Compresia datelor

Asimionesei Daniel

15.11.2019

Cuprins

1 Descrierea problemei rezolvate

Să se realizeze compresia şi decompresia unui fişier text.Să se analizeze doi algoritmi de comprimare LOSSLESS.

Principala limitare a fişierelor text este aceea că fiecare literă este reprezentată pe 8 biţi de informaţie, ceea ce ocupă mult spaţiu,iar mulţi dintre biţi sunt inutili.De aceea vom identifica nişte elemente comune, în modul de distribuire a datelor dintr-un fişier ce urmează să fie comprimat, şi folosirea acestor elemente comune, pentru a elimina partea din date, fără a afecta informaţia.

Aplicaţi practice pentru această problemă sunt programele de compresie ale datelor, cum ar fi WinRar,WinZip,7-Zip etc, când datele sunt "îngrămădite" pentru a ocupa mai puţin spaţiu pe sistemul de stocare.

2 Specificarea soluţiei alese

Soluţiile alese in rezolvarea problemei sunt algoritmi: Huffman coding si Arithmetic Coding.

Algoritmul lui Huffman constă în înregistrarea simbolurilor întâlnite într-un fişier prin coduri de lungime variabilă.Algoritmul lui Huffman este un cod de tip bloc-variabil.Într-un fişier F de lungime n, având un alfabet de m simboluri, se identifică a_{min}, caracterul cu frecvenţa cea mai mică si a_{max}, caracterul cu frecvenţa cea mai mare.Astfel, se asociază simbolului a_{min} o configurare de biţi de lungime mare in timp ce pentru a_{max} vom avea cea mai mică configurare de biţi posibilă.Întrucât compresia Huffman este o codare cu lungime variabilă a cuvântului de cod, aceasta presupune construirea unui dicţionar de cuvinte de cod, in care se află cuvintele de cod ce corespund unui anumit mesaj al sursei.Algoritmul presupune construirea de surse de informaţie restrânse prin reunirea simbolurilor cel mai puţin probabile ale sursei anterioare.Începâd de la utima sursă restrânsă se alocă câte un bit de 0 si de 1 celor două simboluri, iar codurile simbolurilor reunite sunt prefix pentru codurile simbolurilor ce le compun.Se repeta alocarea de biti pâna când s-au codat toate simbolurile sursei iniţiale.Prin citirea bit cu bit a fişierului comprimat se poate decomprima fişierul folosind aceeaşi tabelă de compresie.

Arithmetic Coding ocoleşte complet ideea înlocuirii unui simbol de intrare cu un cod specific.În schimb înlocuieşte simbolurile de intrare cu un sigur număr cu punct flotant($0 < \text{nr} < 1$).Simbolurilor codificate le vor fi atribuite un set de probabilităţi.În implementarea algoritmului se va atribui fiecărui simbol un interval.Începând cu intervalul [0,1] fiecare interval este împarţit in mai multe subintervale(câte simboluri diferite există), dimensiunile lor fiind proporţionale cu probabilitatea curentă a simbolurilor corespunzătoare.Extind prima literă care va fi codată împreună cu intervalul si divizez acest interval în numărul de simboluri.Repet această procedură până la codificarea caracterului de încheiere. Numărul rezultat poate fi decomprimat in mod unic pentru a creea şirul exact de simboluri care au intrat in construcţia sa.Algoritmul de decomprimare se realizează doar inversând procesul de codificare.

3 Criterii de evaluare pentru soluţia propusă

Evaluarea soluţiei va urmări resursele consumate (spaţiu si timpul de compresie).Se va analiza rata de compresie (dimensiunea iniţială şi dimensiunea finală dupa compresie pentru fiecare algoritm în parte si petru aceleaşi fişiere de intrare).

Întrucât algoritmi au la bază frecvenţa caracterelor si eliminarea biţilor nesemnificativi din reprezentarea caracterelor în binar, voi urmări ca fişierele de intrare sa aibă un număr variat de tipuri de caractere pentru a obţine cât mai multe coduri de lungime variabilă.În acelaşi timp voi urmări să am şi simboluri cu probabilităţi cât mai mari ,iar altele cu probabilităţi cât mai mici.

4 Prezentarea soluţiei

4.1 Descriei modului in care functioneaza algoritmi alesi

HUFFMAN:

In implementarea algoritmului am parcurs fisierul text de intrare si am pus toate caracterele intr-un vector(String).Am folosit un unordered map in care am salvat simbolurile distincte cu frecventa de apartie corespunzatoare.Am creat o coada de prioritati, in care am pus nodurile arborelui de codare ce urmeaza sa fie creat.Fiecare nod contine doua informatii:caracterul si frecventa.Nodurile vor fi puse in ordine descrescatoare a frecventei de aparitie a literelor.Iau primele doua noduri din coada si creez un nod intern, avand copii cele doua noduri si frecventa egala cu suma fercventelor si adaug nodul format in coada si repet acest procedeu pana cand in coada ramane doar un nod (nodul radacina cu suma tuturor frecventelor).Arborele astfel creat il voi folosi pentru codarea simbolurilor.Frunzele arborelui vor contine caracterele din unordered_map, iar nodurile interne contin doar un numar (suma frecventelor copiilor).Apoi,pentru fiecare caracter parcurg arborele de la radacina la frunza pentru a genera codul acestuia.La fiecare pas la stanga adaug 0,iar la fiecare pas la dreapta adaug 1.Aceasta secventade 0 si 1 o concatenez la stringul comprim.La urmatorul pas scriu in fisierul binar secventa de biti de 0 si de 1 din comprim.Pentru decomprimarea fisierului binar,voi salva in stringul comprim secventa de biti din acesta si voi citi bit cu bit, iar la gasirea unei secvente de biti care corespunde unui simbol voi scrie simbolul respectiv.

ARITHMETIC CODING

Pentru implementarea algoritmului de codare am urmarit urmatorii pasi:
1) Impart intervalul [0,1] in atatea subintervale, cate simboluri diferite exista in text.
2) Extind prima litera care va fi codata impreuna cu intervalul.Divizez acest interval in numarul de simboluri.
3) Repet acest procedeu pana la codificarea caracterului de incheiere.

Pseudocod codare:
-> apelez encode_symbol in mod repetat pentru fiecare simbol din mesaj;
-> ma asigur ca simbolul terminator este codat ultima data, apoi transmit orice valoare din intervalul [low,high] ;
encode_simbol(symbol,cum_freq)
range = high-low
high = low + range*cum_freq[symbol-1]
low = low + range*cum_freq[symbol]

Pseudocod decodare:
-> value este numarul primit;
-> apelez decode_symbol pana cand simbolul terminatorului este returnat;
decode_symbol(cum_freq)

cum_freq[symbol] <= (value-low)/(high-low) < cum_freq[symbol-1]
range = high - low
high = low + range*cum_freq[symbol-1]
low = low + range*cum_freq[symbol]
return symbol;

Simbolurile sunt numerotate incepand de la 1 la n.Intervalul de frecventa pentru simbolul i este de la cum_freq[i] la cum_freq[i-1].Pe masura ce i scade, cum_freq[i] creste si cum_freq[0]=1.

Intervalul curent este [low,high], iar pentru codare si decodare acesta va fi initializat la [0,1].

Algoritmul de codare descris nu transmite nimic pana cand intregul mesaj a fost codat;nici algoritmul de decodare nu incepe decodarea pana cand nu a primit transmisia completa.

Precizia necesara pentru a reprezenta intervalul [low,high], creste odata cu lungimea mesajului.

Main ENCODING:
-initializez modelul;
-intr-o bucla citesc urmatorul caracter din fisierul text si ies din for cand intalnesc end-of-file;
-transform caracterul intr-un index;
-codez simbolul;
-update model;
-la finalul codului ,codez simbolul EOF si transmit ultimi biti;

Main DECODING:
-initializare model;
-intr-un for decodez urmatorul simbol pana intalnesc EOF simbol;
-transform simbolul in caracter;
-scriu caracterul;
-update model;

4.2 Analiza complexitatii solutiei

Pentru aflarea complexitatii algoritmului lui Huffman vom pleca de la pseudocodul algoritmului implementat:
Input: caracterele si frecventele lor : (c_1,f(c_1)),(c_2,f(c_2)),...,(c_n,f(c_n));
Output: return HuffmanTree;
Make priority queue Q using c_1,c_2,...,c_n
for(int i=1;i++;i<n)
z=allocate new node;
e=Q.extract_min();
r=Q.extract_min();
z.left=1;
z.right=1;
f(z)=f(r)+f(l);
q.insert(z);

return Q.extract_min();
Mutarea unui element de la radacina la nodul frunza necesita comparatii O(log n) si acest lucru se face pe n/2 elemente,in cel mai rau caz.

T(n) = T(n-1) + O(n) => O(n^2)

In cazul in care se foloseste priority queue:

T(n) = T(n-1) + O(log n) => O(nlogn)

Pentru a afla complexitatea algoritmului Arithmetic codind, vom pleca de la urmatorul pseudocod:

```
low=0.0;
high=1.0
range=1;
while(symbol != terminator)
{
    get(symbol);
    low=low+range*Range_low(symbol);
    high=low+range*Range_high(symbol);
    range=high-low;
}
```

out: low <= code < high

T(n) = Nlog(n)

N - numarul simbolurilor de intrare;

n - numarul simbolurilor unice;

4.3 Avantaje si dezavantaje solutii

Dupa cum se poate observa si in complexitatile celor doi algoritmi ,Arithmetic coding depaseste Huffman in abilitatea de comprimare(ESTE ASIMPTOTIC MAI BUN).

Huffman atribuie un numar intreg de biti fiecarui simbol , in timp ce Arithmetic Coding atribuie un singur cod pentru intregul string de intrare.

Arithmetic coding consta in putine operatii aritmetice,lucru care duce la o complexitate mai mica.

Ambi algoritmi nu pierd informatie.

Dezavantaje Huffman:

Codurile binare din datele codificate au lugimi diferite, iar acest lucru face dificila decodificara.

Daca alfabetul este mic, algoritmul nu este eficient.

Avantaje Huffman:

Caractrele cu frecventa mare sunt codificate cu mai putini biti, iar cele care au frecventa mica sunt codificate cu mai multi biti.(economie de spatiu de stocare)

Este eficient cu alfabeturi cu simboluri distribuite uniform.

Avantaje Arithmetic coding:

Separa procedeul de modelare de cel de codificare.Fiecare simbol individual poate avea o probabilitate separata care poate fi predeterminata sau calculata

adaptiv in functie de datele anterioare.

Raport de compresie mic,produce un singur simbol.

Performante de compresie aproape optime pentru surse cu entropii foarte scazute.

Dezavantaje Arithmetic coding:

Intregul cod de ordine trebuie primit pentru a incepe decodarea simbolurilor, iar daca exista un bit corupt in codul de ordine, intregul mesaj ar putea deveni corupt.

Exista o limita a preciziei numarului care poate fi codat, limitand astfel numarul de simboluri de codat.

Spre deosebire de Huffman,decodorul necesita cunoasterea numarului de simboluri care trebuie decodate.

Necesita precizie infinita.

5 Evaluare

5.1 Setul de teste folosite pentru validare

Modul in care am construit testele a fost acela de a urmari vulnerabilitatile celor doi algoritmi,astfel testele variaza in functie de:

->densitatea caracterelor(caractere cu frecvente foarte mari si caractere cu frecvente foarte mici);

->dimensiunea alfabetului (alfabete cu un numar mare de caractere distincte si alfabete cu un numar mic de caractere distincte);

->distribuirea caracterelor (uniforma);

->lungimea sirului de caractere(limitat;)

Cea mai mare influienta asupra algoritmului lui Huffman o are frecventa caracterelor(duce la generarea de coduri de dimensiuni diferite), iar algoritmul Arithmetic coding este influientat cel mai mult de lungimea sirului de intrare(impune limite asupra preciziei numarului care poate fi codat).

5.2 Specificatiile sistemului de calcul

CPU:

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           AuthenticAMD
CPU family:          22
Model:               48
Model name:          AMD A6-7310 APU with AMD R
Stepping:            1
CPU MHz:             1691.038
CPU max MHz:         2000,0000
CPU min MHz:         1000,0000
BogoMIPS:            3992.37
Virtualization:      AMD-V
L1d cache:           32K
L1i cache:           32K
L2 cache:            2048K
NUMA node0 CPU(s):   0-3
```

Memorie disponibila

	total	used	free	shared	buff/cache	available
Mem:	3458920	1917308	486132	72804	1055480	1229264
Swap:	2097148	0	2097148			

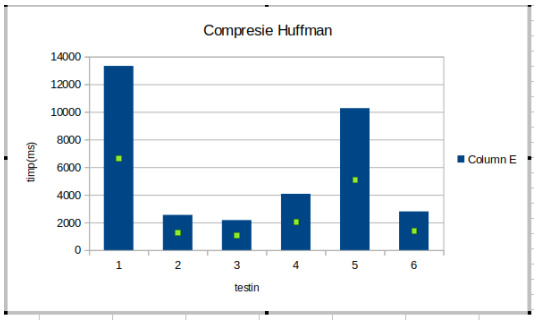
```
MemTotal:        3458920 kB
MemFree:          432824 kB
MemAvailable:    1175332 kB
Buffers:           79880 kB
Cached:           899884 kB
SwapCached:            0 kB
Active:          2088904 kB
Inactive:         586160 kB
Active(anon):    1697340 kB
Inactive(anon):    74132 kB
Active(file):     391564 kB
Inactive(file):   512028 kB
Unevictable:          16 kB
Mlocked:              16 kB
SwapTotal:       2097148 kB
SwapFree:        2097148 kB
Dirty:                12 kB
Writeback:             0 kB
AnonPages:       1695396 kB
Mapped:           454216 kB
Shmem:             76176 kB
Slab:             131836 kB
SReclaimable:      78456 kB
SUnreclaim:        53380 kB
```

```
KernelStack:       14112 kB
PageTables:        51912 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     3826608 kB
Committed_AS:    6647976 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
CmaTotal:              0 kB
CmaFree:               0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      365308 kB
DirectMap2M:     3239936 kB
DirectMap1G:           0 kB
```

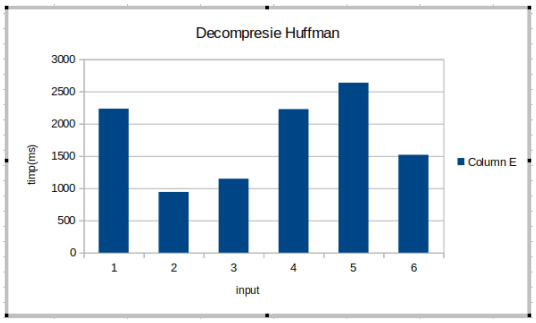
5.3 Rezultatul evaluarii solutiei

Timpul de compresie si de decompresie al algoritmilor:

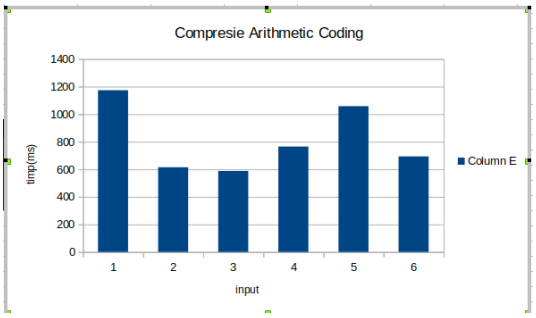
Huffman	Timp compresie1	Timp compresie2	Timp compresie3	Medie
test0	2120	2599	1979	2232.666667
test1	909	935	980	941.3333333
test2	1142	1309	988	1146.333333
test3	2215	1989	2473	2225.666667
test4	2694	2303	2910	2635.666667
test5	1490	1272	1791	1517.666667



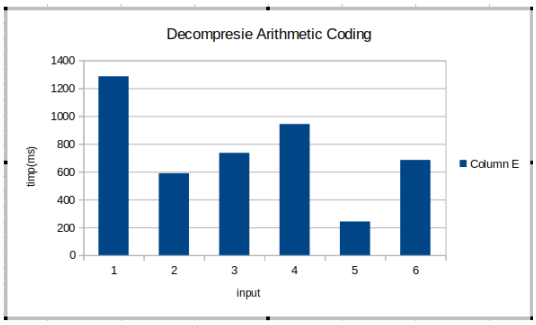
Huffman	Timp decompresie1	Timp decompresie2	Timp decompresie3	Medie
test0	13001	13987	12989	13325.66667
test1	2444	2383	2799	2542
test2	1775	2623	2111	2169.666667
test3	3799	4369	4034	4067.333333
test4	9567	9989	11240	10265.33333
test5	3390	2901	2101	2797.333333



Arith. Coding	Timp compresie1	Timp compresie2	Timp compresie3	Medie
test0	1035	1199	1287	1173.666667
test1	505	648	691	614.6666667
test2	526	604	633	587.6666667
test3	711	769	815	765
test4	981	1141	1052	1058
test5	425	881	774	693.3333333

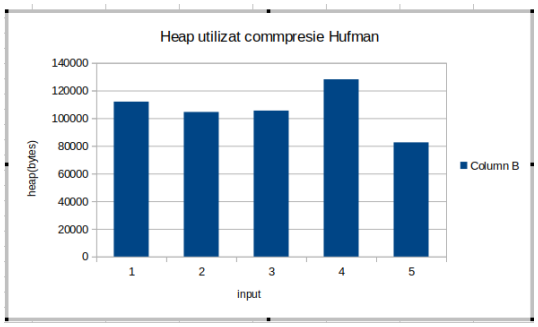


Arith. Coding	Timp decompresie1	Timp decompresie2	Timp decompresie3	Medie
test0	1199	1359	1299	1285.666667
test1	525	636	607	589.3333333
test2	592	821	793	735.3333333
test3	856	1013	959	942.6666667
test4	218	269	239	242
test5	623	741	690	684.6666667

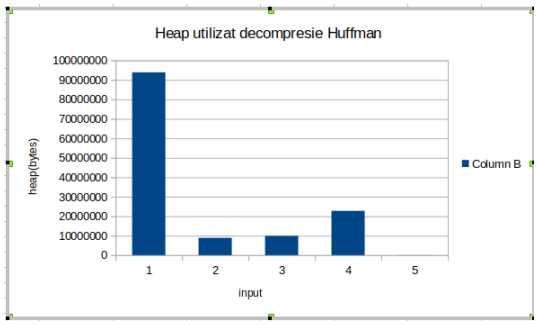


Total heap utilizat in procesele de comprimare si de decomprimare de catre cei doi algoritmi:

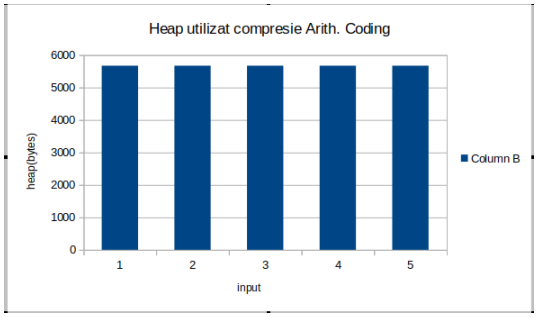
Huffman	Compresie
test0	111950
test1	104500
test2	105477
test3	128097
test4	82504



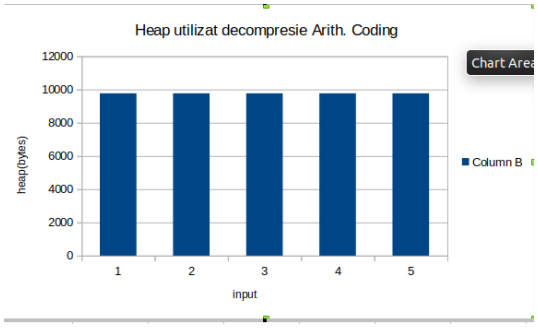
Huffman	Decompresie
test0	93884455
test1	8889150
test2	9913723
test3	22761723
test4	82504



Arith. Coding	Compresie
test0	5672
test1	5672
test2	5672
test3	5672
test4	5672

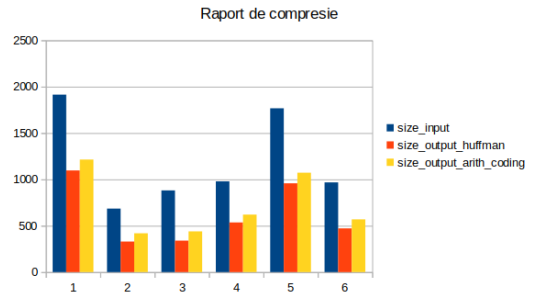


Arith. Coding	Decompresie
test0	9768
test1	9768
test2	9768
test3	9768
test4	9768



Raport de compresie(input/output):

input	size_input	size_output_huffman	size_output_arith_coding
0	1914	1096	1214
1	685	330	419
2	881	340	439
3	979	536	620
4	1767	958	1072
5	968	472	569



5.4 Prezentarea,succinta,a valorilor obtinute pe teste

Analizand tabelele de mai sus se poate preciza ca Arithmetic coding are un timp de compresie si decmopresie ,masurat in msecunde, mult mai bun fata de algoritmul Huffman.

In ceea ce priveste heap-ul utilizat de cele doua programe, huffman foloseste un heap direct proportional cu fisierele de intrare si are diemensiuni mai mari fata de Arithmetic coding, care foloseste un heap constant, indiferent de fisirele de intrare(atat pentru compresie ca si pentru decompresie).

In ceea ce priveste raportul de compresie ,se poate observa ca pentru testele date (special pentru a pune dificutati algoritmilor),Huffman are un raport favorabil fata de Arithmetic coding.In teorie , acest lucru nu se intampla , dar cum am ales alfabete foarte mari ,precizia algoritmului Arithmetic coding trebuie sa fie la fel de mare ,lucru care duce la aceasta consecinta(Huffman comprima mai bine).

6 Concluzii

In practica, daca fisierul de intrare ar avea un alfabet mare cu distribuire uniforma a caraterelor, sau un alfabet mare in care caracterele cu frecventa extrem de mari sunt in numar redus, as alege algoritmul lui Huffman.Dar, daca fisierul de intrare ar avea un alfabet redus ca numar de caractere distincte ,as alege Arithmetic coding.

7 Referinţe

- https://web.stanford.edu/class/ee398a/handouts/papers/WittenACM87ArithmCoding.pdf

- https://arxiv.org/ftp/arxiv/papers/1109/1109.0216.pdf
- www.ijcim.th.org/past_editions/2008V16N3/pp7-64-68-Data_Comprerssion-IJCIM-V16- N3.pdf
- https://www.slideshare.net/ramakantsoni/performance-analysis-of-huffman-and-arithmetic-coding-compression
- https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1126/handouts/220%20Huffman%20Encoding.pdf
- https://www.techiedelight.com/huffman-coding/