Compresia datelor*

Petruc Rareș

Facultatea de Automatică și Calculatoare,
Universitatea POLITEHNICA din București
rares.petruc@stud.acs.upb.ro

Abstract. În cadrul acestui studiu comparativ, vor fi evaluați doi algoritmi cunoscuți de compresie a datelor: Huffman coding, respectiv Lempel-Ziv-Welch (LZW). Scopul temei este de a comprima, respectiv decomprima, fără pierdere de informații, un fișier si de a evalua performanțele obținute de aceștia, în materie de resurse consumate în timpul procesului de comprimare/decomprimare, cât și de raport de compresie a datelor.

Keywords: Compresie a datelor · Decompresie a datelor · fără pierderi · Huffman Coding · Lempel-Ziv-Welch .

* Temă - Analiza Algoritmilor 2020

Table of Contents

1 Introducere

1.1 Descrierea problemei rezolvate

Compresia datelor reprezintă procesul în urma căruia, unul sau mai multe fișiere sunt codificate folosind un număr semnificativ mai redus de biți.

Există două tipuri de compresie a datelor, cunoscute ca: lossless (fară pierdere de informație) și lossy (informația de relevanță mai mică este înlaturată). Conform temei, în cadrul acestui studiu, voi aborda doi algoritmi ce realizează o compresie de tip lossless, anume: Huffman coding și LZW.

1.2 Exemple de aplicații practice pentru problema aleasă

Există numeroase motive pentru care compresia datelor este atât de folositoare, precum:

- Fișierele comprimate vor ocupa mai puțin loc. A lăsa foldere cu fișiere utilizate foarte rar, în forma lor de sine stătătoare nu este deloc practic.
- Transferurile fișierelor nu vor mai fi atât de problematice. Compresia datelor nu rezolvă doar problema timpului de transfer, care este cu mult redus odată ce fișierul este comprimat, dar și probleme legate de faptul că anumite platforme nu permit transferul fișierelor ce depașesc o anumită dimensiune.

1.3 Specificarea soluțiilor alese

Huffman coding: Acest algoritm de compresie realizează o codificare de lungime variabilă (denumire cunoscută și sub numele de variable-length code), ceea ce înseamnă că fiecărui simbol din sursă îi corespunde un număr variabil de biți. În același timp, niciun cod atribuit vreunui simbol nu reprezintă prefixul codului asociat unui alt simbol, ceea ce determină algoritmul să fie de tipul prefix-free. Îmbinate, ambele trasături precizate mai sus, variable-length code și prefix-free, determină ca decodificarea să fie realizată fară ambiguitate.

LZW: Compresia LZW folosește un tabel de coduri, ce are de obicei un număr de 4096 de intrări. Codurile de la 0 la 255 sunt atribuite unui singur byte corespunzător din fișierul de intrare. Este un algoritm simplu și eficient, întrucât nu depinde de probabilitatea apariției unui anumit simbol, ci numai de șirurile codificate anterior. Totuși, nu este întotdeauna o soluție bună pentru compresie, mai ales când avem de a face cu un șir de caractere scurt și/sau divers.

1.4 Criteriile de evaluare pentru soluția propusă

Având în vedere [2,4], dar și în urma rulării pe cod a unor exemple, aș vrea să pun în evidență situațiile în care mă aștept că un anumit algoritm realizează compresia mai bine. Am aplicat algoritmii atât pe fișiere text, cât și pe imagini (fișiere de tip bmp), incercând să creez o oarecare analogie între rezultate astfel comprimând pe rând:

- un text ce conține un singur caracter de un număr mare de ori, respectiv o imagine de o singură culoare;
- un text ce conține text repetitiv (de forma "ABCABCABC..."), respectiv o imagine ce conține dungi verticale de diferite culori (pentru că așa voi face citirea fișierului bmp, parcurgând pe orizontală);
- un text ce conține foarte multe cuvinte ce se repetă, respectiv o imagine simetrică;
- un text divers ce conține terminologii din diferite domenii (respectiv o parte a textului să fie despre un subiect de chimie, o parte despre fizică, alta despre inteligentă artificială, etc.) și o imagine foarte diversă, complexă.

În final, pentru a valida corectitudinea, voi decomprima fișierele și le voi compara cu cele de input, analizând bineînțeles și raportul de compresie obținut, iar pentru evaluarea performanței mă voi referi atât la timpul luat pentru execuția programului, cât și la memoria consumată de program în timpul execuției.

2 Prezentarea soluțiilor

2.1 Descrierea modului în care funcționează algoritmii aleși

Huffman coding: Fiecare simbol din cadrul unui fișier va avea asociat o greutate, reprezentând numărul de apariții ale acestuia în cadrul fișierului. În continuare, vom folosi un arbore binar pentru a implementa algoritmul. La început, vom avea o multitudine de arbori stocați într-un minheap, egali la număr cu numărul simbolurilor unice din fișier. Acești arbori vor avea câte un singur nod, ce va conține simbolul și greutatea acestuia. Se vor alege arborii cu cele mai mici greutăți, să le spunem pentru acum A1 și A2. Din aceștia se va crea un nou arbore a cărui greutate din radăcină este egală cu suma greutăților lui A1 și A2 și a cărui subarbori vor fi: cel stâng - A1 și cel drept - A2. Se va repeta acest pas până când se va obține un singur abore, numit și Arborele lui Huffman sau în engleză, Huffman Tree. Prin convenție, la coborârea în arbore pe subarborele drept se va adăuga bitul 1, respectiv în cazul invers 0, până când se ajunge la frunze (unde se vor afla simbolurile unice din fișier). Decompresia realizează trecerea prin fișierul codificat și folosindu-se de biții întâlniți, realizează concomitent câte o coborâre în arbore pentru fiecare secvență de biți, până când se ajunge la o frunză (simbolul corespunzător secvenței de biți proaspăt parcurse). In fig. 1 este evidentiat un pseudocod al algoritmului de construire pentru arborele Huffman.

```
Huffman(C)
    n = |C|
    Q = C       //Store characters in C in a min-heap, Q

    for i = 1 to n-1
        z = Allocate-Node()                 //Create an empty node
        z.left = x  = Extract-Min(Q)        //Node with lowest freq
        z.right = y = Extract-Min(Q)        //Next lowest freq
        z.freq = x.freq + y.freq
        Insert(Q, z)                    //Insert z in appropriate place in Q
    return Extract-Min(Q)     //Last remaining node in Q is root of tree
```

Fig. 1. Pseudocodul pentru construirea arborelui Huffman.

LZW: În cadrul acestui algoritm, se parcurge șirul de intrare, acesta fiind procesat în felul următor: se actualizează un tabel în mod constant ce realizează legătura dintre cele mai lungi cuvinte întâlnite și codurile asociate. În același timp cu parcurgerea șirului, are loc și înlocuirea cuvintelor întâlnite până atunci în cadrul tabelului, cu codurile corespunzătoare. Decompresia LZW creeaza același tabel cu aceleași codificări. Este necesar un tabel cu primele 256 de intrări inițializate cu caractere unice. Astfel, decodificarea se realizează citind codurile primite ca input și traducându-le folosind tabelul creat. Iată în fig. 2, 3 pseudocodul compresiei și decompresiei LZW, dar mai ales similaritatea operațiilor realizate.

```
STRING = get input symbol
WHILE there are still input symbols DO
   SYMBOL = get input symbol
   IF STRING + SYMBOL is in the STRINGTABLE THEN
       STRING = STRING + SYMBOL
   ELSE
       output the code for STRING
       add STRING + SYMBOL to STRINGTABLE
       STRING = SYMBOL
   END
END

output the code for STRING
```


Fig. 2. Pseudocodul pentru compresia LZW.

```
table initialization with single character strings
prev = first input code;
output translation of prev
WHILE not end of input stream
  now = next input code
  IF now is not in the string table THEN
    S = translation of prev
    S = S + C
  ELSE    S = translation of now
  OUTPUT S
  C = first character of S
  insert prev + C to string table
  prev = now
END WHILE
```

[Pseudocode for LZW decompression]

Fig. 3. Pseudocodul pentru decompresia LZW.

2.2 Analiza complexității soluțiilor

Huffman: are o complexitate O(n log n) în general, excepție făcând cazul în care nodurile sunt deja sortate după frecvență (numărul de apariții în fișier), caz în care complexitatea este O(n), unde n reprezintă numărul simbolurilor ce trebuie codificate. Complexitatea provine din faptul că extracția elementului minim din minheapul în care se află este O(log n), pentru că în spate se apelează și un minHeapify pentru restaurarea minheapului. Aceată extracție se realizează de 2 * (n - 1) ori, motiv pentru care complexitatea finală devine O(n log n).

LZW: are o complexitate O(n), întrucât fiecare byte din fișierul ce trebuie să fie comprimat este parcurs o singură dată, iar operațiile care se realizează pentru fiecare byte sunt în număr finit și sunt constante. Aceeași complexitate este obținută și pentru decodificarea fișierului codificat din aceleași considerente.

2.3 Prezentarea principalelor avantaje și dezavantaje pentru soluțiile luate în considerare

	Avantaje	Dezavantaje
Huffman	– Simbolurile care apar cel mai frecvent în fișierul ce se dorește a fi comprimat vor avea un corespondent în binar de o dimensiune mult mai mică. – Nicio codificare a vreunui simbol din cadrul fișierului primit ca input nu este prefixul unui alt simbol	– Codificarea Huffman este destul de lentă, întrucât spre deosebire de LZW, realizează codficarea propriu-zisă a fișierului într-un pas diferit față de codificarea simbolurilor. – Un alt dezavantaj al codificării Huffman este că șirurile binare din fișierul comprimat au lungimi diferite. Din acest motiv, la pasul de decomprimare, software-ul ce decodifică fișierul nu poate să determine când s-a ajuns la ultimul bit de date integru sau dacă datele codificate sunt corupte.
LZW	– Cu LZW, compresia fișierului primit ca input se realizează printr-o singură trecere – Pentru aplicarea LZW, nu trebuie să se cunoască informații despre datele din fișier, spre deosebire de Huffman, unde este necesar să se știe frecvența simbolurilor. – Obține performanțe foarte bune pentru fișierele ce conțin date ce se repetă	– Deși la o primă vedere pare destul de simplu, implementarea algoritmului devine anevoioasă în ceea ce privește gestionarea dicționarului. – Căutarea patternurilor în dicționar devine destul de lentă, odată ce acesta crește în dimensiune.

3 Evaluare

3.1 Descrierea modalității de construire a setului de teste folosite pentru validare

Cum am decis să lucrez pe imagini și texte similare din punct de vedere al conținutului, am căutat imagini potrivite pentru ceea ce am specificat la 1.4,

pe care le-am transformat din formatul lor (png, jpg, jepg) în 24-bit Bitmap (.bmp) în urma unor cropuri pe care le-am considerat necesare în a evidenția performanțele celor 2 algoritmi. Cât despre fișierele de tip text, pentru primele 3 fișiere de intrare am realizat de unul singur conținutul, iar pentru cel de al 4-lea, am copiat 3 texte de pe internet, fiecare aparținând unui alt domeniu de specialitate pentru a face fișierul cât mai divers.

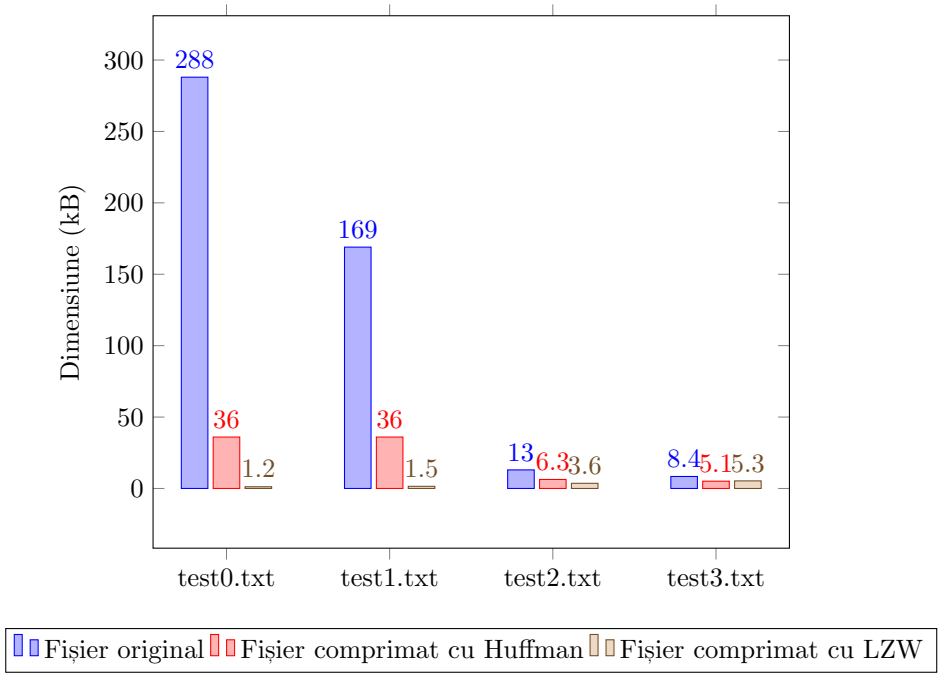
3.2 Specificațiile sistemului de calcul pe care au fost rulate testele

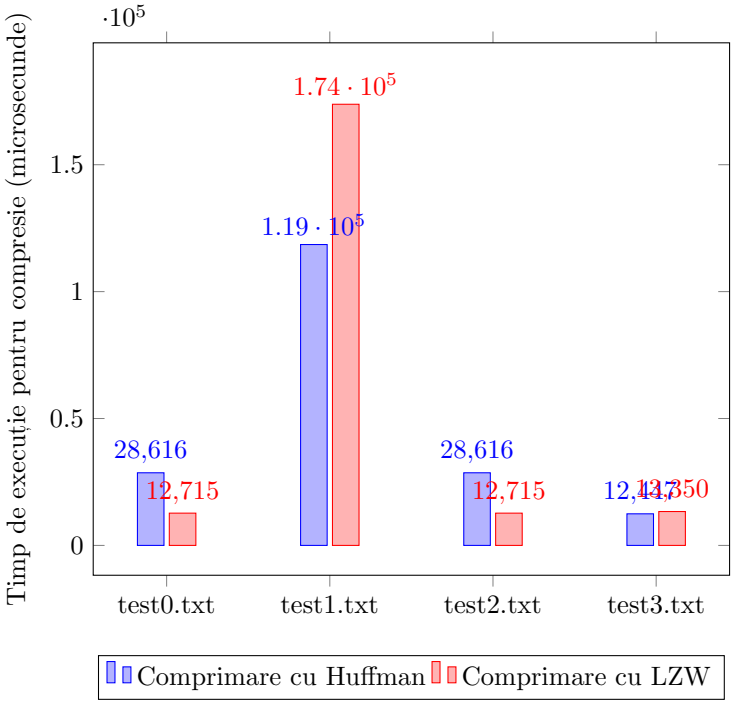
Testarea algoritmilor a fost realizată de pe un laptop Lenovo ideapad 300, pe o mașină virtuală cu sistem Linux, distribuție Ubuntu, versiunea 20.04.1 LTS, cu următoarele specificații ale sistemului: procesor Intel Core i7-6500U, cu frecvență de 2.5 GHz și 4GB memorie RAM.

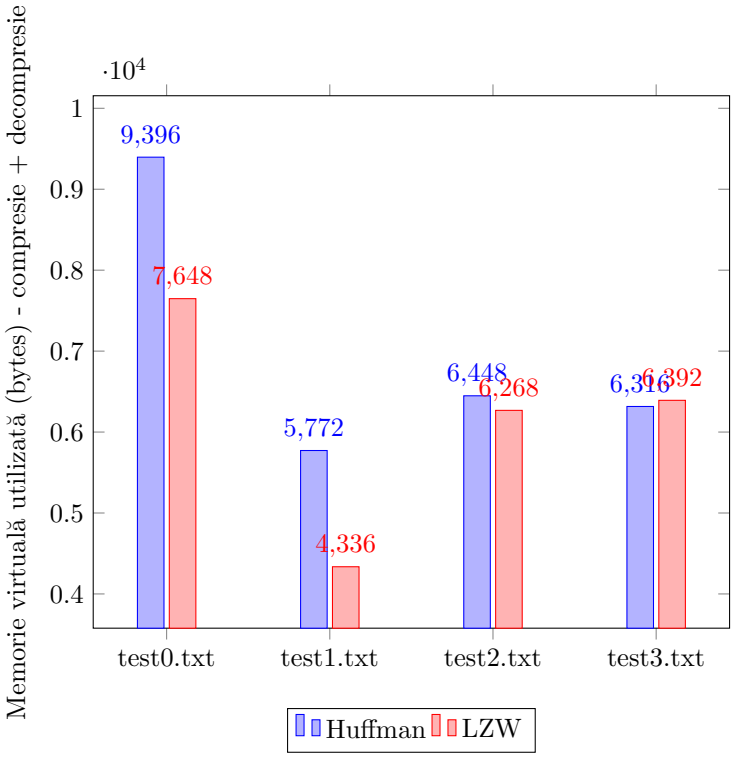
3.3 Ilustrarea folosind grafice a rezultatelor evaluării soluțiilor pe setul de teste

Pentru măsurarea timpului de execuție am folosit biblioteca chrono, iar pentru măsurarea memoriei virtuale utilizate m-am folosit de headerul "unistd.h".

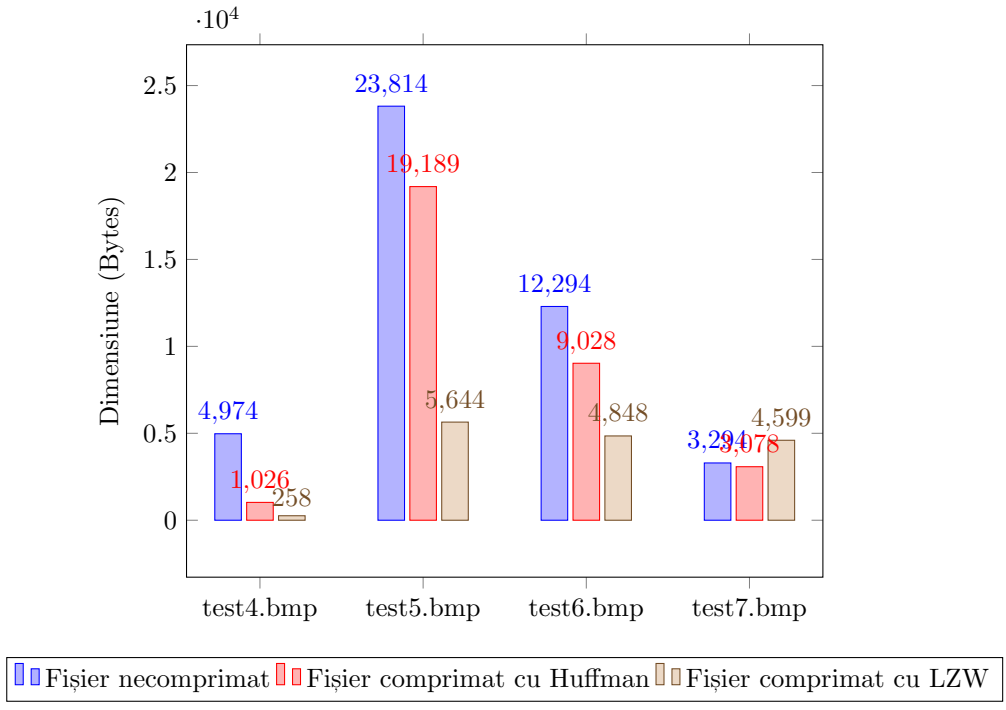
În continuare, se pot observa performanțele celor doi algoritmi pentru fișierele de tip text din folderul de input:

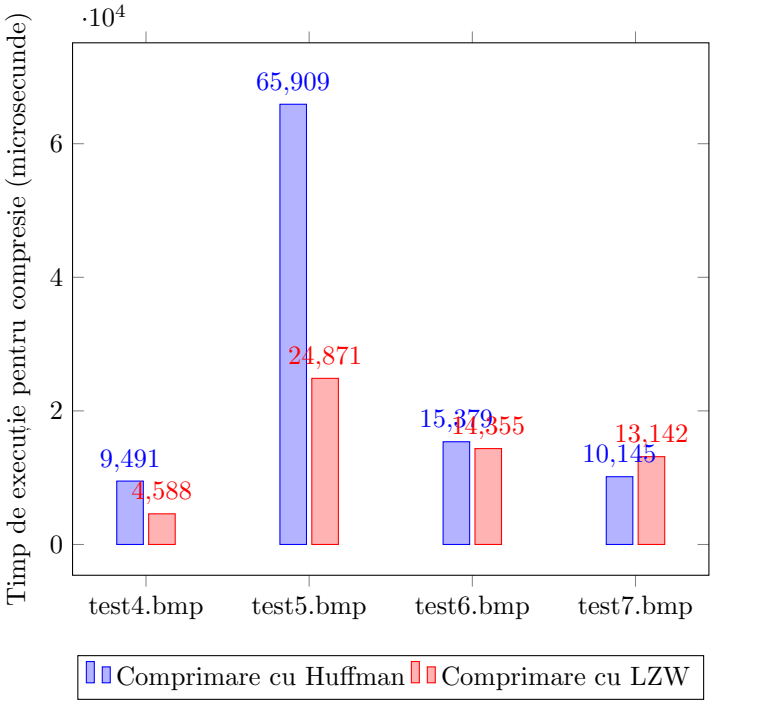


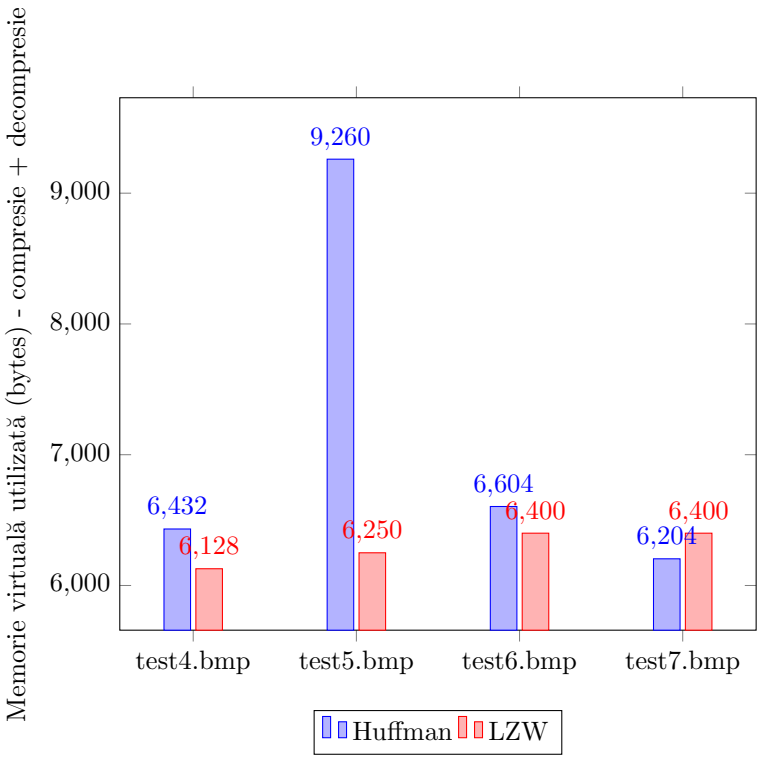

Timp de execuție pentru compresie (microsecunde)
·10^5
1.74 · 10^5
1.19 · 10^5
28,616
12,715
28,616
12,715
12,350
0
0.5
1
1.5
test0.txt
test1.txt
test2.txt
test3.txt
Comprimare cu Huffman
Comprimare cu LZW


Memorie virtuală utilizată (bytes) - compresie + decompresie
·10^4
9,396
7,648
5,772
4,336
6,448
6,268
6,392
0.4
0.5
0.6
0.7
0.8
0.9
1
test0.txt
test1.txt
test2.txt
test3.txt
Huffman
LZW

Iată și performanțele celor doi algoritmi pentru fișierele de tip bmp din folderul de input:




$\cdot 10^4$
Timp de execuție pentru compresie (microsecunde)
0
2
4
6
65,909
24,871
15,379
14,355
13,142
10,145
9,491
4,588
test4.bmp
test5.bmp
test6.bmp
test7.bmp
Comprimare cu Huffman
Comprimare cu LZW


Memorie virtuală utilizată (bytes) - compresie + decompresie
6,000
7,000
8,000
9,000
9,260
6,432
6,128
6,250
6,604
6,400
6,400
6,204
test4.bmp
test5.bmp
test6.bmp
test7.bmp
Huffman
LZW

3.4 Prezentarea succintă a valorilor obținute pe teste

În ceea ce privește graficele ce evidențiază dimensiunea inițială a fișierului în comparație cu dimensiunile fișierului comprimat, se poate observa atât pentru fișierele de tip text, cât și pentru cele de tip bmp, cum o dată ce complexitatea imaginii crește și apar elemente tot mai diverse, LZW scade repede în performanță, iar Huffman pare să se descurce onorabil. Interesant este că în cadrul imaginii test7.bmp, LZW nu numai că nu comprimă bine, dar nu o face deloc, întrucât dimensiunea fișierului comprimat prin LZW este chiar mai mare decat cea a fișierului original. Acest lucru se datorează imaginii, care este foarte mică și foarte diversă.

Graficele ce prezintă timpul de execuție, respectiv memoria virtuală utilizată sunt similare, ceea ce este și intuitiv. În implementare, am lucrat cu structuri de date destul de mari, ceea ce înseamnă, pe lângă multă memorie ocupată, și un timp de execuție mai mare (pentru a realiza codificări în arborele Huffman sau a căuta un pattern într-un dicționar aproape plin în cazul LZW).

4 Concluzii

Răspunusl la întrebarea care algoritm face mai bine compresia ar fi fără doar și poate "depinde". Din grafice observăm inclusiv rapoarte de compresie uriașe de genul 240 pentru test0.txt în cazul compresiei cu LZW, dar și situații mai puțin plăcute, ca în cazul lui test7.bmp unde nu se realizează deloc compresia tot cu LZW.

Pentru Huffman Coding avem rapoarte de compresie cuprinse între 1.07 și 8, raportul crescând odată cu existența unor simboluri ce se folosesc mult mai des decât restul din cadrul unui fișier.

Pentru LZW se pot observa rapoarte de compresie cu valori de la 1.58, ignorând cazul în care nu s-a realizat compresie deloc, până la 240. Cu siguranță, pentru fișierele ce conțin informație redundantă sau secvențe de patternuri repetitive, aș recomanda să fie folosit LZW.

5 Bibliografie

References

1. Generalități despre compresia datelor, `https://www.winzip.com/win/en/learn/file-compression.html`. Ultima dată de acces: 07-Noiembrie-2020
2. Discuție de pe StackOverFlow (HUFFMAN vs LZW), `https://cs.stackexchange.com/questions/50305/huffman-coding-vs-lzw-algorithm/74491`. Ultima dată de acces: 07-Noiembrie-2020
3. Huffman Coding, `https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/`. Ultima dată de acces: 14-Noiembrie-2020
4. Avantaje LZW peste Huffman, `https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/`. Ultima dată de acces: 07-Noiembrie-2020

5. Pseudocodul Huffman, https://slideplayer.com/slide/17068791/. Ultima dată de acces: 03-Decembrie-2020
6. Pseudocodul LZW, http://www0.cs.ucl.ac.uk/staff/B.Karp/3007/s2018/lectures/3007-lecture9-lzw.pdf. Ultima dată de acces: 03-Decembrie-2020