Universal Source Coding: Explorations in Information Theory and Encoding Schemes

Himadri Mandal (BS2327)
Siddhartha Bhattacharya (BS2345)
Abhik Rana (BS2301)
Ayan Ghosh (BS2321)

Advisor/Instructor
DR. ARNAB CHAKRABORTY
Associate Professor, Applied Statistics Unit

Monday 27$^{\text{th}}$ November, 2023

Universal Source Coding: Explorations in Information Theory and Encoding Schemes

Abstract

Morse code sends English characters using dots and dashes. The letter E is represented by a single dot. This is reasonable because E happens to the most frequently occuring letter in normal usage. So assigning the shortest code to it makes sense from the viewpoint of length optimisation. The intuitive idea of "more frequent inputs should be mapped to shorter outputs" gets rigorised in the so-called Huffman encoding scheme. It turns out to be the optimal encoding scheme given you know the source distribution. This project tries to answer the natural extension to that question: to come up with an encoding scheme that guarantees a minimum performace against all input probability distributions, *algorithmically*.

1 Introduction

The broad setup is the following: there's data coming in from some source. If the source distribution is known, then Huffman Encoding gives the optimal encoding scheme. This project tries to figure out a good encoding scheme (in multiple contexts!) that guarantees performance against all source distributions! There are many ways to go about this project, and we thought it would be a very big missed opportunity if we just explore one of them, so we decided to pursue our creativity and explore the multiple perspectives we found out!

Himadri Mandal and Abhik Rana have improved the **Lempel-Ziv-Welch** algorithm in a particular context, Siddhartha Bhattacharya has worked on a **Huffman Encoding focused approach** and Ayan Ghosh has worked on something he calls **Method of Types**.

There were a few other approaches that we also worked on but those weren't particularly promising. First, we would like to go through the transition of ideas we went through.

2 Initial Ponderings

Context

Data comes in character by character. We don't know the source distribution of the data. Therefore, we try to estimate the source distribution and use the many techniques available thereon.

If you know the source distribution already, then you're done. What we can do is try to define a "universe" of distributions and figure out a good "center" of this universe. By universe I mean a probability simplex:

Proposition

Let $\{p_1, p_2, \cdots, p_m\}$ be a set of pmfs. Define the universe of probabilities to be

$$\mathcal{U} = \left\{ \alpha_1 \cdot p_1 + \cdots + \alpha_m \cdot p_m \middle| \alpha_1 + \cdots + \alpha_m = 1, 0 \leq \alpha_i \leq 1 \right\}$$

Proposition

Let $\chi_d(P, Q)$ be a measure of "difference" between two probability distributions P, Q. Let $\mathcal{U}$ be a universe of probabilities. Define the two following centers of universe: $\mathcal{P}_1$ and $\mathcal{P}_2$

$$\mathcal{P}_1 = \arg\min_P \min_Q \mathop{\mathbb{E}}_{\sim \mathcal{U}} [\chi_d(P, Q)]$$

$$\mathcal{P}_2 = \inf_{P \sim U} \sup_{Q \sim U} \chi_d(P, Q)$$

$\mathcal{P}_1, \mathcal{P}_2$ give best average performace and best worst performance, respectively.

The measure of difference between two probability distribution we tried to work with was $\chi_d(P, Q) = \mathrm{D}_{\mathrm{KL}}(Q||P)$. The idea is the following:

- Figure out a good way to fix the universe $\mathcal{U}$.
- Find the centers $\mathcal{P}_!$, $\mathcal{P}_2$ using gradient descemt or such.
- Obtain encoding schemes H_1, H_2 accordingly. Choose as need be.

We believe there is promise in this idea, and should be explored further. However, we figured out new perspectives to look at this problem and so didn't devote any more time into this.

3 Bubbly Lempel-Ziv-Welch

3.1 Theory

We improve the **Lempel-Ziv-Welch** algorithm. We named this "Bubbly" because the initial versions of our work somehow represented the bubble sort.

Context

Lossless File Compression. We have a file F with letters (a-z). We compress the file F and create $E(F)$ in a way that if the **Decoder** has the knowledge of $E(F)$ and the encoding scheme $E(\cdot)$ then it can rediscover F with no loss. Ofcourse we care about time and space complexities.

Proposition (Lempel-Ziv-Welch)

Define dictionary D with all characters in the input alphabet and their encodings. Define a running string variable Pattern, and iteratively add the next character to it. If the resulting string is already in D, continue reading characters until you find a string that is *not* in the dictionary D. Add it to D with its encoding. Then, encode Pattern as $D[\text{Pattern}[0:-1]] + \text{Pattern}[-1]$. Continue.

Example

Text = AABABBABBAABABBA
Dictionary: A - 1, AB - 2, ABB - 3, ABBA - 4, ABA - 5, B - 6, BA - 7
Encoded = A1B2B3A2A#B6A

Canonically, the encodings of D follow a monotonic ordering in the time axis. This is to prevent what we call "ambiguities" as we see. To improve this algorithm: we attack the way encodings are determined. Clearly, in the attempt to make the algorithm simple and the decoding direct **LZW** makes no effort in determining optimal encodings.

To do this, we need two things:

- For any permutation $\xi(D)$ of the encodings, we should be able to perform the decoding scheme *without* a lookup table!
- Optimize $\xi(D)$ for length of text.

The second part is easier to ensure, so let's do that first.

Proposition (Bubbly LZW: Permutation optimization)
Obtain the normal **LZW** dictionary D, and find the number of times each encoding is obtained, for a phrase X call this Counter(X). Sort D in decreasing order according to the key $f(v) = \text{len}(v) \cdot \text{Counter}(v)$.

Define $P_D = \text{SortedD}^{-1}$ such that $\text{SortedD}[P_D(x)] = x$. P_D is the optimized permutation.

Okay, let's try to figure out the first obstacle now.

Definition (Phrase)**.** Phrases are of three types:

1. Parent: (nX), where n is an encoding whose definition can not be realized given the text before this phrase and X is some character.
2. Child: (nX), where n is an encoding whose definition can be realized using the text before this phrase and X is some character.
3. New: (#X), where X is some character.

Proposition (Phrase Decomposition)
Let $\xi(D)$ be a permutation of the encoding. Using this permutation of the encoding, encode the text. Obtain E_ξ(Data). Define **Phrase Decomposition** of E_ξ(Data) to be a decomposition of the encoded text into phrases.

Example
Text = #a3b1b2a1a#b4a
Phrase Decomposition = (#a)(3b)(1b)(2a)(1a)(#b)(4a)

As the decoder reads through the encoded, there could arise what we call "ambiguities". An example of such a thing is:

Example (Ambiguity)
Phrase Decomposition = (#a)(3b)(1b)(2a)(1a)(#b)(4a)
Here "(1a)" is an ambiguous phrase whose meaning cannot be derived by the decoder using the phrases before it.

When we obtain ambiguities it is necessary for us to define it out for the decoder to then use it. Here is a characterisation of ambiguities:

Theorem (Characterisation of The First Ambiguity)

A phrase P is the first ambiguity $\iff$ the phrase P is the last parent phrase before the first child phrase (OR) the last parent phrase before the first new phrase.

This leads us to the algorithm of finding out the ambiguities and then solving them:

Algorithm 3.1 Recursively solve ambiguities.

```
procedure SolveAmbiguity(Encoded)
    while IsAmbiguity: True do
        Find the first occurence of one of the two:
            (parent)(child)
            (parent)(new)
        Perform the definition of the (parent)
        Mark the next phrase a parent, and continue.
    end while
end procedure
```

Proposition (Defining a Parent)

By defining a parent phrase P, we mean giving the decoder some information that can then be used to determine the encoding of P unambiguously. Here we used the most naive idea: we defined a parent phrase P by attaching its definition to it.

Albeit a bit complicated, here's the decoder algorithm:

```
class Decoding:
    def __init__(self, unambiguous):
        self.code = unambiguous

    def nearest_closing_bracket(self, idx):
        j = 1
        while self.code[idx+j] != ')':
            j+=1
        return idx+j, self.code[idx+1:idx+j]

    def farthest_ending_digit(self, idx):
        j = 0
        while self.code[idx+j].isnumeric():
            j += 1
        return idx+j-1, self.code[idx:idx+j]

    def return_key(self, dictionary, idx):
        return list(dictionary.keys())[list(dictionary.values()).index(idx)]

    def decoder(self):
    dictionary = {}
        decode = ''
        i = 0
        while i < len(self.code):
            if self.code[i] == '#':
                if self.code[i+2] == '(':
                    ending_bracket, inside = self.nearest_closing_bracket(i+2)
                    dictionary[self.code[i+1]] = int(inside)
                    i = 1+ending_bracket
                    decode += self.code[i+1]
                else:
                    ending_location, address = self.farthest_ending_digit(i+2)
                    dictionary[self.code[i+1]] = int(address)
                    i = 1 + ending_location
                    decode += self.code[i+1]
            else:
                j, prev = self.farthest_ending_digit(i)
                if self.code[j+2] == '(':
                    ending_bracket, inside = self.nearest_closing_bracket(j+2)
                    dictionary[self.return_key(dictionary, int(prev))+self.code[j+1]]
                        = int(inside)
                    decode += self.return_key(dictionary, int(prev))+self.code[j+1]
                    i = 1+ending_bracket
                else:
                    ending_location, address = self.farthest_ending_digit(j+2)
                    dictionary[self.return_key(dictionary, int(prev))+self.code[j+1]]
                        = int(address)
                    decode += self.return_key(dictionary, int(prev))+self.code[j+1]
                    i = 1 + ending_location
        print(dictionary)
        return decode
```

3.2 Results and Observations

4 Huffman Must Encode

4.1 Huffman Encoding

Formalized description

Input. Alphabet $A = (a_1, a_2, \ldots, a_n)$, which is the symbol alphabet of size n. Tuple $W = (w_1, w_2, \ldots, w_n)$, which is the tuple of the (positive) symbol weights (usually proportional to probabilities), i.e. $w_i = \text{weight}(a_i), i \in \{1, 2, \ldots, n\}$.

Output. Code $C(W) = (c_1, c_2, \ldots, c_n)$, which is the tuple of (binary) codewords, where c_i is the codeword for $a_i, i \in \{1, 2, \ldots, n\}$

Goal. Let $L(C(W)) = \sum_{i=1}^{n} w_i \text{ length } (c_i)$ be the weighted path length of code C. Condition: $L(C(W)) \leq L(T(W))$ for any code $T(W)$.

4.2 Proof of Optimality

Recall that the problem is given frequencies $f_1, \ldots, f_n$ to find the optimal prefix-free code that minimizes

$$\sum_{i}^{n} f_i \cdot \text{ (length of encoding of the } i\text{-th symbol)}.$$

This is the same as finding the full binary tree with n leaves, one per symbol in $1, \ldots, n$, that minimizes

$$\sum_{i=1}^{n} f_i \cdot (\text{ depth of leaf of the } i\text{-th symbol})$$

Recall that we showed in class the following key claim. Claim 1 (Huffman's Claim). There's an optimal tree where the two smallest frequency symbols mark siblings (which are at the deepest level in the tree).

We proved this via an exchange argument. Then, we went on to prove that Huffman's coding is optimal by induction. We repeat the argument in this note.

Claim 2. Huffman's coding gives an optimal cost prefix-tree tree. Proof. The proof is by induction on n, the number of symbols. The base case $n = 2$ is trivial since there's only one full binary tree with 2 leaves.

Inductive Step: We will assume the claim to be true for any sequence of $n - 1$ frequencies and prove that it holds for any n frequencies. Let $f_1, \ldots, f_n$ be any n frequencies. Assume without loss of generality that $f_1 \leq f_2 \leq \ldots \leq f_n$ (by relabeling). By Claim 1, there's an optimal tree T for which the leaves marked with 1 and 2 are siblings. Let's denote the tree that Huffman strategy gives by H. Note that we are not claiming that $T = H$ but rather that T and H have the same cost.

We will now remove both leaves marked by 1 and 2 from T, making their father a new leaf with frequency $f_1 + f_2$. This gives us a new binary tree T' on $n - 1$ leaves with frequencies $f_1 + f_2, f_3, f_4, \ldots, f_n$. We do the same for the Huffman tree giving us a tree H' on $n - 1$ leaves with frequencies $f_1 + f_2, f_3, f_4, \ldots, f_n$. Note that H' is exactly the Huffman tree on frequencies $f_1 + f_2, f_3, f_4, \ldots, f_n$ by definition of Huffman's strategy. By the induction hypothesis,

$$\text{cost}(H') = \text{cost}(T').$$

Observe further that

$$\text{cost}(T') = \text{cost}(T) - (f_1 + f_2)$$

since to get T' from T we replaced two nodes with frequencies f_1 and f_2 at some depth d with one node with frequency $f_1 + f_2$ at depth $d - 1$. This lowers the cast by $f_1 + f_2$. Similarly,

$$\text{cost}\left(H'\right) = \text{cost}(H) - \left(f_1 + f_2\right).$$

Combining the three equations together we have that

$$\text{cost}(H) = \text{cost}\left(H'\right) + f_1 + f_2 = \text{cost}\left(T'\right) + f_1 + f_2 = \text{cost}(T).$$

4.3 Variant-Adaptive Huffman Coding

Motivating the Variant of Adaptive Huffman Coding schema In consideration of a variable character distribution, we engage with data processing tasks wherein data arrives in sequential chunks. Our objective is to encode each incoming chunk efficiently while simultaneously maintaining a dictionary structure to facilitate straightforward decoding operations. This process is fundamental for tasks such as data compression, where the adaptability of encoding schemes to fluctuating character frequencies is paramount.

To achieve this, we implement an Adaptive Huffman coding algorithm. This algorithm dynamically adjusts its encoding tree structure as new symbols are encountered, ensuring adaptability to evolving character distributions. The essence of our approach lies in the concurrent encoding of incoming data chunks and the maintenance of a dictionary that correlates encoded symbols to their respective characters. This dictionary plays a crucial role in the decoding process, enabling the reconstruction of the original data from its encoded representation.

Formally, our Adaptive Huffman coding implementation comprises the following components:

1. **Data Encoding**: - As each chunk of data arrives, we encode it using the Adaptive Huffman algorithm. This involves traversing the encoding tree to generate binary representations of the input symbols.

2. **Dictionary Maintenance**: - Simultaneously, we update and maintain a dictionary that maps each symbol to its corresponding binary code. This dictionary serves as a reference for decoding operations and evolves dynamically alongside the encoding process.

3. **Dynamic Tree Adjustment**: - The Adaptive Huffman algorithm dynamically adjusts the encoding tree structure based on the frequency of encountered symbols. This adaptability ensures optimal encoding efficiency, with more frequently occurring symbols assigned shorter binary codes.

4. **Decoding Process**: - To decode encoded data, we utilize the maintained dictionary to efficiently map binary codes back to their original symbols. The dynamic nature of the encoding tree ensures that decoding operations remain efficient and accurate across varying character distributions.

By formalizing these components, we establish a robust framework for encoding and decoding sequential data chunks while accommodating diverse character distributions. This approach ensures efficient data compression and facilitates seamless communication and storage of information in real-world applications.

4.4 Coding it

4.4.1 Charecter counter

Here is the code which takes in continuous data from the user

```
#File name : number_of_char_counter.py
import numpy as np

def count_ascii_characters(page_content):
    # Initialize a list to store the counts of ASCII characters
    ascii_counts = [0] * 128  # Initialize with zeros for ASCII characters 0 to 127
```

```
7
8       # Iterate over each character in the page content
9       for char in page_content:
10          # Check if the character is an ASCII character
11          ascii_val = ord(char)
12          if ascii_val < 128:
13              # Increment the count for the ASCII character
14              ascii_counts[ascii_val] += 1
15
16      return ascii_counts
17
18  def create_ascii_frequency_matrix(book_file):
19      ascii_frequency_matrix = []
20
21      # Open the book file and read its contents page by page
22      with open('C:\\Users\\inoxb\\Downloads\\input.txt', 'r', encoding='utf-8') as file:
23          for page_content in file:
24              # Count the ASCII characters in the page
25              page_ascii_counts = count_ascii_characters(page_content)
26
27              # Append the ASCII frequency counts of the page to the matrix
28              ascii_frequency_matrix.append(page_ascii_counts)
29
30      # Convert the list of lists to a numpy array (matrix)
31      ascii_frequency_matrix = np.array(ascii_frequency_matrix)
32
33      return ascii_frequency_matrix
34
35  # Example usage: Replace 'book.txt' with the path to your book file
36  book_file_path = 'C:\\Users\\inoxb\\Downloads\\input.txt'
37  ascii_frequency_matrix = create_ascii_frequency_matrix(book_file_path)
38
39  # Print the ASCII frequency matrix
40  print("ASCII Frequency Matrix:")
41  print(ascii_frequency_matrix)
```

4.4.2 Regression!

Why regression you may ask! In traditional static Huffman coding, the probability weights of the Huffman encoding are directly proportional to the probabilities of the symbols in the input stream. This means that symbols with higher probabilities are assigned shorter codewords, leading to efficient compression.

However, there are scenarios where the probability weights of the Huffman encoding may not be directly proportional to the probabilities of the symbols. This typically occurs in adaptive Huffman coding, where the encoding tree is updated dynamically as symbols are encountered. Here are a few situations where this can happen:

1. **Initial Encoding**: In adaptive Huffman coding, the encoding tree starts with a predefined structure, often containing only an escape symbol. During the initial encoding phase, when symbols are encountered for the first time, they are added to the tree without any regard to their probabilities. This means that initially, the probability weights of the codewords may not be proportional to the probabilities of the symbols.

2. **Dynamic Updates**: As symbols are encountered and added to the encoding tree, the tree's structure changes dynamically. Symbols with higher frequencies move closer to the root of the tree, resulting in shorter

codewords. However, this process is not instantaneous and requires multiple occurrences of a symbol to affect its position in the tree. Therefore, there may be periods during which the probability weights of the codewords do not precisely reflect the current probabilities of the symbols.

3. **Tree Balancing**: In adaptive Huffman coding, the encoding tree may become unbalanced due to long sequences of identical symbols or other factors. To maintain efficiency, the tree may need to be rebalanced or reset periodically. During these operations, the probability weights of the codewords may temporarily deviate from the actual probabilities of the symbols.

4. **Escape Mechanism**: Adaptive Huffman coding often includes an escape mechanism to handle unknown symbols or to reset the encoding tree. This escape symbol may have a fixed probability weight or may be assigned dynamically based on the current state of the encoding tree. In either case, the probability weight of the escape symbol may not correspond directly to the probability of encountering unknown symbols in the input stream.

In summary, in adaptive Huffman coding, the probability weights of the Huffman encoding may not always be strictly proportional to the probabilities of the symbols due to the dynamic nature of the encoding tree and the need for adaptive updates and balancing. However, over time and with sufficient data, the encoding tends to converge to an efficient representation that closely approximates the true symbol probabilities.

We write the regression problem as follows

$$\min_{w \in \mathcal{S}} ||y - Xw||_2^2 + \lambda_1 ||w||_1 + \lambda_2 ||w||_2^2$$

with the given constraints $\Sigma_{i=1}^{n} w_i = 1$ with $w_i \geq 0 \forall i \in 1, \cdots, n$

Here $y = [y_1, y_2, \cdots, y_n]'$ is a $n \times 1$ vector of the total number of characters in $X = (x_{i,j})_{n \times 128}$ is the $n \times 128$ matrix of returns on the 128 is the number of ascii characters and n chunks of code here $w = [w_1, w_2, \cdots, w_n]'$ is a $p \times 1$ vector of weights to be determined to for minimizing the error.$x_{i,j}$ represents the number of occurences of the j^{th} ascii character in the i^{th} row **Exponential Gradient Descent** We apply Exponential Gradient Descent to solve the regression problem which is the following We assume the decision space to be the following S is a d-dimensional simplex that is

$$S = \{w | w_i \geq 0 \text{ and } ||w||_1 = 1\}$$

In exponentiated gradient descent at time $t = 1$ we choose the central point of the simplex namely $w_{i,d} = \frac{1}{d}$ then we update in the following manner:

$$\forall i \in [d], w_{t+1,i} = \frac{w_{t,i} \exp\left\{-\eta \left[\nabla c_t\left(w_t\right)\right]_i\right\}}{Z_t}$$

where

$$Z_t = \sum_i w_{t,i} \left[\nabla c_t\left(w_t\right)\right]_i$$

Here $[]_i$ denotes the ith component of the vector.The division by Z_t normalizes so that $w_{t+1} \in S$ that is $||w_{t+1}||_1 = 1$ The file which implements the above is named '$\text{egd}_{for_{stat_{proj_{v}}}}1.py'$

4.4.3 Huffman Encoding Code

```
#Name Of the file: huffman_coding_v1.py
import heapq
from collections import defaultdict

# Import w_optimal from egd_for_stat_proj_v1.py
from egd_for_stat_proj_v1 import w_optimal

```

Algorithm 4.1 Exponential Gradient descent

Require: convex $f : \Delta_n \to \mathbb{R}$

Ensure: : $\eta \geq 0, T > 0$

1: Set $p^0 = \frac{1}{n}1$ (The uniform Distribution) $\in \Delta_n$

2: **for** $t = 0 : T - 1$ **do**

$g^t := \nabla f(p^t)$

$w^{t+1} := p_i^{t+1} e^{-\eta g_i^t}$

$p_i^{t+1} := \frac{w_i^{t+1}}{||w_{t+1}||}$

3: **return** $\bar{p} = \frac{1}{T} \sum_{t=0}^{T-1} p^t$

```
# Read the text from the file
file_path = 'C:\\Users\\inoxb\\Downloads\\input.txt'
with open(file_path, 'r', encoding='utf-8') as file:
    text = file.read()

# Define the Huffman coding function
def huffman_coding(text, weights):
    # Create a dictionary to store the frequency of each character
    frequency = defaultdict(int)
    for char in text:
        frequency[char] += 1

    # Build the Huffman tree
    heap = [[weight, [char, ""]] for char, weight in frequency.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

    # Create a dictionary to store the Huffman codes
    huffman_codes = {}
    for char, code in heap[0][1:]:
        huffman_codes[char] = code

    # Encode the text using Huffman codes
    encoded_text = ''.join(huffman_codes[char] for char in text)

    return encoded_text, huffman_codes

# Apply Huffman coding with the weights from w_optimal
encoded_text, huffman_codes = huffman_coding(text, w_optimal)

# Print the encoded text
print("Encoded text:")
print(encoded_text)
```

```

# Print the mapping of ASCII characters to binary strings
print("\nCharacter — Binary String Mapping:")
for char, code in huffman_codes.items():
print(f"'{char}' — {code}")
```

4.4.4 Huffman Decoding

```
#Filename: huffman_decoder_v1.py
from huffman_coding_v1 import encoded_text
from huffman_coding_v1 import huffman_codes
def huffman_decoder(encoded_texting, huffman_mapping):
    decoded_text = ""
    current_code = ""

    # Invert the huffman_mapping to map codes to characters
    inverted_mapping = {code: char for char, code in huffman_mapping.items()}

    for bit in encoded_texting:
        current_code += bit
        if current_code in inverted_mapping:
            decoded_text += inverted_mapping[current_code]
            current_code = ""

    return decoded_text

# Example usage:
# encoded_text is the Huffman encoded binary string
# huffman_mapping is the mapping of characters to their Huffman codes
# Replace these with your actual encoded text and mapping
encoded_texting = encoded_text
huffman_mapping = huffman_codes

decoded_text = huffman_decoder(encoded_text, huffman_mapping)
print("Decoded Text:", decoded_text)
```

4.5 Combining the Above

To implement the following algorithm effectively, it's crucial to understand its key components and the sequential flow of operations. Here's a formal description of the algorithm:

1.**Input Data Processing**: - As a user continuously inputs data into a file, the module 'number of char counter.py' processes each chunk of data. - Upon receiving a chunk of data, the 'number of char counter.py' module runs the 'egd for stat proj v1.py' file. This process computes the optimal weights based on the received data.

2. **Huffman Encoding**: - Using the obtained optimal weights, the module 'huffman encoder.py' encodes the data and generates a dictionary mapping each character to its corresponding binary string.

3. **Dictionary Concatenation**: - When a user writes another piece of text, the same process is repeated to obtain optimal weights and create a corresponding dictionary. - If the ordering of weights remains the same, indicating no significant changes in the character frequencies, the new dictionary is concatenated with the

previous one. - If the ordering of weights changes, suggesting a shift in character frequencies, the differences in the encoding scheme are noted and incorporated into the dictionary.

4. **Continuous Operation**: - This process continues iteratively as the user inputs more data. At each step, the algorithm adapts to changes in the data and updates the encoding scheme accordingly.

5. **Decoding**: - During decoding: - For the first step, the data is decoded directly using the original dictionary. - For subsequent steps, changes in the dictionary made during encoding are noted, and decoding is performed accordingly to account for these changes.

This algorithm allows for adaptive encoding and decoding, where the encoding scheme adjusts dynamically based on the observed data patterns. By continuously updating the encoding dictionary, the algorithm efficiently adapts to changes in the input data distribution while maintaining the ability to decode previous and current data streams accurately.

First we try and formalize how large should be a chunk of data.Suppose a monkey is randomly hitting keys on a typewriter with all the 128 ascii characters.The expected number of hits required to hit all the characters is $128\mathrm{H}_{128} where H_{128} denotes the 128th harmonic number this fact comes from the Expectation of coupon collecting problem$

We now formally show how the dictionary keeps on getting updated so at every step we dont have to save the dictionary and the process can be described as follows:

1. **Initialization**: - Given an initial encoding scheme E_1, weight vector W_1, and dictionary D_1, encode the first chunk of text using E_1 and maintain D_1.

2. **Updating Encoding Scheme**: - Upon adding a new chunk of text with weight vector W_2, construct a new dictionary D_2 corresponding to the new text. - Check the monotonicity of W_2 and $W_1 + W_2$: - If the monotonicity of W_2 matches $W_1 + W_2$, continue using E_1. - If the monotonicity of W_2 differs from $W_1 + W_2$: - Determine the smallest number of permutations required for W_2 to achieve the same monotonicity as $W_1 + W_2$. - Update D_1 accordingly by applying the permutations, ensuring that the changes preserve existing encodings as much as possible. - Use the modified dictionary D_1 to encode the new chunk of text with E_1.

3. **Recursion and Separate Computation**: - Repeat the above process recursively for subsequent chunks of text. - If the set of indexes of non-zero weights of W_2 is a subset or equal set of indexes of non-zero weights of W_1: - Check if the monotonicity of $W_1 + W_2$ and W_2 is preserved. - If preserved, continue with E_1; otherwise, update D_1 and E_1 as described above. - If none of the indexes of non-zero weights of W_2 is a subset or equal set of indexes of non-zero weights of W_1: - Compute the dictionary D_2 separately. - If the indexes of non-zero weights of W_2 have a non-zero intersection with the indexes of non-zero weights of W_1: - Again, compute the dictionary D_2 separately.

This formal description outlines the procedure for adaptively updating the encoding scheme and dictionary based on changes in the weight vectors of chunks of text. It accounts for scenarios where the monotonicity changes and where the sets of non-zero weights intersect or are disjoint between weight vectors.

Now we can recursively keep on storing the dictionary by noting the changes being made so we dont have to collectively store a complete dictionary everytime we add a new chunk of data which will make the decoding a simple recursive process

The following summarizes this in the manner of a huffman tree 1. **Initial Encoding Tree**: - Adaptive Huffman encoding starts with an initial encoding tree that represents a fixed initial dictionary mapping characters to binary codes. This initial tree is commonly constructed using a predefined method like ASCII or fixed-length codes.

2. **Tree Update on New Symbol Arrival**: - As new symbols are encountered in the input stream, the encoding tree is updated dynamically. If the new symbol is encountered for the first time, it is added to the encoding tree with a new code. If the symbol has been seen before, the encoding tree is adjusted to maintain the Huffman coding property, typically by swapping nodes to maintain the optimal encoding.

3. **Tree Traversal for Encoding**: - To encode a symbol, the encoding tree is traversed from the root to the leaf node corresponding to the symbol. The binary code associated with that leaf node is then output as the encoded representation of the symbol.

4. **Dynamic Nature of the Encoding Tree**: - The encoding tree evolves continuously as new symbols are encountered. This dynamic nature of the tree allows it to adapt to changes in the frequency distribution of symbols in the input stream.

5. **Implicit Dictionary Preservation**: - Since the encoding tree represents a mapping from symbols to binary codes, the dictionary is implicitly preserved as the tree evolves. Each node in the tree corresponds to a symbol, and the path from the root to a leaf node represents the binary code for that symbol. Therefore, at any given point, the encoding tree encapsulates the current dictionary mapping.

6. **Chunk-Based Encoding**: - While adaptive Huffman encoding does not explicitly preserve the dictionary at every chunk, it continuously updates the encoding tree based on the symbols encountered in the input stream. Therefore, the encoding scheme adapts to the characteristics of the input data dynamically, ensuring efficient compression.