Analyzing Text Compressibility of Natural Language and Programming Language Through Huffman Coding and Shannon Entropy

Aufa Tatsbita Zahra - 13525043

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: faaa140307@gmail.com , 13525043@std.stei.itb.ac.id

***Abstract*—Data compression is an essential technique for reducing storage requirements and transmission costs. Huffman coding is a lossless compression method that assigns shorter codewords to more frequent symbols and longer codewords to less frequent symbols. This paper aims to analyze the compressibility of Indonesian text, English text, and C source code through Shannon entropy and Huffman coding. A Huffman coding program was implemented in C to compute entropy values, average codeword lengths, and compression ratios for each corpus. Experimental results show that Indonesian and English texts exhibit nearly identical compression performance, whereas C source code is less compressible due to its larger symbol alphabet and more uniform character distribution. Furthermore, the results confirm that Huffman coding closely approaches the theoretical entropy limit. These findings indicate that the statistical characteristics of a text significantly affect its compressibility.**

***Keywords*—Huffman coding, Shannon entropy, lossless compression, information theory, text compressibility.**

I. INTRODUCTION

The world currently generates approximately 2.5 quintillion bytes of data every day, equivalent to nearly 29 terabytes every second. Global data volume reached 181 zettabytes in 2025 and continues to grow at an annual rate exceeding 23.13% [1], making efficient storage and transmission of digital information fundamental challenges in computer science. From news articles and academic documents to the source code underlying modern software systems, the explosive growth of digital data has increased the importance of effective compression techniques. One of the most influential solutions to this problem is Huffman coding, introduced by David A. Huffman in 1952 [2]. Huffman coding produces variable-length, prefix-free codes by assigning shorter codewords to frequently occurring symbols and longer codewords to rare symbols, thereby minimizing the expected number of bits required to represent a message.

From a discrete mathematics perspective, Huffman coding is particularly interesting because its optimality can be established formally. The algorithm constructs a binary tree using a greedy strategy, and it can be shown that the resulting code minimizes the average codeword length among all prefix-free codes for a given symbol distribution. Moreover, Shannon's entropy provides a theoretical lower bound on the average number of bits per symbol, and Huffman coding guarantees an average codeword length that differs from this lower bound by less than one bit [3].

While the theoretical properties of Huffman coding are well established, the compression efficiency achieved in practice depends heavily on the statistical characteristics of the input data. Different types of text exhibit distinct symbol distributions. Natural languages such as Indonesian and English are shaped by linguistic patterns, while programming languages are governed by formal syntax and contain numerous operators, punctuation symbols, and whitespace characters. These differences raise an interesting question: to what extent does the category of text affect its compressibility?

To investigate this question, this paper analyzes and compares the compression behavior of three equally sized corpora consisting of Indonesian text, English text, and C source code. A Huffman coding algorithm is implemented in the C programming language and applied to each corpus. The resulting entropy values, average codeword lengths, compression ratios, and compression savings are then evaluated and compared. Experimental results are further used to examine the relationship between Shannon entropy and the practical performance of Huffman coding across different categories of text. In addition to providing an experimental comparison, this study illustrates how fundamental concepts from information theory, greedy algorithms, and binary trees are brought together in a practical data compression technique.

II. THEORETICAL BACKGROUND

A. Shannon Entropy

Information theory, introduced by Claude E. Shannon in his seminal 1948 paper "A Mathematical Theory of Communication" [4], provides a mathematical framework for quantifying the information content of a source. Central to this framework is the concept of entropy, which quantifies the average uncertainty associated with the outcomes of a random variable and, equivalently, the average amount of information conveyed by those outcomes.

For a discrete source with alphabet $X = \{x_1, x_2, ..., x_n\}$ and corresponding probability distribution $\{p(x_1), p(x_2), ..., p(x_n)\}$, the Shannon entropy H(X) is defined as:

$$H(X) = -\Sigma\, p(x_i) \log_2 p(x_i) \qquad (1)$$

where the summation is taken over all symbols with $p(x_i) > 0$, and entropy is measured in bits per symbol [4].

Intuitively, symbols that occur frequently are less surprising and therefore carry less information than symbols that occur rarely. Entropy captures this behavior by weighting the information content of each symbol, $-\log_2 p(x_i)$, by its probability of occurrence. Entropy is maximized when all symbols are equally likely and minimized when the distribution is concentrated on a single symbol.

Shannon's source coding theorem establishes the fundamental significance of entropy: no lossless compression scheme can achieve an average code length shorter than H(X) bits per symbol [4]. This makes entropy a theoretical lower bound on the compression limit of any lossless encoder operating on the same source, and it provides the benchmark against which practical coding schemes such as Huffman coding are evaluated.

B. Binary Tree

A tree is an undirected connected graph that contains no cycles. Formally, a tree with n vertices has exactly n − 1 edges [5]. When one vertex is designated as the root, the tree acquires a hierarchical structure with a defined notion of parent, child, and depth.

A binary tree is a rooted tree in which every internal node has at most two children, conventionally referred to as the left child and the right child. Binary trees can be defined recursively:

- Base case: A single node is a binary tree.
- Recursive case: A node with a left subtree and/or a right subtree, each of which is itself a binary tree, is also a binary tree.

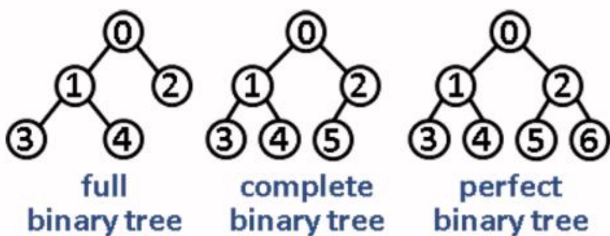

Fig. 2.1. Types of binary trees: full, complete, and perfect (Source: [6])

Several terms are relevant to the Huffman tree constructed in this paper:

- Root: the topmost node, with no parent.
- Leaf node: a node with no children; in a Huffman tree, each leaf corresponds to a distinct symbol in the input alphabet.
- Internal node: a node with at least one child; in a Huffman tree, internal nodes represent merged subtrees and carry the sum of their children's frequencies.
- Depth of a node: the number of edges on the path from the root to that node. In a Huffman tree, the depth of a leaf node equals the length of the codeword assigned to its corresponding symbol.
- Height of a tree: the maximum depth among all leaf nodes.

The recursive structure of binary trees is directly exploited in the Huffman algorithm: the tree is built bottom-up by repeatedly merging pairs of subtrees, and the resulting codewords are generated by a recursive depth-first traversal from root to leaves.

C. Prefix-Free Codes

A binary code assigns a binary string (codeword) to each symbol in an alphabet. A code is called prefix-free (or instantaneous) if no codeword is a prefix of another codeword [2]. This property is essential for unambiguous decoding: because no codeword is a prefix of another, the decoder can identify the boundary of each codeword as it reads the bit stream from left to right, without the need for explicit separators.

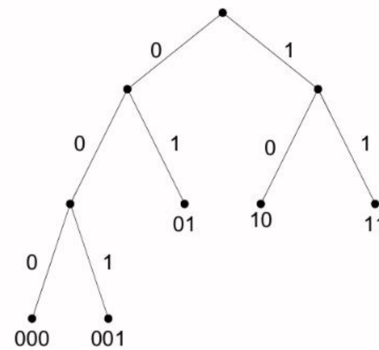

Fig. 2.2. Binary tree representation of the prefix-free code {000, 001, 01, 10, 11} (Source: [6])

The Huffman code is a prefix-free code by construction, since each symbol corresponds to a leaf node in the Huffman tree and the path from the root to any leaf is never a prefix of the path to another leaf. Because Huffman coding represents symbols as leaf nodes in a binary tree, the structural properties of binary trees play a fundamental role in determining codeword lengths and ensuring unique decodability.

D. The Huffman Coding Algorithm

Huffman coding, introduced by David A. Huffman in 1952 [2], is a greedy algorithm that constructs an optimal prefix-free code for a given symbol frequency distribution. The algorithm proceeds in two phases.

Phase 1: Tree construction. Given a set of n symbols with frequencies $f_1, f_2, ..., f_n$, the algorithm initializes a min-heap (priority queue) containing one leaf node for each symbol, keyed by frequency. It then repeatedly extracts the two nodes of minimum frequency, creates a new internal node whose frequency equals the sum of the two extracted nodes, and

reinserts the new node into the heap. This process continues until only one node remains, which becomes the root of the Huffman tree. The tree construction procedure is summarized as pseudocode in Algorithm 1.

Algorithm 1. BuildHuffmanTree(f[1..n])

```
{ Input : symbol frequencies f[1..n] }
{ Output: root node of the Huffman tree }

for i = 1 to n do
    insert leaf(i, f[i]) into min-heap H
end for

while |H| > 1 do
    left  ← extract-min(H)
    right ← extract-min(H)
    parent.freq ← left.freq + right.freq
    parent.left  ← left
    parent.right ← right
    insert parent into H
end while

return extract-min(H)   { the root }
```

Phase 2: Codeword generation. Once the tree is built, codewords are assigned by a recursive depth-first traversal starting from the root. At each internal node, the traversal appends a '0' bit when descending to the left child and a '1' bit when descending to the right child. Upon reaching a leaf node, the accumulated bit string is recorded as the codeword for the corresponding symbol. The recursive code generation procedure is summarized as pseudocode in Algorithm 2.

Algorithm 2: GenerateCode(node, code)

```
{Input: node=current tree node; code=bit str}
{Output: populated code_table[] }

if node is a leaf then
    code_table[node.symbol] ← code
    return
end if

{ recursive call }
GenerateCode(node.left, code + "0")
GenerateCode(node.right, code + "1")
```

This recursive traversal embodies a recurrence relation: the codeword for any node is formed by appending one bit to the codeword of its parent, with the base case being the empty string at the root.

Figure 2.4.1 shows an example of a Huffman tree constructed from the string "MATEMATIKA DISKRIT" [6]. Each leaf node corresponds to a symbol and its frequency, and the codeword assigned to a symbol is obtained by following the path from the root to its leaf, with left and right branches represented by bits 0 and 1, respectively. The resulting codewords are presented in Table I.

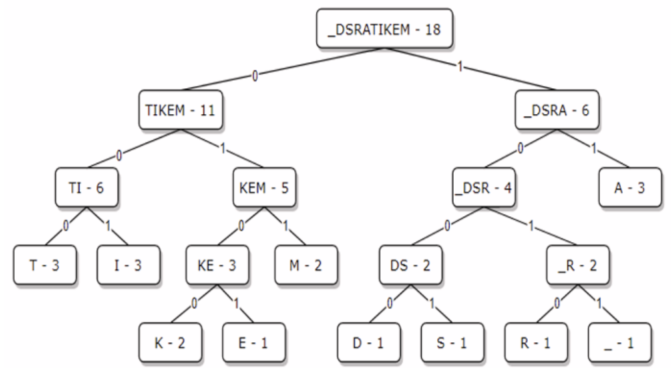

Fig. 2.4.1. Example of a Huffman tree constructed from the string "MATEMATIKA DISKRIT" (Source: [6])

TABLE I. HUFFMAN CODEWORDS FOR THE STRING "MATEMATIKA DISKRIT" (SOURCE: [6])

Simbol	Frekuensi	Kode
A	3	11
T	3	000
I	3	001
M	2	011
K	2	0100
E	1	0101
D	1	1000
S	1	1001
R	1	1010
_	1	1011

The resulting code table encodes the 18-character string "MATEMATIKA DISKRIT" using a total of 58 bits [6].

E. *Time Complexity*

Building the Huffman tree requires $n - 1$ merge operations, each of which involves two heap extractions and one insertion, each costing $O(\log n)$. The total time complexity of the algorithm is therefore $O(n \log n)$, where n is the number of distinct symbols [7].

F. *Average Codeword Length*

The average codeword length $\bar{L}$ of a variable-length code is the expected number of bits required to encode a single symbol, weighted by the probability of each symbol's occurrence. For a source with alphabet $X = \{x_1, x_2, ..., x_n\}$, symbol probabilities $\{p(x_1), p(x_2), ..., p(x_n)\}$, and assigned codeword lengths $\{l_1, l_2, ..., l_n\}$, the average codeword length is defined as [8]:

$$\bar{L} = \Sigma\, p(x_i) \times l_i \qquad (2)$$

A lower average codeword length implies greater compression efficiency, as fewer bits are needed on average to represent each symbol from the source.

The average codeword length serves as the primary metric for evaluating the practical performance of a compression code. While Shannon entropy H(X) establishes the theoretical minimum, the average codeword length measures how closely a specific code approaches that minimum in practice. The difference $\bar{L} - H(X)$ is known as the redundancy of the code [3], and quantifies the inefficiency introduced by rounding ideal codeword lengths to integer values. A well-designed code minimizes this redundancy, and Huffman coding guarantees redundancy of less than one bit per symbol.

G. *Optimality of Huffman Coding*

A fundamental result in information theory guarantees that Huffman coding produces an optimal prefix-free code. Specifically, no other prefix-free code can achieve a smaller average codeword length for the same symbol distribution.

The average codeword length $\bar{L}$, as defined in equation (2), is used here to evaluate compression performance. The following theorem, proven by Huffman [2] and formalized in information theory textbooks [3][4], establishes the relationship between $\bar{L}$ and H(X):

Theorem (Huffman Bound): For any Huffman code constructed from a symbol distribution with entropy H(X), the average codeword length $\bar{L}$ satisfies:

$$H(X) \le \bar{L} < H(X) + 1 \qquad (3)$$

The left inequality follows directly from Shannon's source coding theorem: no lossless code can compress below the entropy limit. The right inequality follows from the ceiling function applied to the ideal codeword lengths $-\log_2 p(x_i)$: rounding up to the nearest integer introduces at most one extra bit per symbol on average [4]. The quantity $\bar{L} - H(X)$ is called the redundancy of the code, and measures how far the code is from the theoretical optimum [3].

The optimality of Huffman coding can be proven by mathematical induction on n (the number of symbols), using the exchange argument: any optimal code for n symbols must assign the two longest codewords to the two least frequent symbols, and those two codewords must differ only in their last bit (i.e., they are siblings in the code tree). Assuming by inductive hypothesis that Huffman is optimal for $n - 1$ symbols, merging the two least frequent symbols into a single composite symbol and applying the inductive hypothesis yields an optimal code for the reduced source [8]; restoring the two symbols from the composite preserves optimality for the original n-symbol source.

III. Implementation

This paper implements the Huffman coding algorithm in the C programming language (C99 standard) to analyze and compare the compression efficiency of three text corpora. The program accepts three input files via command-line arguments and produces a structured analysis report for each corpus, followed by a side-by-side summary.

```
Compilation : gcc src/huffman.c -o huffman

Usage       : ./huffman datasets/indo.txt
                        datasets/english.txt
                        datasets/code.c
```

The implementation is organized into several stages corresponding to the Huffman coding pipeline. The following subsections describe the data structures, processing steps, and overall workflow used throughout the experiment.

A. *Data Structures*

Two primary structs are defined to represent the core components of the algorithm.

```c
typedef struct Node{
    unsigned char ch;
    int freq;
    int order;
    struct Node *left;
    struct Node *right;
}Node;
```

Fig. 3.1.1. Node struct

The Node struct represents a single node in the Huffman tree. Each node stores the character it represents (ch), its frequency (freq), an insertion order counter (order) used to break frequency ties deterministically, and pointers to its left and right children.

```c
typedef struct{
    char code[MAX_CODE];
    int length;
}HuffmanCode;
```

Fig. 3.1.2. HuffmanCode struct

The HuffmanCode struct stores the codeword assigned to each symbol, a bit string of up to 64 characters and its length in bits. An array of 256 HuffmanCode entries (one per ASCII symbol) constitutes the full code table produced after tree traversal.

```c
typedef struct{
    char name[100];
    int total_char;
    int unique_char;
    int freq[MAX_CHAR];
    HuffmanCode table[MAX_CHAR];
    double entropy;
    double avg_length;
    long long original_bits;
    long long compressed_bits;
    double compression_ratio;
    double saving;
}Result;
```

Fig. 3.1.3. Result struct

A Result struct aggregates all analysis outputs for a single corpus: total character count, unique character count, the frequency array, the code table, Shannon entropy, average codeword length, original and compressed bit counts, compression ratio, and space savings.

B. *File Reading and Preprocessing*

```c
int read_file(const char *filename, int freq[]){
    FILE *fp = fopen(filename,"rb");
    if(fp == NULL){
        printf("Gagal membuka %s\n", filename);
        exit(1);
    }
    int total = 0;
    int c;
    unsigned char bom[3];
    if(fread(bom,1,3,fp)==3){
        if(!(bom[0]==0xEF && bom[1]==0xBB && bom[2]==0xBF))
            rewind(fp);
    }else{
        rewind(fp);
    }
    while((c=fgetc(fp))!=EOF){
        if(c=='\n' || c=='\r') continue;
        freq[(unsigned char)c]++;
        total++;
    }
    fclose(fp);
    return total;
}
```

Fig. 3.2.1. The read_file() function

The read_file() function opens each input file in binary mode and reads it character by character. Two preprocessing steps are applied before counting:

- UTF-8 BOM detection: if the first three bytes match the UTF-8 Byte Order Mark (0xEF 0xBB 0xBF), they are skipped to prevent phantom characters from appearing in the frequency table.
- Newline normalization: carriage return ('\r') and newline ('\n') characters are excluded from the analysis. This ensures consistency across operating systems and produces character counts equivalent to those reported by text editors such as Microsoft Word.

The function returns the total number of valid characters counted, which is used as the denominator for all subsequent probability calculations.

C. *Shannon Entropy Calculation*

```c
double calculate_entropy(int freq[], int total_char){
    double H = 0;
    for(int i=0; i<MAX_CHAR; i++){
        if(freq[i] > 0){
            double p = (double)freq[i]/total_char;
            H -= p*log2(p);
        }
    }
    return H;
}
```

Fig. 3.3. The calculate_entropy() function

The calculate_entropy() function computes H(X) as defined in equation (1). It iterates over all 256 possible ASCII values, and for each symbol with a non-zero frequency, calculates its probability p = freq[i] / total_char and accumulates the contribution $-p \times \log_2(p)$. The C standard library function log2() from <math.h> is used for the logarithm computation.

D. *Huffman Tree Construction*

```c
Node* build_huffman(int freq[]){
    Heap heap;
    heap.size = 0;
    for(int i=0; i<MAX_CHAR; i++){
        if(freq[i]>0){
            heap_push(&heap, create_node(i, freq[i]));
        }
    }
    while(heap.size > 1){
        Node *left = heap_pop(&heap);
        Node *right = heap_pop(&heap);
        Node *parent = create_node(0, left->freq + right->freq);
        parent->left = left;
        parent->right = right;
        heap_push(&heap, parent);
    }
    return heap_pop(&heap);
}
```

Fig. 3.4.1. The build_huffman() function

The build_huffman() function constructs the Huffman tree using a min-heap priority queue, implemented as a manually managed array-based binary heap (the Heap struct).

```c
typedef struct{
    Node *data[512];
    int size;
}Heap;
```

Fig. 3.4.2. Heap struct

The construction follows Algorithm 1 from Section II-D:

- A leaf node is created for each symbol with freq > 0 and inserted into the heap.
- The two nodes of minimum frequency are repeatedly extracted, merged into a new internal node, and reinserted until one node remains.

Tie-breaking between nodes of equal frequency is handled via the order field: the node inserted earlier into the heap is treated as smaller, ensuring deterministic and reproducible tree construction across runs.

```c
void heap_push(Heap *heap, Node *node){
    int i = heap->size++;
    heap->data[i] = node;
    while(i > 0){
        int parent =(i-1)/2;
        if(smaller(heap->data[parent], heap->data[i])) break;
        swap(&heap->data[parent], &heap->data[i]);
        i = parent;
    }
}
```

Fig. 3.4.3. The heap_push() procedure

```c
Node* heap_pop(Heap *heap){
    Node *min = heap->data[0];
    heap->data[0] = heap->data[--heap->size];
    int i = 0;
    while(1){
        int left = 2*i + 1;
        int right = 2*i + 2;
        int smallest = i;
        if(left < heap->size && smaller(heap->data[left], heap->data[smallest])) smallest = left;
        if(right < heap->size && smaller(heap->data[right], heap->data[smallest])) smallest = right;
        if(smallest == i) break;
        swap(&heap->data[i], &heap->data[smallest]);
        i = smallest;
    }
    return min;
}
```

Fig. 3.4.4. The heap_pop() function

The heap operations, heap_push() and heap_pop(), maintain the min-heap property through upward and downward sifting respectively, each running in O(log n) time.

E. Codeword Generation

The generate_code() function implements Algorithm 2 from Section II-E: a recursive depth-first traversal of the Huffman tree that assigns a codeword to each leaf node. The function takes the current node, a mutable character buffer accumulating the bit string, the current depth, and the code table as parameters.

At each internal node, the function appends '0' to the buffer before recursing left, and '1' before recursing right. Upon reaching a leaf, the buffer contents are copied into the code table entry for that symbol. A special case handles single-symbol inputs, assigning the codeword "0" when the tree has depth zero.

```
void generate_code(Node *root, char buffer[], int depth, HuffmanCode table[]){
    if(root == NULL) return;
    if(root->left==NULL && root->right==NULL){
        if(depth==0){
            strcpy(table[root->ch].code,"0");
            table[root->ch].length = 1;
        }
        else{
            buffer[depth]='\0';
            strcpy(table[root->ch].code, buffer);
            table[root->ch].length = depth;
        }
        return;
    }
    buffer[depth]='0';
    generate_code(root->left, buffer, depth+1, table);
    buffer[depth]='1';
    generate_code(root->right, buffer, depth+1, table);
}
```

Fig. 3.5. The generate_code() function

This traversal directly embodies the recurrence relation described in Section II-E: the codeword of any node equals the codeword of its parent concatenated with one additional bit, with the base case being an empty string at the root.

F. Compression Metrics Computation

```
Result analyze(const char *filename){
    Result r;
    strcpy(r.name, filename);
    memset(r.freq, 0, sizeof(r.freq));
    memset(r.table, 0, sizeof(r.table));
    r.total_char = read_file(filename, r.freq);
    r.unique_char = 0;

    for(int i=0; i<MAX_CHAR; i++){
        if(r.freq[i]>0) r.unique_char++;
    }
    r.entropy = calculate_entropy(r.freq, r.total_char);
    Node *root = build_huffman(r.freq);
    char buffer[MAX_CODE];
    generate_code(root, buffer, 0, r.table);
    r.avg_length = 0;

    for(int i=0; i<MAX_CHAR; i++){
        if(r.freq[i]>0){
            double p = (double)r.freq[i]/r.total_char;
            r.avg_length += p * r.table[i].length;
        }
    }
    r.original_bits = (long long)r.total_char*8;
    r.compressed_bits = 0;

    for(int i=0; i<MAX_CHAR; i++){
        r.compressed_bits += (long long)r.freq[i]*r.table[i].length;
    }
    r.compression_ratio = (double)r.compressed_bits/r.original_bits;
    r.saving = (1-r.compression_ratio)*100;
    free_tree(root);
    return r;
}
```

Fig. 3.6. The analyze() function

After the code table is populated, the analyze() function computes four key metrics:

1) Average codeword length $\bar{L}$ (equation (2)): computed as the probability-weighted sum of codeword lengths across all symbols.

2) Original size: total_char × 8 bits, reflecting the baseline 8-bit ASCII representation.

3) Compressed size: the sum of freq[i] × table[i].length over all symbols, which represents the total number of bits in the Huffman-encoded output.

4) Compression ratio and space savings:

$$\text{ratio} = \text{compressed bits} / \text{original bits}$$

$$\text{saving} = (1 - \text{ratio}) \times 100\%$$

These metrics are stored in the Result struct and used in both the per-corpus report and the final comparative summary.

G. Output

The program produces two types of output.

```
void print_top10(Result r){
    TopChar top[MAX_CHAR];
    int n = 0;
    for(int i=0; i<MAX_CHAR; i++){
        if(r.freq[i]>0){
            top[n].ch = i;
            top[n].freq = r.freq[i];
            top[n].percent = 100.0*r.freq[i]/r.total_char;
            strcpy(top[n].code, r.table[i].code);
            n++;
        }
    }
    for(int i=0; i<n-1; i++){
        int max = i;
        for(int j=i+1; j<n; j++){
            if(top[j].freq > top[max].freq) max = j;
        }
        TopChar temp = top[i];
        top[i] = top[max];
        top[max] = temp;
    }
    printf("\n==========================================\n");
    printf("TOP 10 KARAKTER : %s\n", r.name);
    printf("==========================================\n");
    printf("%-12s %-10s %-10s %-20s\n", "Karakter", "Frek", "Persen", "Kode");
    printf("------------------------------------------\n");

    int limit = n<10 ? n : 10;
    for(int i=0; i<limit; i++){
        char display[20];
        if(top[i].ch==' ') strcpy(display, "(spasi)");
        else if (top[i].ch=='\t') strcpy(display, "(tab)");
        else sprintf(display, "%c", top[i].ch);
        printf("%-12s %-10d %-5.2f%%     %-20s\n", display, top[i].freq, top[i].percent, top[i].code);
    }
}
```

Fig. 3.7.1. The print_top10() function

The print_top10() function displays, for each corpus, a table of the 10 most frequent characters, showing their frequency, probability percentage, and assigned Huffman codeword. Characters are sorted by frequency in descending order using a simple selection sort. Special characters such as space and tab are printed with descriptive labels for clarity.

```
void print_summary(Result a, Result b, Result c){
    printf("\n==========================\n");
    printf("RINGKASAN HASIL PENELITIAN\n");
    printf("==========================\n\n");
    printf("%-20s %-7s %-8s %-10s %-10s %-8s %-8s\n",
        "Dataset", "Char", "Unique", "Entropy", "AvgCode", "Ratio", "Saving");
    printf("--------------------------------------------------------------------------------\n");
    Result arr[3] = {a,b,c};
    for(int i=0; i<3; i++){
        printf("%-20s %-7d %-8d %-10.4f %-10.4f %-8.4f %-7.2f%%\n",
            arr[i].name, arr[i].total_char, arr[i].unique_char, arr[i].entropy, arr[i].avg_length,
            arr[i].compression_ratio, arr[i].saving);
    }
    printf("\n");
}

int main(int argc, char *argv[]){
    if(argc!=4){
        printf("\nGunakan:\n");
        printf("./huffman indo.txt english.txt nitip.c\n");
        return 0;
    }
    Result teks1 =analyze(argv[1]);
    Result teks2 = analyze(argv[2]);
    Result teks3 = analyze(argv[3]);
    print_summary(teks1, teks2, teks3);
    print_top10(teks1);
    print_top10(teks2);
    print_top10(teks3);

    return 0;
}
```

Fig. 3.7.2. The print_summary() function

The print_summary() function displays a consolidated comparison table across all three corpora, with columns for total characters, unique characters, Shannon entropy H(X), average codeword length $\bar{L}$, compression ratio, and space savings. This table forms the primary data source for the experimental analysis in Section IV.

H. *Experimental Workflow*

The overall workflow of the experiment is illustrated in Figure 3.8. First, the input files are read and preprocessed. Character frequencies are then counted and used to compute Shannon entropy. Based on these frequencies, the Huffman tree is constructed and traversed recursively to generate codewords. Compression metrics including average codeword length, compression ratio, and space savings are subsequently computed. Finally, the results from the three corpora are summarized and compared to analyze their relative compressibility.

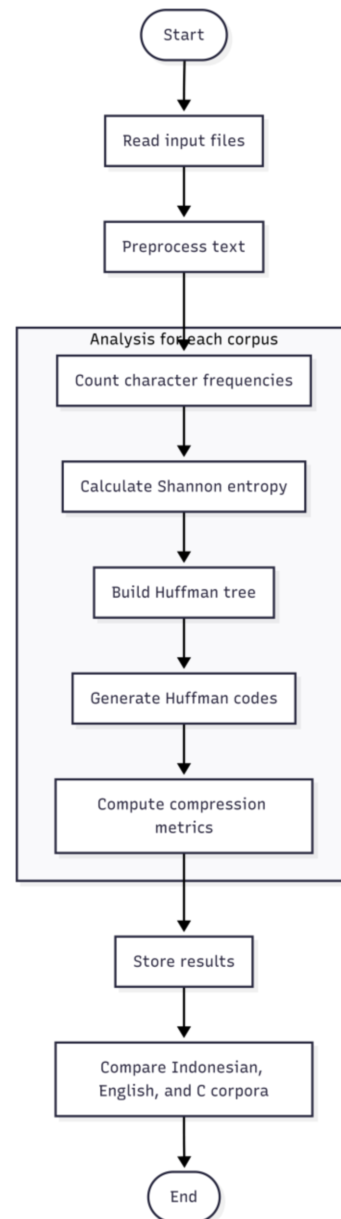

Fig. 3.8. The experimental workflow

IV. Experiment and Results

A. Experimental Setup

To evaluate the compressibility of different text categories under Huffman coding, three corpora were prepared and analyzed using the program described in Section III. Each corpus consists of exactly 5,070 characters, as the datasets were matched in size to ensure a fair and direct comparison of compression metrics. The corpora are as follows:

- Indonesian language corpus (indo.txt): a news article excerpt in Indonesian, taken from Kompas.com, reporting on student demonstrations demanding Vice President Gibran Rakabuming Raka to fulfill student demands within 5×24 hours [9].
- English corpus (english.txt): a news article excerpt in English, taken from BBC News, reporting on the preliminary US–Iran peace agreement signed at the G7 summit [10]. Although the specific topics differ, both corpora are drawn from contemporary social and political news writing, making them comparable in register and prose style.
- C source code corpus (code.c): the first 5,070 characters of the program's own source file (huffman.c), representing a real-world systems programming text. The file is intentionally truncated mid-source and is not intended to be compiled independently.

All corpora were processed in binary mode. UTF-8 BOM bytes and line-ending characters ('\r', '\n') were excluded from the analysis, producing character counts consistent with standard text editors. Analysis is case-sensitive, uppercase and lowercase letters are treated as distinct symbols.

B. Results

```
=========================
RINGKASAN HASIL PENELITIAN
=========================

Dataset               Char   Unique   Entropy   AvgCode   Ratio    Saving
--------------------------------------------------------------------------
datasets/indo.txt     5070   59       4.3970    4.4316    0.5539   44.61 %
datasets/english.txt  5070   62       4.4024    4.4331    0.5541   44.59 %
datasets/code.c       5070   76       4.7798    4.8047    0.6006   39.94 %
```

Fig. 4.2.1 Summary of Huffman coding analysis results across three corpora

```
============================================
TOP 10 KARAKTER : datasets/indo.txt
============================================
Karakter     Frek       Persen      Kode
--------------------------------------------
a            796        15.70%      111
(spasi)      647        12.76%      011
n            375        7.40 %      1100
i            356        7.02 %      1010
e            342        6.75 %      1001
s            248        4.89 %      0100
r            237        4.67 %      0011
k            224        4.42 %      0001
t            213        4.20 %      11011
m            207        4.08 %      11010
```

Fig. 4.2.2. Top 10 characters of Indonesian language corpus

```
============================================
TOP 10 KARAKTER : datasets/english.txt
============================================
Karakter     Frek       Persen      Kode
--------------------------------------------
(spasi)      814        16.06%      110
e            509        10.04%      001
a            415        8.19 %      1111
t            350        6.90 %      1010
n            341        6.73 %      1001
i            289        5.70 %      0110
r            262        5.17 %      0101
o            253        4.99 %      0100
s            227        4.48 %      0000
d            201        3.96 %      11101
```

Fig. 4.2.3. Top 10 characters of English corpus

```
============================================
TOP 10 KARAKTER : datasets/code.c
============================================
Karakter     Frek       Persen      Kode
--------------------------------------------
(spasi)      1217       24.00%      01
e            355        7.00 %      1011
t            239        4.71 %      0001
r            238        4.69 %      0000
a            195        3.85 %      11100
i            183        3.61 %      11010
o            177        3.49 %      10101
n            145        2.86 %      10010
;            138        2.72 %      10000
f            130        2.56 %      00110
```

Fig. 4.2.4. Top 10 characters of C source code corpus

C. Verification of the Huffman Bound

A key theoretical result established in Section II-G is the Huffman bound: $H(X) \le \bar{L} < H(X) + 1$. Table II verifies this bound for all three corpora.

TABLE II. Verification of The Huffman Bound

Corpus	H(X)	$\bar{L}$	H(X)+1	$H(X) \le \bar{L} < H(X)+1$
Indonesia	4.3970	4.4316	5.3970	Satisfied
English	4.4024	4.4331	5.4024	Satisfied
C Source Code	4.7798	4.8047	5.7798	Satisfied

The bound is satisfied across all three corpora, confirming that the implementation correctly produces optimal prefix-free codes. The redundancy values ($\bar{L} - H(X)$) are notably small, measuring 0.0346, 0.0307, and 0.0249 bits per character, respectively. These results indicate that the Huffman codes closely approach the theoretical entropy limit in all three cases.

D. Analysis

Character distribution. The most striking distributional difference across the three corpora is the dominance of the space character in C source code. Whitespace accounts for 24.00% of all characters in the C corpus, nearly double its frequency in English (16.06%) and Indonesian (12.76%). This is a direct consequence of C's syntactic conventions, as spaces are frequently used around operators, between function arguments, and in other language constructs. Huffman coding exploits this extreme skewness by assigning space a 2-bit codeword in the C corpus, compared with 3-bit codewords in both natural language corpora.

Indonesian vs English. The two natural language corpora show remarkably similar results across all metrics. Entropy differs by only 0.0054 bits/character (4.3970 vs 4.4024), and space savings differ by only 0.02 percentage points (44.61% vs 44.59%). This near-identical compressibility is noteworthy given the structural differences between the two languages. Indonesian is morphologically agglutinative and contains many vowel-rich affixes, while a large proportion of Indonesian words are themselves vowel-heavy. As a result, vowels account for a significant fraction of the corpus, with 'a' emerging as the most frequent character at 15.70%. English, by contrast, distributes its frequency more evenly across a broader set of common characters, with space (16.06%) and 'e' (10.04%) leading. Despite these differences, the resulting entropy values and compression ratios converge closely, suggesting that both languages, as written in natural prose, exhibit a comparable degree of statistical regularity.

Shannon entropy. The C source code corpus exhibits substantially higher entropy ($H(X) = 4.7798$ bits/char) than both natural language corpora (4.3970 and 4.4024 bits/char). This is explained by its wider active alphabet: 76 unique characters versus 59 and 62 in the Indonesian and English corpora, respectively. Programming language text makes regular use of characters that rarely appear in natural prose, including semicolons, braces, asterisks, underscores, and digits. As a result, probability mass is distributed more evenly across a larger set of symbols, leading to higher entropy. Higher entropy means the source contains more information per character and consequently leaves less room for compression.

Compression efficiency. The natural language corpora achieve approximately 44.6% space savings, reducing the encoded size from 40,560 bits to roughly 22,470 bits. In contrast, the C source code corpus achieves only 39.94% space savings, about 4.6 percentage points lower, which is consistent with its higher entropy. These findings reinforce the fundamental relationship between entropy and compressibility that sources with higher entropy contain more information per symbol and are therefore inherently more difficult to compress, as their more uniform symbol distributions require longer average codewords.

V. CONCLUSION

The compressibility of a text is fundamentally determined by its statistical structure. Through the implementation of Huffman coding and the analysis of Shannon entropy, this study showed that different categories of text exhibit different compression characteristics. Despite their linguistic differences, Indonesian and English news articles were found to possess nearly identical entropy and compression performance, indicating a similar degree of statistical regularity. In contrast, C source code exhibited higher entropy and achieved lower compression efficiency due to its larger symbol alphabet and more uniform character distribution.

Furthermore, the experimental results confirmed the theoretical optimality of Huffman coding. For all three corpora, the average codeword length satisfied the Huffman bound $H(X) \leq \bar{L} < H(X)+1$, demonstrating that the generated codes closely approached the entropy limit predicted by Shannon's information theory.

Overall, this study illustrates how concepts from discrete mathematics and information theory can be integrated to explain and analyze a practical problem in data compression. Through binary trees, prefix-free codes, greedy algorithms, and mathematical induction, Huffman coding provides an elegant bridge between theoretical principles and practical compression. The results highlight that the category and statistical properties of a text play an important role in determining its compressibility. Further studies may extend the analysis to larger corpora, additional programming languages, or other categories of text to obtain a broader understanding of text compressibility.

SOURCE CODE AT GITHUB

https://github.com/faaa140307/Makalah-Matdis/tree/main

VIDEO LINK AT YOUTUBE

https://youtu.be/ItR2YggSCYs?si=H5zvvGMBXP75-i7O

ACKNOWLEDGMENT

First and foremost, the author would like to express gratitude to God for His blessings and guidance, which made it possible to complete this paper. The author would also like to convey sincere appreciation to Dr. Ir. Rinaldi Munir, M.T., the lecturer of IF1220 Discrete Mathematics for the Even Semester of 2025/2026, Class 01, for his valuable lectures, references, and insights throughout the semester. Finally, the author would like to thank family and friends for their support and encouragement during the process of learning and writing this paper.

REFERENCES

[1] N. Kumar, “Big Data Statistics 2026 (Growth, Trends & Market Size)" DemandSage Dec, 29, 2025. [Online]. Available : https://www.demandsage.com/big-data-statistics/. [Accessed: Jun. 18, 2026].

[2] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” Proceedings of the IRE, vol. 40, no. 9, pp. 1098-1101, 1952.

[3] A. C. Blumer and R. J. Mceliece, “The Renyi Redundancy of Generalized Huffman Codes,” in 1988 IEEE Transactions on Information Theory, vol. 34, pp. 1242‑1249, 1988.

[4] C. E. Shannon, "A Mathematical Theory of Communication," Bell System Technical Journal, vol. 27, no. 3, pp. 379‑423, Jul. 1948.

[5] R. Munir, 2025. "Pohon (Bag. 1)". [Online]. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/23-Pohon-Bag1-2026.pdf. [Accessed: Jun. 16, 2026].

[6] R. Munir, 2025. “Pohon (Bag. 2)”. [Online]. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/24-Pohon-Bag2-2026.pdf. [Accessed: Jun. 16, 2026].

[7] R. Munir, 2025. “Kompleksitas Algoritma (Bagian 1)”. [Online]. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/25-Kompleksitas-Algoritma-Bagian1-2026.pdf. [Accessed: Jun. 16, 2026].

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009, pp. 428–437.

[9] D. E. Nugraheny, D. O. Purba, “5x24 Jam untuk Wapres Gibran Penuhi Tuntutan Mahasiswa,” Kompas.com, Jun. 16, 2026. [Online]. Available: https://megapolitan.kompas.com/read/2026/06/16/08381531/5x24-jam-untuk-wapres-gibran-penuhi-tuntutan-mahasiswa. [Accessed: Jun. 16, 2026].

[10] R. Flynn, “Deal to End the War Already Signed, Says Trump, as Displaced Lebanese Return Home,” BBC News, Jun. 16, 2026. Edited by J. Whitehead and C. Hadfield. [Online]. Available: https://www.bbc.com/news/live/cj0grpyg4v1t. [Accessed: Jun. 16, 2026].

PERSONAL STATEMENT

I hereby declare that this paper is my own work and that it is neither a reproduction, adaptation, nor translation of another person's work, and that it does not contain any form of plagiarism.

Bandung, June 19, 2026

Aufa Tatsbita Zahra 13525043