Hence, there is a need for a method which can provide good compression and also support operations on the compressed sequence. Such a method can be useful not only for efficient storage and transmission, but also for faster downstream analysis. In this work, we propose HiDAC, an iterative framework to identify an optimal set of recurring patterns of different lengths from a DNA sequence. These patterns are used to create a reusable hierarchical dictionary, where repeated subsequences are replaced by special tokens. For this, a custom cost function is used that guides the pattern selection and dictionary construction process. The obtained dictionary is then presented to the Aho-Corasick automaton [1] for pattern matching and replacement with a token. Following this, context-modelled arithmetic coding is used for bit-level compression.

The proposed method was evaluated using three metrics: the net reduction in sequence length of the compressed sequence, space saved at the file level, and the ability to support operations on compressed data. The results show that the proposed method outperforms commonly used general-purpose compressors as well as special purpose genomic compressors. Unlike most existing methods, HiDAC also allows analytical operations on the semi-compressed (tockenized) representation without prior decompression.

2 Literature Review

Specialized DNA compression tools have been developed for efficient storage of genomic data. These tools can be grouped into four broad categories: statistical, reference-based, dictionary-based, and hybrid approaches. Although each category has some advantages, they also have limitations when applied to large genomic datasets.

2.1 Statistical Compression Approaches

Many early DNA compression methods are based on statistical modelling. Methods such as *XM* [32], and *ARM* [26] use probabilistic models of nucleotide sequences, including Markov models, finite-context models [7], context-tree weighting (CTW), and arithmetic coding, to estimate symbol probabilities and assign variable-length codes. These methods mainly exploit short-range correlations by predicting a nucleotide from a fixed-length context of preceding bases. These methods perform better than simple 2-bit encoding, but their gain is limited in practice. Earlier studies have shown that statistical compressors generally achieve compression in the range of 1.7–1.9 bits per base [15]. When model order increases, the improvement becomes small, while memory requirement and computational cost increase. Because of this, such methods are difficult to use for multi-gigabase genomes. DNA sequences also contain

approximate repeats, nested repeats, and long-range dependencies, which are not handled well by fixed-order models. Therefore, even advanced statistical methods such as expert mixing [32] and normalised maximum likelihood modelling [27] provide only limited improvement with high computational cost [15].

2.2 Reference-Based Compression Approaches

Reference-based compression methods use the similarity between related genomes. Methods such as *SAM*[29], and *RLZ* [13] store only the differences between a target genome and a reference genome [19–21, 23]. Usually, substitutions, insertions, deletions, and positional offsets are encoded, while long exact matches are handled using alignment or Lempel–Ziv style parsing. These methods can achieve high compression ratios when a suitable reference genome is available. However, their performance depends strongly on the reference sequence. This becomes a drawback for metagenomic samples, divergent genomes, and newly sequenced organisms where a proper reference may not be available [50]. So, although reference-based methods are effective, their use is limited because of dependence on external reference data.

2.3 Dictionary-Based Compression Approaches

Dictionary-based compression methods reduce the size of DNA sequences by identifying repeated substrings and replacing them with shorter codes. Earlier techniques such as BioCompress2 [18] and Dnacompress [8], used this principle to encode both exact and approximate repeats. After that grammar-based methods, such as those based on Re-Pair [35], were developed to generate compact representations by identifying recurrent sequence patterns. An important fact with these methods is that they do not require an external reference sequence. However, these methods have limitations as they used greedy strategy [15] to construct dictionary. Due to this, these methods are not able to effectively identify the overlapping repeats, especially when the patterns are nested within one another. Furthermore, even if these methods offer effective compression, the format that is produced is primarily meant for data transport and storage. Due to this, in most biological applications, the compressed sequence must first be decompressed before it can be analyzed.

2.4 Hybrid and Modern Approaches

Hybrid compression methods incorporates the advantages of previous compression methods. Their primary objective is to optimize compression while simultaneously decreasing memory requirements and decompression time. This category includes approaches like NAF (Nucleotide Archival Format) [28], GDC2 [12], and graph-based encoders. For example, NAF supports features such as streaming decompression and random access, in addition to offering a high compression ratio. Similarly, GDC2 uses both dictionary-based and reference-based approaches to compress closely related genomic sequences more effectively. Most existing DNA compression methods suffer from some common limitations. Better compression is often obtained at the cost of higher memory usage, longer preprocessing time, or increased computational overhead. In reference-based and dictionary-based approaches, repeated alignment or repeated dictionary construction is often required for every new dataset. This becomes a major bottleneck for large genomes such as the human genome. Another important limitation is that the compressed representation produced by these methods is mainly useful for storage and transmission only. It cannot be used directly for tasks such as pattern search, subsequence retrieval, or sequence comparison.

To address these limitations, in this work, the focus was given to compression efficiency and compressed-domain usability. The main objective was to identify suitable patterns for the construction of a reusable reference dictionary. This helps in representing the repetitive structural characteristics of DNA sequences in a better way. Further, the use of Aho–Corasick algorithm together with context based arithmetic encoding helps in achieving efficient compression.

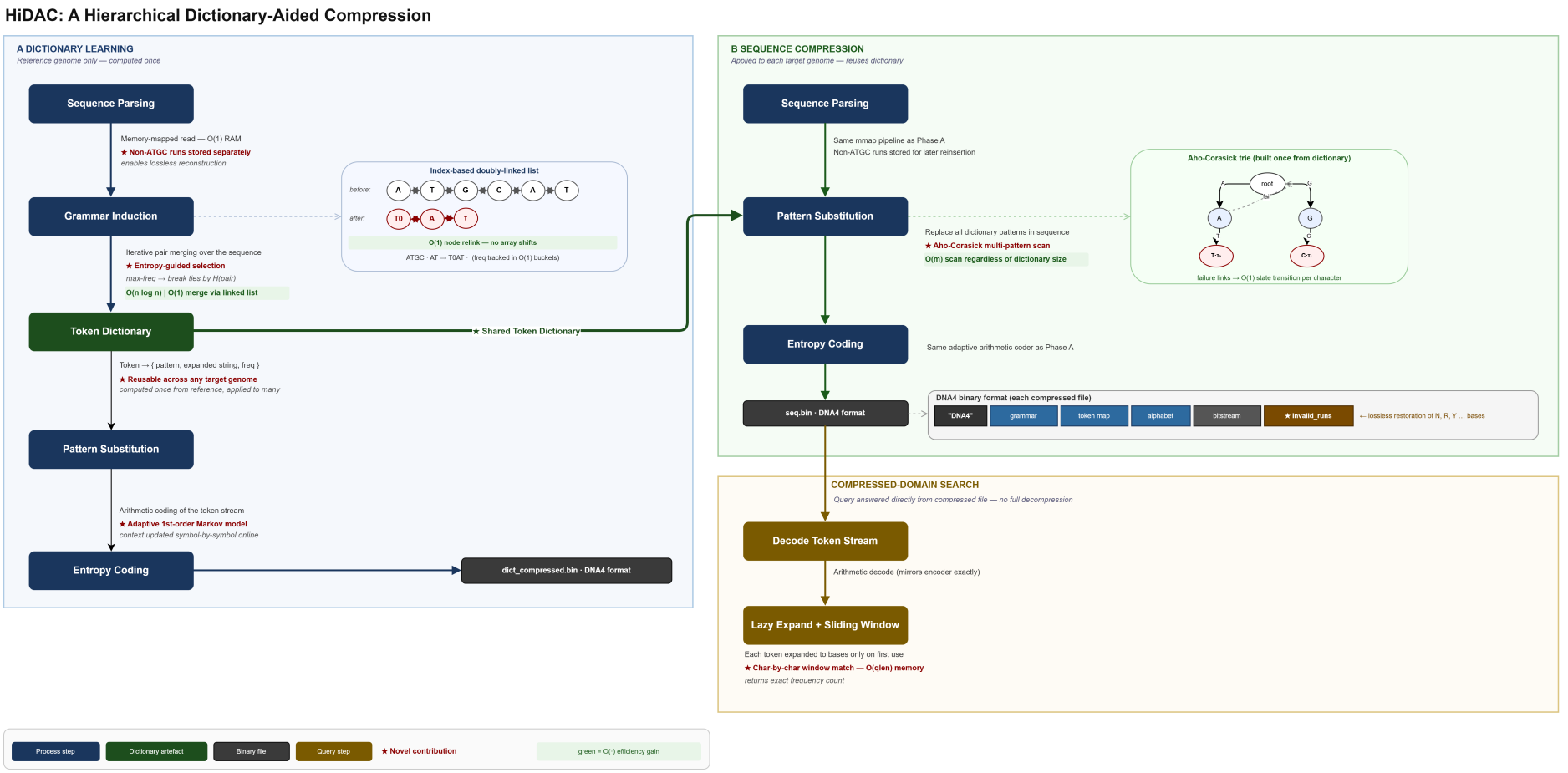


Figure 1: Methodology

3 Methodology

Figure 1 represents the flow of HiDAC. In our proposed method, the DNA sequence is first preprocessed to remove non-base characters, empty spaces, and headers. After preprocessing, the dictionary creation process begins by iteratively scanning the sequence to identify the most frequent k-mer or pattern of length 2; the obtained pattern is then replaced by a unique character token. In each iteration, the discovered token is stored in a dictionary and can appear in subsequent patterns. This iterative process allows patterns to be built in a hierarchy, where newly identified patterns may include previously introduced tokens. The selection of a pattern is guided by a cost function that evaluates the net benefit of substitution in terms of sequence compression and overall file size reduction. Subsequently, we use an Aho–Corasick automaton[1], which is a trie-based multi-pattern matching algorithm for pattern matching and token replacement using the created dictionary. Finally, for storage of compressed data, the tokenised sequence is converted into an integer stream and compressed using context-modelled arithmetic coding[51], where the encoding adapts dynamically based on global token frequencies and local context. Once the bitstream was generated, the final hierarchical dictionary, represented in the form of an integer grammar, along with the compressed sequence, was stored in a compact bit-level format. The metadata required for lossless reconstruction was also preserved during this stage. The detailed description of each step in the proposed methodology is provided in Subsections 3.1 to 3.4.

3.1 Preprocessing

The DNA sequence was cleaned up by removing headers, blank spaces, and invalid nucleotide symbols. After that, all the valid nucleotides were changed to uppercase, leaving only the standard nucleotides A, T, G, and C in the string. Non-ATGC characters were identified and saved as tuples including the start location, character identification, and run length. Consequently, all information required for lossless reconstruction was saved, and a clean and consistent sequence was created for compression.

3.2 Dictionary Creation

Following the preprocessing step, the phase of dictionary creation was initiated, in which frequently recurring patterns were progressively substituted with distinct non-base tokens to create a nested hierarchical structure. The procedure begins by determining the most common two-mer in the sequence. In the event that multiple two-mers are discovered to have the same frequency, Shannon entropy is employed to break the tie. The reason for this is that high entropy patterns contain less redundancy and convey more information[30, 36, 46, 52]. After this, the selected two-mer is replaced with a new non-base token and its corresponding token-pattern mapping were stored in the dictionary. This token will be used later to construct longer composite patterns.

As the dictionary grows, the sequence becomes a mixture of bases and tokens. In each subsequent iteration, we again select a candidate pattern P with the highest frequency in the current sequence. In case of a tie, the same entropy-based rule as mentioned before shall be used. Notably, the candidate P can be either (i) a pure k-mer (i.e., solely bases), or (ii) a composite pattern (i.e., containing both tokens and bases). If P is a pure k-mer, it is simply replaced by a new unique token and added to the dictionary, as in the initialization phase. However, if P is composite, we assess whether it is worthwhile to substitute P using a cost-based criterion whose goals are: (i) maximize the effective compression gain and (ii) maintain a balanced frequency distribution of tokens in the dictionary, since replacing a pattern can change both the frequencies of existing dictionary tokens and the compressed sequence length.

To formalize this, let T be the collection of tokens in the dictionary. Suppose that we have found a candidate pattern P in the current sequence with frequency f_p. Let $t \in T$ be an existing token, with (current) sequence frequency f_t, and expanded nucleotide length ℓ_t (i.e., the number of base characters acquired when token t is recursively expanded). Let $n_{t,p}$ be the number of occurrences of the token t in the pattern P. We establish two values, OriginalCost and NewCost, to compare the token-encoding capability of the sequence before and after substitution.

First, the OriginalCost reflects the total amount of underlying nucleotide characters represented by the tokens currently encoding the sequence:

$$\text{OriginalCost} = \sum_{t \in \mathcal{T}} \ell_t \cdot f_t \tag{1}$$

This value serves as a reference threshold to evaluate whether the introduction of a particular token for P yields sufficient benefit.

Next, assuming we substitute all occurrences of P with a new token, the NewCost is computed as:

$$\text{NewCost} = \ell_P \cdot f_P + \sum_{t \in \mathcal{T}} \ell_t \cdot (f_t - f_P \cdot n_{t,P}) \tag{2}$$

where ℓ_P denotes the actual nucleotide length of pattern P (i.e., the true base-character length of the fully expanded pattern) and f_P denotes the frequency of pattern P. The first term $\ell_P \cdot f_P$ quantifies the nucleotide content represented by the new token introduced for P. The second term updates the contribution of each existing token t. Specifically, because token t occurs $n_{t,P}$ times inside a single occurrence of P, replacing f_p occurrences of P removes $f_p \cdot n_{t,P}$ occurrences of token t from the sequence. Hence, the updated frequency of token t becomes $(f_t - f_P \cdot n_{t,P})$, and its updated nucleotide contribution is $\ell_t \cdot (f_t - f_P \cdot n_{t,P})$. Summing this over all $t \in T$ yields the total nucleotide content represented by the remaining (updated) token frequencies after substitution.

After obtaining these two quantities, we accept the substitution of P with a new token if:

$$\text{NewCost} > \theta_1 \cdot \text{OriginalCost} \tag{3}$$

where θ_1 is an empirically determined threshold that controls the sensitivity of the replacement decision. This prevents the vocabulary from expanding due to substitutions that provide only minor improvements, resulting in a rise in dictionary size without a significant net gain.

A specific edge situation occurs when the candidate pattern P is formed only of tokens. Here, adding a new token for P adds no additional base-level content. Thus, to address this issue, we analyze the substitution from an encoding-efficiency point of view, by considering the average number of nucleotide characters encoded per token before and after substituting P. Let F_{orig} be the cumulative token frequency in the current encoded sequence, calculated as the sum of token occurrences of all dictionary tokens:

$$F_{\text{orig}} = \sum_{t \in \mathcal{T}} f_t \tag{4}$$

After replacing P the cumulative token frequency is now becomes:

$$F_{\text{new}} = \sum_{t \in \mathcal{T}} (f_t - f_P \cdot n_{t,P}) + f_P \tag{5}$$

Here, every current token frequency is reduced by the proportion of how many times it appears in P, since those occurrences are now accounted for by the new token; the term $+f_P$ adds the frequency of the new token that has been introduced. We then examine the average bases encoded per token before and after substitution using a second empirically-tuned threshold θ_2:

$$\frac{\text{NewCost}}{F_{\text{new}}} > \theta_2 \cdot \frac{\text{OriginalCost}}{F_{\text{orig}}} \tag{6}$$

This criterion ensures that substitutions are only performed if they lead to meaningful efficiency improvements and supports both compact representation as well as controlled dictionary expansion.

If a candidate pattern P satisfies the appropriate condition(s), it is replaced with a new unique token and the token-pattern mapping is added to the dictionary. This procedure of selection, evaluation and substitution is repeated until either the requisite number of iterations is achieved or no further feasible candidate patterns are left. The obtained tokens signify the most common and representative patterns learned from the sequence.

3.3 Multi-Pattern Matching and Greedy Replacement

Upon the completion of the dictionary's construction, the Aho–Corasick algorithm [1] is utilized to perform concurrent multi-pattern matching on the DNA sequence. Let the input sequence as $S = s_1 s_2 \dots s_n$; where $s_i \in \{A, T, G, C\}$. Let the dictionary have K tokens: $\mathcal{D} = \{(\tau_k, P_k, f_k)\}_{k=1}^{K}$,; where τ_k represents the token identifier, $P_k \in \Sigma^*$ signifies its expanded string pattern, and f_k indicates its frequency score utilized in greedy selection.

Algorithm 1 HiDAC: Hierarchical Dictionary-Aided Compression

Require: S: DNA sequence; I: max tokens; θ_1, θ_2: gain thresholds
Ensure: C: compressed sequence; D: token dictionary

Represent S as a doubly linked list; initialise pair-tracking structures and symbol caches
$D \leftarrow \{\}$; merges $\leftarrow 0$
while merges $< I$ **and** $f_{\max} \geq 2$ **do**
    Select the highest-frequency adjacent symbol pair P with maximum entropy
    Collect non-overlapping occurrences of P; let f be their count
    if P contains only bases {A,T,G,C} **then**
        Assign token $\tau \leftarrow P$
        Add τ to D
        merges $\leftarrow$ merges $+1$
    else
        if $f < 2$ **then**
            discard P; **continue**
        end if
        Compute *OrigCost*, *NewCost*, F_{orig}, F_{new}
        if gain condition not met **then**
            break
        end if
        Assign token $\tau \leftarrow P$
        Add τ to D
        for each non-overlapping occurrence of P **do**
            Substitute P with τ in the linked list; update neighbour pairs
        end for
        merges $\leftarrow$ merges $+1$
    end if
end while
Traverse linked list to collect C
return C, D

3.3.1 Automaton Construction

The automaton is formed by placing each pattern P_k into a trie, which stores all patterns according to their common prefixes, so that common substrings are not processed more than once. For every terminal node we keep the pair (P_k, τ_k) that directly links the matched pattern to its allocated token τ_k. However, a trie can only match patterns starting from the root. If a partial match breaks at an intermediate character, the search must be restarted from the beginning. To avoid this problem, two additional structures are added to the trie to form the full Aho–Corasick automaton.

The first is the *failure link*. For every node in the trie representing a string x, its failure link points to the node representing the longest proper suffix of x that is also a valid prefix of some pattern in the trie. When the automaton encounters a character that does not continue the current partial match, rather than abandoning all progress and returning to the root, it follows the failure link and resumes matching from the most informative fallback position. This design ensures that each input character is examined only once during scanning. That is why the automaton runs in linear time [1].

The second is the *output link*. Consider an example where the dictionary has two patterns, say `ATGC` and `GC`. When the automaton reaches the end of a match for `ATGC`, the shorter pattern `GC` also ends at the same location, but a naive trie traversal would just report the longer match and miss the shorter one entirely. Each node's output link points to the longest other pattern in the trie that is a suffix of the string at that node, so that all valid matches terminating at a given place can be reported in a single step. The automaton finds every occurrence of every pattern in $O(M)$ construction time, where $M = \sum_{k=1}^{K} |P_k|$ is the total length of all patterns, using failure links and output links together. This is the one-time setup cost of the matching pipeline; the automaton is built once and reused for each sequence in the collection of sequences.

3.3.2 Multi-Pattern Matching

After the automaton has been built, the input string S of length n is scanned from left to right in one pass. At each stage active partial matches are tracked, and whenever a match is completed it is reported. When the match terminates at location j, the automaton produces the triplet $(j,\ P_k,\ \tau_k)$. Here, j is the location at the end of S, P_k is the pattern matched and τ_k is the token assigned to P_k. Then the start position is retrieved as $i = j - |P_k| + 1$.

All matches at the same position i are gathered in the candidate set

$$\mathcal{M}(i) = \{\, (j,\ \tau_k,\ \ell_k,\ f_k) \,\}, \tag{7}$$

where j is the end position of the match, τ_k is the assigned token, $\ell_k = |P_k|$ is the pattern length, and f_k is the corpus frequency of P_k stored in the dictionary. This categorization is helpful because It allows us to consider all competing matches at position i together before any substitution. The time complexity of matching step is $O(N + Z)$, where N is the length of S and Z is the total number of matches given. This cost is independent of the amount of patterns examined [1].

3.3.3 Greedy Replacement Strategy

Once the candidate matches at location i are available, the greedy based selection approach selects one for compression. The underlying notion is simple: we prefer a longer match, because it replaces more characters with one token. If two matches of the same length occur, the one that occurs most often is selected. This rule is written as:

$$\tau^* = \arg \max_{(j,\,\tau_k,\,\ell_k,\,f_k)\,\in\,\mathcal{M}(i)} (\ell_k,\ f_k), \tag{8}$$

where the comparison is lexicographic. The first term, ℓ_k, gives priority to the longest match. The second term, f_k, is used to break ties and favors the pattern with higher corpus frequency. Therefore, no lookahead is needed beyond the current candidate set which keeps the decision step simple.

After selection, the token τ^* is emitted. The position pointer is then updated to $i \leftarrow j + 1$, so all characters covered by the chosen match are skipped in one move. If no candidate is found at position i, the symbol $S[i]$ is emitted as a literal. The pointer then moves ahead by one position. The final compressed sequence is written as $(t_1,\ t_2,\ \ldots,\ t_m)$, where each t_r is either a token τ_k or an unmatched literal symbol.

Although the greedy pointer advances through the input sequence monotonically, the Aho–Corasick automaton may report overlapping matches and multiple candidate patterns at the same position. Therefore, when the reported matches are grouped by starting position, the matching and greedy replacement stage requires $O(N + Z)$ time, where Z is the total number of reported matches.

3.4 Entropy Coding of the Token Stream

In the final stage of the pipeline, a compact binary representation of the token sequence is produced that may be stored or transmitted. At this time structural compression has been achieved through pattern replacement, but the resulting token stream may still contain statistical redundancy. For example, some tokens might occur more frequently than others and neighboring tokens might be correlated. The remaining redundancy is reduced using arithmetic coding [51].

Arithmetic coding encodes a whole sequence as a single interval on a number line. The interval is refined progressively according to the coding model when each symbol is analyzed. The width of the final interval is proportional to the likelihood of the sequence: more probable sequences have wider intervals, and can be stored with fewer bits, less probable sequences have narrower intervals, and require more bits. This allows arithmetic coding to approach the entropy limit of the sequence. For arithmetic coding to be effective, symbol probabilities must be estimated consistently. This role is provided by the context model.

3.4.1 Context Modelling

Rather than assuming alphabets (includes both tokens and bases) are independent, the probability of each alphabet is estimated from the alphabet that immediately precedes it. This order-1 context model [10] captures short-range dependencies in the tokenised stream. Let $C(x, y)$ denote the number of times token y has followed token x in the portion of the sequence processed so far, and let $\mathcal{A}$ denote the

alphabet, defined as the sorted set of all distinct alphabet identifiers in the compressed sequence. The conditional probability assigned to alphabet t_i given its predecessor t_{i-1} is computed using the transition counts, and is defined as follows:

$$P(t_i \mid t_{i-1}) = \frac{C(t_{i-1},\, t_i) + 1}{C(t_{i-1}) + |\mathcal{A}|}, \tag{9}$$

where $C(t_{i-1}) = \sum_{a \in \mathcal{A}} C(t_{i-1}, a)$ represents the total number of alphabets observed after t_{i-1} up to that point. The transition count $C(t_{i-1}, t_i)$ is incremented by 1 in the numerator, while $|\mathcal{A}|$ is added to the denominator. This corresponds to Laplace smoothing [34], which prevents any transition from being assigned zero probability.

For the first alphabet, no preceding context is available, so a uniform prior is used. All alphabets are initialized with frequency 1, giving each token an initial probability of $1/|\mathcal{A}|$. Both the context counts $C(x, y)$ and the global counts are updated after each alphabet is coded, allowing the decoder to reproduce the same probability estimates independently during decompression. These probabilities are then used to refine the arithmetic coding interval at each step.

3.4.2 Integer Arithmetic Encoding

The arithmetic encoder begins with a working interval $[L, H]$, initially set to $[0,\ 2^{32} - 1]$. This interval is gradually narrowed as each symbol is encoded. At every step, the current interval is divided according to the cumulative probabilities produced by the context model, and the sub-interval assigned to the current symbol t_i becomes the new working interval.

Given the interval $[L_{i-1}, H_{i-1}]$ before encoding the $i-1$ symbol, the updated bounds are calculated as

$$L_i = L_{i-1} + \left\lfloor \frac{\text{range}_{i-1} \cdot \text{low}_i}{\text{total}_i} \right\rfloor$$

$$H_i = L_{i-1} + \left\lfloor \frac{\text{range}_{i-1} \cdot \text{high}_i}{\text{total}_i} \right\rfloor - 1$$

where $\text{range}_{i-1} = H_{i-1} - L_{i-1} + 1$ is the width of the interval before the update. The values (low_i) and (high_i) represent the lower and upper cumulative frequency bounds of symbol t_i under the current smoothed context model. In particular, low_i is the sum of the frequencies of all symbols that occur before t_i in the alphabet ordering. The value of high_i is obtained by adding the frequency of t_i to this cumulative sum. The term total_i represents the total frequency of all symbols in the alphabet $\mathcal{A}$ under the current model.

These values assign t_i to a unique sub-interval within $[L_{i-1}, H_{i-1}]$. The width of this sub-interval is proportional to the probability of the symbol. As more symbols are encoded, the working interval becomes progressively smaller. After all m symbols have been processed, the final interval $[L_m, H_m]$ uniquely represents the complete tokenised sequence.

Because the interval cannot continue shrinking indefinitely within a fixed 32-bit representation, renormalization is performed throughout the encoding process. A bit 0 is emitted when both interval bounds lie in the lower half of the 32-bit range, that is, when $H_i < 2^{31}$. Similarly, a bit 1 is emitted when both bounds lie in the upper half, which occurs when $L_i \geq 2^{31}$. The underflow condition occurs when the interval lies within the middle half of the range, such that $L_i \geq 2^{30}$ and $H_i < 3 \cdot 2^{30}$. This case is handled using the standard Witten-Neal-Cleary procedure [51]. As a result, output bits are generated progressively during encoding rather than being produced only after the complete sequence has been processed. Once the final symbol has been encoded, a flushing step is performed to resolve any remaining pending bits.The final output is a compressed bitstream whose expected length can be analysed in relation to the entropy of the token sequence.

3.4.3 Compression Optimality

The expected length of the bitstream produced by this scheme satisfies:

$$\mathbb{E}[\, L(\mathcal{T})\,] \approx \sum_{i=1}^{m} -\log_2 P(t_i \mid t_{i-1}) \tag{10}$$

where each term $-\log_2 P(t_i \mid t_{i-1})$ is the ideal code length for alphabet t_i given its context t_{i-1} under the current model [46]. The sum accumulates this cost over all m tokens in the sequence. This quantity corresponds to the empirical first-order conditional entropy $H_1(\mathcal{T})$ of the alphabet stream. In that sense, the scheme is asymptotically optimal among lossless coders that use only the immediately preceding token as context [43, 51]. In the limit, no coder restricted to the same order-1 context can consistently produce a shorter output on a stationary source. For short sequences, the bound is not exact. A small overhead is introduced by Laplace smoothing. This causes the actual code length to exceed the entropy bound by a bounded additive term. As the sequence becomes longer, that effect becomes smaller [34].

Taken together with the earlier pattern substitution stage, the pipeline uses redundancy at two levels. Recurring substrings are first replaced by compact tokens. Those tokens are then encoded according to their contextual probability. That combination is what makes the full pipeline effective. The encoded stream must still be stored in a form that the decoder can reconstruct exactly. For that reason, the output file also includes the supporting metadata and dictionary information.

3.4.4 Output File Format

The compressed file is self-contained. Everything required for exact reconstruction of the original sequence is stored in it. The arithmetic-coded bitstream is padded to byte boundaries and written to disk. Along with it, the hierarchical token dictionary is stored as an integer grammar. This grammar records the replacement rules that map each token identifier back to the substring it represents.

The token-to-integer mapping and the coder alphabet are also stored explicitly. This is needed because the decoder must rebuild the same alphabet ordering to reproduce the same probability estimates during decompression. Sequence headers and the positions of invalid characters detected during preprocessing are preserved as metadata. This ensures that the lossless decompression of the genome sequence is possible.

3.5 Query Search on the Tokenised Representation

The goal of this study is to use the shorter tokenised representation of the DNA sequence to perform operations like pattern matching and frequency count. Here we explore a simple way by which the tokenised representation can be used to find a pattern and also find its frequency in the DNA sequence. As the tokenised stream is significantly smaller, such operations need only a fraction of the time compared to naive search on the complete representation. The tokenised sequence $T = t_1 t_2 \dots t_m$ obtained at the end of Section 3.3, where each t_i is either a token or a base, can be directly used to find the presence and number of occurrences of a query sequence without full decompression.

Let $Q = q_1 q_2 \dots q_\ell$ be the query pattern of length ℓ. To search, we first find the tokens that cannot contain the query and skip them; for the remaining tokens, we use the dictionary to expand them as they appear and match against the query as shown in algorithm 2. This process uses two mechanisms, both operating directly on tokens.

Token filtering: A match of Q can only start at a base equal to the first character q_1 of the query. So, whenever no partial match is currently active, a token whose expansion does not contain q_1 anywhere cannot start a match and can be skipped without being expanded into bases. This allows the algorithm to search without expanding a significant portion of the sequence. When a partial match is active, subsequent tokens are expanded and matched as they appear.

Automaton-based matching: Tokens that are not filtered out must be scanned base by base. Rather than re-comparing a sliding window against Q at every base, matching is done with a Knuth–Morris–Pratt (KMP) automaton [25]. A failure function π is computed once for Q, in $O(\ell)$ time, before scanning begins [25]. A single integer state s, initialised to 0, tracks the length of the currently matched prefix of Q. For each base b read:

- while $s > 0$ and $b \neq q_{s+1}$, set $s \leftarrow \pi(s)$;
- if $b = q_{s+1}$, set $s \leftarrow s + 1$;
- if $s = \ell$, a match ends here; increment the count and set $s \leftarrow \pi(\ell)$.

This makes the automaton fall back to the correct shorter partial match on a mismatch, instead of discarding the match state entirely, which is necessary for patterns with self-overlap. This is common on a four-letter alphabet, since short DNA motifs frequently repeat internal bases.

Algorithm 2 Query Search on a Tokenised Sequence

Require: Tokenised sequence $T = t_1, \ldots, t_m$, dictionary D, query sequence Q
Ensure: Number of occurrences c of Q

```
Compute the KMP failure function π for Q
s ← 0,   c ← 0
for each token t_i in T do
    if s = 0 and t_i cannot contain Q[1] then
        continue
    end if
    Lazily expand t_i using D (cache the result on first use)
    for each base b in the expansion of t_i do
        while s > 0 and b ≠ Q[s + 1] do
            s ← π(s)
        end while
        if b = Q[s + 1] then
            s ← s + 1
        end if
        if s = |Q| then
            c ← c + 1
            s ← π(|Q|)
        end if
    end for
end for
return c
```

The method adds no new data structure beyond π, an array of size ℓ. It reuses the dictionary and lazy expansion cache already built during decompression, and each base is processed in $O(1)$ amortised time by the automaton, with no backtracking over the expanded sequence. The filtering step reduces the number of tokens that need to be expanded at all, and its benefit depends on how often q_1 occurs across the token set; tokens whose expansion never contains q_1 are skipped regardless of their length.

4 Results and Discussion

4.1 Dataset

To evaluate HiDAC, five diverse genome sets spanning multiple species are used. Table 1 describes the dataset used in our study. The first three datasets are used for general-purpose evaluation. They are used to examine sequence length reduction, file size reduction, and the ability of the learned dictionary to generalize across related genomic sequences. The remaining two datasets are standard benchmark datasets used for comparative assessment. The proposed approach was also compared with other state-of-the-art methods to evaluate its performance. All the results are acquired by compressing the files using the created dictionaries mentioned in the supplementary file.

4.2 Evaluation Criteria

For these datasets, we assess performance using two complementary metrics: sequence compression (SC) and space saved (SS). Sequence compression quantifies the percentage reduction in the length of the DNA sequence after token substitution and is defined as

$$SC = \left(1 - \frac{\text{Compressed sequence length}}{\text{Original sequence length}}\right) \times 100. \tag{11}$$

Table 1: Summary of datasets used for evaluation of HiDAC (Hierarchical Dictionary-Aided Compression)

Dataset	Description	Citation
DS_1	10 complete genome sequences of *Escherichia coli*, used to evaluate compression efficiency and dictionary generalisation across closely related bacterial strains.	[3, 5]
DS_2	10 complete genome sequences of *Mycobacterium tuberculosis*, used for assessing performance on pathogenic bacterial genomes with different structural characteristics.	[3, 11]
DS_3	High-quality human genome assembly (GCA_050656345.1) corresponding to the RPE1v1.1 haplotype 1 assembly of the diploid RPE-1 cell line, generated using long-read and chromosome conformation capture techniques, providing a phased representation of the human genome for accurate structural and sequence analysis.	[3, 49]
DS_4	A comprehensive and balanced benchmark dataset composed of 15 genomic sequences from diverse species, including: (1) chromosome 2 of *Gallus gallus* (GaGa), (2) chromosome 3 of *Danio rerio* (DaRe), (3) chromosome 1 of *Oryza sativa Japonica* (OrSa), (4) chromosome 2 of *Drosophila miranda* (DrMe), (5) chromosome 4 of the reference human genome (HoSa), (6) genome of *Entamoeba invadens* (EnIn), (7) genome of *Schizosaccharomyces pombe* (ScPo), (8) genome of *Plasmodium falciparum* (PlFa), (9) genome of *Escherichia coli* (EsCo), (10) genome of *Haloarcula hispanica* (HaHi), (11) genome of *Aeropyrum camini* (AeCa), (12) genome of *Helicobacter pylori* (HePy), (13) genome of Yellowstone Lake mimivirus (YeMi), (14) genome of *Aggregatibacter* phage S1249 (AgPh), and (15) genome of *Bundibugyo ebolavirus* (BuEb). This diversity ensures robust evaluation across organisms with varying genomic structures.	[40]
DS_5	Genome of *Escherichia coli* K-12 substr. MG1655 generated using the MinION sequencer, comprising 4.64 million bases, used to evaluate performance on real sequencing data.	[42]

Space saved measures the percentage reduction in storage size at the file level, capturing the combined effect of token substitution and entropy coding and is defined as

$$SS = \left(1 - \frac{\text{Compressed file size}}{\text{Original file size}}\right) \times 100. \tag{12}$$

Together, SC and SS provide a two-level view of compression: SC reflects how effectively the dictionary reduces structural redundancy in the sequence, while SS reflects the final storage benefit delivered to the user.

4.3 File Compression Results

For DS_1 and DS_2, one file from each dataset was selected as the reference genome to construct the dictionary. The same dictionary was then applied without modification to compress all remaining files in the dataset. This approach evaluates the generalisation capability of the constructed dictionary. This allows the captured information to be transferred to similar genomes of the ,same class. A consistent compression of nearly 76% and 76.8% was observed across for DS_1 and DS_2 respectively.

Tables 2 presents the SC and SS results for DS_1 and DS_2, respectively. HiDAC outperforms all existing general-purpose compressors as well as NAF across both datasets. Furthermore, the consistency of results across all files within each dataset support the generalisation capability of the constructed dictionary, as discussed in subection 3.2, confirming that the learned tokens capture characteristic recurring patterns of the genome class rather than being specific to the reference sequence.

Table 2: Compression percentages for various compressors across different genomic datasets.

Dataset	File	gzip (SS)	bzip2 (SS)	lzma (SS)	brotli (SS)	zstd (SS)	NAF (SS)	HiDAC (SS)	HiDAC (SC)
DS_1	GCF_000005845.2	70.64%	71.74%	73.15%	72.66%	68.39%	75.31%	**75.90%**	55.47%
	GCF_000008865.2	70.63%	71.71%	73.72%	73.27%	68.48%	75.33%	**75.90%**	55.47%
	GCF_000010385.1	70.66%	71.69%	73.24%	72.74%	68.38%	75.35%	**75.88%**	55.44%
	GCF_000013265.1	70.62%	71.69%	73.06%	72.53%	68.36%	75.31%	**75.89%**	55.47%
	GCF_000019645.1	70.65%	71.68%	73.15%	72.64%	68.36%	75.30%	**75.89%**	55.47%
	GCF_000210475.1	70.66%	71.72%	73.72%	73.15%	68.48%	75.35%	**75.89%**	55.47%
	GCF_002853715.1	70.64%	71.71%	73.13%	72.62%	68.37%	75.31%	**75.89%**	55.46%
	GCF_003018455.1	70.64%	71.71%	74.28%	73.83%	68.57%	75.32%	**75.87%**	55.46%
	GCF_003697165.2	70.62%	71.71%	73.07%	72.53%	68.38%	75.33%	**75.89%**	55.46%
	GCF_048568945.1	70.63%	71.70%	74.20%	73.74%	68.48%	75.35%	**75.87%**	55.45%
DS_2	GCA_000277735.2	71.41%	72.32%	73.87%	73.33%	69.17%	76.16%	**76.78%**	59.75%
	GCA_014899965.1	71.44%	72.34%	73.95%	73.40%	69.21%	76.17%	**76.77%**	59.75%
	GCA_014899985.1	71.43%	72.34%	73.91%	73.39%	69.21%	76.15%	**76.77%**	59.74%
	GCA_017901095.1	71.41%	72.33%	73.88%	73.33%	69.18%	76.14%	**76.77%**	59.75%
	GCA_030566105.1	71.52%	72.28%	73.87%	73.37%	69.24%	76.24%	**76.71%**	59.63%
	GCA_030566675.1	71.54%	72.30%	73.88%	73.37%	69.26%	76.23%	**76.72%**	59.64%
	GCA_033124885.1	71.45%	72.34%	73.96%	73.43%	69.23%	76.24%	**76.77%**	59.74%
	GCA_040208995.1	71.44%	72.35%	73.93%	73.43%	69.22%	76.13%	**76.77%**	59.74%
	GCA_963525475.1	71.48%	72.35%	73.96%	73.43%	69.23%	76.22%	**76.77%**	59.73%
	GCF_000195955.2	71.40%	72.33%	73.86%	73.33%	69.16%	76.16%	**76.77%**	59.75%
DS_3	GCA_050656345.1	69.85%	72.50%	75.01%	74.54%	67.76%	76.31%	**76.38%**	58.42%

For DS_3, which consists of the individual chromosomes of the human genome, we construct a separate dictionary for each chromosome independently. To obtain a single unified dictionary that generalises across the full genome, we retain only those tokens that appear in the dictionaries of at least 90% of the chromosomes. The resulting combined dictionary is then used to compress all chromosomes, allowing us to evaluate how well a genome-wide token vocabulary represents the full diversity of human genomic sequences. We recorded the SC and SS as defined above. Table 2 presents the SC and SS results for DS_3. We observe a 76.38% reduction in file size and 58.42% reduction in sequence size. This result further demonstrates that the combined dictionary constructed by retaining tokens shared across at least 90% of individual chromosome dictionaries, is capable of capturing not only local sequence-level patterns but also global recurring motifs that persist across the full human genome.

The Figure 2 represents the heatmap that showes the proposed method consistently achieves the highest compression across the three datasets, followed closely by NAF. Traditional compressors such as gzip and zstd exhibit lower performance, while lzma and brotli provide moderate improvements. The consistent darker shading for the proposed method across all datasets highlights its robustness and superior performance .

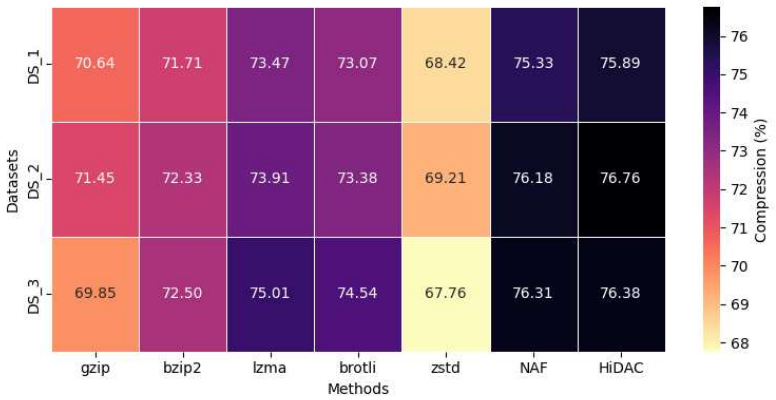


Figure 2: Heatmap illustrating compression performance (in %) of different methods across DS_1, DS_2, and DS_3 datasets, with darker shades indicating higher compression efficiency.

Table 3: Compression performance comparison (higher is better). Maximum value in each row is bold.

Dataset	File	gzip	bzip2	zstd	NAF	HMG	HiDAC
DS_4	AeCa	3.58	3.71	3.32	4.01	3.95	**4.12**
	AgPh	3.37	3.64	3.15	4.00	3.99	**4.01**
	BuEb	3.25	3.62	3.12	3.95	**4.09**	3.84
	DaRe	3.77	3.88	3.55	4.28	**4.47**	4.19
	DrMe	3.60	3.71	3.34	4.02	3.98	**4.08**
	EnIn	3.61	3.72	3.38	4.11	4.14	**4.18**
	EsCo	3.57	3.71	3.33	4.00	3.99	**4.10**
	GaGa	3.65	3.74	3.40	4.08	3.93	**4.19**
	HaHi	3.63	3.77	3.37	4.09	4.01	**4.20**
	HePy	3.67	3.77	3.42	4.10	**4.35**	4.22
	HoSa	3.73	3.89	3.50	**4.23**	4.04	4.19
	OrSa	3.65	3.74	3.41	4.05	**4.30**	4.08
	PlFa	3.77	3.82	3.50	4.27	4.22	**4.32**
	ScPo	3.57	3.69	3.33	4.07	3.80	**4.10**
	YeMi	3.59	3.79	3.33	4.21	4.28	**4.28**
DS_5	Ecoli	3.57	3.71	3.33	4.00	3.87	**4.15**

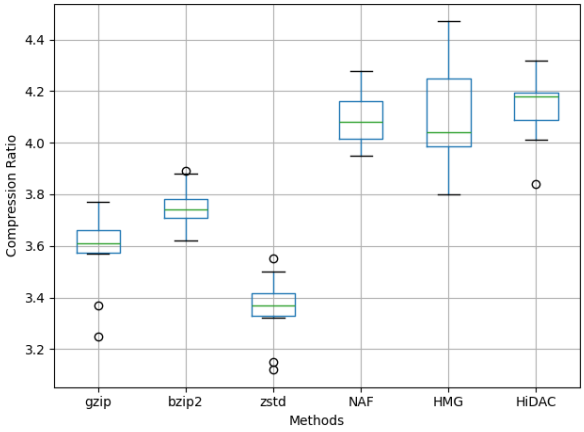


Figure 3: Box plot showing the compression ratio distributions of HMG and HiDAC across the DS_4 dataset, highlighting greater consistency in compression of the proposed approach across diverse genomic sequences.

For the benchmark datasets DS_4 and DS_5, we compare the proposed method against HMG (Heuristic-driven MDL-based Genome sequence compressor) [37], a recent state-of-the-art genomic compression method based on the Minimum Description Length (MDL) principle. HMG is designed to identify the optimal set of k-mers for maximal compression of a dataset, making it conceptually the closest existing method to our approach as described in Section 3.2. To ensure a fair comparison, we report the same metrics used by the authors of HMG: compression ratio (CR), defined as

$$CR = \frac{\text{Original file size}}{\text{Compressed file size}}. \tag{13}$$

A CR greater than 1 reflects greater storage reduction. Table 3 show the CR and SS results of DS_4 and DS_5 in comparison with HMG (Heuristic-driven MDL-based Genome sequence compressor) [37]. Note that we were not able to run the program on further datasets other than those made public by the authors, hence the comparison is limited to the sequences for which we have HMG findings. We chose the variation HMG-GA(CC/SM) for comparison since it is the strongest published configuration of the tool.

Figure 3 displays a box plot of the consistency of HMG and HiDAC on distinct genomic sequences. Both approaches produce competitive compression ratios, however the interquartile range and whiskers of HMG are larger, indicating a bigger fluctuation in the performance of HMG depending on genome type. The results for HiDAC, whereas standard are more closely grouped around the median, indicating more steady compression performance. The decreased spread indicates that HiDAC is capable of generating

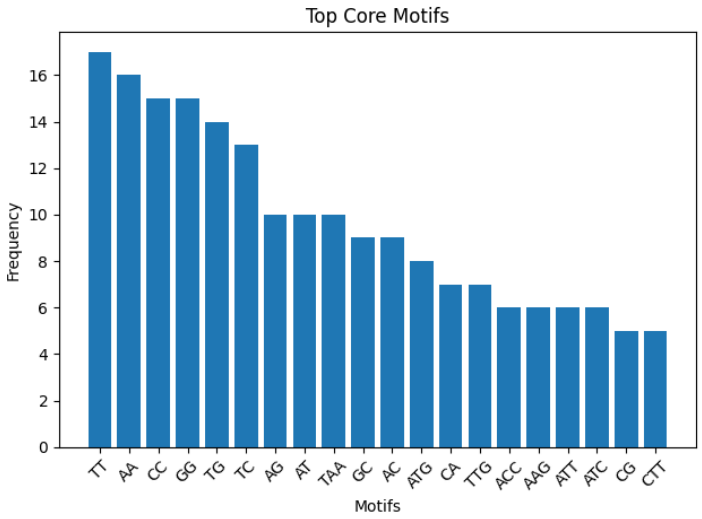


Figure 4: Frequency of the top 20 core motifs across the 17 constructed dictionaries. The frequency axis indicates the number of dictionaries in which each motif appears as a learned token.

consistent results on a diverse variety of genomic datasets, even if the sequences differ in length, structure and complexity. In a few settings, HMG has a slightly better, but in general the performance is less consistent. HiDAC delivers better consistency and resilience and is better suited for compressing diverse genomic data.

The detailed comprehensive results are provided in the supplementary Material. Tables S1-S5 show the file-level compression performance in terms of compressed size, and storage saved. The results for sequence-level compression with the tokenized representation created are shown in Tables S6–S10. To facilitate reproducibility, the dataset accession details, source code, and dictionaries used for compression are made publicly available through the GitHub repository referenced in the Supplementary Material.

To further characterize the learned representations, we analyzed the dictionaries created for each dataset given in Section 4.1. For each of the 17 genomes, a distinct dictionary was created, resulting in 17 different token vocabularies. To identify the most consistently learned patterns, we selected the 20 most frequent tokens across these dictionaries, and the resulting distribution is displayed in Figure 4. The distribution clearly indicates the dominance of dinucleotide patterns, with TT occurring in all 17 dictionaries, followed by AA in 15 and CC and GG in 14 of 17 dictionaries. The co-occurrence of GG with CC and AA with TT arises from the strand-compositional symmetry of genomic DNA, where the relative abundances of dinucleotides are roughly preserved between complementary strands [24]. Other frequently selected dinucleotides, notably TG and TC, occurring in 13 and 11 dictionaries, respectively, followed by AT in 10 dictionaries, also show that short compositionally biased patterns are reliably captured across different genomes. AG and GC occur in 9 dictionaries each, while AC, ATG, and CA occur in 8 dictionaries. This observation is consistent with the established role of short k-mer frequencies as genomic signatures that represent species-specific as well as conserved compositional properties [24]. Among the trinucleotide patterns, TAA is the most prevalent, occurring in 10 dictionaries, followed by ATG at 8 dictionaries and TTG at 7 dictionaries. AAG, ACC, ATC, and ATT each appear in 6 dictionaries. Several of these trinucleotide patterns correspond to codons, the three-base units used in protein-coding regions. The repeated occurrence of patterns such as TAA, ATG, and TTG in dictionaries constructed independently suggests that the method is identifying recurring sequence features across different genomes. Their presence is therefore unlikely to result solely from dataset-specific repetition or simple local redundancy [39].

To conclude, the frequency distribution from 17 to 5 shows a consistent decline demonstrating that the genomic sequences have structured and not random patterns. The change from shared dinucleotide motifs to less pervasive trinucleotides also demonstrates the hierarchical structure of genomic sequences. Shorter compositional motifs are more strongly conserved between species, while longer motifs are more strongly driven by genome composition, biological function and evolutionary history. This behaviour agrees with previous studies showing that k-mer distributions in genomic sequences are highly non-random and are shaped by nucleotide composition, functional constraints, and evolutionary processes [9, 24]. The recurrence of these core motifs across all 17 dictionaries, constructed from genomes with different

sizes and biological characteristics, demonstrates the generalization capability of HiDAC. Since k-mer frequency profiles have been widely used for alignment-free comparison and classification of genomes, even among distantly related organisms [47, 48], the presence of both short compositional motifs and biologically interpretable trinucleotides in the learned dictionaries confirms that the proposed method encodes characteristic recurring patterns of genomic sequences rather than patterns specific to any single reference genome.

4.4 Time Complexity Analysis

HiDAC consists of two main phases: dictionary construction, which identifies and encodes recurring patterns from the sequence, and compression, which applies the learned dictionary to encode the sequence into a compact binary representation. We analyse the time complexity of each phase in turn, followed by decompression and a comparison with existing methods.

4.4.1 Dictionary Construction

Algorithm 1 describes the dictionary construction procedure. Let N denote the length of the sequence used to construct the dictionary, and let I denote the maximum number of merge iterations. The algorithm first constructs the linked-list arrays `prev` and `next_`, and initializes the pair-tracking structures `occ`$[\pi]$, `freq`$[\pi]$, and `buckets`$[f]$. These steps require a single pass over the sequence and therefore cost $O(N)$.

In each iteration, the best pair is selected from `buckets`$[f_{\max}]$ rather than by scanning the entire sequence. If B_i pairs are tied at the maximum frequency, the selection step costs $O(B_i)$. This is because the entropy of each candidate pair is computed from `comp_cache` over the fixed DNA alphabet $\{A, T, G, C\}$, making the entropy evaluation constant time for each pair.

Let F_i be the number of recorded occurrences of the selected pair in iteration i. Since `occ`$[\pi]$ is stored as an unordered hash set, these positions are sorted before non-overlapping matches are selected. This sorting step costs $O(F_i \log F_i)$. The subsequent scan and linked-list updates cost $O(F_i)$, as each substitution modifies only a constant number of adjacent pairs. Therefore, the cost of iteration i is $O(B_i + F_i \log F_i + F_i)$, and the total cost of dictionary construction is $O\Big(N + \sum_{i=1}^{I} (B_i + F_i \log F_i + F_i)\Big)$. Using the bound $F_i \leq N$, a conservative worst-case complexity is $O(N + IN \log N)$, assuming that B_i remains small. In practice, I is fixed and the active sequence length decreases after each merge. Accordingly, the observed running time can be expressed as $O(N \log N)$ with respect to the input length N.

4.4.2 Compression

The compression phase combines dictionary-based pattern matching with entropy encoding. The Aho–Corasick automaton is constructed in $O(M)$ time, where $M = \sum_{k=1}^{K} |P_k|$ is the total expanded length of the K dictionary patterns. Once constructed, the automaton scans the input sequence in $O(N + Z)$ time, where Z denotes the total number of reported matches. The greedy replacement step also requires $O(N + Z)$ time, since reported matches are grouped by their starting positions and the longest, most frequent candidate is selected at each position.

After replacement, the resulting token sequence of length m is encoded using first-order context arithmetic coding. In the current implementation, cumulative intervals are computed by scanning the token alphabet. Consequently, arithmetic encoding takes $O(m \cdot A + R)$ time, where A is the size of the token alphabet and R is the number of emitted bits. The overall compression cost is therefore $O(M + N + Z + m \cdot A + R)$.

With the experimental limit set to 60 dictionary tokens, A can be treated as a small constant, while M remains much smaller than N. Under these conditions, the arithmetic coding stage runs effectively in linear time, and the overall compression process behaves approximately as $O(N + Z)$ in practice. The value of Z is generally observed to increase almost linearly with N. However, it cannot be strictly limited by the number of non-overlapping replacements. This is because the Aho-Corasick automaton detects overlapping occurrences and may identify several matching patterns at the same sequence position. Once the dictionary has been built, it can be reused to compress each remaining sequence with near-linear practical runtime.

4.4.3 Decompression

Once compression is complete, the decompression process reverses each stage in sequence. Let m denote the length of the decoded token stream and let N' denote the length of the reconstructed sequence. Arithmetic decoding with first-order context modelling processes the encoded token stream in a single pass. In the current implementation, cumulative intervals are computed by scanning the token alphabet; therefore, arithmetic decoding requires $O(m \cdot A + R)$ time, where A is the token alphabet size and R is the number of bits read from the encoded bitstream. Since A is bounded by $4 + I$ and remains small under the experimental cap of 60 dictionary tokens, this step reduces to $O(m + R)$ in practice.

Grammar expansion is performed using a memoised traversal of the hierarchical grammar rules, ensuring that each rule is expanded at most once. The total cost of expanding all decoded tokens is therefore proportional to the length of the reconstructed sequence, giving $O(N')$ time. Reinsertion of invalid character runs and other preserved metadata is likewise linear in the output size. Hence, under the bounded-alphabet setting used in the experiments, the overall time complexity becomes $O(m + R + N')$, and since the decompressor must produce N' output characters, the decompression time is linear in the reconstructed sequence size in practice.

4.4.4 Comparison with Existing Methods

Widely used general-purpose compression tools differ considerably in their computational complexity. gzip uses the DEFLATE algorithm [14], which combines LZ77[54] with Huffman coding [22]. Because it relies on a fixed-size sliding window and a bounded matching process, its running time is linear $\Theta(N)$. This allows gzip to provide fast compression and decompression, although it is less effective at detecting long-range patterns in genomic sequences. bzip2, in contrast, is based on the Burrows–Wheeler Transform [6]. Constructing this transform generally requires $O(N \log N)$ time for each block, making bzip2 more computationally expensive than gzip. However, the additional processing often results in better compression. NAF-based genomic compressors [28] use approximate factorization together with sliding-window similarity searches and heuristic techniques. Their theoretical complexity may be superlinear, but in practice, bounded window sizes and aggressive pruning allow them to achieve runtimes that are close to linear. In the conservative worst case, HiDAC requires $O(N + I \cdot N \log N)$ time to construct the dictionary. Under the experimental limit of $I \leq 60$ tokens, this becomes $O(N \log N)$. Once the dictionary has been created, each additional sequence can be compressed or decompressed in linear time, $O(N)$. A key feature of HiDAC is that the dictionary is built only once from a representative reference sequence and is then reused for all sequences in the dataset. As a result, the cost of dictionary construction is distributed across the entire collection. This provides a practical balance between computational cost and compression performance, while also producing a compact and reusable dictionary that general-purpose compression tools do not provide.

4.5 Results of Compressed Domain Analysis

To compare the efficiency of the tokenised representation in performing pattern search and frequency count, we compared the search algorithm for tokenised sequence proposed in Section 3.5 with naive search on the raw sequence. Figure 5 shows the time taken to find the frequency of a query of length between 13–466 in two genome sequences. The first one is the genome for Chromosome 2 of *Drosophila miranda* (DrMe) with raw size 3,21,81,429 and tokenised length of 1,44,92,463 characters. The second is the Chromosome 4 of the human genome (HoSa) with raw size of 18,97,52,667 and tokenised length of 9,77,83,859 characters. This corresponds to a sequence length reduction of 54.97% for DrMe and 48.47% for HoSa.

Across both genomes, the tokenised search was consistently faster than the raw sequence search. For DrMe, the mean search time was 4.67 s (SD = 0.28) for the raw sequence and 1.79 s (SD = 0.14) for the tokenised representation, giving a mean speedup of 2.62× (median 2.61×, range 2.29×–2.92×). For HoSa, the mean search time was 25.26 s (SD = 2.35) for the raw sequence and 9.22 s (SD = 1.11) for the tokenised representation, giving a mean speedup of 2.76× (median 2.74×, range 2.30×–3.27×). Both raw and tokenised search times increase slightly as query length increases, but the increase is much smaller for the tokenised search, so the speedup is maintained across the full range of query lengths tested.

This is clear evidence that HiDAC is not only able to provide good compression results but also make pattern matching and frequency count a lot faster using the semi-decompressed (tokenised) representation. The main reason why the search algorithm proposed works so efficiently is because of the cost function used to create the dictionary. The function ensures that the dictionary is balanced as much as

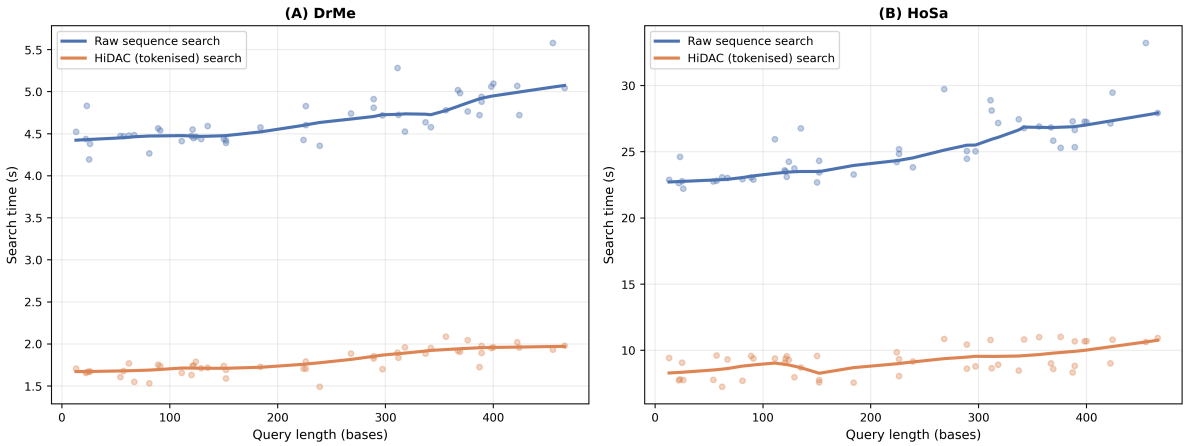


Figure 5: Query search time as a function of query length (13–466 bases, n = 49 queries) for raw-sequence search versus HiDAC tokenised search, evaluated on two genomes: (A) DrMe and (B) HoSa. Scatter points show individual query measurements; solid lines show LOWESS-smoothed trends. In DrMe, mean search time was 4.67 s (SD = 0.28) for the raw sequence versus 1.79 s (SD = 0.14) for the tokenised representation, corresponding to a mean speedup of 2.62× (SD = 0.15, median = 2.61×, range 2.29×–2.92×). In HoSa, mean search time was 25.26 s (SD = 2.35) for the raw sequence versus 9.22 s (SD = 1.11) for the tokenised representation, a mean speedup of 2.76× (SD = 0.29, median = 2.74×, range 2.30×–3.27×). Both raw-sequence and tokenised search times increase mildly with query length, with the tokenised representation maintaining a consistent 2.6–2.8× advantage across the tested range and across genomes of different composition.

possible, therefore even when a single token is ignored during search it has a significant effect on time, as it has strong representation in the sequence. Because each token typically covers many bases and occurs frequently, skipping one token during the first-character check removes a large part of the sequence from consideration in a single step, instead of the one-character-at-a-time progress made when scanning the raw sequence. Also, once a token has been expanded and checked, the result is reused every time that same token appears again, so repeated tokens are not processed more than once. This is consistent with both genomes showing a similar speedup despite differing considerably in size and repeat structure, which suggests the improvement comes from the token-level design of the dictionary rather than being specific to one genome.

5 Conclusion

A two-stage DNA compression framework HiDAC (Hierarchical Dictionary-Aided Compression) is presented to address the limitations of existing genomic compressors. In the first stage, a hierarchical dictionary-based representation of the sequence is constructed. In the second stage, context-modelled arithmetic coding is applied to the resulting token stream. As shown in Section 4, the proposed method achieves better compression than general-purpose compressors and also outperforms NAF across all tested datasets. A comparison with HMG, which is the closest method in terms of design, also shows improved compression ratios. The gain is obtained within a similar pattern-based encoding framework, so the comparison remains fair.

An important aspect of the method is that the hierarchical token representation is not only used for compression. It also provides a structured and interpretable view of the genome. Because of this, some analysis tasks can be carried out directly in the compressed domain, without full decompression. This separates the method from most conventional genomic compressors, where compression and analysis are usually treated as different steps. This was confirmed experimentally in Section 4.5, where the tokenised representation was used to perform pattern search and frequency counting directly on the tokenised sequence. Across query lengths ranging from 13 to 466 bases, the tokenised search was consistently faster than search on the raw sequence.

Additional improvement is still possible. In the current study, the cost function was designed using an empirical approach, and more careful refinement of this design may further improve compression performance. Another possible direction is to investigate higher-order context models in the arithmetic

coding stage. Further, better techniques can be used to improve the pattern matching in the tokenised domain which is essentially a grammar.

References

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Behshad Behzadi and Fabrice Le Fessant. Dna compression challenge revisited: a dynamic programming approach. In *Proceedings of the 16th Annual Conference on Combinatorial Pattern Matching*, CPM'05, page 190–200, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and Eric W. Sayers. Genbank. *Nucleic Acids Research*, 37(suppl_1):D26–D31, 01 2009.

[4] Philip Bille, Inge Li Gørtz, and Nicola Prezza. Practical and effective re-pair compression, 2017.

[5] Frederick R Blattner, Guy Plunkett III, Craig A Bloch, Nicole T Perna, Valerie Burland, Monica Riley, Julio Collado-Vides, Jeremy D Glasner, Christopher K Rode, George F Mayhew, et al. The complete genome sequence of escherichia coli k-12. *science*, 277(5331):1453–1462, 1997.

[6] Michael Burrows, D J Wheeler D I G I T A L, Robert W. Taylor, David J. Wheeler, and David Wheeler. A block-sorting lossless data compression algorithm. 1994.

[7] Xin Chen, Sam Kwong, and Ming Li. A compression algorithm for dna sequences and its applications in genome comparison. *Genome informatics. Workshop on Genome Informatics*, 10:51–61, 02 1999.

[8] Xin Chen, Ming Li, Bin ma, and John Tromp. Dnacompress: Fast and effective dna sequence compression. *Bioinformatics (Oxford, England)*, 18:1696–8, 01 2003.

[9] Benny Chor, David Horn, Nati Goldman, Yaron Levy, and Tim Massingham. Genomic dna k-mer spectra: models and modalities. *Genome Biology*, 10(10):R108, 2009.

[10] John Cleary and H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32, 03 2003.

[11] Stewart Cole, R Brosch, Julianne Parkhill, T Garnier, Carol Churcher, D. Harris, S.V. Gordon, K Eiglmeier, S Gas, Clifton Barry, Fredj Tekaia, K Badcock, Daryl Basham, D Brown, T Chillingworth, R Connor, R Davies, K Devlin, T Feltwell, and B. Barrell. Deciphering the biology of mycobacterium tuberculosis from the complete genomic sequence. *Nature*, 393:537–44, 07 1998.

[12] Sebastian Deorowicz, Agnieszka Danek, and Marcin Niemiec. GDC 2: Compression of large collections of genomes. *CoRR*, abs/1503.01624, 2015.

[13] Sebastian Deorowicz and Szymon Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 09 2011.

[14] Peter Deutsch. Deflate compressed data format specification version 1.3. RFC 1951, 1996.

[15] Rosario Gilmary, Akila Gopu, and V. Govindasamy. Compression techniques for dna sequences: A thematic review. *Journal of Computing Science and Engineering*, 15:59–71, 06 2021.

[16] Sara Goodwin, John D McPherson, and W Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nat. Rev. Genet.*, 17(6):333–351, June 2016.

[17] S. Grumbach and F. Tahi. Compression of dna sequences. In *[Proceedings] DCC '93: Data Compression Conference*, pages 340–350, 1993.

[18] Stéphane Grumbach and Fariza Tahi. A new challenge for compression algorithms: Genetic sequences. *Information Processing & Management*, 30(6):875–886, 1994.

[19] Faraz Hach, Ibrahim Numanagić, and S Cenk Sahinalp. DeeZ: reference-based compression by local assembly. *Nat. Methods*, 11(11):1082–1084, November 2014.

[20] Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, and Ewan Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res.*, 21(5):734–740, May 2011.

[21] Zhi-An Huang, Zhenkun Wen, Qingjin Deng, Ying Chu, Yiwen Sun, and Zexuan Zhu. LW-FQZip 2: a parallelized reference-based compression of FASTQ files. *BMC Bioinformatics*, 18(1):179, March 2017.

[22] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[23] Daniel C. Jones, Walter L. Ruzzo, Xinxia Peng, and Michael G. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Research*, 40(22):e171–e171, 08 2012.

[24] Samuel Karlin, Jan Mrázek, and A. M. Campbell. Compositional biases of bacterial genomes and evolutionary implications. *Journal of Bacteriology*, 179(12):3899–3913, 1997.

[25] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[26] Gergely Korodi, Ioan Tabus, Jorma Rissanen, and J. Astola. Dna sequence compression - based on the normalized maximum likelihood model. *Signal Processing Magazine, IEEE*, 24:47 – 53, 02 2007.

[27] Gergely Korodi, Ioan Tabus, Jorma Rissanen, and J. Astola. DNA sequence compression based on the normalized maximum likelihood model. *IEEE Signal Processing Magazine*, 24:47–53, 2007.

[28] Kirill Kryukov, Mahoko Takahashi Ueda, So Nakagawa, and Tadashi Imanishi. Nucleotide archival format (naf) enables efficient lossless reference-free compression of dna sequences. *Bioinformatics*, 35(19):3826–3828, 02 2019.

[29] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 08 2009.

[30] Yulan Liang and Arpad Kelemen. Sequential support vector regression with embedded entropy for SNP selection and disease classification. *Stat. Anal. Data Min.*, 4(3):301–312, June 2011.

[31] Yuansheng Liu, Hui Peng, Limsoon Wong, and Jinyan Li. High-speed and high-ratio referential genome compression. *Bioinformatics*, 33(21):3364–3372, 06 2017.

[32] David Loewenstern and Peter Yianilos. Significantly lower entropy estimates for natural DNA sequences. *Journal of Computational Biology*, 6:125–142, 1999.

[33] Jean loup Gailly and Mark Adler. gzip: Gnu zip compression utility. `https://www.gnu.org/software/gzip/`, 1992. GNU Project.

[34] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[35] Takuya Masaki and Takuya Kida. Online grammar transformation based on re-pair algorithm. In *2016 Data Compression Conference (DCC)*, pages 349–358, 2016.

[36] Aleksandr Morgulis, E Michael Gertz, Alejandro A Schäffer, and Richa Agarwala. A fast and symmetric dust implementation to mask low-complexity dna sequences. *Journal of computational biology*, 13(5):1028–1040, 2006.

[37] M. Zohaib Nawaz, M. Saqib Nawaz, Philippe Fournier-Viger, Shoaib Nawaz, Jerry Chun-Wei Lin, and Vincent S. Tseng. Efficient genome sequence compression via the fusion of mdl-based heuristics. *Information Fusion*, 120:103083, 2025.

[38] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2 and 3):103–116, 1997.

[39] Joshua B Plotkin and Grzegorz Kudla. Synonymous but not the same: the causes and consequences of codon bias. *Nature Reviews Genetics*, 12(1):32–42, 2011.

[40] Diogo Pratas and Armando Pinho. A dna sequence corpus for compression benchmark. pages 208–215, 06 2018.

[41] Diogo Pratas, Armando Pinho, and Paulo Ferreira. Efficient compression of genomic sequences. 03 2016.

[42] Joshua Quick, Aaron R Quinlan, and Nicholas J Loman. A reference bacterial genome dataset generated on the minion™ portable single-molecule nanopore sequencer. *GigaScience*, 3(1):2047–217X–3–22, 10 2014.

[43] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979.

[44] Subrata Saha and Sanguthevar Rajasekaran. Ergc: an efficient referential genome compression algorithm. *Bioinformatics*, 31(21):3468–3475, 07 2015.

[45] Julian Seward. bzip2 and libbzip2. `https://sourceware.org/bzip2/`, 1996.

[46] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[47] Hanno Teeling, Jörg Waldmann, Timo Lombardot, Michael Bauer, and Frank Oliver Glöckner. Tetra: a web-service and a stand-alone program for the analysis and comparison of tetranucleotide usage patterns in dna sequences. *BMC Bioinformatics*, 5(1):163, 2004.

[48] Susana Vinga and Jonas Almeida. Alignment-free sequence comparison—a review. *Bioinformatics*, 19(4):513–523, 2003.

[49] Emilia Volpe, Alessio Colantoni, Luca Corda, Elena Di Tommaso, Franca Pelliccia, Riccardo Ottalevi, Andrea Guarracino, Danilo Licastro, Luigi Faino, Mattia Capulli, Giulio Formenti, Evelyne Tassone, and Simona Giunta. The reference genome of the human diploid cell line rpe-1. *Nature Communications*, 16, 09 2025.

[50] Linqi Wang, Renpeng Ding, Shixu He, Qinyu Wang, and Yan Zhou. A pipeline for constructing reference genomes for large cohort-specific metagenome compression. *Microorganisms*, 11(10), 2023.

[51] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30:520–, 01 1987.

[52] John C Wootton and Scott Federhen. Statistics of local complexity in amino acid sequences and sequence databases. *Computers & chemistry*, 17(2):149–163, 1993.

[53] Haichang Yao, Yimu Ji, Kui Li, Shangdong Liu, Jing He, and Ruchuan Wang. HRCM: An efficient hybrid referential compression method for genomic big data. *Biomed Res. Int.*, 2019:3108950, November 2019.

[54] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.