ĐẠI HỌC BÁCH KHOA HÀ NỘI
TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

BÀI TẬP LỚN

Lý thuyết thông tin

PHẠM NGỌC HẢI
hai.pn207601@sis.hust.edu.vn

Ngành Công nghệ Thông tin

Giảng viên hướng dẫn: TS. Trịnh Văn Chiến _____

Chữ kí GVHD

Khoa: Kỹ thuật máy tính

Trường: Công nghệ thông tin và Truyền thông

HÀ NỘI, 06/2023

MỤC LỤC

CHƯƠNG 1. GIỚI THIỆU BÀI TOÁN

1.1 Bài toán 1

a) Nhập vào ma trận xác suất kết hợp P(x,y) có cỡ MxN với M và N nhập từ bàn phím. Cảnh báo nếu nhập xác suất âm, yêu cầu nhập lại.

b) Tính và hiển thị H(X), H(Y), H(X|Y), H(Y|X), H(X,Y), H(Y) - H(Y|X), I(X;Y)

c) Tính D(P(x)||P(y)) và D(P(y)||P(x))

1.2 Bài toán 2

a) Nhập vào một chuỗi ký tự không dấu có chiều dài bất kỳ, không phân biệt chữ hoa, chữ thường.

b) Mã hóa Huffman cho chuỗi trên

c) Mã hóa Shannon-Fano cho chuỗi trên, tính hiệu suất mã hóa, và tính dư thừa.

CHƯƠNG 2. PHÂN TÍCH YÊU CẦU

2.1 Bài toán 1

Để tính các giá trị ở câu b và c, ta cần áp dụng các công thức sau:

1. Entropy (Self-information: lượng tin riêng):

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$
$$H(Y) = -\sum_{y \in Y} p(y) \log_2 p(y)$$

2. Join Entropy (Lượng tin chung):

$$H(X,Y) = -\sum_{x \in X} \sum_{y \in Y} p(x,y) \log_2 p(x,y)$$

3. Conditional Entropy (Lượng tin riêng có điều kiện):

$$H(Y|X) = -\sum_{x \in X} \sum_{y \in Y} p(x,y) \log_2 p(y|x)$$
$$H(X|Y) = -\sum_{x \in X} \sum_{y \in Y} p(x,y) \log_2 p(x|y)$$

4. Mutual Information (Thông tin tương hỗ):

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)}$$

5. Relative Entropy (Lượng tin riêng tương đối):

$$D(p(x)||q(x)) = \sum_{x \in X} p(x) \log_2 \frac{p(x)}{q(x)}$$

6. Mối quan hệ giữa Joint Entropy và Conditional Entropy:

$$H(X,Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$$

7. Mối quan hệ giữa Mutual Information và Entropy:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) = H(X) + H(Y) - H(X,Y)$$

2.2 Bài toán 2

2.2.1 Mã hóa Huffman

Mã hóa Huffman là một thuật toán nén dữ liệu mà sử dụng mã hóa độ dài biến để biểu diễn thông tin một cách hiệu quả. Thuật toán này được phát minh bởi David A. Huffman vào năm 1952.

Ý tưởng chính của mã hóa Huffman là sử dụng mã hóa với độ dài biến cho các ký tự khác nhau dựa trên tần số xuất hiện của chúng trong dữ liệu ban đầu. Các ký tự có tần số xuất hiện cao hơn sẽ được mã hóa với độ dài mã ngắn hơn, trong khi các ký tự có tần số xuất hiện thấp hơn sẽ được mã hóa với độ dài mã dài hơn.

Thuật toán mã hóa Huffman bắt đầu bằng việc xây dựng một cây Huffman từ tần số xuất hiện của các ký tự. Cây Huffman là một cấu trúc dữ liệu cây nhị phân có các ký tự trên các nút lá và các nút không phải lá đại diện cho các tổ hợp của các ký tự. Quá trình xây dựng cây Huffman sẽ xếp các nút theo tần số xuất hiện và tạo nút mới từ hai nút có tần số thấp nhất cho đến khi chỉ còn một nút duy nhất.

Sau khi cây Huffman được xây dựng, các mã Huffman được gán cho từng ký tự. Điều này đảm bảo rằng không có mã nào là tiền tố của mã khác, đồng nghĩa rằng không có từ nào có thể được hiểu sai khi giải mã. Mã Huffman của một ký tự là đường dẫn từ gốc đến nút lá tương ứng với ký tự đó.

Khi mã hóa dữ liệu, mỗi ký tự trong dữ liệu được thay thế bằng mã Huffman tương ứng của nó. Kết quả là dữ liệu được nén với độ dài nhỏ hơn so với dữ liệu ban đầu, đặc biệt là khi các ký tự phổ biến có độ dài mã ngắn.

Khi giải nén dữ liệu, mã Huffman được giải mã bằng cách đi từ gốc đến nút lá tương ứng với mã nhận được để khôi phục lại dữ liệu ban đầu.

Mã hóa Huffman được sử dụng rộng rãi trong lĩnh vực nén dữ liệu và được coi là một trong những phương pháp nén không mất mát hiệu quả nhất.

2.2.2 Mã hóa Shannon-Fano

Mã hóa Shannon-Fano là một phương pháp mã hóa dữ liệu được phát minh bởi Claude Shannon và Robert Fano vào những năm 1940. Nó cũng là một phương pháp nén dữ liệu nhưng khác với mã hóa Huffman, nó sử dụng mã hóa với độ dài biến dựa trên xác suất xuất hiện của các ký tự.

Quá trình mã hóa Shannon-Fano bắt đầu bằng việc sắp xếp các ký tự theo thứ tự giảm dần của xác suất xuất hiện. Sau đó, các ký tự được chia thành hai phân đoạn sao cho tổng xác suất xuất hiện của các ký tự trong mỗi phân đoạn là gần bằng nhau. Quá trình này được lặp lại cho từng phân đoạn cho đến khi không thể phân chia được nữa.

Sau khi tạo các phân đoạn, mã Shannon-Fano được gán cho từng ký tự. Mã của một ký tự được tạo bằng cách thêm một bit 0 hoặc 1 vào mã của phân đoạn chứa ký tự đó. Mã của một ký tự sẽ là đường dẫn từ gốc đến phân đoạn chứa ký tự đó, được xác định bằng cách theo các bit 0 và 1.

Khi mã hóa dữ liệu, mỗi ký tự trong dữ liệu được thay thế bằng mã Shannon-Fano tương ứng của nó. Kết quả là dữ liệu được nén với độ dài nhỏ hơn so với dữ liệu ban đầu, tương tự như mã hóa Huffman.

Tuy nhiên, mã hóa Shannon-Fano thường không hiệu quả như mã hóa Huffman

trong việc nén dữ liệu, vì phân chia dựa trên xác suất không đảm bảo tối ưu hóa tổng độ dài mã. Mã hóa Huffman thường được sử dụng rộng rãi hơn trong thực tế.

CHƯƠNG 3. CÔNG NGHỆ SỬ DỤNG

3.1 Ngôn ngữ lập trình Python

Python là một ngôn ngữ lập trình thông dịch, được phát triển bởi Guido van Rossum vào những năm 1980. Python được thiết kế để đơn giản hóa việ lập trình và tăng cường tính hiệu quả của các ứng dụng. Python có cú pháp đơn giản, dễ đọc và dễ học, giúp cho người mới bắt đầu có thể nhanh chóng tiếp cận và phát triển các ứng dụng.

Python được sử dụng rộng rãi trong nhiều lĩnh vực, bao gồm khoa học dữ liệu, trí tuệ nhân tạo, web development, game development, và nhiều lĩnh vực khác. Python cung cấp nhiều thư viện và framework hỗ trợ cho các lĩnh vực này, giúp cho việc phát triển ứng dụng nhanh chóng và hiệu quả.

Một trong những đặc điểm nổi bật của là tính đa năng. Python có thể được sử dụng để phát triển các ứng dụng desktop, web, mobile, và cả các ứng dụng trên các thiết bị nhúng. Điều này giúp cho Python trở thành một trong những ngôn ngữ lập trình phổ biến nhất hiện nay.

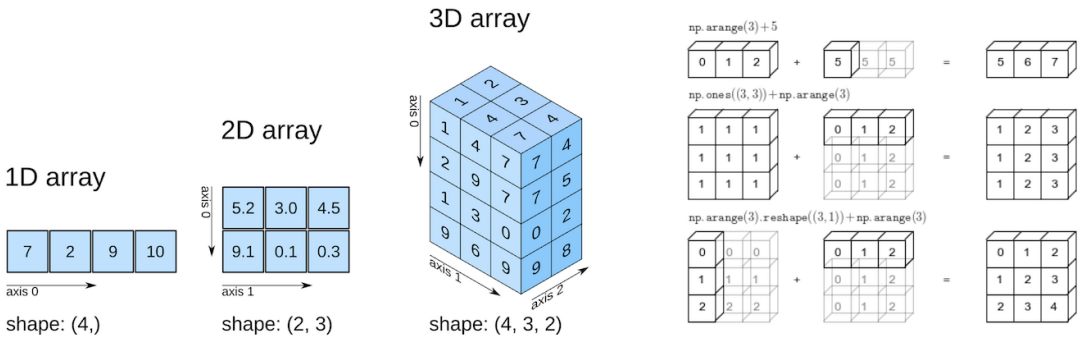
Ngoài ra, Python còn có cộng đồng lớn và nhiều tài liệu hướng dẫn. Cộng đồng Python rất tích cực và hỗ trợ nhau trong việc giải quyết các vấn đề lập trình. Nếu bạn gặp phải vấn đề trong quá trình lập trình bằng Python, bạn có thể tìm kiếm giải pháp trên các diễn đàn hoặc nhóm trên mạng.

Cuối cùng, Python là một ngôn ngữ lập trình miễn phí và mã nguồn mở. Điều này có nghĩa là bạn có thể tải về và sử dụng Python mà không cần phải trả bất kỳ khoản phí nào. Bạn cũng có thể tùy chỉnh mã nguồn của Python để phù hợp với nhu cầu của mình.

3.2 Thư viện Numpy

NumPy là một thư viện Python mã nguồn mở được sử dụng rộng rãi trong khoa học dữ liệu và tính toán khoa học. NumPy cung cấp một đối tượng mảng đa chiều (ndarray), cho phép lưu trữ và xử lý dữ liệu nhanh chóng và hiệu quả hơn so với

các cấu trúc dữ liệu khác trong Python.



Một trong những ưu điểm của NumPy là tính tối ưu hóa cao. NumPy được phát triển trên interface của C, giúp cho nó hoạt động rất nhanh trên Python. Các hàm tính toán đại số được tối ưu để cho tốc độ cao. Khi bắt đầu tiếp cận machine learning, chúng ta cần thành thạo cách xử lý dữ liệu và tính toán trên NumPy.

NumPy cũng cung cấp nhiều hàm tính toán và phương thức xử lý mảng, giúp cho việc xử lý dữ liệu trở nên dễ dàng hơn. Ngoài ra, NumPy còn có khả năng tích hợp với các thư viện khác trong Python như SciPy, Pandas, Matplotlib, và nhiều thư viện khác.

Với những ưu điểm trên, NumPy đã trở thành một trong những thư viện quan trọng nhất trong khoa học dữ liệu và tính toán khoa học.

3.3 Module Counter trong thư viện colections

Module Counter trong thư viện collections cung cấp một cách đơn giản và hiệu quả để đếm và theo dõi tần số xuất hiện của các phần tử trong một đối tượng có thể lặp (iterable). Nó là một công cụ mạnh mẽ để phân tích và xử lý dữ liệu.

Để sử dụng module Counter, trước tiên bạn cần import nó từ thư viện collections như sau:

from collections import Counter

Sau đó, bạn có thể tạo một đối tượng Counter bằng cách truyền vào một iterable như danh sách, chuỗi, hoặc bất kỳ đối tượng có thể lặp nào. Ví dụ:

my_list = [’a’, ’b’, ’c’, ’a’, ’b’, ’a’]

my_counter = Counter(my_list)

Trong ví dụ trên, my_counter sẽ đếm tần số xuất hiện của mỗi phần tử trong my_list. Kết quả sẽ là một đối tượng Counter với các phần tử là các cặp key-value, trong đó key là phần tử trong my_list, và value là số lần xuất hiện của key trong my_list.

Bạn có thể truy cập vào các phần tử và tần số bằng cách sử dụng cú pháp dictionary thông thường. Ví dụ:

print(my_counter['a']) # Output: 3

print(my_counter['b']) # Output: 2

print(my_counter['c']) # Output: 1

Ngoài ra, module Counter cung cấp các phương thức hữu ích như most_common() để trả về danh sách các phần tử theo thứ tự giảm dần của tần số xuất hiện, hoặc update() để cập nhật đếm với một iterable mới.

Module Counter rất hữu ích trong việc đếm và phân tích dữ liệu, đặc biệt trong các tác vụ như tính toán tần số xuất hiện, xử lý văn bản, phân tích ngôn ngữ tự nhiên, và các tác vụ liên quan đến thống kê và phân tích dữ liệu.

3.4 Thư viện heapq

Thư viện 'heapq' trong Python cung cấp các chức năng để làm việc với các cấu trúc dữ liệu heap (đống). Heap là một cấu trúc dữ liệu cây nhị phân đặc biệt có tính chất đặc thù là phần tử gốc (root) luôn có giá trị nhỏ hơn hoặc bằng các phần tử con của nó (nếu có). Thư viện 'heapq' cung cấp các chức năng để thực hiện các thao tác cơ bản trên heap như thêm phần tử, loại bỏ phần tử nhỏ nhất, truy xuất phần tử nhỏ nhất, và xây dựng heap từ một danh sách.

Để sử dụng thư viện 'heapq', trước tiên bạn cần import nó từ thư viện chuẩn như sau:

Các chức năng chính trong thư viện 'heapq' bao gồm:

- 'heapify(iterable)': Chuyển đổi một iterable thành một heap.

- 'heappush(heap, item)': Thêm một phần tử vào heap và duy trì tính chất của heap.

- 'heappop(heap)': Loại bỏ và trả về phần tử nhỏ nhất trong heap và duy trì tính chất của heap.

- 'heappushpop(heap, item)': Thêm một phần tử vào heap và trả về phần tử nhỏ nhất. Tương đương với việc thực hiện 'heappush(heap, item)' sau đó 'heappop(heap)', nhưng hiệu quả hơn vì chỉ cần điều chỉnh heap một lần.

- 'heapreplace(heap, item)': Loại bỏ và trả về phần tử nhỏ nhất trong heap, sau đó thêm phần tử mới vào heap. Tương đương với việc thực hiện 'heappop(heap)' sau đó 'heappush(heap, item)', nhưng hiệu quả hơn vì chỉ cần điều chỉnh heap một lần.

- ‘nlargest(k, iterable)‘: Trả về danh sách gồm k phần tử lớn nhất từ iterable.

- ‘nsmallest(k, iterable)‘: Trả về danh sách gồm k phần tử nhỏ nhất từ iterable.

Các chức năng này cho phép bạn tạo, sắp xếp và truy xuất các phần tử trong heap một cách hiệu quả với độ phức tạp thời gian O(log n), nơi n là số lượng phần tử trong heap.

Thư viện ‘heapq‘ được sử dụng rộng rãi trong nhiều bài toán như tìm kiếm phần tử nhỏ/lớn nhất, xử lý hàng đợi ưu tiên (priority queue), tìm k phần tử nhỏ/lớn nhất, và giải các thuật toán như thuật toán Dijkstra, thuật toán Prim, và nhiều thuật toán khác liên quan đến đồ thị và tối ưu hóa.

CHƯƠNG 4. THỰC THI GIẢI PHÁP VÀ ĐÁNH GIÁ

4.1 Giải bài tập 1

Dưới đây là mã nguồn đầy đủ của chương trình:

```python
import numpy as np


def input_matrix(M, N):
    P = np.zeros((M, N))
    for i in range(M):
        for j in range(N):
            while True:
                value = float(input(f"\tEnter P(x{i+1}, y{j+1}): "))
                if value >= 0:
                    P[i, j] = value
                    break
                else:
                    print("Probability cannot be negative. Please re-enter!")
    return P


def entropy(P):
    return -np.sum(P*np.log2(P, where=P != 0))


def joint_entropy(P):
    return entropy(P)


def marginal_probability(P, axis):
    return np.sum(P, axis=axis)


def conditional_entropy(P, axis):
    marginal = marginal_probability(P, 1 - axis)
    return entropy(P) - entropy(marginal)


def mutual_information(P):
    P_x = marginal_probability(P, axis=1)
```

```
    P_y = marginal_probability(P, axis=0)
    return entropy(P_x) + entropy(P_y) - joint_entropy(P)


def kullback_leibler_divergence(P, Q):
    return np.sum(P * np.log2(P/Q, where=(P != 0) & (Q !=
    ↪  0)))


def print_matrix(P):
    P_x = marginal_probability(P, axis=1)
    P_y = marginal_probability(P, axis=0)
    print("".center(8, ' '), end=' ')
    for i in range(P.shape[1]):
        print(f"y{i+1}".center(8, ' '), end=' ')
    print("P(x)".center(8, ' '))
    for i in range(P.shape[0]):
        print(f"x{i+1}".center(8, ' '), end=' ')
        for j in range(P.shape[1]):
            print(f"{P[i, j]:.5f}".center(8, ' '), end=' ')
        print(f"{P_x[i]:.5f}".center(8, ' '))
    print("P(y)".center(8, ' '), end=' ')
    for i in range(P.shape[1]):
        print(f"{P_y[i]:.5f}".center(8, ' '), end=' ')
    print()


def main():
    # Sample matrix
    # P = [[0.0625, 0.0625, 0],
    #      [0.375, 0.1875, 0.1875],
    #      [0.0625, 0, 0.0625]]
    # P = np.array(P)

    # Enter the number of rows and columns of the matrix
    M = int(input("Enter number of rows: "))
    N = int(input("Enter number of columns: "))
    P = input_matrix(M, N)
    print(" Input ".center(50, '-'))
    print()
    print_matrix(P)

    # Calculate the entropy, joint entropy, marginal
    ↪  probability, conditional entropy, mutual
    ↪  information, and Kullback-Leibler divergence
```

```
    P_x = marginal_probability(P, axis=1)
    P_y = marginal_probability(P, axis=0)
    H_x = entropy(P_x)
    H_y = entropy(P_y)
    H_xy = joint_entropy(P)
    H_y_given_x = conditional_entropy(P, axis=0)
    H_x_given_y = conditional_entropy(P, axis=1)
    MI_xy = mutual_information(P)
    D_Px_Py = kullback_leibler_divergence(P_x, P_y)
    D_Py_Px = kullback_leibler_divergence(P_y, P_x)

    # Print the results
    print()
    print(" Results ".center(50, '-'))
    print()
    print(f"H(X) = {H_x:.5f} bits")
    print(f"H(Y) = {H_y:.5f} bits")
    print(f"H(X,Y) = {H_xy:.5f} bits")
    print(f"H(Y|X) = {H_y_given_x:.5f} bits")
    print(f"H(X|Y) = {H_x_given_y:.5f} bits")
    print(f"H(Y)-H(Y|X) = {H_y-H_y_given_x:.5f} bits")
    print(f"I(X;Y) = {MI_xy:.5f} bits")
    print(f"D(Px||Py) = {D_Px_Py:.5f} bits")
    print(f"D(Py||Px) = {D_Py_Px:.5f} bits")
    print()
    print("-"*50)

if __name__ == "__main__":
    main()
```

Kết quả:

```
Enter number of rows:3
Enter number of columns: 3
        Enter P(x1, y1): 0.0625
        Enter P(x1, y2): 0.0625
        Enter P(x1, y3): 0
        Enter P(x2, y1): 0.375
        Enter P(x2, y2): 0.1875
        Enter P(x2, y3): 0.1875
        Enter P(x3, y1): 0.0625
        Enter P(x3, y2): 0
        Enter P(x3, y3): 0.0625
```

```
---------------------- Input ----------------------

            y1       y2       y3       P(x)
   x1    0.06250  0.06250  0.00000  0.12500
   x2    0.37500  0.18750  0.18750  0.75000
   x3    0.06250  0.00000  0.06250  0.12500
  P(y)   0.50000  0.25000  0.25000

--------------------- Results ---------------------

H(X) = 1.06128 bits
H(Y) = 1.50000 bits
H(X,Y) = 2.43628 bits
H(Y|X) = 1.37500 bits
H(X|Y) = 0.93628 bits
H(Y)-H(Y|X) = 0.12500 bits
I(X;Y) = 0.12500 bits
D(Px||Py) = 0.81372 bits
D(Py||Px) = 0.85376 bits

---------------------------------------------------
```

4.2 Giải bài tập 2

Dưới đây là mã nguồn đầy đủ của chương trình:

```python
from collections import Counter
import heapq
import numpy as np


def huffman_encoding(input_string):
    freqs = Counter(input_string)
    heap = []
    for char, freq in freqs.items():
        heap.append([freq, [char, ""]])
    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:])

    return dict(sorted(heap[0][1:], key=lambda x:
    ↪ len(x[1])))


def shannon_fano_encoding(input_string):
    freqs = Counter(input_string)
    symbols = [["", (char, freq)] for char, freq in
    ↪ freqs.items()]
    symbols.sort(key=lambda x: x[1][1], reverse=True)

    def find_optimal_pivot(symbols):
        pivot = 0
        size = len(symbols)
        diff_min = float("inf")

        for i in range(size-1):
            left_sum = sum([freq for _, (_, freq) in
            ↪ symbols[:i+1]])
            right_sum = sum([freq for _, (_, freq) in
            ↪ symbols[i+1:]])
            diff = abs(left_sum - right_sum)
            if diff < diff_min:
                diff_min = diff
                pivot = i

        return pivot

    def shannon_fano(symbols):
        if len(symbols) == 1:
            return
        elif len(symbols) == 2:
            symbols[0][0] += "0"
            symbols[1][0] += "1"
            return
        else:
            pivot = find_optimal_pivot(symbols)
            for symbol in symbols[:pivot+1]:
                symbol[0] += "0"
            for symbol in symbols[pivot+1:]:
                symbol[0] += "1"
            shannon_fano(symbols[:pivot+1])
```

```
            shannon_fano(symbols[pivot+1:])

    shannon_fano(symbols)
    shannon_fano_tree = {char: code for code, (char, _) in symbols}
    return shannon_fano_tree


def encoding_efficiency(input_string, encoding_tree):
    total_chars = len(input_string)
    total_bits = sum(len(encoding_tree[char]) for char in input_string)

    everage_code_length = total_bits / total_chars

    freqs = Counter(input_string)
    P = [freq / total_chars for freq in freqs.values()]
    H = -np.sum(P * np.log2(P))

    return H / everage_code_length


def redundancy(input_string, encoding_tree):
    return 1 - encoding_efficiency(input_string, encoding_tree)


def main():
    str = input("Enter a string: ").lower()
    huffman_tree = huffman_encoding(str)
    print("Huffman encoding:")
    for char, code in huffman_tree.items():
        print(f"\t{char}: {code}")

    encoded_str = ""
    for char in str:
        encoded_str += huffman_tree[char]

    print(f"Encoded string by Huffman encoding: {encoded_str}")
    print(
        f"Huffman encoding efficiency: {encoding_efficiency(str, huffman_tree)}")
    print(f"Huffman encoding redundancy: {redundancy(str, huffman_tree)}")
```

```
97
98      print()
99      shannon_fano_tree = shannon_fano_encoding(str)
100     print("Shannon-Fano encoding:")
101     for char, code in shannon_fano_tree.items():
102         print(f"\t{char}: {code}")
103
104     encoded_str = ""
105     for char in str:
106         encoded_str += shannon_fano_tree[char]
107
108     print(f"Encoded string by Shannon-Fano encoding:
        ↪ {encoded_str}")
109     print(
110         f"Shannon-Fano encoding efficiency:
            ↪ {encoding_efficiency(str, shannon_fano_tree)}")
111     print(
112         f"Shannon-Fano encoding redundancy: {redundancy(str,
            ↪ shannon_fano_tree)}")
113
114
115 if __name__ == "__main__":
116     main()
117
```

Kết quả:

```
1  Enter a string: PHAM NGOC HAI
2  Huffman encoding:
3          n: 000
4          o: 001
5          p: 010
6           : 011
7          a: 100
8          h: 110
9          c: 1010
10         g: 1011
11         i: 1110
12         m: 1111
13 Encoded string by Huffman encoding:
   ↪ 01011010011110110001011001101001111010011110
14 Huffman encoding efficiency: 0.9792027054845164
15 Huffman encoding redundancy: 0.020797294515483622
16
```

```
Shannon-Fano encoding:
        h: 00
        a: 010
         : 011
        p: 100
        m: 1010
        n: 1011
        g: 1100
        o: 1101
        c: 1110
        i: 1111
Encoded string by Shannon-Fano encoding:
 ↪ 10000010101001110111100110111100110001011111
Shannon-Fano encoding efficiency: 0.9792027054845164
Shannon-Fano encoding redundancy: 0.020797294515483622
```

4.2.1 Giải thích triển khai thuật toán mã hóa Huffman

Đầu tiên, chúng ta sử dụng module Counter từ thư viện collections để đếm số lần xuất hiện của mỗi ký tự trong input_string. Kết quả được lưu trong biến freqs.

Khởi tạo một heap (đống) rỗng và duyệt qua từng ký tự và tần số tương ứng trong freqs. Mỗi ký tự và tần số được đóng gói vào một list [freq, [char, ""]], sau đó được thêm vào heap. Mục đích của việc này là tạo ra các nút lá ban đầu trong cây Huffman.

Sử dụng hàm heapq.heapify để biến heap thành một cây heap (đống). Điều này sắp xếp các phần tử trong heap theo tần số xuất hiện, đảm bảo rằng các phần tử có tần số thấp nhất sẽ được xử lý trước.

Bắt đầu quá trình xây dựng cây Huffman bằng cách lặp lại cho đến khi chỉ còn một phần tử duy nhất trong heap. Trong mỗi vòng lặp, hai phần tử có tần số thấp nhất được lấy ra (lo và hi). Sau đó, các bit '0' được thêm vào mã của tất cả các ký tự trong lo và các bit '1' được thêm vào mã của tất cả các ký tự trong hi. Kết quả là lo và hi được ghép lại thành một phần tử mới, có tần số là tổng tần số của lo và hi. Phần tử mới này được đẩy vào heap.

Khi chỉ còn một phần tử duy nhất trong heap, ta lấy danh sách mã hóa của nó từ pair[1:] và sắp xếp theo độ dài mã (theo chiều tăng dần). Kết quả cuối cùng là một từ điển (dictionary) chứa các ký tự và mã Huffman tương ứng.

Trả về từ điển mã Huffman đã được sắp xếp.

4.2.2 Giải thích triển khai thuật toán mã hóa Shannon-Fano

Đầu tiên, ta sử dụng module Counter để đếm số lần xuất hiện của mỗi ký tự trong input_string. Kết quả được lưu trong biến freqs.

Tạo một list symbols để lưu trữ các ký tự và tần số tương ứng của chúng. Mỗi phần tử trong symbols là một list chứa một ký tự, tần số và một chuỗi mã (ban đầu là chuỗi rỗng).

Sắp xếp symbols theo thứ tự giảm dần của tần số, sử dụng hàm sort và key=lambda x: x[1][1]. Điều này đảm bảo rằng các phần tử với tần số cao nhất được xử lý trước.

Định nghĩa hàm find_optimal_pivot để tìm điểm chia tối ưu (pivot) trong symbols. Hàm này duyệt qua các phần tử của symbols và tính toán tổng tần số của các phần tử bên trái và bên phải pivot, sau đó tìm sự khác biệt nhỏ nhất giữa hai tổng này. Pivot là chỉ số của phần tử có sự khác biệt nhỏ nhất. Hàm trả về giá trị pivot.

Định nghĩa hàm shannon_fano để thực hiện mã hóa Shannon-Fano. Nếu symbols chỉ có một phần tử, không làm gì cả. Nếu symbols có hai phần tử, gán mã "0" cho phần tử đầu tiên và mã "1" cho phần tử thứ hai. Trong trường hợp khác, tìm pivot tối ưu bằng cách gọi hàm find_optimal_pivot. Sau đó, gán mã "0" cho các phần tử bên trái pivot và mã "1" cho các phần tử bên phải pivot. Gọi đệ quy shannon_fano cho symbols bên trái pivot và symbols bên phải pivot.

Gọi hàm shannon_fano với list symbols ban đầu để thực hiện mã hóa Shannon-Fano.

Tạo một từ điển shannon_fano_tree từ list symbols. Mỗi phần tử trong symbols đại diện cho một ký tự và mã tương ứng. Từ điển này sẽ được trả về là kết quả cuối cùng.

Trả về từ điển shannon_fano_tree.

CHƯƠNG 5. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

5.1 Kết luận

Bài toán 1: Tôi đã thành công trong việc xây dựng chương trình để tính toán các đại lượng thông tin như entropy, joint entropy, conditional entropy, mutual information và Kullback-Leibler divergence dựa trên ma trận xác suất kết hợp. Chương trình cho phép người dùng nhập ma trận xác suất và tính toán các đại lượng thông tin liên quan một cách chính xác. Kết quả thu được từ chương trình cung cấp thông tin quan trọng về tính toán và phân tích thông tin.

Bài toán 2: Tôi đã thành công trong việc triển khai mã hóa Huffman và mã hóa Shannon-Fano cho một chuỗi đầu vào. Chương trình cho phép người dùng mã hóa chuỗi và tính toán hiệu suất mã hóa cũng như dư thừa của hai phương pháp mã hóa. Bằng cách sử dụng thư viện heapq và collections.Counter, tôi đã tạo ra cây Huffman và cây Shannon-Fano và xây dựng mã hóa tương ứng. Kết quả thu được từ chương trình giúp hiểu rõ hơn về hiệu suất và dư thừa của hai phương pháp mã hóa.

5.2 Định hướng phát triển

5.2.1 Bài toán 1

Tăng tính tương tác của chương trình: Hiện tại, chương trình yêu cầu người dùng nhập ma trận xác suất thủ công từ bàn phím. Bạn có thể mở rộng chương trình để cho phép người dùng nhập ma trận từ file hoặc tự động tạo ma trận xác suất dựa trên dữ liệu đầu vào.

Thực hiện phân tích thông tin thêm: Bên cạnh các đại lượng thông tin đã tính toán, bạn có thể xem xét thêm các khía cạnh khác của thông tin như độ không chắc chắn tương đối, thông tin tối đa và cân bằng thông tin. Điều này có thể cung cấp thêm thông tin về tính phân bố và tính đa dạng của dữ liệu.

5.2.2 Bài toán 2

Hỗ trợ mã hóa và giải mã: Hiện tại, chương trình chỉ triển khai phần mã hóa. Bạn có thể cải thiện chương trình bằng cách thêm chức năng giải mã để người dùng có thể giải mã chuỗi đã được mã hóa bằng hai phương pháp Huffman và Shannon-Fano.

Đánh giá sự hiệu quả của mã hóa: Bên cạnh tính toán hiệu suất mã hóa và dư thừa, bạn có thể mở rộng chương trình để đánh giá các yếu tố khác như tốc độ mã hóa và giải mã, độ tin cậy của mã hóa và khả năng khôi phục lỗi.

Phát triển các phương pháp mã hóa khác: Ngoài mã hóa Huffman và Shannon-Fano, có nhiều phương pháp mã hóa khác nhau như mã hóa LZW, mã hóa Arithmetic, hay mã hóa Golomb. Bạn có thể nghiên cứu và triển khai thêm các phương pháp mã hóa này để so sánh và đánh giá hiệu quả của chúng.