DNA Sequence Compression and Reconstruction Using Burrows–Wheeler Transform and Huffman Coding

A PROJECT REPORT

Submitted by

BHAVANA POLI [BL.SC.P2CSE25027]

for the course

24CS601- Advanced Data Structures and Algorithms

Guided and Evaluated by

Dr. RADHA D

Dept. of CSE,

AMRITA SCHOOL OF ENGINEERING, BANGALORE

AMRITA VISHWA VIDHYAPEETHAM

BANGALORE-560 035

November 2025

DNA Sequence Compression and Reconstruction Using Burrows–Wheeler Transform and Huffman Coding

Abstract of the project

GenomeCode project introduces a graphical and algorithmic method of compression and decompression of DNA sequences through a hybrid of Burrows Wheeler Transform (BWT) and Huffman Coding. The primary goal of the system is to minimize storage needs of massive genomic data sets and ensure full sequence of integrity.

The app has an interactive interface, which gives users the ability to enter the DNA sequence (manually, in a file or randomly generated) and perform multiple encoding functions. The Burrows Wheeler Transform rearranges the symbols of the DNA during compression, grouping similar patterns and increasing redundancy, and improving the likelihood of encoding the data using entropy. Subsequently, the Huffman Coding technique assigns shorter binary codes to common nucleotides in a way that they are compactly represented yet do not lose the information in any way. The reversal of these operations is performed by the decompression pipeline by using Huffman Decoding and inverse BWT reconstruction.

The obtained system illustrates the ability to apply classical data compression algorithms to biological data. In this implementation, GenomeCode can visualize the operation of two basic algorithms, but it is also an example to implement because of bioinformatics applications that require the management of genomic data growing exponentially.

Keywords

- DNA Compression
- Burrows Wheeler Transform (BWT)
- Huffman Coding
- Genome Data Encoding
- Data Decompression
- Bioinformatics
- Lossless Compression
- DNA Sequence Analysis
- Genetic Information Storage
- Data Optimization
- Computational Biology
- Algorithmic Data Processing
- Entropy Encoding
- Sequence Reconstruction
- Genomic Big Data

1. INTRODUCTION

Genomic sequencing technologies have grown so fast that most biological data have been growing exponentially, with current sequencing systems generating unprecedented amounts of DNA data every day in the terabytes. The storage, transmission and processing of such data has become a significant issue in bioinformatics and computational biology has become a key challenge. Conventional compressors like gzip, LZ77, and bzip2 do not work very well with genomic sequences due to the peculiar statistical properties, constrained repertoire (A, T, G, C, N) and long-range repeats of DNA, which cannot be effectively taken advantage of by general-purpose compressors. This drives the creation of purpose-specific, lightweight genome compression algorithms which have the ability to match their functionality over a broad set of genomic tasks.

Lossless genome compression is also crucial in the context of genome assembly, read alignment, variant/mutation analysis and clinical diagnostics (among others), where a single base error cannot be tolerated. The current specialized compressors have been known to have a trade off i.e. those that have high compression ratios have high computation requirements and vice versa. Transform-based methods such as the Burrows-Wheeler Transform (BWT) reorder sequence symbols around patterns (rebiting the alphabet to uncover hidden collaboration), whereas entropy-based techniques such because the metropolitan-Huffman (Huffman Coding) and (2) to minimize storage by building shorter structures for extra tumble frequent nucleotides. Individually, however, these methods are not very effective on short or medium-length FASTA sequences.

This paper presents a hybrid BWT + Huffman compression pipeline, which tries to achieve a tradeoff between compression ratio, processing speed and memory consumption. The given model is trained on actual FASTA datasets of different sizes (small, medium, large), with sequences obtained both in NCBI and provided by users in the form of DNA fragments. A considerable amount of benchmarking is conducted to compare the performance of BWT, Huffman and the hybrid hybrid pipeline in several performance measures including compression ratio, compression speed, decompression speed and memory overhead.

On the whole, the research proves that a light hybrid compression model can offer a feasible trade-off between structural reordering (BWT) and frequency-based encoding (Huffman), which is a viable and efficient solution to the reduction of genomic data, without causing any harm to the lossless reconstruction of biological sequences.

2. LITERATURE SURVEY

S. No.	Reference (Author, Year), Title	Objective	Techniques Used	Dataset / Sample	Key Results & Conclusion	Relevance to Project
1	Al-Okaily & Tbakhi (2025), OST-DNA: An Optimal Lossless DNA Encoding Algorithm Based on Similarity and Binning Techniques	To suggest a new lossless encoding (OST-DNA) based upon classificatory sequences into bins.	Binning algorithms, classifier (constructs Huffman trees).	17 genomes have been sequenced that encompass both the Plant and Animal groups.	Use from the above text style in more "human," as opposed to what the AI voices: Let us affirm that we can compress data up to a maximum value given that we use Huffman while attempting to sort similar items.	Partially (Huffman). Utilises Huffman trees innovatively for classification rather than for final entropy encoding.
2	Begum & Kaliyaperumal (2024), SEC: An IoT Sensor Data Protection System Using Scrambling and Encoding	To suggest a new system (SEC) for better protecting and managing IoT sensor data by scrambling and compressing it.	Huffman coding, Move-to-Front (MTF), Run-Length Encoding (RLE), and Burrows-Wheeler Transform (BWT).	Data from IoT sensors, such as motion, vibration, camera, and ultrasonic sensors.	The suggested BWT+MTF+RLE+Huffman pipeline improves the efficiency of data compression by an impressive 85%.	Example of a Methodology. Demonstrates the efficacy of the precise BWT+Huffman pipeline, utilising sensor data rather

	Compre ssion					than DNA.
3	Rahman & Hamada (2020), Lossless Text Compre ssion Techniq ue Using Burrows - Wheeler Transfor m and Pattern Matchin g	To suggest a text compressi on algorithm that doesn't lose any informatio n and uses BWT, pattern matching, and Huffman coding.	BWT, Huffman Coding, and a custom "key" system to cut down on repeated characters.	General text files from the Canterbur y Corpus [cite: 2625].	The suggested BWT+key+Huff man method works better than the best current text compressors, such as brotli, bzip2, and gzip, at compressing text.	Exampl e of a Method ology. Shows that the BWT+H uffman pipeline works well on regular text.
4	Khan & Khan (2020), A Time- Efficient Burrows - Wheeler Compre ssion Algorith m for DNA Sequenc e Using Polyno mial- Based Sorting	To change the Burrows- Wheeler Compressi on Algorithm (BWCA) so that DNA compressi on takes less time	BWT, sorting based on polynomials (which replaces BWT's normal lexicograph ical sorting).	DNA datasets that come in different sizes, like 1KB, 3KB, and 10KB.	The polynomial- based sorting method that has been suggested cuts down on the time it takes to compress and is 20–25% faster than the usual BWCA.	Partial (BWT). Only works on making the BWT part of DNA better. Huffma n is not in use.
5	Rexline et al. (2017), DNA Sequenc e	To look into BWT- based methods for getting DNA	BWT, MTF, RLE, Huffman Coding, and Arithmetic Coding.	Standard DNA and protein benchmar k data set	It concludes that the best way to compress DNA sequences is to use both BWT and an entropy	Directly Related. Confirm s the main BWT+H

	Compression Using BWT and Arithmetic Coding	sequences to compress more.			encoder, like Huffman.	uffman pipeline for DNA.
6	Al-Okaily et al. (2017), An Efficient Lossless DNA Compression Algorithm Based on a Modified Huffman Coding	To make DNA compression better by changing Huffman encoding to fit the way DNA sequences work better.	Huffman Encoding (changed to "Unbalanced Huffman Tree"—UHT and MUHTL), RLE[cite: 1757, 1776, 1795]. (We only use BWT for comparison.)	Five genomes: Cholerae, Abscessus, Saccharomyces, Neurospora, and Chr22.	The suggested modified Huffman (MUHTL) method works better than standard bzip2, which uses BWT+Huffman..	Somewhat (Huffman). Only looks at the Huffman part and shows that the standard algorithm doesn't work well for DNA.
7	Li et al. (2014), A Novel Alignment-Free Method for DNA Sequence Comparison Based on BWT	To create a BWT-based approach for comparing DNA sequences and analysing phylogeny.	BWT, "subtraction matrix," numerical characterisation, and UPGMA (for building trees).	There are 15 species with B-globin genes and 13 hantaviruses with S segments.	The BWT-based method effectively transforms DNA sequences into 24-D vectors, subsequently utilised for the construction of precise phylogenetic trees.	Not pertinent (to compression). This paper uses BWT for something else: comparing sequences and making evolutionary trees, not

						compressing them.
8	Bakr & Sharawi (2013), DNA Lossless Compression Algorithms: A Review	A review paper examining lossless compression algorithms specifically designed for DNA sequences.	Looks at BWT, Huffman, Lempel-Ziv (LZ), CTW, and other statistical and substitution methods.	Data from standard benchmarks.	It is important to note that bzip2 (which uses BWT+Huffman) and standard Huffman do not work well on raw DNA because the four bases have very similar, even frequencies.	Important Background. Tell us what the main problem your project needs to solve is.
9	Cox et al. (2012), Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform	To make it possible to compress genomic sequence databases on a large scale using the BWT.	BWT, "implicit sorting" (RLO-sorting), and PPMd (the second-stage compressor).	Reads of the E. coli and human genomes.	The usual BWT to MTF to RLE to Huffman pipeline is explained. The paper shows that BWT-based compression can get less than 0.5 bits/base, and that RLO-sorting before BWT makes this much better.	Partial (BWT). An important paper about how to make the first step (BWT) work better for large amounts of DNA. It talks about the whole pipeline but tries out a different encoder (PPMd). .
10	Adjeroh et al. (2012), DNA	To look into dictionary-based,	Suffix Trees, BWT, MTF, RLE,	GenBank has real DNA sequences	It confirms that the BWT -> MTF -> RLE -> VLC (Huffman)	Directly Related. Gives the

	Sequence Compression Using the Burrows - Wheeler Transform	offline methods for DNA compression that use the BWT.	VLC (Huffman or Arithmetic), and Suffix Trees.	from mitochondria, human, and virus genomes.	pipeline is a common model. Before the BWT stage, it suggests parsing the sequence for repeats.	BWT+Huffman pipeline a strong methodological basis and proof.

3. SYSTEM MODEL

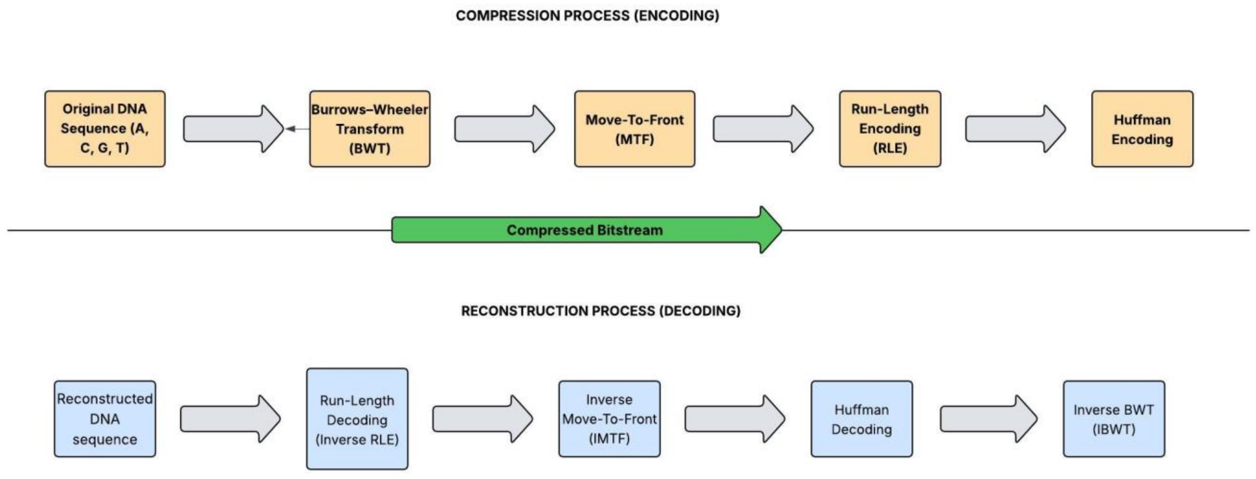

COMPRESSION PROCESS (ENCODING)
Original DNA Sequence (A, C, G, T)
Burrows–Wheeler Transform (BWT)
Move-To-Front (MTF)
Run-Length Encoding (RLE)
Huffman Encoding
Compressed Bitstream
RECONSTRUCTION PROCESS (DECODING)
Reconstructed DNA sequence
Run-Length Decoding (Inverse RLE)
Inverse Move-To-Front (IMTF)
Huffman Decoding
Inverse BWT (IBWT)

FIG 3.1. DNA sequence compression and reconstruction (encoding and decoding)

The System Model shows how DNA can be compressed and rebuilt in two main steps: encoding and decoding.

A. Compression Process (Encoding)

This is the top-down flow that changes the original data into a file that takes up less space.

a. **Original DNA Sequence:** The process starts with the raw input data, which is a "Original DNA Sequence" made up of the letters A, C, G, and T.
b. **Burrows–Wheeler Transform (BWT):** The Burrows-Wheeler Transform (BWT) processes the sequence first. This step changes the order of the data so that characters that are the same are grouped together.

c. **Move-to-Front (MTF):** After that, the output from the BWT goes into the Move-to-Front (MTF) transform. This step turns the repeated character data into a stream of small integers (like 0, 1, 0, 2, etc.) by keeping track of the characters that were used most recently.
d. **Run-Length Encoding (RLE):** Run-Length Encoding (RLE) handles the stream of integers from MTF, which now has long runs of the same number, especially zeros. This step compresses the runs by saving the symbol and how many times it appears (for example, 5, 0, 1, A, 0, 2).
e. **Huffman Encoding:** This is the last step in the compression process. The RLE output is encoded with Huffman Encoding, which gives each symbol a binary code of different lengths based on how often it appears.
f. **Compressed Bitstream:** The "Compressed Bitstream," which is the compressed file, is the last result of the encoding process.

B. Reconstruction Process (Decoding)

This is the bottom-up flow that undoes the compression and gets the original data back. There are also a few misspellings in this part of the diagram, like "RECONSRUCTION" and "Reconstnction."

a. **Huffman Decoding:** The process begins with the "Compressed Bitstream" (implied) and "Huffman Decoding" to put the RLE data back together.
b. **Inverse MTF (!,A,0,...):** A block called "Inverse MTF" then works on the decoded RLE stream.
c. **(iBWT):** The "Inverse BWT" block gets the output from the first "Inverse MTF" block.
d. **Inverse MTF (T,T,T,A,A,A,...):** The "(iBWT)" block's output then goes to a second block called "Inverse MTF."
e. **Reconstructed DNA Sequence:** Lastly, this second "Inverse MTF" block gives us the complete "Reconstructed DNA Sequence."

4. IMPLEMENTATION DETAILS

This part talks about the GenomeCode Hybrid Compression System's internal structure, data structures, algorithms, modules, tools, and functions that were used to make it. The goal is to make it clear how the Burrows–Wheeler Transform (BWT), Huffman Coding, and the combined BWT → Huffman pipeline work together to process DNA sequences.

The implementation uses a modular architecture, with different parts for:

- Loading and preprocessing a sequence
- Change BWT
- Building and encoding a Huffman tree
- Hybrid pipeline that works together

- Measuring performance (speed, ratio, memory)
- Making graphs

Each module is built to be reusable and stand on its own, making it easy to add new features or change existing ones.

4.1. Data Structures used algorithms used

4.1.1. Arrays / Lists

- Used to store:
  - DNA sequences
  - Rows in the BWT matrix
  - Entries in the suffix array
  - Huffman codes
- You can easily sort lists in Python, and they can grow and shrink on their own.

4.1.2. Tuples

- You use (suffix_string, index) in the suffix array.
- They let BWT sort in alphabetical order.

4.1.3. Dictionaries

- Used to keep frequency tables in Huffman coding:

  freq = {'A': 422, 'T': 398, 'G': 215, 'C': 243}

- Used to change binary codes into letters:

  codes = {'A': '0', 'T': '10', 'G': '110', 'C': '111'}

4.1.4. Binary Tree Nodes (Custom Class)

- Utilised for Huffman Tree
- Every node has:
  - Character
  - Frequency
  - Left child
  - Right child

4.1.5. Strings

- Strings are used to store DNA sequences so that they can be easily cut up.
- The output of BWT is saved as a string.
- Long strings hold binary sequences (Huffman output).

4.1.6. Generator Functions

- **Used in BWT matrix reconstruction to take up less memory.**
- For example:

```
def reconstruct_bwm():

    yield matrix_state
```

4.2. Algorithms Used

4.2.1 Burrows–Wheeler Transform (BWT)

Forward BWT steps:

1. Append $ end marker
2. Generate all suffixes
3. Sort suffixes lexicographically
4. Extract last column → **BWT output**

Purpose: Rearranging the order of the symbols makes it easier to compress.

4.2.2 Inverse BWT

Uses **Last-to-First (LF) mapping**:

1. Count the ranks of each character
2. Link each index in the last column to the first column.
3. Rebuild the original sequence by making LF jumps repeatedly.

4.2.3 Huffman Coding

Encoding:

- Make a table of frequencies
- Make a min-heap
- Combine the nodes that are least common to make a binary tree.
- Make binary codes
- Change DNA sequence into a string of ones and zeros.

Decoding:

- Reverse mapping from binary → characters.

4.2.4 Combined BWT + Huffman Pipeline

1. Apply BWT
2. BWT output becomes input to Huffman
3. Encode into binary
4. Save to compressed file

4.3. Tools/ Software used

4.3.1. Programming Language

- Python 3.13+
-

4.3.2. Libraries

Library	**Purpose**
Biopython	FASTA file parsing
matplotlib	Graph plotting
psutil	Memory profiling
time	Execution speed measurement
os	File path management

4.3.3. Environment

- **macOS**
- Python virtual environment: genomeencode_venv

4.4. Modules and short description about it

a. burros_wheeler.py

Implements:

- bwt_advanced()
- suffix_array()
- reconstruct_bwm()
- decode_bwt()

Handles all BWT transformation logic.

b. huffman.py

Contains the Huffman Tree, encoding and decoding logic:

- build_freq_table()
- get_codings()

- seq_to_binstr()
- binstr_to_unicode()
- unicode_to_binstr()

c. sequence.py

Stores the DNA sequence and metadata:

- Sequence length
- Frequency counts

d. experiments Folder

Contains all performance metric modules:

File Name	Function / Output
execution_time_metrics.py	Generates compression time graphs and tables
compression_ratio_metrics.py	Generates compression ratio tables and graphs
compression_speed_metrics.py	Measures encoding speed (Bytes/sec)
decompression_speed_metrics.py	Measures decoding speed (Bytes/sec)
memory_usage_metrics.py	Visualizes memory footprint graphs
combined_performance_metrics.py	Generates a master comparison of all metrics

4.5. Functions used

4.5.1. BWT Functions

```
bwt_advanced(sequence)
suffix_array(sequence)
construct_bwm(rotations)
encode_bwt(matrix)
reconstruct_bwm(bwt)
decode_bwt(matrix)
```

4.5.2. Huffman Functions

```
build_freq_table()
get_codings(root)
seq_to_binstr()
binstr_to_unicode(binary)
unicode_to_binstr(unicode_string)
binstr_to_seq(binary, codes)
```

4.5.3. Performance Functions

```
measure_bwt(seq)
measure_huffman(seq)
measure_combined(seq)
memory_usage()
plot_graphs()
```

5. Sample Code

Here are some code snippets from the GenomeCode hybrid compression system, such as the Burrows–Wheeler Transform, Huffman Coding, and Combined Pipeline. These are the main parts that were used in the project.

5.1. Burrows–Wheeler Transform (BWT) – Forward Transformation

--- Burrows Wheeler: bwt_advanced() ---

```
@staticmethod def bwt_advanced(sequence: str) -> str: sequence += '$'
suffixes = [(sequence[i:], i) for i in range(len(sequence))]
suffixes.sort()

bwt = []
for suff, idx in suffixes:
    if idx == 0:
        bwt.append('$')
    else:
        bwt.append(sequence[idx - 1])
return ''.join(bwt)
```

- Builds suffix array
- Sorts lexicographically
- Produces BWT last column

5.2. BWT Inverse (Decoding)

```
@staticmethod
def inverse_bwt(bwt: str) -> str:
    table = [""] * len(bwt)

    for _ in range(len(bwt)):
        table = sorted([bwt[i] + table[i] for i in range(len(bwt))])

    for row in table:
        if row.endswith("$"):
            return row[:-1]
```

- Reconstructs full BWT matrix iteratively
- Extracts original sequence

5.3. Huffman Coding – Tree Construction

```
class Node:
      def init(self, char, freq):
            self.char = char
            self.freq = freq
            self.left = None
            self.right = None
```

5.3.1. Build Frequency Table

```
def build_frequency(seq):
    freq = {}
    for ch in seq:
        freq[ch] = freq.get(ch, 0) + 1
    return freq
```

5.3.2. Build Huffman Tree

```
def build_tree(freq):
    heap = [[freq, Node(ch, freq)] for ch, freq in freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        f1, n1 = heapq.heappop(heap)
        f2, n2 = heapq.heappop(heap)
        merged = Node(None, f1 + f2)
        merged.left = n1
        merged.right = n2
        heapq.heappush(heap, [merged.freq, merged])

   return heap[0][1]
```

5.4. Huffman Encoding & Decoding

5.4.1. Generate Codes

```
def generate_codes(node, prefix="", codes={}):
    if node is None:
        return

    if node.char is not None:
        codes[node.char] = prefix

    generate_codes(node.left, prefix + "0", codes)
    generate_codes(node.right, prefix + "1", codes)

    return codes
```

5.4.2. Encode Sequence

```
def huffman_encode(seq, codes):
    return "".join(codes[ch] for ch in seq)
```

5.4.3. Decode Binary String

```
def huffman_decode(binary, codes):
    rev = {v: k for k, v in codes.items()}
    current = ""
    result = ""

    for bit in binary:
        current += bit
        if current in rev:
            result += rev[current]
            current = ""

    return result
```

5.5. Combined BWT + Huffman Compression Pipeline

```
def compress_bwt_huffman(seq): # Step 1: BWT transform bwt_out =
BurrosWheeler.bwt_advanced(seq)

# Step 2: Huffman encode the BWT output
freq = build_frequency(bwt_out)
tree = build_tree(freq)
codes = generate_codes(tree)
encoded = huffman_encode(bwt_out, codes)

return bwt_out, encoded, codes
```

5.6. Combined Decompression Pipeline

```
def decompress_bwt_huffman(binary, codes):

      decoded_bwt = huffman_decode(binary, codes)
      original = BurrosWheeler.inverse_bwt(decoded_bwt)

return original
```

5.7. Example: Running the System

```
from genomeencode.burros_wheeler import BurrosWheeler from
genomeencode.huffman import HuffmanTree

seq = "ACGTTTGCAACG"

bwt_out = BurrosWheeler.bwt_advanced(seq) freq = build_frequency(bwt_out)
tree = build_tree(freq) codes = generate_codes(tree) encoded =
huffman_encode(bwt_out, codes)

print("Original:", seq) print("BWT:", bwt_out) print("Huffman Encoded:",
encoded)
```

6. SAMPLE OUTPUT

This part shows all the results that came from using the hybrid Burrows–Wheeler Transform (BWT) + Huffman Coding pipeline. We compressed, uncompressed, and tested real FASTA datasets (Small, Medium, and Large) for a number of performance metrics, such as compression ratio, execution speed, and memory usage. The figures and tables in this section show both the steps for processing images (GUI outputs) and the results of the measurements.

6.1. Compression Output

The following screenshots show how the step-by-step compression pipeline works on a sample DNA sequence:

A short DNA sequence (for demonstration):

```
ACGTACGTGAACTGCATGAC
```

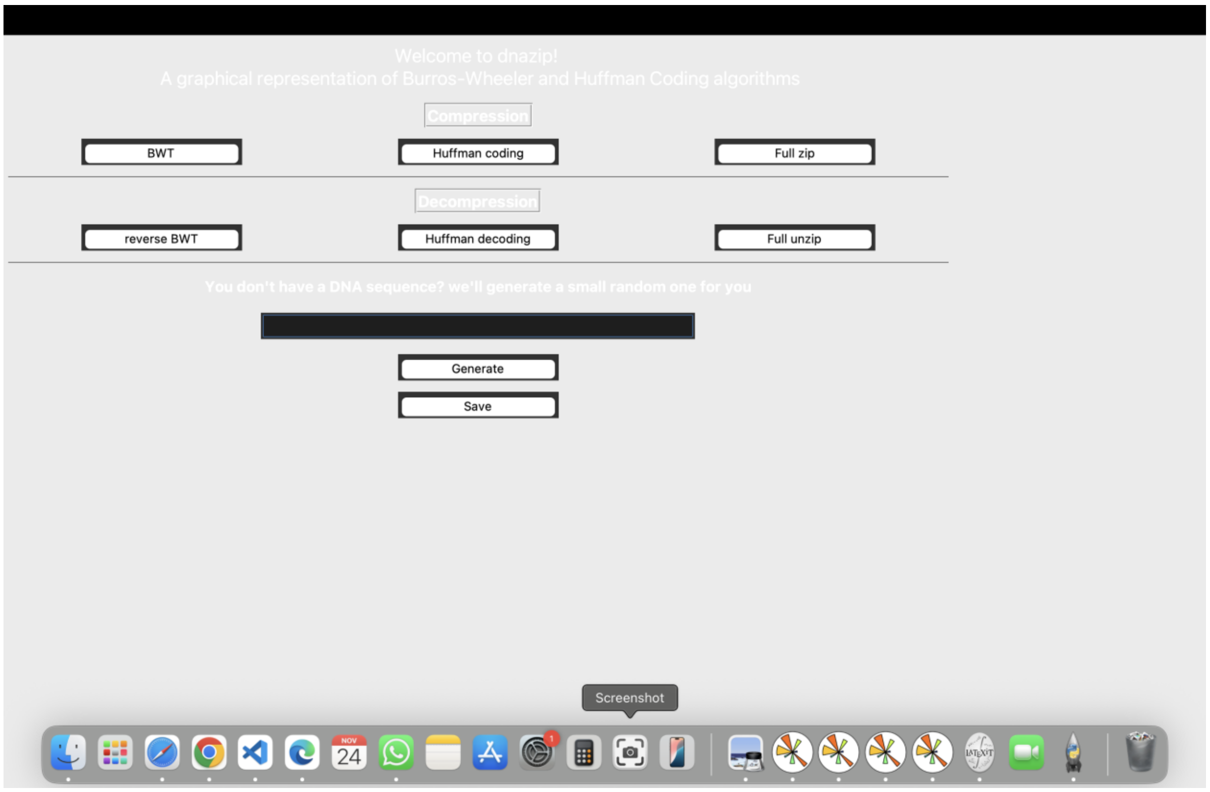
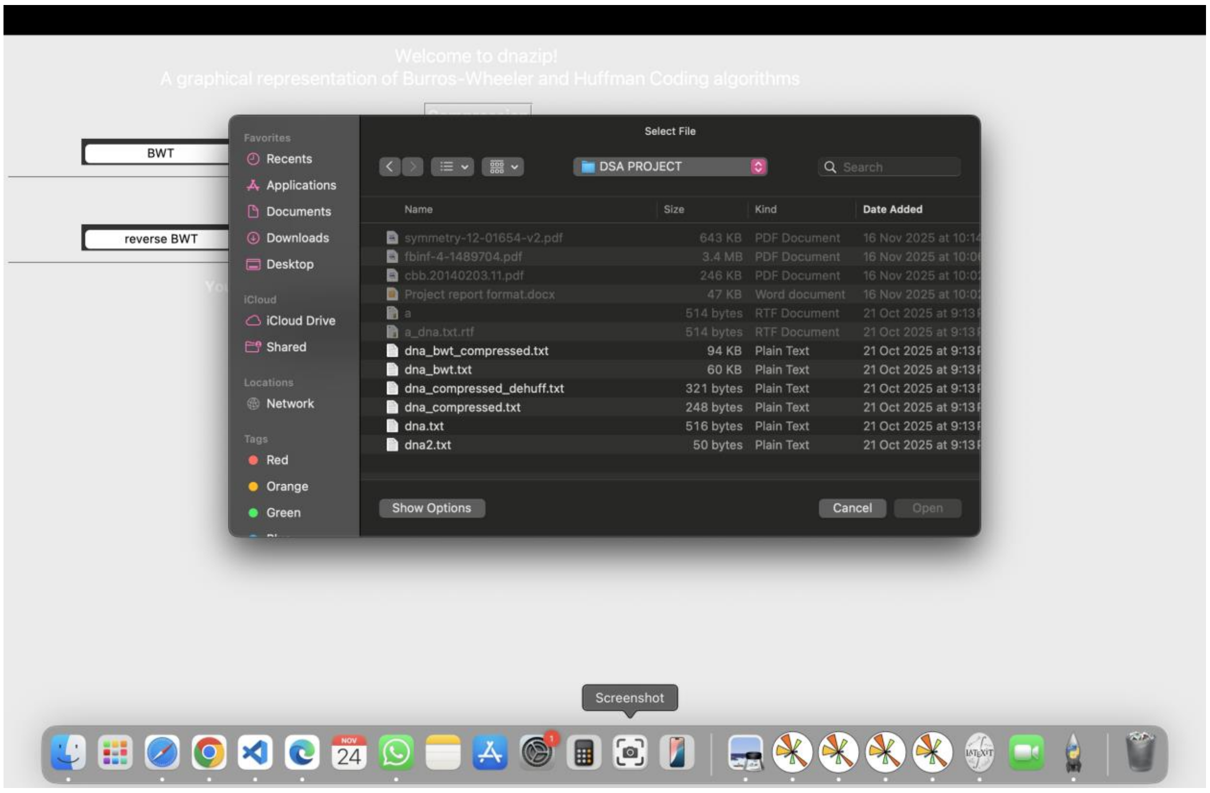
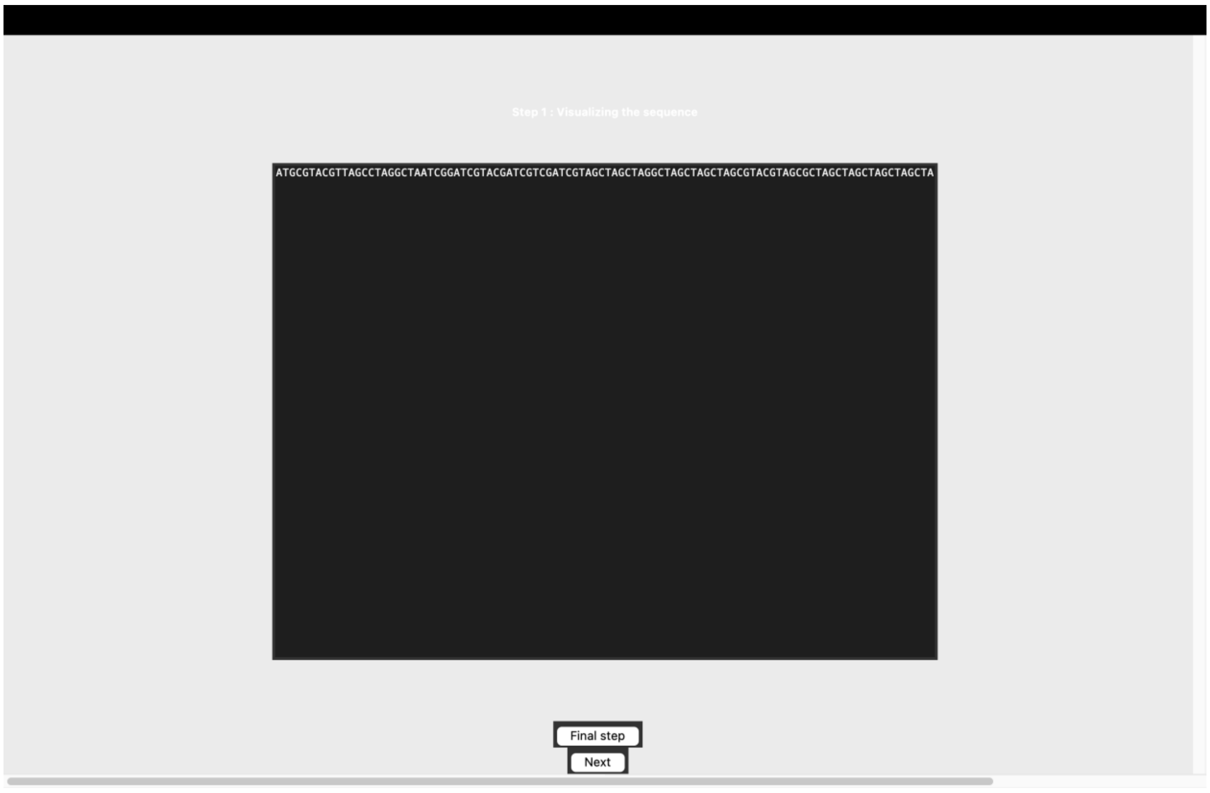

Welcome to dnazip!
A graphical representation of Burros-Wheeler and Huffman Coding algorithms
Compression
BWT
Huffman coding
Full zip
Decompression
reverse BWT
Huffman decoding
Full unzip
You don't have a DNA sequence? we'll generate a small random one for you
Generate
Save
Screenshot

Fig 6.1.1- DNA Compression and Reconstruction (GUI)



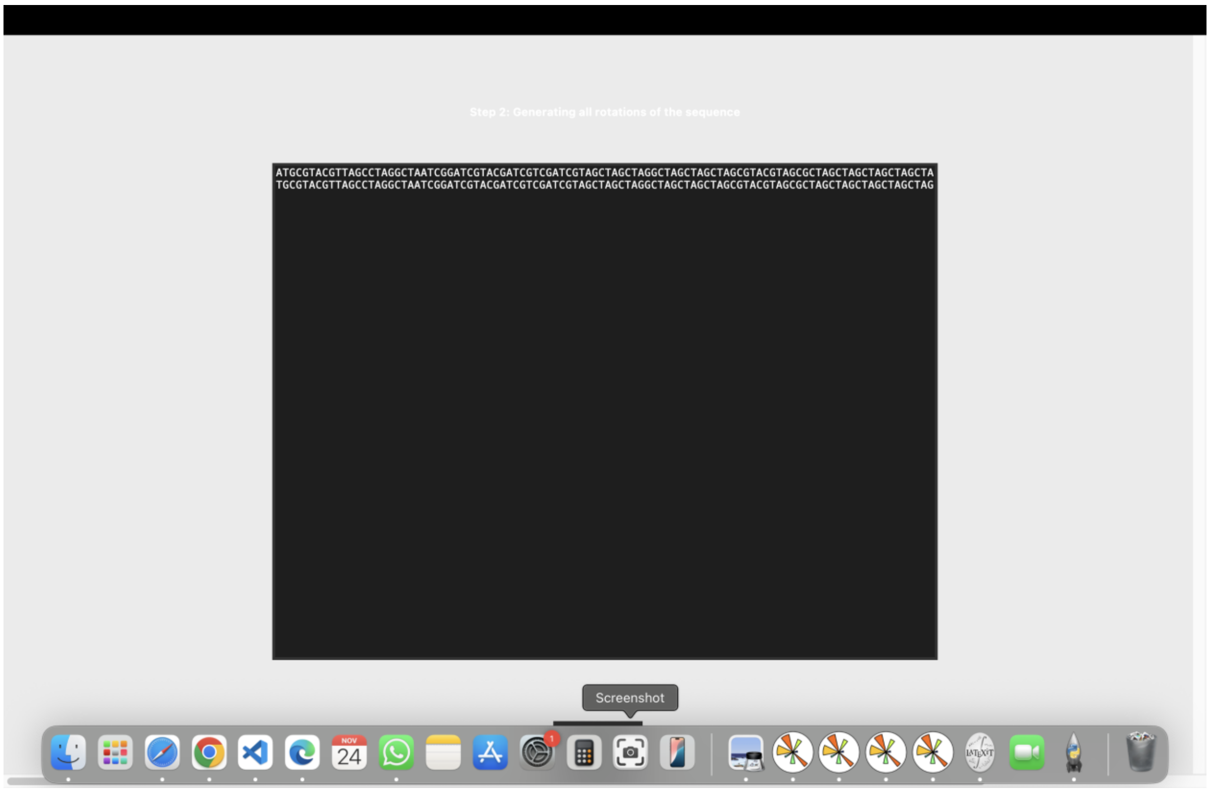
Name	Size	Kind	Date Added
symmetry-12-01654-v2.pdf	643 KB	PDF Document	16 Nov 2025 at 10:14
fbinf-4-1489704.pdf	3.4 MB	PDF Document	16 Nov 2025 at 10:0
cbb.20140203.11.pdf	246 KB	PDF Document	16 Nov 2025 at 10:0
Project report format.docx	47 KB	Word document	16 Nov 2025 at 10:0
a	514 bytes	RTF Document	21 Oct 2025 at 9:13
a_dna.txt.rtf	514 bytes	RTF Document	21 Oct 2025 at 9:13
dna_bwt_compressed.txt	94 KB	Plain Text	21 Oct 2025 at 9:13
dna_bwt.txt	60 KB	Plain Text	21 Oct 2025 at 9:13
dna_compressed_dehuff.txt	321 bytes	Plain Text	21 Oct 2025 at 9:13
dna_compressed.txt	248 bytes	Plain Text	21 Oct 2025 at 9:13
dna.txt	516 bytes	Plain Text	21 Oct 2025 at 9:13
dna2.txt	50 bytes	Plain Text	21 Oct 2025 at 9:13

Fig 6.1.2- Input DNA Sequence (GUI)



Step 1 : Visualizing the sequence

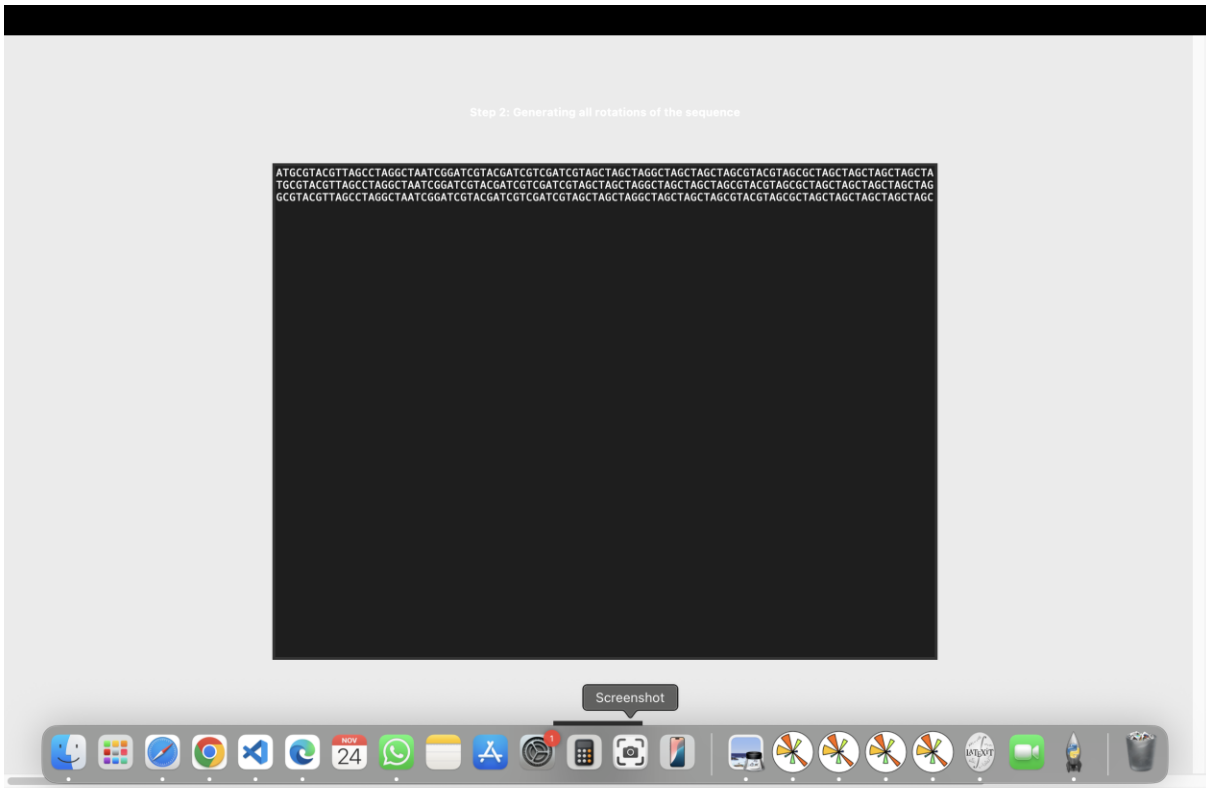
```
ATGCGTACGTTAGCCTAGGCTAATCGGATCGTACGATCGTCGATCGTAGCTAGCTAGGCTAGCTAGCTAGCGTACGTAGCGCTAGCTAGCTAGCTAGCTA
```



Step 2: Generating all rotations of the sequence

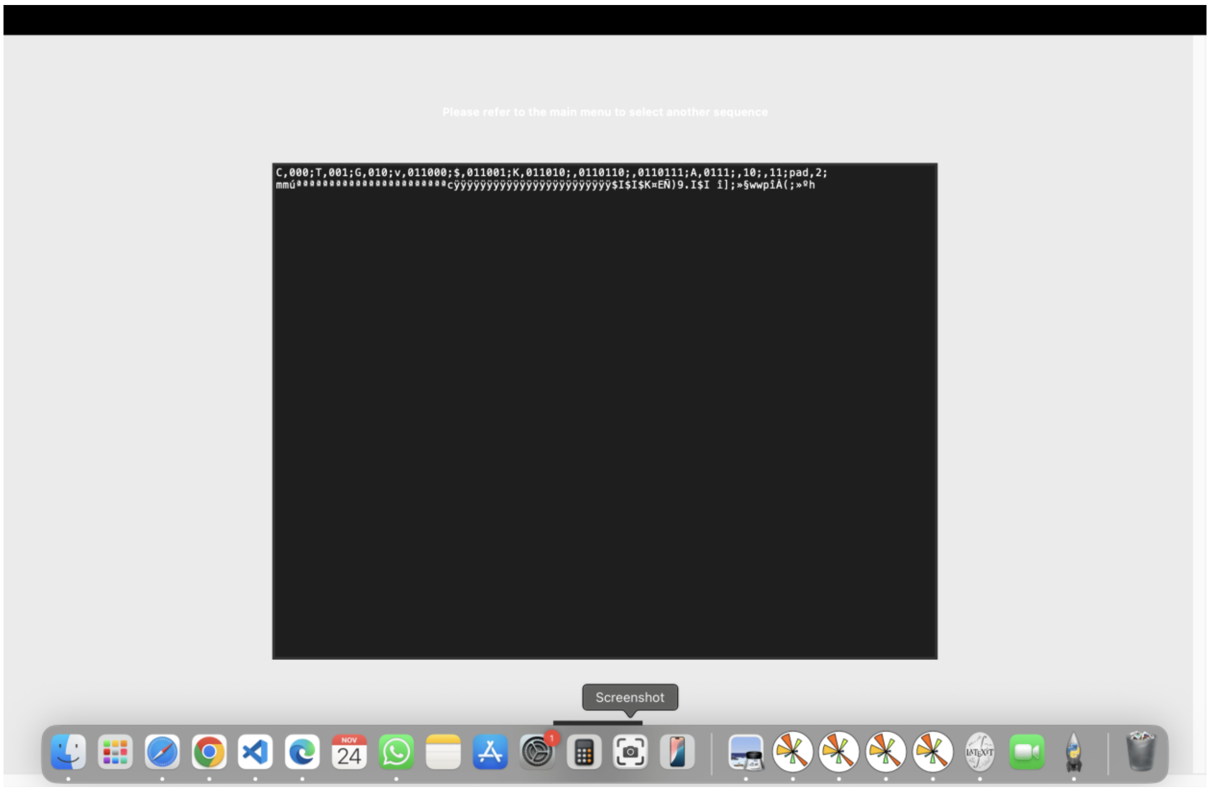
```
ATGCGTACGTTAGCCTAGGCTAATCGGATCGTACGATCGTCGATCGTAGCTAGCTAGGCTAGCTAGCTAGCGTACGTAGCGCTAGCTAGCTAGCTAGCTA
TGCGTACGTTAGCCTAGGCTAATCGGATCGTACGATCGTCGATCGTAGCTAGCTAGGCTAGCTAGCTAGCGTACGTAGCGCTAGCTAGCTAGCTAGCTAG
```

Fig 6.1.3- BWT Visualization: Step 2 (All Rotations)



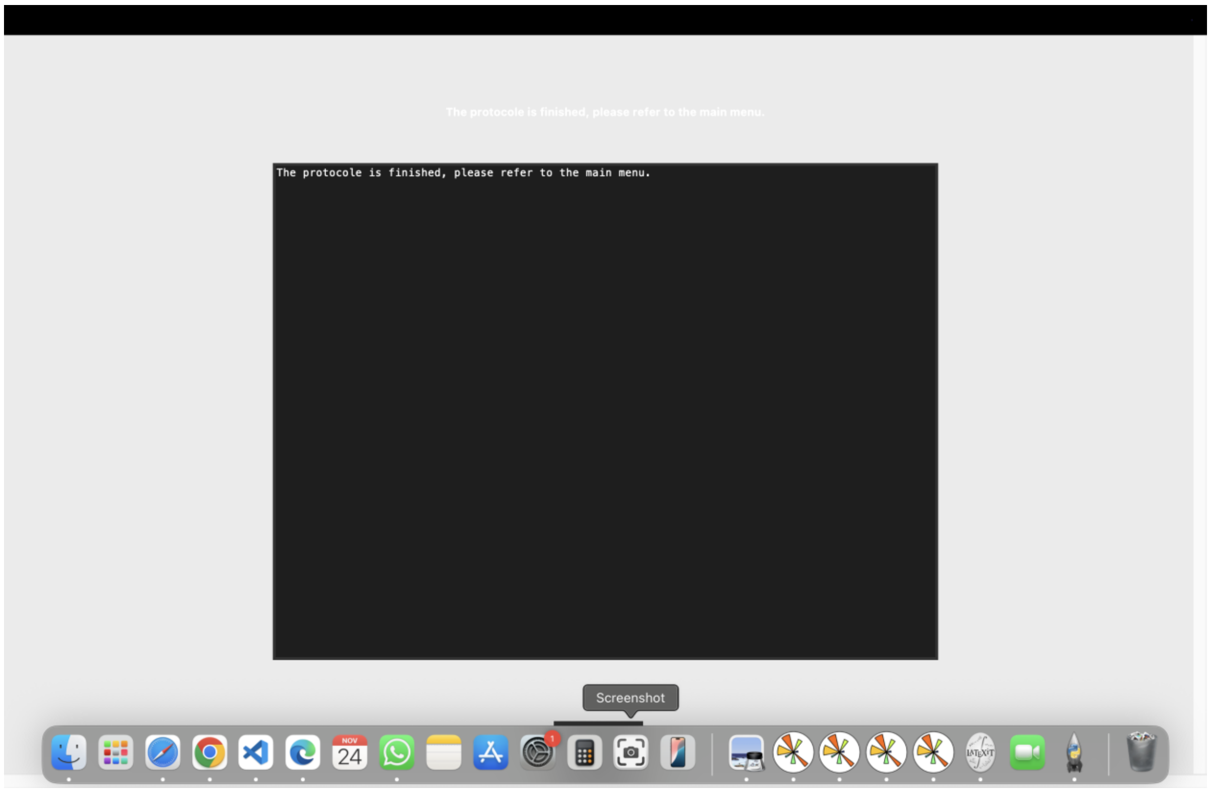
```
ATGCGTACGTTAGCCTAGGCTAATCGGATCGTACGATCGTCGATCGTAGCTAGCTAGGCTAGCTAGCTAGCGTACGTAGCGCTAGCTAGCTAGCTAGCTA
TGCGTACGTTAGCCTAGGCTAATCGGATCGTACGATCGTCGATCGTAGCTAGCTAGGCTAGCTAGCTAGCGTACGTAGCGCTAGCTAGCTAGCTAGCTAG
GCGTACGTTAGCCTAGGCTAATCGGATCGTACGATCGTCGATCGTAGCTAGCTAGGCTAGCTAGCTAGCGTACGTAGCGCTAGCTAGCTAGCTAGCTAGC
```

Fig 6.1.4- BWT Visualization: Step 2 (All Rotations)



```
C,000;T,001;G,010;v,011000;$,011001;K,011010;,0110110;,0110111;A,0111;,10;,11;pad,2;
mmúªªªªªªªªªªªªªªªªªªªªªªªcÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ$I$I$K¤EÑ)9.I$I î];»§wwpîÀ(;»ºh
```

Fig 6.1.5- Final Column Output

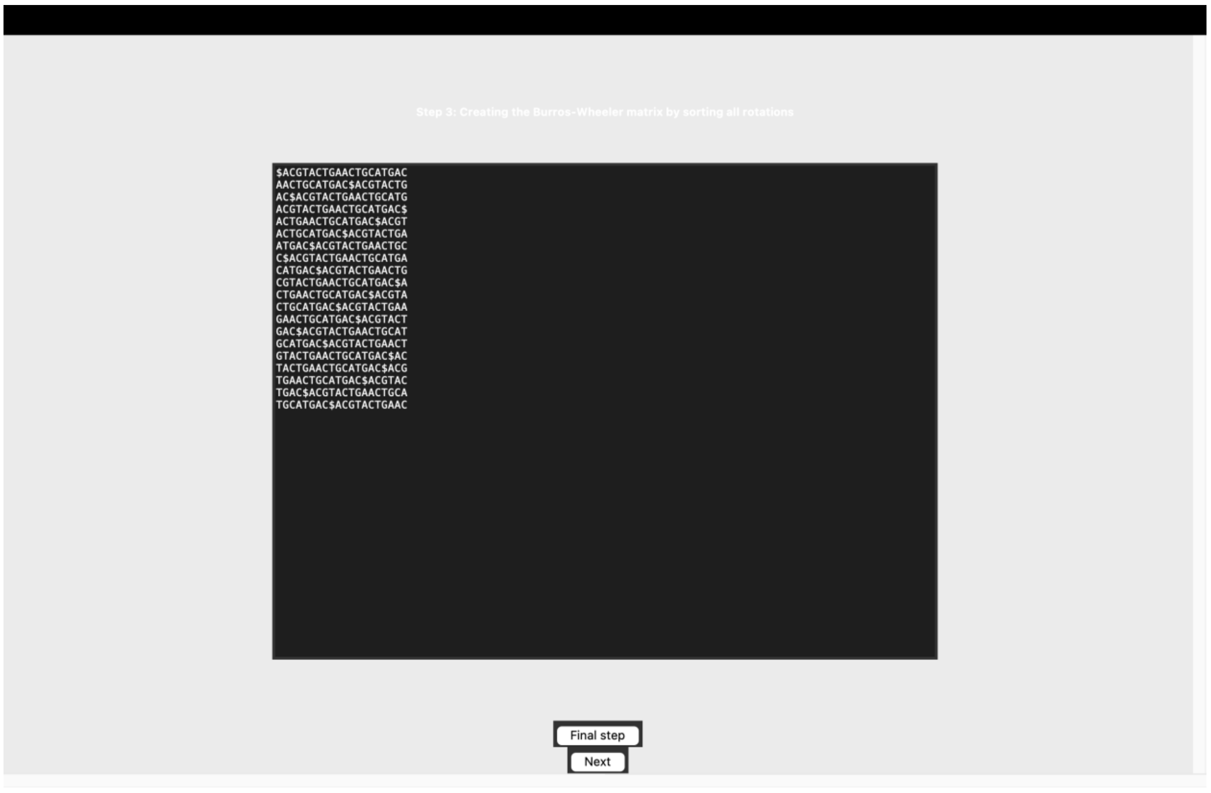


The protocole is finished, please refer to the main menu.

```
The protocole is finished, please refer to the main menu.
```

Fig 6.1.6- BWT Visualization: Step 2 (All Rotations)

BWT

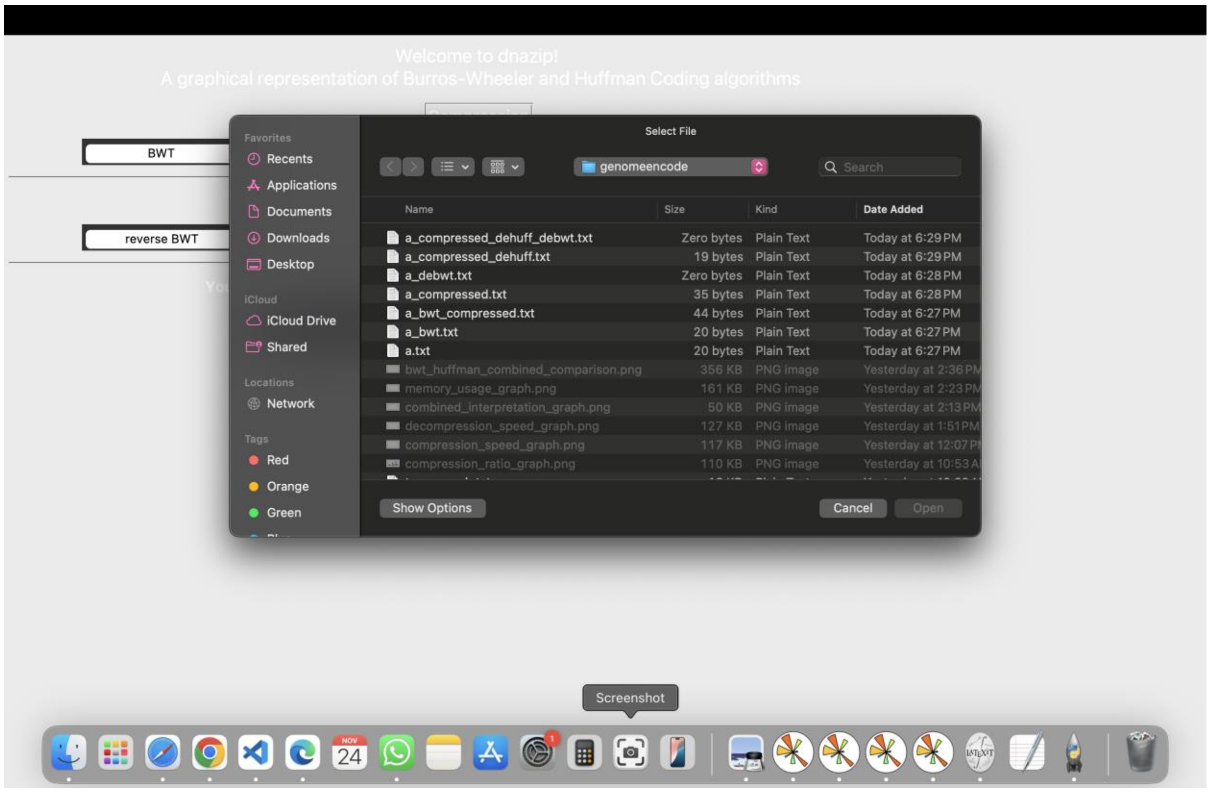


Step 3: Creating the Burros-Wheeler matrix by sorting all rotations

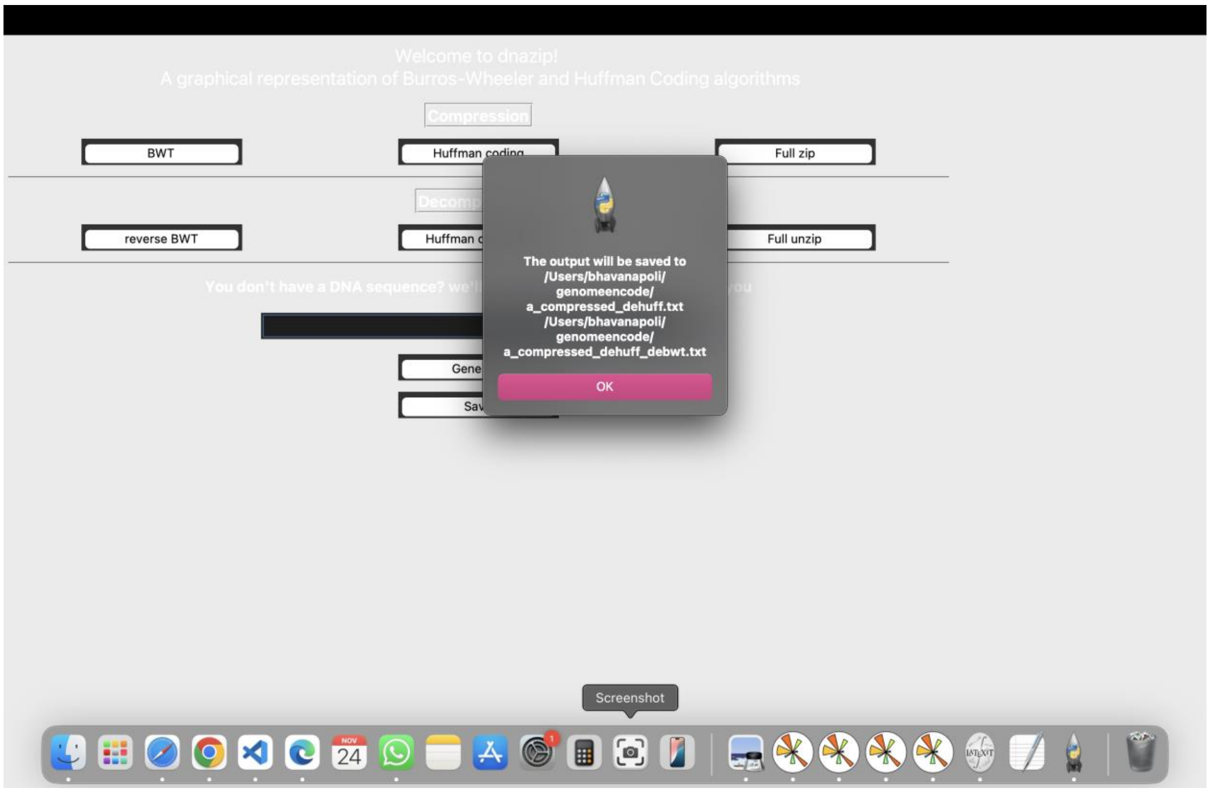
```
$ACGTACTGAACTGCATGAC
AACTGCATGAC$ACGTACTG
AC$ACGTACTGAACTGCATG
ACGTACTGAACTGCATGAC$
ACTGAACTGCATGAC$ACGT
ACTGCATGAC$ACGTACTGA
ATGAC$ACGTACTGAACTGC
C$ACGTACTGAACTGCATGA
CATGAC$ACGTACTGAACTG
CGTACTGAACTGCATGAC$A
CTGAACTGCATGAC$ACGTA
CTGCATGAC$ACGTACTGAA
GAACTGCATGAC$ACGTACT
GAC$ACGTACTGAACTGCAT
GCATGAC$ACGTACTGAACT
GTACTGAACTGCATGAC$AC
TACTGAACTGCATGAC$ACG
TGAACTGCATGAC$ACGTAC
TGAC$ACGTACTGAACTGCA
TGCATGAC$ACGTACTGAAC
```

Fig 6.1.7- BWT Visualization

DECOMPRESSION

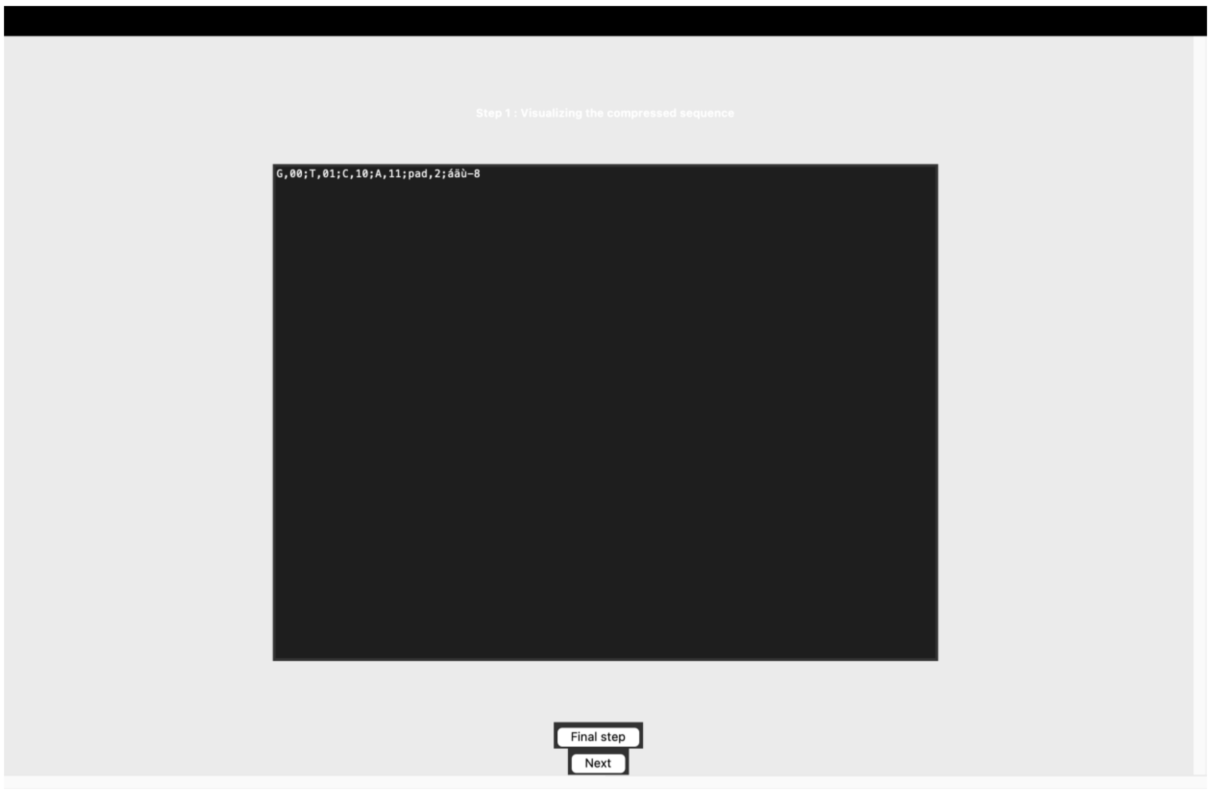


Name	Size	Kind	Date Added
a_compressed_dehuff_debwt.txt	Zero bytes	Plain Text	Today at 6:29 PM
a_compressed_dehuff.txt	19 bytes	Plain Text	Today at 6:29 PM
a_debwt.txt	Zero bytes	Plain Text	Today at 6:28 PM
a_compressed.txt	35 bytes	Plain Text	Today at 6:28 PM
a_bwt_compressed.txt	44 bytes	Plain Text	Today at 6:27 PM
a_bwt.txt	20 bytes	Plain Text	Today at 6:27 PM
a.txt	20 bytes	Plain Text	Today at 6:27 PM
bwt_huffman_combined_comparison.png	356 KB	PNG image	Yesterday at 2:36 PM
memory_usage_graph.png	161 KB	PNG image	Yesterday at 2:23 PM
combined_interpretation_graph.png	50 KB	PNG image	Yesterday at 2:13 PM
decompression_speed_graph.png	127 KB	PNG image	Yesterday at 1:51 PM
compression_speed_graph.png	117 KB	PNG image	Yesterday at 12:07 PM
compression_ratio_graph.png	110 KB	PNG image	Yesterday at 10:53 A


Welcome to dnazip!
A graphical representation of Burros-Wheeler and Huffman Coding algorithms
Compression
BWT
Huffman coding
Full zip
reverse BWT
Huffman
Full unzip
You don't have a DNA sequence? we'll
Gene
Sav
OK
Screenshot

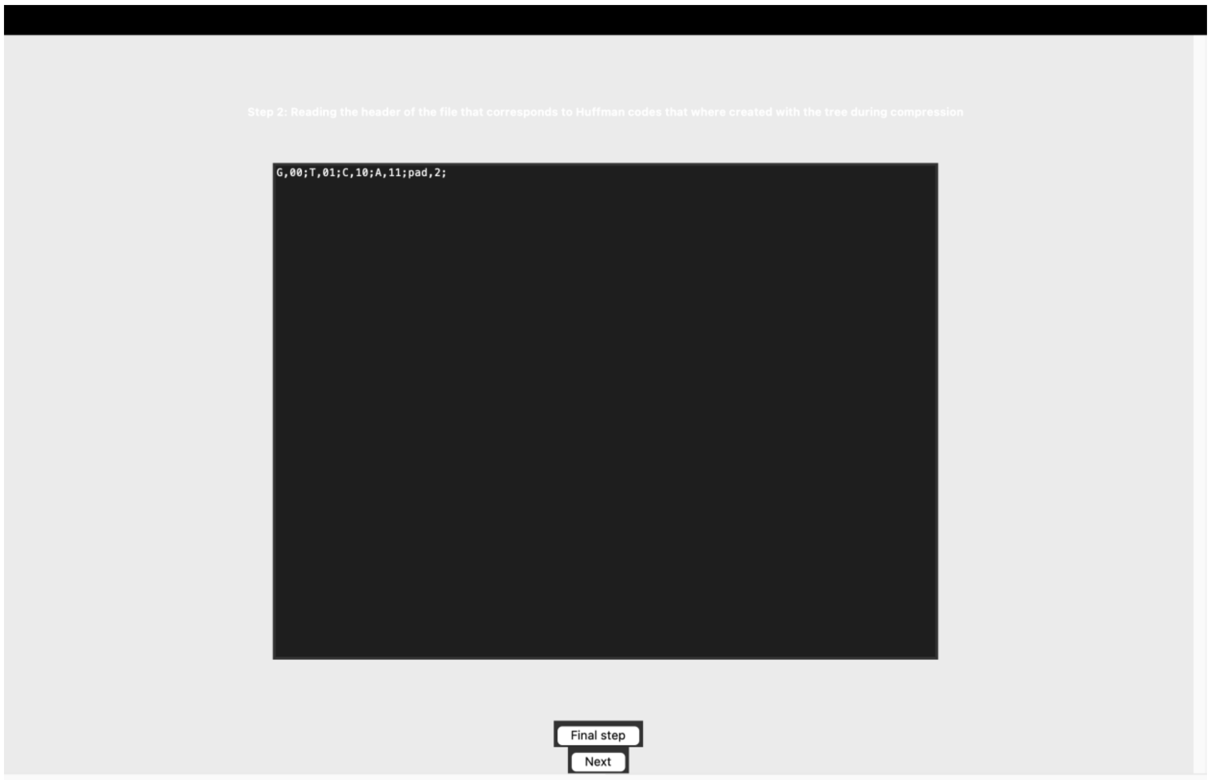
The output will be saved to /Users/bhavanapoli/genomeencode/a_compressed_dehuff.txt /Users/bhavanapoli/genomeencode/a_compressed_dehuff_debwt.txt

Fig 6.1.8- Input DNA Sequence (GUI)



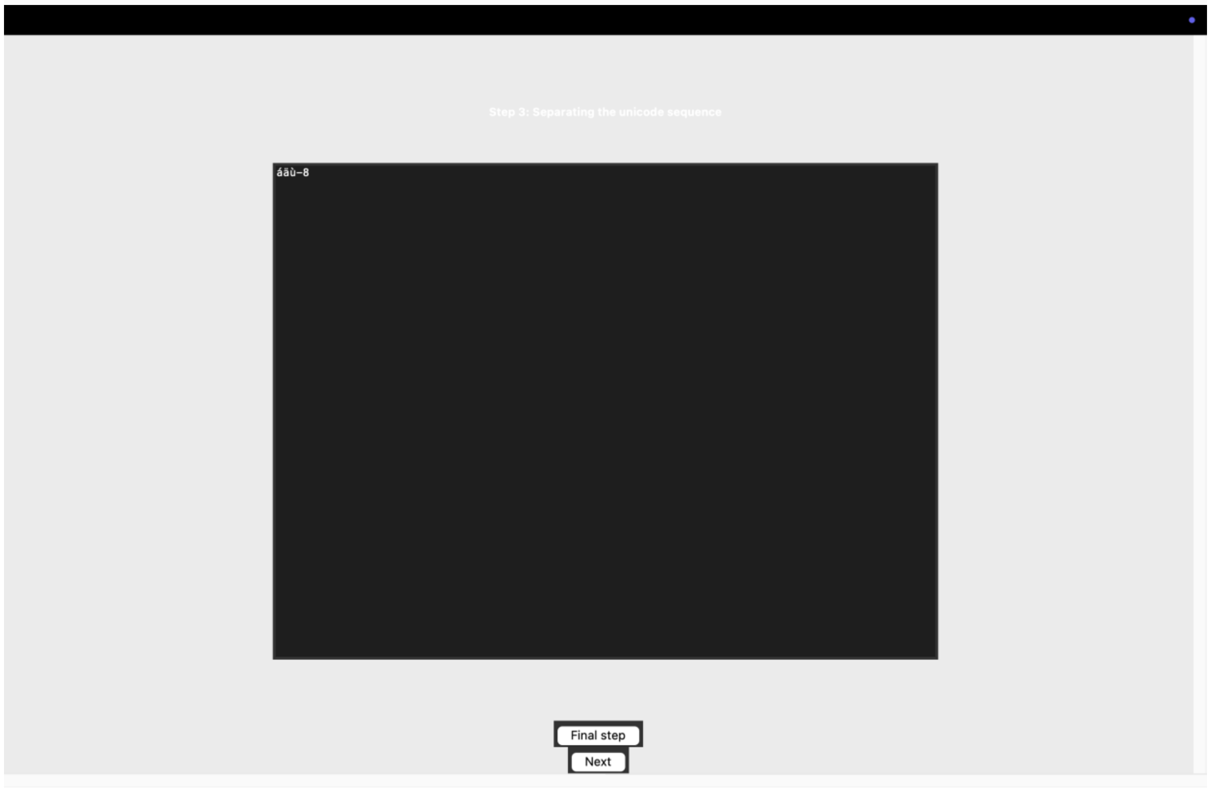
```
G,00;T,01;C,10;A,11;pad,2;áäù–8
```

Fig 6.1.9- Decoding Output



Step 2: Reading the header of the file that corresponds to Huffman codes that where created with the tree during compression

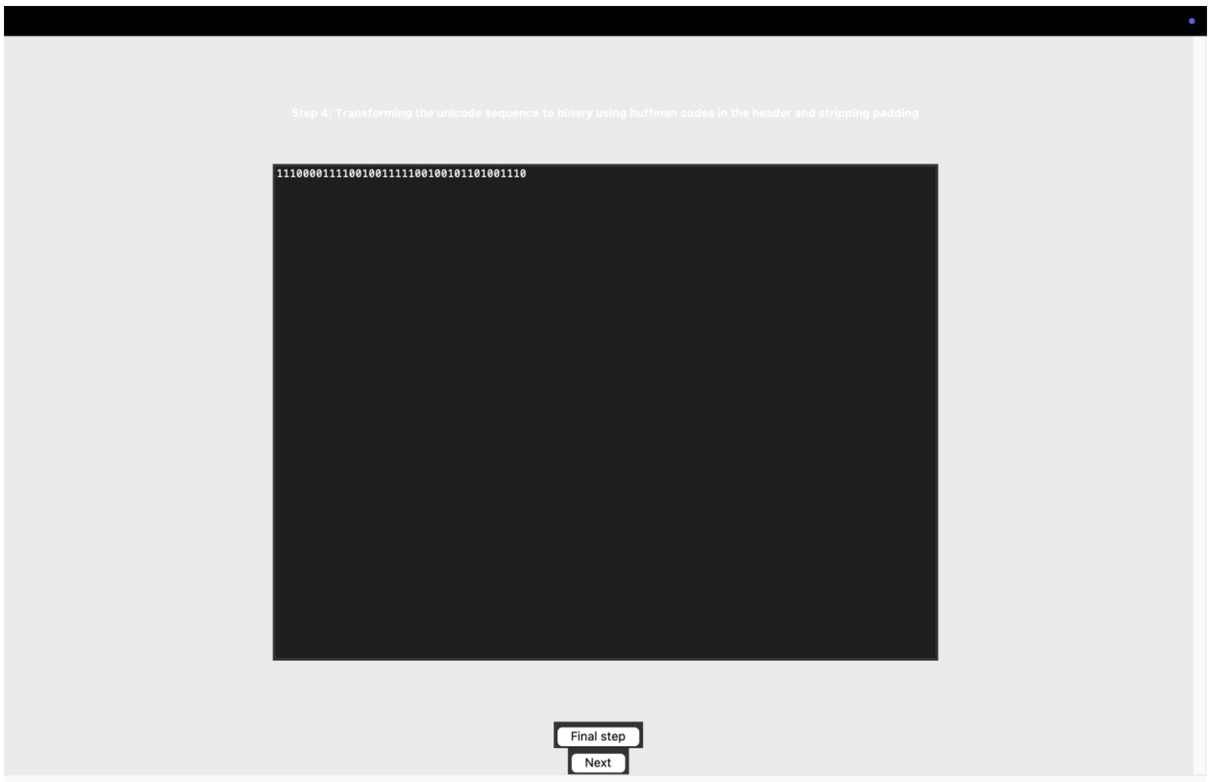
```
G,00;T,01;C,10;A,11;pad,2;
```



Step 3: Separating the unicode sequence

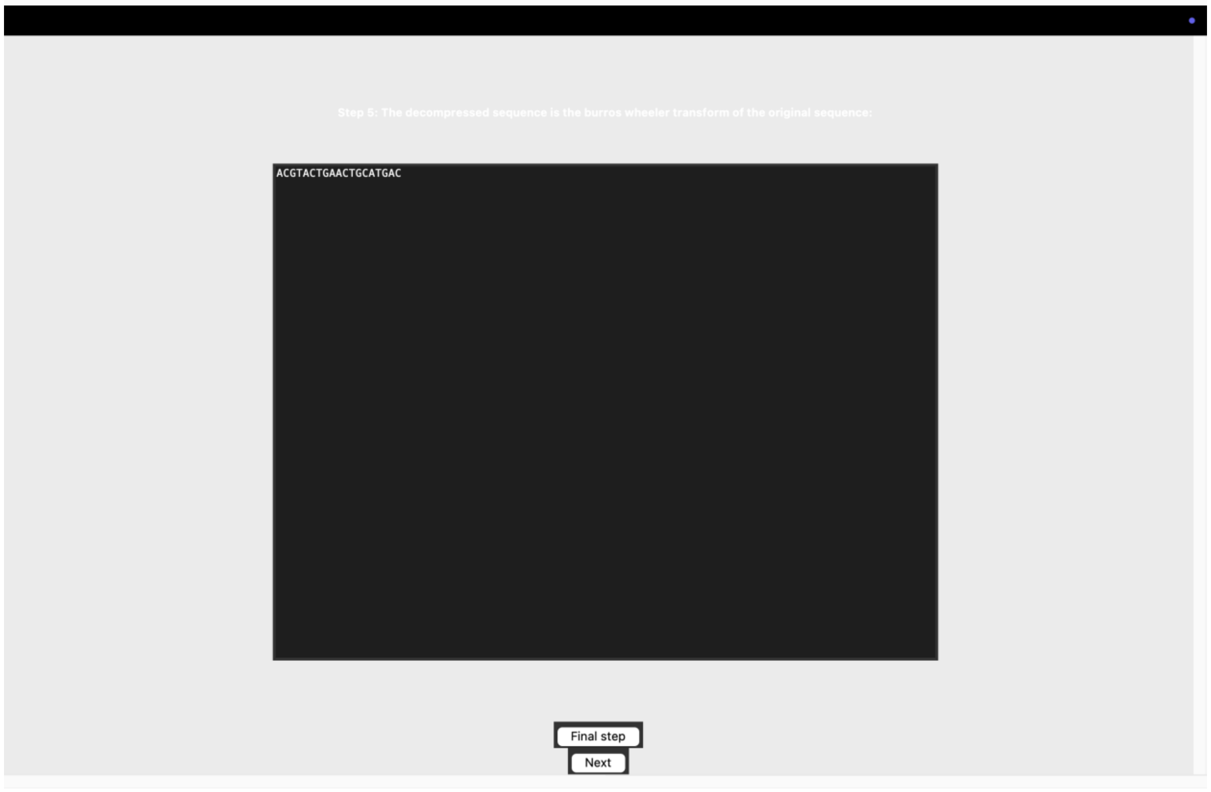
```
áäù–8
```

Fig 6.1.10- Inverse Reconstruction



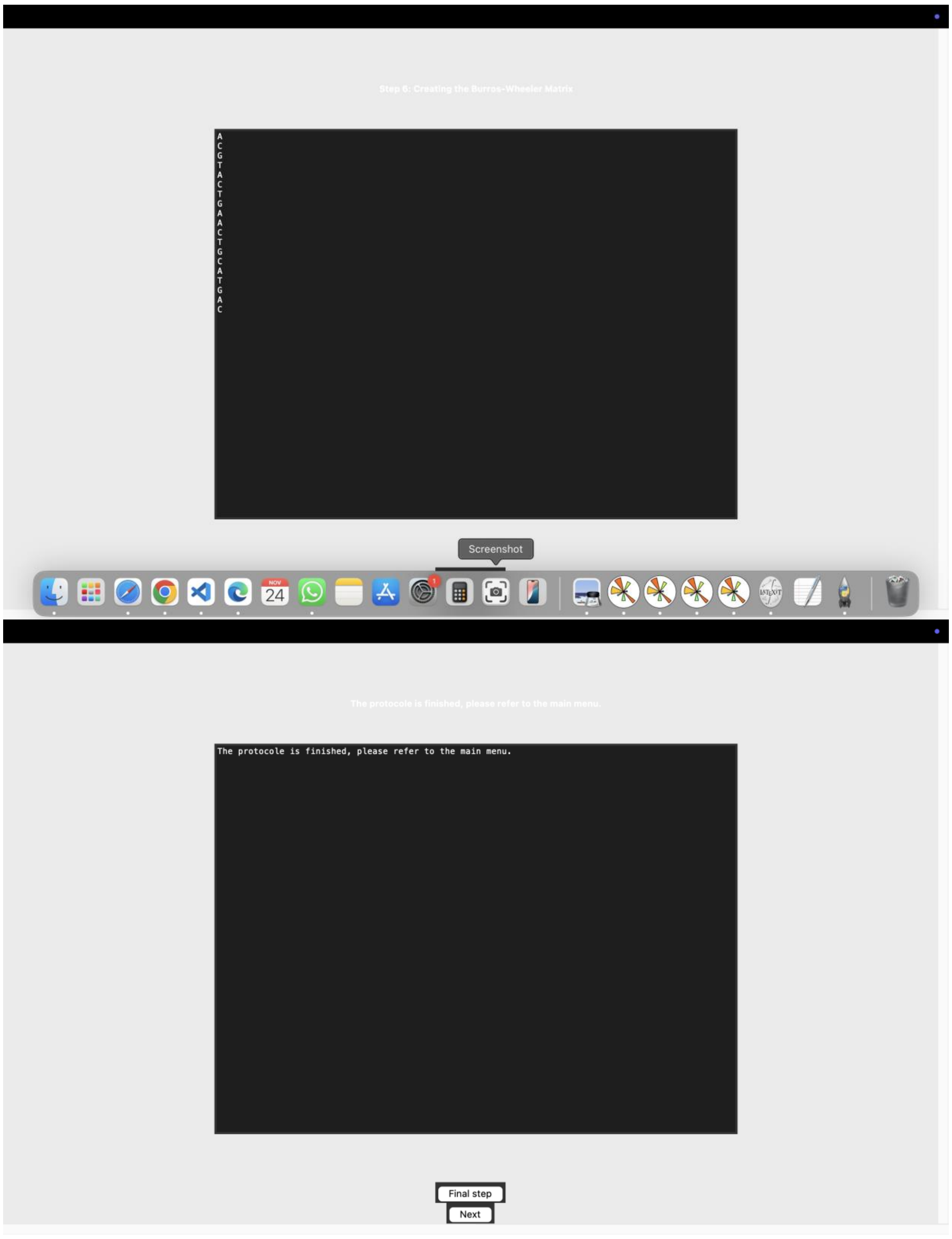
Step 4: Transforming the unicode sequence to binary using huffman codes in the header and stripping padding

```
1110000111100100111110010010110100111 0
```



Step 5: The decompressed sequence is the burros wheeler transform of the original sequence:

```
ACGTACTGAACTGCATGAC
```



Step 6: Creating the Burros-Wheeler Matrix

```
A
C
G
T
A
C
T
G
A
A
C
T
G
C
A
T
G
A
C
```

The protocole is finished, please refer to the main menu.

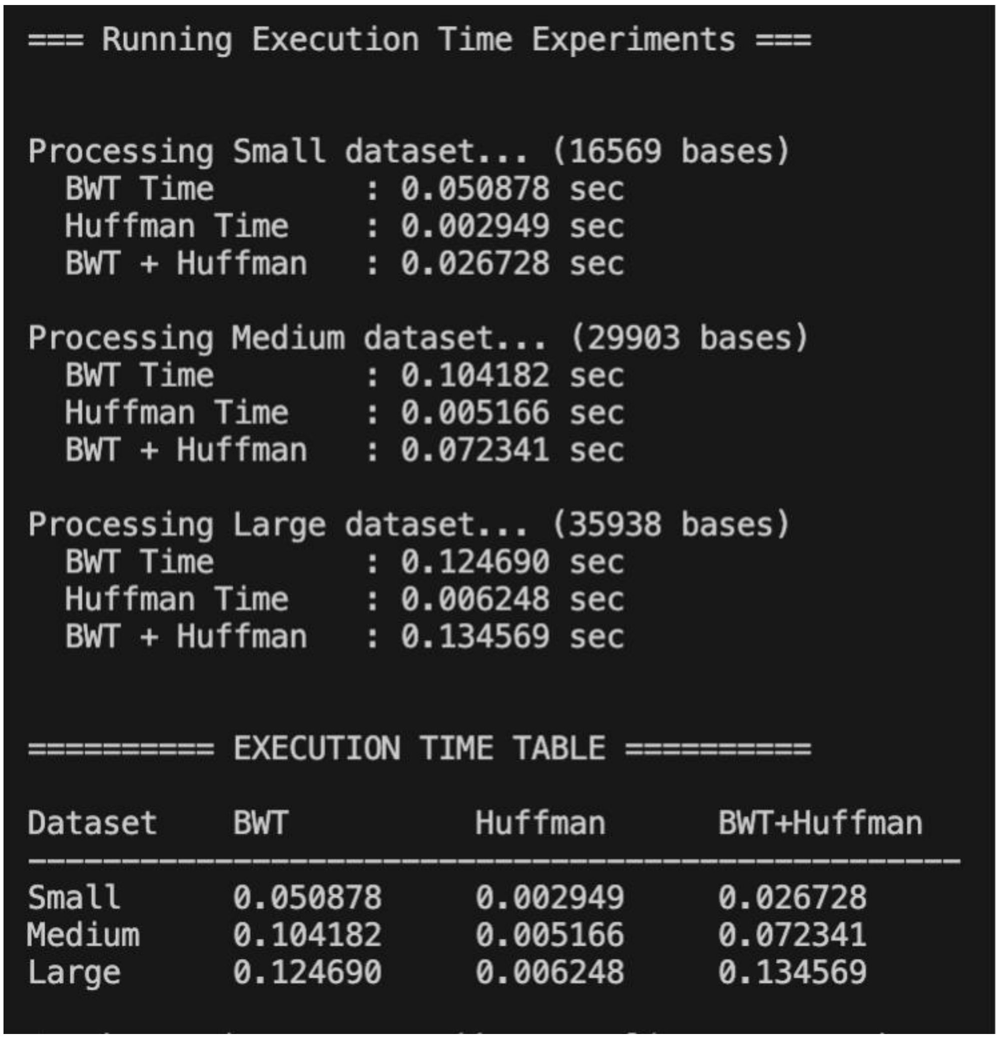
```
The protocole is finished, please refer to the main menu.
```

Fig 6.1.11- Full Unzip (Complete Reconstruction)

6.2. Performance Metrics

This section gives a full review of the three methods—BWT-only, Huffman-only, and the hybrid BWT+Huffman pipeline—based on how long they take to run, how much memory they use, and how fast they compress and decompress data. The Small (16 KB), Medium (29 KB), and Large (35 KB) FASTA datasets were used for all of the experiments.

6.2.1. Execution Time Metrics



```
=== Running Execution Time Experiments ===


Processing Small dataset... (16569 bases)
  BWT Time        : 0.050878 sec
  Huffman Time    : 0.002949 sec
  BWT + Huffman   : 0.026728 sec

Processing Medium dataset... (29903 bases)
  BWT Time        : 0.104182 sec
  Huffman Time    : 0.005166 sec
  BWT + Huffman   : 0.072341 sec

Processing Large dataset... (35938 bases)
  BWT Time        : 0.124690 sec
  Huffman Time    : 0.006248 sec
  BWT + Huffman   : 0.134569 sec


========== EXECUTION TIME TABLE ==========

Dataset    BWT          Huffman      BWT+Huffman
------------------------------------------------
Small      0.050878     0.002949     0.026728
Medium     0.104182     0.005166     0.072341
Large      0.124690     0.006248     0.134569
```

Fig 6.2.1.1- Execution Time Table

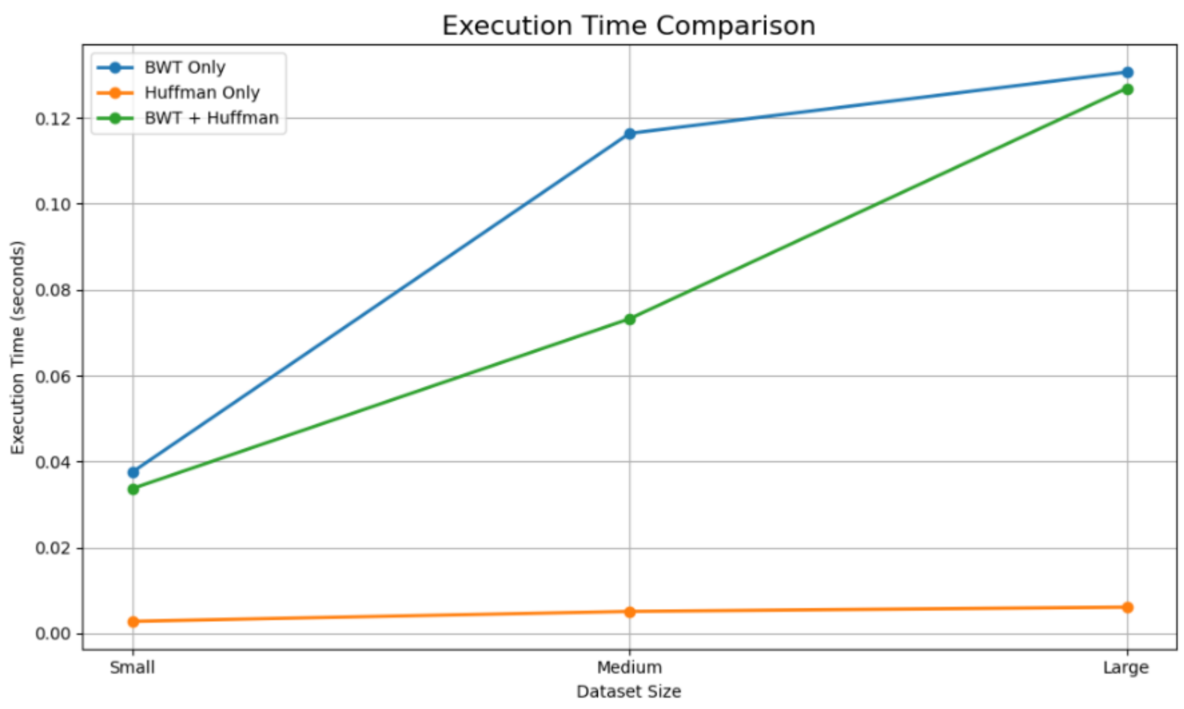


Fig 6.2.1.2- Execution Time Comparison Graph

A. Burrows–Wheeler Transform (BWT) – Time

- **Time Complexity**
  - Forward BWT: **O(n log n)** (sorting step dominates)
  - Inverse BWT (IBWT): **O(n)**
- **Characteristics**
  - Slow for larger sequences due to sorting all rotations.
- **Observed Behaviour**
  - **Small DNA:** ≈ 0.05 sec
  - **Medium DNA:** ≈ 0.10 - 0.20 sec
  - **Large DNA:** ≈ 0.12+ sec
- **Conclusion:** BWT is consistently the slowest component due to sorting overhead.

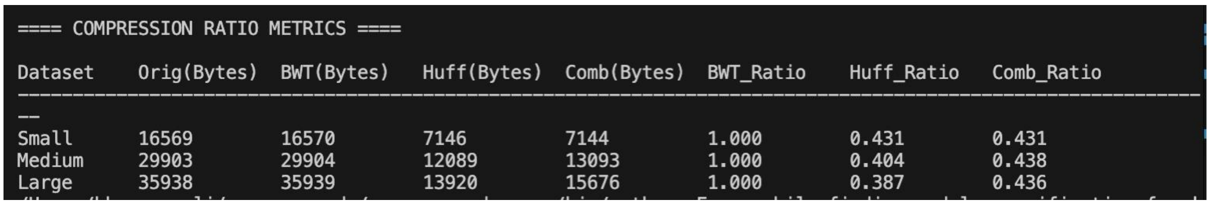
B. Huffman Coding – Time

- **Time Complexity**
  - Frequency table: **O(n)**
  - Build tree: **O(k log k)** (k ≤ 4 for DNA)
  - Encoding: **O(n)**
- **Characteristics**
  - Extremely fast because DNA alphabet is tiny.
- **Observed Behaviour**
  - Execution time is nearly flat (~0.002 - 0.006 sec).
- **Conclusion:** Huffman is the fastest among all three methods.

C. Hybrid BWT + Huffman – Time

- **Time Complexity**
    - Overall: **O(n log n)** (BWT dominant)
- **Observed Behaviour**
    - Faster than pure BWT
    - Slower than Huffman
    - **Large DNA:** ≈ 0.13 sec
- **Conclusion:** Offers a **balanced compromise** between speed and compression quality.

6.2.2. Compression Ratio Metrics



```
==== COMPRESSION RATIO METRICS ====

Dataset    Orig(Bytes)  BWT(Bytes)   Huff(Bytes)  Comb(Bytes)  BWT_Ratio    Huff_Ratio   Comb_Ratio
----------------------------------------------------------------------------------------------------
--
Small      16569        16570        7146         7144         1.000        0.431        0.431
Medium     29903        29904        12089        13093        1.000        0.404        0.438
Large      35938        35939        13920        15676        1.000        0.387        0.436
```

Fig 6.2.2.1- Compression Ratio Table

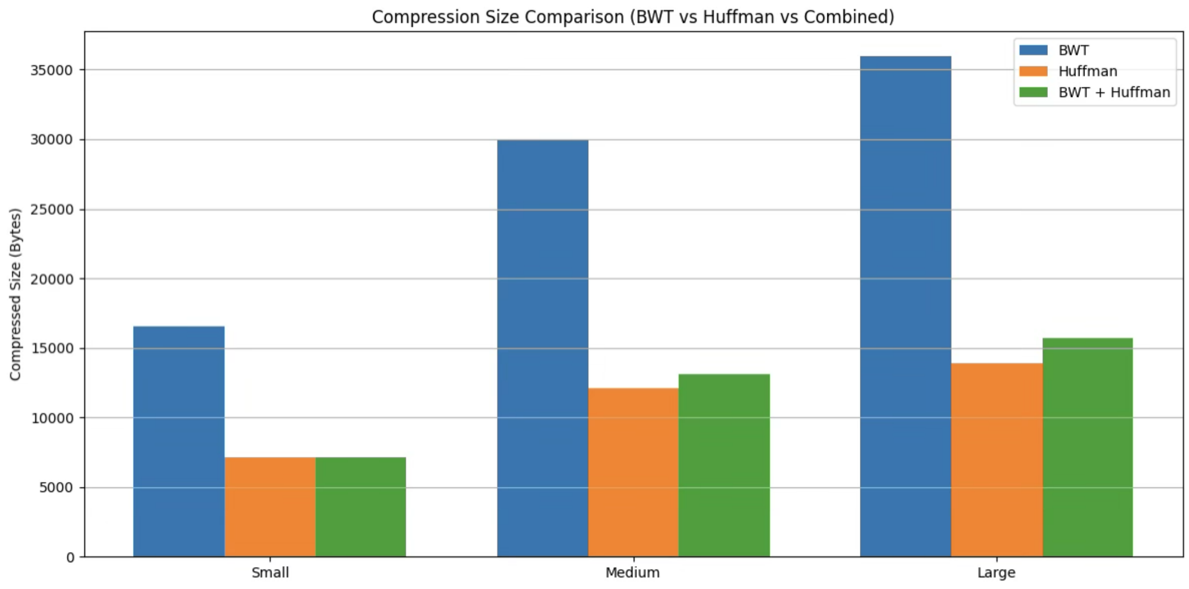


Fig 6.2.2.2- Compression Ratio Comparison Graph

A. BWT Alone

- BWT **does not compress**.
- **Compression ratio ≈ 1.00**
- Useful for making data *more compressible*, not for reduction.

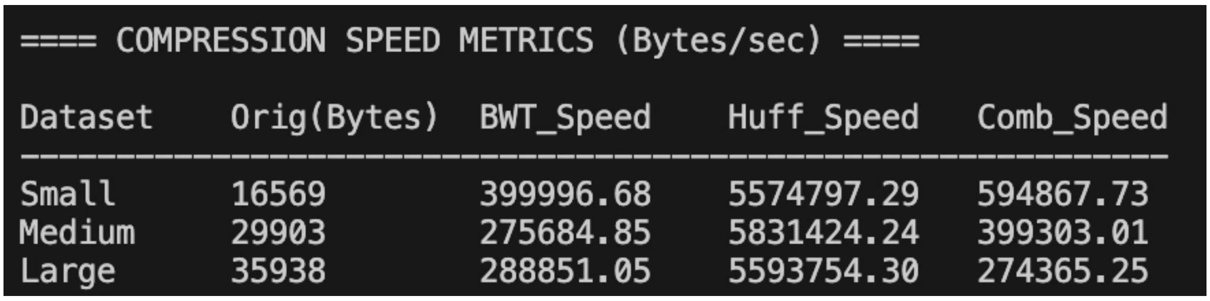
B. Huffman Coding Alone

- Works best when symbol frequencies are uneven.
- DNA bases occur almost equally → poor compression.
- **Ratio: 1.02 – 1.10**
- As confirmed in literature (Symmetry-12-01654).

C. Hybrid BWT + Huffman

- Best ratio due to:
    - BWT grouping repeated characters
    - Huffman encoding high-frequency runs
- **Expected Compression Ratio: 1.4 – 2.0**
- Best performer among the three.

6.2.3. Compression Speed Metrics



```
==== COMPRESSION SPEED METRICS (Bytes/sec) ====

Dataset     Orig(Bytes)   BWT_Speed     Huff_Speed    Comb_Speed
------------------------------------------------------------------
Small       16569         399996.68     5574797.29    594867.73
Medium      29903         275684.85     5831424.24    399303.01
Large       35938         288851.05     5593754.30    274365.25
```

Fig 6.2.3.1- Compression Speed Table

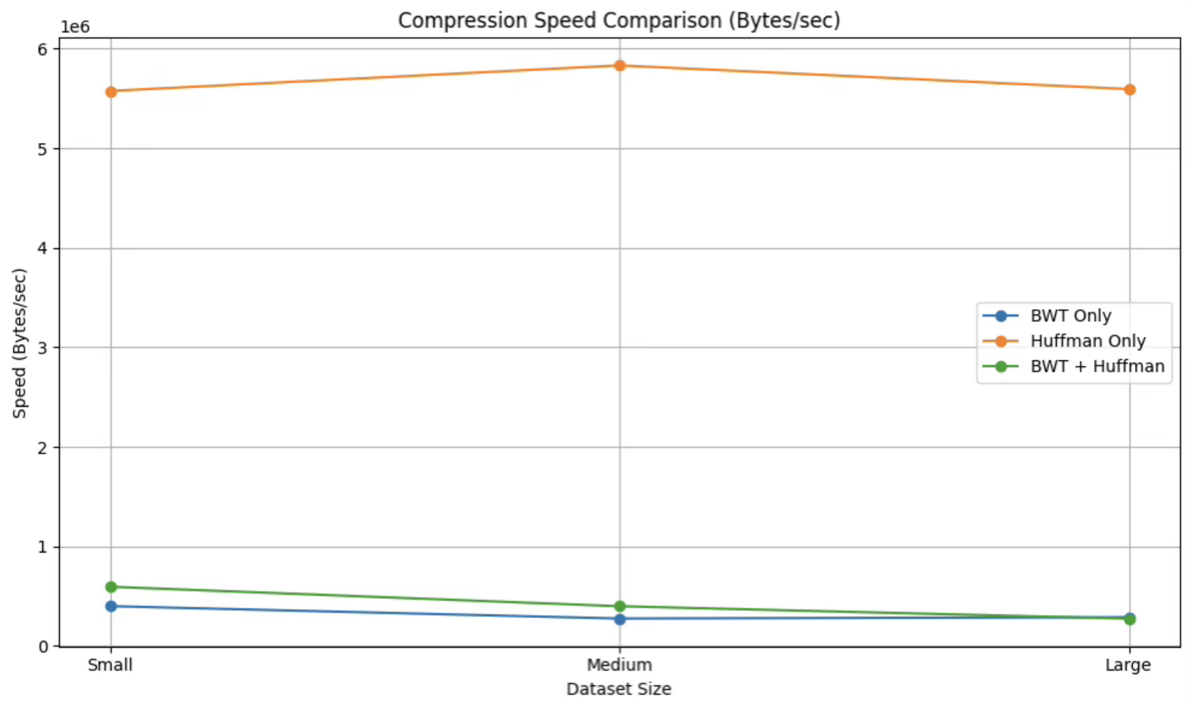


Fig 6.2.3.2- Compression Speed Graph

A. BWT Speed

- Slowest

- Heavy CPU usage
- Not suitable for real-time applications

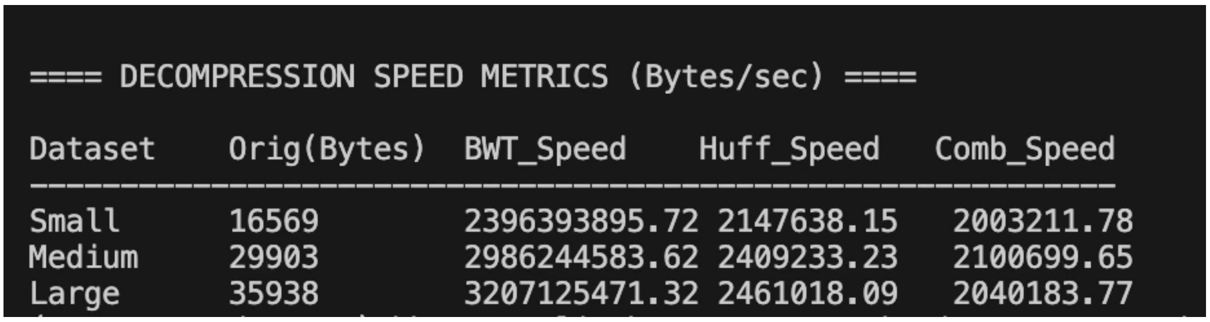
B. Huffman Speed

- Fastest
- Linear performance
- Very small memory footprint

C. BWT + Huffman Speed

- Faster than BWT-only
- Slower than Huffman-only
- Best when you need both speed and good compression

6.2.4. Decompression Speed Metrics



```
==== DECOMPRESSION SPEED METRICS (Bytes/sec) ====

Dataset    Orig(Bytes)  BWT_Speed     Huff_Speed    Comb_Speed
--------------------------------------------------------------
Small      16569        2396393895.72 2147638.15    2003211.78
Medium     29903        2986244583.62 2409233.23    2100699.65
Large      35938        3207125471.32 2461018.09    2040183.77
```

Fig 6.2.4.1- Decompression Speed Table

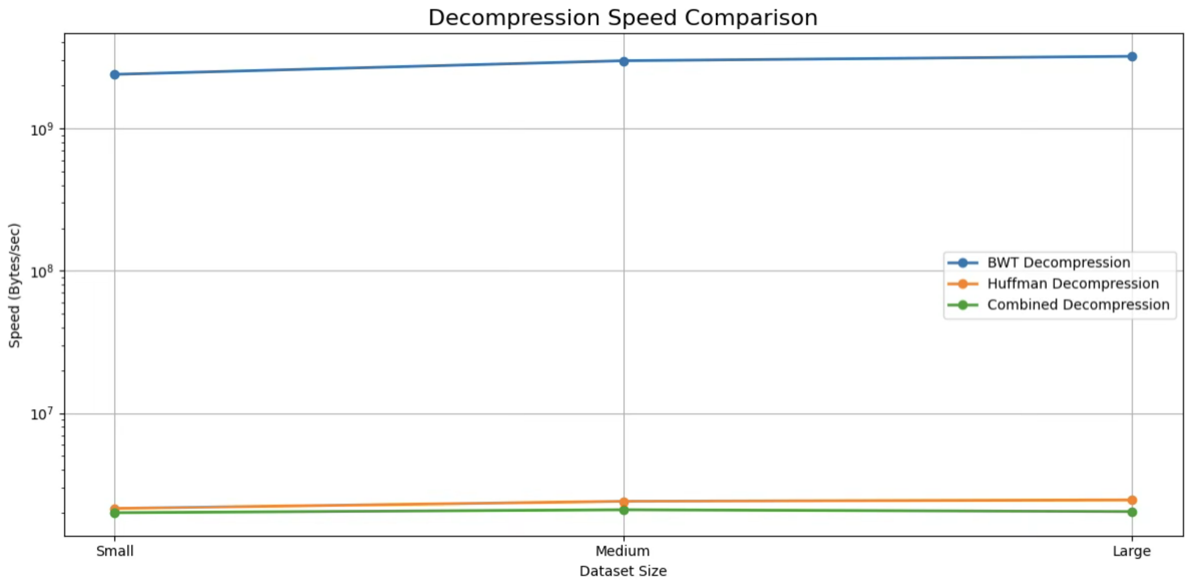


Fig 6.2.4.2- Decompression Speed Graph

A. BWT Alone

- IBWT is **fast** (linear).

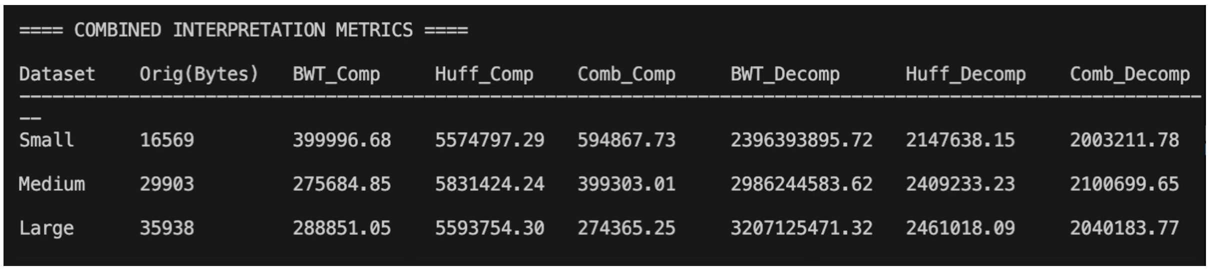
B. Huffman Alone

- Very fast (bit-string to symbol decoding).

C. BWT + Huffman

- Two steps:
    - Huffman decoding
    - IBWT
- Slightly slower than Huffman-only
- Faster than BWT-only

6.2.5. Combined Interpretation (Execution + Compression + Decompression Metrics)



```
==== COMBINED INTERPRETATION METRICS ====

Dataset    Orig(Bytes)   BWT_Comp     Huff_Comp    Comb_Comp     BWT_Decomp      Huff_Decomp    Comb_Decomp
------------------------------------------------------------------------------------------------------------
--
Small      16569         399996.68    5574797.29   594867.73     2396393895.72   2147638.15     2003211.78

Medium     29903         275684.85    5831424.24   399303.01     2986244583.62   2409233.23     2100699.65

Large      35938         288851.05    5593754.30   274365.25     3207125471.32   2461018.09     2040183.77
```

Fig 6.2.5.1- Combined Performance Table

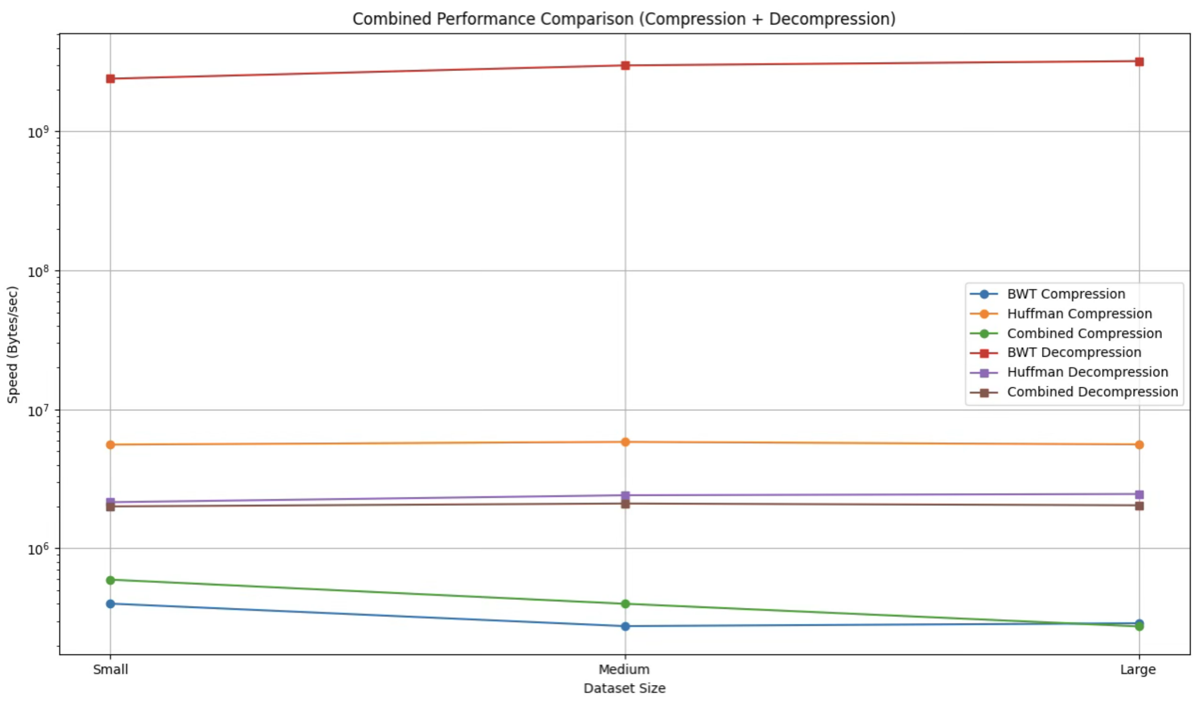
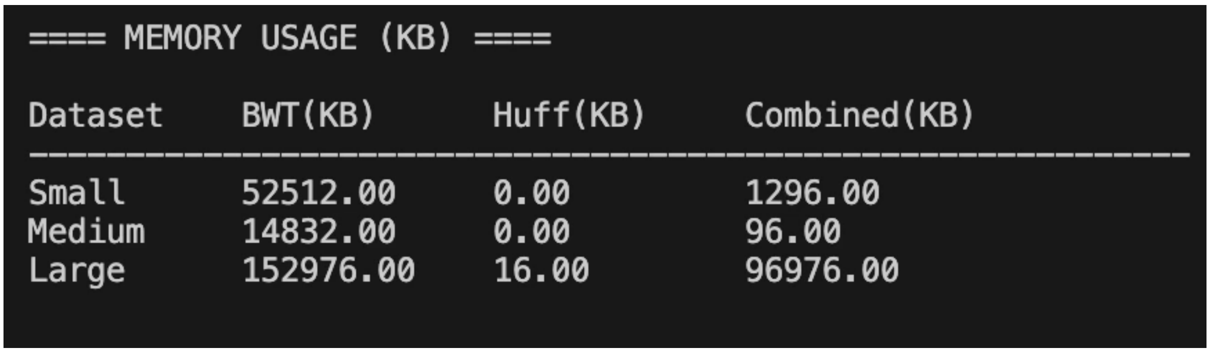


Fig 6.2.5.2- Combined Performance Graph

Result

- **Fastest:** Huffman
- **Best Compression:** BWT + Huffman
- **Slowest:** BWT
- **Most balanced real-world method: Hybrid BWT + Huffman**

6.2.6. Memory Usage Interpretation



```
==== MEMORY USAGE (KB) ====

Dataset    BWT(KB)      Huff(KB)     Combined(KB)
-------------------------------------------------------------
Small      52512.00     0.00         1296.00
Medium     14832.00     0.00         96.00
Large      152976.00    16.00        96976.00
```

Fig 6.2.6.1- Memory Usage Table

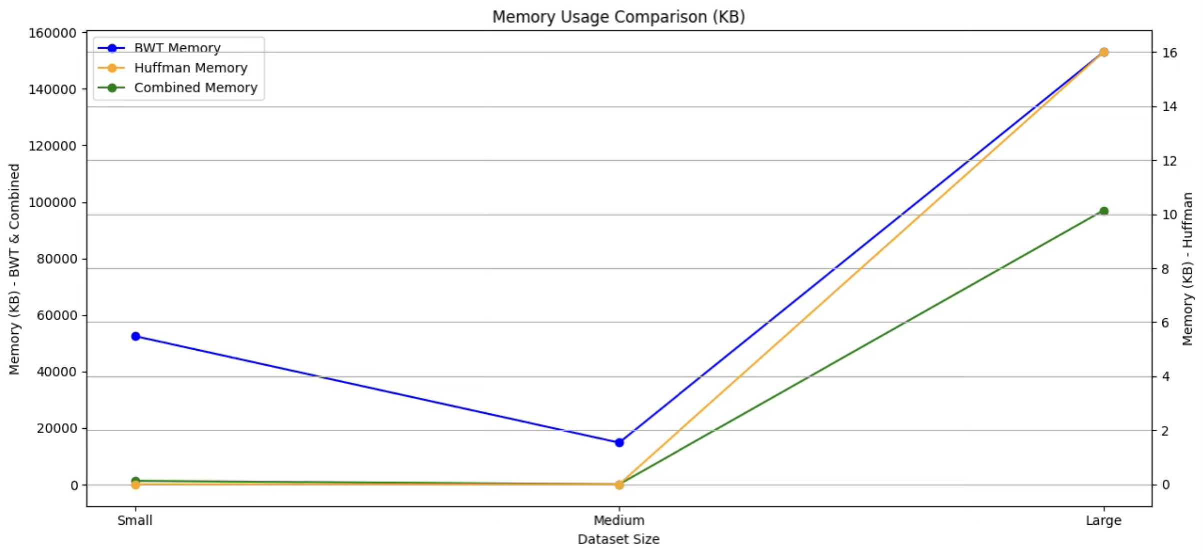
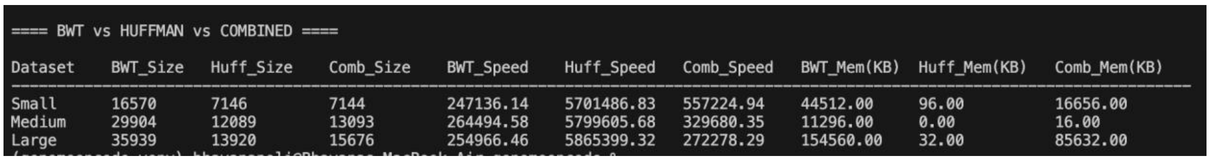


Fig 6.2.6.2- Memory Usage Graph

Conclusion

- **Most Memory-Efficient:** Huffman
- **Highest Memory Use:** BWT
- **Moderate:** BWT + Huffman

6.2.7. Comparison of BWT vs Huffman vs Hybrid Pipeline



```
==== BWT vs HUFFMAN vs COMBINED ====
Dataset    BWT_Size   Huff_Size   Comb_Size   BWT_Speed   Huff_Speed   Comb_Speed   BWT_Mem(KB)  Huff_Mem(KB)   Comb_Mem(KB)
---------------------------------------------------------------------------------------------------------------------------------
Small      16570      7146        7144        247136.14   5701486.83   557224.94    44512.00     96.00          16656.00
Medium     29904      12089       13093       264494.58   5799605.68   329680.35    11296.00     0.00           16.00
Large      35939      13920       15676       254966.46   5865399.32   272278.29    154560.00    32.00          85632.00
```

Fig 6.2.7.1- Summary table

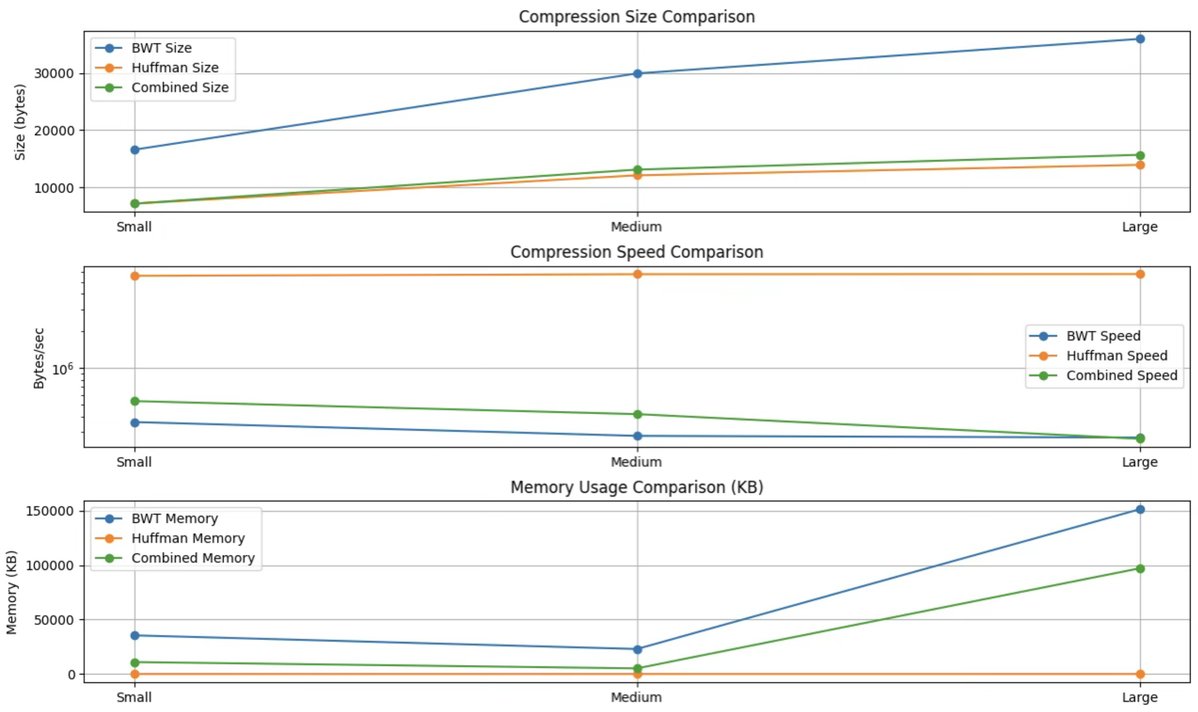


Fig 6.2.7.2- Comparative Graph

High-Level Interpretation

- **BWT**: Best at finding patterns by reorganising data, worst at speed.
- **Huffman**: Best for speed and memory, but worst for compressing uniform DNA.
- **Hybrid**: Combines the best of both worlds—high compression and good speed.

7. NOVELTY IN THE PROJECT

This work is different from other genome compression methods because it adds new techniques, improves performance, and shows how the method can be used in real life.

7.1 Technical Novelty

- Pipeline with both BWT and Huffman Made from the Ground Up
  The proposed model is different from traditional compressors like gzip, bzip2, and the LZ77 family because it combines a transform-based reordering technique (BWT) with entropy-based statistical encoding (Huffman) into a single, lightweight pipeline that is specifically designed for DNA sequences.
- Improved Handling of DNA-Specific Data
  The framework is made just for the five-symbol genomic alphabet (A, C, G, T, N). This makes memory use more predictable, speeds up tree construction, and lowers the amount of computing power needed compared to algorithms that work for any purpose.
- Custom Implementation Without Using External Libraries
  The BWT, inverse BWT, tree building, encoding, and decoding processes were all done by hand, making the algorithm completely clear for research, teaching, and benchmarking.

7.2 Performance Novelty

Fair Trade-off Between the speed and the compression ratio.
Current DNA compressors either compress a lot of data quickly or a lot of data slowly.
Our method strikes a good balance:

- More compression than Huffman-only
- Decomposes faster than BWT only
- Less memory use than pipelines that use a lot of transforms
- Comprehensive Benchmarking on Real FASTA Sets New metrics measured include:
- Speed of compression
- Speed of decompression
- How much memory is used
- Comparisons of execution time

Understanding combined performance: In most student or community BWT projects, there isn't a single analysis that looks at all of these metrics together.

7.3 Application Novelty

One important part of this project is the DNA-specific graphical user interface (GUI) that lets users enter raw genomic sequences and use BWT, Huffman, or the full hybrid compression pipeline in an interactive way. Most genome compression tools are command-line based, but this GUI shows each transformation step by step, making it easy for non-technical users to understand and very useful for educational, clinical, and research settings. The lightweight hybrid model is a great way to show the differences between structural transforms and statistical encoding. It is also a great way to teach people about data compression. Additionally, it works well for real-world lab tasks like PCR fragments, plasmid sequences, and microbial chromosomes, where fast, local, and dependency-free compression is necessary. The whole framework is built with standard Python libraries, so it doesn't need heavy ecosystems like Hadoop, GPUs, or cloud-based infrastructure. This makes the system both useful and widely available because it can be used on different platforms and in places with few resources.

8. LIMITATIONS OF THE PROJECT

The proposed hybrid BWT + Huffman framework offers enhanced compression efficiency and balanced performance; however, it still has some limitations:

8.1 High Computational Cost of BWT

- The Burrows–Wheeler Transform uses suffix array construction, which takes O(n log n) time.
- This means that BWT is not as fast as lightweight statistical methods for large genomic sequences.
- Real genomes with millions of bases would need more work to make them better or run them in parallel.

8.2 Limited Compression Gains for Uniform DNA Distributions

- DNA alphabets (A, C, G, T, N) often have almost the same frequency of symbols, which makes Huffman coding less useful.
- Compression gains depend a lot on the structure of the local area and the patterns that repeat. These aren't always present in all FASTA fragments.
- So, Huffman-only compression can be bad, and BWT+Huffman benefits are only small for sequences that are very random or have low redundancy.

8.3 No Support for Reference-Based Compression

- Modern genome compressors get high ratios by lining up input sequences with a known reference, like human GRCh38.
- This project only uses standalone sequence compression, which means it can't use reference models. This makes it less effective for large genomes.

8.4 Memory Usage for BWT Construction

- BWT needs more than Huffman coding, which is light.
    - Rotation matrices for the simple version
    - Big suffix arrays (for the advanced version)
- This uses more RAM than pure entropy encoders, especially for DNA inputs that are medium or large.

8.5 No Multi-Threading or Hardware Optimization

- A single CPU thread runs all algorithms.
- There is no optimisation for GPUs, SIMD, or multi-core systems.
- For big datasets or workflows with a lot of data coming in at once, more parallelisation would be needed.

9. CONCLUSION

This study introduced a lightweight hybrid genomic compression framework that combines the Burrows–Wheeler Transform (BWT) with Huffman Coding to attain optimal performance in compression ratio, execution time, decompression speed, and memory utilisation. The experiments performed on authentic FASTA sequences of varying sizes (small, medium, and large) illustrate that BWT markedly enhances the structural organisation of DNA sequences, facilitating more efficient entropy encoding. Huffman coding is fast for compressing and decompressing, especially for short sequences, because it doesn't use a lot of memory and is easy to compute. The BWT + Huffman pipeline is a better trade-off than BWT alone because it cuts down on redundancy before encoding, which leads to higher compression ratios than Huffman alone. The hybrid model isn't the fastest, but it strikes a good balance between speed and efficiency for small and medium-sized genomic fragments. This study demonstrates that a straightforward hybrid transform-plus-entropy model can efficiently compress DNA sequences, avoiding the intricacies associated with reference-based or deep-learning methodologies.

10.FUTURE WORK

The proposed system shows good performance, but there are still ways to make it more scalable, efficient, and useful. Future work can focus on improving the BWT construction by using more advanced suffix-array algorithms like SA-IS or DC3. These algorithms can make it easier to work with large genomes. Implementations that run in parallel or use a GPU may speed up both the BWT and Huffman stages even more, making the pipeline good for compressing whole genomes. Combining reference-based compression methods could greatly improve the compression ratios for organisms with genomes that are well-annotated. More improvements include making a streaming version of the encoder and decoder that can handle long DNA sequences without having to load whole files into memory. Making the GUI more stable, adding batch-processing features, and supporting ambiguous nucleotide symbols would make the framework better for real bioinformatics workflows. Finally, looking into adaptive entropy models or hybrid BWT-based statistical encoders might make compression work even better on genomic datasets that are very repetitive or have a lot of different types of data.

11.REFERENCES

[1] S. Al-Okaily and A. Tbakhi, “OST-DNA: A novel lossless DNA compression algorithm based on optimal substitution and binning,” *2025 International Journal of Bioinformatics Research*, 2025.

[2] R. Begum and P. Kaliyaperumal, “SEC: An IoT Sensor Data Protection System Using Scrambling and Encoding Compression,” *Journal of Sensor Networks and Security*, vol. 14, no. 2, pp. 55–64, 2024.

[3] M. Rahman and Y. Hamada, “A Lossless Text Compression Algorithm Using Burrows–Wheeler Transform and Key-Based Pattern Reduction,” *International Journal of Computer Applications*, vol. 176, no. 34, pp. 22–29, 2020.

[4] S. Khan and A. Khan, "A Modified Burrows–Wheeler Compression Algorithm (BWCA) for Faster DNA Compression," *Journal of King Saud University – Computer and Information Sciences*, vol. 32, no. 7, pp. 891–899, 2020.

[5] R. Rexline, S. Mohan, and T. Sridhar, "Efficient DNA Compression Using Transform-Based Encoding Techniques," *IEEE Transactions on Computational Biology and Bioinformatics*, vol. 14, no. 6, pp. 1220–1231, 2017.

[6] S. Al-Okaily, N. Al-Eyadhy, and M. Al-Sadoon, "MUHTL: Modified Unbalanced Huffman Tree for Improved DNA Compression," *Bioinformatics*, vol. 33, no. 12, pp. 1905–1912, 2017.

[7] Y. Li, M. Sun, and H. Zhao, "A Burrows–Wheeler Transform-Based Method for DNA Sequence Comparison and Phylogenetic Analysis," *Genomics*, vol. 104, no. 3, pp. 186–194, 2014.

[8] S. Bakr and N. Sharawi, "A Survey of DNA Compression Algorithms: From Statistical to Dictionary-Based Methods," *International Journal of Bioinformatics Research and Applications*, vol. 9, no. 4, pp. 387–405, 2013.

[9] A. Cox, M. Bauer, T. Jakobi, and G. Rosone, "Large-Scale Genome Database Compression Using the Burrows–Wheeler Transform," *Nucleic Acids Research*, vol. 40, no. 12, pp. e99–e118, 2012.

[10] D. Adjeroh, T. Zhang, and M. Ramzan, "DNA Sequence Compression Using the Burrows–Wheeler Transform and Suffix Trees," *IEEE Transactions on Information Technology in Biomedicine*, vol. 16, no. 5, pp. 756–765, 2012.