ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN - CƠ - TIN HỌC

Huffman Trees and Codes

Tiểu luận cuối kỳ
Ngành: Máy tính và Khoa học thông tin
(Chương trình đào tạo chuẩn)

Sinh viên thực hiện:

La Thị Anh Thư	**20001980**
Phạm Bá Thắng	**20001976**
Vũ Thị Ngọc Quyến	**20001969**
Trịnh Thị Ngọc Mai	**20001948**

Hà Nội - 2023

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA TOÁN - CƠ - TIN HỌC

Huffman Trees and Codes

Tiểu luận cuối kỳ
Ngành: Máy tính và Khoa học thông tin
(Chương trình đào tạo chuẩn)

Người hướng dẫn: Cô Nguyễn Thị Hồng Minh

Sinh viên thực hiện:

La Thị Anh Thư	**20001980**
Phạm Bá Thắng	**20001976**
Vũ Thị Ngọc Quyến	**20001969**
Trịnh Thị Ngọc Mai	**20001948**

Hà Nội - 2023

Lời cảm ơn

Đầu tiên, chúng em xin chân thành cảm ơn cô Nguyễn Thị Hồng Minh - người giảng viên "vừa có tâm, vừa có tầm" đã cho sinh viên chúng em có cơ hội được tìm hiểu chuyên sâu hơn các chủ đề về thuật toán nói chung và về thuật toán Huffman nói riêng thông qua môn học Thiết kế và Đánh giá thuật toán.

Cảm ơn cô Nguyễn Thị Hồng Minh, thầy Trần Bá Tuấn, thầy Đặng Trung Du đã dạy dỗ, truyền đạt những kiến thức quý báu cho chúng em trong suốt thời gian học tập các buổi lý thuyết cũng như thực hành để chúng em có đủ kĩ năng và hiểu biết để làm nên bài tiểu luận này.

Tuy nhiên, vì thời gian hữu hạn và nhiều hạn chế nhất định, bài báo cáo khó có thể tránh khỏi những thiếu sót. Chúng em rất mong được thầy cô xem xét và góp ý để bài báo cáo tiểu luận cuối kỳ được hoàn thiện hơn.

Kính chúc thầy cô sức khỏe, hạnh phúc, thành công trên con đường sự nghiệp giảng dạy.

Mục lục

Danh sách hình vẽ

Danh sách bảng

Danh mục các ký hiệu và chữ viết tắt

STT	Chữ viết tắt	Diễn giải
1	ASCII	American Standard Code for Information Interchange Chuẩn mã trao đổi thông tin Hoa Kỳ
2	RFC	Request for Comments - Request for Comments Các bản ghi nhớ về các công nghệ, đổi mới của Internet
3	IETF	Internet Engineering Task Force Lực lượng Chuyên trách về Kỹ thuật Liên mạng, chuyên phát triển và thúc đẩy các chuẩn của Internet

Danh mục các cụm từ tương đương tiếng Anh và tiếng Việt

STT	Tiếng Anh	Tiếng Việt
1	Lossless Data Compression	Nén dữ liệu mà không mất thông tin
2	Greedy Method	Giải thuật tham lam
3	Encode	Mã hóa
4	Fixed-length	Độ dài cố định
5	Prefix code	Mã tiền tố
6	Data compression ratio	Tỉ số nén
7	Word size	Kích thước từ
8	Window size	Kích thước cửa sổ
9	Fixed Huffman Code	Bảng mã Huffman cố định
10	Dynamic Huffman Code	Bảng mã Huffman linh động

Giới thiệu

Nén dữ liệu, nhất là trong thời đại bùng nổ thông tin như hiện nay, trở thành một nhu cầu cần thiết thường nhật. Việc nén dữ liệu sẽ giúp chúng ta giảm được nguồn tài nguyên cũng như dung lượng lưu trữ, băng thông đường truyền.

Bên cạnh đó, các tệp tin văn bản thường hay có các kí tự xuất hiện nhiều lần, hình ảnh có thể có các vùng màu sắc không thay đổi trong nhiều pixel, các tập tin dạng âm thanh đã được số hóa cũng có thể có nhiều đoạn lặp lại nhiều lần... Những điều ấy dẫn đến việc dư thừa về dữ liệu và ta hoàn toàn có thể nén chúng lại mà không làm mất các thông tin.

Có rất nhiều phương pháp để nén dữ liệu, nhưng trong khuôn khổ, bài báo cáo sẽ chỉ đề cập đến thuật toán Huffman - một thuật toán nén cơ bản, dựa trên giải thuật tham lam trên cây nhị phân Huffman, bên cạnh đó là một số biến thể của nó.

I. Dẫn nhập

1 Ra đời

Thuật toán Huffman được đề xuất bởi David A. Huffman công bố năm 1952 trong bài báo "A Method for the Construction of Minimum - Redundancy Codes".

2 Ý tưởng

Samuel Morse vào giữa thế kỷ XIX đã phát minh ra mã điện báo. Trong đó, các chữ cái phổ biến như e (.) và a (.−) được gán các chuỗi dấu chấm và gạch ngang ngắn trong khi các chữ cái hiếm gặp như q (− − .−) và z (− − ..) có các biểu diễn dài hơn.

Huffman cũng gần giống vậy. Giả sử chúng ta phải mã hóa một đoạn văn bản gồm nhiều kí tự và mỗi kí tự được mã hóa bởi một dãy các bit gọi là codeword. Ta có thể dùng mã hóa với độ dài cố định như ASCII, mỗi ký tự được gán cho một xâu bit có cùng độ dài. Tuy nhiên, để kết quả xâu sau mã hóa ngắn hơn, ta sẽ mã hóa dựa trên tần suất xuất hiện các ký tự cần mã hóa, xuất hiện càng nhiều lần, codeword càng ngắn và ngược lại.

3 Mã tiền tố. Biểu diễn mã tiền tố trên cây nhị phân

Với ý tưởng trên, thay vì mã hóa với độ dài cố định (như ASCII là 8 bit), ta sẽ mã hóa với độ dài thay đổi. Khi đó, lại có vấn đề mới: Vậy khi giải mã làm thế nào phân biệt được xâu bít nào là mã hóa của ký hiệu nào? Một trong các giải pháp là dùng các dấu phẩy (",") hoặc một ký hiệu quy ước nào đó để tách từ mã của các ký tự đứng cạnh nhau. Nhưng như thế số các dấu phẩy sẽ chiếm một không gian đáng kể trong bảng mã. Vì vậy, ta cần đến khái niệm "mã tiền tố".

Mã tiền tố là bộ các từ mã của một tập hợp các ký hiệu sao cho từ mã của mỗi ký hiệu không là tiền tố (phần đầu) của từ mã một ký hiệu khác trong bộ mã ấy.

Ví dụ: : Giả sử mã hóa từ "ARRAY", tập các ký hiệu cần mã hóa gồm ba chữ cái "A","R","Y".

- Nếu mã hóa bằng các từ mã có độ dài bằng nhau ta dùng ít nhất 2 bit cho một chữ cái chẳng hạn "A"=00, "R"=01, "Y"=10. Khi đó mã hóa của cả từ là 0001010010. Để giải mã ta đọc hai bit một và đối chiếu với bảng mã.
- Nếu mã hóa "A"=0, "R"=01, "Y"=11 thì bộ từ mã này không là mã tiền tố vì từ mã của "A" là tiền tố của từ mã của "R". Để mã hóa cả từ ARRAY phải đặt dấu ngăn cách vào giữa các từ mã 0,01,01,0,11.
- Nếu mã hóa "A"=0, "R"=10, "Y"=11 thì bộ mã này là mã tiền tố. Với bộ mã tiền tố này khi mã hóa xâu "ARRAY" ta có 01010011.

Để có thể thực thi mã hóa Huffman trên máy tính, ta cần chọn một cấu trúc dữ liệu phù hợp để biểu diễn mã tiền tố, ta nghĩ đến cấu trúc cây. Ví dụ: Nếu có một cây nhị phân n lá ta có thể tạo một bộ mã tiền tố cho n ký hiệu bằng cách đặt mỗi ký hiệu vào một lá. Từ mã của mỗi ký hiệu được tạo ra khi đi từ gốc tới lá chứa ký hiệu đó, nếu đi sang nhánh trái thì ta thêm số 0, đi sang nhánh phải thì thêm số 1.

Ta sẽ đi cụ thể hơn về thuật toán Huffman (thiết kế và đánh giá) ở phần sau.

II. Thuật toán Huffman

1 Phát biểu bài toán

Cho tập A các ký hiệu và trọng số (tần suất xuất hiện) của chúng. Tìm một bộ mã tiền tố với tổng độ dài mã hóa là nhỏ nhất.

2 Giải thuật trong thuật toán Huffman

Thuật toán Huffman là một ví dụ của giải thuật tham lam. Nó được gọi là tham lam, vì hai nút nhỏ nhất được lựa chọn tại mỗi bước, và kết quả quyết định này là cục bộ trong một cây mã hóa tối ưu trên toàn cục.

3 Các bước thực hiện thuật toán Huffman

3.1 Mã hóa

- Bước 1: Tính tần số xuất hiện của mỗi ký tự trong chuỗi cần mã hóa và tạo một danh sách các nút lá, mỗi nút lá chứa một ký tự và tần số của nó.
- Bước 2: Sắp xếp danh sách các nút lá theo thứ tự tăng dần của tần số và lấy hai nút có tần số nhỏ nhất để tạo một nút cha mới, có giá trị bằng tổng tần số của hai nút con. Gán nhãn nút con trái là 0 và nút con phải là 1. Thêm nút cha vào danh sách và loại bỏ hai nút con khỏi danh sách.
- Bước 3: Lặp lại bước 2 cho đến khi chỉ còn một nút duy nhất trong danh sách, đó là nút gốc của cây Huffman. Cây Huffman được xây dựng từ dưới lên bằng cách kết hợp các nút có tần số thấp nhất thành các nhánh mới.
- Bước 4: Duyệt cây Huffman từ gốc đến lá để tạo mã cho mỗi ký tự. Mã của một ký tự là chuỗi các bit biểu diễn đường đi từ gốc đến lá chứa ký tự đó. Tập hợp này được gọi là bảng Huffman. Ví dụ, nếu đường đi từ gốc đến lá chứa ký tự a là 0-1-0-1, thì mã của a là 0101.

- Bước 5: Mã hóa chuỗi ban đầu bằng cách thay thế mỗi ký tự bằng mã tương ứng của nó. Ví dụ, nếu chuỗi ban đầu là "abac" và mã của a là 0, b là 10, c là 11, thì chuỗi mã hóa là "010011".
- Bước 6: Sinh bảng Huffman:
  - Bước 1: Duyệt cây và biểu diễn theo dạng pre-order. Nút trái biểu diễn bằng 0, nút phải biểu diễn bằng 1.
  - Bước 2: Đếm số nút trên cây trừ nút gốc.
  - Bước 3: Tại các nút lá ta lưu lại kí tự theo pre-order.
    Giả sử có cây Root[Left: [Node a] Right [Node Left[Node b] Right[Node c]] thì số nút sẽ là 5. Thứ tự duyệt cây post-prefix không tính nút gốc là 0101, danh sách nút lá là abc. Thường cả đoạn này sẽ được chuyển sang nhị phân.
- Bước 7: Trả về bảng Huffman và chuỗi đã được mã hóa.

3.2 Giải mã

- Bước 1: Tạo cây Huffman từ bảng Huffman.
  - Bước 1: Lấy số nút trên cây.
  - Bước 2: Lấy thứ tự pre-order trên cây. Chiều dài mảng này luôn nhỏ hơn số nút là 1.
  - Bước 3: Sinh cây, xét từ vị trí nút gốc:
    - Gặp số 0, chúng ta sẽ thêm một cây con vào bên trái và xét tiếp tại nút này.
    - Gặp số 1, ta quay lui cho tới khi nào gặp nút cha chưa có cây con phải thì thêm cây con phải và xét nút vừa thêm.
  - Bước 4: Lấy tất cả nút lá theo thứ tự pre-order và chèn các kí tự.
- Bước 2: Giải mã chuỗi mã hóa bằng cách duyệt cây Huffman từ gốc đến lá theo từng bit trong chuỗi. Nếu bit là 0, chuyển sang nút con trái, nếu bit là 1, chuyển sang nút con phải. Khi gặp một nút lá, lấy ký tự trong nút lá và thêm vào chuỗi giải mã. Quay lại nút gốc và tiếp tục duyệt cho đến khi hết chuỗi mã hóa.
  Ví dụ, để giải mã chuỗi "010011", ta bắt đầu từ gốc và đi theo các bit như sau: 0 -> trái -> a, 1 - phải -> 0 - trái -> b, 0 - trái -> a, 1 - phải -> 1 - phải -> c. Chuỗi giải mã là "abac".

4 Tính đúng đắn của giải thuật tham lam Huffman

Để chứng minh thuật toán tham lam Huffman là đúng, ta chỉ ra rằng bài toán xác định mã tiền tố tối ưu thể hiện lựa chọn tham lam và những tính chất cấu trúc con tối ưu cục bộ.

a) Bổ đề 1

Cho bảng ký tự C mà mỗi ký tự $c \in C$ có tần số là $f[c]$, chiều sâu là $dT(c)$. Cho x và y là hai ký tự trong C có tần số thấp nhất. Tồn tại mã tiền tố tối ưu đối với C mà trong đó từ mã của x và y có cùng chiều dài và chỉ khác duy nhất ở bit cuối cùng.

Chứng minh:

Cho hai ký tự a và b là các lá anh em ruột có chiều sâu cực đại trong T. Không mất tính tổng quát, ta mặc nhận rằng $f[a] \leq f[b]$ và $f[x] \leq f[y]$. Vì $f[x]$, $f[y]$ lần lượt là hai tần số của lá thấp nhất và $f[a]$, $f[b]$ là các tần số tùy ý theo thứ tự nên ta có: $f[x] \leq f[a]$ và $f[y] \leq f[b]$.

Ta tráo đổi các vị trí trong T của a và x để tạo ra cây mới T'. Sau đó, ta tráo đổi vị trí của b và y trong cây T' để tạo ra cây mới T''. Trong cây tối ưu T, các lá a và b là các lá sâu nhất và là anh em ruốt với nhau. Các lá x và y là hai lá mà giải thuật Huffman kết hợp với nhau đầu tiên, chúng xuất hiện ở các vị trí tùy ý trên cây T. Trao đổi các lá a và x để thu được cây T'. Sau đó, trao đổi các lá b và y để thu được cây T''. Bởi vì mỗi lần trao đổi không làm tăng mức phí, dẫn đến cây T'' cũng là cây tối ưu.

Ta có: $f[a] - f[x]$ và $dT(a) - dT(x)$ là không âm. $f[a] - f[x]$ là không âm vì x là lá có tần số cực tiểu và $dT(a) - dT(x)$ là không âm vì a là lá có chiều sâu cực đại trên T. Việc trao đổi a và b với x và y không làm tăng mức hao phí và $B(T') - B(T'')$ là không âm. Dẫn đến $B(T'') \leq B(T)$ và vì T là tối ưu nên $B(T) \leq B(T'')$, suy ra $B(T) = B(T'')$. Vì vậy, T'' là cây tối ưu trong đó x và y xuất hiện như những lá anh em ruột có chiều sâu cực đại.

b) Bổ đề 2:

Cho bảng ký tự C mà mỗi ký tự $c \in C$ có tần số là $f[c]$. Cho x và y là hai ký tự trong C có tần số thấp nhất. Nếu T' là mã tiền tố tối ưu của $C' = C \setminus \{x,y\} \cup \{z\}$ với $f[z] = f[x] + f[y]$ thì T là mã tiền tố tối ưu của C.

Chứng minh:

Ta chứng minh bổ đề bằng phản chứng.

Giả sử cây T biểu diễn mã tiền tố không tối ưu cho C. Tồn tại một cây T'' sao cho $B(T'') \leq B(T)$. Không mất tính tổng quát (theo Bổ đề 1), T'' có x và y là anh em ruột. Cho cây T''' là cây T'' với nút cha của x và y được thay thế bằng nút z có tần số $f[z] = f[x] + f[y]$. Điều này dẫn đến mâu thuẫn với giả thiết T' biểu diễn mã tiền tố tối ưu cho C'. Vì vậy, T phải biểu diễn mã tiền tố tối ưu cho C.

c) Chứng minh tính đúng đắn của giải thuật tham lam Huffman:

- Trường hợp cơ sở: $|C| = 2$.
- Giả thiết quy nạp: Giả sử thuật toán Huffman đúng với mọi $|C| < n$.
- Bước quy nạp: Xét trường hợp $|C| = n$. Theo thuật toán Huffman, ta quy bài toán về bài toán dựng mã tiền tố tối ưu T' cho C' trong đó $|C'| = n - 1$. Theo giả thiết quy nạp, T' là mã tiền tố tối ưu cho C'. Vậy T là mã tiền tố tối ưu cho C (theo Bổ đề 2).

5 Cài đặt thuật toán Huffman

5.1 Cài đặt Node

- Node là một nút trong cây Huffman gồm 4 thuộc tính:
  - char: ký tự của nút, tổng quát hơn là từ (word) lưu trong nút, mặc định là None. Nếu nút là nút lá thì char sẽ là ký tự của nút, nếu nút là nút cha thì char sẽ là None (kích thước của nó được gọi là word size).
  - freq: tần số xuất hiện của ký tự, mặc định là None. Nếu nút là nút lá thì freq sẽ là tần số của ký tự, nếu nút là nút cha thì freq sẽ là tổng tần số của các nút con.
  - left: nút con trái.
  - right: nút con phải.
- Các hàm trong Node:
  - `__init__`: khởi tạo một nút mới, đã được cài đặt để thỏa mãn các yêu cầu trên.
  - Bộ hàm so sánh `__lt__`, `__eq__`, `__gt__`, `__le__`, `__ge__`: so sánh tần số của hai nút.

- `__str__`, `__repr__`: trả về chuỗi mô tả nút.
- `to_bit`: trả về ba đối tượng.
  1. Số nút.
  2. Cây Huffman nhị phân, biểu diễn dưới dạng Preorder bit.
  3. Các lá Bảng Huffman.
- `leaf`: trả về danh sách các lá của cây.

```
class Node:
    def __init__(self, char: str = None, freq: int = None, left=None,
        ↪ right=None, father=None):
        if left is not None or right is not None:
            self.char = None
            self.freq = (left.freq if left is not None else 0) + (right
                ↪ .freq if right is not None else 0)
            self.left = left
            self.right = right
            self.father = father
            if left is not None:
                left.father = self
            if right is not None:
                right.father = self
        else:
            self.char = char
            self.freq = freq
            self.left = None
            self.right = None
            self.father = father

    def __lt__(self, other):
        return self.freq < other.freq

    def __eq__(self, other):
        return self.freq == other.freq

    def __gt__(self, other):
        return self.freq > other.freq

    def __le__(self, other):
        return self.freq <= other.freq
```

```
31
32      def __ge__(self, other):
33          return self.freq >= other.freq
34
35      def __str__(self):
36          return f'Node[({self.char}), left: [{self.left}], right: [{
                ↪ self.right}]]'
37      def __repr__(self):
38          return str(self)
39
40      def to_table(self):
41          node = 0
42          tree = ''
43          char = []
44          if self.char is not None:
45              return 1, tree, [self.char]
46          else:
47              node_l, tree_l, char_l = self.left.to_table() if self.left
                    ↪ is not None else (0, '', [])
48              node_r, tree_r, char_r = self.right.to_table() if self.
                    ↪ right is not None else (0, '', [])
49              char = char_l + char_r
50              node = node_l + node_r + 1
51              if self.left is not None:
52                  tree += '0' + tree_l
53              if self.right is not None:
54                  tree += '1' + tree_r
55              return [node, tree, char]
56      def leafs(self):
57          if self.left is None and self.right is None:
58              return [self]
59          else:
60              return (self.left.leafs() if self.left is not None else [])
                    ↪  + (self.right.leafs() if self.right is not None
                    ↪ else [])
```

5.2 Encode

a) freq: đếm tần số xuất hiện của các ký tự trong chuỗi

```
def calc_freq(data):
    list_char = {}
    for char in data:
        if char in list_char:
            list_char[char] += 1
        else:
            list_char[char] = 1
    return list_char
```

b) build_tree: chuyển dữ liệu thành cây Huffman

```
def build_tree(data):
    list_char = calc_freq(data)
    heap = []
    for char, freq in list_char.items():
        heap.append(Node(char, freq))
    while len(heap) > 1:
        heap.sort()
        left = heap.pop(0)
        right = heap.pop(0)
        heap.append(Node(left=left, right=right))
    if heap[0].char is not None:
        return Node(left=heap[0])
    return heap[0]
```

c) tree_to_dict: chuyển cây Huffman thành bảng mã

```
def tree_to_dict(root):
    huffman_dict = {}
    def traverse(node, code):
        if node is None:
            return

        if node.char is not None:
```

```
            huffman_dict[node.char] = code
            return

        traverse(node.left, code + '0')
        traverse(node.right, code + '1')

    traverse(root, '')
    return huffman_dict
```

d) encode: chuyển dữ liệu thành chuỗi mã hóa và bảng mã

- Bước 1: Dựng cây Huffman từ dữ liệu.
- Bước 2: Chuyển cây Huffman thành bảng mã.
- Bước 3: Duyệt dữ liệu, mã hóa từng ký tự.

```
def encode(data):
    root = build_tree(data)
    huffman_dict = tree_to_dict(root)
    res = ''
    for char in data:
        res += huffman_dict[char]
    res = root.to_table() + [res]
    return res
```

5.3 Decode

a) table_to_tree: chuyển bảng mã thành cây Huffman

```
def table_to_tree(code):
    node = code[0] - 1# Lấy số nút trừ root
    preorder=code[1] # Lấy preorder củ a cây
    # Xây cây
    root = Node()
    current = root
    for i in range(len(preorder)):
        if preorder[i] == '0':
            current.left = Node()
            current.left.father = current
```

```
            current = current.left
        elif preorder[i] == '1':
            current = current.father
            while current.right is not None:
                current = current.father
            current.right = Node()
            current.right.father = current
            current = current.right
    # Chèn kí tự cho nút lá
    leafs = root.leafs()
    for i in range(len(leafs)):
        leafs[i].char = code[2][i]
    return root, code[-1]
```

b) decode: giải mã chuỗi mã hóa thành dữ liệu

- Bước 1: Chuyển bảng mã thành cây Huffman.
- Bước 2: Duyệt chuỗi mã hóa, để đi đến vị trí đích của ký tự trong cây Huffman.
- Bước 3: Lấy ký tự tại vị trí đích.
- Bước 4: Về nút gốc và lặp lại cho đến khi duyệt hết chuỗi mã hóa.

```
def decode(code, huffman_tree=None):
    if huffman_tree is None:
        huffman_tree, code = table_to_tree(code)
    current = huffman_tree
    data = ''
    for c in code:
        if c == '0':
            current = current.left
        elif c == '1':
            current = current.right
        if current.char is not None:
            data += current.char
            current = huffman_tree
    return data
```

6 Đánh giá thuật toán

a) Độ phức tạp về thời gian

- Tính tần số xuất hiện của từng ký tự (list_char): O(n)
- Tạo ra danh sách các nút của cây: O(n)
- Tạo cây:
  - Sắp xếp các nút con (thuật toán TimSort): O(nlogn)
  - Tạo nút con trái/phải: O(1)
  - Thêm một nút con mới: O(1)

=> Độ phức tạp về thời gian của thuật toán: O(nlogn)

b) Độ phức tạp về không gian

- Tính tần số xuất hiện của từng ký tự (list_char): O(n)
- Tạo ra danh sách các nút của cây: O(n)
- Tạo cây:
  - Sắp xếp các nút con (thuật toán TimSort): O(n)
  - Tạo nút con trái/phải: O(1)
  - Thêm một nút con mới: O(1)

=> Độ phức tạp về không gian của thuật toán: O(n)

c) Độ hiệu quả của thuật toán

Đối với các thuật toán nén, hiệu quả của thuật toán được tính bằng tỉ số nén. Tỉ số nén được tính bằng kích thước dữ liệu trước khi nén chia cho kích thước dữ liệu sau khi nén.

Giả sử một chuỗi n kí tự, mỗi kí tự được biểu diễn bằng đoạn mã có độ dài cố định f bit. Như vậy độ dài trước khi mã hóa là $f \times n$ bit.

Sau khi dựng xong cây Huffman, thu được một cây Huffman có o node, số này sẽ được biểu diễn bằng a bit, cùng m nút lá chứa các kí tự. Đoạn mã hóa sẽ được thể hiện bằng e bit. Như vậy, theo đoạn mã thể hiện cài đặt của Huffman độ dài sau khi mã hóa là $a + (o-1) + f \times m + e$ bit.

Để thuật toán hiệu quả thì $(n-m) \times f > a + (o-1) + e$.

Từ đó ta có có hai tình huống xấu:

- $f < \frac{a+(o-1)+e}{n-m}$, khi đó sẽ có những kí tự phải biểu diễn bằng số bit lớn hơn ban đầu. Nếu n đủ lớn chuỗi vẫn sẽ nén được.
- $n \approx m$, phình thêm $a+(o-1)+e$ bit do tần suất xuất hiện thấp, xấu nhất n = m.

Trường hợp 1:

Giả sử ta mã hóa chuỗi '1234567890' và các chuỗi được lặp lại nhiều lần của nó. Mỗi kí tự sẽ tính 5 bit, $a = 5$.

Số lần lặp	**Trước khi nén**	**Sau khi nén**
2	100	141
4	200	209
5	250	243

Bảng II.1.: Tổng hợp lại kết quả khi chạy với mỗi kí tự 5 bit

Ta thấy lặp lại chuỗi đến 5 lần, n mới đủ lớn để chuỗi được nén hiệu quả.

Trường hợp 2:

Ở đây sẽ tính một kí tự sẽ tính là 32 bit theo chuẩn UTF-8, số lượng node cũng sẽ tính là 32 bit.

Đoạn kí tự	**Trước khi nén**	**Sau khi nén**	**Hiệu suất**
abcdefghiklmnopqrstuvwxyz	800	998	0.8
aaaaaaaaaaaaaa	480	80	6
Trăm năm trong cõi người ta, Chữ tài chữ mệnh khéo là ghét nhau. Trải qua một cuộc bể dâu, Những điều trông thấy mà đau đớn lòng.	4192	2082	2.0

Bảng II.2.: Tổng hợp lại kết quả khi chạy với mỗi kí tự 32 bit

Nhận xét:

- Tỉ số nén cho chuỗi 1 là 0.8, độ dài bit từ 800 bit lên 998 bit, kết quả nén và giải nén không đổi (lossless) nặng hơn ban đầu.
- Đối với chuỗi 2, dữ liệu dễ nén tần suất xuất hiện cao, tỉ lệ nén lên đến 6.

- Đối với chuỗi 3 là một đoạn văn bản, tỉ số nén bao gồm cả từ điển là 2.0 từ hơn 4192 bit xuống 2082 bit.

Bên cạnh đó, thuật toán Huffman có một điểm đặc biệt mà các thuật toán khác không có là nó xử lý theo từng bit một, chỉ cần một cây có sẵn chúng ta không cần có hết các bit vẫn có thể giải được, không cần quan tâm bit tiếp theo. Việc này phù hợp cho việc truyền thông tin trực tiếp, không cần phải đợi đến khi có đủ dữ liệu để giải mã như HTTP, SMB, ...

Bên cạnh đó để tiết kiệm dung lượng cho file nén, người ta có thể dùng chung cây như thuật toán. Còn không có cây, không thể giải mã, một hình thức bảo mật. Nhưng Huffman không được sử dụng như một thuật toán mã hóa (encrypt).

III. Trực quan hóa

1 Hướng dẫn cài đặt và sử dụng Visualization

a) Cài đặt

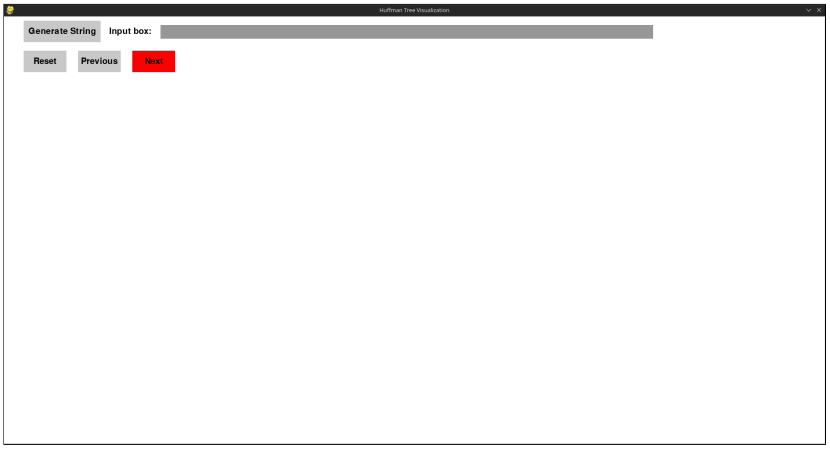
- Tải thư mục chứa source code.
- Visualization sử dụng thư viện pygame của python, vì thế trước khi chạy, ta cần cài thư viện pygame với lệnh: `pip install pygame`

b) Sử dụng

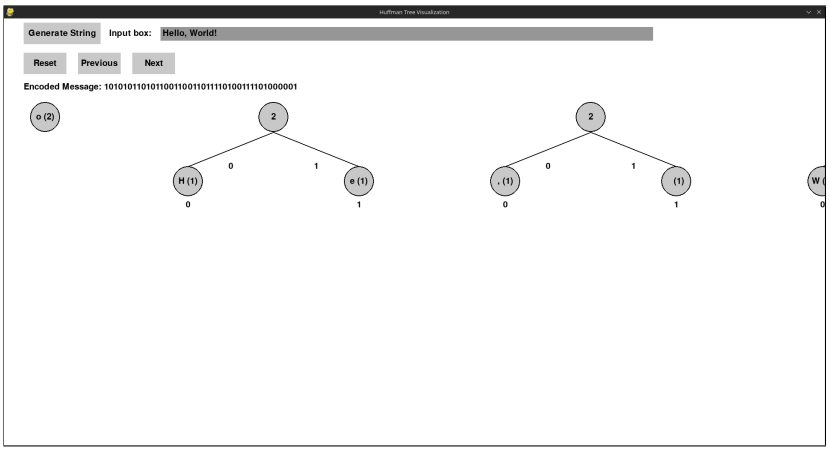
Giao diện gồm các phím và nút lệnh:

- Generate String: Chạy với chuỗi khởi tạo là "Hello, World!".
- Input Box: Tại đây chứa chuỗi cần mã hóa, có thể nhập chuỗi tùy ý sau đó nhấn Enter.
- Next: Bước tiếp theo của thuật toán.
- Previous: Bước trước đó của thuật toán.
- Decode: Sau khi mã hóa hết chuỗi input sẽ xuất hiện nút Decode để giải mã. Bấm vào đây sẽ hiện nút Next trên màn hình để chạy từng bước giải mã.
- Cuộn trang
  - Phím $\leftarrow$: dịch sang trái.
  - Phím $\rightarrow$: dịch sang phải.
  - Phím $\uparrow$: dịch xuống.
  - Phím $\downarrow$: dịch lên.
  - Phím Tab: Về đầu cây Huffman.
  - Phím Space: Về cuối cây Huffman.
  - Phím +: Tăng tốc độ cuộn.
  - Phím -: Giảm tốc độ cuộn.

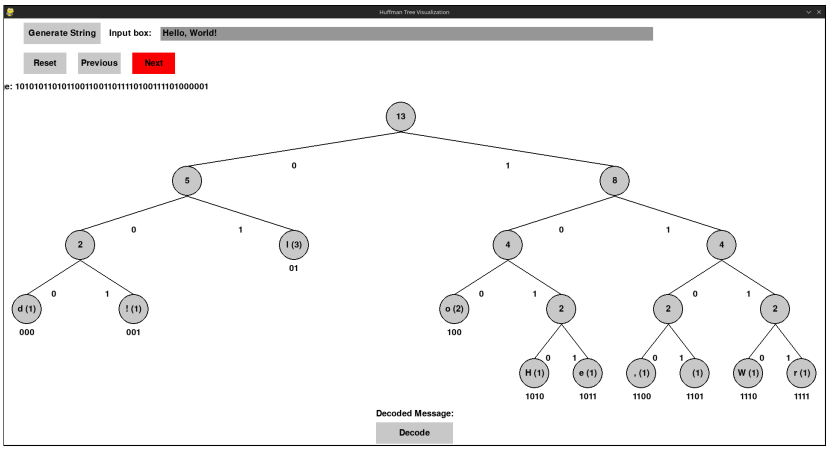
c) Hình ảnh minh họa



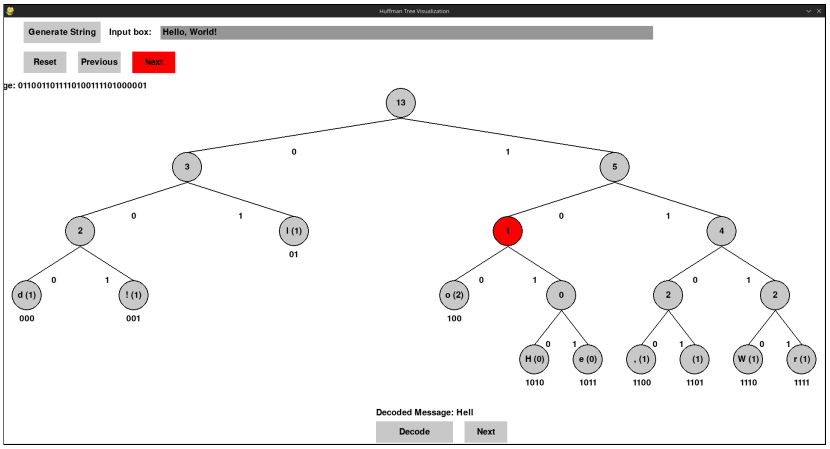
Hình III.1.: Giao diện mở đầu



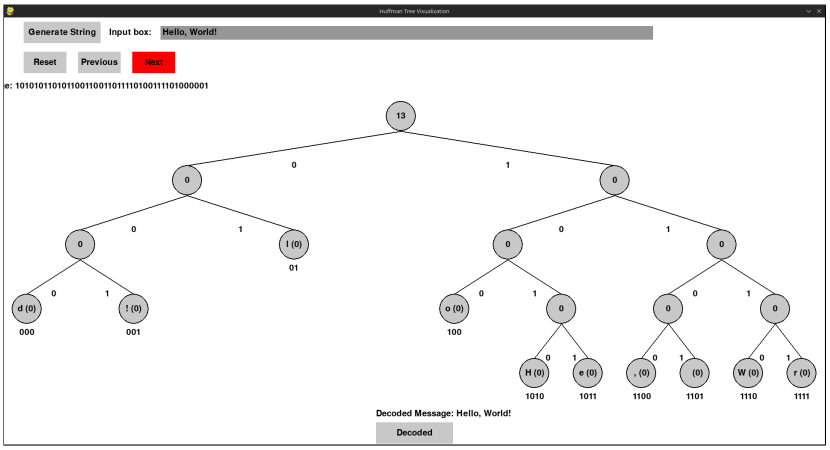
Hình III.2.: Minh họa quá trình dựng cây



Hình III.3.: Kết quả sau khi hoàn thành dựng cây



Hình III.4.: Minh họa quá trình giải mã



Hình III.5.: Kết quả sau khi hoàn thành giải mã

IV. Một số biến thể của Huffman Coding

1 N-ary Huffman Coding

a) Ý tưởng

N-ary Huffman Coding là Huffman code sử dụng cấu trúc dữ liệu là cây n-ary (cây dữ liệu cho phép mỗi nút có tối đa n nút con, khác với Huffman cơ bản dùng cây nhị phân chỉ cho phép mỗi nút có tối đa hai nút con).

Ưu điểm: Có thể giảm độ dài đường dẫn trung bình (weighted path length) của cây Huffman, từ đó giảm số lượng bit cần thiết để mã hóa một ký hiệu; tăng tốc độ decode do giảm số lượng nhánh cần dự đoán trong quá trình duyệt cây.

Nhược điểm: Khi tăng bậc của cây lên, số lượng bit cần thiết để biểu diễn một nút trong cây cũng tăng lên, do đó có thể làm tăng dung lượng lưu trữ.

b) Xây dựng cây n-ary Huffman và mã hóa:

Tương tự cách xây dựng cây nhị phân Huffman (trên thực tế, cây Huffman cơ bản là một trường hợp của n-ary Huffman tree với n=2), ta có các bước dựng cây như sau:

- Sắp xếp các ký hiệu theo xác suất giảm dần.
- Lấy n ký hiệu có xác suất thấp nhất và tạo một nút mới có chúng làm con. Xác suất của nút mới là tổng của xác suất của các con. Gán một mã khác nhau (0, 1,. . . , n - 1) cho mỗi nhánh.
- Thay thế n ký hiệu bằng nút mới trong danh sách đã sắp xếp và sắp xếp lại danh sách theo xác suất.
- Lặp lại các bước 2 và 3 cho đến khi chỉ còn một nút duy nhất, đó là gốc của cây.
- Duyệt cây từ gốc đến mỗi lá và nối các mã trên đường đi để lấy mã cho mỗi ký hiệu.

c) Giải mã

- Bắt đầu từ nút gốc của cây, đọc mỗi b bit của chuỗi mã hóa (b là số bit cần thiết để biểu diễn n, ví dụ b = 2 cho n = 4).
- Duyệt nhánh tương ứng với bit đó và di chuyển đến nút con.
- Nếu nút con là một nút lá, lấy ký tự trong nút lá và thêm vào chuỗi giải mã. Quay lại nút gốc và tiếp tục quá trình.
- Nếu nút con không phải là một nút lá, thì tiếp tục đọc bit tiếp theo và lặp lại các bước trên cho đến khi hết chuỗi mã hóa.

d) Cài đặt thuật toán

Ở đây sẽ triển khia với b = 2, mọi hàm đều gần như tương tự

- Ta sẽ tạo class `Node4` kế thừa từ `Node`, thay vì 2 biến để lưu cây con `left` và `right` thì ta dùng `node00`, `node01`, `node10`, `node11`

```
class Node4(Node):
    def __init__(self, char: str = None, freq: int = None, node00
        ↪ =None, node01=None, node10=None, node11=None, father=
        ↪ None):
        if node00 is not None or node01 is not None or node10 is
            ↪ not None or node11 is not None:
            self.char = None
            self.freq = (node00.freq if node00 is not None else 0)
                ↪  + (node01.freq if node01 is not None else 0) +
                ↪ (node10.freq if node10 is not None else 0) + (
                ↪ node11.freq if node11 is not None else 0)
            self.node00 = node00
            self.node01 = node01
            self.node10 = node10
            self.node11 = node11
            self.father = father
            if node00 is not None:
                node00.father = self
            if node01 is not None:
                node01.father = self
            if node10 is not None:
                node10.father = self
            if node11 is not None:
```

```
            node11.father = self
        else:
            self.char = char
            self.freq = freq
            self.node00 = None
            self.node01 = None
            self.node10 = None
            self.node11 = None
            self.father = father

    def __str__(self):
        return f'Node[({self.char}), node00: {self.node00}, node01
            ↪ : {self.node01}, node10: {self.node10}, node11: {
            ↪ self.node11}]'

    def to_table(self):
        node = 0
        tree = ''
        char = []
        if self.char is not None:
            return 1, tree, [self.char]
        else:
            node00, tree00, char00 = self.node00.to_table() if
                ↪ self.node00 is not None else (0, '', [])
            node01, tree01, char01 = self.node01.to_table() if
                ↪ self.node01 is not None else (0, '', [])
            node10, tree10, char10 = self.node10.to_table() if
                ↪ self.node10 is not None else (0, '', [])
            node11, tree11, char11 = self.node11.to_table() if
                ↪ self.node11 is not None else (0, '', [])
            char = char00 + char01 + char10 + char11
            node = 1 + node00 + node01 + node10 + node11
            if node00 > 0:
                tree += '00' + tree00
            if node01 > 0:
                tree += '01' + tree01
            if node10 > 0:
                tree += '10' + tree10
            if node11 > 0:
                tree += '11' + tree11
```

```
52          return [node, tree, char]
53      def leafs(self):
54          if self.node00 is None and self.node01 is None and self.
                ↪ node10 is None and self.node11 is None:
55              return [self]
56          else:
57              return (self.node00.leafs() if self.node00 is not None
                    ↪  else []) + (self.node01.leafs() if self.node01
                    ↪ is not None else []) + (self.node10.leafs() if
                    ↪ self.node10 is not None else []) + (self.node11.
                    ↪ leafs() if self.node11 is not None else [])
```

- Nhóm hàm mã hóa
  - Hàm dựng cây

```
1   def build_tree4(data):
2       list_char = calc_freq(data)
3       heap = []
4       for char, freq in list_char.items():
5           heap.append(Node4(char, freq))
6       while len(heap) >= 4:
7           heap.sort()
8           node00 = heap.pop(0)
9           node01 = heap.pop(0)
10          node10 = heap.pop(0)
11          node11 = heap.pop(0)
12          heap.append(Node4(node00=node00, node01=node01, node10
                ↪ =node10, node11=node11))
13      if len(heap) >= 3:
14          heap.sort()
15          node00 = heap.pop(0)
16          node01 = heap.pop(0)
17          node10 = heap.pop(0)
18          heap.append(Node4(node00=node00, node01=node01, node10
                ↪ =node10))
19      if len(heap) >= 2:
20          heap.sort()
21          node00 = heap.pop(0)
22          node01 = heap.pop(0)
23          heap.append(Node4(node00=node00, node01=node01))
24      if heap[0].char is not None:
```

```
        node00 = heap.pop(0)
        heap.append(Node4(node00=node00))
    return heap[0]
```

- Hàm chuyển cây sang từ điển

```
def tree_to_dict4(root):
    huffman_dict = {}
    def traverse(node: Node4, code):
        if node is None:
            return

        if node.char is not None:
            huffman_dict[node.char] = code
            return

        traverse(node.node00, code + '00')
        traverse(node.node01, code + '01')
        traverse(node.node10, code + '10')
        traverse(node.node11, code + '11')

    traverse(root, '')
    return huffman_dict
```

- Hàm mã hóa

```
def encode4(data):
    root = build_tree4(data)
    huffman_dict = tree_to_dict4(root)
    res = ''
    for char in data:
        res += huffman_dict[char]
    res = root.to_table() + [res]
    return res
```

- Nhóm hàm giải mã
  - Chuyển mã thành cây

```
def table_to_tree4(code):
    node = code[0] - 1
    preorder=code[1]
    root = Node4()
```

```
    current = root
    for i in range(len(preorder)//2):
        if preorder[i*2:i*2+2] == '00':
            current.node00 = Node4()
            current.node00.father = current
            current = current.node00
        elif preorder[i*2:i*2+2] == '01':
            current = current.father
            while current.node01 is not None:
                current = current.father
            current.node01 = Node4()
            current.node01.father = current
            current = current.node01
        elif preorder[i*2:i*2+2] == '10':
            current = current.father
            while current.node10 is not None:
                current = current.father
            current.node10 = Node4()
            current.node10.father = current
            current = current.node10
        elif preorder[i*2:i*2+2] == '11':
            current = current.father
            while current.node11 is not None:
                current = current.father
            current.node11 = Node4()
            current.node11.father = current
            current = current.node11
    leafs = root.leafs()
    for i in range(len(leafs)):
        leafs[i].char = code[2][i]
    return root, code[-1]
```

– Giải mã

```
def decode4(code, huffman_tree=None):
    if huffman_tree is None:
        huffman_tree, code = table_to_tree4(code)
    current = huffman_tree
    data = ''
    for i in range(len(code)//2):
        if code[i*2:i*2+2] == '00':
```

```
            current = current.node00
        elif code[i*2:i*2+2] == '01':
            current = current.node01
        elif code[i*2:i*2+2] == '10':
            current = current.node10
        elif code[i*2:i*2+2] == '11':
            current = current.node11
        if current.char is not None:
            data += current.char
            current = huffman_tree
    return data
```

e) So sánh giữa 2-ary(Huffman bình thường) và 4 ary

Đoạn kí tự	Trước khi nén	Hiệu suất nén 2-ary	Hiệu suất nén 4-ary
abcdefghiklmnopqrstuvwxyz	800	0.8	0.78
aaaaaaaaaaaaaa	480	6	5
Trăm năm trong cõi người ta, Chữ tài chữ mệnh khéo là ghét nhau. Trải qua một cuộc bể dâu, Những điều trông thấy mà đau đớn lòng.	4192	2.0	1.91

Bảng IV.1.: Tổng hợp lại kết quả khi chạy, mỗi kí tự 32 bit

Với bộ dữ liệu này, 4-ary có hiệu suất kém hơn Huffman bình thường.

2 Length-limited Huffman Coding

a) Ý tưởng:

Length-limited Huffman Coding (Mã hóa Huffman có giới hạn độ dài) là một biến thể của mã hóa Huffman, có giới hạn độ dài của mã hóa của mỗi từ, là thuật toán có độ phức tạp thời gian O(nL) và độ phức tạp không gian O(n), xây dựng mã Huffman

tối ưu cho các kí tự có độ dài lưu trữ là n, mà ở đó, độ dài kí tự sau khi mã hóa không vượt quá L.

Bằng cách giới hạn độ dài mã hóa của mỗi từ, ta có thể tránh các mã hóa quá dài khi có một số ký tự có xác suất xuất hiện rất thấp. Khi có sự khác biệt về chi phí truy cập giữa các cấp bộ nhớ, ví dụ như cache và RAM, giới hạn độ dài mã hóa của mỗi từ giúp giải mã nhanh và tiết kiệm hơn, dễ dàng phù hợp với các yêu cầu cụ thể cố định của các thiết bị phần cứng. Ngoài ra, việc sử dụng mã hóa của mỗi từ có độ dài giới hạn sẽ giúp giảm thiểu rủi ro khi có sai lệch về xác suất khi xây dựng cây/bảng Huffman trong trường hợp không biết chính xác xác suất của tập nguồn cần mã hóa.

b) Lịch sử phương pháp

Từ các lý thuyết về: xây dựng cây nhị phân có độ dài đường đi trọng số nhỏ nhất và độ dài đường đi tối đa bị hạn chế (của Hu & Tan năm 1972); "Constructing codes with bounded codeword lengths" (của D. C. Van Voorhis năm 1974), năm 1990 Larmore và Hirschberg đề xuất mô hình "package-merge", thực thi ý tưởng trên một cách hiệu quả. "Package-merge" xây dựng các package (gói) ký hiệu, theo cách tương tự như thuật toán của Huffman, nhưng không mục nào có thể tham gia nhiều hơn L bước kết hợp. Hạn chế đó được thực thi bằng cách tạo tất cả các cây con có chi phí thấp nhất có độ sâu tối đa khác nhau và giữ lại thông tin cho từng độ sâu nút có thể có trong một cấu trúc riêng biệt.

c) Package-merge

- Phương pháp package-merge có thể đưa về bài toán coin collector nhị phân.
- Cách thuật toán hoạt động như sau:
  - Giả sử bộ chữ cái cần mã hóa có n ký tự với tần suất $p1, p2, \ldots, pn$ và độ dài tối đa của mỗi từ mã là L.
  - Tạo ra L đồng xu cho mỗi ký tự, với mệnh giá $2^{-1}, 2^{-2}, \ldots, 2^{-L}$ và giá trị tiền tệ bằng với tần suất của ký tự đó.
  - Sắp xếp các đồng xu theo thứ tự tăng dần của giá trị tiền tệ và mệnh giá.
  - Package (Gói) các đồng xu có mệnh giá nhỏ nhất thành các cặp, bắt đầu từ cặp có tổng giá trị tiền tệ nhỏ nhất. Nếu có một đồng xu còn thừa, nó sẽ là đồng xu có giá trị tiền tệ cao nhất của mệnh giá đó, và nó sẽ bị bỏ qua. Mỗi cặp được coi như một gói có mệnh giá bằng tổng mệnh giá của hai đồng xu trong cặp và giá trị tiền tệ bằng tổng giá trị tiền tệ của hai đồng xu trong cặp.

– Merge (Hợp nhất) các gói vừa tạo với danh sách các đồng xu có mệnh giá kế tiếp, lại theo thứ tự tăng dần của giá trị tiền tệ và mệnh giá. Các phần tử trong danh sách này cũng được gói thành các cặp và hợp nhất với danh sách kế tiếp, và cứ tiếp tục như vậy.
– Cuối cùng, sẽ có một danh sách các phần tử, mỗi phần tử là một đồng xu có mệnh giá 1 hoặc một gói gồm hai hoặc nhiều đồng xu có tổng mệnh giá bằng 1. Chúng cũng được sắp xếp theo thứ tự tăng dần của giá trị tiền tệ.
– Chọn ra $n - 1$ phần tử có giá trị tiền tệ nhỏ nhất trong danh sách cuối cùng. Gọi *hi* là số lượng phần tử có giá trị tiền tệ *pi* được chọn. Mã Huffman có giới hạn độ dài tối ưu sẽ mã hóa ký tự *i* với một chuỗi bit có độ dài *hi*

3 Kết hợp của các thuật toán nhóm LZ và Huffman

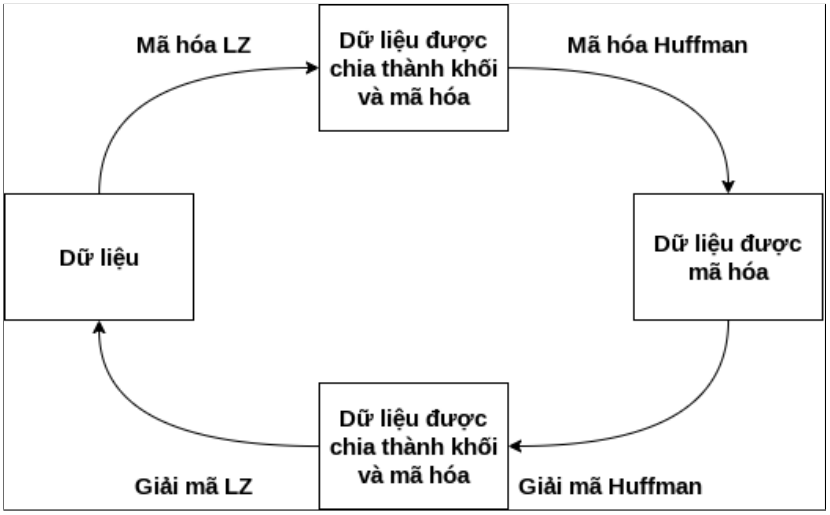
a) Ý tưởng:

Nếu như chuỗi cần nén có ít chữ cái như một chuỗi bit chẳng hạn, việc nén bằng Huffman sẽ làm tăng dung lượng cho việc lưu cây và các node. Hay trong một chuỗi có nhiều từ giống nhau, chỉ cần một đoạn bit, chúng ta có thể biểu diễn được cả một từ nhiều kí tự.

Vậy nên tham lam hơn, thay vì chọn một ký tự, ta có thể chọn một cụm ký tự. Đây là lý do họ thuật toán nén LZ (Lempel-Ziv) kết hợp cùng với Huffman. Nhóm LZ chia dữ liệu thành các khối rất hiệu quả.

Điển hình là thuật toán Deflate cùng các thuật toán tương tự phổ biến như zstd, LZX, Brotli, ... Hiện nay chuẩn nén quốc tế được quy định trong RFC của IETF để truyền tải dữ liệu sẽ sử dụng hai chuẩn chính là Deflate (RFC1951 - 5/1966) và Brotli (RFC 7932 - 7/2016) bởi hiệu suất cũng như tốc độ của nó.

b) Mô hình:



Hình IV.1.: Mô hình khi kết hợp LZ với Huffman

c) Cấu trúc của Deflate

Nén DEFLATE có sự linh hoạt lớn trong việc nén dữ liệu. Dữ liệu được chia thành các khối. Ở header mỗi khối được biểu diễn như sau:

- 1 bit biểu diễn còn khối đằng sau không.
- 2 bit biểu diễn chế độ nén của khối.
- 2 bit độ dài khối.
- đệm bit cho đủ 1 byte.

Deflate có ba chế độ nén mà bộ nén có sẵn:

- Không nén ứng với mã 00.
- Nén trước với LZ77 và sau đó với mã hóa Huffman sử dụng cây được định nghĩa bởi thông số Deflate ứng với mã 01.
- Nén trước với LZ77 và sau đó với mã hóa Huffman sử dụng cây do bộ nén tạo ra ứng với mã 10.

Đối với khối sử dụng cây do bộ nén tạo ra, bộ nén sẽ sẽ lưu hai cây vào phần dữ liệu của khối:

- Cây đầu tiên để ánh xạ bảng chữ cái và độ dài trùng lặp của con trỏ LZ77.
- Cây thứ hai cho khoảng cách lùi của con trỏ LZ77.

V. Kết hợp Huffman Coding với thuật toán LZW

Với ý tưởng từ biến thể kết hợp của các thuật toán nhóm LZ và Huffman, ta sẽ xây dựng thuật toán kết hợp Huffman Coding với thuật toán LZW.

1 Các bước thuật toán

1.1 Mã hóa

- Phân chia bằng LZ
  - Bước 1: Lấy từng kí tự.
  - Bước 2: Tạo một chuỗi mới gồm chuỗi a và kí tự vừa được lấy và kiểm tra chuỗi đó có trong từ điển chưa.
    - Nếu rồi thì gán chuỗi vừa rồi thành chuỗi a rồi chạy tiếp.
    - Nếu chưa, ta thêm vào mảng trả về chuỗi a, thêm chuỗi vừa rồi vào từ điển nếu nó nhỏ hơn kích thước từ điển và gán kí tự đang xét vào chuỗi a
  - Bước 3: Thêm kí tự còn lại vào mảng encoded.

  Do đặc trưng của LZ, nó sẽ tạo ra các chuỗi có độ dài tăng dần, có nhiều chuỗi là chuỗi con của 1 chuỗi khác trong dãy, vậy nên để tối ưu, chúng ta sẽ đệ quy để đảm bảo tỉ lệ 90% đầu ra đạt kích thước yêu cầu.
  Kết quả thu được là một mảng chứa các bộ kí tự. Các bộ này sẽ được gọi là từ.
- Bước 2: Tính tần số xuất hiện của mỗi từ trong chuỗi cần mã hóa và tạo một danh sách các nút lá, mỗi nút lá chứa một từ và tần số của nó.
- Bước 3: Sắp xếp danh sách các nút lá theo thứ tự tăng dần của tần số và lấy hai nút có tần số nhỏ nhất để tạo một nút cha mới, có giá trị bằng tổng tần số của hai nút con. Gán nhãn nút con trái là 0 và nút con phải là 1. Thêm nút cha vào danh sách và loại bỏ hai nút con khỏi danh sách.
- Bước 4: Lặp lại bước 2 cho đến khi chỉ còn một nút duy nhất trong danh sách, đó là nút gốc của cây Huffman. Cây Huffman được xây dựng từ dưới lên bằng cách

kết hợp các nút có tần số thấp nhất thành các nhánh mới.

- Bước 5: Duyệt cây Huffman từ gốc đến lá để tạo mã cho mỗi từ. Mã của một ký tự là chuỗi các bit biểu diễn đường đi từ gốc đến lá chứa ký tự đó.
- Bước 6: Mã hóa các từ chuỗi ban đầu bằng cách thay thế mỗi từ bằng mã tương ứng của nó.
- Bước 7: Sinh bảng Huffman. Giống như Huffman bình thường.
- Bước 8: Trả về bảng Huffman và chuỗi đã được mã hóa.

1.2 Giải mã

- Bước 1: Tạo cây Huffman từ bảng Huffman.
  - Bước 1: Lấy số nút trên cây.
  - Bước 2: Lấy thứ tự pre-order trên cây. Chiều dài mảng này luôn nhỏ hơn số nút là 1.
  - Bước 3: Sinh cây, xét từ vị trí nút gốc:
    - Gặp số 0, chúng ta sẽ thêm một cây con vào bên trái và xét tiếp tại nút này.
    - Gặp số 1, ta quay lui cho tới khi nào gặp nút cha chưa có cây con phải thì thêm cây con phải và xét nút vừa thêm.
  - Bước 4: Lấy tất cả nút lá theo thứ tự pre-order và chèn các từ.
- Bước 2: Giải mã chuỗi mã hóa bằng cách duyệt cây Huffman từ gốc đến lá theo từng bit trong chuỗi. Nếu bit là 0, chuyển sang nút con trái, nếu bit là 1, chuyển sang nút con phải. Khi gặp một nút lá, lấy từ trong nút lá và thêm vào chuỗi giải mã. Quay lại nút gốc và tiếp tục duyệt cho đến khi hết chuỗi mã hóa.

2 Cài đặt thuật toán

2.1 Cài đặt Node_Block

Chúng ta sẽ cài đặt Node tương tự như với thuật toán Huffman, tạo lớp kế thừa từ lớp `Node`, `Node_Block` ở trên

```
class Node_Block(Node):
    def to_table(self):
        node = 0
        tree = ''
```

```
        char = []
        if self.char is not None:
            return 1, tree, [self.char]
        else:
            node_l, tree_l, char_l = self.left.to_table() if self.left
                ↪ is not None else (0, '', [])
            node_r, tree_r, char_r = self.right.to_table() if self.
                ↪ right is not None else (0, '', [])
            char = char_l + char_r
            node = node_l + node_r + 1
            if self.left is not None:
                tree += '0' + tree_l
            if self.right is not None:
                tree += '1' + tree_r
            return [node, tree, char]
```

2.2 Encode

Tương tự như với thuật toán Huffman nhưng có một số chỉnh sửa để có thể hoạt động với list.

a) lzw_split: có hai tham số đầu vào là chuỗi kí tự và kích thước cho mỗi từ (ở đây mặc định 4 kí tự (128 bit))

```
def lzw_split(input_string, word_size = 4, dictionary = None, rate
    ↪ =0.9):
    if len(input_string) == 0 or input_string is None:
        return []
    if word_size > len(input_string):
        word_size = len(input_string)
    if dictionary is None:
        dictionary = []
    current_string = []
    encoded = []

    for char in input_string:
        current_string_plus_char = current_string + [char]
        if current_string_plus_char in dictionary:
            current_string = current_string_plus_char
```

```
        else:
            encoded.append(current_string) if len(current_string) != 0
                ↪ else None
            if len(current_string) < word_size:
                dictionary += [current_string_plus_char]
            current_string = [char]
    if current_string:
        encoded.append(current_string)
    ratio=0
    for i in encoded:
        if len(i) == word_size:
            ratio += 1
    if ratio < len(encoded) // 10 * int(rate * 10):
        encoded = lzw_split(input_string, word_size, dictionary)
    return encoded
```

b) calc_freq_list: đếm tần số xuất hiện của các cụm ký tự trong chuỗi

```
def calc_freq_list(data):
    list_block = {}
    for block in data:
        subdata = ''.join(block)
        if subdata in list_block:
            list_block[subdata]['freq'] += 1
        else:
            list_block[subdata] = {
                'freq' : 1,
                'char' : block
            }
    return list_block
```

c) build_tree_list chuyển dữ liệu thành cây Huffman

```
def build_tree_list(data):
    list_block = calc_freq_list(data)
    heap = []
    for block in list_block.values():
        heap.append(Node_Block(block['char'], block['freq']))
```

```
    while len(heap) > 1:
        heap.sort()
        left = heap.pop(0)
        right = heap.pop(0)
        heap.append(Node_Block(left=left, right=right))
    if heap[0].char is not None:
        return Node_Block(left=heap[0])
    return heap[0]
```

d) tree_to_dict_block: chuyển cây Huffman thành bảng mã

```
def tree_to_dict_block(root):
    def traverse(node, code):
        if node is None:
            return
        elif node.char is not None:
            return [[node.char, code]]
        else:
            return traverse(node.left, code + '0') + traverse(node.
                ↪ right, code + '1')
    return traverse(root, '')
```

e) encode_block: chuyển dữ liệu thành chuỗi mã hóa và bảng mã

```
def encode_block(data,word_size = 4):
    split_block = lzw_split(data,word_size)
    root = build_tree_list(split_block)
    huffman_dict = tree_to_dict_block(root)
    res = ''
    for block in split_block:
        for i in huffman_dict:
            if i[0] == block:
                res += i[1]

    res = [word_size] + root.to_table() + [res]
    return res
```

2.3 Decode

Tương tự như thuật toán Huffman nhưng có một vài chỉnh sửa để có thể hoạt động với đoạn mã hóa mới được trả về.

a) table_to_tree_block: trả về mã tương ứng với cụm ký tự trong bảng mã

```
def table_to_tree_block(code):
    wordsize = code[0]
    node = code[1] - 1# Lấy số nút trừ root
    preorder=code[2] # Lấy preorder củ a cây
    # Xây cây
    root = Node()
    current = root
    for i in range(len(preorder)):
        if preorder[i] == '0':
            current.left = Node()
            current.left.father = current
            current = current.left
        elif preorder[i] == '1':
            current = current.father
            while current.right is not None:
                current = current.father
            current.right = Node()
            current.right.father = current
            current = current.right
    # Chèn kí tự cho nút lá
    leafs = root.leafs()
    for i in range(len(leafs)):
        if len(code[3][i]) > wordsize:
            raise Exception('Invalid code')
        leafs[i].char = code[3][i]
    return root, code[-1]
```

b) decode_block: giải mã chuỗi mã hóa thành dữ liệu

```
def decode_block(code, huffman_tree=None):
    if huffman_tree is None:
```

```
        huffman_tree, code = table_to_tree_block(code)
    current = huffman_tree
    data = ''
    for c in code:
        if c == '0':
            current = current.left
        elif c == '1':
            current = current.right
        if current.char is not None:
            data += ''.join(current.char)
            current = huffman_tree
    return data
```

3 Đánh giá thuật toán

a) Độ phức tạp về thời gian

- Tính tần số xuất hiện của từng ký tự (list_char): O(n)
- Tạo ra danh sách các nút của cây: O(n)
- Thuật toán LZW: O(n)
- Tạo cây:
  - Sắp xếp các nút con (thuật toán TimSort): O(nlogn)
  - Tạo nút con trái/phải: O(1)
  - Thêm một nút con mới: O(1)

=> Độ phức tạp về thời gian của thuật toán: O(nlogn)

b) Độ phức tạp về không gian

- Tính tần số xuất hiện của từng ký tự (list_char): O(n)
- Tạo ra danh sách các nút của cây: O(n)
- Thuật toán LZW: O(n)
- Tạo cây:
  - Sắp xếp các nút con (thuật toán TimSort): O(n)
  - Tạo nút con trái/phải: O(1)
  - Thêm một nút con mới: O(1)

=> Độ phức tạp về không gian của thuật toán: O(n)

c) Đánh giá hiệu quả

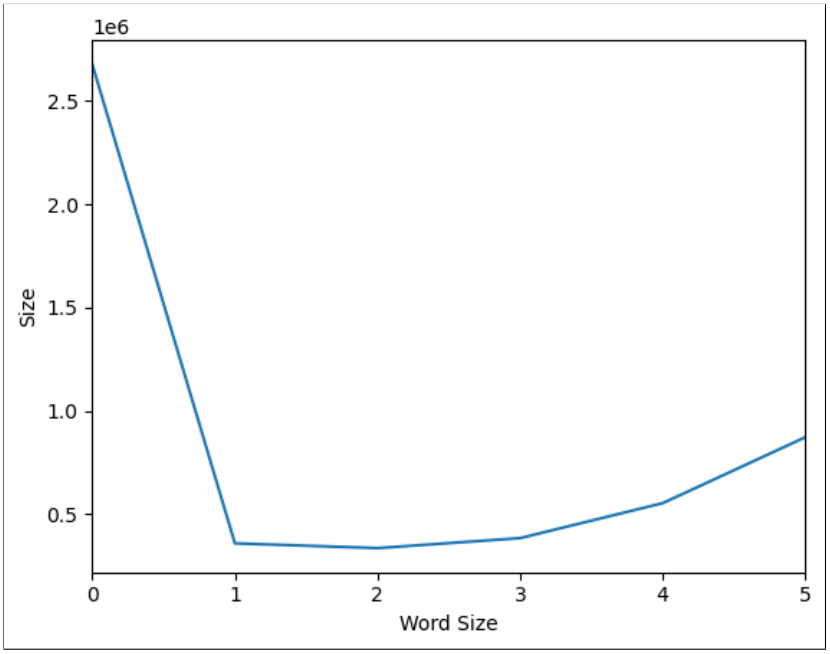
Giờ dữ liệu sau khi mã hóa sẽ thêm 1 số nữa là kích thước của mỗi node sẽ lưu word size - w. Trong thực tế người ta không thể đọc từng byte môt sẽ mất rất nhiều thời gian, chúng ta còn có kích thước cửa số, một lần đọc file sẽ đọc số lượng byte nhất định cho nhanh, kích thước từ sẽ rất khác nhau.

Để đơn giản hóa tính toán, chúng ta sẽ giả sử chia chuỗi được m khối, các khối đều bằng nhau. Như vậy ta có độ dài sau khi mã hóa là $w + a + (o-1) + f \times m \times w + e$

Để thuật toán hiệu quả thì $(n - m \times w) \times f > a + w + (o-1) + e$. Các tình huống xấu của thuật toán vẫn tương tự với trước khi cải tiến.

Để thấy được kích thước tối đa mỗi node ảnh hưởng như thế nào, ta sẽ vẽ đồ thị. a, w, f sẽ tạm tính 32 bit bới bộ dữ liệu là 30 lần đoạn văn. Chi tiết xem file đính kèm bài báo cáo

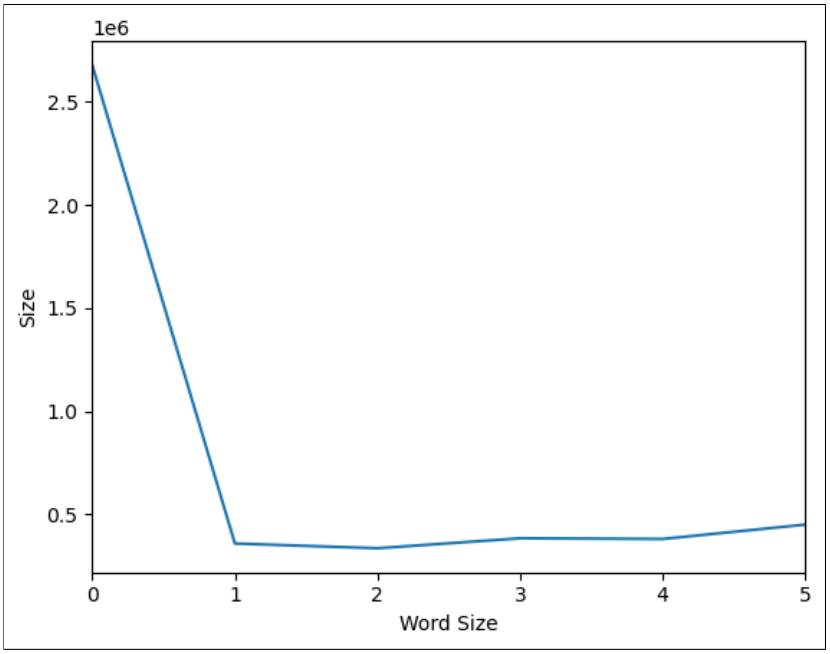
Tại trục hoành Word Size = 0 là dữ liệu gốc, Word Size = 1 là dữ liệu né chỉ với mỗi Huffman, Word Size > 1 là dữ liệu được chia bằng LZW rồi mới nén



Hình V.1.: Kết hợp với LZW

Ta thấy dữ liệu này có tính lặp lại cao nên sau khi nén, kích thước đã giảm rất nhiều, sau khi được chia với word size là 2, khối lượng dữ liệu đã nhỏ hơn 1 chút và từ đó điểm 3 trở về sau, kích thước bắt đầu to dần lên.

Do đặc tính của LZW đã nói ở trên, chúng ta sẽ chạy lại cho đủ 90%



Hình V.2.: Kết hợp với LZW cải tiến

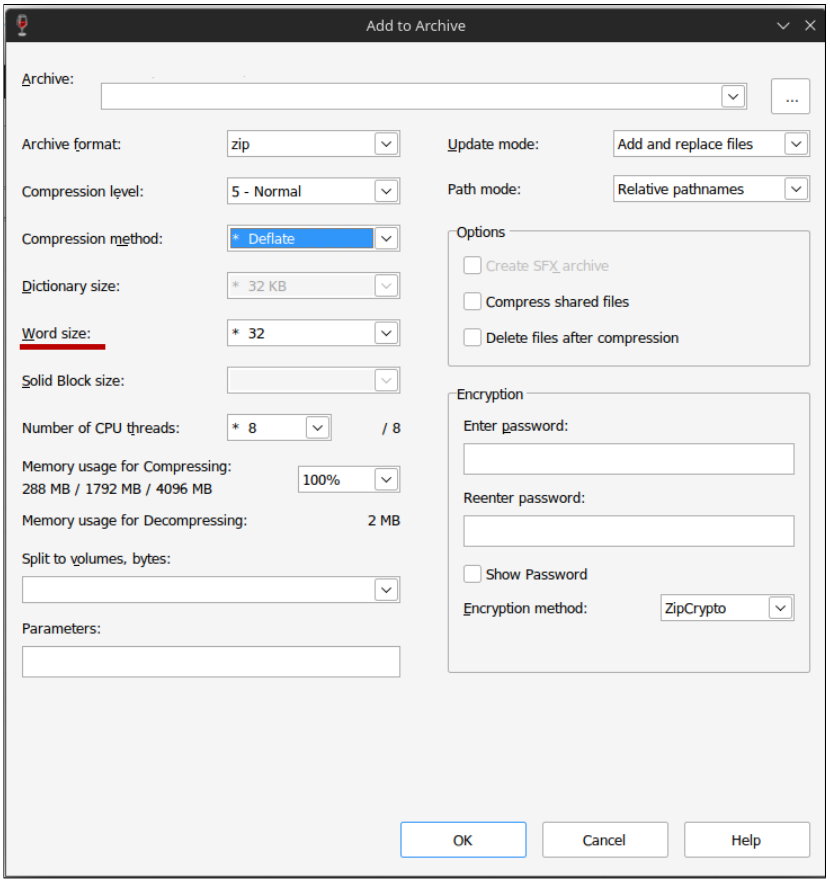
Điểm 1,2,3 thì không đổi còn 4 và 5 đã thấp hơn trước rất nhiều khi chia lại rất nhiều vậy nên, việc chia lại sẽ rất cần thiết khi dữ liệu lớn. Nhưng đây cũng là bài toán đánh đổi thời gian và hiệu suất

Với mỗi dữ liệu, chúng ta sẽ cần 1 word size phù hợp. Word size càng nhỏ các tập sẽ càng lặp nhiều thì tần số của các block sẽ càng lớn. Còn nếu word size càng lớn thì các tập sẽ càng khác nhau thì tần số của các block sẽ càng thấp.

Ta có bài toán tối ưu: Tìm w sao cho $a + (o-1) + \Sigma_{i=1}^{m} l + w_i \times f + e$ là nhỏ nhất với w_i là kích thước từ của từng node và l là số bit để lưu độ dài của w_i và $w_i < w$. Bên cạnh đó trong thực tế, chúng ta còn phải tính thêm kích thước cửa sổ để có thể tăng tốc độ tính toán nhưng việc này cũng làm giảm hiệu suất nén của thuật toán

Nhưng không, chưa ai giải được bài toán trên hết, muốn biết cỡ nào tốt nhất, hiện tại chúng ta phải thử. Các phần mềm sẽ cho chọn word size.

Ví dụ: Phần mềm 7-Zip với mã hóa Deflate được kết hợp từ LZ77 và Huffman. Word size mặc định 32



Hình V.3.: Deflate trong 7-Zip

Nếu chúng ta chọn Word size quá lớn, nén dữ liệu sẽ không còn hiệu quả nữa. Thử lại với đoạn 4 câu đầu của chuyện Kiều với thuật toán vừa xây dựng

Dữ liệu	**Kích thước**	**Tỉ lệ nén**
Ban đầu	4192	-
Huffman	2082	2.01
Chia bằng LZW với kích thước từ là 2 Nén Huffman	4001	1.04
Chia bằng LZW với kích thước từ là 3 Nén Huffman	4628	0.90

Bảng V.1.: Tổng hợp lại kết quả khi chạy Huffman kết hợp LZW

Chúng ta khó có thể so sánh với thuật toán Deflate, chuẩn nén quốc tế bây giờ do nó sử dụng cả Bảng mã Huffman cố định và chỉ xây dựng Bảng mã Huffman linh động khi cần thiết đồng thời có các khối không được nén

VI. Kết luận

1 Tóm tắt kết quả

Bài báo cáo đã tìm hiểu về thuật toán Huffman và thực thi thuật toán cụ thể bằng ngôn ngữ python. Nhóm cũng đã tạo được minh họa visualization sinh động cho Huffman Coding và Huffman Decode. Từ những hạn chế của thuật toán Huffman cơ bản, nhóm đã thử cải tiến thuật toán bằng cách kết hợp cùng với thuật toán nén LZW và so sánh hiệu quả cải tiến. Ngoài ra, chúng em đã cũng tìm hiểu thêm về những cách cải tiến / biến thể khác của thuật toán, so sánh ưu nhược điểm giữa các thuật toán.

2 Hướng đi tương lai

- Áp dụng Huffman coding cho dữ liệu hình ảnh, âm thanh,...
- Xây dựng ứng dụng chương trình nén sử dụng thuật toán nén hiệu quả, phù hợp nhất.

Tài liệu tham khảo

[1] Anany Levitin *Introduction to the Design and Analysis of Algorithms-Pearson (2012)*

[2] Huffman coding: https://en.wikipedia.org/wiki/Huffman-coding

[3] Huffman Coding | Greedy Algo-3: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

[4] Lossless compression algorithms: https://en.wikipedia.org/wiki/Category:Lossless-compression-algorithms

[4] Larmore and Hirschberg (1990). *A fast algorithm for optimal length-limited Huffman codes*. Journal of the ACM (JACM), 37(3), 464-473.

[5] An Explanation of the DEFLATE Algorithm - Antaeus Feldspar (23 August 1997) https://www.cs.ucdavis.edu/ martel/122a/deflate.html

[6] Deflate Algorithm: https://thuc.space/posts/deflate