

Kompresija podataka

Farena, Nikolina

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: <https://um.nsk.hr/um:nbn:hr:126:881938>

Rights / Prava: [In copyright](#)/[Zaštićeno autorskim pravom.](#)

Download date / Datum preuzimanja: **2024-07-24**



Repository / Repozitorij:

[Repository of School of Applied Mathematics and Computer Science](#)



Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku

Nikolina Farena

Kompresija podataka

Diplomski rad

Osijek, 2020.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku

Nikolina Farena

Kompresija podataka

Diplomski rad

Mentor: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2020.

Sadržaj

1	Uvod	1
2	Teorija informacija	2
2.1	Entropija	2
2.2	Uvjetna entropija i Markovljevi lanci	3
2.3	Prefiks kodovi	4
2.3.1	Veza s entropijom	5
3	Algoritmi za kompresiju bez gubitaka podataka	7
3.1	Run-length Encoding	7
3.2	Huffmanov algoritam	8
3.2.1	Huffmanovi algoritmi s minimalnom varijancom	12
3.3	Aritmetičko kodiranje	13
3.3.1	Cjelobrojna implementacija	15
3.4	Lempel-Ziv algoritmi	18
3.4.1	LZ77 - Pomični prozori	18
3.4.2	LZ78	19
4	Algoritmi za kompresiju s gubitcima podataka	22
4.1	Tehnike koje se koriste pri kompresiji s gubitcima podataka	22
4.1.1	Skalarna kvantizacija	22
4.1.2	Vektorska kvantizacija	23
4.1.3	Transformacijsko kodiranje	23
4.2	JPEG	24
4.3	MPEG	28

1 Uvod

Kompresija podataka je proces u kojem smanjujemo fizički prostor potreban za pohranu podataka korištenjem određenih metoda za njihovo zabilježavanje. Najzastupljeniji primjeri kompresije s kojima su upoznati gotovo svi korisnici računala su .rar i .zip datoteke, takozvane arhive. Cilj takvih arhiva je bez gubitka podataka pretvoriti više vrsta datoteka u jednu datoteku koristeći različite algoritme, kako bi na lakši i brži način mogli razmjenjivati podatke putem interneta.

Promatrajući neki tekst možemo primjetiti da se određeni izrazi, pa čak i rečenice, u tom tekstu ponavljaju. Prilikom kompresije takvog teksta, bilježili bi izraze te mjesta na kojima se oni pojavljuju. Možemo primjetiti da ovakvim načinom kompresije ne gubimo nikakve informacije. Drugi način kompresije podataka je onaj u kojemu se dio podataka prilikom kompresije izgubi. Takav oblik kompresije koristi se dosta često prilikom slanja fotografija preko društvenih mreža, pri čemu fotografija koju pošaljemo sa svoga uređaja nije jednake kvalitete kao ona koju primatelj otvara na svom uređaju. Ova dva jednostavna primjera opisuju dva načina kompresije podataka kojima ćemo se baviti u ovom diplomskom radu.

Kod kompresije bez gubitka podataka bitno je da se početni podatci ne razlikuju od podataka nakon dekompresije. Ovakav oblik kompresije izrazito je bitan u bankarstvu, jer gubitkom i samo jedne 0 cijeli izvještaji više ne bi imali smisla. Također, kod fotografija, video i audio snimki koje će naknadno biti korištene u svrhu istraživanja bilo bi nerazumno dozvoliti da dio podataka bude izgubljen.

Kod kompresija s gubitcima podataka originalni podatci ne mogu se u potpunosti rekonstruirati iz kompresiranih podataka. Bez obzira na to što je dio podataka izgubljen to ne znači nužno da je kvaliteta podataka smanjena. Kao primjer, možemo promatrati video u kojem je jak pozadinski šum. Kada bi se taj šum izgubio prilikom kompresije sama kvaliteta videozapisa bi bila poboljšana. Upravo zbog gubitaka koji se dopuštaju u ovakvim algoritmima postiže se bolji omjer kompresije. Međutim, bitno je obratiti pažnju da gubici koji se pojavljuju pri kompresiji budu što manje vidljivi. Upravo iz tog razloga algoritmi za ovakvu kompresiju podataka uvelike se razlikuju ovisno o mediju na kojem vršimo kompresiju.

U ovom radu proći ćemo kroz osnovne pojmove teorije informacija te analizirati nekoliko algoritama za kompresiju podataka. Kod kompresija bez gubitaka podataka to će biti Run-Length algoritam, Huffmanov algoritam i Lempel-Ziv algoritmi, dok kod kompresije s gubitcima podataka bit će obrađene tehnike koje se koriste pri kompresiji i JPEG i MPEG algoritmi.

KOMPRESIJA S GUBITCIMA PODATAKA	KOMPRESIJA BEZ GUBITKA PODATAKA
Eliminira podatke koji nisu uočljivi. Originalne podatke ne možemo rekonstruirati. Kvaliteta podataka može biti narušena. Smanjuje veličnu originalnih podataka. Koristi se za slike, zvučne zapise i video zapise.	Ne eliminira podatke. Originalni podaci se mogu rekonstruirati. Kvaliteta podataka ostaje ista. Ne smanjuje veličinu originalnih podataka. Koristi se za tekst, slike i zvučne zapise.

Tablica 1.0.1: Razlika između kompresije bez i s gubitcima podataka

2 Teorija informacija

2.1 Entropija

Pojam entropije preuzet je iz fizike, pri čemu entropija predstavlja randomiziranost u nekom sustavu. Točnije, za određeni sustav pretpostavlja se da ima konačan broj stanja u kojima se može nalaziti i za dano vrijeme postoji vjerojatnosna distribucija za to stanje. Entropija je tada definirana na sljedeći način ¹:

$$H(s) = \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)},$$

gdje je S skup svih mogućih stanja, a $p(s)$ je vjerojatnost stanja $s \in S$. Po definiciji zaključujemo da je, ako su vjerojatnosti za sva stanja približno jednaka, entropija veća i suprotno, što je veća razlika u vjerojatnostima entropija je manja, te ukoliko točno znamo u kojem se stanju sustav nalazi $H(S) = 0$.

U kontekstu teorije informacija poruka će nam predstavljati stanje, tj. S je skup svih mogućih poruka, a $p(s)$ je vjerojatnost poruke $s \in S$. Vlastiti sadržaj informacije (*self information* ²) definira se kao:

$$i(s) = \log_2 \frac{1}{p(s)}.$$

Vlastiti sadržaj informacije predstavlja količinu informacija koje se nalaze u poruci. Po definiciji možemo zaključiti da poruka veće vjerojatnosti sadrži manje informacija. Dakle, entropija je vjerojatnost težinskog prosjeka vlastitih sadržaja informacija svih poruka, tj. prosječan broj bitova za informaciju u poruci koja je nasumično izabrana iz vjerojatnosne distribucije. Što je set poruka više randomiziran, sadržavati će u prosjeku više informacija.

Primjer 2.1.1. Pokažimo primjerima da što su veće razlike među vjerojatnostima, entropija će biti manja.

$$\begin{aligned} p(S) &= \{0.25, 0.25, 0.25, 0.125, 0.125\} \\ H &= 3 \cdot 0.25 \cdot \log_2 4 + 2 \cdot 0.125 \cdot \log_2 8 \\ &= 1.5 + 0.75 \\ &= 2.25 \end{aligned}$$

$$\begin{aligned} p(S) &= \{0.5, 0.125, 0.125, 0.125, 0.125\} \\ H &= 0.5 \cdot \log_2 2 + 4 \cdot 0.125 \cdot \log_2 8 \\ &= 0.5 + 1.5 \\ &= 2 \end{aligned}$$

$$\begin{aligned} p(S) &= \{0.75, 0.0625, 0.0625, 0.0625, 0.0625\} \\ H &= 0.75 \cdot \log_2 \frac{4}{3} + 4 \cdot 0.0625 \cdot \log_2 8 \\ &= 0.3 + 1 \\ &= 1.3 \end{aligned}$$

¹I.S.Pandžić, A.Bažant, Ž.Ilić, Z.Vrdoljak, M.Kos, V. Sinković (2009.), *Unod u teoriju informacija i kodiranje*, 2.izd, Element, Zagreb

²Guy E.Belloch, *Introduction to data compression*, Carneige Mellon University, 2013

2.2 Uvjetna entropija i Markovljevi lanci

Vjerojatnost nekog događaja (poruke) ovisi o uvjetima u kojima će se dogoditi, a poznavajući te uvjete možemo popraviti vjerojatnosti događaja, u našem slučaju smanjiti entropiju.

Uvjetnu vjerojatnost ³ događaja e obzirom na uvjet c označavat ćemo s $p(e|c)$. Vjerojatnost događaja e možemo računati kao

$$p(e) = \sum_{c \in C} p(c)p(e|c),$$

pri čemu je C skup svih mogućih uvjeta. Obzirom na uvjetne vjerojatnosti možemo definirati uvjetni vlastiti sadržaj informacije događaja e uz uvjet c kao:

$$i(e|c) = \log_2 \frac{1}{p(e|c)}.$$

Kao i u slučaju bez uvjeta, možemo definirati prosječni uvjetni vlastiti sadržaj informacije, koji ćemo zvati uvjetna entropija izvora poruke. Za skup poruka S uz skup uvjeta C , uvjetna entropija računa se kao:

$$H(S|C) = \sum_{c \in C} p(c) \sum_{s \in S} p(s|c) \log_2 \frac{1}{p(s|c)}.$$

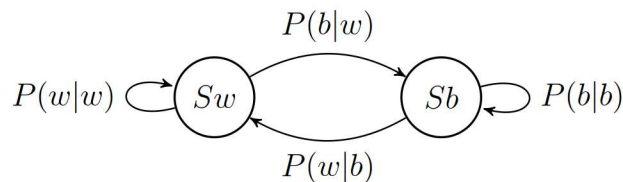
Ukoliko je vjerojatnosna distribucija skupa S neovisna o uvjetima iz C tada je $H(S|C) = H(S)$, inače $H(S|C) < H(S)$. Drugim rječima, poznavajući kontekst možemo smanjiti entropiju.

Entropiju također možemo definirati pomoću izvora informacija. Izvor informacija generira beskonačan niz poruka $X_k, k \in \{-\infty, \dots, \infty\}$ iz fiksnog seta poruka S . Ukoliko vjerojatnost poruke ne ovisi o poruci prije nje tada izvor zovemo **neovisan jednako distribuirani izvor** (independent and identically distributed -iid). Entropija takvog izvora zove se **bezuovjetna ili entropija prvog reda**.

Druga vrsta izvora poruka su Markovljevi procesi ili preciznije Markovljevi lanci s diskretnim vremenom. Niz prati Markovljev model k reda ukoliko vjerojatnost svake poruke (ili događaja) ovisi samo o k prethodnih poruka, tj.

$$p(x_n | x_{n-1}, \dots, x_{n-k}) = p(x_n | x_{n-1}, \dots, x_{n-k}, \dots),$$

gdje je x_i i -ta poruka generirana od izvora. Entropija Markovljevog procesa definirana je uvjetnom entropijom, koja se zasniva na uvjetnim vjerojatnostima $p(x_n | x_{n-1}, \dots, x_{n-k})$



Slika 2.2.1: Markovljev model prvog reda s dva stanja

³M. Benšić, N. Šuvak - Uvod u vjerojatnost i statistiku, Odjel za matematiku, Osijek, 2013

Slika 2.2.1 prikazuje nam primjer Markovljevog modela prvog reda. Ovaj Markovljev model predstavlja vjerojatnosti da je izvor generirao crni (black b) ili bijeli (white w) piksel. Svaki luk predstavlja uvjetne vjerojatnosti da je generiran taj piksel. Na primjer, $p(w|b)$ je uvjetna vjerojatnost da će biti generiran bijeli piksel ukoliko je prethodni piksel bio crni. Svaki čvor predstavlja jedno stanje, što je u slučaju Markovljevog modela prvog reda prethodno generirana poruka.

Primjer 2.2.1. Promotrimo slučaj kada su dane vjerojatnosti $p(b|w) = 0.01$, $p(w|w) = 0.99$, $p(b|b) = 0.7$, $p(w|b) = 0.3$, $p(b) = \frac{1}{31}$ i $p(w) = \frac{30}{31}$. Tada entropija iznosi:

$$\frac{30}{31} \left(0.01 \log \frac{1}{0.01} + 0.99 \log \frac{1}{0.99} \right) + \frac{1}{31} \left(0.7 \log \frac{1}{0.7} + 0.3 \log \frac{1}{0.3} \right) \approx 0.107$$

Ovaj broj predstavlja očekivani broj bitova informacije sadržane u svakom pikselu koji izvor generira. Primijetimo da je entropija prvog reda izvora:

$$\frac{30}{31} \log \frac{31}{30} + \frac{1}{31} \log \frac{1}{30} \approx 0.206,$$

koja je skoro dvostruko veća.

Neka je A^n skup stringova duljine n iz abecede A . Tada je normalizirana entropija n -tog reda definirana s

$$H_n = \frac{1}{n} \sum_{x \in A^n} p(x) \log \frac{1}{p(x)}.$$

Kažemo da je entropija normalizirana jer dijelimo s n , što predstavlja količinu informacija po znaku. Entropija izvora tada je definirana kao:

$$H = \lim_{n \rightarrow \infty} H_n.$$

2.3 Prefiks kodovi

Kod C za poruku S je mapiranje poruke u bit string. Svaki bit string nazivamo kodna riječ i kodove ćemo označavati kao $C = \{(s_1, w_1), (s_2, w_2), \dots, (s_m, w_m)\}$. U smislu kompresije, ne želimo da kodne riječi budu fiksne duljine (kao npr. kod ASCII gdje je svaki znak mapiran u 7 bitova), nego želimo da one mogu ovisiti o vjerojatnosti poruke. Pri korištenju kodnih riječi promjenjive duljine može doći do problema pri slanju takvih kodova, jer je teško ili čak i nemoguće odrediti gdje jedna kodna riječ počinje, a druga završava. Na primjer, ako je kod $\{(a,1), (b,01), (c,101), (d,011)\}$, 1011 može biti dekodirano kao **aba**, **ca** ili **ad**. Kako bi se izbjegle ovake zabune može se dodati poseban "stop" znak na kraju svake kodne riječi ili uz svaki znak poslati i duljinu. Međutim, ovaka rješenja zahtijevaju slanje dodatnih podataka. Efikasnije rješenje je dizajnirati kodove koji se uvijek mogu jedinstveno dešifrirati.

Prefiks kod je jedan oblik kodova koji se mogu jedinstveno dešifrirati, gdje nijedan bit string nije prefiks drugom, na primjer $\{(a,1), (b,01), (c,000), (d,001)\}$. Svi prefiksni kodovi mogu se dešifrirati na jedinstven način.

Prednost prefiksni kodova nad svim ostalim kodovima koji se mogu jedinstveno dešifrirati je u tome što se svaka poruka može dešifrirati bez da se gleda početak sljedeće poruke. Ovo svojstvo je važno i dolazi do izražaja kada se šalju poruke različitih tipova (npr. iz različitih vjerojatnosnih distribucija).

Prefiks kod možemo promatrati kao binarno stablo na sljedeći način:

- Svaka poruka je list u stablu.
- Kod za svaku poruku dobije se na način da se prati put od korijena do lista, te se dodaje 0 svaki put kada se uzima lijevo, te 1 kada se uzima desno dijete.

Ovakvo stablo zove se **stablo prefiks koda** te se može koristiti i pri dešifriranju prefiks kodova. Kako bitovi pristižu, dekodirer može pratiti put u stablu dok ne dosegne list, te tada vraća poruku i vraća se u korijen kako bi dešifrirao sjedeći bit.

Neka nam je dana vjerojatnosna distribucija skupa poruka i pridružena varijabilna duljina koda C . Prosječnu duljinu koda definiramo kao:

$$l_a(C) = \sum_{(s,w) \in C} p(s)l(w),$$

gdje je $l(w)$ duljina kodne riječi w . Kažemo da je prefiksni kod C **optimalan** ukoliko je $l_a(C)$ minimalna (tj. ne postoji drugi prefiksni kod za danu vjerojatnosnu distribuciju koji ima manju prosječnu duljinu).

2.3.1 Veza s entropijom

Prosječnu duljinu prefiks kodova možemo povezati s entropijom skupa poruka.

Lema 2.3.1. [Kraft-McMillanova nejednakost]⁴

Za jedinstveni kod C koji možemo dekodirati vrijedi:

$$\sum_{(s,w) \in C} 2^{-l(w)} \leq 1,$$

pri čemu je $l(w)$ duljina kodne riječi w . Nadalje, za svaki skup duljina L takav da je:

$$\sum_{l \in L} 2^{-l} \leq 1,$$

postoji prefiksni kod C jednake duljine takav da je $l(w_i) = l_i$ ($i = 1, \dots, |L|$).

Lema 2.3.2. [Jensenova nejednakost]⁵

Neka je f konveksna funkcija jedne varijable. Neka su $x_1, \dots, x_n \in \mathbb{R}$ i $a_1, \dots, a_n \geq 0$ takvi da je $a_1 + \dots + a_n = 1$. Tada vrijedi:

$$f(a_1x_1 + \dots + a_nx_n) \leq a_1f(x_1) + \dots + a_nf(x_n).$$

Ako je funkcija f konkavna, tada vrijedi:

$$f(a_1x_1 + \dots + a_nx_n) \geq a_1f(x_1) + \dots + a_nf(x_n).$$

Lema 2.3.3. Za svaki skup poruka S s vjerojatnosnom distribucijom i pridruženim kodom C koji se može jedinstveno dešifrirati, vrijedi:

$$H(s) \leq l_a(c).$$

⁴Guy E.Belloch, *Introduction to data compression*, Carneige Mellon University, 2013

⁵Jensens inequality, URL: <https://mathworld.wolfram.com/JensensInequality.html>

Dokaz. Za poruku $s \in S$, $l(s)$ označava duljinu pridruženog koda u C .

$$\begin{aligned}
 H(S) - l_a(C) &= \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)} - \sum_{s \in S} p(s) l(s) \\
 &= \sum_{s \in S} p(s) (\log_2 \frac{1}{p(s)} - l(s)) \\
 &= \sum_{s \in S} p(s) (\log_2 \frac{1}{p(s)} - \log_2 2^{l(s)}) \\
 &= \sum_{s \in S} p(s) \log_2 \frac{2^{-l(s)}}{p(s)} \\
 &\leq \log_2 \left(\sum_{s \in S} 2^{-l(s)} \right) \\
 &\leq 0
 \end{aligned}$$

Pretposljednja linija poziva se na Jensenovu nejednakost, obzirom da je logaritamska funkcija konkavna, dok se posljednja linija poziva na Kraft-McMillanovu nejednakost. \square

Pokazali smo da je entropija donja međa za prosječnu duljinu koda. Pokažimo da je gornja međa ovisna o entropiji optimalnih prefiksnih kodova.

Lema 2.3.4. *Za svaki skup poruka S s vjerojatnosnom distribucijom i pridruženim optimalnim prefiksnim kodom C vrijedi:*

$$l_a(C) \leq H(S) + 1$$

Dokaz. Neka je $s \in S$ poruka kojoj je pridružena njena duljina $l(s) = \lceil \log \frac{1}{p(s)} \rceil$. Tada je

$$\begin{aligned}
 \sum_{s \in S} 2^{-l(s)} &= \sum_{s \in S} 2^{\lceil \log \frac{1}{p(s)} \rceil} \\
 &\leq \sum_{s \in S} 2^{-\log \frac{1}{p(s)}} \\
 &= \sum_{s \in S} p(s) \\
 &= 1
 \end{aligned}$$

Tada po Kraft-McMillanovoj nejednakosti postoji prefiksni kod C' s kodnom riječi duljine $l(s)$. Sada

je

$$\begin{aligned}
 l_a(C') &= \sum_{(s,w) \in C'} p(s)l(w) \\
 &= \sum_{(s,w) \in C'} p(s) \lceil \log \frac{1}{p(s)} \rceil \\
 &\leq \sum_{(s,w) \in C'} p(s) (1 + \log \frac{1}{p(s)}) \\
 &= 1 + \sum_{(s,w) \in C'} p(s) \log \frac{1}{p(s)} \\
 &= 1 + H(S)
 \end{aligned}$$

Po definiciji optimalnog prefiksnog koda, $l_a(C) \leq l_a(C')$. □

Teorem 2.3.1. *Ako je C optimalan prefiksni kod s vjerojatnostima $\{p_1, p_2, \dots, p_n\}$ takav da $p_i > p_j$, onda je $l(c_i) \leq l(c_j)$.*

Dokaz. Pretpostavimo da vrijedi $l(c_i) > l(c_j)$. Promotrimo sada kod koji dobijemo zamjenom c_i s c_j . Ukoliko je l_a prosječna duljina originalnog koda, tada će novi kod imati duljinu

$$\begin{aligned}
 l'_a &= l_a + p_j(l(c_i) - l(c_j)) + p_i(l(c_j) - l(c_i)) \\
 &= l_a + (p_j - p_i)(l(c_i) - l(c_j))
 \end{aligned}$$

Uz naše pretpostavke $(p_j - p_i)(l(c_i) - l(c_j))$ je negativno, što je u kontradikciji s tim da je l_a optimalan prefiksni kod. □

3 Algoritmi za kompresiju bez gubitaka podataka

3.1 Run-length Encoding

Run-Lenght Encoding (**RLE**) je vrlo jednostavan algoritam za kompresiju bez gubitaka podataka. Osnovna ideja algoritma je identificirati dijelove podataka koji se ponavljaju i zamjeniti ponavljajući dio brojem ponavljanja. Na primjer, poruku "bbbdddahhhhhhuu" kopresirali bi u "3b3d2a6h2u". Primijetimo da smo poruku od 16 znakova zapisali u 10. Dakle, što se više jednakih vrijednosti ponavlja to je veći omjer kompresije. S druge strane, ukoliko imamo dani niz podataka u kojem se vrijednosti često izmjenjuju, na primjer "abdard" ovaj algoritam bi ga zapisao kao "1a1b1d1a1r1d", pri čemu se svaki znak iz početnog niza zamjenjuje s dvije znamenke. Možemo zaključiti da je ovaj algoritam prikladan samo za određene podatke, tj. za podatke u kojima postoji velika količina ponavljanja.

Dekompresija podataka koji su kompresirani ovim algoritmom je jednostavna, jer se u kodu nalazi broj ponavljanja nakon kojeg slijedi dio kojeg trebamo ponoviti točno toliko puta. Dakle, kod ovakvog šifriranja i dešifriranja ne dolazi do nikakvog gubitka podataka.

U slučaju RLE algoritma implementacija je dosta jednostavna. Kodiranje i dekodiranje smo proveli na primjeru navedenom gore.


```
def encode(input):
    encoding = ''
    prev = ''
    count = 1

    if not input: return ""

    for char in input:
        if char != prev:
            if prev:
                encoding += str(count) + prev
            count = 1
            prev = char
        else:
            count += 1
    encoding += str(count) + prev
    return encoding
```

```
encoded_val = encode("bbbdddaahhhhhhuu")
print(encoded_val)
```

3b3d2a6h2u

(a) Run-Length Encode

```
def decode(input):
    decode = ''
    count = ''
    for char in input:
        if char.isdigit():
            count += char
        else:
            decode += char * int(count)
            count = ""
    return decode
```

```
decoded_val = decode("3b3d2a6h2u")
print(decoded_val)
```

bbbdddaahhhhhhuu

(b) Run-Length Decode

Slika 3.1.1: Run-Length Python implementacija

3.2 Huffmanov algoritam

Huffmanov algoritam najkorišteniji je među algoritmima za kompresiju podataka. Ovaj algoritam vrlo je jednostavno opisati ukoliko gledamo način na koji on kreira stablo prefiks-kodova.

Algoritam ⁶:

- Započeti s šumom stabala, jednim za svaku poruku. Svako stablo sadrži korijen težine $w_i = p_i$, pri čemu je p_i vjerojatnost poruke.
- Ponavljati dok ne ostane samo jedno stablo:
 - Odabrati dva stabla s najmanjim težinama u korijenu (w_1 i w_2)
 - Spojiti ih u jedno stablo tako da dodamo novi korijen težine $w_1 + w_2$ a početna dva stabla postaju djeca. Nebitno je koje je stablo lijevo, a koje desno dijete, no uobičajeno je postaviti za lijevo stablo ono koje je imalo manju težinu korijena ukoliko je $w_1 \neq w_2$.

Za kod duljine n ovaj algoritam zahtijeva $n - 1$ koraka, obzirom da svako potpuno binarno stablo s n listova sadrži $n - 1$ unutarnjih čvorova i svaki korak kreira jedan unutarnji čvor.

⁶Guy E.Belloch, *Introduction to data compression*, Carnegie Mellon University, 2013

Teorem 3.2.1. *Huffmanov algoritam generira optimalan prefiksni kod⁷.*

Dokaz. Dokaz se provodi indukcijom po broju poruka u kodu. Pokazat ćemo da ako Huffmanov kod generira optimalan prefiksni kod za sve vjerojatnosne distribucije n poruka, tada generira optimalan prefiksni kod za sve distribucije $n + 1$ poruke.

Baza indukcije je trivijalna, obzirom da je prefiksni kod za jednu poruku jedinstven i stoga optimalan.

Pokažimo prvo da za bilo koji skup poruka S postoji optimalan kod u kojem dvije poruke s najmanjim vjerojatnostima imaju istog roditelja u prefiksnom stablu. Po lemi 2.3.1 znamo da su dvije najmanje vjerojatnosti na najnižoj razini u stablu. Također, možemo zamijeniti bilo koja dva lista na najnižoj razini pri čemu se neće promijeniti prosječna duljina koda jer su svi kodovi iste duljine. Stoga možemo uvijek zamijeniti dvije najmanje vjerojatnosti kako bi one imale istog roditelja.

Sada, pretpostavimo da je skup poruka S veličine $n + 1$ i pripadno stablo T dobiveno Huffmanovim algoritmom. Neka su x i y čvorovi u stablu s najmanjom vjerojatnošću, pri čemu oni imaju zajedničkog roditelja zbog načina na koji algoritam kreira stablo. Neka je stablo T' dobiveno zamjenom x i y s njihovim roditeljem z , s vjerojatnošću $p_z = p_x + p_y$. Dubina od z je d i tada je

$$\begin{aligned} l_a(T) &= l_a(T') + p_x(d + 1) + p_y(d + 1) - p_z d \\ &= l_a(T') + p_x + p_y. \end{aligned}$$

Kako bi pokazali da je T optimalno, primijetimo da postoji optimalno stablo u kojem x i y imaju zajedničkog roditelja, i da gdje god se oni nalazili oni uvijek dodaju $p_x + p_y$ na prosječnu duljinu bilo kojeg prefiksnog stabla u kojem su x i y zamijenjeni s njihovim roditeljem z . Po pretpostavci indukcije $l_a(T')$ je minimizirana, obzirom da je T' duljine n i dobiven Huffmanovim algoritmom, i stoga je $l_a(T)$ minimizirana i T je optimalno. \square

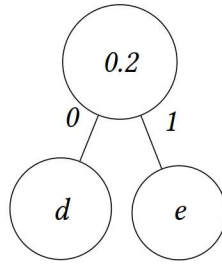
Primjer 3.2.1. *Neka su nam poznate vjerojatnosti pojavljivanja simbola: $p(a) = 0.2$, $p(b) = 0.4$, $p(c) = 0.2$, $p(d) = 0.1$, $p(e) = 0.1$. Pokažimo kako algoritam radi.*

Prvo, sortiramo elemente po vjerojatnostima.

Element	Vjerojatnost
b	0.4
a	0.2
c	0.2
d	0.1
e	0.1

U prvom koraku spajamo elemente s najmanjim vjerojatnostima u stablo, u našem slučaju d i e , na čije grane zapisujemo vrijednosti 0 i 1, a vjerojatnost korijena je 0.2.

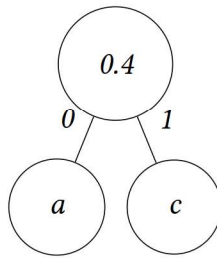
⁷Khalid Sayood, *Introduction to Data Compression*, University of Nebraska, Third edition



Tablica vjerojatnosti sada izgleda ovako:

Element	Vjerojatnost
b	0.4
a	0.2
c	0.2
$d + e$	0.2

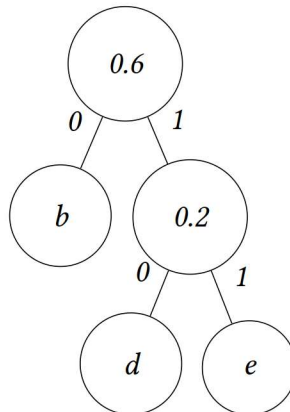
U sljedećem koraku spajamo u stablo opet dva elementa s najmanjim vjerojatnostima, u našem slučaju a i c , te dodajemo novi korijen koji će sada imati vjerojatnost 0.4 .



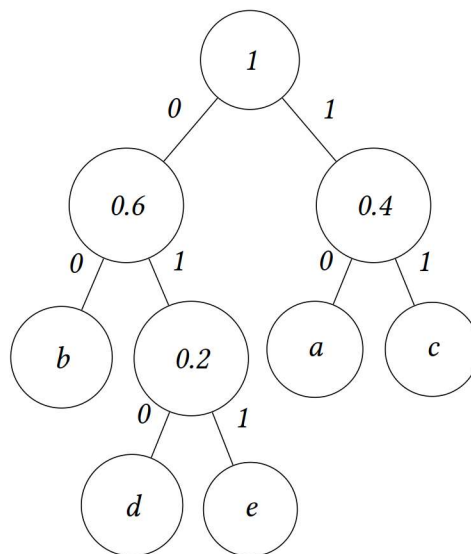
Tablica vjerojatnosti sada izgleda ovako:

Element	Vjerojatnost
b	0.4
$a + c$	0.4
$d + e$	0.2

Postupak analogno ponavljamo, tako da sada spajamo stablo dobiveno u prvom koraku s elementom b .



U posljednjem koraku dobijamo stablo:



Sljedeća tablica prikazuje kodove koje dobijemo za dane simbole koristeći dobiveno stablo.

Element	Kod
<i>a</i>	10
<i>b</i>	00
<i>c</i>	11
<i>d</i>	010
<i>e</i>	011

Možemo primijetiti da smo prilikom kreiranja stabla imali slučajeve kada smo birali između 2 čvora jednake vjerojatnosti. S obzirom na odabrani čvor može se promijeniti duljina koda, ali budući da je Huffmanov kod optimalan, prosječna duljina koda ne može se promijeniti.

```
def frequency (str):
    freqs = {}
    for ch in str :
        freqs[ch] = freqs.get(ch,0) + 1
    return freqs

freqs = frequency("aabbccde")
print (freqs)

{'a': 2, 'b': 4, 'c': 2, 'd': 1, 'e': 1}

def sortFreq (freqs):
    letters = freqs.keys()
    tuples = []
    for let in letters:
        tuples.append((freqs[let], let))
    tuples.sort()
    return tuples

tuples = sortFreq(freqs)
print(tuples)

[(1, 'd'), (1, 'e'), (2, 'a'), (2, 'c'), (4, 'b')]
```

```
def buildTree(tuples) :
    while len(tuples) > 1 :
        leastTwo = (tuples[0], tuples[1])
        theRest = tuples[2:]
        combFreq = leastTwo[0][0] + leastTwo[1][0]
        tuples = theRest + [(combFreq,leastTwo)]
    return tuples[0]
```

```
def trimTree (tree) :
    p = tree[1]
    if type(p) == type(""): return p
    else : return (trimTree(p[0]), trimTree(p[1]))
```

```
tree = buildTree(tuples)
trim = trimTree(tree)
trim = (trim[1],trim[0])
print(trim)

(('b', ('d', 'e')), ('a', 'c'))
```

```
def assignCodes (node, pat='') :
    if type(node) == type(""):
        codes[node] = pat
    else :
        assignCodes(node[0], pat+"0")
        assignCodes(node[1], pat+"1")
```

```
codes = {}
assignCodes(trim)
print(codes)

{'b': '00', 'd': '010', 'e': '011', 'a': '10', 'c': '11'}
```

Slika 3.2.1: Kodiranje Huffmanov algoritam

3.2.1 Huffmanovi algoritmi s minimalnom varijancom

Kao što već naveli, kod Huffmanovog algoritma postoji doza fleksibilnosti kada pri kodiranju naiđemo na jednake vjerojatnosti. Konačan kod i duljina koda mogu se promijeniti obzirom na izbor koji odaberemo. Budući da su svi Huffmanovi kodovi optimalni, to ne mijenja optimalnu duljinu koda. Na primjer, promotrimo opet primjer iz prethodnog poglavlja.

simbol	vjerojatnost	kod 1	kod 2
a	0.2	01	10
b	0.4	1	00
c	0.2	000	11
d	0.1	0010	010
e	0.1	0011	011

Tablica 3.2.1: Različiti kodovi dobiveni Huffmanovim algoritmom

U nekim situacijama korisno je smanjiti varijancu u duljini kodova. Varijanca je definirana kao:

$$\sum_{c \in C} p(c)(l(c) - l_a(c))^2.$$

U 3.2.1 vidimo da kod 1 ima veću varijancu od koda 2. Kako bi Huffmanov algoritam vraćao kodove manje varijance pri odabiru čvorova koje spajamo treba birati onaj čvor koji je kreiran ranije u algoritmu. Smatramo da su listovi u stablu kreirani prije svih unutarnjih čvorova. U primjeru, nakon što su spojeni *d* i *e*, njihov spoj ima istu vjerojatnost kao *c* i *a*, ali je taj čvor kasnije kreiran pa spajamo *c* i *a*. Slično biramo *b* umjesto *a* + *c* i spajamo ga s *d* + *e* jer je kreiran ranije. Ovakav algoritam vraća nam kod 2.

3.3 Aritmetičko kodiranje

Aritmetičko kodiranje je algoritam za kompresiju bez gubitka podataka koji vraća kodove varijabilne duljine. Ono generira kodne riječi za dio poruke (ne za cijelu poruku odjednom). Glavna ideja aritmetičkog kodiranja je predstaviti svaki mogući dio poruke intervalom unutar $[0, 1)$. S obzirom na to da je unutar intervala beskonačno mnogo brojeva, svakom nizu možemo pridružiti jedinstveni interval.

Za dijelove poruka s vjerojatnostima p_1, \dots, p_n algoritam će dijelu pridružiti interval duljine $\prod_{i=1}^n p_i$, počevši od intervala duljine 1 i sužavati ga za faktor p_i za svaki dio poruke i .

Pretpostavit ćemo da dekodier može prepoznati kraj niza, tj. da zna duljinu ili je poslan znak za kraj. Vjerojatnosnu distribuciju skupa poruka označavat ćemo s $\{p(1), \dots, p(m)\}$ i definirat ćemo **akumuliranu vjerojatnost** za vjerojatnosnu distribuciju kao:

$$f(j) = \sum_{i=1}^{j-1} p(i), j = 1, \dots, m$$

Na primjer, vjerojatnostima $\{0.2, 0.5, 0.3\}$ odgovaraju akumulirane vjerojatnosti $\{0, 0.2, 0.7\}$. S obzirom na to da ćemo razmatrati dijelove poruke, pri čemu oni mogu biti iz različitih vjerojatnosnih distribucija, vjerojatnosnu distribuciju i -te poruke označavat ćemo kao $\{p_i(1), \dots, p_i(m_i)\}$ i akumulirane vjerojatnosti kao $\{f_i(1), \dots, f_i(m_i)\}$. Za dani dio poruke vrijednost s indeksom i označavat ćemo kao v_i . Kao skraćeni zapis koristit ćemo p_i za $p_i(v_i)$ i f_i za $f_i(v_i)$.

Aritmetičko kodiranje pridružuje intervale dijelovima poruka na sljedeći način ⁸:

$$l_i = \begin{cases} f_i, & i = 1 \\ l_{i-1} + f_i \cdot s_{i-1}, & 1 < i \leq n \end{cases}$$

$$s_i = \begin{cases} p_i, & i = 1 \\ s_{i-1} \cdot p_i, & 1 < i \leq n \end{cases}$$

gdje je l_n donja granica intervala, a s_n je veličina intervala, tj. interval je određen s $[l_n, l_n + s_n)$. Ovo pravilo sužava interval u svakom koraku na dio prethodnog intervala. Ovakvim načinom pridruživanja intervala svi jedinstveni dijelovi poruka duljine n imat će pridružene intervale koji imaju prazan presjek. Tako ćemo, ako odaberemo interval, moći jedinstveno odrediti dio poruke.

Ostaje nam odrediti kako efikasno slati niz bitova koji predstavlja interval ili broj unutar intervala. Realni brojevi između 0 i 1 mogu se zapisati u binarnom zapisu kao $0.b_1b_2b_3\dots$, npr. $0.75 = 0.11$, $\frac{9}{16} = 0.1001$. Kako bi reprezentirali interval mogli bi odabrati broj iz tog intervala koji ima najmanje bitova i njega koristiti kao kod. Npr. za intervale $[0, 0.33)$, $[0.33, 0.67)$ i $[0.67, 1)$ reprezentirali bi ih redom s $0.01(\frac{1}{4})$, $0.1(\frac{1}{2})$ i $0.11(\frac{3}{4})$. Problem s ovakvim kodovima je što oni nisu prefiksni kodovi. Kako bi izbjegli ovaj problem svaki binarni zapis interpretiramo kao interval. Npr. kod 0.010 reprezentira interval $[\frac{1}{4}, \frac{3}{8})$ jer je najmanji broj koji može nadopuniti kod $0.010\bar{0} = \frac{1}{4}$ i najveći je $0.010\bar{1} = \frac{3}{8} - \epsilon$. Trenutni interval dijela poruke zvat ćemo **sekvencni interval**, a interval koji odgovara vjerojatnosti i -te poruke zvat ćemo **interval poruke**, a interval koda ćemo zvati **kodni interval**. Važno svojstvo kodnih intervala je da postoji veza između preklapanja intervala i jesu li ti kodovi prefiksni.

⁸Guy E. Belloch, *Introduction to data compression*, Carnegie Mellon University, 2013

Lema 3.3.1. *Neka je dan kod C . Ako se nijedna dva intervala predstavljena svojim kodnim riječima $w \in C$ ne preklapaju tada je kod prefiksni.*

Dokaz. Pretpostavimo da je kodna riječ a prefiks kodne riječi b , tada je b moguća nadopuna od a i interval koji tu riječ predstavlja mora biti uključen u interval od a , što je kontradikcija. \square

Kako bi pronašli prefiksni kod, umjesto korištenja bilo kojeg broja iz intervala kako bi ga kodirali, uzimamo kodnu riječ čiji je interval u potpunosti unutar sekvencnog intervala.

Lema 3.3.2. *Za svaki l i s , takve da je $l + s < 1$, interval reprezentiran binarnom reprezentacijom $\frac{l+s}{2}$ i skraćen na $\lceil -\log_2 s \rceil + 1$ bitova sadržan je u intervalu $[l, l + s)$.*

Dokaz. Binarna reprezentacija s l znamenaka predstavlja interval duljine manje od $2^{-l} s$ obzirom na to da razlika između minimalne i maksimalne nadopune su sve jedinice, počevši od $l + i$. lokacije. Ova vrijednost iznosi $2^{-l} - \epsilon$. Interval duljine $\lceil \log_2 s \rceil + 1$ bitova predstavlja vrijednost manju od $\frac{s}{2}$. S obzirom na to da smanjujemo $\frac{l+s}{2}$ gornja granica intervala je manja od $l + s$. Smanjivanjem na $\lceil \log_2 s \rceil + 1$ bitova vrijednost najviše možemo smanjiti na $s/2$. Stoga donja granica za smanjivanje $\frac{l+s}{2}$ je najmanje l . Stoga interval je sadržan u $[l, l + s)$. \square

Algoritam dobiven generiranjem intervala na na ovaj način uz korištenje metode smanjivanja nazivamo **RealArithCode** algoritam.

Teorem 3.3.1. *Za niz od n poruka, s vlastitim sadržajima s_1, \dots, s_n duljina aritmetičkog koda generiranog s RealArithCode omeđena je $s 2 + \sum_{i=1}^n s_i$ i kod neće biti prefiks ni jednom drugom nizu od n poruka.*

Dokaz. S obzirom na način na koji generiramo sekvencni interval on će biti duljine $s = \prod_{i=1}^n p_i$. Po prethodnoj lemi znamo da interval duljine s može biti reprezentiran s $1 + \lceil -\log s \rceil$ bitova, te imamo:

$$\begin{aligned} 1 + \lceil -\log s \rceil &= 1 + \lceil -\log_2 \left(\prod_{i=1}^n p_i \right) \rceil \\ &= 1 + \lceil \sum_{i=1}^n -\log_2 p_i \rceil \\ &= 1 + \lceil \sum_{i=1}^n s_i \rceil \\ &< 2 + \sum_{i=1}^n s_i \end{aligned}$$

Tvrđnja da kod nije prefiks za druge poruke slijedi direktno iz leme 3.3.1. \square

Dekoder za RealArithCode mora dobiti uputu kako čitati input bitove kako bi znao kada je input string završen. Zapravo, algoritam će imati petlju od n interakcija gdje je n broj poruka u nizu.

U praksi postoji nekoliko problema s aritmetičkim kodiranjem kojeg smo opisali. Algoritam mora raditi s velikom preciznošću kako bi točno manipulirao s l i s , što u praksi može biti jako skupo. Drugi problem je što se cijeli input mora kodirati odjednom. Kako bi se izbjegao ovaj problem moguće je poruke razlomiti na dijelove te svaki dio kodirati zasebno aritmetičkim kodiranjem. Ovakav pristup ima prednost u tome što su takvi nizovi fiksne duljine, pa enkoder ne mora slati duljinu poruke, osim možda za posljednju grupu koja može biti kraća od ostalih.

3.3.1 Cjelobrojna implementacija

Korištenjem fiksne preciznosti gubimo na efikasnosti algoritma. Takva implementacija neće davati precizne aritmetičke kodove, ali ako osiguramo da i koder i dekoder zaokružuju na isti način tada će dekoder uvijek moći precizno interpretirati poruku.

Za ovakav algoritam pretpostavit ćemo da su vjerojatnosti dane brojem ponavljanja $c(1), c(2), \dots, c(m)$ i akumulirani broj ponavljanja je definiran kao $f(i) = \sum_{j=1}^{i-1} c(j)$. Ukupan broj označavat ćemo s $T = \sum_{j=1}^m c(j)$. Korištenjem broja ponavljanja izbjeći ćemo potrebu za razlomačkim ili realnim reprezentacijama vjerojatnosti. Umjesto intervala između 0 i 1 koristit ćemo intervale između 0 i $R - 1$ gdje je $R = 2^k$, pri čemu je $R > 4T$. Što je veći R algoritam će biti sličniji RealArithCode algoritmu. Svaka poruka može biti iz svoje vjerojatnosne distribucije kao i ranije.

PSEUDOCODE 1 (IntArithCode).

```
function IntArithCode(file, k, n)
     $R = 2^k$ 
     $l = 0$ 
     $u = R - 1$ 
     $m = 0$ 
    for  $i = 1$  to  $n$ 
         $s = u - l + 1$ 
         $u = l + \lfloor s * \frac{f_i(v_i+1)}{T} \rfloor - 1$ 
         $l = \lfloor s * \frac{f_i(v_i)}{T} \rfloor$ 
        while true
            if  $(l \geq \frac{R}{2})$ 
                WriteBit(1)
                 $u = 2u - R + 1$ 
                 $l = 2l - R$ 
                for  $j = 1$  to  $m$  WriteBit(0)
                 $m = 0$ 
            else if  $(u < \frac{R}{2})$ 
                WriteBit(0)
                 $u = 2u + 1$ 
                 $l = 2l$ 
                for  $j = 1$  to  $m$  WriteBit(1)
                 $m = 0$ 
            else if  $(l \geq \frac{R}{4} \text{ and } u < \frac{3R}{4})$ 
                 $u = 2u - \frac{R}{2} + 1$ 
                 $l = 2l - \frac{R}{2}$ 
                 $m = m + 1$ 
            else continue
        end while
    end for
    if  $(l \geq \frac{R}{4})$ 
        WriteBit(1)
    for  $j = 1$  to  $m$  WriteBit(0)
```

```

    WriteBit(0)
else
    WriteBit(0)
    for j = 1 to m WriteBit(1)
    WriteBit(1)

```

PSEUDOCODE 2 (IntArithDecode).

```

function IntArithDecode(file, k, n)
    R = 2k
    l = 0
    u = R - 1
    lb = 0
    ub = R - 1
    j = 1
    while j ≤ n do
        s = u - l + 1
        i = 0
        do
            i = i + 1
            u' = l + ⌊s *  $\frac{f_j(i+1)}{T_j}$  - 1⌋
            l' = l + ⌊s *  $\frac{f_j(i)}{T_j}$ 
        while i ≤ mj and not((lb ≥ l') and (ub ≤ u'))
        if i > mj then
            b = ReadBit(file)
            sb = ub - lb + 1
            lb = lb + b *  $\frac{s_b}{2}$ 
            ub = lb +  $\frac{s_b}{2}$  - 1
        else
            Output(i)
            u = u'
            l = l'
            j = j + 1
        while true
            if (l ≥  $\frac{R}{2}$ )
                u = 2u - R + 1
                l = 2l - R
                ub = 2ub - R + 1
                lb = 2lb - R
            else if (u <  $\frac{R}{2}$ )
                u = 2u + 1
                l = 2
                ub = 2ub + 1
                lb = 2lb
            else if (l ≥  $\frac{R}{4}$  and u <  $\frac{3R}{4}$ )

```


$$\begin{aligned}
u &= 2u - \frac{R}{2} + 1 \\
l &= 2l - \frac{R}{2} \\
u_b &= 2u_b - \frac{R}{2} + 1 \\
l_b &= 2l_b - \frac{R}{2} \\
&\textit{else continue} \\
&\textit{end if} \\
&\textit{end while}
\end{aligned}$$

Algoritmi za kodiranje i dekodiranje opisani su u 1 i 2. Trenutni interval određen je je s l i u , tj interval $[l, u + 1)$ je trenutni interval. Ideja algoritma je da je veličina intervala uvijek veća od $\frac{R}{4}$ tako što interval proširujemo uvijek kad on postane premali (while petlja u algoritmu). U ovoj petlji, kada god se sekvenčni interval u potpunosti nalazi u gornjoj polovici intervala (između $\frac{R}{2}$ i R) znamo da će sljedeći bit biti 1 jer se interval može samo smanjivati. Tada dodajemo 1 u output i proširujemo gornju polovicu intervala. Slično, ukoliko je sekvenčni interval unutar donje polovice intervala u output dodajemo 0 i proširujemo donju polovicu intervala. Treći slučaj je kada se sekvenčni interval nalazi u sredini intervala (između $\frac{R}{4}$ i $\frac{3R}{4}$). U ovom slučaju ne možemo u output dodati ni 0 ni 1, jer ne znamo što dodati. Međutim, možemo proširiti taj interval i pratiti broj njegovih proširivanja s brojačem m . Kada algoritam proširi gornju (donju) polovicu u output dodajemo 1 (0) koju prati m 0 (1). Kako bi pojasnili zašto je ovakav način ispravan promatrajmo proširivanje srednjeg intervala m puta i nakon toga gornjeg dijela. Prvo proširivanje srednjeg dijela nalazi se između $\frac{1}{4}$ i $\frac{3}{4}$ početnog intervala, drugo između $\frac{3}{8}$ i $\frac{5}{8}$. Nakon m proširivanja interval je smanjen na područje $(\frac{1}{2} - \frac{1}{2^{m+1}}, \frac{1}{2} + \frac{1}{2^{m+1}})$ od početnog intervala. Sada, kada proširujemo gornju polovicu interval je smanjen na $(\frac{1}{2}, \frac{1}{2} + \frac{1}{2^{m+1}})$. Svi intervali koji se nalaze u ovom području počinjat će s 1 koji prati m 0.

Promotrimo sada kako algoritam završava. Kao i kod RealArthCode algoritma, kako bi uvijek mogli jedinstveno dekodirati želimo osigurati da kod za jedan dio poruke nije prefiks nijednom drugom kodu za drugi dio poruke. To možemo osigurati na način da je kodni interval u potpunosti sadržan u sekvenčnom intervalu. Kada IntArthCode izađe iz **for** petlje, znamo da sekvenčni interval $[l, u]$ u potpunosti pokriva ili dio od $\frac{R}{4}$ do $\frac{R}{2}$ ili od $\frac{R}{2}$ do $\frac{3R}{4}$, jer u slučaju da to nije istina primijenili bi pravilo proširivanja. Algoritam tada otkriva koji od ova dva dijela interval pokriva i vraća kod koji sužava kodni interval za jedan od ta dva dijela, 01 za $\frac{R}{4}$ do $\frac{R}{2}$ (jer sve nadopune od 01 se nalaze u tom dijelu) i 10 za $\frac{R}{2}$ do $\frac{3R}{4}$. Nakon što algoritam vrati prva dva bita kod nadopunjuje s m bitova koji odgovaraju prijašnjim proširivanjima u sredini.

Primjer 3.3.1. Promotrimo primjer kodiranja niza poruka iz iste vjerojatnosne distribucije, pri čemu je $c(1) = 1$, $c(2) = 10$, $c(3) = 20$. Akumulirane vrijednosti su $f(1) = 0$, $f(2) = 1$, $f(3) = 11$ i $T = 31$. Neka je $k=8$ i $R = 256$. Ovo zadovoljava uvjet da je $R > 4T$. Promotrimo sada kodiranje niza poruka 3,1,2,3.

Svaki red u tablici predstavlja korak u algoritmu. Svaki red koji počinje s + predstavlja korištenje jednog od pravila proširivanja.

i	v	$f(v_i)$	$f(v_i + 1)$	l	u	s	m	pravilo proširivanja	output
start				0	255	256			
1	3	11	31	90	255	166	0		
2	2	1	11	95	147	53	0		
+				62	167	106	1	srednja polovina	
3	1	0	1	62	64	3	1		
+				124	129	6	0	donja polovina	01
+				120	131	12	1	srednja polovina	
+				112	135	24	2	srednja polovina	
+				96	143	48	4	srednja polovina	
+				64	159	96	5	srednja polovina	
+				0	191	192	6	srednja polovina	
4	2	1	11	6	67	62	6		
+				12	135	124	0	donja polovina	0111111
end							0		01

Algoritam vraća kod: 01011111101.

Promotrimo sada na koji način algoritam kojeg smo ranije opisali dekodira poruku. Glavna ideja je odvojiti gornju i donju granicu za kodni interval (l_b i u_b) i sekvenčni interval (l i u). Algoritam čita bit po bit i smanjuje kodni interval u pola za svaki bit koji je pročitan (donja polovica kada je bit 0 i gornja polovica kada je bit 1). Kad god kodni interval pada u interval za iduću poruku, poruka je output i sekvenčni interval se smanjuje za interval poruke. Sekvenčni interval prati isti skup donjih i gornjih granica kao i kod kodiranja.

3.4 Lempel-Ziv algoritmi

Lempel-Ziv algoritmi vrše kompresiju na način da kreiraju rječnik od već pređenih stringova. Ovi algoritmi kodiraju grupe znakova promjenjive duljine. Originalni algoritmi nisu koristili vjerojatnosti ponavljanja stringova, tj. stringovi su ili bili u rječniku ili nisu i svi koji su se pojavljivali u rječniku imali su jednaku vjerojatnost.

Algoritme možemo najjednostavnije opisati na sljedeći način. S obzirom na poziciju u datoteci na kojoj se nalazimo, proći kroz dio datoteke koji kodiramo i pronaći najdulji string koji se podudara sa stringom koji počinje na poziciji na kojoj se nalazimo, te vratiti kod koji se odnosi na to podudaranje. Nakon toga se pomičemo iza tog podudaranja.

Dvije osnovne varijante algoritama opisali su Lempel i Ziv 1977. i 1978. godine, te ih zbog toga najčešće nazivaju LZ77 i LZ78. Algoritmi se razlikuju u tome koliko daleko naprijed i na koji način traže podudaranja.

3.4.1 LZ77 - Pomični prozori

Algoritam LZ77 i njegove varijante koriste pomične prozore koji se pomiču zajedno s pokazivačem. Prozor možemo podijeliti na dva dijela, dio prije pokazivača koji nazivamo rječnik, te dio koji počinje kod pokazivača koji nazivamo međuspremnik. Veličina ova dva dijela mijenja se prilikom izvođenja

algoritma. Algoritam ⁹ se izvodi tako da naizmjenice izvodimo sljedeća 3 koraka :

- Pronađi najveće podudaranje stringa koji počinje na mjestu pokazivača i kompletno se nalazi u međuspremniku sa stringom koji se nalazi u rječniku.
- Ispiši uređenu trojku (p, n, c) , gdje je p pozicija na kojoj se nalazimo u prozoru, n duljina podudaranja i c znak koji slijedi nakon podudaranja.
- Pomakni pokazivač za $n + 1$ znak unaprijed.

Kako bi dekodirali poruku, promatramo jedan korak. Induktivno, pretpostavljamo da je dekoder točno konstruirao string do trenutnog pokazivača te želimo pokazati da dana trojka (p, n, c) može rekonstruirati string do sljedećeg pokazivača. Kako bi to napravili, dekoder može pogledati string unazad za p pozicija i uzeti sljedećih n znakova, te nakon toga staviti znak c . Jedini problem koji može nastati je ukoliko je $n > p$. Problem je što se poklapanje onda nalazi u međuspremniku koji dekoder još nije ispunio. U tom slučaju, dekoder može rekonstruirati poruku uzimajući p znakova prije pokazivača i ponavlja ih dovoljno puta dok ne popuni n pozicija. Ako je npr. kod bio $(2, 7, d)$ i 2 znaka ispred pokazivača su ab , tada bi algoritam vratio $abababa$ i znak d postavio nakon pokazivača.

3.4.2 LZ78

LZW je najkorištenija varijanta LZ78 algoritma. Pretpostavit ćemo da algoritam kodira bajt-stringove (svaka poruka je bajt). Algoritam radi s rječnikom stringova. Rječnik je inicijaliziran za svaki od 256 mogućih bajtnih vrijednosti i ti stringovi su stringovi duljine 1. Kako algoritam napreduje dodavat će nove stringove u rječnik na način da je string dodan ako je prefiks jedan bajt kraći. Npr, John je dodan u rječnik ako se Joh već pojavio u poruci.

Pretpostavimo da je svakom unosu u rječniku dodan indeks, pri čemu se indeksi dodaju inkrementalno počevši od 256 (prvih 256 je rezervirano za bajtne vrijednosti). Prije nego predstavimo pseudokodove za ovaj algoritam, promotrimo funkcije koje ćemo koristiti:

- $C' = \text{AddDict}(C, x)$ - kreira novi unos u rječnik, tako da element u rječniku s indeksom C proširuje za bajt x i vraća indeks novog unosa.
- $C' = \text{GetIndex}(C, x)$ - Vraća indeks stringa dobivenog produljenjem stringa kojem odgovara indeks C s bajtom x . Ukoliko unos ne postoji vraća -1 .
- $W = \text{GetString}(C)$ - Vraća string kojem odgovara indeks C .
- $\text{Flag} = \text{IndexInDict?}$ Vraća true ako je indeks C u rječniku, inače vraća false.

PSEUDOCODE 3 (LZW - Encode).

```
function LZW_Encode(File)
  C = ReadByte(File)
  while C  $\neq$  EOF do
    x = ReadByte(File)
    C' = GetIndex(C,x)
```

⁹Guy E.Belloch, *Introduction to data compression*, Carneige Mellon University, 2013

```

while  $C' \neq -1$  do
     $C = C'$ 
     $x = \text{ReadByte}(\text{File})$ 
     $C' = \text{GetIndex}(C, x)$ 
    Output( $C$ )
    AddDict( $C, x$ )
     $C = x$ 

```

PSEUDOCODE 4 (LZW - Decode).

7

```

function LZW_Decode(File)
     $C = \text{ReadIndex}(\text{File})$ 
     $W = \text{GetString}(C)$ 
    Output( $W$ )
    while  $C \neq \text{EOF}$  do
         $C' = \text{ReadIndex}(\text{File})$ 
        if IndexInDict?( $C'$ ) then
             $W = \text{GetString}(C')$ 
            AddDict( $C, W[0]$ )
        else
             $C' = \text{AddDict}(C, W[0])$ 
             $W = \text{GetString}(C')$ 
        Output( $W$ )
     $C = C'$ 

```

Algoritam za kodiranje opisan je u 3. Svaka iteracija vanjske petlje radi tako da prvo pronalazi najveće podudaranje W u rječniku za string koji počinje na trenutnoj poziciji (unutarnja petlja pronalazi to podudaranje). Iteracija tada vraća indeks za W i dodaje string Wx u rječnik, pri čemu je x sljedeći znak nakon podudaranja. Korištenje rječnika je slično kao u LZ77 samo što direktno spremamo rječnik, a ne njegove indekse u prozoru. S obzirom na to da eksplicitno koristimo rječnik, svaki indeks odgovara točno stringu, pa kod LZW algoritma ne moramo specificirati duljinu stringa.

Algoritam opisan u 4 točno će dekodirati s obzirom na to da kreira rječnik na isti način kao i koder, i u suštini možemo samo gledati indekse koje će primiti u svojoj kopiji rječnika. Problem je što je rječnik dekodera uvijek korak iza kodera. Do toga dolazi jer koder može dodati Wx u rječnika u trenutnoj iteraciji, ali dekoder će x primiti tek u sljedećoj poruci. Jedini slučaj kada bi to mogao biti problem je kada koder šalje indeks za string W a string je praćen s $WW[0]$, gdje $W[0]$ označava prvi znak od W . U iteraciji u kojoj koder šalje indeks za W dodaje $WW[0]$ u rječnik, koji se u njemu u tom trenutku još ne nalazi. Međutim, s obzirom na to da je dekoder u mogućnosti dekodirati W , može rekonstruirati i $WW[0]$, što radi **else** petlja u *LZW_decode*.

Nedostatak ovog algoritma je što rječnik može postati prevelik. Postoji nekoliko načina što napraviti u tom slučaju:

- Odbaciti rječnik ukoliko dosegne određenu veličinu. (GIF)
- Odbaciti rječnik kada nije koristan. (Unix Compress)
- Odbaciti najmanje korištene unose u rječniku kada dosegne određenu veličinu. (BLTZ - British Telecom Standard)

Primjer 3.4.1. LZW Kodiranje i dekoriranje *abcabca*. Redovi označeni s + pri kodiranju su iteracije unutarnje while petlje.

	C	x	GetIndex(C,x)	AddDict(C,x)	Output(C)
init	a				
	a	b	-1	256 (a,B)	a
	b	c	-1	257 (b,c)	a
	c	a	-1	258(c,a)	c
+	a	b	256		
	256	c	-1	259(256,c)	256
+	c	a	258		
	258	EOF	-1	-	258

Tablica 3.4.1: Kodiranje

	C	C'	W	IndexInDict?(C')	AddDict(C, W[0])	Output(W)
Init	a		a			a
	a	b	b	true	256 (a,b)	b
	b	c	c	true	257 (b,c)	c
	c	256	ab	true	258 (c,a)	ab
	256	258	ca	true	256 (256,c)	ca

Tablica 3.4.2: Dekodiranje

Primjer 3.4.2. LZW kodiranje i dekodiranje *aaaaaa*. Ovo je primjer u kojem dekoder nema indeks u rječniku.

	C	x	GetIndex(C,x)	AddDict(C,x)	Output(C)
init	a				
	a	a	-1	256 (a,a)	a
+	a	a	256		
	256	a	-4	257 (256,a)	256
+	a	a	256		
+	256	a	257		
	257	EOF	-1	-	257

Tablica 3.4.3: Kodiranje

	C	C'	W	IndexInDict?(C')	AddDict(C,W[0])	Output(W)
Init	a		a			a
a	256	aa	false	256 (a,a)	aa	
	256	257	aaa	false	257 (256,a)	aaa

Tablica 3.4.4: Dekodiranje

4 Algoritmi za kompresiju s gubitcima podataka

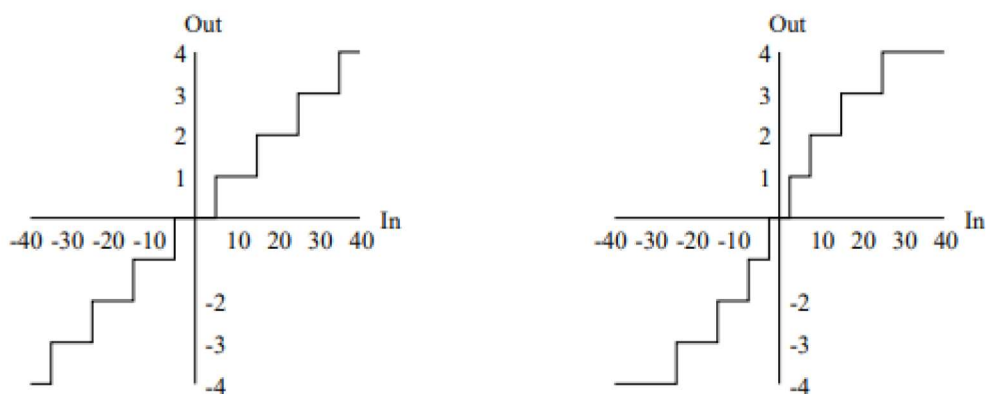
4.1 Tehnike koje se koriste pri kompresiji s gubitcima podataka

4.1.1 Skalarna kvantizacija

Jednostavan način na koji možemo implementirati kompresiju podataka s gubicima je da početni skup podataka S reduciramo na manji skup S' tako da svakom elementu iz skupa S pridružimo element iz skupa S' . Na primjer, neka je skup $S = \{\text{skup svih velikih i malih slova abecede}\}$ i $S' = \{\text{skup svih velikih slova abecede}\}$. Mala slova u poruci bi u ovom slučaju zamijenili s velikim slovima. Ovakva tehnika naziva se **kvantizacija**¹⁰. S obzirom na to da je pridruživanje više prema jedan, ovakvo pridruživanje je nepovratno i stoga dolazi do gubitka podataka.

U slučaju kada je skup podataka totalno uređen, totalni uređaj je podijeljen po dijelovima u skupu S' i takvo pridruživanje zovemo **skalarna kvantizacija**. Skalarna kvantizacija koristi se kako bi se smanjio broj bitova koji predstavljaju boje u fotografijama ili razine kod grayscale slika, te kod klasifikacija intenziteta frekvencija komponenata kod slika ili zvučnih zapisa (koristi se u JPEG kompresiji).

Ukoliko je opisano pridruživanje linearno tada kažemo da se radi o **uniformnoj skalarnoj kvantizaciji**. U praksi je često bolje koristiti neuniformnu skalarnu kvantizaciju. Na primjer, ljudsko oko je osjetljivije na niske vrijednosti crvene boje nego na visoke. Stoga možemo postići bolju kompresiju kod slika tako da područja s niskim vrijednostima budu manja nego ona s visokim.



Slika 4.1.1: Primjer uniformne i neuniformne skalarne kvantizacije

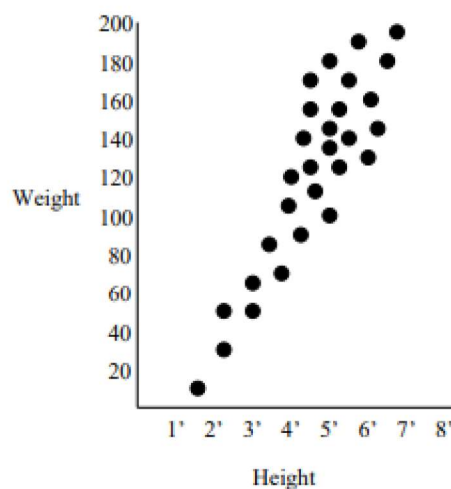
¹⁰Kvantizacija, URL:<https://hr.wikipedia.org/wiki/Kvantizacija>

4.1.2 Vektorska kvantizacija

Skalarna kvantizacija omogućava da svakoj boji u slici pridružimo boju iz manjeg seta boja. Međutim, u praksi je efikasnije pridruživanje raditi za trodimenzionalno područje boja. Na takav način možemo postići bolji omjer kompresije uz manje gubitke podataka.

Mapiranje višedimenzionalnog prostora na manji skup nazivamo **vektorska kvantizacija**. Vektorsku kvantizaciju implementiramo na način da odabiremo skup reprezentanata iz ulaznih podataka, i svim ostalim točkama u prostoru pridružimo najbližeg reprezentanta. Reprezentanti mogu biti fiksni kroz cijeli proces kompresije ili mogu biti određeni posebno za svaki dio kompresije. Uobičajeno je implementirano na način da se koriste algoritmi za klasteriranje koji pronalaze određen broj klastera u podacima. Reprezentant je onda odabran za svaki klaster tako da odaberemo jednu točku iz klastera ili tražimo najboljeg reprezentanta klastera.

Najveću efikasnost vektorska kvantizacija ima na podacima koji su u korelaciji.



Slika 4.1.2: Korelirani podaci

Na slici 4.1.2 vidimo da postoji jaka korelacija između visine i težine osobe. Reprezentanti će se onda nalaziti u područjima gdje su podaci gušći. Korištenje ovakvih reprezentanata je efikasnije nego korištenje skalarne kvantizacije zasebno na visini i težini.

Vektorska, kao i skalarna kvantizacija, može se koristiti i pri kompresiji bez gubitaka podataka. U slučaju vektorske kvantizacije kod kodiranja, osim slanja najboljeg reprezentanta, prosljedili bi i udaljenost točke do tog reprezentanta kako bi pri dekodiranju originalna točka mogla biti rekonstruirana.

4.1.3 Transformacijsko kodiranje

Transformacijsko kodiranje ulazne podatke transformira u drugačiji oblik podataka koji mogu biti bolje komprimirani ili u oblik podataka u kojima jednostavnije možemo izostaviti određeni dio podataka pri čemu će biti smanjeni gubitci prilikom kompresije. Jedan oblik transformacije je da odaberemo baznu funkciju (ϕ_i) koja obuhvaća prostor koji želimo transformirati. Neke od najčešće korištenih baznih funkcija su sin, cos, polinomijalne i wavelet funkcije.

Za skup od n vrijednosti, transformaciju možemo zapisati u obliku matrice T dimenzija $n \times n$. Množenjem ulaznih vrijednosti matricom T dobijemo **transformacijske koeficijente**. Množenjem koeficijenata matricom T^{-1} dobijemo opet originalne podatke.

Na primjer, koeficijenti za **diskretnu kosinusnu transformaciju** su:

$$T_{ij} = \begin{cases} \sqrt{\frac{1}{n}} \cos \frac{(2j+1)i\pi}{2n}, & i = 0, 0 \leq j < n \\ \sqrt{\frac{2}{n}} \cos \frac{(2j+1)i\pi}{2n}, & 0 < i < n, 0 \leq j < n \end{cases}$$

Diskretna kosinusna transformacija najčešće se koristi u kompresiji slika.

U smislu kompresije, transformacijom želimo postići sljedeće:

- maknuti korelaciju između podataka
- postići da su transformacijski koeficijenti što manji
- postići da su, u smislu percepcije, određena svojstva ulaznih podataka važnija od drugih

4.2 JPEG

JPEG je algoritam za kompresiju s gubitcima grayscale i slika u boji, koji je dizajniran kako bi se koristio u fotografskim radovima. Ne preporuča se korištenje ovog algoritma za linijske crteže, tekstualne slike ili slike s velikim područjima iste boje ili limitiranim brojem različitih boja. JPEG je dizajniran tako da promjenom faktora gubitka biramo između smanjenja veličine slike i kvalitete slike, te je dizajniran tako da je gubitak što manje uočljiv ljudskoj percepciji. Međutim anomalije su uočljive kada je omjer kompresije velik.

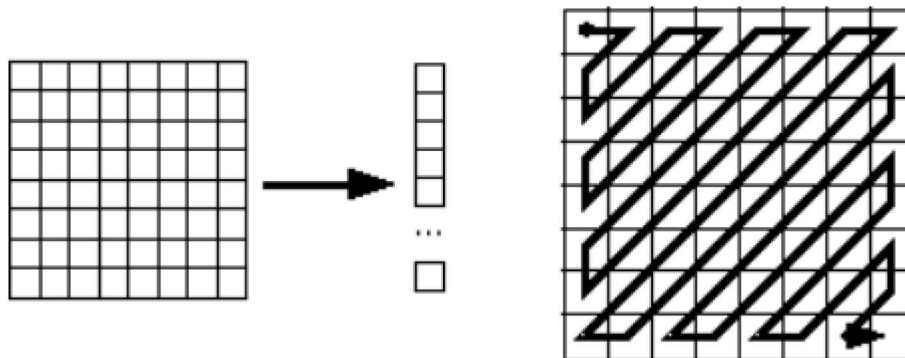
Input za JPEG algoritam su tri ravnine od 8 bitova po pikselu koje svaka reprezentiraju crvenu, plavu i zelenu boju (RGB). Prvi korak u JPEG kompresiji je konvertirati ulaz u alternativni oblik YIQ. Y ravnina je dizajnirana kako bi reprezentirala jačinu (brightness) slike. To je težinska aritmetička sredina crvene, plave i zelene ($0.59 \text{zelena} + 0.3 \text{crvena} + 0.11 \text{plava}$). Težine nisu ravnomjerno raspoređene s obzirom na to da je ljudsko oko osjetljivije na zelenu nego na crvenu, te osjetljivije na crvenu nego na plavu boju. I i Q komponente predstavljaju nijanse boja. Razlog zašto se koristi konverzija u YIQ format je da je važnije (u smislu percepcije) zadržati intenzitet nego nijansiranje slike. Stoga JPEG zadržava sve piksele za intenzitet, ali smanjuje ostaje komponente za faktor 2 (konačno za faktor 4). To je prvi korak u kojem gubimo podatke te dobijemo faktor kompresije: $(1 + 2 \cdot 0.25)/3 = 0.5$

Sljedeći korak u JPEG algoritmu je podijeliti svaku od ravnina na blokove veličine 8×8 . Svaki od ovih blokova će biti kodiran posebno. Prvi korak pri kodiranju svakog od blokova je primijeniti kosinusnu transformaciju. To nam vraća 8×8 blok 8-bitnih frekvencijskih uvjeta. Nakon kosinusne transformacije, sljedeći korak je korištenje uniformne skalarne kvantizacije na svaki frekvencijski uvjet. Kvantizacija se može kontrolirati pomoću ulaznih parametara i ona je glavni uzrok za gubitak podataka u JPEG kompresiji. S obzirom na to da je ljudsko oko osjetljivije na određene frekvencije komponenti više nego na frekvencije drugih, JPEG dozvoljava da je faktor skaliranja kod kvantizacije različit za svaku frekvencijsku komponentu. Faktori skaliranja su zadani u obliku 8×8 tablice koji redom odgovaraju 8×8 tablici frekvencijskih komponenti.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Tablica 4.2.1: Standardna JPEG kvantizacijska tablica

U tablici 4.2.1 prikazana je tablica za Y. U tablici se najveće komponente nalaze u donjem desnom kutu. To je zato jer su to frekvencijske komponente na koje je ljudsko oko manje osjetljivo nego na komponente u gornjem lijevom kutu. Izbor brojeva u tablici se čini nasumičan, ali je baziran na ljudskoj percepciji. Ukoliko želimo, pri kodiranju možemo koristiti i drugačiju kvantizacijsku tablicu. Kako bi dalje kompresirali sliku, cijelu tablicu rezultata možemo podijeliti nekom konstantom, koja je skalar definiran od korisnika. Kvantizacijom ćemo često smanjiti elemente u donjem lijevom kutu na nulu. JPEG kompresija tada kompresira DC komponentu (gornji lijevi element) odvojeno od ostalih komponenti. Točnije, vrijednost DC komponente oduzima se od DC vrijednosti sljedećeg bloka. Nadalje, ta razlika se kodira Huffmanovim ili aritmetičkim algoritmom. AC komponente (ostale komponente) se sada kompresiraju. Prvo se konvertiraju u linearni red tako da se cik-cak pomičemo po frekvencijskoj tablici, kako je prikazano na slici.



Slika 4.2.1: Cik-cak pomicanje kroz JPEG blokove

Ovakvo pomicanje kroz tablicu koristi se kako bi frekvencije koje su aproksimacijski jednake duljine bile blizu jedna drugoj u linearnom redu. Preciznije, većina nula će se pojaviti kao veliki blok na kraju reda. Kako bi kodirali dobiveni linearni red koristimo Run-length algoritam. Kodira se kao niz (skip,value) parova, gdje je skip broj nula prije vrijednosti, a value je vrijednost. Par (0,0) označava kraj bloka. Na primjer, niz [4,3,0,0,1,0,0,0,1,0,0,0,...] reprezentira se kao [(0,4),(0,3), (2,1), (3,1),(0,0)]. Ovaj iz se dalje kompresira koristeći aritmetičko ili Huffmanovo kodiranje.

Implementaciju JPEG algoritma pokazat ćemo na grayscale slici.

```
In [ ]: import numpy as np
import pandas as pd
import matplotlib
from matplotlib import pyplot as plt
import colorsys
from PIL import Image
from scipy.fftpack import dct, idct
import os
from IPython.display import Image
```

```
In [2]: def showImage(img):
plt.figure(figsize=(15,15))
plt.imshow(img,cmap='gray')
plt.xticks([], plt.yticks([]))
plt.show
```

```
In [3]: #definirajmo kvantizacijsku tablicu

Q = np.array([[16,11,10,16,24,40,51,61],
              [12,12,14,19,26,58,60,55],
              [14,13,16,24,40,57,69,56],
              [14,17,22,29,51,87,80,62],
              [18,22,37,56,68,109,103,77],
              [24,35,55,64,81,104,113,92],
              [49,64,78,87,103,121,120,101],
              [72,92,95,98,112,100,130,99]])
```

```
In [4]: # prebacimo rgb sliku u graycale
def rgb2gray(rgb):
    gray = rgb[:, :, 0]

    return gray
```

```
In [5]: from skimage import color
from skimage import io

img = rgb2gray(io.imread('last'))
showImage(img)
```



```
In [6]: img.shape
```

```
Out[6]: (600, 800)
```

```
In [7]: height= len(img)
width = len(img[0])
sliced=[]
block = 8
print("The image height is " + str(height)+", and image width is " + str(width)+ " piksels")
```

The image height is 600, and image width is 800 piksels

```
In [8]: #podjela na bolokove 8x8
```

```
currY=0 #trenutni Y
for i in range(block, height+1,block):
    currX =0
    for j in range(block,width+1,block):
        sliced.append(img[currY:i,currX:j]-np.ones((8,8))*128)
        currX=j
    currY = i

print("Size of the sliced image: " + str(len(sliced)))
print("Each element of sliced list contains a " + str(sliced[0].shape)+ " element")
```

Size of the sliced image: 7500
Each element of sliced list contains a (8, 8) element

```
In [10]: DCToutput=[]
for slice in sliced:
    currDCT =dct(slice, type=3)
    DCToutput.append(currDCT)
DCToutput[0][0]
```

```
Out[10]: array([308.2908326 , 46.39211211, -25.826151 , -32.87505281,
 0.64617311, 159.63295476, 11.80108413, -84.0619529 ])
```

```
In [11]: for ndct in DCToutput:
    for i in range(block):
        for j in range(block):
            ndct[i,j]= np.around(ndct[i,j]/Q[i,j])
```

```
DCToutput[0][0]
```

```
Out[11]: array([19., 4., -3., -2., 0., 4., 0., -1.])
```

```
In [12]: #rekonstrukcija slike
invList=[]
for ipart in DCToutput:
    ipart
    curriDCT = idct(ipart,type=3)
    invList.append(curriDCT)
invList[0][0]
```

```
Out[12]: array([42.          , 37.32482356, 39.25128189, 47.65133677, 15.55634919,
 8.31938412, 9.7640601 , -4.35838243])
```

```
In [13]: row = 0
rowNcol = []
for j in range(int(width/block),len(invList)+1,int(width/block)):
    rowNcol.append(np.hstack((invList[row:j])))
    row = j
res = np.vstack((rowNcol))
```




(a) Originalna slika



(b) Slika dobivena JPEG algoritmom

Slika 4.2.2: Usporedba originalne slike i slike nakon JPEG kompresije

4.3 MPEG

Korelacija među podacima poboljšava kompresiju. U svim pristupima kompresiji možemo primijetiti da što bolje možemo uspostaviti korelaciju među podacima, možemo bolje kompresirati podatke. Ovaj princip je najizraženiji u MPEG kdiranju, koji kompresira video. U teoriji, video je niz fotografija. U praksi, uzastopne fotografije su blisko povezane osim u slučajevima promjene scene. MPEG iskorištava jaku korelaciju između fotografija kako bi se postigla bolja kompresija od one koju možemo postići kompresiranjem pojedinih fotografija. Svaki isječak u MPEG-u kodira se koristeći jednu od sljedeće tri sheme:

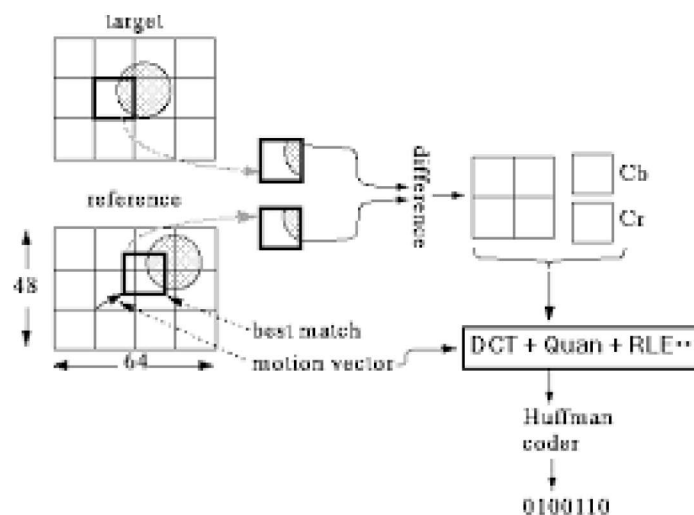
- **I-isječak** : kodiraju se izolirane fotografije.
- **P-isječak** : (prediktivno kodiraj isječak) bazira se na prethodnom I ili P isječku.
- **B-isječak**: bazira se na prethodnom i sljedećem I ili P isječku.

Playback order:	0	1	2	3	4	5	6	7	8	9
Frame type:	I	B	B	P	B	B	P	B	B	I
Data stream order:	0	2	3	1	5	6	4	8	9	7

Tablica 4.3.1: Vizualizacija pomicanja B-isječaka

Tablica 4.3.1 nam prikazuje MPEG niz koji sadrži sva tri tipa isječaka. I-isječak i P-isječak pojavljuju se u MPEG nizu u kronološkom poretku. Međutim, B-isječci se pomiču kako bi se pojavili nakon susjednih I i P-isječaka. To garantira da će se svaki isječak pojaviti nakon svih isječaka o kojima može ovisiti. Kada je scena u videu relativno statična koriste se P i B-isječci, dok kod čestih promjena scene koristi se I-isječak. U praksi, pri kodiranju se koriste fiksni uzorci. S obzirom na to da su I-isječci neovisne slike, možemo ih kodirati kao da su obične slike. Točnije, za kodiranje koristimo neku varijantu JPEG-a. I-isječci su važni jer se koriste kao sidrišta tako da se nasumično može pristupiti isječcima iz videa bez da se moraju dekodirati svi prethodni isječci. Kako bi dekodirali bilo koji isječak iz videa, moramo pronaći najbliži prethodni I-isječak te krenuti od njega. To je bitno kako bi u videu mogli ponovno reproducirati neki prethodni dio ili preskočiti dio videa.

Ideja za kodiranje P-isječaka je pronaći podudaranja, tj. grupe piksela koji imaju slične uzorke u prethodnom isječku i kodirati razliku između P-isječka i pronađenog podudaranja. Kako bi pronašli podudaranja MPEG algoritam dijeli P-isječak na blokove 16x16. Proces kodiranja ovakvih blokova ilustriran je na sljedećoj slici.



Slika 4.3.1: Kodiranje P-isječka ¹¹

Za svaki blok u P-isječku koder pronalazi referentne blokove u prethodnom P ili I-isječku koji se najviše podudaraju s njim. Referentni blok ne mora biti unutar granica bloka 16x16 tj. može se nalaziti bilo gdje unutar slike. U praksi, pomicanje x-y je jako malo. Ovakvo pomicanje naziva se vektor smjera. Kada je podudaranje pronađeno, pikseli referentnog bloka izuzimaju se iz trenutnog bloka. To nam daje rezidual koji je u idealnom slučaju približno nula u svim komponentama. Rezidual se kodira koristeći sheme slične JPEG-u, ali će dati bolji omjer kompresije zbog manjeg intenziteta. Kako bi koder poslao kodirani rezidual, mora proslijediti i vektor smjera. Taj vektor se kodira Huffmanovim algoritmom. Motivacija za traženje podudaranja izvan okvira je u traženju načina kako efikasno kodirati objekt u pokretu. Naime, ako se objekt pomiče kroz niz slika, tada se najbolje podudaranje neće uvijek nalaziti na istom mjestu u slici. Ukoliko nije moguće pronaći dobro podudaranje, tada se blok kodira kao I-isječak.

U praksi, potraga za dobrim podudaranjem je najzahtjevniji dio MPEG kodiranja. Primijetimo da, iako je traženje podudaranja "skupo", rekonstrukcija slike je vrlo "jeftina" s obzirom na to da će dekodeer primiti i vektor smjera i samo treba pogledati u prethodnu sliku.

B-isječak koristi se u situaciji kada se dio P-isječka ne pojavljuje u prethodnim isječcima, ali se može pojaviti u sljedećim isječcima. Zamislimo da se auto sa strane pojavljuje u sceni. Pretpostavimo da I-isječak kodira scenu prije pojavljivanja automobila, a sljedeći I-isječak se pojavljuje kada je automobil već u potpunosti vidljiv. Želimo koristiti P-isječke za uzastopne scene. Međutim, s obzirom na to da se automobil ne pojavljuje u prvom I-isječku, P-isječak neće moći dobiti potrebne informacije, a činjenica da se automobil pojavljuje u sljedećem I-isječku nam ne pomaže s obzirom na to da P-isječak promatra samo prethodne isječke.

B-isječak informacije koje može iskoristiti traži u oba smjera. U principu je tehnika jako slična onoj koja se koristi kod P-isječaka, ali razlika je u tome što umjesto traženja podudaranja u prethodnim I ili P-isječcima, podudaranja također traži i u sljedećim isječcima. Ukoliko je dobro podudaranje pronađeno u oba slučaja, uzima se prosjek dva referentna bloka. Ukoliko je pronađeno samo jedno dobro podudaranje, ono se koristi kao referentno. Koder treba proslijediti informaciju koji isječci su korišteni i pripadni/e vektor/e smjera.

Pitanje koje se nameće je koliko je efektivna MPEG kompresija. Možemo promatrati omjer kompresije za svaki tip isječka i izračunati težinski prosjek s obzirom na to koji se isječci češće pojavljuju.

Krenemo li sa 24-bitnom slikom 356x260, uobičajeni omjeri kompresije za MPEG su sljedeći:

Tip	Veličina	Omjer
I	18Kb	7:1
P	6Kb	20:1
B	2.5Kb	50:1
Prosjek	4.8	27:1

Ako jedan 356x260 isječak treba 4.8Kb, kolika je propusnost potrebna za MPEG kako bi vratio video od 30 isječaka u sekundi?

$$30 \text{ frames/sec} * 4.8 \text{ Kb/frame} * 8 \text{ b/bit} = 1.2 \text{ Mbits/sec}$$

Ukoliko video ima i zvuk, propusnost bi trebala biti otprilike 0.25Mbits/sec veća, što bi ukupno bilo 1.45Mbits/sec.

Literatura

- [1] G. E. Belloch, *Introduction to data compression*, Carneige Mellon University, 2013
- [2] M. Benšić, N. Šuvak, Uvod u vjerojatnost i statistiku, Odjel za matematiku, Osijek, 2013
- [3] D. A. Lelewer, D. S. Hirschenberg, *Data compression*
- [4] I. S. Pandžić, A. Bažant, Ž. Ilić, Z. Vrdoljak, M. Kos, V. Sinković (2009.), *Unod u teoriju informacija i kodiranje*, 2.izd, Element, Zagreb
- [5] K. Sayood, *Introduction to Data Compression*, University of Nebraska, Third edition
- [6] Data compression,
URL:<https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/dct.htm>
- [7] Jensens inequality,
URL:<https://mathworld.wolfram.com/JensensInequality.html>
- [8] Kvantizacija,
URL:<https://hr.wikipedia.org/wiki/Kvantizacija>
- [9] Self information,
URL:<https://psychology.wikia.org/wiki/Self-information>

Sažetak

Kompresija podataka koristi se za različite oblike podataka. O samim podacima ovisi i koju metodu i koji algoritam za kompresiju ćemo koristiti. Bolji omjer kompresije postiže se algoritmima s gubitkom podataka, no oni nisu uvijek prikladni za korištenje. Najčešće u praksi, pri kompresiji podataka koristi se nekoliko algoritama odjednom kako bi se postigla što bolja kompresija. Želimo li osigurati da originalni podaci mogu biti u potpunosti rekonstruirani nakon kompresije potrebno je koristiti algoritme za kompresiju bez gubitaka podataka, a ukoliko u podacima možemo izgubiti neke podatke, bez da izgubimo informacije koje ti podaci nose, bolji omjer kompresije postići ćemo korištenjem algoritama za kompresiju s gubitkom podataka.

Ključne riječi: Kompresija podataka, algoritmi za kompresiju podataka, kompresija podataka bez gubitaka, kompresija podataka s gubitcima, Huffmanov algoritam, RLE, LZW, JPEG, MPEG

Summary

Data compression is used for different data types. Method which will be used for data compression depends on which data type we want to compress. We can get better compression rate with lossy data compression, but this type of compression can't be used for all data types. In practice, we use more than one algorithm for data compression to achieve better compression rate. If we want to be sure that all of original data can always be restored from compressed data, we shall use lossless data compression, and otherwise, if we can lose some of the data in process, without losing information in data, we can use lossy data compression, and achieve better compression rate.

Key words: Data compression, Algorithms for data compression, Lossless data compression, Lossy data compression, Huffman algorithm, RLE, LZW, JPEG, MPEG

Životopis

Nikolina Farena rođena je 02.01.1992. u Slavonskom Brodu. Pohađala je Klasičnu gimnaziju fra. Marijana Lanosovića u rodnom gradu. Preddiplomski studij matematike na Odjelu za Matematiku u Osijeku upisala je 2013. godine. Nakon stečenog zvanja sveučilišnog prvostupnika matematike, 2017. upisuje diplomski studij matematike, smjer matematika i računarstvo, na istoimenom fakultetu.

Trenutno je zaposlena kao junior developer u Cadenas Services d.o.o u Slavonskom Brodu.