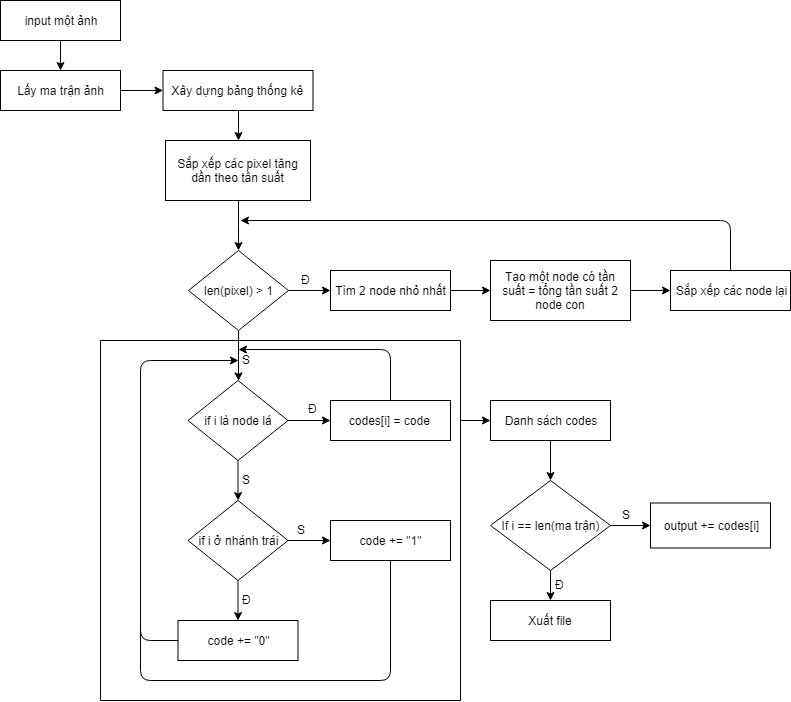
1. Thuật toán mã hóa Huffman:
   1. Giới thiệu:

* Trong khoa học máy tính và lý thuyết thông tin, mã hóa Huffman là một thuật toán mã hóa dùng để nén dữ liệu. Nó dựa trên bảng tần suất xuất hiện của các ký tự cần mã hóa để xây dựng một bộ mã nhị phân cho các ký tự đó sao cho dung lượng (số bit sau khi mã hóa là nhỏ nhất).
* Thuật toán được đề xuất bởi David A. Huffman 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”.
  1. Thuật toán
     1. Quá trình nén dữ liệu

Input: là một hình ảnh cần được nén

Output: Cây mã huffman

* Bước 1: Xây dựng bảng thống kế tần suất xuất hiện của các pixel
* Bước 2: Sắp xếp các pixel theo tần suất tăng dần
* Bước 3: Tìm 2 node pixel có tần suất xuất hiện thấp nhất và ghép lại thành 1 node với tần suất bằng tổng tần suất của 2 node pixel này
* Bước 4: Thêm node vừa tạo ở bước 3 vào danh sách có node.
* Bước 5: Gán bit 0 cho nhánh bên trái, nhánh bên phải gán bit 1.
* Bước 6: Tổng hợp tất cả các node lại với nhau để được tạo một cây mã nhị phân.

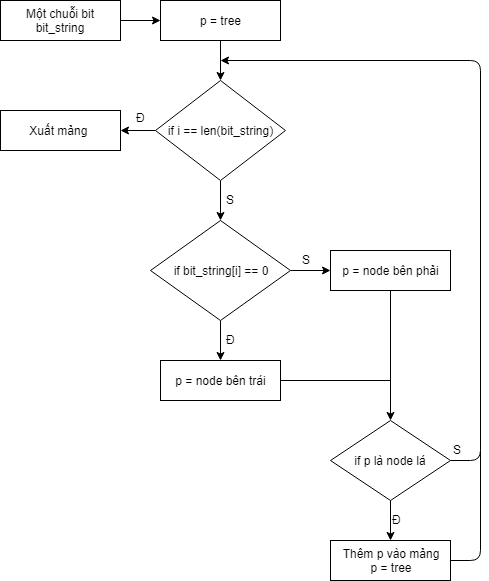


* + 1. Quá trình giải nén dữ liệu

Input: là một file binary của ảnh cầ được giải nén

Output: là một ảnh

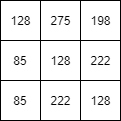
Truy vấn lại cây mã nhị phân vừa tạo ra ở quá trình nén dữ liệu, nếu bit là 0 thì truy hồi lại nhánh bên trái, nếu bit là 1 thì truy hồi lại nhánh bên phải. Quá trình được lặp lại cho đến node cuối.



* 1. Ưu điểm và nhược điểm của thuật toán mã hóa huffman
     1. Ưu điểm:
* Tối ưu cực tiểu hóa mã code
* Quá trình nén và giải nén tương đối đơn giản
* Pixel xuất hiện nhiều hơn thì mã code ngắn hơn
  + 1. Nhược điểm:
* Tốn nhiều thời gian xây dựng cây mã
* Cây huffman của mã code và bộ mã code dùng để mã hóa phải được lưu lại để mã hóa. Điều này làm giảm hiệu suất nén

1. Ví dụ:

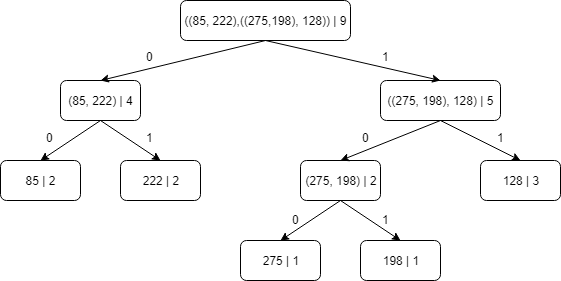
Đây là ma trận pixel 3x3:



Bảng tần suất xuất hiện của các pixel đã được sắp xếp:

|  |  |
| --- | --- |
| Pixel | Tần suất xuất hiện |
| 275 | 1 |
| 198 | 1 |
| 85 | 2 |
| 222 | 2 |
| 128 | 3 |

Cây mã huffman:



Bảng mã cây huffman:

|  |  |
| --- | --- |
| Pixel | Code |
| 275 | 100 |
| 198 | 101 |
| 85 | 00 |
| 222 | 01 |
| 128 | 11 |

1. Cài đặt:
   1. Cấu trúc chương trình

get\_matrix\_image (): là hàm lấy ma trận pixel của hình ảnh

cal (): là hàm tính tần suất xuất hiện của các pixel

sortFreq (): hàm sắp xếp các pixel theo tần suất tăng dần

buildTree (): hàm ghép 2 node có tần suất xuất hiện thấp nhất và tạo một node cha của 2 node này thêm vào mảng node (node có tần suất xuất hiện và giá trị pixel).

Tree (): hàm lấy giá trị pixcel của mảng node (loại bỏ tần suất xuất hiện của các node) tạo thành cây hoàn chỉnh

assignCodes (): hàm tìm mã code của từng pixel

encode () và read (): hàm thực hiện lưu cây mã huffman dưới dạng file binary

decode (): giải nén cây mã huffman

* 1. Một số hàm được xây dựng trong chương trình

**def** buildTree**(**vector**):**

**while** len**(**vector**)** **>** 1**:**

#lấy 2 node có tần suất nhỏ nhất

lowestTwo **=** tuple**(**vector**[**0**:**2**])**

#lưu các node còn lại

theRest **=** vector**[**2**:]**

#kết hợp 2 node có tần suất nhỏ nhất

sumPro **=** lowestTwo**[**0**][**0**]** **+** lowestTwo**[**1**][**0**]**

#lưu node vừa tạo vào mảng

vector **=** theRest **+** **[(**sumPro**,** lowestTwo**)]**

sorted**(**vector**,** key **=** getKey**)**

**return** vector**[**0**]**

**def** Tree**(**tree**):**

a **=** 3

a **=** np**.**dtype**(**'uint8'**).**type**(**a**)**

p **=** tree**[**1**]**

**if** type**(**p**)** **==** type**(**a**):**

**return** p

**else:**

**return** **(**Tree**(**p**[**0**]),** Tree**(**p**[**1**]))**

code**=** **{}**

**def** assignCodes**(**n**,** pat **=** ''**):**

a **=** 3

a **=** np**.**dtype**(**'uint8'**).**type**(**a**)**

**if** type**(**n**)** **==** type**(**a**):**

code**[**n**]** **=** pat

**else:**

#node bên trái thêm bit 0

assignCodes**(**n**[**0**],** pat**+**"0"**)**

#node bên phải thêm bit 1

assignCodes**(**n**[**1**],** pat**+**"1"**)**

**def** decode**(**tree**,** str**,** path**):**

a **=** 3

high **=** getHigh**(**path**[**0**:**len**(**path**)-** 4**])**

weight **=** getWeight**(**path**[**0**:**len**(**path**)** **-** 4**])**

#output là ma trận 3 chiều để tiện việc convert lại thành ảnh

output **=** np**.**zeros**((**high**,**weight**,**3**))**

output **=** np**.**uint8**(**output**)**

k **=**0

j **=** 0

n **=** 0

p **=** tree

**for** i **in** str**:**

**if** j **==** weight**:**

j **=** 0

k**+=**1

**if** n **==** 3**:**

j**+=**1

n **=** 0

**if** i **==** '0'**:**

#bit 0 thì đi qua trái của cây huffman

p **=** p**[**0**]**

**else:**

#bit 1 thì đi qua phải cây huffman

p **=** p**[**1**]**

**if** type**(**p**)** **==** type**(**a**):**

#node lá thì thêm phần tử đó vào output

p **=** np**.**dtype**(**'uint8'**).**type**(**p**)**

output**[**k**][**j**][**n**]** **=** p

n**+=**1

p **=** tree

**return** output

1. Nhận xét:

* Thuật toán thực hiện theo nguyên tắc “bớt hai, thêm một”, bớt đi 2 node có tần suất xuất hiện thấp thêm vào một node có tần suất bằng tổng tần suất của hai node vừa loại bỏ, cho đến khi chỉ còn một node duy nhất.
* Các pixel có tần suất xuất hiện càng nhiều thì càng gần node gốc và ngược lại. Điều này, làm cho các pixel có tần suất xuất hiện nhiều sẽ có mã code ít bit hơn
* Nhận vào một chuỗi bit có thể giải mã ngay lập tức.
* Chỉ có một con đường duy nhất đi từ node gốc đến một node lá
* Cây huffman được xây dựng theo thứ tự trái qua phải và từ dưới lên trên thì các node sẽ tuân theo thứ tự tăng dần của tần suất xuất hiện.

1. Các vấn đề gặp phải khi thực hiện và cách giải quyết
   1. Các vấn đề gặp phải

* Vì xây dựng cây huffman bằng tuple của python nên khi sắp xếp xảy ra việc không biết cách sắp xếp theo tần suất xuất hiện của các pixel.
* Xây dựng chương trình chưa thuần hướng đối tượng nên gặp một số vấn đề khi nén. Như lưu mã code vào mảng code thực hiện ở hàm assignCodes (), nó phải thực hiện đệ quy cho đến node lá của cây huffman mới thêm mã code vào mảng code, việc này chỉ dùng khi nó chạy hết các node lá nên không khó trả về mảng code ngay ghi ở node lá được.
* Việc nén các pixel ảnh do các phần tử quá nhiều thực hiện hiện phép cộng chuỗi bị lâu.
* Ban đầu thực hiện chuyển ảnh sang ma trận 2 chiều nên khi mã hóa khó chuyển sang hình ảnh.
* Xảy ra các vấn đề về kiểu dữ liệu khi so sánh type ().
* Thực hiện việc lưu mã code vào file text dẫn đến kích thước file nén cao hơn ảnh gốc.
  1. Cách giải quyết:
* Viết thêm hàm getKey () để khi dùng hàm sorted để sắp xếp các tuple theo cấu trúc: sorted (tuple, key = getKey).
* Mang mảng code thành biến toàn cục cho cả chương trình.
* Thực hiện việc lưu chuỗi code vào file thay vì thực hiện phép cộng.
* Thực hiện việc chuyển ảnh sang ma trận 3 chiều thay vì sang ma trận 2 chiều.
* Ép kiểu cho cùng kiểu dữ liệu.
* Thực hiện việc lưu mã code dưới dạng file binary.

1. Kết quả đạt được sau khi cài đặt:

* Cài đặt thành công thuật toán mã hóa huffman
* Giảm được 2 % so với ảnh gốc



* Nén thành công các loại dữ liệu như văn bản, hình ảnh
* Kỹ năng làm việc nhóm, phân chia các công việc và giúp đỡ nhau trong giải quyết khó khăn.
* Tìm hiểu và nghiên cứu các thư viện trong python hỗ trợ việc thực hiện cài đặt.

1. Hướng phát triển:

* Tiếp tục nâng cấp để thuật toán thực hiện nhanh hơn.
* Thiết kế lại theo thuần hướng đối tượng để dàng sử dụng và thực tế hơn.
* Phát triển lên các thuật toán khác như jpeg 2000.