Algoritmi sažimanja bez gubitaka

Pešut, Matea

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Matea Pešut

ALGORITMI SAŽIMANJA BEZ GUBITAKA

Diplomski rad

Voditelj rada:
doc. dr. sc. Goranka Nogo

Zagreb, rujan, 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Posvećujem ovaj rad svojoj obitelji. Bez njihove neizmjerne podrške i poticaja svaka prepreka bila bi teža, a svaki sretan trenutak manje poseban. ♡

Sadržaj

Uvod

U doba obilježeno naglim porastom količine digitalnih podataka, koji proizlaze iz različitih aspekata ljudskog djelovanja, javljaju se izazovi povezani s učinkovitom pohranom, prijenosom i obradom podataka. Od ogromnih skupova znanstvenih podataka i multimedijskih sadržaja pa do poslovnih transakcija i osobnih komunikacija, upravljanje informacijama postalo je središnji element današnjeg društva. Usprkos svim izazovima, to je istovremeno i prilika za razvoj inovativnih tehnologija za kompresiju podataka.

Upravo u ovom kontekstu, algoritmi sažimanja bez gubitaka imaju poseban značaj jer omogućuju kompresiju podataka bez ugrožavanja integriteta i vjernosti izvornih informacija. Ova tehnika je od presudnog značaja za rješavanje problema preopterećenosti resursa, kao što su kapacitet pohrane, brzina prijenosa i propusnost mrežnih kanala. Daljnje istraživanje i razumijevanje ovih algoritama neophodno je za optimizaciju upravljanja podacima u doba koje karakterizira obilje informacija.

Cilj ovog diplomskog rada je pružiti pregled najpozantijih algoritama sažimanja bez gubitaka. Rad se sastoji od tri poglavlja i zaključka, U prvom su navedeni osnovni pojmovi teorije sažimanja. U drugom je dan pregled najpoznatijih algoritama sažimanja bez gubitaka te su analizirane njihove karakteristike. U trećem poglavlju opisana je implementacija tih algoritama te su analizirani rezultati dobiveni na standardnim testnim primjerima. U zaključku se navode područja primjene opisanih algoritama.

Poglavlje 1

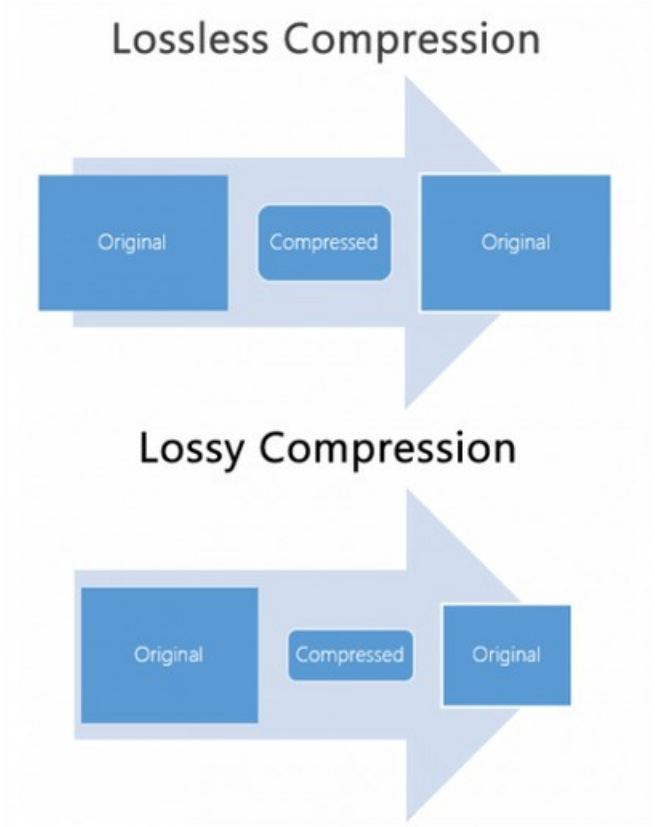
Sažimanje podataka

Proces u kojem se smanjuje broj bitova potrebnih za zapis objekata u memoriji računala se naziva sažimanje podataka. Algoritmi sažimanja dijele se u dvije velike skupine, algoritme bez gubitaka i algoritme s gubitcima. Pri kompresiji s gubitcima, određeni podaci mogu se izgubiti čime se izvorni oblik više ne može rekonstruirati, odnosno obnoviti. Učinak na izvorne podatke prikazan je na slici 1.1.

Trenutno postoji jako velik broj algoritama za sažimanje podataka koji se mogu podijeliti u pet velikih skupina, a to su variable-length codes (VLC), statističko kodiranje, univerzalno kodiranje, kontekstno modeliranje i višekontekstno modeliranje, pri čemu je svaki algoritam ili bez gubitaka (*lossless*) ili s gubitcima (*lossy*). Algoritme iz više skupina možemo kombinirati iz razloga što je uloga pojedinih algoritama transformirati podatke kako bi neki drugi bili što efikasniji pri njihovu sažimanju.
Glavni cilj svakog algoritma jest maksimalno sažeti podatke, ali na način da ih poslije možemo vratiti u početni oblik. S obzirom na to, dvije glavne ideje su:

- Smanjiti broj jedinstvenih simbola (smanjiti abecedu).
- Simbole koji se pojavljuju češće zamijeniti s manjim brojem bitova.

Jedan od prvih, i najjednostavnijih, primjera sažimanja podataka jest Morseov kod kojeg su u 19. stoljeću izmislili Samuel F. B. Morse, Joseph Henry i Alfred Vail. Ova metoda je osmišljena za prijenos poruka putem telegrafskih žica. Tekstualne datoteke koje su se slale na ovaj način kodirale su se točkama i crticama kao što je prikazano na slici 1.2. Kako bi se smanjilo vrijeme slanja poruke učestalijim slovima se pridružio kraći zapis, pa je tako '·' predstavljao slovo 'e' dok je zapis '–·-' predstavljao slovo 'q'.



Slika 1.1: Učinak algoritama s gubitcima i bez gubitaka na izvorne podatke

1.1 Entropija

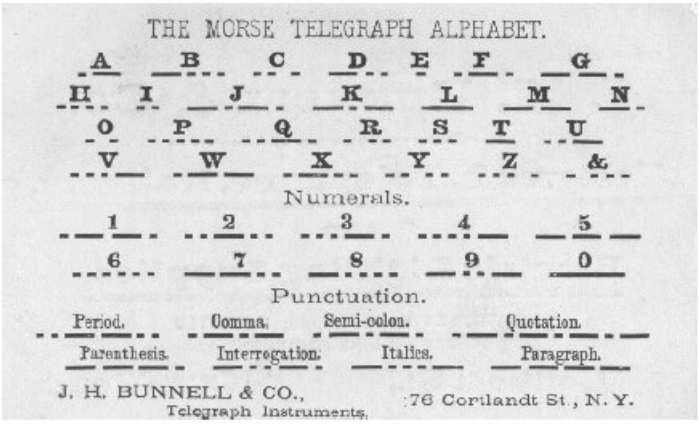
Bilo koji broj u dekadskom zapisu možemo na jedinstveni način zapisati u binarnom sustavu. Broj znamenaka potreban za takav zapis računa se po formuli

$$LOG2(x) = \lceil (log(x+1)/log(2)) \rceil. \tag{1.1}$$

Prvotno je Claude Shannon upravo tu funkciju, $LOG2$, nazvao entropijom, ali je poslije istim tim imenom nazivao i najmanji broj bitova potreban za reprezentaciju cijelog skupa podataka. Time je formula za entropiju $H(s)$ cijelog skupa podataka postala

$$H(s) = -\sum_{i=1}^{n} p_i log_2(p_i), \tag{1.2}$$

pri čemu je n broj znakova abecede, a p_i vjerojatnost pojavljivanja i-tog znaka u zapisu. Ono što je bitno razumjeti jest da što se više puta jedan znak pojavljuje u skupu, to on



Slika 1.2: Morseov kod engleske abecede

Probability set $P(G)$	Entropy $H(G)$	For a set of 1000 symbols…
[0.001, 0.002, 0.003, 0.994]	0.06	994 would be the same
[0.25, 0.25, 0.003, 0.497]	1.53	497 would be the same
[0.1, 0.1, 0.4, 0.4]	1.72	800 would be equally shared by two symbols
[0.1, 0.2, 0.3, 0.4]	1.84	One symbol would dominate but not by much
[0.25, 0.25, 0.25, 0.25]	2	Each symbol would occur 250 times

Slika 1.3: Tablica ovisnosti entropije o vjerojatnostima pojavljivanja znakova u skupu podataka.

manje pridonosi količini informacija, a time se smanjuje vrijednost entropije. Pogledamo li tablicu na slici 1.3 uočit ćemo kako se boljom raspodjelom vjerojatnosti pojavljivanja entropija povećava te je najveća kada su simboli jednoliko zastupljeni u skupu podataka.

Važno je naglasiti da je moguće sažeti podatke na način da je entropija manja od one određene formulom (1.2). Naime, ona ne uzima u obzir redoslijed ili odnos među znakovima koji može smanjiti entropiju, ali nešto više o tome može se pročitati u [3].

1.2 Entropija engleskog jezika

Entropija nekog jezika može poslužiti kao ograda pri kompersiji te je na taj način moguće uspoređivati količinu informacija koju jezici sadrže. Promotrimo li engleski jezik koji sadrži 96 znakova i pretpostavimo li jednolike vjerojatnosti pojavljivanja svakog od njih, za reprezentaciju svakog znaka je tada potrebno $\lceil log(96) \rceil = 7$ bitova, dok entropija iznosi

6.6 bitova/simbolu.
Ukoliko se promatraju vjerojatnosti određene rječnikom engleskog jezika (slovo 'e' je najučestalije, dok se slovo 'z' najrjeđe koristi) entropija iznosi 4.5 bitova/simbolu, dok zasebnim kodiranjem svakog znaka (kao u Huffmanovom algoritmu) ona iznosi 4.7. Nadalje, grupiranje znakova dodatno smanjuje duljinu kodiranog zapisa, pa ukoliko se simboli grupiraju u grupe po osam, entropija iznosi 19 bitova, ali uzimajući u obzir duljinu grupe, entropija po znaku jednaka je 2.4. Povećanjem veličine grupe procijenjeno je kako se entropija može spustiti do 1.3, ali to je nemoguće za dokazati zbog prevelikog broja mogućih stringova.

Vrijednost 1.3 bita/simbolu je procjena količine informacija koju engleski jezik sadrži. Pretpostavljajući da je ona točna, taj broj ograničava koliku kompresiju engleskog teksta možemo očekivati. Na slici 1.4 prikazana je tablica stopa kompresije nekoliko metoda, međutim sve navedene metode su opće namjene te nisu posbeno dizajnirane za engleski jezik.

	bits/char
bits $\lceil \log(96) \rceil$	7
entropy	4.5
Huffman Code (avg.)	4.7
Entropy (Groups of 8)	2.4
Asymptotically approaches:	1.3
Compress	3.7
Gzip	2.7
BOA	2.0

Slika 1.4: Stope kompresije engleskog jezika

Potpuniji set omjera kompresije za Calgary korpus (standardni skup podataka koji se koristi za mjerenje performansi algoritama za kompresiju, a većinom se sastoji od teksta na engleskom jeziku) i trend poboljšavanja novijih tehnika kroz godine prikazan je na slici 1.5.

Date	bpc	scheme	authors
May 1977	3.94	LZ77	Ziv, Lempel
1984	3.32	LZMW	Miller and Wegman
1987	3.30	LZH	Brent
1987	3.24	MTF	Moffat
1987	3.18	LZB	Bell
.	2.71	GZIP	.
1988	2.48	PPMC	Moffat
.	2.47	SAKDC	Williams
Oct 1994	2.34	PPM*	Cleary, Teahan, Witten
1995	2.29	BW	Burrows, Wheeler
1997	1.99	BOA	Sutton
1999	1.89	RK	Taylor

Slika 1.5: Omjeri kompresije tekstualnih podataka s obzirom na Calgary korpus

Poglavlje 2

Metode kompresije bez gubitaka

U ovom poglavlju opisujemo najpoznatije algoritme sažimanja bez gubitaka, ilustriramo ih na jednostavnim primjerima te navodimo, bez dokaza, njihovu vremensku i memorijsku složenost.

2.1 Variable-Length Codes

Tehnika sažimanja *Variable-Length Codes* (VLC) konstruira reprezentaciju zapisa s obzirom na vjerojatnost pojavljivanja pojedinog znaka. Postupak možemo podjeliti u tri koraka:

1. Potrebno je proći po zapisu i izračunati učestalost svakog simbola.
2. Dodjeljujemo kod simbolu ovisno o njegovoj učestalosti na način da učestaliji simboli imaju kraći kod.
3. Još jednim prolaskom zamjenjujemo simbol njegovim kodom i ispisujemo taj kod.

Postupak prikazujemo na stringu "TOBEORNOTTOBEORTOBEORNOT". Na slici 2.1 je prikazana tablica svih slova i broja njihovih ponavljanja u stringu.

Sljedeći korak jest prvo sortirati histogram, silazno po broju ponavljanja, i zatim pridružiti kodove simbolima počevši od najmanjeg koda u odabranom VLC setu kao što je prikazano na slici 2.2.

Posljedni korak ove metode, kompresiju, provodimo ponovnim prolaskom po zapisu na način da za svaki simbol u tablici pogledamo njegov kod te ga njime zamijenimo. Postupak je prikazan na slici 2.3.

Dekodiranje je proces konverzije kodiranih podataka nazad u njihovu originalnu formu. Njega provodimo suprotnim postupkom u odnosu na kodiranje. Prolaskom po kodiranom zapisu i provjerom u tablici kodova pohlepnim se pristupom ispisuju znakovi jedan po

TOBEORNOTTOBEORTOBEORNOT

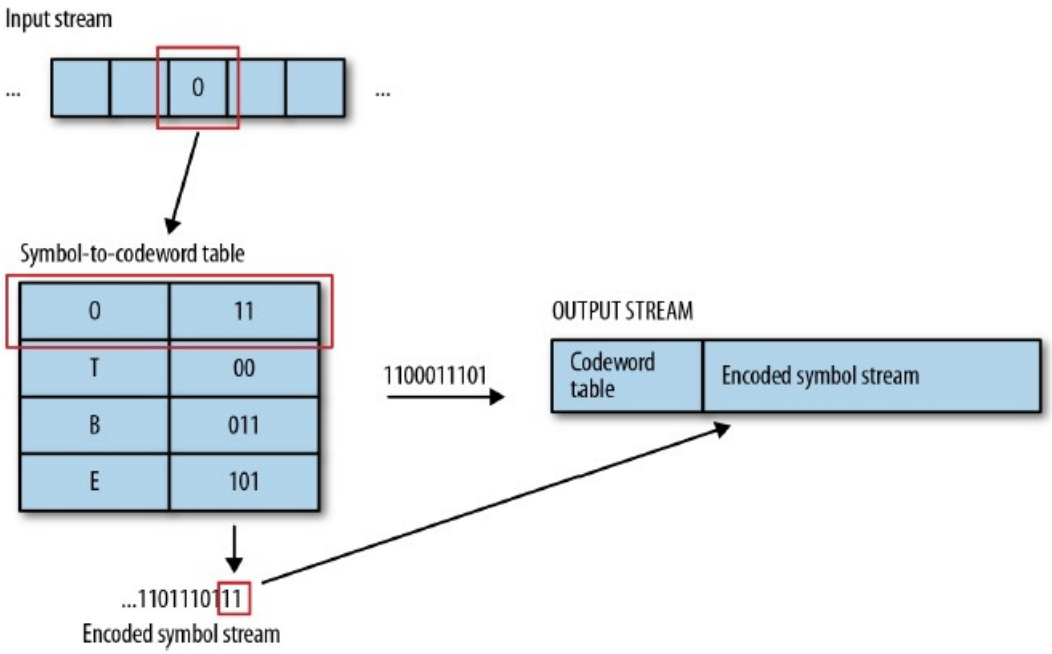
Symbol	Count
T	5
O	8
B	3
E	3
R	3
N	2

Slika 2.1: String i njegov histogram

TOBEORNOTTOBEORTOBEORNOT

Symbol	Count	Code
O	8	11
T	5	00
B	3	011
E	3	101
R	3	0100
N	2	0101

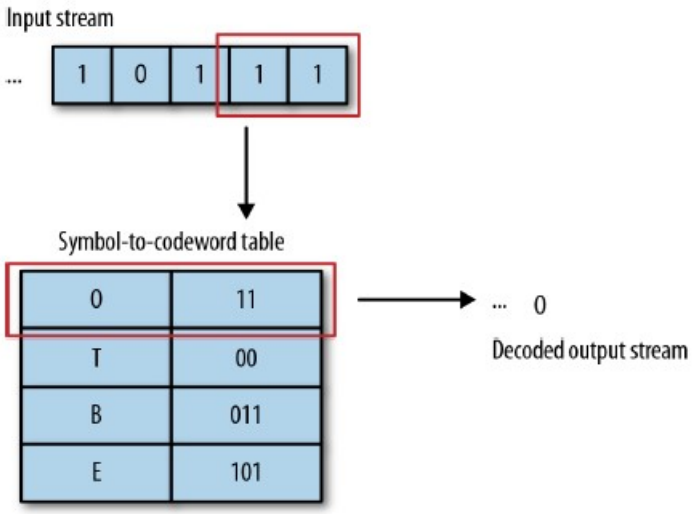
Slika 2.2: Sortirana tablica simbola s pripadnim kodovima



Slika 2.3: Postupak kodiranja ulaznog stringa vrijednostima iz pripadne tablice

jedan kao što je prikazano na slici 2.4.
Možemo primjetiti da više kodova može započinjati istim skupom bitova, npr. kodovi za ’B’ i ’R’ započinju s ’01’. Kako proces dekodiranja ne bi bio dvosmislen, kod niti jednog simbola ne može započinjati kodom nekog drugog, odnosno ako je kod slova ’A’ jednak ’01’, niti jedan drugi kod ne može započinjati s ’01’ (npr. ’0111’). To svojstvo se naziva prefiksalno.

Kako bi ova tehnika bila što efikasnija potrebno je odabrati VLC set koji će najbolje odgovarati početnom setu podataka. Kako je svaki VLC set kreiran na osnovu neke druge distribucije vjerojatnosti, odabirom pogrešnog seta postoji mogućnost da se u konačnici podaci ne sažmu, već dodatno povećaju. Ova metoda se pretežno koristila sredinom 20. stoljeća zbog čega je razvijen veliki broj VLC setova (unarni, binarni, elias kodovi itd.) o kojima se više može pročitati u [5].
Vremenska složenost ovog algoritma je $O(nlogn)$, za svaki simbol (njih n) tražimo u stablu pripadni kod za što je potrebno $O(logn)$ vremena. Prostorna složenost je $O(n)$, koliko je potrebno za spremanje ulaznog podatka. Više detalja o vremenskoj i prostoronoj složenosti može se pročitati u [4].



Slika 2.4: Primjer dekodiranja jednog simbola

2.2 Statističko kodiranje

Statističko kodiranje je jedna on najbitnijih klasa algoritama korištenih pri kompresiji podataka. Zasiva se na konstrukciji jedinstvenog VLC seta u ovisnosti o frekvenciji pojavljivanja pojedinog znaka. Time je povećana efikasnost u usporedbi s korištenjem već postojećih VLC setova.

Huffmanov algoritam

Najpoznatiji i najčešće korišteni algoritam za kompresiju podataka je Huffmanov algoritam. Za konstrukciju VLC kodova koristi se Huffmanovo stablo, binarno stablo koje se oblikuje od dole prema gore. Kao i ranije, simbolima koji se češće pojavljuju dodjeljuju se kraći kodovi pa je time osigurano brže komprimiranje i dekomprimiranje.
Za provođenje algoritma stvaramo listu simbola koji se pojavljuju u ulaznom zapisu te određujemo njihove frekvencije pojavljivanja. Odabiremo dva simbola s najmanjom frekvencijom te stvaramo binarno stablo u čijem je korijenu novonastali simbol dobiven spajanjem dva prvotno odabrana. Dva izvorna simbola zamjenjujemo novonastalim binarnim stablom čija je frekvencija jednaka zbroju frekvencija spajanih simbola. Postupak ponavljamo dok ne ostane samo jedan čvor koji postaje korijen Huffmanovog stabla.
Dodjeljivanje kodova provodi se prolaskom od korijena prema listovima tako da se kretanjem lijevo dodaje bit ’0’, a kretanjem desno bit ’1’. U trenutku dolaska do lista jednog simbola zapisujemo putanju do tog lista, ona onda predstavlja kod tog simbola.

Naposlijetku se generira tablica simbola i pripadnih Huffmanovih kodova koja će služiti pri kodiranju i dekodiranju. Konstrukcija Huffmanovog stabla i pripadnih kodova prikazana je na slici 2.5. Za kodiranje ulaznog niza jednostavno se zamijeni simbol njegovim odgovarajućim Huffmanovim kodom prema generiranoj tablici.

Dekodiranje se odvija prolaskom bit po bit po kodiranoj sekvenci. Ukoliko je pročitan bit '1' krećemo se prema desno, a u suprotnom ulijevo u stablu. U trenutku kada se dođe do lista stabla, taj simbol je dekodiran. Postupak se ponavlja dok se ne pročita cijela sekvenca.

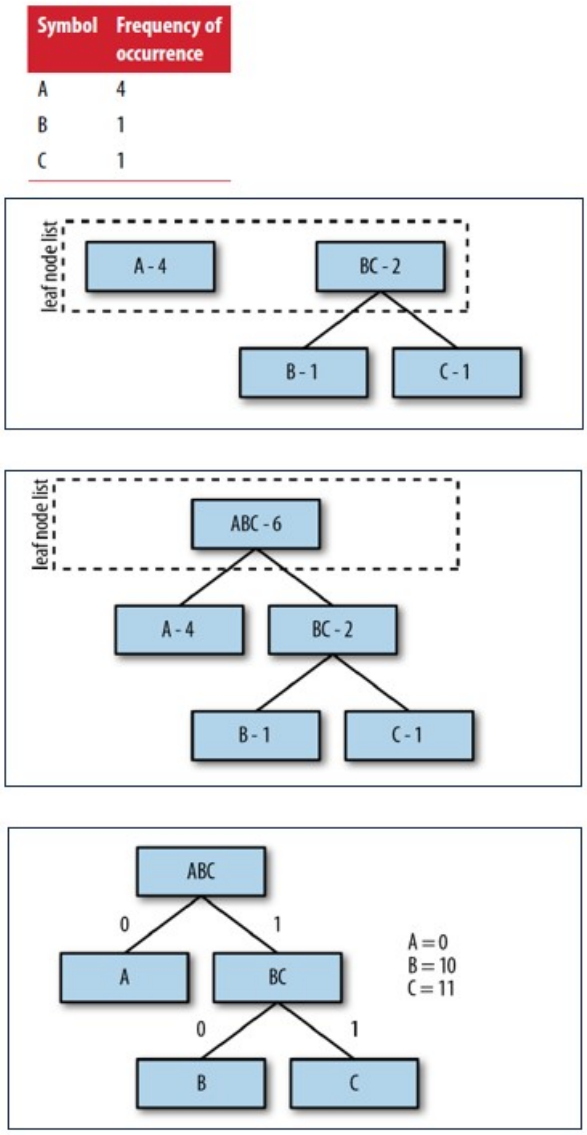
Vremenska složenost ovog algoritma je $O(nlogn)$, za svaki simbol (njih n) tražimo u stablu pripadni kod za što je potrebno $O(logn)$ vremena. Prostorna složenost je $O(n)+O(k)$, gdje je k broj čvorova u Huffmanovom stablu, a osim stabla sprema se i ulazni podatak. Pseudokod algoritma prikazan je u 1.

Algorithm 1 Huffmanov algoritam

```
function HUFFMAN(C)
    n ← C.size()
    Q ← priority_queue()
    for i = 1 to n-1 do
        z ← node(C[i])
        Q.push(z)
    end for
    while Q.size() is not equal to 1 do
        z ← new_node()
        z.left ← x ← Q.pop()
        z.right ← y ← Q.pop()
        z.prob ← x.prob + y.prob
        Q.push(z)
    end while
    return Q
end function
```

Aritmetičko kodiranje

Iako je Huffmanovo kodiranje jednostavno, efikasno i proizvodi najbolje kodove za pojedine simbole, ne proizvodi uvijek najbolje kodove za cijeli set podataka. Zapravo, jedini slučaj kada generira idealni VLC set je kada su se simboli ponavljaju s vjerojatnostima koje su negativne potencije broja dva (npr. 1/2, 1/4 ili 1/8). Za simbol čija je vjerojatnost pojavljivanja jednaka 0.4 idealna duljina koda bi bila 1.32 bita ($-log_2(0.4) \approx 1.32$), ali Huffmanova metoda će tom simbolu dodijeliti kod duljine ili 1 ili 2 bita. Dok god znakove reprezentiramo kodovima cjelobrojne duljine, postojati će razlika između duljine kodirane sekvence i broja bitova potrebnog za zapis koji je određen entropijom.



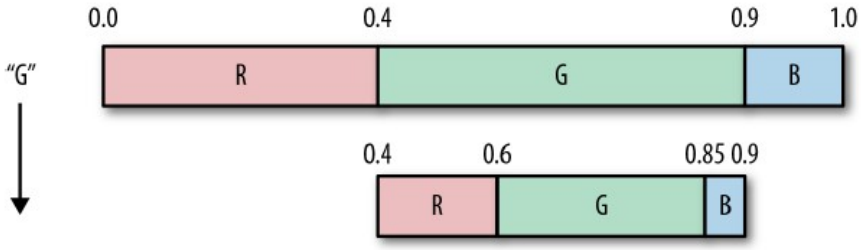
Slika 2.5: Primjer Huffmanovog algoritma

Aritmetičko kodiranje pridržuje cijelom setu podataka jednu, dugu, numeričku vrijednost čija je vrijednost funkcije *LOG*2 bliža entropiji. Taj broj se dobiva kompleksnim procesom nalik na binarno pretraživanje. Algoritam se izvršava na intervalu $[0, 1\rangle$ koji se dijeli s obzirom na vjerojatnosti pojavljivanja simbola (slika 2.6).

Kada se pročita neki znak, njegov podinterval se podijeli jednako kao i na početku uz

Symbol	Probability	Interval
R	0.4	[0,0.4)
G	0.5	[0.4,0.9)
B	0.1	[0.9,1)

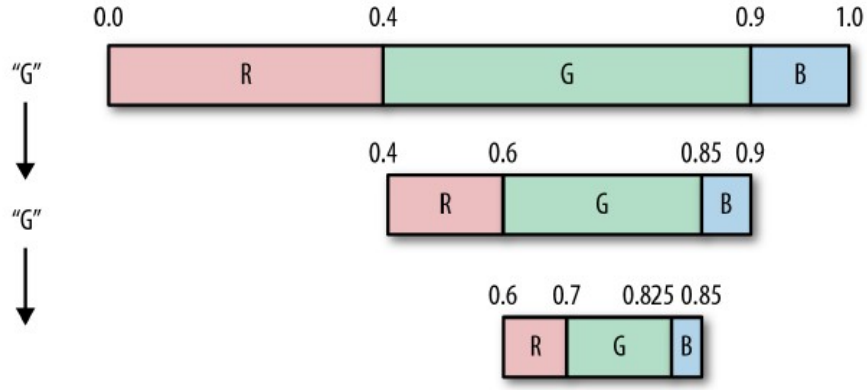
Slika 2.6: Podjela intervala prema tablici.



Symbol	Probability	Updated interval
R	0.4	[0.4,0.6)
G	0.5	[0.6,0.85)
B	0.1	[0.85,0.9)

Slika 2.7: Prvi korak aritmetičkog kodiranja.

promjenu granica podintervala. Taj postupak se ponavlja dok se ne pročita cijeli ulazni podatak. Postupak prikazujemo na zapisu 'GGB'.
Prvi simbol koji se čita jest 'G' pa podinterval $[0.4, 0.9\rangle$ dijelimo jednako kao i u prvom koraku kao što je prikazano na slici 2.7. Time su dobivene nove granice podintervala. Nakon ponovnog čitanja simbola 'G' stanje možemo vidjeti na slici 2.8. Na kraju, čitanjem slova 'B', dobiva se završno stanje (slika 2.9). Kao kôd početnog zapisa uzima se bilo koji broj iz tog intervala, u ovom slučaju iz intervala $[0.825, 0.85\rangle$.



Symbol	Probability	Updated interval
R	0.4	[0.6,0.7)
G	0.5	[0.7,0.825)
B	0.1	[0.825,0.85)

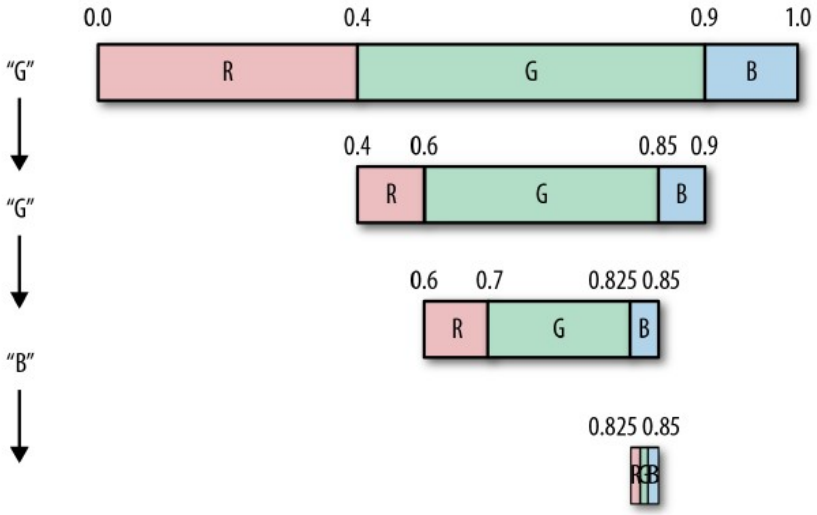
Slika 2.8: Stanje nakon pročitana dva simbola 'G'.

Kako je cilj prikazati ulazni set podataka u minimalno bitova, bira se broj kojim je to moguće postići. U ovom intervalu, taj broj je 0.83. Međutim, poznato je da je taj broj decimalan i da je nula vodeća pa se ona odbacuje, a finalno rješenje tada iznosi 83.
Dekodiranje se odvija na suprotan način u odnosu na kodiranje. Na početku se na jednak način podijeli interval $[0, 1\rangle$ i pogleda kojem podintervalu pripada rezultat kodiranja (pri čemu je dodana vodeća nula). Simbol tog intervala je prva izlazna vrijednost. Postupak kodiranja, odnosno dijeljenje intervala s obzirom na vjerojatnosti, ponavlja se onoliko puta kolika je duljina izvornog zapisa (slika 2.10).

Vremenska složenost ovog algoritma je $O(n)$, za svaki simbol (njih n) se modificira interval za što je potrebno $O(1)$ vremena. Prostorna složenost je $O(n)$ za spremanje ulaznog podatka. Pseudokod aritmetičkog kodiranja može se vidjeti u 2.

2.3 Univerzalno kodiranje

Do sada se moglo zaključiti da metode statističkog kodiranja promatraju pojedine simbole i njihove vjerojatnosti bez obzira na okolinu u kojoj se oni nalaze. Takav način efikasno funckionira na određenim tipovima podataka, no promotrimo li frazu "TO BE OR NOT TO BE" te kodiramo li riječi umjesto slova dobiti ćemo kodove prikazane u tablici na



Symbol	Probability	Updated interval
R	0.4	[0.825,0.835)
G	0.5	[0.835,0.8475)
B	0.1	[0.8475,0.85)

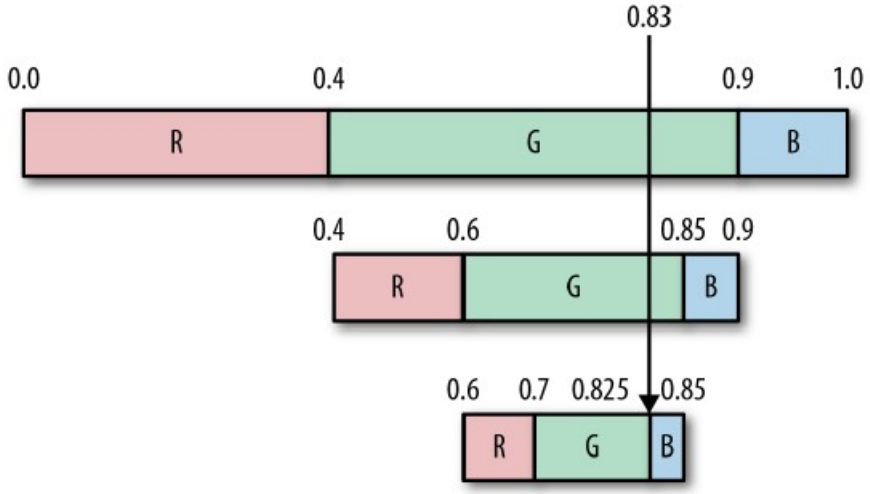
Slika 2.9: Završno stanje algoritma.

slici 2.11. Ovakvim načinom je duljina koda cijelog zapisa 12 bita, dok bi kodiranjem pojedinog slova duljina iznosila 104 bita.

Kada se prestanu promatrati pojedinačni simboli i fokus se stavi na grupe simbola, iz domene statističkog kodiranja se prebaci u svijet univerzalnog kodiranja i metoda rječnika. Metoda rječnika nije zamjena za statističko kodiranje već transformacija koja se prva primjenjuje na ulaznim podacima kako bi statističko kodiranje bilo efikasnije. Najučinkovitija je kada uspije prepoznati dugačke, često ponavljane podstringove zapisa kojima onda dodjeljuje kraće kodove.
Problem koji se nadalje javlja jest kako pronaći ”riječi” čija će kombinacija rezultirati s kodom najmanje entropije. Taj proces pronalaska idealnih ”riječi” se naziva tokenizacija.

Lempel-Ziv algoritmi

1977. i 1978, godine Abraham Lempel i Jacob Ziv opisali su dva algoritma koji rješavaju problem ”idealne tokenizacije” i nazivaju se LZ77 i LZ78. Razlika među njima jest u tome koliko naprijed i na koji način traže podudaranja. Do danas nije pronađen niti jedan algo-



Slika 2.10: Dekodiranje aritmetički kodiranog zapisa.

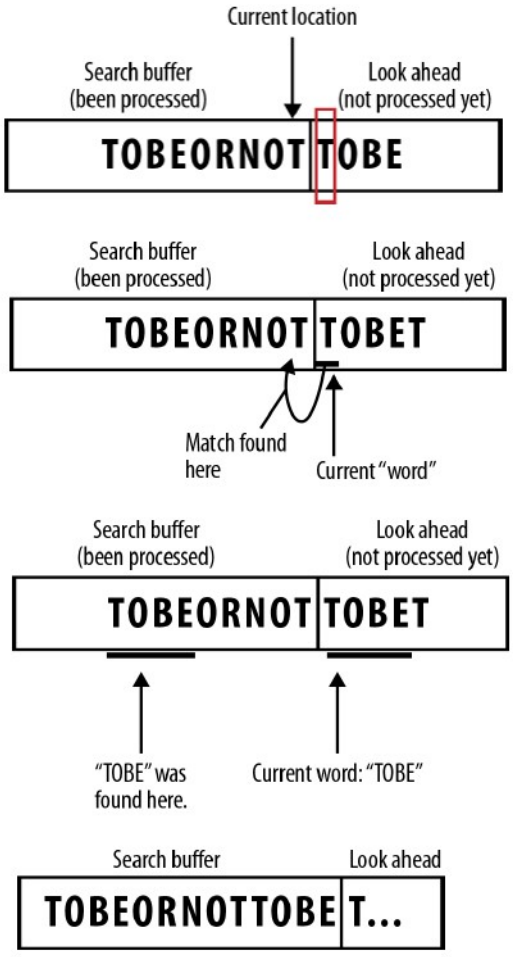
Algorithm 2 Aritmetičko kodiranje

```
function ARITHMETIC(C)
    lower ← 0
    upper ← 1
    high_range := dictionary of upper bound for each symbol
    lower_range := dictionary of lower bound for each symbol
    for symbol in C do
        range ← upper − lower
        upper ← lower + range ∗ high_range[symbol]
        lower ← lower + range ∗ lower_range[symbol]
        update high_range and lower_range
    end for
    return (upper + lower)/2
end function
```

ritam koji bi ih mogao zamijeniti.
LZ77 i LZ78 se temelje na ideji stvaranja rječnika od već pročitanih simbola datoteke. S obzirom na mjesto s kojeg se trenutno čita, zapis se dijeli na dva dijela, *buffera* za pretraživanje i *look-ahead buffera*. Nakon toga, pokušava se pronaći najdulji podstring koji započinje na tom mjestu, a nalazi se u već pročitanom dijelu zapisa, odnosno *bufferu* za pretraživanje. Nekoliko koraka algoritma je prikazano na slici 2.12.

Symbol	Frequency	Codeword
TO	0.33	00
BE	0.33	01
OR	0.16	10
NOT	0.16	11

Slika 2.11: Rezultat kodiranja riječi umjesto slova



Slika 2.12: Nekoliko koraka Lempel-Ziv algoritma

LZ77

LZ77 koristi pomične prozore, što znači da *buffer* za pretraživanje nije cijela već kodirana sekvenca, već samo njen zadnji dio koji je u praksi veličine od oko 32 KB. *Look-ahead buffer* također neće biti cijeli zapis po kojem se treba proći, već je njegova duljina oko 10 bajtova. Isto tako, kako se pomiče pokazivač po zapisu tako se pomiču i granice pomičnog prozora (slika 2.13).

Slika 2.13: Pomak granica pomičnog prozora nakon jednog koraka algoritma LZ77.

Kodiranje se odvija u nekoliko koraka, a prvi je pronaći najdulji podstring koji počinje na mjestu pokazivača, a nalazi se u *bufferu* za pretraživanje. Zatim se ispiše uređena trojka (p, d, n) u kojoj je p pozicija početka tog stringa u *bufferu* za pretraživanje, d duljina podudaranja, a n prvi sljedeći znak koji slijedi nakon podudaranja u *look-ahead bufferu*. Ukoliko nije pronađeno podudaranje, izlaz će biti uređena trojka $(0, 0, n)$ pri čemu je n simbol koji se trenutno proučava. Algoritam je prikazan na slici 3.

Dekodiranje se izvršava čitanjem tokena koje je koder konstruirao. Ukoliko je token oblika $(0, 0, n)$ dekoder ispisuje samo simbol n, a u suprotnom broji p koraka unazad, od trenutne pozicije, u rekonstruiranom stringu i vraća podstring duljine d te simbol n.
S obzirom na Huffmanov algoritam, LZ77 ima sporije vrijeme komprimiranja, ali bolji omjer kompresije dok je dekodiranje nešto brže.

Vremenska složenost algoritma LZ77 je $O(n)$ zbog toga što za svaki pročitani simbol, vrijeme provjere *buffera* za pretraživanje neće biti veće od $O(n)$. Prostorna složenost je $O(n)$ za spremanje ulaznog podatka.
Pseudokod algoritma LZ77 prikazan je u 3.

LZW (Lempel-Ziv-Welch)

Prethodni algoritam ima nekoliko problema kao što su fiksna duljina prozora koja propušta uzorke koji se pojavljuju van prozora ili mogućnost povećanja duljine izlaznog niza ukoliko podaci nemaju mnogo ponavljajućih uzoraka. Kako bi se ti problemi izbjegli Lempel i Ziv su 1978. godine razvili novi algoritam LZ78. Njegova najpopularnija varijanta je LZW koju je 1984. godine razvio Terry Welch.

Search buffer	Look ahead buffer	Output
	TOBEORNOTTOBE	0,0,T
T	OBEORNOTTOBE	0,0,O
TO	BEORNOTTOBE	0,0,B
TOB	EORNOTTOBE	0,0,E
TOBE	ORNOTTOBE	3,1,R
TOBEOR	NOTTOBE	0,0,N
TOBEORN	OTTOBE	3,1,T
TOBEORNOT	TOBE	9,4,<eos>
		<eos>

Slika 2.14: Kompresija algoritmom LZ77

LZW ne koristi ni *buffere* niti pomične prozore već postoji rječnik prethodno pročitanih stringova. Na početku se rječnik sastoji od svih ASCII kodova, njih 256, duljine jedan. Princip rada je da se simboli unose jedan po jedan i zapisuju u niz I. Nakon svakog dodavanja simbola u I, rječnik se pretražuje te ukoliko se I nalazi unutar rječnika postupak se nastavlja. U trenutku kada se doda simbol x i u rječniku se ne pronalazi string Ix, on se dodaje na prvo sljedeće slobodno mjesto, ispisuje se pokazivač na string I te on postaje x. Primjer kodiranja ovim algoritmom prikazan je na slici 2.15.

Pri dekodiranju stvara se isti rječnik kao i ranije, a dekoder započinje s prvih 256 kodova predefiniranih za prevođenje. Nakon svakog čitanja koda (osim prvog) tablica se nadopunjuje. U prvom koraku se pročita kod i dohvati simbol I. U svakom sljedećem, nakon pročitanog koda i dohvaćanja stringa J iz rječnika, na sljedeće slobodno mjesto se upisuje string Ix, pri čemu je x prvi simbol stringa J. Na kraju, J postoje I, a dekoder nastavlja isti postupak dok ne pročita zadnji kod s ulaza. Dekodiranje prethodnog primjera prikazano je na slici 2.16.

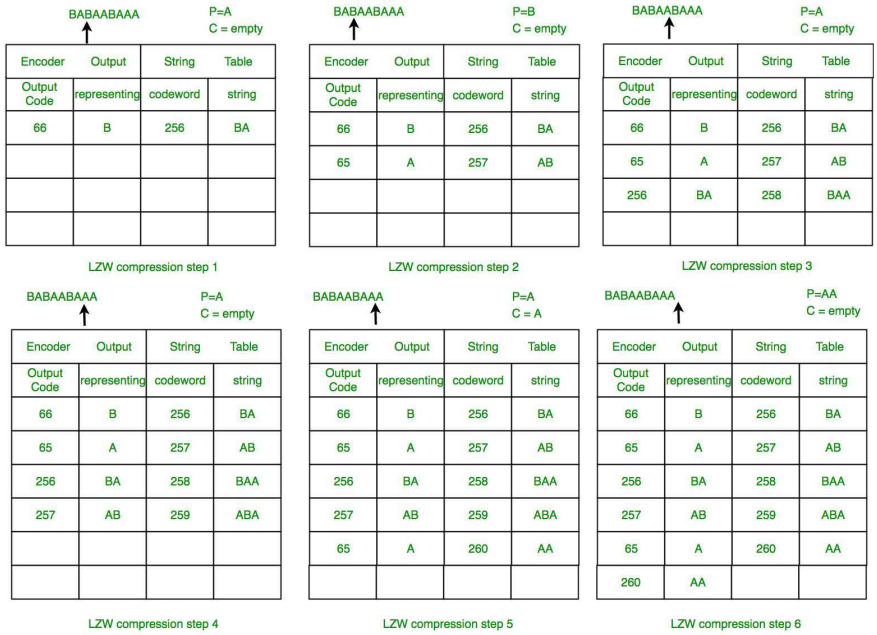
Vremenska složenost LZW algoritma je $O(n)$ pošto niti za jedan korak algoritma (čitanje, pretraživanje i upisivanje u rječnik) ne treba više od $O(n)$ vremena. Prostorna složenost je $O(n)$ koliko je potrebno za spremanje ulaznog podatka i rječnika.
Pseudokod algoritma LZW se može vidjeti u 4.

Algorithm 3 LZ77

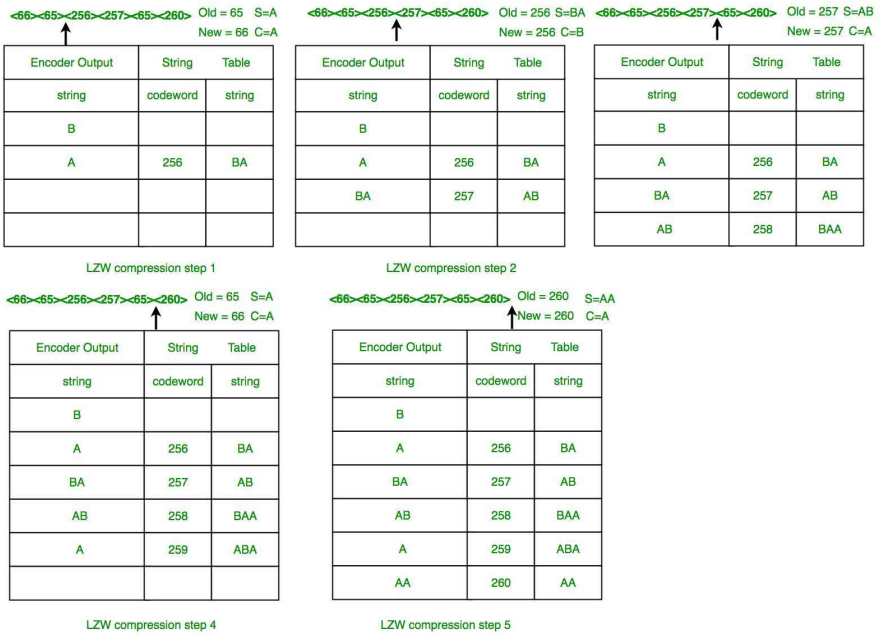
```
function LZ77(C)
    initialize empty list codes and buffer
    n ← C.size()
    for i = 0 to n do
        s ← find_longest_substring(i, buffer)
        if s exists then
            i ← distance from s in buffer
            l ← size of the s
            n ← character after the s
        else
            i ← 0
            l ← 0
            n ← first character in s
        end if
        codes += (i, l, n)
        i += l + 1
        buffer += C[i : l + 1]
    end for
    return codes
end function
```

Algorithm 4 LZW

```
function LZW(C)
    initialize list dictionary
    string ← first symbol in C
    while there are still symbols in C do
        symbol ← get next symbol from C
        if string + symbol in dictionary then
            string += symbol
        else
            output the code for string
            dictionary += string + symbol
            string ← symbol
        end if
    end while
end function
```



Slika 2.15: Primjer LZW algoritma na ulaznom stringu BABAABAAA



Slika 2.16: Dekodiranje stringa BABAABAAA

Poglavlje 3

Implementacija i usporedba algoritama

Nakon opisa algoritama, u ovom poglavlju navodimo detalje implementacije te uspoređujemo performanse na više skupova podataka. U tablici 3.1 prikazane su vremenske i prostorne složenosti odabranih algoritama.

	Huffmanov algoritam	Aritmetičko kodiranje	LZW	LZ77
Vremenska složenost	$O(nlogn)$	$O(n)$	$O(n)$	$O(n)$
Prostorna složenost	$O(k) + O(n)$	$O(n)$	$O(n)$	$O(n)$

Tablica 3.1: Složenosti algoritama

3.1 Opis implementacije

Svi algoritmi implementirani su u programskom jeziku Python. Računalo na kojem su se izvršavali programi sadrži procesor AMD Ryzen 5 5500U frekvencije radnog takta 2.10 GHz i 8.0 GB RAM memorije. Korišten je 64-bitni operacijski sustav Windows 11 Pro. Glavne funkcije pojedinih algoritama prikazujemo u nastavku dok je cjelokupni kod dostupan u [1].
Funkcija koja implementira Huffmanov algoritam, `Huffman_encoding`, prima zapis i prolaskom po njemu stvara čvorove stabla koje onda spaja po danim pravilima. Svaki čvor dio je klase `Node`. Povratna vrijednost funkcije jest kodirani zapis i korijen nastalog Huffmanovog stabla.

```
def Huffman_encoding(data):
    symbols_and_prob = Calculate_Probability(data)
    symbols = symbols_and_prob.keys()

    nodes = []

    for symbol in symbols:
        nodes.append(Node(symbols_and_prob.get(symbol),symbol))

    while len(nodes) > 1:

        nodes = sorted(nodes, key = lambda x : x.prob)

        right = nodes[0]
        left = nodes[1]

        left.code = 0
        right.code = 1

        newNode = Node(left.prob+right.prob,
                left.symbol+right.symbol, left, right)
        nodes.remove(left)
        nodes.remove(right)
        nodes.append(newNode)

    huffman_encoding = Calculate_Codes(nodes[0])
    print("symbols with codes", huffman_encoding)
    encoded_output = Output_Encoded(data,huffman_encoding)
    return encoded_output, nodes[0]
```

Nadalje, funkcija koja izvršava artimetičko kodiranje, osim skupa podataka prima i tablicu vjerojatnosti svih simbola. Nakon pročitanog ulaza i dijeljenja intervala u skladu s njim, funkcija kao izlaz vraća broj iz posljednjeg intervala koji predstavlja kod ulaznog zapisa.

```
def Arithmetic_encoding(data, prob_table):

    lower , upper = Decimal(0.0), Decimal(1.0)
    low, up = Split_Interval(lower, upper, prob_table)
    for element in data:
        lower, upper = Decimal(low[element]), Decimal(up[element])
        low, up = Split_Interval(low[element], up[element], prob_table)

    l = len(data)

    return Decimal((upper+lower)/2)
```

Funkcije koje izvrašavaju algoritme univerzalnog kodiranja, `LZW_Encoding` i `LZ77`, primaju samo ulazni skup podataka, a kao izlazni rezultat vraćaju listu pokazivača, odnosno uređenih trojki pokazivača, duljine i simbola.

```
def LZW_Encoding(data):

    codes = []
    table = initialize_table()

    p = data[0]
    c = ""
    code = 256
    for i in range(1,len(data)):
        if i != len(data):
            c += data[i]

        if (p+c) in table:
            p = p + c
        else:
            print(p, "  ", table[p],"\t", p+c, "\t", code)
            codes.append(table[p])
            table[p+c] = code
            code += 1
            p = c

        c = ""

    print(p, "\t", table[p])
    codes.append(table[p])
    return code
```

```
def LZ77(data):
    codes = []
    buffer = []
    i = 0
    while i < len(data):
        p = find_last(buffer,data[i],0)
        duljina = 1
        pok = p
        while p != -1:
            l = 1
            for j in range(1,15):
                if(i+j < len(data) and p+j < len(buffer)
                    and buffer[p+j] == data[i+j]):
                    l += 1
                else:
                    break
```

```
            if(duljina <= l):
                duljina = l
                pok = p
            p = find_last(buffer,data[i],p+1)

        l = duljina
        p = pok
        if(pok != -1):
            if(i + l < len(data)):
                codes.append([len(buffer) - p,l,data[i+l]])
            else:
                codes.append([len(buffer) - p,l,'<eof>'])
            for k in range(l+1):
                if(len(buffer) > 400):
                    buffer.pop(0)
                if( i + k < len(data)):
                    buffer.append(data[i+k])
            i = i + l + 1
        else:
            codes.append([0,0,data[i]])
            if(len(buffer) > 400):
                buffer.pop(0)
            buffer.append(data[i])
            i += 1
            continue

    return codes
```

3.2 Rezultati testiranja

Korišteni testni podaci se mogu pronaći u [2]. Riječ je o tekstualnim datotekama *Hamlet.txt*, *GreenEggsAndHam.txt*, *Trump-Inaugural.txt* i *TomSawyer.txt*. Navedene datoteke razlikuju se po duljini zapisa i složenosti teksta. Najmanje složen tekst i najkraći zapis ima *GreenEggsAndHam.txt* dok *TomSawyer.txt* ima najdulji i najsloženiji tekst. Veličine datoteka navedenih u tablici 3.2 su redom, 188 KB, 4 KB, 9 KB i 378 KB.

U tablici 3.2 prikazane su duljine kodiranog zapisa ulazne datoteke dobivenog četirima algoritmima dok su u tablici 3.3 prikazane stope kompresije za odabrane algoritme. Može se uočiti da je aritmetičko kodiranje ima najbolje rezultate gledajući duljinu koda, dok LZ77 ima najlošije. Nedostatak aritmetičkog kodiranja jest vrijeme kodiranja koje je nekoliko puta veće u odnosu na ostale. Npr. na ulaznom podatku ”Trump-Inaugural.txt” vrijeme potrebno za kodiranje aritmetičkim algoritmom jest 15.66 s, dok je vrijeme po-

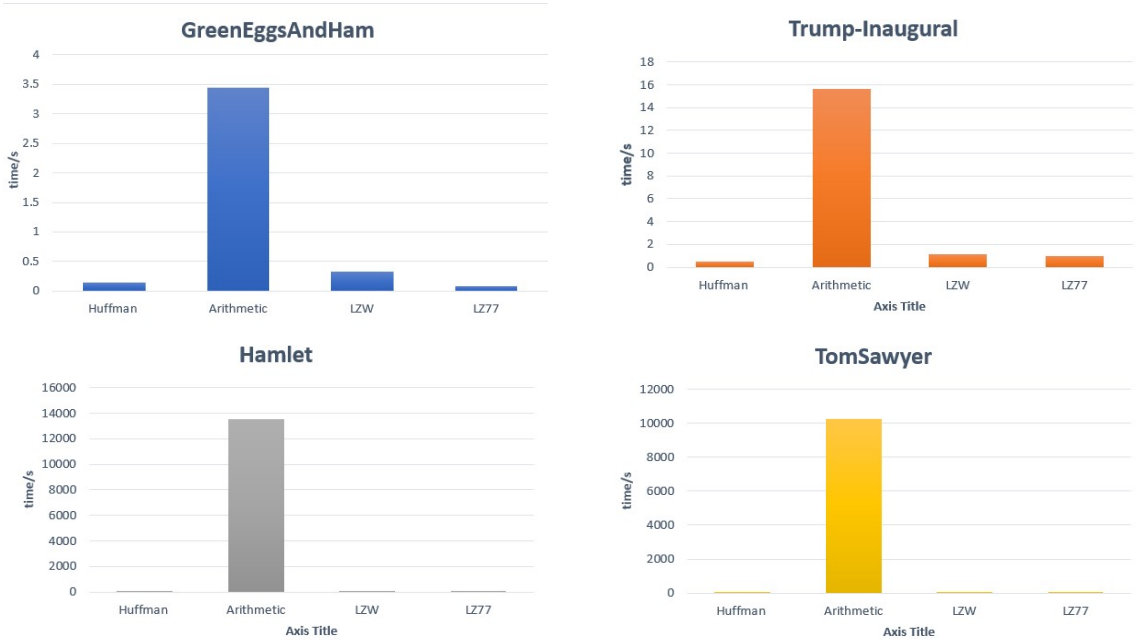
trebno LZW algoritmu 1.45 s. Na slici 3.1 prikazana je usporedba vremena za odabrana četiri algoritma i četiri skupa podataka.

Datoteka	**Huffmanov algoritam**	**Aritmetičko kodiranje**	**LZW**	**LZ77**
Hamlet	843 328	697 601	772 941	1 534 891
GreenEggsAndHam	13 359	13 286	12 173	18 729
Trump-Inaugural	37 246	39 861	43 914	66 393
TomSawyer	1 766 292	1 494 866	1 560 972	3 262 168

Tablica 3.2: Usporedba algoritama na četiri skupa podataka

Datoteka	**Huffmanov algoritam**	**Aritmetičko kodiranje**	**LZW**	**LZ77**
Hamlet	55 %	45 %	50.5 %	100 %
GreenEggsAndHam	51 %	50.7 %	46.5 %	71 %
Trump-Inaugural	56 %	59.5 %	65.7 %	99 %
TomSawyer	57 %	48 %	50.5 %	105 %

Tablica 3.3: Stope kompresije odabranih algoritama



Slika 3.1: Usporedba vremena kodiranja na četiri skupa podataka

Zaključak

Razvojem društva i tehnologije povećava se potreba za što bržim prijenosom informacija što ne bi bilo moguće bez različitih metoda kompresije. Velika većina podataka, odnosno slika i video zapisa koji se postavljaju i preuzimaju s interneta je u komprimiranom obliku. U suprotnom, slanje samo jedne stranice dokumenta trajalo bi cijeli dan.

Svaki od opisanih algoritama u ovom radu koristi se u različitim aspektima i aplikacijama za kompresiju podataka. Huffmanov algoritam često se koristi za kompresiju teksta, slika (JPEG format), zvuka (FLAC) i drugih vrsta podataka. Također, koristi se i u mnogim komunikacijskim protokolima i formatima datoteka kako bi se smanjila količina podataka za prijenos, odnosno pohranjivanje.
Aritmetičko kodiranje koristi se u sličnim područjima kao i Huffmanov, pri kompresiji slika (JPEG 2000) i kompresiji teksta u adaptivnim kompresijskim algoritmima. Iako ima bolje stope kompresije, matematički je zahtjevniji pa samo kodiranje duže traje.
Najpoznatiji primjer primjene LZW algoritma jest kompresija slika u formatima GIF, PNG i TIFF. LZ77 je osnova za formate kompresije poput Deflate, koji se koristi u ZIP datotekama i GZIP (korišten u komprimiranim arhivima).
Svaki od navedenih algoritama ima svoje prednosti i nedostatke. Odabirom odgovarajućeg algoritma za određeni scenarij može se postići značajna ušteda u prostoru za pohranu, odnosno smanjenje brzine prijenosa podataka.

Bibliografija

[1] https://github.com/mpesut/Data-compression-algorithms.

[2] https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1234/assignments/a8/.

[3] A. Haecky C. McAnlis, *Understanding Compression: Data Compression for Modern Developers*, O'Reilly, 2016.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest i C. Stein, *Introduction to Algorithms, 3rd Edition*, MIT Press, 2009.

[5] D. Salomon, *A Concise Introduction to Data Compression*, Springer, 2008.

Sažetak

U današnjem digitalnom svijetu, brza i efikasna kompresija podataka od iznimne je važnosti za učinkovitu pohranu, prijenos i obradu informacija. Ovaj radi fokusira se na algoritme sažimanja bez gubitaka koji u potpunosti mogu obnoviti ulazne podatke nakon kodiranja. Opisano je nekoliko algoritama koji pripadaju trima tehnikama kompresije, *Variable-Length*, statističkom i univerzalnom kodiranju i uspoređene su performanse odabranih algoritama na nekoliko skupova podataka. Rezultati su pokazali da ne postoji univerzalno najbolji algoritam sažimanja. Umjesto toga, izbor algoritma ovisi o specifičnim potrebama aplikacije. Ukoliko je najvažnija mala veličina kodiranog podatka i brzina nije od presudne važnosti, aritmetičko kodiranje odličan je izbor. Međutim, za brze aplikacije koje zahtijevaju dobru kompresiju LZW i Huffmanov algoritam pružaju razumne rezultate.
Ovaj rad naglašava važnost razumijevanja različitih algoritama sažimanja podataka i njihovih performansi kako bi se odabrao najprikladniji za određenu situaciju.

Summary

In today's digital world, fast and efficient data compression is of utmost importance for efficient storage, transmission, and processing of information. This paper focuses on lossless compression algorithms that can fully restore input data after encoding. Several algorithms belonging to three compression techniques, Variable-Length, Statistical, and Universal coding, are described, and the performance of selected algorithms on multiple datasets is compared. The results have shown that there is no universally best compression algorithm. Instead, the choice of algorithm depends on the specific needs of the application. If achieving a small encoded data size is crucial, and speed is not of topmost importance, Arithmetic coding is an excellent choice. However, for fast applications requiring good compression, LZW and Huffman coding provide reasonable results.
This paper emphasizes the importance of understanding different data compression algorithms and their performance to select the most suitable one for a specific situation.

Životopis

Matea Pešut rođena je 1. 9. 1999. u Rijeci gdje završava Osnovnu školu Srdoči i Gimnaziju Andrije Mohorovičića Rijeka. Nakon završene srednje škole 2018. godine upisuje preddiplomski sveučilišni studij Fizika na Odjelu za fiziku Sveučilišta u Rijeci koji uspješno završava 2021. godine. Svoje obrazovanje nastavlja upisom sveučilišnog diplomskog studija Računarstvo i matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu.