BÁO CÁO ĐỒ ÁN MÔN HỌC
(Đồ án tìm hiểu CTDL/Giải thuật)
Lớp: IT003.O21.CNTT

SINH VIÊN THỰC HIỆN

Mã sinh viên: 23521421 **Họ và tên: Đồng Quốc Thắng**

TÊN ĐỀ TÀI: Huffman Coding

Link Github: https://github.com/LowTechTurtle/IT003/tree/main/HuffmanCoding

I. Giới thiệu đồ án:

Đồ án của em phân tích về thuật toán Huffman Coding. Huffman Coding là một thuật toán map các kí tự với một dãy nhị phân có chiều dài không xác định trước(khác với Ascii, Unicode), từ đó có thể lưu trữ được tiết kiệm hơn bằng cách lưu các kí tự thường được sử dụng với dãy nhị phân có chiều dài ngắn.

Ngoài ra em còn viết một chương trình minh họa áp dụng thuật toán Huffman coding để nén các file text.

II. Quá trình thực hiện:

a. Tuần 1:

- Đọc và tìm hiểu về thuật toán Huffman trên các trang web thông dụng[1][2][3]
- Viết các phần sau trong file nghiên cứu:

Fixed-Length Binary Codes

Prefix-Free Codes[4]

Biểu diễn Codes dưới dạng cây[5][6]

Tính đúng đắn của thuật toán[6][7][8]

b. Tuần 2:

- Viết mã giả của Huffman coding
- Tham khảo các cách cài Huffman coding bằng C++[2]

- Tìm hiểu về các cách tối ưu thuật toán[9][10]
- Thêm 1 vài ví dụ minh họa vào báo cáo

III. Kết quả đạt được

1. *Các định nghĩa/khái niệm cơ bản*

Huffman coding: Là một kỹ thuật sử dụng để nén dữ liệu bằng cách map các kí tự khác nhau với những biểu diễn nhị phân khác nhau để tìm cách giảm bớt chi phí lưu trữ

Frequency/tần số xuất hiện: Đôi khi có thể sử dụng "xác suất xuất hiện", ý để chỉ rằng mức độ xuất hiện của một kí tự trong một văn bản đang cần nén bằng cách sử dụng Huffman coding

Fixed-Length Binary Codes: Là cách biểu diễn các kí tự trong máy tính sao cho mỗi kí tự được biểu diễn bằng một dãy nhị phân có độ dài không đổi(một số cách biểu diễn thông dụng như là các cách biểu diễn của Asciii, Unicode)

Variable-Length Codes: Ngược lại với Fixed-Binary Codes, Variable-Length Codes là cách biểu diễn các kí tự khác nhau bằng những dãy nhị phân có thể có chiều dài khác nhau(có thể có chiều dài giống nhau)

Prefix-Free Codes: Prefix-Free Codes là một dạng Variable-Length Codes nhưng đảm bảo rằng với mọi cặp kí tự có xuất hiện trong text, kí tự này sẽ không là tiền tố của kí tự khác và ngược lại.

2. *Các đặc trưng nổi bậc*

2.1 Lưu trữ tiết kiệm hơn bằng Prefix-Free Codes

Bảng dưới là một cách biểu diễn Prefix-Free Codes cho bảng chữ cái α = {A, B, C, D}

Chữ cái	Biểu diễn
A	0
B	10
C	110
D	111

Ta biểu diễn chữ cái "A" bằng số 0, nên các chữ cái khác không được bắt đầu bằng số 0. Tương tự ta biểu diễn "B" bằng 10 nên các chữ cái "C" và "D" phải bắt đầu bằng "11".

Cách biểu diễn trên chỉ sử dụng 1 bit cho chữ cái A, tức ít hơn 1 bit so với cách biểu diễn thông thường nhưng lại sử dụng 3 bit cho C và D, tăng 1 bit so với cách biểu diễn thông thường, vì vậy để biết được cách biểu diễn này có tiết kiệm hơn hay không, ta phải xét đến xác suất một chữ cái xuất hiện trong 1 văn bản cần nén.

Giả sử một văn bản sử dụng 4 chữ cái trên với xác suất được phân bố như bảng dưới:

Chữ cái	Xác suất
A	50%
B	30%
C	10%
D	10%

Với Fixed-Length Binary Codes, ta sử dụng 2 bit để biểu diễn mỗi chữ cái trong bảng chữ α trên, để so sánh cách biểu diễn nào là tiết kiệm hơn, ta tính số bit trung bình được sử dụng trong cách Variable-Length Binary Codes.

Gọi E là số bit trung bình cho cách biểu diễn Variable-Length Binary Codes thì ta có:

E = 1 x 0.5 + 2 x 0.3 + 3 x 0.1 + 3 x 0.1 = 1.7

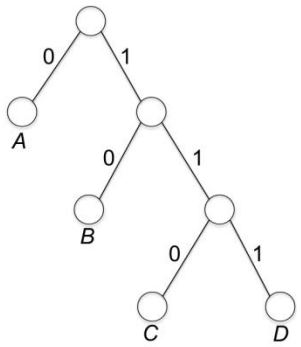
Ta có thể thấy ở trường hợp sử dụng Variable-Length Binary Codes, nếu ta chọn biểu diễn các chữ cái xuất hiện với tần số cao bằng số bit nhỏ hơn thì ta có thể đạt được số bit trung bình trên mỗi chữ cái trong văn bản nhỏ hơn, từ đó ta có thể lưu trữ tiết kiệm hơn.

2.2 <u>Sử dụng cây để biểu diễn Prefix-Free codes</u>

Ví dụ: Với cách biểu diễn Prefix-Free theo bảng dưới

Chữ cái	Biểu diễn
A	0
B	10
C	110
D	111

Thì ta có cây nhị phân tương ứng là



Một chi tiết có thể thấy là độ cao của mỗi nút lá thuộc cây đó sẽ bằng với số bit được sử dụng để biểu diễn chữ cái thuộc nút đó. Lý do đơn giản là vì với mỗi bit được sử dụng thêm thì ta phải đi 1 bước đến nút con bên trái nếu bit đó là 0 hoặc đi 1 bước đến nút con bên phải nếu bit đó là 1. Vậy với một dãy n bit thì ta phải đi đến n nút con, hay ta ở chiều cao n của cây đó.

Ta có thể thấy một cây biểu diễn prefix-free codes chỉ lưu trữ các kí tự ở các nút lá

(vì nếu lưu trữ ở 1 nút x không phải lá thì các nút lá có tổ tiên là x sẽ có tiền tố của

cách biểu diễn là cách biểu diễn của x)

Cách decode một dãy bit theo cây này như sau: đi từ gốc theo các nút, nếu gặp số 0 thì đi về nút bên trái, nếu gặp số 1 thì đi theo nút bên phải, đi cho đến khi gặp nút lá thì dãy bit vừa được đi qua đó sẽ tương đương với kí tự được lưu trữ ở nút lá .

Ví dụ: dãy "010111" sẽ được đi từ gốc tới lá 3 lần, biểu diễn 3 chữ cái "A", "B" và

"D"

2.3 Lossless compression

Thuật toán Huffman coding có thể cung cấp dịch vụ lossless compression: tức sau khi nén và giải nén, thì dữ liệu ban đầu và dữ liệu sau khi giải nén hoàn toàn giống nhau.

Điều này là có thể bởi vì Huffman Codes map từng kí tự với cách biểu diễn nhị phân mới, sau khi có được cách biểu diễn nhị phân mới thì sẽ lưu file ở dạng biểu diễn nhị phân này. Khi cần giải nén, Huffman Codes chỉ cần tìm lại cách biểu diễn nhị phân đó, đi theo các bit và giải nén ra thành các kí tự ban đầu. Vì vậy sẽ không có mất mát về dữ liệu sau khi dùng Huffman coding để nén các file text.

2.4 Huffman coding là thuật toán tham lam

2.4.1 Trình bày thuật toán

Với 1 set các cây trong rừng, chúng ta nên chọn 2 cây nào để ghép lại thành 1 cây trong 1 vòng lặp bất kì?

Để trả lời câu hỏi trên, ta xét 4 cây(chỉ có gốc) có chứa 4 kí tự A, B, C, D, ta gọi tên của 4 cây ban đầu đó lần lượt là A, B, C, D. Giả sử khi ta ghép 2 nút C và D lại với nhau, chiều cao của 2 nút C và D tăng từ 0 lên 1, sau đó giả sử ta ghép B và nút cha của C và D lại với nhau, chiều cao của B tăng lên 1 và chiều cao của C và D tăng lên 1. Sau đó ta lại ghép nút A và nút cha của B, làm cho chiều cao của nút A từ 0 tăng lên 1, chiều cao của B từ 1 tăng lên 2, và chiều cao của nút C và D tăng từ 2 lên 3.

Vì vậy nên, mỗi lần ghép 2 cây lại với nhau(ta xem 1 nút riêng biệt là 1 cây chỉ có gốc, có chiều cao là 0) thì ta lại tăng chiều cao của mọi nút lá thuộc 2 cây đó lên 1, tức làm tăng số bit biểu diễn của các kí tự có trong các nút lá của 2 cây đó lên 1.

Vì vậy nên mỗi khi ta ghép 2 cây lại với nhau, số bit trung bình để biểu diễn các kí tự tăng lên 1 khoảng:

$$\sum_{a \in T_1} p_\alpha + \sum_{a \in T_2} p_\alpha$$

Với T_1, T_2 lần lượt là 2 cây được ghép với nhau/

Tức là tổng của các xác suất xuất hiện của các kí tự có trong các nút lá của 2 cây.

Vì vậy nên thuật toán tham lam Huffman sẽ ghép 2 cây sao cho tổng xác suất xuất hiện trong văn bản của các kí tự có trong các nút lá của 2 cây là nhỏ nhất.

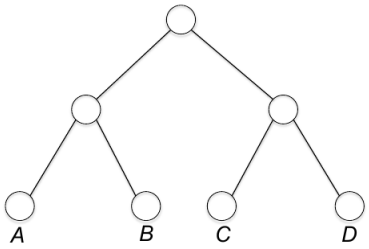
Ta xét 1 ví dụ về cách hoạt động của cách map các chữ cái với cách biểu diễn nhị phân mới của thuật toán Huffman

Vấn đề được xử lý theo hướng "bottom-up", tức từ n nút(với n là số kí tự trong bảng kí tự α), mỗi nút chứa một kí tự khác nhau trong bảng chữ cái α thì ta sẽ dựng nên một cây bằng cách dần dần ghép các nút lại với nhau.

Giả sử ban đầu có 4 cây A, B, C, D(cây chỉ có gốc). Giả sử tần số xuất hiện của các kí tự trong các cây A, B, C, D lần lượt là 30%, 30%, 20%, 20%.

Vì C và D có tần số xuất hiện thấp nhất, ta ghép 2 cây C và D lại thành một cây với C và D có chung 1 nút cha

Sau đó vì 2 cây A và B có tần số thấp nhất nên ta ghép 2 cây A và B lại thành 1 cây có chung 1 nút cha

Sau đó ta ghép 2 nút cha của 2 cây lại và được 1 cây 

Thực ra, thuật toán Huffman duy trì 1 rừng, tức là 1 hay nhiều cây nhị phân, và mỗi lá của mỗi cây sẽ chứa 1 kí tự phân biệt trong bảng kí tự α.

Mỗi vòng lặp sẽ chọn ra 2 cây trong rừng và ghép lại thành một cây cho chung nút cha, 2 cây được ghép sẽ là 2 nút con của nút cha đó. Nút cha không chưa bất kì kí tự nào. Thuật toán sẽ dừng khi chỉ còn 1 cây trong rừng.

3. Mã giả, cách cài đặt và tối ưu

3.1 Mã giả

Thuật toán Huffman sẽ tạo 1 cây-α theo hướng bottom-up, mỗi vòng lặp bất kì thuật toán sẽ ghép 2 cây sao cho 2 cây đó có tổng xác suất xuất hiện của các kí tự có trong nút lá của 2 cây là nhỏ nhất

//init

For each a in α do:

T_a = tree containing only one node, labeled “a”

$P(T_a) = p_a$

F = {T_a} //invariant: for all T in F, $P(T) = \sum_{a \in T} p_\alpha$

//main loop

While length(F) >= 2 do

T_1 = tree with smallest P(T)

T_2 = tree with second smallest P(T)

remove T1 and T2 from F

T_3 = merger of T1 and T2

// root of T1 and T2 is left, right children of T3

$P(T_3) = P(T_1) + P(T_2)$

add T_3 to F

Return F[0]

3.2 Thời gian chạy và tối ưu

Cách implement naive sẽ có thời gian chạy là $O(n^2)$ với n là số kí tự cần được biểu diễn.

Lý do: mỗi vòng lặp trong main loop sẽ tìm ra trong rừng 2 cây sao cho xác suất xuất hiện của 2 cây là nhỏ nhất và nhỏ nhì nên sẽ tốn chi phí thời gian là O(n), mà lại có n-1 vòng lặp như vậy(vì ban đầu xuất phát với n cây, mỗi vòng lặp giảm 1 cây, cho đến khi chỉ còn 1 cây trong rừng) nên thời gian chạy cho main loop sẽ là $O(n^2)$. Các phần khác của công việc như tạo n cây từ n kí tự, thay đổi vị trí trỏ của pointer chỉ tốn chi phí là O(n). Nên thời gian chạy sẽ là $O(n^2)$.

Một cách implement tốt hơn sẽ sử dụng heap để tăng tốc.

Heap có thể hỗ trợ các operation như thêm, xóa, lấy phần tử nhỏ nhất trong O(log(n)). Vì vậy nên ta sẽ sử dụng rừng F là một heap gồm các cây, được phân loại dựa trên xác suất xuất hiện của cây đó.

Khi đó bước tìm cây có xác suất nhỏ nhất và nhỏ nhì sẽ chỉ tốn chi phí là O(log(n)), lặp lại n-1 bước như vậy(bắt đầu từ n cây và giảm dần còn 1 cây, mỗi vòng lặp giảm 1 cây) nên main loop có độ phức tạp chỉ là O(nlog(n)), các chi phí chuẩn bị khác chỉ tốn O(n) nên thuật toán có độ phức tạp là O(nlog(n)).

Mã giả của cách sử dụng heap:

//init

F is an empty heap

For each a in α do:

T_a = tree containing only one node, labeled “a”

$P(T_a) = p_a$

insert T_a into heap F

//invariant: for all T in F, $P(T) = \sum_{a \in T} p_\alpha$

//main loop

While length(F) >= 2 do

T_1 = tree with smallest P(T)

T_2 = tree with second smallest P(T)

remove T1 and T2 from heap F

T_3 = merger of T1 and T2

// root of T1 and T2 is left, right children of T3

$P(T_3) = P(T_1) + P(T_2)$

add T_3 to heap F

Return F[0]

Ta còn có thể tăng tốc thuật toán này hơn nữa bằng cách sort trước tần suất xuất hiện của các kí tự sau đó thực hiện thêm 1 lượng công việc O(n) nữa. Bước sort thường sẽ tốn chi phí là O(nlogn) nhưng sẽ có hằng số nhỏ hơn khi dùng heap nên vẫn sẽ tăng tốc được. Độ phức tạp của thuật toán vẫn sẽ là O(nlogn). Nhưng với 1 số trường hợp đặc biệt, ta có thể tìm cách sử dụng

các thuật toán sắp xếp có độ phức tạp là O(n) thì độ phức tạp của thuật toán huffman sẽ có thể là O(n) trong trường hợp đặc biệt.

Ta có thể hiện thực huffman code bằng cách sort trước theo thuật toán dưới đây:

1. Tạo 2 queue(rỗng)

2. Tạo các cây chỉ có 1 nút và lưu trữ các kí tự vào từng cây đó, thêm các cây vào queue 1 theo thứ tự không giảm

3. Thực hiện bước phía dưới 2 lần:

Nếu queue 1 rỗng thì lấy phần tử ra từ queue 2

Nếu queue 2 rỗng thì lấy phần tử ra từ queue 1

Nếu cả 2 trường hợp trên không xảy ra thì so sánh 2 phần tử đầu 2 queue, và lấy ra phần tử nhỏ hơn

1. Ghép 2 cây vừa lấy được ở bước 3 thành 1 cây và thêm vào queue 2

2. Lặp lại bước 3 và 4 cho đến khi chỉ còn 1 cây trong cả 2 queue

Mã giả của cách làm trên như sau:

create 2 queue: queue1 and queue2

input array A of character and their frequencies

// array of tuples (character, frequency)

Sort all character in array A by their frequencies

For i = 0 to length(A) - 1 do:

T_a = tree containing only one node, labeled A[i][0]

$P(T_a) = p_a$

insert T_a into queue1

//main loop

While there are more than one tree in queue1 and queue2 do

if queue1 is empty then

T_1 = pop(queue2)

if queue2 is empty then

T_1 = pop(queue1)

if peek(queue1) < peek(queue2) then

T_1 = pop(queue1)

else

T_1 = pop(queue2)

if queue1 is empty then

T_2 = pop(queue2)

if queue2 is empty then

T_2 = pop(queue1)

if peek(queue1) < peek(queue2) then

T_2 = pop(queue1)

else

T_2 = pop(queue2)

T_3 = merger of T1 and T2

// root of T1 and T2 is left, right children of T3

$P(T_3) = P(T_1) + P(T_2)$

add T_3 to queue2

Return F[0]

4. **Các thư viện hỗ trợ (C++/Python)**

Thư viện hỗ trợ Huffman coding trong python: dahuffman(https://pypi.org/project/dahuffman/), Huffpy(https://github.com/w-henderson/Huffpy),...

C++ cũng có thư viện như libhuffman(https://github.com/ybubnov/libhuffman)

Còn về phần tự cài đặt, em sử dụng các thư viện như:

Map: để hỗ trợ map các kí tự với các chuỗi nhị phân tương ứng

Priority Queue: thay thế cho heap(priority queue có hỗ trợ lấy phần tử nhỏ nhất với độ phức tạp thời gian là O(logn) giống như heap)

5. **Vận dụng thực tế**

Thuật toán Huffman được sử dụng trong thực tế để tạo các công cụ nén file như Winzip, Gzip

Ngoài nén file, thuật toán Huffman còn có thể dùng để nén hình ảnh. Vì tính chất lossless compression của thuật toán Huffman coding, hình ảnh có thể được nén để lưu trữ tiết kiệm, tốn ít băng thông khi gửi, và khi giải nén ảnh không bị mờ, bể nét

Thuật toán Huffman được sử dụng trong hầu hết các định dạng nén phổ biến như Gzip, PKzip, Bzip2,...[*]

IV. Tài liệu tham khảo

[*] https://stackoverflow.com/questions/2199383/what-are-the-real-world-applications-of-huffman-coding

[1] https://en.wikipedia.org/wiki/Huffman_coding

[2] https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

[3] https://www.programiz.com/dsa/huffman-coding

[4] https://en.wikipedia.org/wiki/Prefix_code

[5] https://huffman.ooz.ie/

[6] https://home.cse.ust.hk/faculty/golin/COMP271Sp03/Notes/MyL17.pdf
[7] https://www.cs.toronto.edu/~vassos/teaching/c73/handouts/huffman.pdf
[8] https://inst.eecs.berkeley.edu/~cs170/fa20/assets/notes/huffman.pdf
[9] https://www.geeksforgeeks.org/efficient-huffman-coding-for-sorted-input-greedy-algo-4/?ref=lbp
[10] https://github.com/heidi-holappa/tira-labra-2022/tree/master/documentation

V. Phụ lục 1: Chương trình minh họa

Chương trình minh họa được viết trong file huffmancoding.cpp
Cách chạy chương trình như sau:

- Để nén một file text:

Chạy chương trình với 3 command line argument lần lượt là --compress *original_file compressed_file*

Với *original_file* là đường dẫn đến file cần nén

compressed_file là đường dẫn đến file đã được nén(file output)

- Giải nén một file text:

Chạy chương trình với 3 command line argument lần lượt là --compress *compressed_file original_file*

Với *compressed_file* là đường dẫn đến file output khi nén một file text

và *original_file* là đường dẫn đến file output mong muốn khi giải nén.

Lưu ý: Documentation cho chương trình minh họa đã được tạo bằng Doxygen. Đã chuẩn bị 2 test cho chương trình là 2 đoạn văn được trích trong truyện A Game Thrones(trong A Song of Ice and Fire) và trong truyện The Last Wish(trong bộ sách The Witcher)

VI. Phụ lục 2: Tính đúng đắn của thuật toán

Định lý: Với mọi alphabet α và các tần số p_a không âm(a thuộc α), thuật toán Huffman sẽ output ra một prefix-free code sao cho trung bình số bit sử dụng là nhỏ nhất.

Để chứng minh định lý trên, ta sẽ dùng quy nạp và đổi biến.

1.Ý tưởng chính

Quy ước sử dụng trong phần chứng minh này cho thuận tiện:

- Cây-α là một cây nhị phân mà chỉ có lá được chứa các kí tự có trong text, và mọi kí tự trong text đều có trong cây α

- a, b lần lượt là hai kí tự có tần suất xuất hiện thấp nhất trong tập hợp ký tự α mà ta đang xét.

Xét vòng lặp đầu tiên, ta thấy thuật toán Huffman gộp 2 nút chứa a và b thành 1 cây trong đó a và b có cùng 1 nút cha, ý tưởng chính ở đây là:

Ta chứng minh rằng trong tất cả các output có cây có a và b có cùng 1 nút cha, thì thuật toán Huffman output ra cây có độ cao của lá trung bình là thấp nhất, hay trung bình số bit sử dụng cho mỗi kí tự là ít nhất.

Nếu ý tưởng thứ nhất được chứng minh thì vấn đề về sự đúng đắn của thuật toán Huffman được đưa về bài toán nhỏ hơn chính là:
Chứng minh rằng trong tất cả các cây-α có a và b có cùng 1 nút cha thì có 1 cây có độ cao lá trung bình thấp nhất trong tất cả các cây có thể.

Nếu 2 ý tưởng trên có thể chứng minh được, ta có thể dùng quy nạp để chứng minh định lý trên.

2.Chi tiết

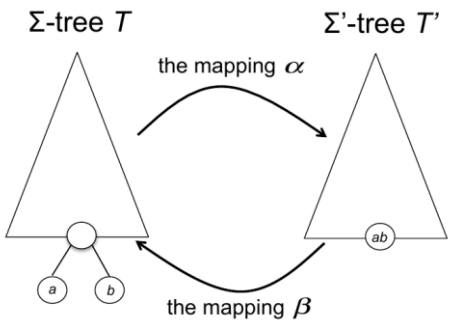
Ta gọi T_{ab} là một cây-α mà trong đó 2 nút chứa a và b lần lượt là con bên trái và con bên phải của cùng 1 nút nào đó trong cây.

Ta sẽ chứng minh rằng trong tất cả các cây $T_{ab,}$ thì thuật toán Huffman output ra cây T_{ab} có độ cao trung bình của lá thấp nhất

Gọi 1 tập hợp ký tự α' là tập hợp giống tập hợp kí tự α nhưng trong đó, 2 kí tự a và b được hợp thành 1 "kí tự" ab, tần số xuất hiện của ab sẽ là tổng tần số xuất hiện của kí tự a cộng với tần số xuất hiện của kí tự b

Tương tự, ta có khái cây-α' cũng giống cây-α nhưng mà trong đó thay vì có 2 kí tự a và b thì 2 kí tự đó được gộp thành 1 kí tự ab, khái niệm cây T_{ab} và cây T'_{ab} cũng có ý tưởng tương tự.

Gọi α mapping là phép đưa cây T_{ab} thành cây T'_{ab}



Gọi β mapping là phép đưa cây T'_{ab} thành cây T_{ab}

Trong vòng lặp chính đầu tiên của thuật toán Huffman, ta gộp 2 nút có chứa kí tự a và kí tự b lại với nhau, nên ta có thể nhập input với set kí tự α', output cho ra cây-α' và sau đó sử dụng β mapping để cho ra kết quả giống với output khi ta cho input là set kí tự α

Từ hình trên, ta cũng có thể thấy là cây α' sẽ có nút ab cao hơn 1 so với vị trí của nút a và nút b trong cây α, do đó ta có:

$$L(T, p) = L(T' , p') + p_a + p_b$$

Vì $p_a + p_b$ là một hằng số nên L(T, p) nhỏ nhất khi L(T', p') nhỏ nhất, do đó: cây T sử dụng số bit trung bình cho 1 kí tự ít nhất khi cây T' có số bit trung bình cho 1 kí tự ít nhất. (*)

Chứng minh ý tưởng 1:

Ta chứng minh bằng quy nạp:

Với base case:

Ta có input chỉ có 2 nút a và b, nên cây được output ra là cây có nút gốc là nút cha của 2 nút a và b.

Giả sử thuật toán Huffman output ra cây có độ cao lá trung bình nhỏ nhất cho input có số kí tự là k-1(k > 3), ta chứng minh thuật toán Huffman cũng output ra cây có độ cao lá nhỏ nhất cho input có số kí tự là k.

Ta có:

- Output của thuật toán Huffman với input là set kí tự α, xác suất p là β(T'), với T' là output của của input là set kí tự α', xác suất p'

- Vì số kí tự trong set kí tự α' là k-1, nên theo giả thiết quy nạp, thuật toán Huffman output ra một cây T' có số bit trung bình cho 1 kí tự là ít nhất.

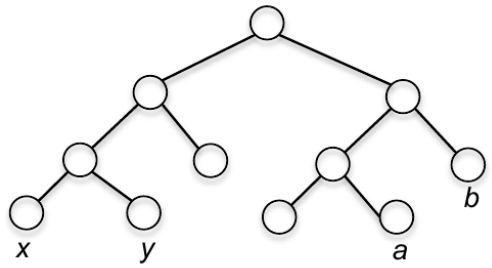
Từ 2 điều trên, kết hợp với (*), ta chứng minh được ý tưởng thứ nhất: *trong tất cả các output có cây có a và b có cùng 1 nút cha, thì thuật toán Huffman output ra cây có độ*

cao của lá trung bình là thấp nhất, hay trung bình số bit sử dụng cho mỗi kí tự là ít nhất.

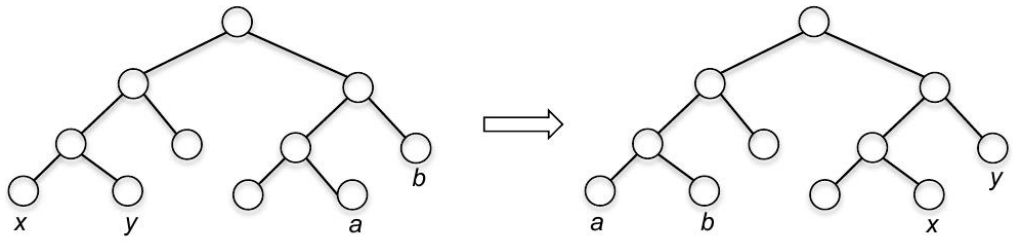
Chứng minh ý tưởng 2:

Để chứng minh ý tưởng này, ta xét với mọi cây-α và chứng minh rằng tồn tại một cây T* thuộc T_{ab} sao cho L(T*, p*) <= L(T, p).

Không mất tính tổng quát, ta giả sử mỗi nút của cây T đều là nút lá hoặc có 2 nút con. Giả sử có 2 nút có chung nút cha và ở độ cao lớn nhất của cây, gọi nút con bên trái có chứa kí tự x và nút con bên phải chứa kí tự y.



Khi đó ta có thể đổi cây T phía trên thành cây T* thuộc T_{ab} bằng cách đổi chỗ x với a và y với b



Ta xét hiệu L(T) - L(T*) =

$$\sum_{z \in \{a,b,x,y\}} p_z.(chiều\ cao\ z\ trong\ T\ -\ chiều\ cao\ z\ trong\ T^*)$$

= (p_x - p_a) . (chiều cao x trong T - chiều cao a trong T)

+ (p_y - p_b) . (chiều cao của y trong T - chiều cao của b trong T)

>= 0

Vì a, b được chọn là 2 kí tự có tần số xuất hiện thấp nhất nên $p_x > p_a$ và

$p_y > p_b$, ta còn có giả thiết là x, y có chiều cao lớn nhất nên chiều cao của x và y sẽ lớn hơn chiều cao của a và b. Do đó nên tổng trên luôn không âm.

Vậy ta có thể kết luận với mọi cây-α bất kì, luôn tồn tại một cây T* thuộc T_{ab} sao cho L(T*, p*) <= L(T, p).

Vậy ta đã chứng minh được ý tưởng chính thứ 2.

Từ đó ta chứng minh được định lý trên và hoàn thành phần proof of correctness cũng như hoàn thành việc trình bày Huffman algorithm.