Coduri Huffman Optimizate

Roscaneanu George, 508

Ce este un cod Huffman ?

- Este un cod prefix optim care este adesea folosit la comprimarea “lossless” a datelor.
- Procesul de căutare și folosire unui astfel de cod Huffman a fost descris printr-un algoritm creat și publicat de către David A. Huffman in 1952 in hartia "A Method for the Construction of Minimum-Redundancy Codes"[1].

Alte detalii despre codurile Huffman ?

- Este un cod de lungime variabila (un alt exemplu de cod cu lungime variabila ar fi UTF-8)
- Ele sunt optime pentru o anumita colecție de date doar dacă a fost generat special folosind distribuția de probabilități ale caracterelor unice din acea colecție
- De-asemenea, ele sunt optime doar pentru codarea a cate unui simbol odata. In majoritatea cazurilor nu este mai optim decât un algoritm de comprimare a datelor care ia în considerare întregi bucăți de date care se repeta.

Algoritmul

- **Intrare**: un array cu toate caracterele si cat de des apar ele
- **Iesire**: un arbore in care toate frunzele reprezinta caracterele de mai devreme;

  caracterele care apar mai des vor fi pozitionate mai aproape de varf, iar cele care apar mai rar, vor fi pozitionate mai departe

Pseudocod

```
struct Node{ Node *l,*r; int q; int c;};
Node* generateHuffmanTree(vector<pair<int,int>> v){
    //functia extract va extrage pointer catre nodul cu cel mai mic q
    minCountPriorityQueue q;
    for(int i=0;i<v.size();i++){
        Node *n = new Node; n→l = n→r = 0; n→c = v[i].first; n→q = v[i].second;
        q.push(n);
    }
    while(q.size() > 1){
        Node *n=new Node; n→r = q.extract(); n→l = q.extract();
        n→q = n→l→q + n→r→q; q.push(n);
    }
    return q.extract();
}
```

Complexitatea pseudocului anterior ?

Operatia de inserare a unui element intr-un priority_queue are complexitatea O(log n)

Avem n elemente de inserat la inceput, deci primul for are complexitatea O(n * log n)

Extragerea unui element din priority_queue are complexitatea O(log n)

Un arbore binar cu n frunze, are n-1 noduri interne.

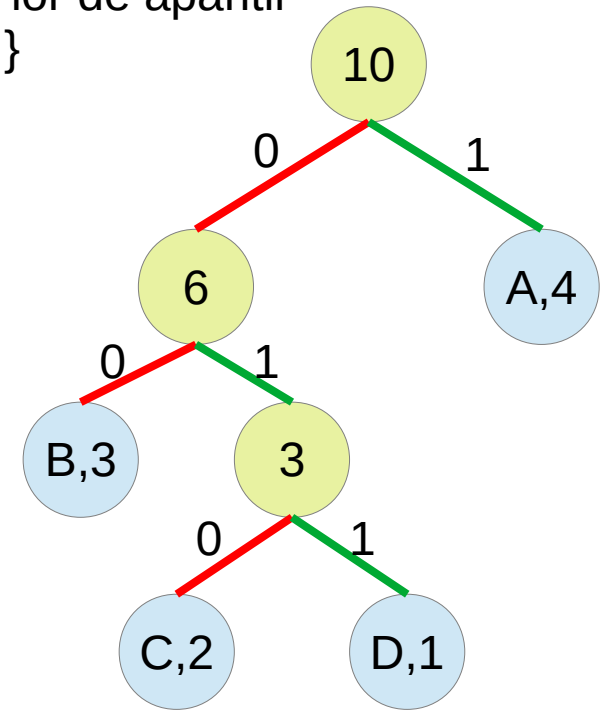
While loop-ul este chemat de n-1 ori, astfel avand complexitatea O(n * log n)

Complexitatea pseudocodului = O(n * log n)

Avand in vedere faptul ca de obicei numarul de caractere unice dintr-un text este relativ mic (256 daca vorbim despre fisiere binare), complexitatea O(n*log n) nu este o problema.

Dar unde sunt codurile ?

Fie caracterele si numarul lor de aparitii
{ {A,4} , {B,3}, {C,2}, {D,1} }



Cod
A : 1
B : 00
C : 010
D : 011

Cum este stocata “cheia” ?

Deoarece numarul maxim de noduri intr-un copac Huffman pentru fisiere binare este 256 (frunze) + 255 (noduri interne) = 511, putem stoca cheia astfel:

- n = o valoare de 8 biti pentru numarul de noduri interne
- n perechi de numere de cate 9 biti fiecare care reprezinta copii stangi si drepti ale nodurilor interne

In cazul cel mai fericit ocupa 8 + 1*2*9 = 26 biti ~ 3 octeti

In cazul cel mai rau ocupa 8 + 255*2*9 = 4598 biti ~ 574 octeti

Problema 1)

Sa presupunem ca am generat arborele Huffman pentru un fisier si am obtinut urmatoarele coduri: { A : 0 , B : 1}

Si ca “payload”-ul care a ramas sa fie procesat dintr-un fisier comprimat dupa ce am obtinut cheia este

01010000_b

Daca ar fi sa decomprimam acest fisier am obtine

“ABABAAAA”

Ceea ce-ar fi fantastic ! Doar ca fisierul original poate era doar

“ABAB”

Din pacate pentru a rezolva problema acesta am folosit o valoare de 8 octeti la inceputul fisierului pentru a descrie cat de mare a fost fisierul original.

Problema 2)

Fisierul original trebuie sa contina cel putin 2 caractere diferite. Altfel implementarea curenta nu va functiona.

De ce maxim 64 biti pentru cod ?

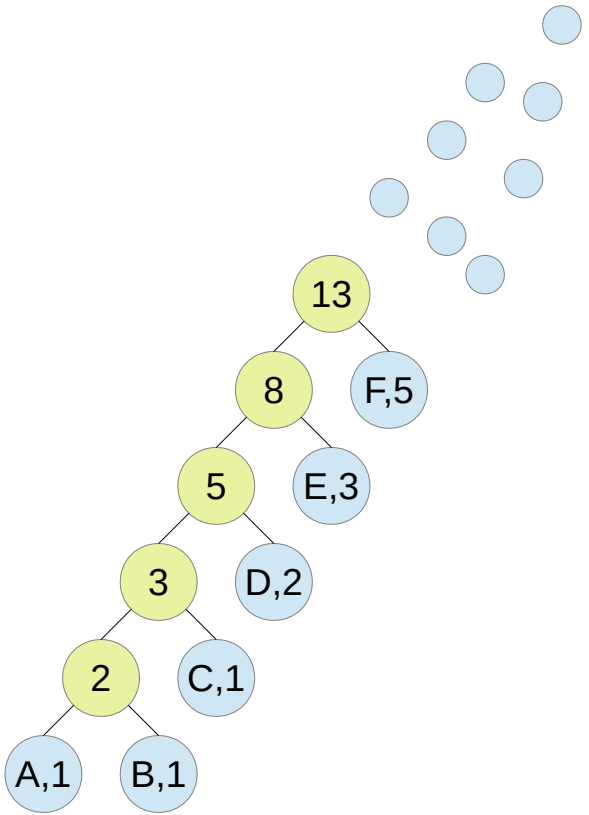
Nu exista un fisier care este fezabil suficient de mare incat sa genereze un arbore Huffman in care sa fie macar un cod de cel putin 64 de biti.

O incercare ar fi sa generezi in mod intentionat un fisier unde numarul de caracter corespund sirului Fibonacci.

A:1, B:1, C:1, D:2, E:3, F:5, G:8, H:13, I:21 ….

Acel A:1 este intentionat pus, si vom vedea imediat de ce.

De ce maxim 64 biti pentru cod ?



Ca acest patern sa continue tot asa, este nevoie de foarte multe caractere.

Cel de-al 48-lea caracter trebuie sa apara de mai mult de 2^{31} ori.
Al 49-lea de cel putin 2^{32} ori.
Al 94-lea de cel putin 2^{64} ori.
Iar in sfarsit, al 256-lea caracter de cel putin 2^{175} ori.

Prezentarea programului !

Referinte

1) http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf
Huffman, D. (1952). "A Method for the Construction of Minimum-Redundancy Codes"