Đồ án về giải thuật nén Huffman

Vu Cong Duy - 19120212
Nguyen Hoang Anh Kiet - 19120266
To Gia Thuan - 19120389

January 2021

Chapter 1

Giới thiệu

1.1 Vấn đề về việc nén dữ liệu

Từ những ngày đầu của kỷ nguyên công nghệ thông tin, việc nén thông tin để tiết kiệm dung lượng lưu trữ đã là một đề tài được đem ra thảo luận rất sôi nổi.

Trong đó, David A. Huffman đã cho ra đời một thuật toán đơn giản, dễ cài đặt và tính hiệu quả cao được đặt tên theo chính ông - thuật toán Huffman.

Thuật toán được đề xuất khi ông còn là sinh viên Ph.D. tại MIT, và công bố năm 1952 trong bài báo "A Method for the Construction of Minimum-Redundancy Codes". Sau này Huffman đã trở thành một giảng viên ở MIT và sau đó ở khoa Khoa học máy tính của Đại học California, Santa Cruz, Trường Kỹ nghệ Baskin (Baskin School of Engineering).

Thuật toán nén Huffman là 1 thuật toán mã hoá thông tin dùng để nén dữ liệu dựa trên việc tối ưu hoá việc mã hoá các kí tự trong chuỗi ban đầu bằng việc xây dựng bộ mã nhị phân đại diện cho từng kí tự trong đó. Thuật toán có mục tiêu sẽ xây dựng được bảng mã nhị phân đại diện cho từng kí tự sao cho những kí tự có tần suất xuất hiện

nhiều sẽ có mã nhị phân đại diện cho nó ngắn và ngược lại.

1.2 Tại sao ta có thể nén được dữ liệu?

Các con chữ (kí tự) chúng ta thường hay sử dụng trong văn bản trên máy tính được hiển thị dựa trên bộ mã kí tự ASCII (gồm 256 kí tự). Vì thế, mỗi kí tự được biểu diễn bởi 8 bit ($2^8 = 256$)

Sở dĩ mỗi kí tự phải đủ 8 bit là để chúng có thể được phân biệt vời 255 kí tự còn lại. Tuy nhiên, trong một văn bản thông tin, hiếm khi nào ta sử dụng hết 256 kí tự. Vì thế, một câu hỏi được đặt ra là : liệu có cần dùng tới 8 bit để có thể phân biệt các kí tự với nhau không khi mà số kí tự thực tế sử dụng là rất ít.

Hầu hết mọi hệ điều hành đọc các ký tự vẫn yêu cầu phải biểu diễn dưới dạng 8 bit. Nhưng nếu xét trên phương diện lưu trữ, việc dùng số bit ít hơn 8 để biểu diễn ký tự vẫn đảm bảo được tính toàn vẹn của dữ liệu (và tất nhiên là tiết kiệm bộ nhớ hơn). Đây cũng chính là cơ sở của thuật toán Huffman

Chapter 2

Cách thực hoạt động

2.1 Ý tưởng chính

2.1.1 Làm rõ việc thay thế bảng mã ASCII để biểu diễn

Để mã hóa các kí hiệu (kí tự, chữ số, ...) ta thay chúng bằng các xâu nhị phân, được gọi là từ mã của kí hiệu đó. Chẳng hạn bộ mã ASCII, mã hóa cho 256 kí hiệu là biểu diễn nhị phân của các số từ 0 đến 255, mỗi từ mã gồm 8 bít. Trong ASCII từ mã của kí tự "a" là 1100001, của kí tự "A" là 1000001. Trong cách mã hóa này các từ mã của tất cả 256 kí hiệu có độ dài bằng nhau (mỗi từ mã 8 bít). Nó được gọi là mã hóa với độ dài không đổi.

Khi mã hóa một tài liệu có thể không sử dụng đến tất cả 256 kí hiệu. Hơn nữa trong tài liệu chữ cái "a" chỉ có thể xuất hiện 1000000 lần còn chữ cái "A" có thể chỉ xuất hiện 2, 3 lần. Như vậy ta có thể không cần dùng đủ 8 bít để mã hóa cho một ký hiệu, hơn nữa độ dài (số bít) dành cho mỗi kí hiệu có thể khác nhau, kí hiệu nào xuất hiện nhiều lần thì nên dùng số bít ít, ký hiệu nào xuất hiện ít thì có thể mã hóa bằng từ mã dài hơn. Như vậy ta có việc mã hóa với độ dài thay

đổi.

2.1.2 Ví dụ

[1]Để làm rõ, ta đi đến ví dụ sau :

Cho một đoạn tập tin có dữ liệu là : ABCDBADA

Ta tạo các dãy bit mới để biểu diễn các ký tự thay cho bảng ASCII. (Cách tạo sẽ được làm rõ ở phần sau)

Ký tự trong bảng mã ASCII	Bit để biểu diễn thông thường	Bit để biểu diễn sau khi nén
A	01000001	11
B	01000010	01
C	01000011	00
D	01000100	10

Figure 2.1: Dãy bit biểu diễn ký tự

Với cách lưu trữ thông thường thì cần: 8 (ký tự) * 8 bit = 64 bit = 8 byte để lưu trữ.

ABCDBADA

01000001 01000010 01000011 01000100 01000010 01000001 01000100 01000001

Thực tế văn bản chỉ có 4 chữ cái A,B,C,D nên chỉ cần 2 bit cho mỗi kí tự, cần: 8 (ký tự) * 2 bit = 16 bit = 2 byte để lưu trữ.

ABCDBADA

11 01 00 10 01 11 10 11

Ò

Trong máy tính chỉ lưu theo byte, tức là những số bit lẻ cần phải được thêm vào để cho đủ 1 byte (8 bit). Ví dụ trên giả định rằng thông

tin cần thiết cho việc giải nén được lưu trữ và xử lý riêng, nên phương pháp nén Huffman giúp giảm đi 75% (2 byte so với 8 byte) dung lượng lưu trữ đối với đoạn văn bản trong ví dụ. Những đoạn mã thu gọn sau khi nén được gọi là mã Huffman. Để tạo ra mã Huffman, cấu trúc dữ liệu cây nhị phân tìm kiếm được áp dụng, được gọi là cây Huffman.

2.2 Chi tiết cách nén

Ở phần này, từng bước sẽ được minh họa dựa vào ví dụ ở phần trước : Nén dãy ký tự **ABCDBADA**

2.2.1 Bước 1: Lập bảng tần số

Đếm số lượng xuất hiện của từng ký tự có trong văn bản, bao gồm cả các ký tự đặc biệt như ký tự xuống dòng (\n), ký tự null (\0) hay ký tự khoảng cách.

Đoạn văn bản ví dụ: ***ABCDBADA***

Ký tự	Tần số xuất hiện
A	3
B	2
C	1
D	2

Figure 2.2: Ví dụ bảng tần số

Cách cài đặt ở bước này tương đối đơn giản : ta tạo một mảng với các phần tử có cấu trúc gồm ký tự, tần số xuất hiện của ký tự đó và

dãy bit để biểu diễn ký tự đó (dữ liệu này sẽ được thiết lập vào các bước sau).

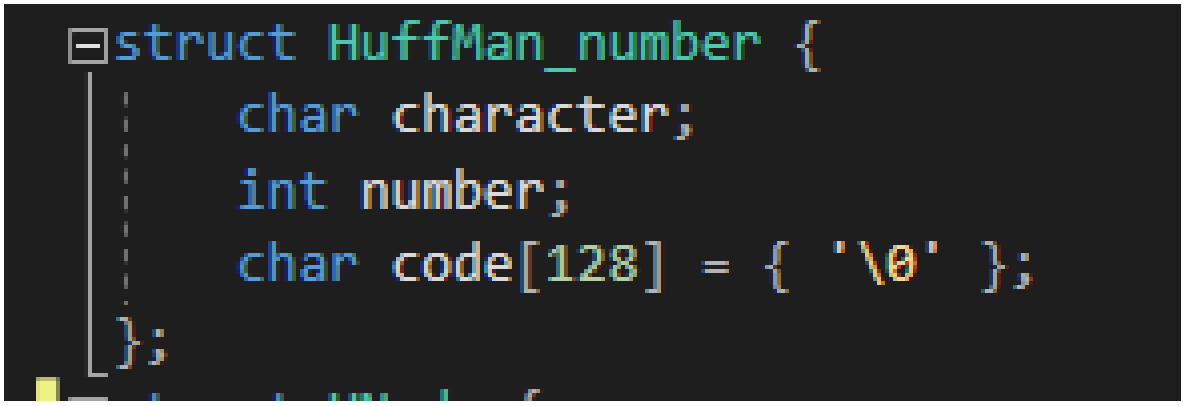

Figure 2.3: Hàm lập bảng tần số

```
void input_array(vector<HuffMan_number>& number, string FileName)
{
    fstream input;
    input.open(FileName, ios::in);
    char ch;
    while (input.get(ch))
    {
        int pos;
        if ((pos = exist(number, ch)) >= 0)
        {
            number[pos].number++;
        }
        else
        {
            HuffMan_number temp;
            temp.character = ch;
            temp.number = 1;
            number.push_back(temp);
        }
    }
    input.close();
}
```

Figure 2.4: Hàm lập bảng tần số

2.2.2 Bước 2: Xây dựng cây Huffman

*Yêu cầu trước khi thực hiện : cài đặt các giải thuật cho hàng đợi ưu tiên của nút dựa với chỉ số sắp xếp là tần số xuất hiện.

```
void insertHNode(vector<HNode*>& array, HNode* x)
{
    int pos = BinarySearchHNode(array, 0, array.size() - 1, x);
    vector<HNode*>::iterator nth = array.begin() + pos;

    array.insert(nth, x);
}
int BinarySearchHNode(vector<HNode*>& a, int left, int right, HNode* key)
{
    if (right<=0)
        return 0;
    if (right <= left)
        return (key->number > a[left]->number) ?
        (left + 1) : left;
    int mid = (left + right) / 2;
    if (key->number == a[mid]->number)
        return mid + 1;
    if (key->number > a[mid]->number)
        return BinarySearchHNode(a, mid + 1, right, key);
    return BinarySearchHNode(a, left, mid - 1, key);
}
```

Figure 2.5: Cài đặt hàng đợi ưu tiên

Bước đầu để có thể tạo dãy bit cho từng ký tự chính là xây dựng cây Huffman dựa vào tần số của từng ký tự. Cây Huffman là một cấu trúc heap với nút cha lớn hơn nút con. Các node bao gồm phần dữ liệu, tần số, nút con trái và nút con phải.

1. Ta tiến hành tạo các nút từ bảng tần số ở bước 1 đã lập. Sau đó đưa chúng vào hàng đợi ưu tiên (xếp từ bé đến lớn).

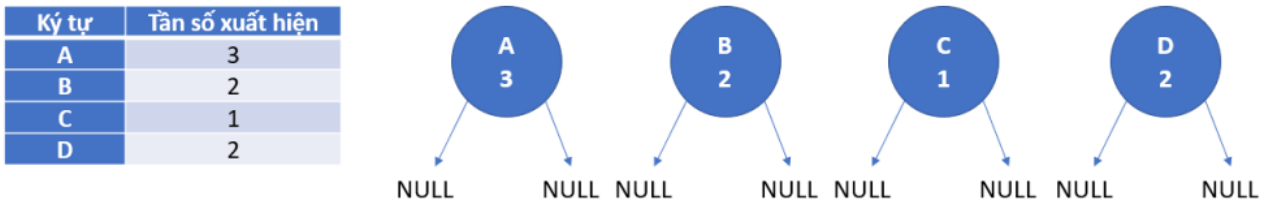

Figure 2.6: Tạo node từ bảng tần số

2. Dequeue 2 node từ hàng đợi ưu tiên (cũng chính là 2 node có tần số bé nhất), tạo một nút cha có tần số là tổng tần số 2 nút con, dữ liệu của nút cha là chuỗi ghép từ chuỗi của 2 node con. Sau đó enqueue nút cha vào hàng đợi ưu tiên (tức cho node cha vào vị trí sao cho hàng đợi gồm các node vẫn được sắp xếp từ bé đến lớn theo tần số)

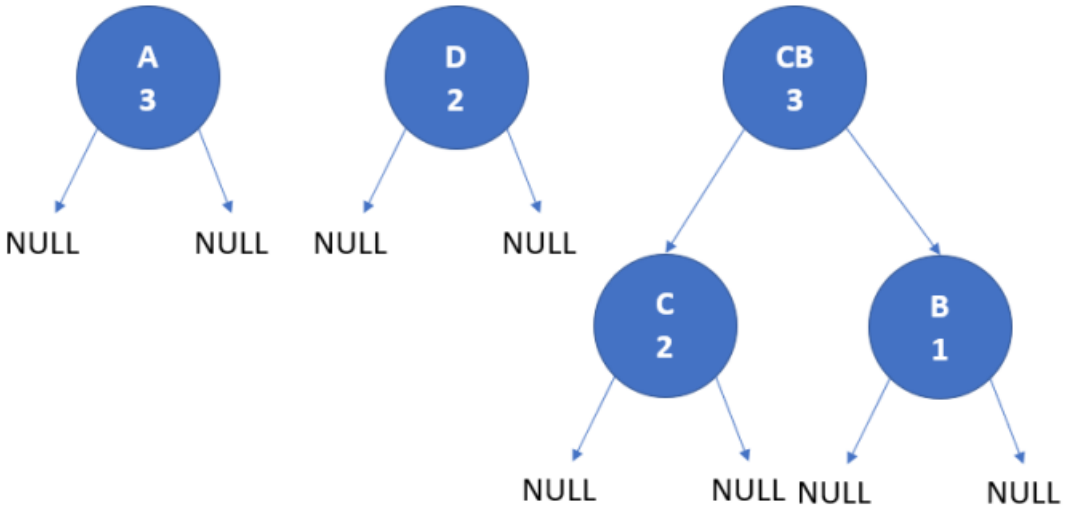

Figure 2.7: Tạo node cha và enqueue vào priority queue

3. Lặp lại bước đầu cho đến khi hàng đợi chỉ còn lại 1 phần tử, đó cũng chính là nút gốc của cây Huffman.

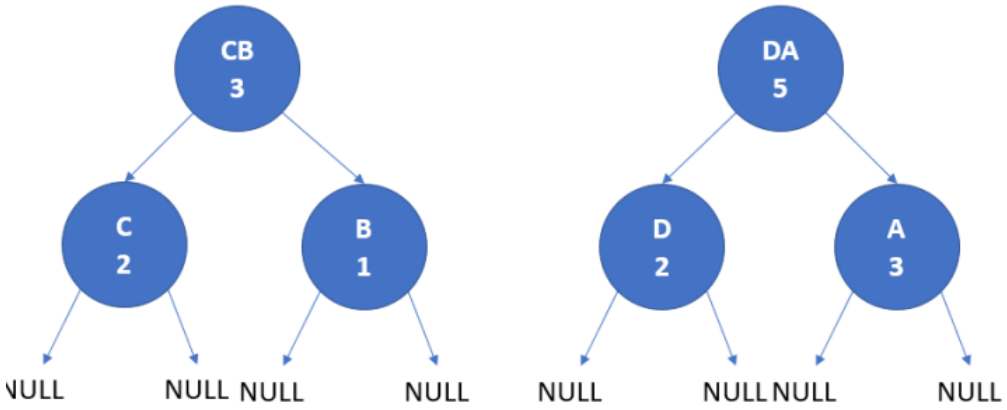

Figure 2.8: Tiếp tục vòng lặp

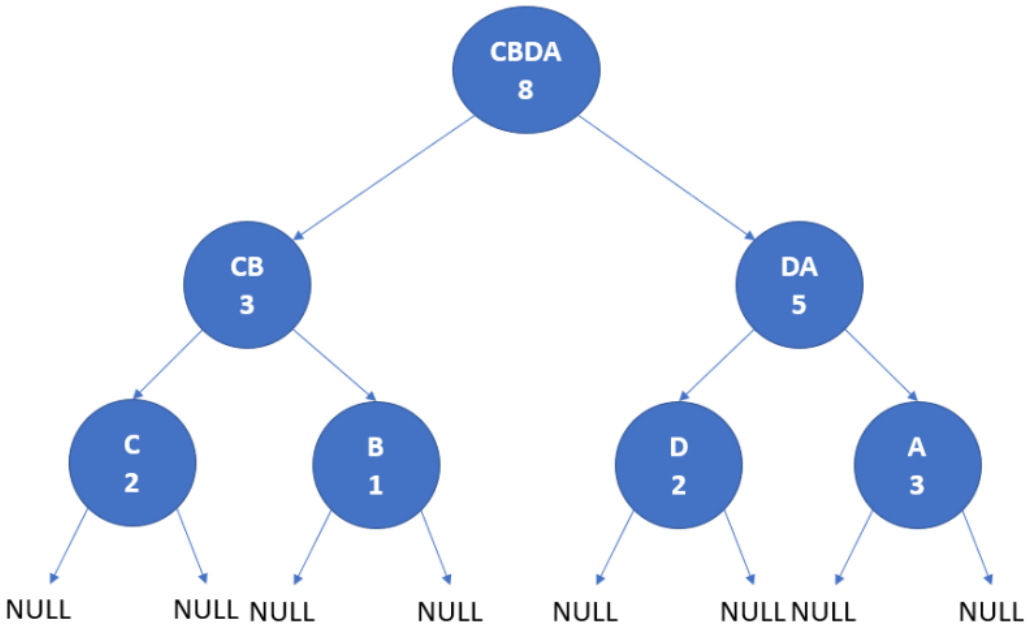

Figure 2.9: Phần tử còn lại duy nhất trong hàng đợi là nút gốc của cây nhị phân Huffman

```
HNode* CreateHNode(vector<HuffMan_number> data_array)
{
    vector<HNode*> node_array(data_array.size());
    for (int i = 0; i < data_array.size(); i++)
    {
        node_array[i] = new HNode();
        node_array[i]->data[0] = data_array[i].character;
        node_array[i]->number = data_array[i].number;
    }

    while (node_array.size() > 1)
    {
        //Create parent node of 2 least appear node
        HNode* a = node_array[0];
        node_array.erase(node_array.begin());
        HNode* b = node_array[0];
        node_array.erase(node_array.begin());
        HNode* parent = new HNode();
        parent->left = a;
        parent->right = b;
        parent->number = a->number + b->number;
        parent_data(parent, a, b);
        insertHNode(node_array, parent);
    //  cout << check(node_array);
    }
    return node_array[0];
}
```


Figure 2.10: Mã nguồn tạo Hnode

2.2.3 Bước 3: Điền dãy cho cho từng ký tự

Dưa trên cây Huffman đã tạo ở bước 2, ta duyệt cây Huffman để tìm ra chuỗi bit của từng ký tự.

Chuỗi bit của ký tự dựa vào đường đi từ nút gốc đến nút lá chứa ký tự đó. Ta tìm kiếm nút lá chứa ký tự bằng cách dò tìm xem ký tự đó nằm trong chuỗi ký tự của nút con bên trái hay bên phải, nếu bên trái thì tiếp tục tiếp kiếm với nút gốc là nút bên trái, bên phải tương tự. Với mỗi lần đi sang trái, ta thêm số 0 vào sau chuỗi, và sang phải là số 1. Lặp lại đến khi nào nút lá có chuỗi chính là kí tự ta cần tìm.

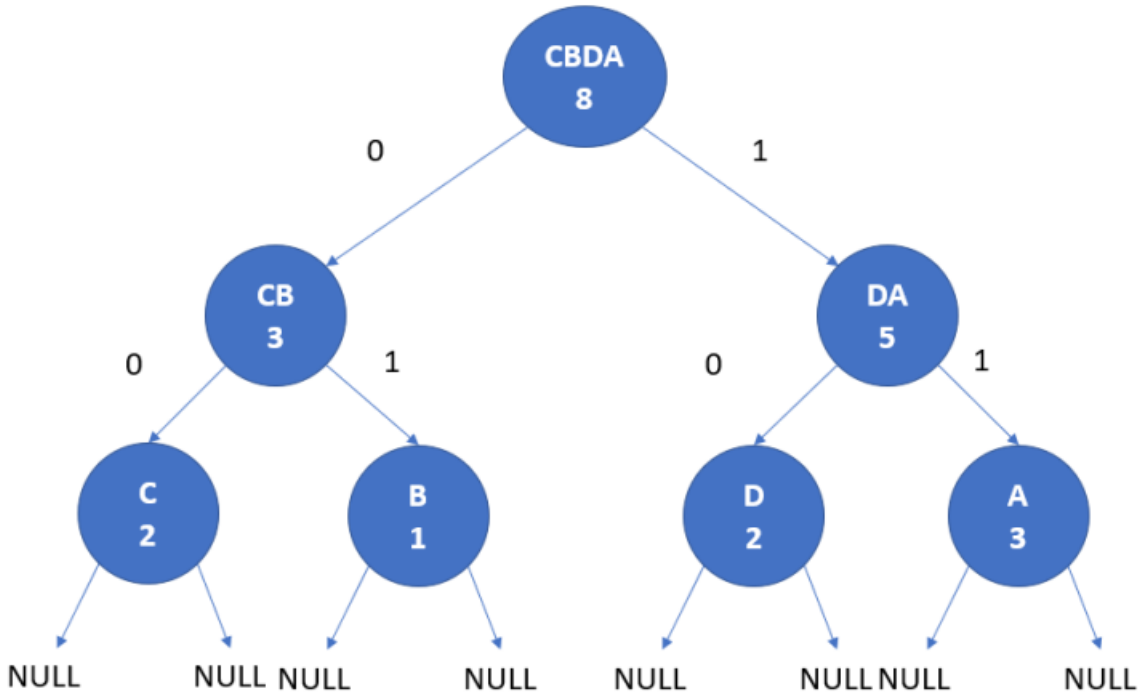

Figure 2.11: Minh hoa đường đi

Ký tự	Mã code Huffman
A	11
B	01
C	00
D	10

Figure 2.12: Chuỗi bit cho các kí tự sau khi thực hiện thuật toán

```
void get_code(vector<HuffMan_number>& array)
{
    HNode* root = CreateHNode(array);
    for (int i = 0; i < array.size(); i++)
    {
        HNode* temp = root;
        while (strlen(temp->data) > 1)
        {
            if (exist(temp->left->data, array[i].character)) {
                temp = temp->left;
                strcat(array[i].code, "0");
            }
            else
            {
                temp = temp->right;
                strcat(array[i].code, "1");
            }
        }
    }
}
```

Figure 2.13: Mã nguồn lấy mã huffman cho tất cả các kí tự

2.2.4 Bước 4: Tạo file nén

File nén sẽ bao gồm 2 phần : bảng tần số và dữ liệu cần nén. Khi giải nén, bảng tần số sẽ được sử dụng để tạo dựng lại cây tìm kiếm Huffman. Bảng tần số : có cấu trúc là <ký tự><ký tự ascii số 16><tần số>.
Trong đó kí tự ascii số 16 dùng để nhận biết giữa <ký tự> và <tần số> trong trường hợp <ký tự> là một chữ số và do kí tự ascii số 16 không có trong nhập liệu.

Dữ liệu : duyệt từng ký tự trong tập tin, lưu mã bit tuần tự sao cho đủ 8 bit thì xuất ký tự đó ra tập tin nén. Do đó cần thực hiện các thao tác trên bit dữ liệu. Một số thao tác trên bit:

```
//position là vị trí 7-i với i là vị trí cần on/off bit
void OnBit(unsigned char& byte, int position)
{
    byte = byte | (1 << position);
}
void OffBit(unsigned char& byte, int position)
{
    byte = byte & ~(1 << position);
}
void OnBit(char& byte, int position)
{
    byte = byte | (1 << position);
}
void OffBit(char& byte, int position)
{
    byte = byte & ~(1 << position);
}
```

Figure 2.14: Hàm thao tác bit

```
void writeHuffManTree(string FileName, vector<HuffMan_number> array)
{
    fstream output;
    output.open(FileName, ios::out | ios::trunc | ios::binary);
    output << array.size() << char(16);
    for (int i = 0; i < array.size(); i++){
        output << array[i].character << char(16) << array[i].number << char(16)
    }
    output.close();
}
```

Figure 2.15: Hàm ghi bảng tần số

```
void write_code_2(string FileName_out, vector<HuffMan_number> array, string File
{
    fstream input;
    fstream output;
    input.open(FileName_in, ios::in);
    output.open(FileName_out, ios::out | ios::app | ios::binary);
    char ch;
    string binary_str="";
    while (!input.eof()){
        int index;
        while (binary_str.length() < 8){
            if (!input.get(ch)){
                for (int i = binary_str.length(); i < 8; i++){
                    binary_str+='0';
                }
                break;
            }
            index = find_code(ch, array);
            binary_str += array[index].code;
        }
        while (binary_str.length() >= 8){
            BinaryStringToChar(binary_str, ch);
            output << ch;
            binary_str.erase(0, 8);
        }
    }
}
```

Figure 2.16: Hàm ghi dữ liệu

2.3 Chi tiết cách giải nén

Việc giải nén đơn giản hơn rất nhiều vì đã có sẵn các hàm từ việc nén và ta chỉ việc làm ngược lại khi nén. 1. Đọc bảng tần số 2. Xây dựng

cây huffman từ bảng tần số (hàm có sẵn) 3. Phân mảnh kí tự trong phần dữ liệu ra thành các dãy bit, rồi dựa vào các dãy bit để dò tìm đến nốt lá chứa kí tự.

```
void DeCode_HuffManFile(string inputFile, string outputFile)
{
    fstream input;
    fstream output;
    output.open(outputFile, ios::trunc | ios::out | ios::binary);
    input.open(inputFile, ios::in | ios::binary);
    int size = output.tellg();
    vector<HuffMan_number> array;
    int n;
    long number_of_char = 0;
    input >> n;
    input.ignore();
    array.resize(n);
    for (int i = 0; i < n; i++)
    {
        input.get(array[i].character);
        input.ignore();
        input >> array[i].number;
        input.ignore();
        number_of_char += array[i].number;
    }
    HNode* root = CreateHNode(array);
```

Figure 2.17: Đọc bảng tần số

```
HNode* root = CreateHNode(array);
HNode* temp = root;
for (int i = 0; (number_of_char > 0);)
{
    char ch;
    input.get(ch);
    bitset<8> bin(ch);
    for (int j = 0; j < 8;)
    {
        while (!(temp->left == NULL && temp->right == NULL))
        {

            if (bin[7 - j] == 0) {
                temp = temp->left;
            }
            else {
                temp = temp->right;
            }
            j++;
            if (j >= 8)
                break;
        }
        if (j <= 7) // chua doc het day bit cua ch -> da doc duoc chu can output -> xuat ra file
        {
            output << temp->data;
            temp = root;
            number_of_char--;
            if (number_of_char == 0)
                break;
        }
    }
}
```

Figure 2.18: Phân giải từ dữ liệu file nén

2.4 Văn bản có kí tự ngoài bảng ASCII 8-bit

Câu hỏi được đặt ra rằng liệu thuật toán Huffman có thể dùng cho các ngôn ngữ có những kí tự nằm bên ngoài bảng ASCII 8-bit như Tiếng Việt, Trung, Hàn,... không? Để giải đáp cho câu hỏi này, ta phải hiểu các bảng mã và quy tắc biểu diễn của các ngôn ngữ đó.

Đa số các ký tự đặc biệt hiện nay (và kể cả kí tự nằm trong bảng ASCII 8-bit) đều được biểu diễn bằng bảng mã UTF-8. Bảng mã UTF-8 biểu diễn các ký tự đặc biệt bằng dãy các byte (8-bit). Những byte này có thể biểu diễn các kí tự ở bảng mã ASCII.

Figure 2.19: Các

Vì vậy, khi thực hiện thuật toán ta có thể xem các byte kí tự ấy là các byte kí tự riêng lẻ, mỗi byte kí tự riêng lẻ ấy đều được gán cho mã huffman khác nhau. Khi giải nén, các byte kí tự riêng lẻ ấy viết kề nhau thì chương trình sử dụng bảng mã UTF-8 sẽ tự nhận biết và biểu diễn kí tự đặc biệt ấy.

Chapter 3

Tổng kết

3.1 Độ phức tạp thuật toán

Ta phân tích độ phực tạp dựa vào số kí tự cần nén. Gọi số kí tự cần nén là n. Số phép toàn cần thực hiện là $T(n)$

3.1.1 Thuật toán nén

Ở thuật toán nén, bước đầu tiên là phải duyệt file để xây dựng bảng tần số, ta cho mỗi lặp cần 1 phép toán để đọc vào bộ nhớ, số phép toán thực hiện :

$$n$$

Các bước để xây dựng mã Huffman cho từng kí tự được cho là rất nhỏ bởi chỉ có thể có tối đa chưa đến 256 kí tự, vì vậy số phép toán là hữu hạn. Khi số kí tự cần nén n lớn thì số phép toán hựu hạn ấy lại không đáng kể.

Tạo file nén cũng cần duyệt hết file để xây dựng file nén, số phép toán thực hiện :

$$n$$

Vậy số phép toán của giải thuật Huffman là :

$$T(n) \approx 2n = O(n)$$

3.1.2 Thuật toán giải nén

Gọi tỉ số nén là $\frac{1}{a}$ với $a \in N$. Ở lần duyệt file nén, ta có số lần lặp là :

$$\frac{1}{a}.n$$

Các bước xây dựng mã Huffman như đã đề cập ở phần trên : không đáng kể.

Gọi kỳ vọng số lần đi sáng trái/phải trong cây nhị phân Huffman để tìm được kí tự cần ghi là k với $k \in N$. Ở bước giải nén, số lần lặp là :

$$k.n$$

Vậy số phép toán thực hiện ở bước giải nén là :

$$T(n) \approx k.n + \frac{1}{a}.n = O(n)$$

3.2 Độ hiệu quả thuật toán

Để đánh giá độ hiệu quả thuật toán, ta dựa vào hệ số nén là thương của độ lớn file sau khi nén và trước khi nén.

Với văn bản của các ngôn ngữ mà các chữ các có trong bảng mã ASCII 8 bit như Tiếng Anh, Pháp, Đức, Ý, Tây Ban Nha,... thì tỉ số nén thấp, xấp xỉ 50%

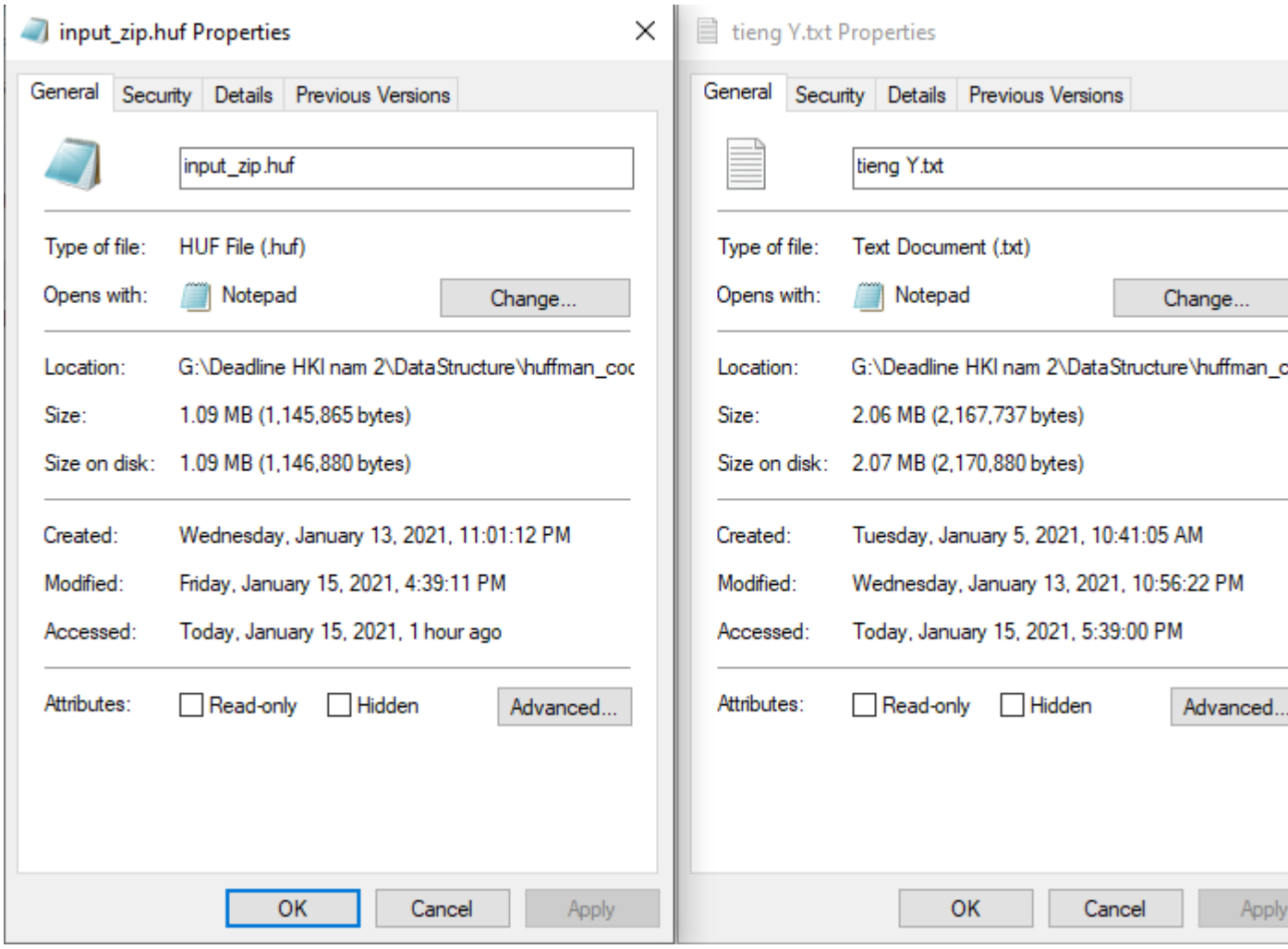

Figure 3.1: Tỉ số nén của văn bản Tiếng Ý : 52.9%

Những văn bản mà các chứ không nằm trong bảng ASCII như Tiếng Việt, Trung, Hàn, Thái,... thì tỉ số nén cao, dao động trong khoảng 70%

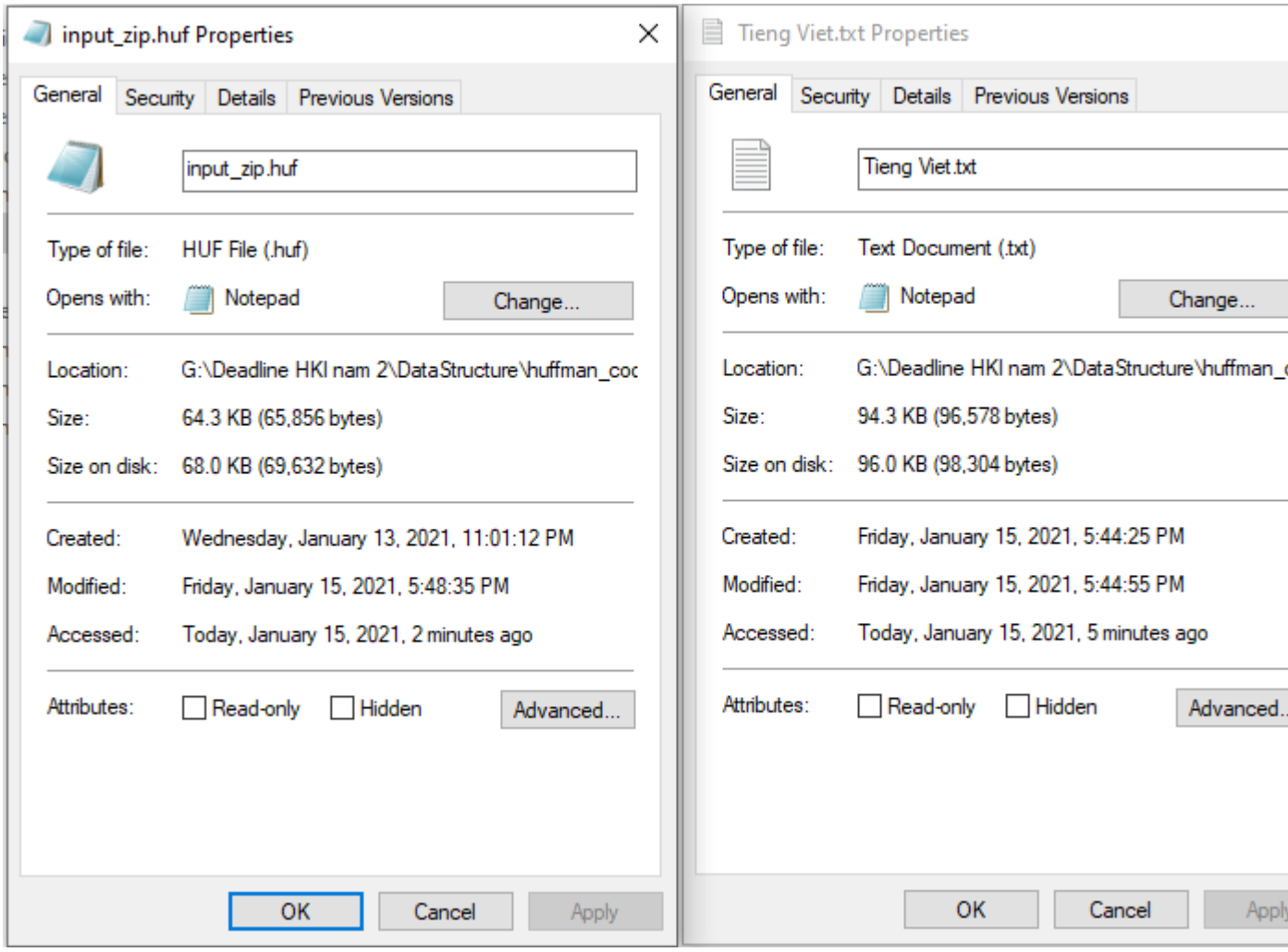

Figure 3.2: Tỉ số nén của văn bản Tiếng Việt : 68.1%

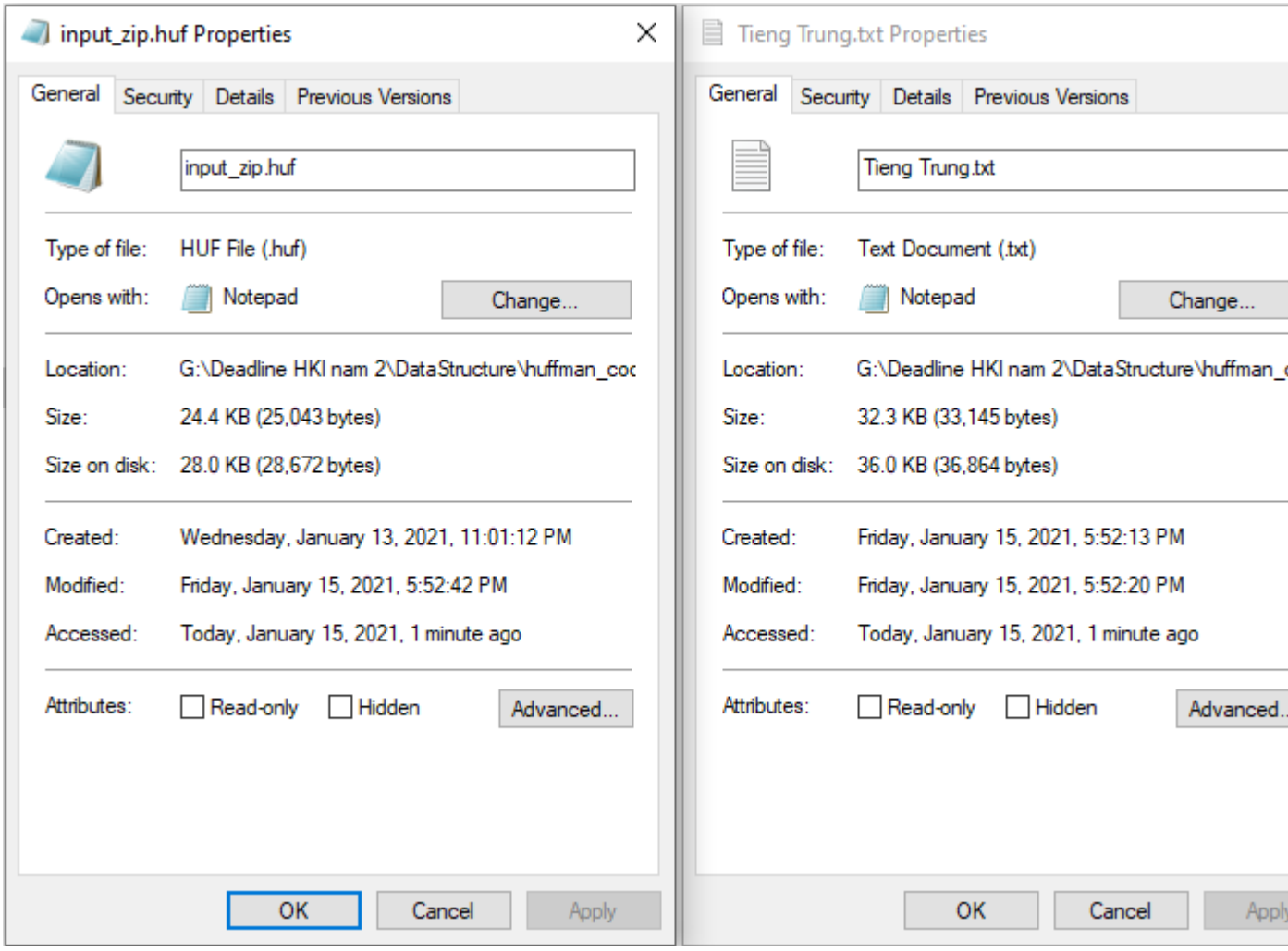

Figure 3.3: Tỉ số nén của văn bản Tiếng Trung : 74%

3.3 Ứng dụng

Tuy giải thuật nén Huffman đã ra đời khá lâu nhưng vẫn có những ứng dụng quan trọng hiện nay.

Tất cả những thuật toán nén như zip cũng lấy nền tảng là giải

thuật Huffman làm gốc, có thể coi Huffman-code là "back-end" của các thuật toán nén hiện nay.

Nói rộng hơn, tất cả những kiểu lưu trữ dữ liệu mà có sự lặp lại nhiều lần của các phần tử chứa thông tin đều có thể sử dụng giải thuật Huffman để tiết kiệm dữ liệu:

- File văn bản : là file đơn giản nhất, các phần tử chứa thông tin ở đây là các byte kí tự. Đây cũng chính là thuật toán được cài đặt ở trên.

- File hình ảnh : phần tử chứa thông tin ở đây là pixel. Trong thực tế, trong một file ảnh sẽ có nhiều pixel có thông tin trùng nhau (xác suất trùng nhau có thể gia tăng bằng cách đồng hóa những pixel có thông tin gần giống nhau bằng thuật toán). Khi đó, ta sẽ xem các pixel là những phần tử, sau đó lập bảng tần số , lập cây Huffman,... và nén các pixel của ảnh. Từ đó tiết kiệm được rất nhiều dung lượng. Tuy điều này sẽ làm giảm chất lượng ảnh vì đồng hóa các pixel nhưng vẫn chấp nhận được.

- File video : những phần tử chứa thông tin là các frame hình ảnh. Có thể phân hóa từng frame hình ảnh bằng cách thuật toán phức tạp.

Có thể phân những thuật toán nói trên làm 2 nhóm:[4]

3.3.1 Nén lossless

Đây là kiểu nén không mất dữ liệu, có những đặc điểm sau : - Dữ liệu có thể được giải nén ra với tình trạng giống với từng bit của dữ liệu trước khi nén. - Được dùng trong nén file (.zip). - Hệ số nén trung bình (từ 2:1 đến 3:1). - Được sử dụng trong những ứng dụng đặc biệt quan trọng như ảnh y học, lưu trữ thông tin hộp đen,... - File format sử dụng nén lossless có thể kể đến là PNG (hình ảnh), MP3 (âm thanh), MPEG-4/HD-ACC (video).

3.3.2 Nén lossy

Còn được gọi là nén mất mát dữ liệu, có thể dùng những thuật toán để làm xuất hiện nhiều phần tử giống nhau : - Dữ liệu giải nén ra có thể không giống từng bit với dữ liệu trước khi nén. - Hệ số nén tương đối cao (từ 20:1 - 40:1). - File format sử dụng nén lossless có thể kể đến là JPEG(hình ảnh), MPEG(video),... Trong thực tế, file video, hình ảnh dân dụng sẽ sử dụng nén lossy vì sự khác biệt khi mất mát dữ liệu rất khó phân biệt bằng mắt thường.

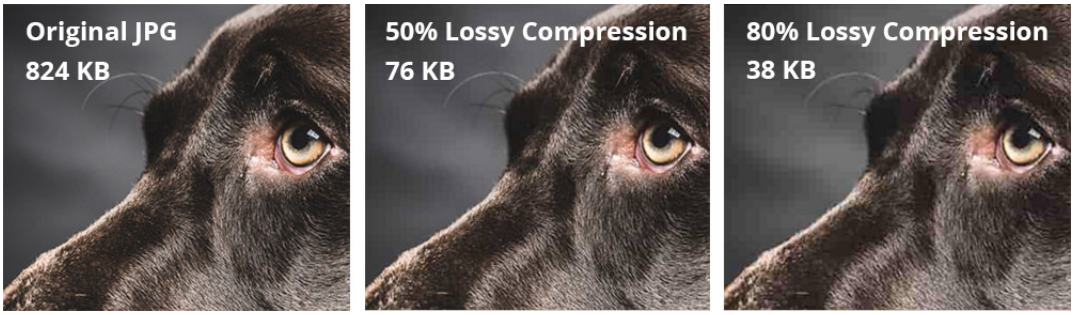

Figure 3.4: Sự khác biệt không đáng kể giữa ảnh gốc/lossless và lossy

Bibliography

[1] Đại học Khoa học tự nhiên TPHCM . *Tài liệu hướng dẫn thực hành học phần CS104 – Cấu trúc dữ liệu - ITEC*. 2020.

[2] Thomas H.Cormen .*Introduction to Algorithms*. 2009.

[3] https://vi.wikipedia.org/wiki/M%C3%A3_h%C3%B3a_Huffman.

[4] Kumar, Vikas. (2015). *Compression Techniques Vs Huffman Coding. International Journal of Informatics and Communication Technology (IJ-ICT)*. 4. 29. 10.11591/ijict.v4i1.pp29-37.