



UNIVERSITATEA DIN  
BUCUREŞTI



FACULTATEA DE  
MATEMATICA ŞI  
INFORMATICA

SPECIALIZAREA INFORMATICĂ

Lucrare pentru cursul de Prelucrarea Semnalelor

## ALGORITMII LZ77 & LZ78

Student  
Maşala Doru-Andrei  
Grupa 331

Profesor coordonator  
Cristian Rusu

Bucureşti, februarie 2026

# Cuprins

<b>1</b>	<b>Introducere</b>	<b>3</b>
1.1	State of the art . . . . .	3
<b>2</b>	<b>Algoritmii LZ77 și LZ78</b>	<b>5</b>
2.1	LZ77 . . . . .	5
2.2	LZ78 . . . . .	7
2.3	Comparație . . . . .	9
<b>3</b>	<b>Concluzii</b>	<b>11</b>
	<b>Bibliografie</b>	<b>12</b>

# Capitolul 1

## Introducere

În paradigma contemporană a tehnologiei informației, procesarea volumelor masive de date impune optimizarea continuă a mecanismelor de stocare și transmisie. Astfel, odată cu creșterea exponențială a volumului de date la nivel global, a apărut și necesitatea de a reprezenta o bucată de informație în cât mai puțini biți, fără a pierde conținutul esențial al acesteia. Problema fundamentală a compresiei datelor, aşa cum a fost formulată de Claude Shannon în lucrarea sa din 1948, "A Mathematical Theory of Communication", rezidă în identificarea și eliminarea redundanței statistice.

În teoria compresiei, se disting 2 tipuri majore de algoritmi de compresie: lossless (prin care nu se pierde informație în urma compresiei) și lossy (prin care se pierd anumite bucăți de informație mai puțin importante). Filozofia algoritmilor de compresie implică un trade-off între spațiul de stocare și timpul de procesare: reducerea volumului de date necesită resurse computaționale și timp pentru procesare. Designul acestor scheme urmărește echilibrarea a trei factori cheie: rata de compresie, gradul de distorsiune (în cazul compresiei lossy) și complexitatea algoritmică necesară codării și decodării.

### 1.1 State of the art

O componentă relevantă când vine vorba despre orice tip de algoritm de comprimare îl reprezintă entropia lui Shannon. Aceasta spune că niciun fișier nu poate fi comprimat dincolo de limita entropiei sale fără a pierde informație.

$$H = - \sum p(x) \log p(x) \tag{1.1}$$

Pe scurt, entropia măsoară cât de multă informație aduce apariția unui simbol. Dacă acesta are probabilitatea ridicată, înseamnă ca are entropia scăzută (pentru că te aștepți să apară, nu aduce multă "informație" NOUA sa apariție).

În zilele noastre, orice algoritm lossless folosit la scară largă (fie el Snappy, LZ4, zstd sau gzip) încearcă să se apropie cat mai mult de această valoare H (adică încearcă să reducă

redundanță). În cele ce urmează, voi prezenta doi algoritmi care, deși rudimentari, stau la baza mulțor algoritmi contemporani de compresie (precum DEFLATE care este folosit pentru PNG sau ZIP; GIF, 7ZIP și altele).

Este vorba de LZ77 și LZ78, propuși de Abraham Lempel and Jacob Ziv în 1977, respectiv 1978]. Ambii algoritmi fac parte teoretic din categoria encriptorilor pe bază de dicționar, care funcționează căutând match-uri între textul care trebuie comprimat și o structură de date care conține bucăți din textul necomprimat (în cazul lui LZ77, un buffer de căutare și o fereastră glisantă, respectiv un dicționar efectiv în cazul lui LZ78).

# Capitolul 2

## Algoritmii LZ77 și LZ78

### 2.1 LZ77

LZ77, supranumit și Lempel-Ziv 1, marchează începutul erei compresiei adaptive, fiind algoritmul care a demonstrat că datele pot fi comprimate eficient prin simpla referențiere a trecutului recent al fluxului de caractere. LZ77 funcționează înlocuind aparițiile ale substring-urilor care se repetă cu referința unei singure apariții în stream-ul de date ne-comprimat. Înlocuirea este făcută cu o pereche denumită length-distance, reprezentând câte pozitii trebuie să mergem în spate pentru a regăsi de unde începe substring-ul, respectiv câte pozitii trebuie să avansăm de la început pentru a reconstrui cuvântul căutat. Pentru a reuși să găsească aceste repetiții, algoritmul ține evidența ultimilor 1, 2, 32KB (în funcție de preferință) necomprimăți. Aceasta se numește de regulă și sliding-window, întrucât la fiecare pas, fereastra se mută câte o poziție.

Mai jos, voi prezenta pseudocodul implementat de mine:

```
cursor <- 0
cat timp cursor < lungime input
    // findLongestMatch cauta subsirul cel mai lung pe care il poate
    // regasi in fereastra glisanta, care coincide cu subsirul care
    // porneste de la pozitia actuala a cursorului (pentru eficienta,
    // am ales sa caut incepand cu primele 3 caractere din subsir)
    (distanta, lungime_match) <- findLongestMatch(input, cursor)
    index_urmar <- cursor + lungime_match
    daca index_urmar < lungime_totala
        caracter_urmar <- input[index_urmar]
        altfel caracter_urmar <- 0
        output(distanta, lungime_match, caracter_urmar)
        cursor <- cursor + lungime_match + 1
    scrie lungime_totala, output in fisier
```

Secretul eficienței algoritmului LZ77 rezidă în potrivirea parametrilor într-un mod

optim astfel încât să existe o oarecare balanță între timpul efectiv de compresie și eficiența acesteia. Astfel, se observă că variabilele care influențează eficiența algoritmului sunt window size-ul (informația în care algoritmul caută “în trecut” asemănări) și lookahead bufferul (cât de mare este string-ul pe care încerc să-l regăsească). Pentru window, am ales valori de 1, 2, 32 și 64 de KB, iar pentru lookahead buffer 15, 63, respectiv 127 de bytes.

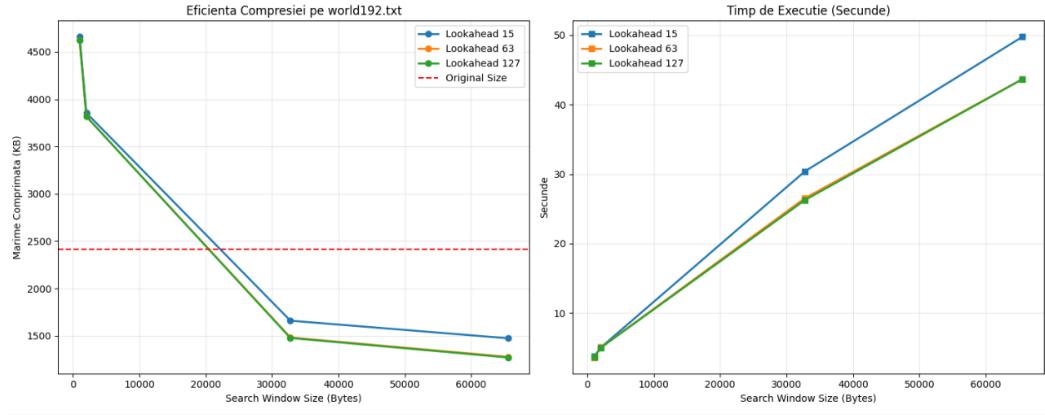


Figura 2.1: LZ77 worlds192.txt

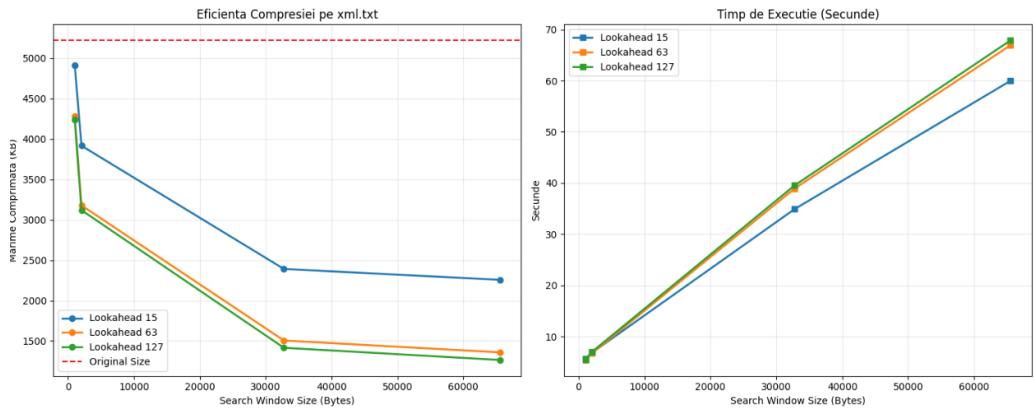


Figura 2.2: LZ77 xml.txt

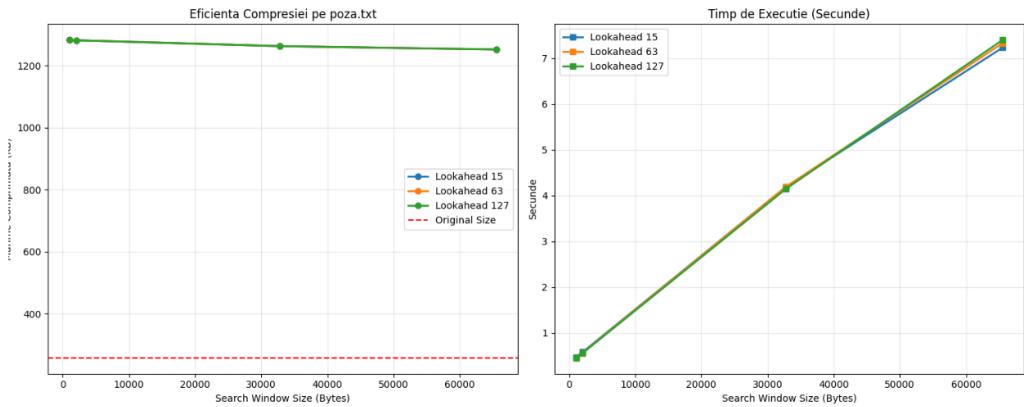


Figura 2.3: LZ77 poza.txt

Cât despre input-uri, am ales un fișier în limbaj natural, denumit world192.txt preluat din Canterbury Corpus, un fișier xml din Silesia Corpus și o imagine jpg stock. Pentru primele 2 fișiere, cum era și de așteptat, algoritmul se descurcă destul de bine în a comprima informația, întrucât în acestea există redundanță, spre deosebire de imaginea care are multe valori aleatoare, chiar mărind dimensiunea fișierului comprimat. De notat este de asemenea că, valorile lookahead-ului de 63 și 127 dau rezultate apropiate în ceea ce privește nivelul de compresie, având în vedere că este foarte greu să se regăsească repetiții de mai mult de 2-3 cuvinte într-un text (cel mai lung cuvânt în limba engleză este de 45 de litere).

## 2.2 LZ78

Spre deosebire de “fratele” său, care caută potriviri într-o fereastră glisantă, algoritmul Lempel-Ziv 2 memorează secvențele noi într-un dicționar indexat, emițând token-uri formate dintr-un index al unui prefix deja cunoscut și un caracter care completează o nouă unitate.

Pe scurt, algoritmul încearcă să găsească cea mai lungă secvență de caractere care există deja în dicționar. Odată identificată, el emite un token format dintr-o pereche: (indexul prefixului, următorul caracter). Imediat după emitere, noua frază (prefixul + caracterul nou) este adăugată în dicționar sub un index nou.

Pseudcodul implementat de mine este:

```
Initializez dicționarul cu sirul vid pe pozitia 0, cursor <- 0 și
size_dict <- 1
cat timp cursor < lungime_input
    string_curent <- ""
    ultimul_match <- 0
    // Aici caut cel mai lung prefix din dicționar
```

```

cat timp (string_curent + caracter[cursor]) exista in dictionar
    string_curent <- string_curent + caracter[cursor]
    ultimul_match <- index(string_curent)
    cursor += 1

// Verific daca nu am terminat fisierul
daca cursor < lungime_input
    next_char <- input[cursor]
    cursor += 1
altfel
    next_char <- 0

scrie(ultimul_match, next_char)

daca size_dict < 65535
    adauga (string_curent + next_char) in dictionar la pozitia
        size_dict
    size_dict += 1
altfel
    goleste dict
    size_dict = 1

```

Deși mult mai robust în ceea ce privește “hyperparameter tuning-ul” (spre deosebire de LZ77), toată filozofia algoritmului stă în abordarea dicționarului. În testele mele, am încercat mai multe metode de a optimiza utilizarea structurii de date, ajungând la următoarele concluzii:

Pentru anumite fișiere relativ mici, un dicționar cu un număr maxim de 65535 intrări a fost suficient (spre exemplu pentru cazul “bible.txt” din Canterbury Corpus de aproximativ 4000 KB). Apoi mi-am îndreptat testele către niște seturi mai ample, cum ar fi executabilul de mozilla 1.0 din Silesia Corpus, de aproximativ 50000 KB. În acest caz, dicționarul se umplea foarte repede și de la un moment dat compresia se oprea. Așa că am încercat să cresc dimensiunea dicționarului, implicit și dimensiunea în bytes a encodării (de la 3 bytes, la 5 bytes, având o dimensiune de aproximativ 4 miliarde de intrări). Totuși, deși aveam destule intrări, encriptarea era prea lungă. Așa că am revenit la prima variantă, doar că în loc să mă opresc din a encoda atunci când dicționarul se umple, am ales să reinitializez cu totul dicționarul, pentru a permite algoritmului să învețe noi secvențe din text, fapt care a ajutat foarte mult reducerea dimensiunii file-ului.

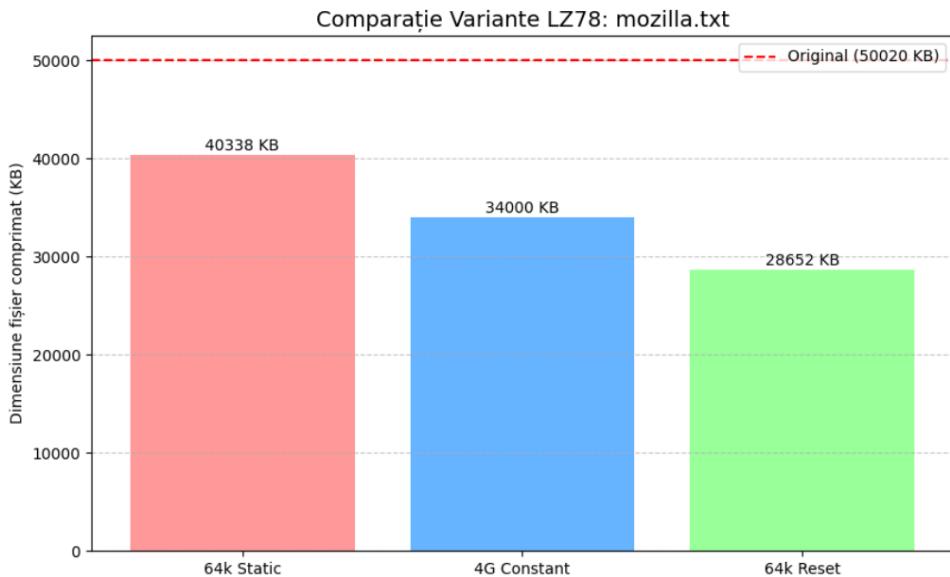


Figura 2.4: Comparatie LZ78

## 2.3 Comparatie

Pentru a vedea dacă unul dintre cei 2 algoritmi este mai puternic decât celălalt, am ales să preiau din testele mele varianta care s-a mulat cel mai bine pe seturile de date: pentru LZ77 am ales un sliding window de 32 de KB și un lookahead buffer de 63 de bytes, iar pentru LZ78 am optat pentru dicționarul de 65535 intrări care atunci când se umple este reinițializat. Am rulat ambii algoritmi pe fișierele pe care am rulat la pasul 3 algoritmul LZ77 și am mai adăugat încă 2 fișiere: lorem.txt (în care am paste-uit lorem ipsum de 430,2 KB) și log.txt (în care am paste-uit fraza “ACESTA ESTE UN STRING EXTREM DE LUNG SI REPETITIV CARE VA DISTRUGE LZ78 DEOARECE ARE Multe CARACTERE SI LZ78 TREBUIE SA LE INVETE PE FIECARE IN PARTE LITERA CU LITERA PANA FACE FRAZA COMPLETA”). Am observat din teste că LZ78 are tendința de a se decurca puțin mai bine pe fișierele date aşa că ultimele 2 au fost adăugate pentru a arăta punctele slabe (și implicit cele puternice) ale lui Lempel-Ziv 2 (respectiv Lempel-Ziv 1).

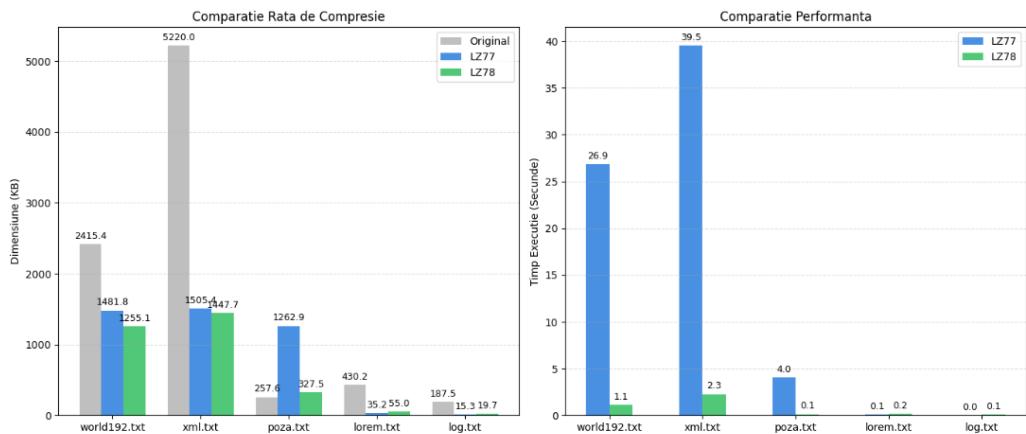


Figura 2.5: LZ77 vs LZ78

# Capitolul 3

## Concluzii

În ceea ce privește cei 2 algoritmi, ambii depind foarte tare de alegerea corectă a parametrilor (dimensiunea sliding window-ul pentru LZ77 și dimensiunea dicționarului pentru LZ78). Puși în paralel, în ceea ce privește compresia datelor, aceștia au un comportament similar: funcționează bine pe informație redundantă (world192.txt și xml.txt) și dau greș când vine vorba de date aleatoare (poza.txt). Totuși, se observă faptul că LZ78 are o tendință de a comprima puțin mai bine datele pe majoritatea testelor făcute de mine.

Însă, se observă că în zone în care există foarte multe repetiții de fraze lungi (cum este în lorem.txt și log.txt), LZ77 face o treabă mai bună decât "fratele" său. De aici, putem trage concluzia că LZ78 se pricepe mult mai bine pe a comprima date care se repetă global, în timp ce LZ77 triumfă când vine vorba de repetiții locale (lucru care reiese direct și din cum funcționează algoritmii, LZ77 folosind o fereastră glisantă fixă, iar LZ78 utilizând un dicționar care se măreste până la refuz, apoi se reinicializează).

În plus, se observă că algoritmul LZ77 este observabil mai lent pentru fișiere mai mari. O primă îmbunătățire a implementării ar proveni chiar din acest fapt: ar fi mult mai eficientă o implementare în C/C++. Deși folosesc funcția `find` din librăria numpy, din cauza while-urilor folosite, se creează un bottleneck care trage în jos viteza algoritmului. Totodată, ar fi utilă implementarea unui hash table pentru a prelua direct poziția unde am mai găsit caracterele, fără a mai trece secvențial prin toată fereastra. De asemenea, în ceea ce privește algoritmul LZ78, în loc să șterg tot dicționarul, ar fi utilă o abordare de tip least recently used, ștergând doar string-urile care nu au fost regăsite de mult timp.

În concluzie, nu există un algoritm care ar trebui folosit mai mult decât celălalt ; este foarte importantă natura input-ului și resursele de care dispunem. În cazul unor fișiere cu limbaj natural în care se repetă multă informație locală ar fi mai indicată o abordare de tip LZ77, iar dacă avem informații mai ample unde informația se repetă la nivel global, ar fi ideală o abordare de tip LZ78.

# Bibliografie

- [1] „Adaptive data compression system with systolic string matching logic”, 5532693, US Patent, 1996.
- [2] *Canterbury Corpus*, URL: <https://corpus.canterbury.ac.nz/descriptions/>.
- [3] IEEE Global History Network, *Milestones: Lempel-Ziv Data Compression Algorithm, 1977*, Institute of Electrical și Electronics Engineers, 22 Iul. 2014, (accesat în 9.11.2014).
- [4] *Lossless Data Compression: LZ78*, URL: [cs.stanford.edu](http://cs.stanford.edu).
- [5] *Silesia Corpus*, URL: <https://sun.aei.polsl.pl/~sdeor/index.php?page=silesia>.
- [6] Jacob Ziv și Abraham Lempel, „A Universal Algorithm for Sequential Data Compression”, în *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343, DOI: [10.1109/TIT.1977.1055714](https://doi.org/10.1109/TIT.1977.1055714).
- [7] Jacob Ziv și Abraham Lempel, „Compression of Individual Sequences via Variable-Rate Coding”, în *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536, DOI: [10.1109/TIT.1978.1055934](https://doi.org/10.1109/TIT.1978.1055934).