Bài tập CS112-Phân tích thuật toán không đệ quy

Bài 1: Huffman coding.

Huffman coding là một thuật toán nén nổi tiếng và trong thực tế, thường được sử dụng rộng rãi trong các công cụ nén như Gzip, Winzip.
Dưới đây là một đoạn mã giả về cách tạo Huffman Tree:

```
//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) = ∑_{a∈T} p_α
//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]
```

1) Hãy phân tích và xác định độ phức tạp của thuật toán trên
2) Hãy đưa ra một giải pháp để tối ưu thuật toán vừa được nêu trên

Note:
Với những ai chưa làm quen với Huffman coding, có thể tham khảo bài viết về thuật toán Huffman Coding của mình ở phía dưới, nếu ai đã quen về thuật toán này rồi, xin bỏ qua phần dưới và đọc bài tập 2.

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

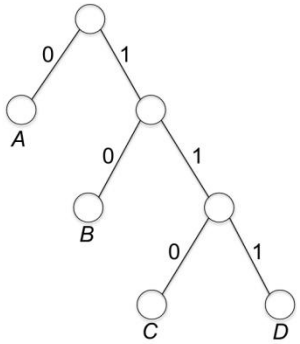
Ta biểu diễn chữ cái "A" bằng số 0, nên các chữ cái khác không được bắt đầu bằng số 0. Tương tự ta biểu diễn "B" bằng 10 nên các chữ cái "C" và "D" phải bắt đầu bằng "11".
Cách biểu diễn trên chỉ sử dụng 1 bit cho chữ cái A, tức ít hơn 1 bit so với cách biểu diễn thông thường nhưng lại sử dụng 3 bit 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 Sử dụng cây để biểu diễn Prefix-Free codes
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

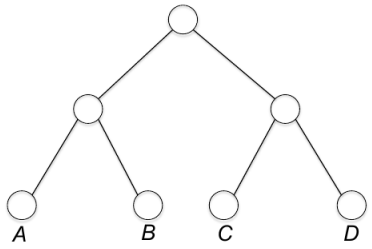
2.4.1 Trình bày thuật toán

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.

Bài 2: Thuật toán Minimum Spanning Tree.

Sau khi học môn *cấu trúc rời rạc* thì ai cũng đã quen với 2 thuật toán thường sử dụng trong môn học là thuật toán Prim và thuật toán Kruskal. Nên trong bài này, ta sẽ phân tích về 2 thuật toán này.

Câu 1:

Prim

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.
Output: the edges of a minimum spanning tree of G.

```
// Initialization
X := {s}    // s is an arbitrarily chosen vertex
T := ∅      // invariant:  the edges in T span X
// Main loop
while there is an edge (v, w) with v ∈ X, w ∉ X do
    (v*, w*) := a minimum-cost such edge
    add vertex w* to X
    add edge (v*, w*) to T
return T
```

1) Hãy đưa ra mã giả chi tiết cho thuật toán trên và phân tích độ phức tạp của thuật toán.

2) Phương pháp mà bạn đã đề ra có cho độ phức tạp là O((n+m)log(n)) (với n là số đỉnh còn m là số cạnh) không? Nếu không thì bạn hãy đề xuất một phương pháp khác cho độ phức tạp như trên.

Câu 2:

`Kruskal`

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.
Output: the edges of a minimum spanning tree of G.

```
// Preprocessing
T := ∅
sort edges of E by cost  // e.g., using MergeSort[26]
// Main loop
for each e ∈ E, in nondecreasing order of cost do
    if T ∪ {e} is acyclic then
        T := T ∪ {e}
return T
```

1) Hãy đưa ra mã giả chi tiết cho thuật toán trên và phân tích độ phức tạp của thuật toán.
2) Phương pháp mà bạn đã đề ra có cho độ phức tạp là O((n+m)log(n)) (với n là số đỉnh còn m là số cạnh) không? Nếu không thì bạn hãy đề xuất một phương pháp khác cho độ phức tạp như trên.