

Compresia Datelor

Stoica Robert-Valentin

Universitatea Politehnica, Bucuresti

Abstract. Scopul acestei lucrari este acela de a aduce in atentie si a analiza eficienta algoritmilor de compresie Huffman si Lempel-Ziv-Welch (LZW) pe un set cat mai amplu de date.

Keywords: Compresie · Huffman Coding · Lempel-Ziv-Welch

1 Introducere

1.1 Descrierea problemei

Compresia datelor este un aspect foarte important atunci cand vorbim de lucrul cu fisiere de orice tip (audio, video, text etc) deoarece permite partajarea si pastrarea lor intr-o forma redusa, simplificata, putand astfel economisi resurse.

Folosirea fisierelor comprimate in locul celor originale ne ofera atat eficienta in cadrul utilizarii memoriei fizice (pastrarea lor intr-o forma redusa va ocupa mai putin spatiu pe HDD/ SSD) cat si una temporala in cazul in care dorim sa partajam resurse prin intermediul internetului, un fisier de dimensiune mai mica fiind incarcat si transferat mult mai usor.

1.2 Specificarea solutiilor alese

Cele doua metode de compresie ce urmeaza a fi analizate fac parte din clasa "lossless data compression" (compresie fara pierdere a datelor), asta insemnand ca, in urma procesului de decompresie, fisierul poate fi adus la forma sa initiala fara a suferi modificari de orice tip.

In cazul compresiei Huffman, algoritmul creeaza un cod binar unic pentru fiecare caracter (sau octet) existent in cadrul fisierului. Acest cod va fi mai scurt sau mai lung in functie de frecventa fiecarui caracter. Astfel, caracterele ce apar foarte des vor avea un cod binar **scurt** iar cele ce apar rar, vor avea unul mult mai **lung**. "Magia" acestui algoritm sta in capacitatea sa de a compensa codurile scurte ce apar de multe ori cu codurile lungi ce apar mult mai rar, in medie acesta reusind sa aduca fisierul la o dimensiune mai mica decat cea originala.

In schimb, metoda de compresie Lempel-Ziv-Welch aduce in plus posibilitatea de a codifica **secvente de mai multe caractere ce se repeta**. Algoritmul aduce cu sine un dictionar ce isi mareste capacitatea cu fiecare secventa noua gasita. Desi codificarile vor fi facute pe un numar fix de biti (in general mai mare decat dimensiunea unui octet, ajungand pana la 20 de biti), daca fisierul are cuvinte sau secvente de litere ce se repeta, acestea vor aparea in fisierul comprimat sub forma unui singur cod.

1.3 Evaluarea soluțiilor

Singurul aspect ce ne interesează în cazul algoritmilor de compresie este dat de raportul dintre dimensiunea fișierului comprimat și cea a fișierului original, altfel spus, eficiența algoritmilor constă în **procentul de reducere** al fișierului original.

Pentru evaluarea eficienței algoritmilor voi încerca să acopăr o plajă cât mai mare a datelor de intrare atât prin tipul, dimensiunea dar și conținutul lor.

În cazul fișierelor text, voi genera date de intrare mai mult sau mai puțin favorabile pentru ambele tipuri de compresie. Un exemplu bun în cazul fișierelor text ar fi un fișier ce conține **o singură literă de un număr mare de ori**:

- în cazul algoritmului Huffman, această compresie este foarte bună, în noul fișier reținându-se un singur cod foarte scurt (+ datele auxiliare necesare decompresiei)
- pentru algoritmul LZW, compresia este una mult mai slabă, acesta adăugând grupuri de aceeași literă dar dimensiuni diferite în dicționar și în fișierul nou rezultat

Totuși, în ambele cazuri, fișierul comprimat va fi **mai mic** decât cel inițial.

Un alt exemplu ar putea fi un text ce conține **caractere complet diferite**. În acest caz, pentru date de intrare mai mari, ambii algoritmi vor fi **extrem de ineficienți**, existând posibilitatea ca noul fișier să fie chiar mai mare decât cel original.

Testele vor fi făcute atât pentru cazuri bine alese (ca cele de mai sus), cât și pentru fișiere generate aleator (text copiat de undeva sau imagini care nu au un tipar anume).

O altă abordare ar putea fi data de testarea algoritmilor pe date de intrare care deja sunt supuse unui grad mare de compresie (imagini, fișiere PDF etc), moment în care ne așteptăm ca algoritmi să aibă un impact **nesemnificativ** (sau chiar unul rău - dimensiunea fișierului crește) asupra fișierului de intrare.

Pentru o parte din teste voi genera date **complet aleatoare** folosind comanda: `cat /dev/urandom`.

Scopul lucrării este acela de a analiza toate aceste situații și a observa în ce caz este mai bun un algoritm decât celălalt.

2 Prezentarea soluțiilor

Algoritmul Huffman are la bază o structură de date arborescentă ce poartă același nume (arborele Huffman). Acest arbore este special construit astfel încât, parcurgând nodurile sale de la rădăcina spre oricare dintre frunze, drumul parcurs va fi mai scurt cu cât caracterul atribuit frunzei are o frecvență mai mare în fișierul de intrare. Pentru a realiza această proprietate, construcția arborelui începe cu frunzele acestuia și înaintea spre rădăcina, nodurile fiind adăugate

crescator in functie de frecventa. Pentru a face asta, am folosit un minheap care imi va extrage mereu litera cu frecventa minima ce nu a fost inca adaugata si ii va crea un nod in arborele Huffman. Fiecare doua noduri vor fi unite printr-un nod additional ce nu contine niciun caracter si are frecventa egala cu suma frecventelor celor doua noduri copil. Deoarece dorim coduri cat mai scurte, in momentul in care frecventa nodului parinte devine mai mare decat frecventa oricarei alte litere, va fi creat un subarbore ce respecta aceleasi proprietati si care va fi ulterior atasat de arborele principal. Acest proces este realizat prin adaugarea in minheap a noului nod.

Odata obtinut arborele, **comprimarea** se realizeaza prin retinerea intr-un fisier binar a fiecarui caracter intalnit alaturi de frecventa sa (date necesare decomprimarii), dupa care fisierul de input este parcurs si fiecare litera din acesta este inlocuita in fisierul binar cu codul corespunzator. Pentru un acces mai rapid, am folosit un hashtable ce are cheia egala cu valoarea numerica a caracterului (byte-ului), valoarea fiind reprezentata de codificarea sa binara.

Pentru procesul de **decomprimare** sunt citite din fisierul binar caracterele impreuna cu frecventa lor dupa care este refacut arborele Huffman. Odata creat, se citeste fisierul bit cu bit si se parcurge arborele in functie de acesta (0 - stanga, 1 - dreapta). Cand se ajunge la o frunza putem fi siguri ca aceasta contine un caracter valid, caracterul fiind scris apoi in noul fisier.

Complexitatea temporală a acestui algoritm este $O(N \log N)$ deoarece fiecare iteratie prin minheap pentru a pastra proprietatea acestuia ne va costa $\log N$ pasi, aceasta fiind refacuta de N ori (o data pentru fiecare element). In plus, pentru a construi arborele, la fiecare pas sunt extrase 2 noduri din minheap, ceea ce ar insemna $2 \log N$ operatii pentru a reface heapul si apoi este adaugat in minheap nodul parinte obtinut (inca $\log N$ operatii pentru a reface heapul). Aceste operatii sunt executate de N ori (pentru ca la fiecare pas sunt scoase 2 noduri si adaugat unul nou) ceea ce in final ne va duce la o complexitate de $O(3N \log N)$ adica tot $O(N \log N)$, unde N este numarul de caractere diferite intalnite in fisier.

Cum N -ul nostru nu poate fi mai mare de 256, putem considera aceste operatii ca fiind chiar constante, adevarata "greutate" a algoritmului fiind de fapt data de parcurgerea datelor de intrare, complexitatea fiind in acest caz $O(M)$, unde M este numarul de caractere (bytes) din fisier.

Algoritmul Lempel-Ziv-Welch (LZW) are la baza o structura de tip dictionar ce adauga secvente noi de litere folosindu-se de cele anterioare. Acest dictionar porneste cunoscand initial doar caracterele ASCII si se extinde continuu pana la finalul algoritmului. Codul secventei nou aparute va primi urmatorul index disponibil in dictionar. Fata de Huffman care putea genera un cod cu numar diferit de biti, algoritmul LZW are de la inceput stabilit pe cati biti vor fi facute codificarile. Deoarece testele alese de mine contin si fisiere mai mari de 1MB, codificarea a fost facuta pe un numar de 20 de biti.

Pentru a fluidiza putin codul am separat functiile care comprima si decompima fisierul de cele care fac citirea si scrierea in si din fisiere binare, acestea comunicand intre ele prin intermediul unui vector de inturi.

Am ales sa scriu acest algoritm in C++ deoarece aveam nevoie de implementarea unor structuri de date cu o capacitate mare de stocare ce isi puteau extinde usor dimensiunea (atat pentru dictionar cat si pentru vectorul auxiliar).

In vederea **comprimarii** fisierului de input, acesta este parcurs litera cu litera. Odata primita o litera, acesta o adauga la secventa pe care o detinea anterior. In momentul in care secventa curenta nu mai este gasita in dictionar, se scrie codul ultimei secventa cunoscute in vectorul de inturi, noua secventa este adaugata in dictionar iar algoritmul isi reincepe cautarea.

In cazul **decompresiei**, codurile consecutive duc la adaugarea in dictionar a celor doua stringuri concatenate, algoritmul urmarind procesul invers celui de comprimare.

Complexitatea temporală este, la fel ca si la Huffman, data de dimensiunea datelor de intrare, atat indexarea prin tabela de dispersie cat si adaugarea in ea (si in vector) fiind efectuate in timp constant. Singurul factor ce induce o complexitate diferita de una constanta se datoreaza fisierului ce trebuie parcurs pentru citirea datelor de intrare. De asemenea, pentru parcurgerea vectorului auxiliar se ajunge la aceeasi complexitate ca si la citire, aceasta fiind tot $O(M)$, unde M reprezinta dimensiunea fisierului de intrare.

Avantaje si dezavantaje

Algoritmul Huffman are avantajul ca poate rula pe fisiere de orice dimensiune fara a fi limitat la un anumit numar de biti pe codificare, in timp ce algoritmul LZW are nevoie de la inceput de un numar fix de biti pe care sa fie reprezentate codurile. Aceasta constrangere influenteaza in mod negativ performanta algoritmului deoarece, daca am alege un numar mare de biti pentru fiecare codificare, chiar daca algoritmul ar putea acum sa comprime fisiere mari fara probleme, acesta ar avea un randament foarte prost pe fisiere mici. Acest lucru se datoreaza faptului ca nu vor fi folositi toti bitii pe care noi ii oferim pentru codificare, caz in care vom avea o buna parte din memorie irosita.

In schimb, algoritmul LZW castiga eficienta in cazul in care, caractere cu aceeasi frecventa apar dupa un tipar stabilit. In acest caz, el isi poate folosi dictionarul intr-un mod foarte eficient, compresia rezultata fiind si ea una foarte buna. Pe aceeasi intrare, algoritmul Huffman s-ar descurca foarte prost tocmai pentru ca nu exista o diferenta intre numarul de aparitii al anumitor caractere, arborele fiind construit in mod aleator, fara a avea un randament bun cand atribuie codurile caracterelor.

Pe un text normal ce contine doar caractere alfa-numerice, ambii algoritmi s-ar descurca bine, micile diferente fiind date de numarul de repetitii al caracterelor/ cuvintelor (daca se repeta multe cuvinte, balanta se va inclina in favoarea LZW, daca se repeta multe litere, Huffman va fi mai eficient). Diferenta poate fi facuta si de numarul fix de biti ai codificarii LZW, caz in care multi biti ar putea fi irositi.

Din punct de vedere al memoriei folosite, Algoritmul Huffman are o eficienta mult mai buna decat LZW, acesta ajungand sa aiba alocat maxim 256 de noduri+ nodurile aditionale pentru parinti. In schimb, LZW isi va mari constant

dictionarul chiar daca informatia ar putea fi inutila, acesta continuand sa creasca pana la terminarea programului, atingand un numar foarte mare de codificari.

3 Evaluare

In testele generate am incercat sa ating atat cazuri limita cat cazuri medii si bune pentru ambii algoritmi. O parte din testele folosite au fost generate de mine folosind un program simplu in C, celelalte fiind descarcate de pe internet (pozele si fisierele audio).

In cazul fisiereleor .txt am incercat sa acopar cazuri de genul:

- acelasi caracter de un numar mare de ori
- 3 caractere de un numar mare de ori (aceeasi frecventa toate 3)
- intregul alfabet de un numar mare de ori (aceeasi frecventa toate 3)
- toate caracterele ascii de un numar mare de ori (aceeasi frecventa toate 3)
- toate caracterele ascii de un numar mare de ori (frecvente diferite)
- caractere cu diferenta mare intre numarul de aparitii
- text lung care nu se repeta
- text scurt care se repeta de un numar mare de ori

In cazul imaginilor, am ales formatul .bmp deoarece bitii imaginii sunt stocati in mod "direct", imaginea nefiind deja comprimata (asa cum sunt fisierele .jpg sau .png). Cu aceste teste am incercat sa ating urmatoarele posibilitati:

- imagini cu numar redus de culori (doua sau trei)
- imagini colorate cu dimensiune mica
- imagini colorate cu dimensiune mare

Am folosit in continuare 3 teste cu fisiere care sunt deja intr-o forma comprimata (fisiere .webp si .mp3) si un fisier binar generat random folosind comanda cat /dev/urandom.

Ultimele teste contin fisiere sursa din diverse limbaje de programare. Acestea sunt niste exemple bune de fisiere text deoarece atat elementele de sintaxa cat si numele variabilelor se repeta, caz in care algoritmii ar fi destul de eficienti.

Ruland cei doi algoritmi pe datele de intrare mentionate mai sus, avand la dispozitie in procesor I7 genetaria 11, 16 GB de ram si o placa grafica de 4GB, rezultatele au fost urmatoarele:

	INPUT	HUFFMAN	LZW
test1	1000	13	116
test2	293K	62K	3,3K
test3	254K	152K	9,0K
test4	98K	99K	18K
test5	4,9K	4,4K	496K
test6	265K	168K	11K
test7	54K	34K	38K
test8	153K	95K	54K
test9	85K	24K	5,1K
test10	235K	47K	8,0K
test11	118K	20K	3,0K
test12	516K	184K	124K
test13	226K	221K	315K
test14	807K	741K	816K
test15	1,2M	937K	1,2M
test16	438K	440K	622K
test17	924K	913K	1,1M
test18	1,2M	1,2M	1,5M
test19	5,2K	3,3K	4,4K
test20	14K	8,3K	8,6K
test21	2,2K	1,7K	2,5K
test22	7,7K	5,1K	6,0K
test23	218K	214K	299K
test24	9,8K	12K	23K

Fig. 1. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

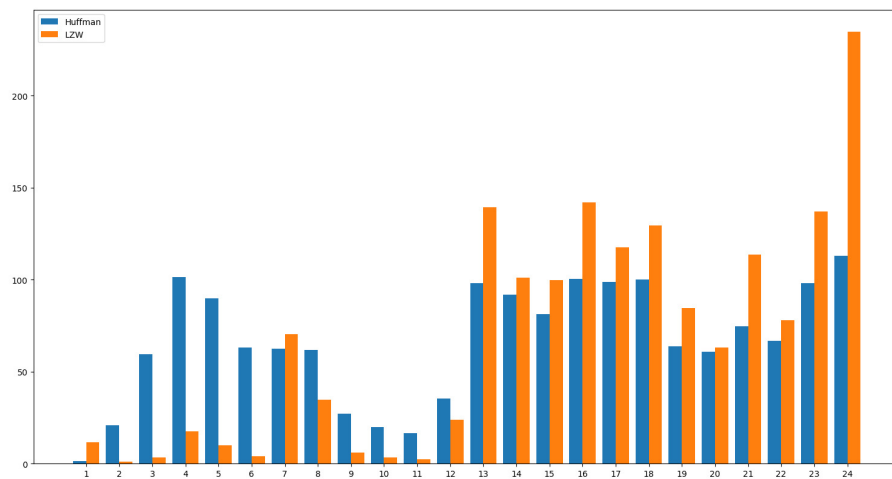


Fig. 2. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.