Usporedba odabranih algoritama za kompresiju teksta

Nikolaus, Filip

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij računarstva

Diplomski rad

USPOREDBA ODABRANIH ALGORITAMA ZA KOMPRESIJU TEKSTA

Rijeka, studeni 2022.

Filip Nikolaus

0069082375

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij računarstva

Diplomski rad

USPOREDBA ODABRANIH ALGORITAMA ZA KOMPRESIJU TEKSTA

Mentor: izv. prof. dr. sc. Jonatan Lerga

Rijeka, studeni 2022.

Filip Nikolaus

0069082375

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA DIPLOMSKE ISPITE

Rijeka, 21. ožujka 2022.

Zavod: **Zavod za računarstvo**
Predmet: **Teorija informacija i kodiranje**
Grana: **2.09.03 obradba informacija**

ZADATAK ZA DIPLOMSKI RAD

Pristupnik: **Filip Nikolaus (0069082375)**
Studij: Sveučilišni diplomski studij računarstva
Modul: Računalni sustavi

Zadatak: **Usporedba odabranih algoritama za kompresiju teksta/Comparison of Selected Text Compression Algorithms**

Opis zadatka:

Potrebno je usporediti odabrane algoritme (entropijske i temeljene na metodama rječnika) za kompresiju teksta. Kao predstavnike entropijskih algoritama potrebno je implementirati Huffmanov i Shannon-Fanov algoritam, a kao one temeljene na rječnicima LZW i LZSS algoritme. Dodatno je potrebno iste usporediti s DEFLATE i LZMA algoritmima. Performanse algoritama potrebno je usporediti u vidu omjera kompresije na odabranim tekstovima i vremenu izvršavanja u programskim jezicima C++ i Python.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Filip Nikolaus

Zadatak uručen pristupniku: 21. ožujka 2022.

Mentor:

Izv. prof. dr. sc. Jonatan Lerga

Predsjednik povjerenstva za diplomski ispit:

Prof. dr. sc. Kristijan Lenac

IZJAVA

Izjavljujem da sam samostalno izradio diplomski rad „Usporedba odabranih algoritama za kompresiju teksta“ iz kolegija „Teorija informacija i kodiranje“ uz vodstvo i stručnu pomoć mentora izv. prof. dr. sc. Jonatana Lerge.

_____

Filip Nikolaus

ZAHVALA

Zahvaljujem se mentoru izv. prof. dr. sc. Jonatanu Lergi na stručnoj pomoći i podršci tijekom cijelog procesa izrade i pisanja ovog diplomskog rada. Posebno se zahvaljujem svojim roditeljima i bratu na potpori i podršci tijekom cijelog studija.

SADRŽAJ

1. UVOD

Kompresija ili sažimanje podataka proces je kojim se modificiraju podaci s ciljem smanjenja veličine. Drugim riječima, kompresijom podataka smanjuje se potrebni fizički prostor za pohranu podataka. Kompresija se ostvaruje korištenjem određenih metoda i tehnika, odnosno algoritama, ovisno o vrsti podatka koji se žele kompresirati. Kompresirati se mogu razni podaci, od videozapisa i fotografija do glazbe i teksta. Kompresija podataka dijeli se na onu bez gubitaka (*engl.* lossless compression) i onu s gubicima dijelova podataka (*engl.* lossy compression).

Kod kompresije bez gubitaka, kao što sam naziv kaže, rezultat procesa su podaci smanjene veličine, ali bez ikakvih gubitaka. Logično, kod takve kompresije omjer kompresije[1] je manji nego kod one s gubicima. Takva se kompresija najčešće koristi za tekst, budući da bi gubici kod kompresije teksta mogli uništiti smisao ili informaciju koju tekst sadržava. Upravo su u ovom radu obrađeni algoritmi koji kompresiraju tekst bez ikakvih gubitaka koristeći razne tehnike i metode. Osim za tekst, kompresija bez gubitaka koristi se i za glazbu i fotografiju. O tome najbolje svjedoče popularni FLAC format za glazbu i PNG format za fotografiju [1].

S druge strane, kompresija s gubicima daje puno bolji omjer kompresije, odnosno podacima koji su kompresirani algoritmima koji koriste tehnike s gubicima potrebno je puno manje fizičkog prostoru za pohranu. Takvim algoritmima najčešće se kompresiraju videozapisi, fotografije i glazba. Kod navedenih medija gubitak nekoliko sličica u sekundi ili piksela neće značajno oštetiti informaciju koju taj medij sadrži, za razliku od teksta gdje je gubitak svakog simbola značajna šteta. Kompresija s gubicima ne označuje nužno da će podaci nakon kompresije biti manje kvalitete. Najbolji primjer za to je neka fotografija koja sadrži visoku razinu šuma. U tom slučaju, kada se šum ukloni, veličina fotografije će biti manja, a kvaliteta veća nego kod originalne. Primjeri kompresije s gubicima su JPEG format za fotografije te MP3 format za glazbu [1].

Općenito, danas primarni razlog korištenja kompresije podataka nije nedostatak fizičkog prostora za pohranu. U posljednje vrijeme, hardver za pohranu podataka dostupan je širokoj masi ljudi i, što je još bitnije, vrlo je jeftin. Razlog korištenja kompresije leži u prijenosu

[1] Odnos veličina podataka prije i nakon kompresije

podataka, odnosno ograničavajući faktor su brzine prijenosa podataka, bilo lokalno ili putem Interneta. Naravno, što su podaci manje veličine to će se brže prenijeti na željeno odredište.

Kao što je već spomenuto, u ovom je radu obrađeno i opisano šest odabranih algoritama za kompresiju bez gubitaka. Odabrani algoritmi testirani su na raznim ulaznim tekstovima prilikom čega su praćeni rezultati kompresije, točnije omjeri kompresije. Nadalje, odabrani algoritmi implementirani su u dva vrlo popularna programska jezika – C++-u i Pythonu, zbog direktne usporedbe u brzini izvršavanja.

Odabrane algoritme možemo svrstati u tri kategorije:

1. entropijski algoritmi za kompresiju,
2. algoritmi za kompresiju temeljeni metodama rječnika te
3. napredni algoritmi za kompresiju.

Dakle, Huffmanovo kodiranje i Shannon-Fanov algoritam predstavljaju prvu kategoriju, LZW i LZSS algoritmi predstavljaju drugu kategoriju, dok su predstavnici posljednje kategorije DEFLATE i LZMA algoritmi.

U sljedećem poglavlju opisane su korištene tehnologije i ulazni podaci za programsku implementaciju, odnosno testiranje navedenih algoritama. U trećem, četvrtom i petom poglavlju opisane su kategorije algoritama te pripadajući algoritmi, respektivno kako su prethodno navedeni. U šestom poglavlju opisane su programske implementacije navedenih algoritama, dok su u sedmom poglavlju tablično i grafički prikazani rezultati testiranja algoritama na ulaznim tekstovima. U posljednjem je poglavlju napisan zaključak.

2. OPIS KORIŠTENIH TEHNOLOGIJA I SETA PODATAKA

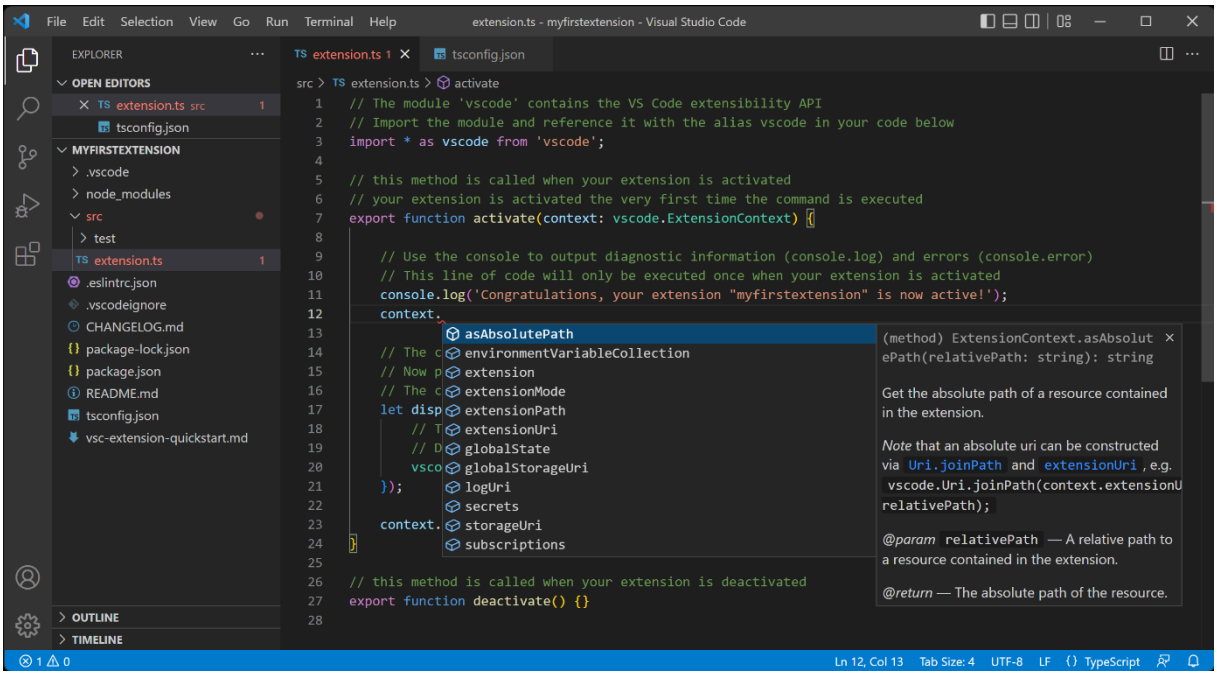
Za izradu praktičnog dijela ovog rada korištena su dva vrlo popularna programska jezika – C++ i Python. Prvi je poznatiji kao „brži“ jezik, odnosno programski jezik niže razine. S druge strane, potonji je više apstrahiran te je lakši za čitanje (razumijevanje), ali je zato, u pravilu, mnogo „sporiji“ [2]. U ovom radu se, osim glavnog zadatka u vidu usporedbe omjera kompresije, želi provjeriti je li C++ brži od Pythona kada se radi o algoritmima za kompresiju teksta.

Za pravilno izvođenje i kompajliranje programskih kodova algoritama potrebno je uspostaviti radna okruženja o kojima će biti riječ u nastavku.

2.1. Visual Studio Code (VS Code)

Visual Studio Code, koji se obično skraćeno naziva VS Code, uređivač je otvorenog koda koji je napravio Microsoft u suradnji s Electron Frameworkom za Windows, Linux i macOS. Njegove najbitnije značajke su: podrška za otklanjanje pogrešaka (*engl.* debugging*)*, isticanje sintakse, pametno dovršavanje koda, mogućnost jednostavnog refaktoriranja koda te ugrađeni Git (popularni alat za kontrolu verzija). Osim toga, korisnici mogu promijeniti temu korisničkog sučelja, stvarati prečace na tipkovnici za brži pristup željenim alatima, detaljno podešavati postavke te instalirati proširenja koja dodaju dodatne funkcije [3].

Tako je za potrebe ovog rada instalirano proširenje po imenu „*Microsoft C/C++ extension*“ koje opisanom uređivaču dodaje podršku za kompajliranje programskih kodova napisanih u C/++ programskim jezicima. Detaljnije o mogućnostima VS Codea i samog C++ biti će opisano postepeno kako se bude opisivala implementacija algoritama u šestom poglavlju. Korisničko sučelje VS Codea vidljivo je na slici 2.1.

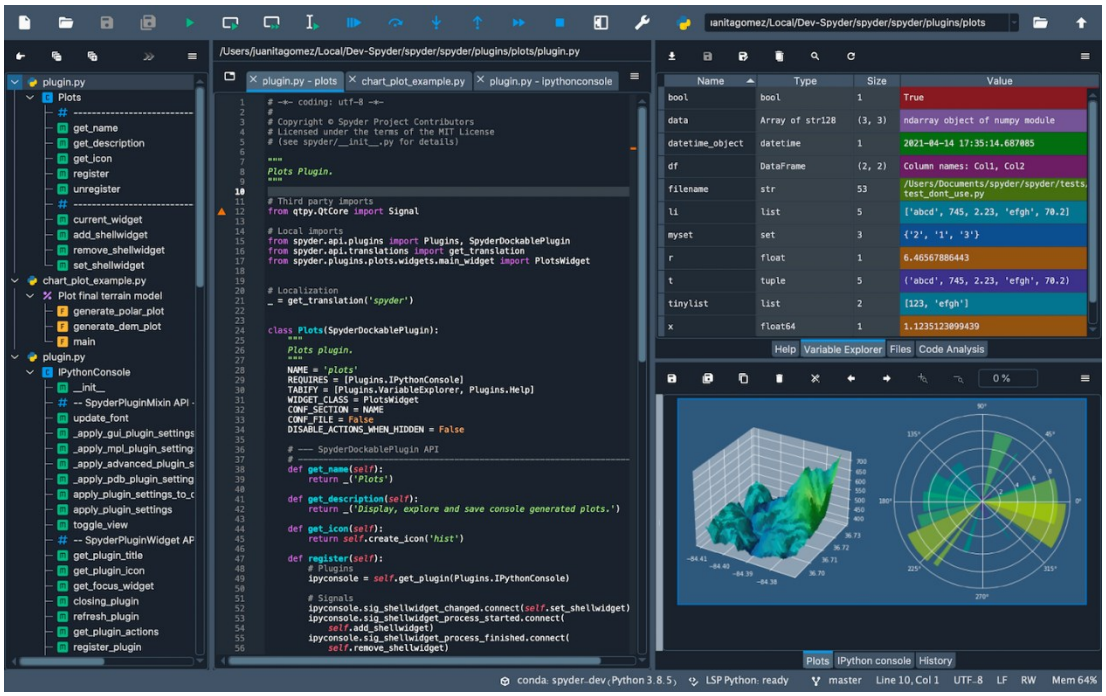


Slika 2.1: Korisničko sučelje VS Codea [3]

2.2. Spyder

Spyder dolazi kao dio znanstvenog okruženja Anaconda te predstavlja višeplatformsko integrirano razvojno okruženje (IDE) otvorenog koda za programiranje u programskom jeziku Python. Spyder je integriran s nizom istaknutih znanstvenih knjižnica uključujući *NumPy*, *SciPy*, *Matplotlib*, *pandas*, kao i mnoge druge knjižnice otvorenog koda. Kao i kod VS Codea, korisnik ima mnogo mogućnosti za prilagodbu korisničkog sučelja svojim željama i potrebama. Sam Python dolazi predinstaliran sa Spyderom te nije potrebna instalacija dodataka ili ekstenzija [4].

Za potrebe ovog rada korištena je verzija 5.1.5 Spyder razvojnog okruženja te 3.9 verzija programskog jezika Python. Detaljnije o mogućnostima Spydera i samog Pythona biti će opisano postepeno kako se bude opisivala implementacija algoritama u šestom poglavlju. Korisničko sučelje Spydera vidljivo je na slici 2.2.



Slika 2.2: Korisničko sučelje Spyder-a [4]

2.3. Korišteni set podataka za testiranje algoritama

Kako bi se odabrani algoritmi za kompresiju teksta pravilno istestirali, potrebno ih je primijeniti na odabranim tekstovima. Odabrano je pet tekstova od kojih su tri na engleskom jeziku (s napomenom da je jedan od njih sastavljen od slučajnih simbola) i dva na hrvatskom jeziku. Kao što je vidljivo iz detaljne specifikacije u tablici 2.1, ulazni tekstovi različitih su duljina kako bi se algoritmi testirali u različitim uvjetima.

Tablica 2.1: Detaljna specifikacija ulaznih tekstova

Ime teksta	Veličina teksta	Jezik teksta
Europski zeleni plan[5]	77330 bajtova	Hrvatski
Zakon o radu[6]	214598 bajtova	Hrvatski
Uvod u algoritme[7]	2388618 bajtova	Engleski
Sveta Biblija[8]	4653740 bajtova	Engleski
Slučajni tekst[9]	100000 bajtova	Engleski

3. ENTROPIJSKI ALGORITMI ZA KOMPRESIJU

Prije nego li se krene u detaljnu analizu odabranih entropijskih algoritama, definirat će se i objasniti osnovni pojmovi koji su potrebni za njihovo razumijevanje.

3.1.Osnovni pojmovi

3.1.1. Entropija

Diskretna slučajna varijabla je varijabla odabrana iz skupa od *n* mogućih ishoda, pri čemu je za svaki ishod poznata pripadajuća vjerojatnost. Entropija diskretne slučajne varijable definira se kao:

$$H(X) = -\sum_{i=1}^{n} p(x_i) * log_2 p(x_i) \left[\frac{bit}{simbol}\right]$$

gdje X predstavlja diskretnu slučajnu varijablu koja poprima vrijednosti iz skupa $\{x_1,..., x_i,...x_n\}$, a $p(x_i)$ predstavlja vjerojatnost ishoda x_i, $1 \leq i \leq n$. Poruka sastavljena od ovakvih simbola sadrži u prosjeku $H(X)$ bita po simbolu. Entropiju možemo shvatiti kao mjeru sa sadržaj informacije, odnosno za neodređenost izvora informacije [10].

Ako koristimo logaritam s bazom 2, entropiju izražavamo u bitovima. Općenito, definicija entropije vrijedi uz bilo koju odabranu bazu logaritma, samo što se tada ne izražava u bitovima nego u drukčijim jedinicama (npr. nit ili nat – baza *e*, dit – baza 10); u praksi se gotovo uvijek koristi baza 2 i bit.

Konkretno kod entropijskih algoritama, entropija ovisi o vjerojatnosti pojavljivanja određenog simbola u odabranom tekstu. Neki simboli pojavljuju se češće, dok se ostali pojavljuju rjeđe. Kao što se zaključuje iz primjera u nastavku, vjerojatnost pojavljivanja određuje i entropiju događaja. Svi entropijski algoritmi pretpostavljaju da se u odabranom tekstu ne pojavljuju svi simboli sa istom vjerojatnošću. Iz toga slijedi da su simboli koji se češće pojavljuju, nakon provedbe algoritama, kodirani kraćim kodovima, dok su simboli koji se rjeđe pojavljuju kodirani duljim kodovima [11].

Postoji još mnogo svojstava i zanimljivosti u vezi entropije, međutim u ovome radu neće se ići u teoretske detalje, već će se entropija objasniti na svakodnevnom primjeru.

3.1.1..1. Primjer: entropija bacanja novčića

Recimo da novčić baca u zrak i želi se izračunati entropija tog događaja. U tom slučaju, postoje dva moguća ishoda, odnosno $n = 2$. Oba ishoda imaju jednaku vjerojatnost - $p(x_i) = 0{,}5$. Uvrštavanjem te vjerojatnosti u prethodno navedenu formulu dobiva se da je entropija bacanja novčića jednaka jednom bitu po simbolu ($H(X) = 1$ bit/simbol). Drugim riječima, srednji sadržaj informacije poruke iznosi jedan bit po simbolu. U slučaju „nepravednog" novčića koji uvijek daje isti ishod, na primjer glavu, pripadajuće vjerojatnosti su $p(x_1) = 1$ i $p(x_2) = 0$. Tada entropija bacanja novčića iznosi nula bita po simbolu ($H(X) = 0$ bit/simbol). U slučaju „asimetričnog" novčića koji daje 70% pismo, entropija bacanja novčića iznosi 0,88 bita po simbolu ($H(X) = 0{,}88$). Dakle, srednji sadržaj informacije poruke nastale bacanjem takvog novčića daje u prosjeku 0,88 bita po simbolu.

Maksimalna entropija (1 bit po simbolu) postiže se kada su vjerojatnosti oba ishoda jednaka, odnosno kada je vjerojatnost glave jednaka vjerojatnosti pisma ($p = 0{,}5$). Drugim riječima, maksimalna entropija postiže se kada je najmanje sigurno koji će od mogućih ishoda biti ishod događaja. Za entropiju je svejedno koji se ishod pojavljuje s većom, odnosno manjom vjerojatnošću jer se zamjenom njihovih uloga situacija ne mijenja (s informacijskog gledišta) [10].

3.1.2. Prefiksni kodovi

Kada se neki simbol kompresira koristeći neki entropijski algoritam, rezultat kompresije je prefiksni kod (*engl.* prefix code). To znači da kod tog simbola ne može biti nastavak prethodnog koda, niti mu sljedeći kod ne može biti nastavak. Drugim riječima, svi kodovi su unikatni te je svaki simbol jednoznačno kodiran [11].

Upotreba prefiksnih kodova kod entropijskih algoritama uvelike pojednostavljuje proces kodiranja i dekodiranja jer, kao što je objašnjeno, ne postoje nizovi kodova, već je svaki kod jedinstven i može se dekodirati samostalno, odnosno neovisno o ostalim kodovima. Također, važno je naglasiti kako su prefiksni kodovi kod entropijskih algoritama ujedno i binarni kodovi. To znači da svaka znamenka prefiksnog koda zauzima samo jedan bit memorije.

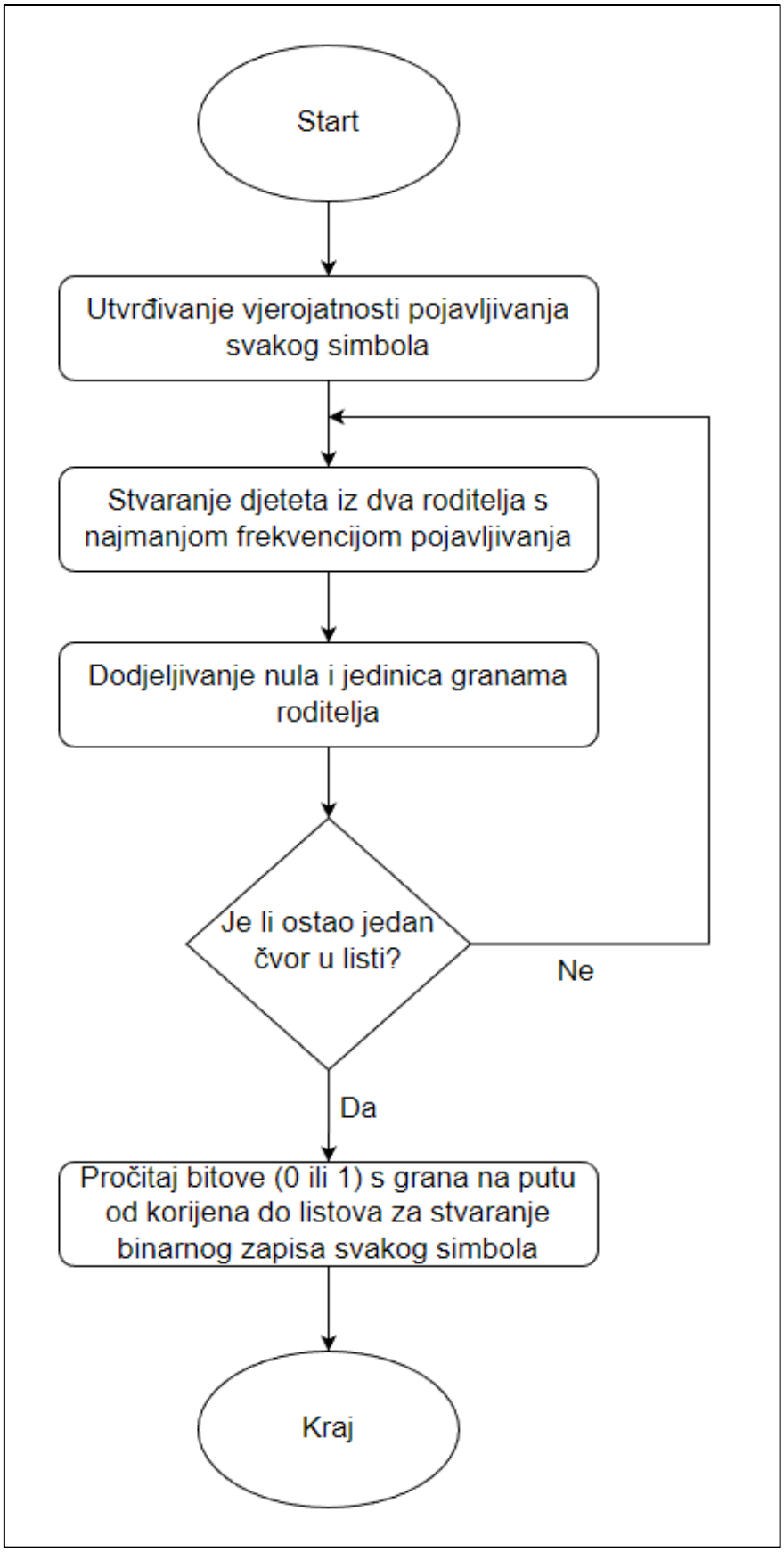
3.2. Huffmanovo kodiranje

Huffmanovo kodiranje algoritam je za kompresiju bez gubitaka koji je razvio David A. Huffman 1951. godine, dok je bio student na MIT-u, a koji je objavio u radu „*A Method for the Construction of Minimum-Redundancy Codes*“ 1952. godine. Navedeni algoritam bio je prava mala revolucija u svoje vrijeme jer je bio prvi koji je koristio obrnutu konstrukciju binarnog stabla. Drugim riječima, Huffmanovo kodiranje konstruira stablo od dolje prema gore, za razliku od ostalih algoritama koji koriste klasičnu konstrukciju od gore prema dolje. Takvo stablo naziva se i Huffmanovo stablo [12].

Navedeni je algoritam tipični primjer entropijskog algoritma jer kodira simbole u tekstu prefiks kodovima varijabilne duljine, ovisno o frekvenciji, odnosno vjerojatnosti pojavljivanja određenog simbola u tekstu. Shodno tome, simboli koji se češće pojavljuju kodirani su kraćim prefiksnim kodovima, dok su simboli koji se rjeđe pojavljuju kodirani duljim prefiksnim kodovima. Tako se postiže optimalnost, kojoj navedeni algoritam teži.

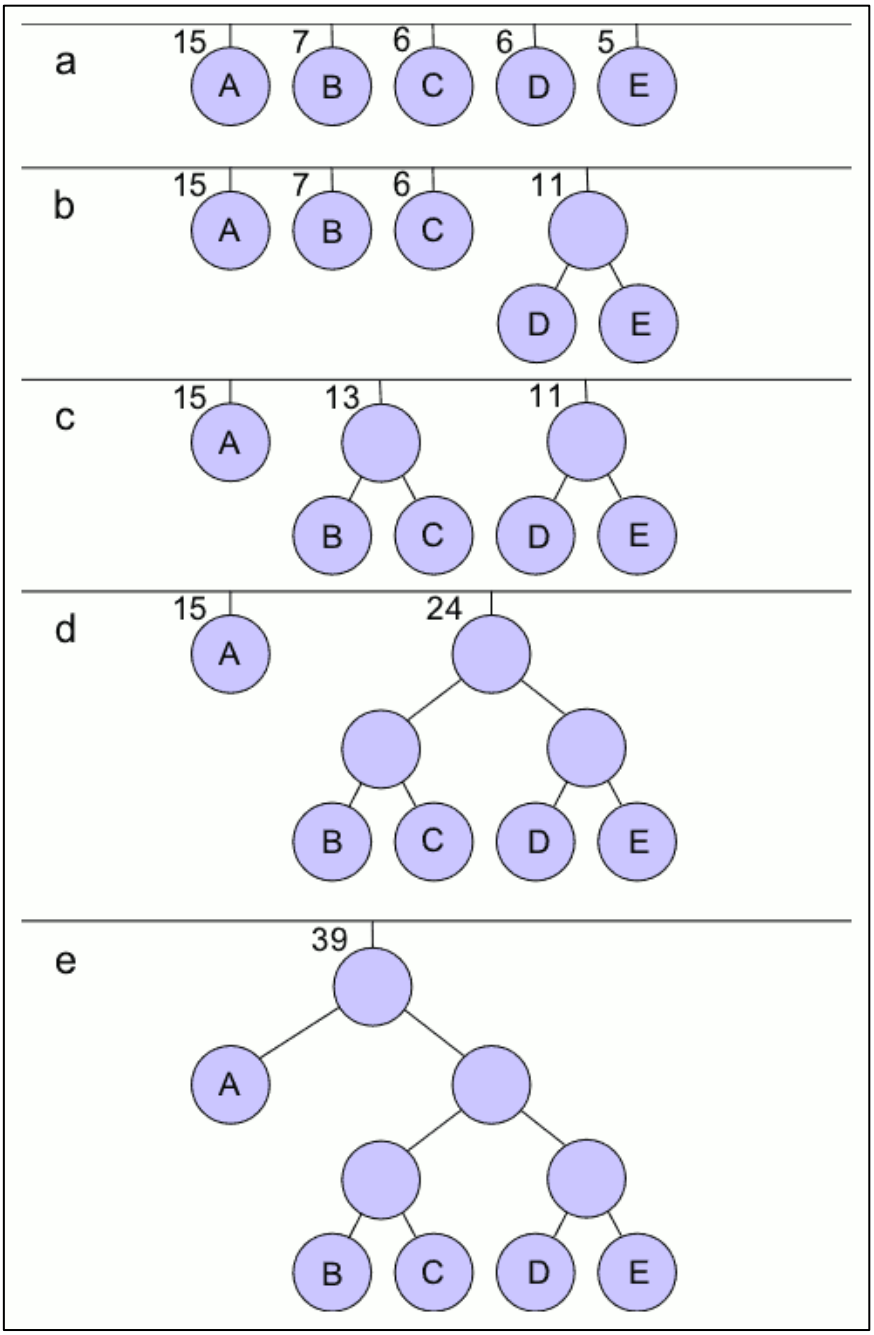
Također, navedeni algoritam temelji se na teoremu koji kaže da su u optimalnom kodu simboli s najmanjim vjerojatnostima pojavljivanja kodirani pripadajućim kodnim riječima jednake duljine [11]. Navedeno ima smisla jer ako su kodovi simbola s najmanjim vjerojatnostima pojavljivanja kraći od kodova bilo kojih drugih simbola – kod nije optimalan.

Kao što je vidljivo iz blok dijagrama na slici 3.1, Huffmanovo kodiranje može se svesti na pet koraka. Prvi korak je utvrditi vjerojatnost, odnosno frekvenciju pojavljivanja svakog simbola u tekstu. Nakon toga svaki simbol zajedno sa svojom frekvencijom pojavljivanja predstavlja slobodni čvor stabla. Drugi korak je odabir dva čvora sa najmanjom frekvencijom pojavljivanja koji se kombiniraju u binarno podstablo čiji je korijen novonastali simbol, odnosno zbroj vjerojatnosti odabranih čvorova. Nadalje, granama koje vode od roditelja prema djeci dodjeljuju se vrijednosti 0 ili 1. Nije bitno hoće li lijeva ili desna grana biti 0 ili 1, već je bitno da dodjela uvijek bude konzistentna. Drugim riječima, ako se odabere da će se desnoj strani dodjeljivati vrijednost 1, a lijevoj strani vrijednost 0, onda taj princip treba slijediti kod svih grana. Sljedeći korak je provjera je li ostao samo jedan slobodni čvor. Ako nije, algoritam se ponavlja od drugog koraka, a ako je, iz nastalog stabla čitaju se prefiks kodovi za svaki simbol. Čitanje prefiks koda za svaki simbol vrši se od korijena do čvora sa tim simbolom.



Slika 3.1: Blok dijagram Huffmanovog kodiranja

Na slici 3.2. može se vidjeti prethodno opisani algoritam, korak po korak. Rezultat algoritma je takozvano Huffmanovo stablo koje je vidljivo u posljednjem koraku.



Slika 3.2: Konstrukcija Huffmanovog stabla [12]

U tablici 3.1. može se vidjeti rezultat provedbe Huffmanovog kodiranja nad simbolima *A*, *B*, *C*, *D* i *E* sa pripadajućim frekvencijama *15*, *7*, *6*, *6* i *5*, na temelju slike 3.2. Rezultati zadovoljavaju pretpostavke optimalnog koda, odnosno simbol koji se najčešće pojavljuje ima najkraći kod, dok ostali simboli imaju jednako duge kodove.

Tablica 3.1: Huffmanovi kodovi na temelju slike 3.2

Simbol	**Frekvencija pojavljivanja**	**Huffmanov kod**
A	15	0
B	7	100
C	6	101
D	6	110
E	5	111

Stabla koja Huffmanovo kodiranje koristi moraju biti poznata koderu i dekoderu. Za dekodiranje koda dobivenog opisanim kodiranjem potrebno je poznavati pripadajuće Huffmanovo stablo kako bi se mogao rekonstruirati izvorni niz. Samo dekodiranje svodi se na zamjenu prefiks kodova sa pripadajućim simbolima.

Na koncu, Huffmanovo kodiranje danas ima široku promjenu, a najčešće se koristi u kombinaciji s ostalim algoritmima za kompresiju (teksta, slika, videozapisa…).

3.3. Shannon-Fanov algoritam

U području kompresije podataka, Shannon–Fanovo kodiranje, nazvano po Claudeu Shannonu i Robertu Fanou, naziv je za dvije različite, ali povezane metode za konstruiranje prefiks koda na temelju skupa simbola i njihovih vjerojatnosti.

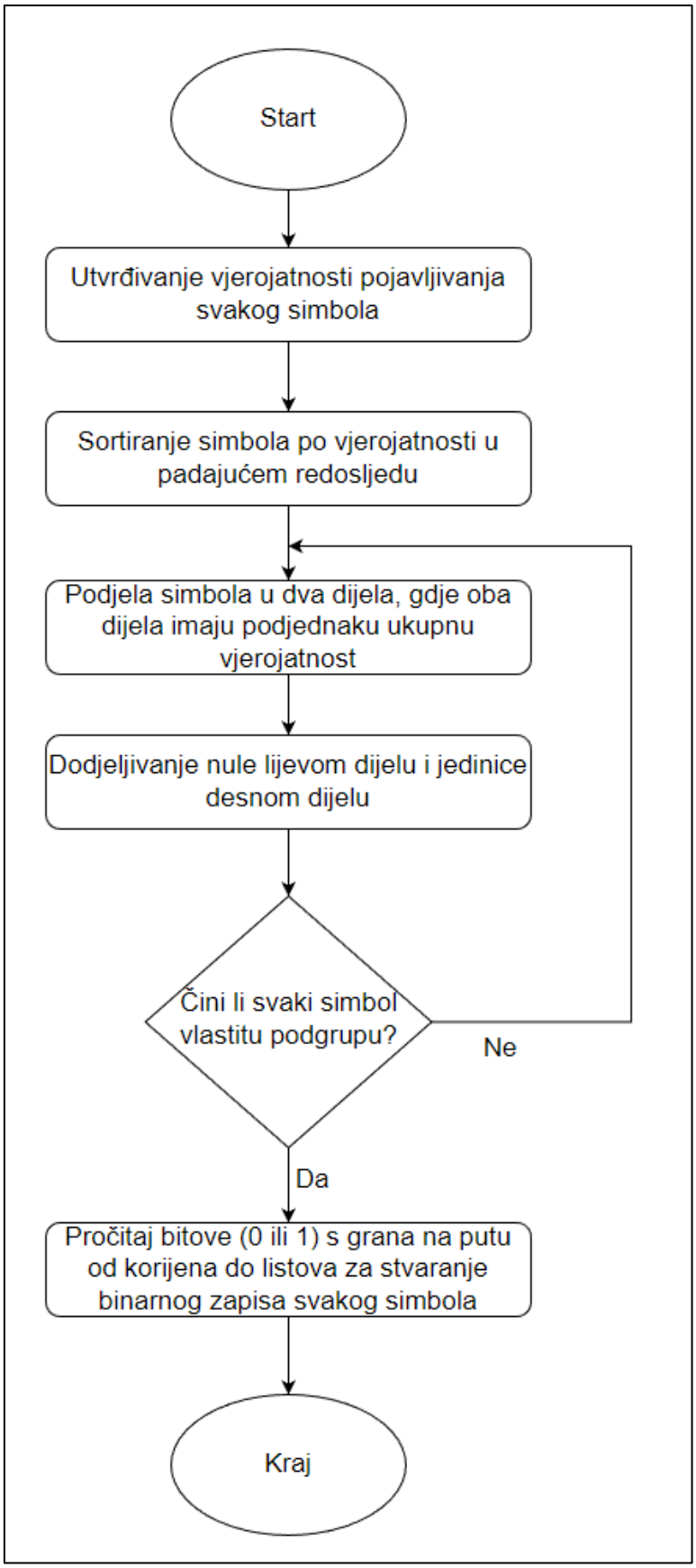
Shannonova metoda određuje duljinu prefiksnog koda određenog simbola, na način da se simbolu *i* dodjeljuje pripadajući kod duljine $l_i = [-log_2 p(i)]$. Ova je metoda predložena 1948. godine u Shannonovom članku koji nas upoznaje s teorijom informacija - „*A Mathematical Theory of Communications"* [13].

Fanova metoda dijeli simbole u dva skupa s vjerojatnostima što je moguće bliže 0,5. Zatim se ti skupovi opet dijele na dva dijela, i tako sve dok svaki skup ne sadrži samo jedan simbol. Kodna riječ za određeni simbol je niz nula i jedinica, ovisno o strani na kojoj se simbol nalazi u binarnom stablu [13].

Kombinacijom tih dviju metoda dobivamo Shannon-Fanov algoritam. On za razliku od Huffmanovog kodiranja koristi „normalno" binarno stablo, odnosno binarno se stablo konstruira od gore prema dolje.

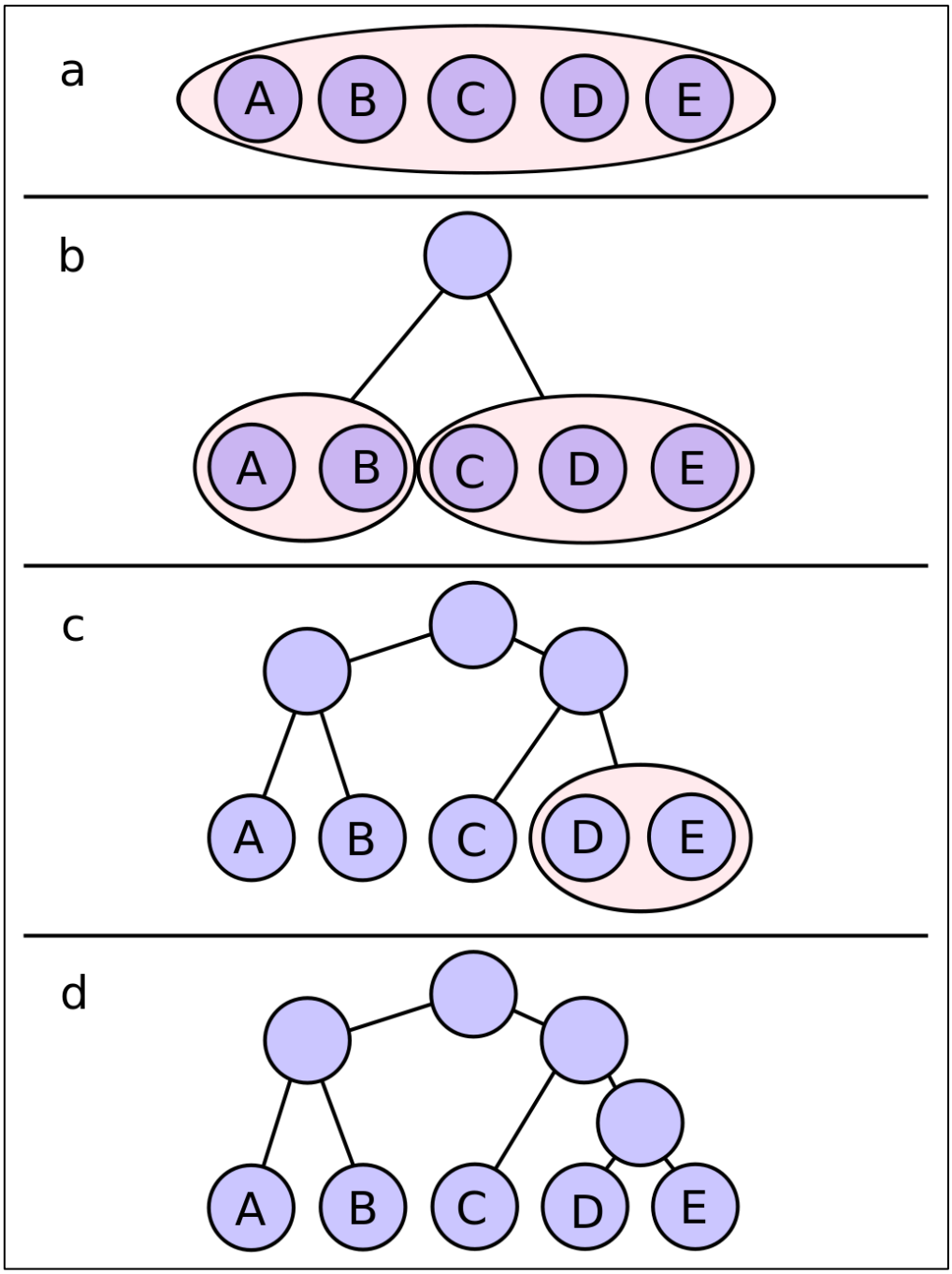
Iz prethodno opisanih teza može se zaključiti da ovaj algoritam, kao i Huffmanovo kodiranje, koristi prefiks kodove te, za razliku od Huffmanovog kodiranja, teži da se simboli 0 i 1 pojavljuju jednak broj puta u kodiranoj poruci [14]. Potonje stvara problem, odnosno onemogućuje da kodne riječi budu optimalne.

Kao što je vidljivo iz blok dijagrama na slici 3.3, Shanon-Fanov algoritam može se svesti na šest koraka. Prvi korak je utvrditi frekvenciju, odnosno vjerojatnost pojavljivanja svakog simbola u tekstu. Nakon toga svaki simbol zajedno sa svojom vjerojatnosti pojavljivanja predstavlja slobodni čvor stabla. Drugi korak je sortiranje simbola (čvorova) po vjerojatnosti pojavljivanja u padajućem redoslijedu. Sljedeći korak je podjela simbola u dva dijela, tako da im zbroj vjerojatnosti bude otprilike jednak. Nadalje, lijevom dijelu dodjeljuje se 0, dok se desnom dijelu dodjeljuje 1. Kao i kod Huffmanovog kodiranja, ovaj redoslijed može biti i obrnut, bitna je jedino konzistentnost. Pretposljednji korak je provjera čini li svaki simbol vlastitu podgrupu. Ako ne, algoritam se ponavlja od trećeg koraka, a ako da, iz nastalog stabla čitaju se prefiks kodovi za svaki simbol. Čitanje prefiks koda za svaki simbol vrši se od korijena do čvora sa tim simbolom.



Slika 3.3: Blok dijagram Shannon-Fanovog algoritma

Recimo da simboli *A*, *B*, *C*, *D* i *E* imaju pripadajuće vjerojatnosti *0.35*, *0.2*, *0.175*, *0.15* i *0.125*. Na slici 3.4. može se vidjeti prethodno opisani algoritam na zadanim simbolima sa pripadajućim vjerojatnostima, korak po korak. Rezultat algoritma je binarno stablo koje je vidljivo u posljednjem koraku.



Slika 3.4: Konstrukcija binarnog stabla koristeći Shannon-Fanov algoritam [13]

U tablici 3.2. može se vidjeti rezultat provedbe Shannon-Fanovog algoritma nad prethodno navedenim simbolima sa pripadajućim vjerojatnostima. Rezultati nisu optimalni, odnosno simbol koji se najčešće pojavljuje ima kod jednake duljine kao i simbol koji se pojavljuje duplo rjeđe (simbol A u odnosu na simbol C).

Tablica 3.2: Shannon-Fanov algoritam na primjeru

Simbol	Vjerojatnost	Prvi korak	Drugi korak	Treći korak	Kod
A	0.35	0	0		00
B	0.2	0	1		01
C	0.175	1	0		10
D	0.15	1	1	0	110
E	0.125	1	1	1	111

Shannon–Fanovi kodovi su suboptimalni u smislu da ne postižu uvijek najnižu moguću očekivanu duljinu kodne riječi, kao što to postiže Huffmanovo kodiranje. Međutim, Shannon–Fanovi kodovi imaju očekivanu duljinu kodne riječi unutar jednog bita od optimalne [15].

Iz tablice 3.3 vidljivo je da stvarne duljine kodova zadovoljavaju Shanonovu definiciju o duljini kodnih riječi. Jedino je simbol *C* na granici, dok ostali u potpunosti zadovoljavaju predviđenu duljinu.

Tablica 3.3: Provjera Shannonove definicije

Simbol	A	B	C	D	E
Vjerojatnost	0.35	0.2	0.175	0.15	0.125
$-log_2 p(i)$	1.51	2.32	2.51	2.74	3
Duljina koda	2	2	2	3	3

Kao i kod Huffmanovog kodiranja, stabla koja Shannon-Fanov algoritam koristi moraju biti poznata koderu i dekoderu. Za dekodiranje koda dobivenog opisanim kodiranjem potrebno je poznavati pripadajuće binarno stablo kako bi se mogao rekonstruirati izvorni niz. Također, samo dekodiranje svodi se na zamjenu prefiks kodova sa pripadajućim simbolima.

Danas se ovaj algoritam, u odnosu na Huffmanovo kodiranje, puno rjeđe koristi, a najveći razlog tome je što ne daje uvijek optimalan kod.

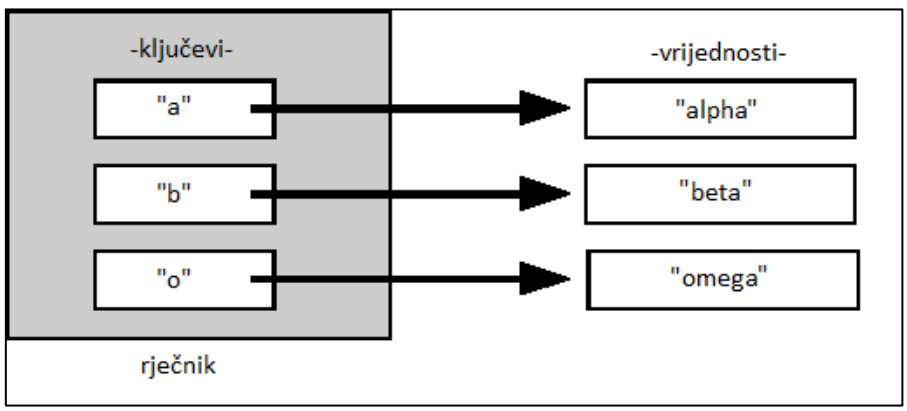
4. ALGORITMI ZA KOMPRESIJU TEMELJENI NA METODAMA RJEČNIKA

Metode rječnika, kao što i samo ime govori, koriste i održavaju rječnik za simbole ili nizove istih koji su se prethodno pojavljivali u određenom tekstu. Kompresija se postiže zamjenjivanjem nizova simbola u tekstu s referencama iz rječnika. Rječnik mora biti dinamične veličine, odnosno mora biti prilagodljiv u smislu da se proširuje dodavanjem novih simbola. S druge strane, mora dopuštati i brisanje simbola ili nizova istih koji se rijetko ili uopće ne koriste u slučaju da se premaši maksimalna dopuštena veličina rječnika [11].

Kod ovih algoritama pretpostavka je da vjerojatnost pojavljivanja nekog simbola ovisi o jednom ili više simbola koji prethode tom simbolu. Na primjer, ako imamo tri simbola: *a*, *u,* i *t*, vrlo je vjerojatno da će posljednji simbol biti *o* jer se tada formira riječ *auto*, koja može biti česta u našem tekstu. Nadalje, ovi algoritmi mogu pamtiti i cijele riječ ili čak i fraze ili dijelove teksta, ovisno o tome koliko se često ponavljaju.

Temelj metoda rječnika je kodiranje simbola ili nizova istih koji se često pojavljuju, što predstavlja kompresiju, ali i referencu na simbol ili nizove istih u rječniku. Za razliku od entropijskih algoritama, algoritmi temeljeni na metodama rječnika ne zahtijevaju poznavanje frekvencija, odnosno vjerojatnosti pojavljivanja simbola u tekstu. Drugim riječima, algoritmi jednostavno kodiraju određene skupine simbola koje se ponavljaju, bez potrebe za informacijom o vjerojatnosti tog pojavljivanja.

Na slici 4.1 vidi se primjer jednostavnog rječnika gdje su *ključevi* i *vrijednosti* parovi koji predstavljaju ključeve, odnosno pripadajuće vrijednosti u rječniku.



Slika 4.1: Primjer jednostavnog rječnika [16]

4.1. LZW

Lempel–Ziv–Welch (LZW) kodiranje predstavlja algoritam za kompresiju podataka bez gubitaka, temeljen na metodama rječnika, koji su stvorili Abraham Lempel, Jacob Ziv i Terry Welch. Objavio ga je Welch 1984. godine kao poboljšanu implementaciju LZ78 [17] algoritma koji su objavili Lempel i Ziv 1978. godine [18].

Algoritam je jednostavan za implementaciju i ima potencijal za vrlo visoku propusnost u hardverskim implementacijama. Najbolje performanse pokazuje u tekstovima koji imaju puno ponavljanja, odnosno redundantnih nizova. Nadalje, ovaj se algoritam koristi za kompresiju datoteka u Unix sustavima, točnije u *compress* alatu. Osim toga, LZW se koristi i kod izrade GIF formata za fotografije [18].

Algoritam prije samog izvođenja inicijalizira početni rječnik na sve standardne ASCII simbole. Točnije, prvih 256 mjesta rječnika zauzeto je prije samog unosa podataka koji se žele kompresirati. Obično se uzima 4096 kao maksimalna veličina do koje rječnik može narasti kako bi se čim bolje kompenzirao omjer između brzine izvođenja algoritma i performansi samog algoritma (omjera kompresije) [19]. U tom slučaju, svaka kodna riječ u rječniku duga je 12 bitova, što slijedi iz sljedeće jednakosti: $2^{12} = 4096$. Kao što se može zaključiti, kodne riječi dulje su od simbola kod unosa (1 simbol = 1 bajt = 8 bitova), ali to ima smisla jer se jednom kodnom riječju može kodirati više simbola. Kod jako redundantnih tekstova, jedna kodna riječ (12 bitova) može predstavljati niz simbola (cijelu riječ) na ulazu.

Za razliku od prethodnih algoritama, čiji je princip rada prikazan blok dijagramima, princip LZW kodiranja bit će objašnjen na temelju pseudokoda:

```
1.      Inicijalizacija početnog rječnika (256 mjesta)
2.      w = radni niz
3.      DOK ima simbola na ulazu RADI
4.           x = sljedeći simbol na ulazu
5.           AKO wc je u rječniku ONDA
6.             w = w + c
7.           INAČE
8.             IZLAZ: kod za w
9.             dodaj wc u rječnik
10.            w = x
11.     KRAJ PETLJE
12.     IZLAZ: kod za w
```

Slika 4.2: Pseudokod LZW kodiranja

Prvi korak algoritma je prethodno opisani postupak inicijalizacije rječnika i konfiguriranja veličine kodnih riječi. U sljedećem koraku algoritam uzima prvi simbol s ulaza koji postaje radni niz. Nakon toga uzima sljedeći simbol poruke i njime proširuje radni niz te provjerava postoji li ta nova riječ (radni niz + novi simbol) u rječniku. Ako postoji onda nova riječ postaje novi radni niz, a ako ne postoji onda na izlaz šalje kod za radni niz, novu riječ zapisuje u rječnik i dodjeljuje joj pripadajući kod te novi radni niz postaje sljedeći simbol na ulazu. Algoritam se ponavlja od trećeg do desetog koraka sve dok ima simbola na ulazu. Na samome kraju, na izlazu se ispiše posljednji radni niz.

Primjer LZW kodiranja niza „*ABABBAB*“ sa početnim rječnikom $A = 0$, $B = 1$ vidljiv je u tablici 4.1, korak po korak. Rezultat kompresije je kodni niz: *0*, *1*, *2*, *3*, i *1*. Za potrebe primjera nije korišten prethodno opisani početni rječnik veličine 256, već samo veličine 2. Unatoč tome, krajnji rezultat je isti, osim što bi novi nizovi koji se dodaju u rječnik dobivali pozicije od 257 pa nadalje te bi time rječnik bio veći, odnosno zauzimao bi više prostora.

Tablica 4.1: LZW kodiranje na primjeru

A	**B**	**A**	**B**	**B**	**A**	**B**	**w**	**x**	**Kod**	**Rječnik (indeks)**
●								A		
	●						A	B	0	AB (2)
		●					B	A	1	BA (3)
			●				A	B		
				●			AB	B	2	ABB (4)
					●		B	A		
						●	BA	B	3	BAB (5)
							B		1	

Budući da kod ovog algoritma dekompresija, odnosno dekodiranje nije jednostavno kao kod entropijskih algoritama (zamjena prefiks kodova sa pripadajućim simbolima), ono će također biti objašnjeno.

Prva kodna riječ se dekodira tako da se pročita pripadajući simbol iz početnog rječnika. Nakon toga slijedi ključan korak koji se ponavlja sve dok ima simbola na ulazu; kodna riječ se dekodira na način da se pročita pripadajući simbol iz rječnika te se u rječnik, kao novi zapis, zapisuju svi simboli iz prethodnog koraka zajedno sa prvim simbolom iz trenutnog koraka.

Recimo da se želi dekodirati poruka *0*, *0*, *1*, *2, 1*, *4* koristeći LZW sa početnikom rječnikom gdje je $A = 0$ i $B = 1$. U tablici 4.2 vidi se LZW dekodiranje navedenog primjera, korak po korak, po prethodno opisanom algoritmu. Rezultat dekompresije je niz simbola: *AABAABBA*.

Tablica 4.2: LZW dekodiranje na primjeru

Kod	Simbol	Rječnik (indeks)
0	A	
0	A	AA (2)
1	B	AB (3)
2	AA	BA (4)
1	B	AAB (5)
4	BA	BB (6)

4.2. LZSS

Lempel–Ziv–Storer–Szymanski (LZSS) algoritam je za kompresiju podataka bez gubitaka, koji su stvorili James A. Storer i Thomas Szymanski. Algoritam predstavlja unaprijeđenu verziju LZ77 [20] algoritma te je prvi puta opisan u članku „*Data compression via textual substitution*“, objavljenom u listu „*Journal of the ACM“* 1982. godine [21].

Ovaj algoritam se, kao i LZW, konceptualno temelji na metodama rječnika, ali za razliku od njega ne kodira simbole ni nizove istih već ih zamjenjuje sa referencom na pripadajuće mjesto u rječniku.

Glavna razlika između spomenutog LZ77 algoritma i LZSS-a je ta što kod LZ77 referenca na mjesto u rječniku može biti dulja od niza kojeg zamjenjuje. Kod LZSS-a, takve reference su izostavljene ako je duljina teksta kojeg treba zamjenjivati manja od „točke rentabilnosti“.

Drugim riječima, LZ77 svaki simbol, odnosno nizove istih koji se ponavljaju zamjenjuje sa referencom, dok LZSS zamjenjuje niz s referencom samo ako je niz dulji od prethodno postavljenog minimuma, a u protivnom zapisuje niz bez promjena [21].

Druga velika razlika je struktura spomenutih referenci. LZ77 koristi reference u obliku <*pomak, duljina, kodna riječ*>, dok LZSS ne koristi komponentu kodne riječi, nego koristi referencu u obliku <*pomak, duljina*>, ostavljajući samo pomak i duljinu. Spomenuta referenca znači: „Pomakni se u natrag za *pomak* simbola i kopiraj *duljina* simbola“.

Nadalje, LZSS algoritam kao novost uvodi jednobitnu zastavicu koja predstavlja prefiks svakom elementu u kodu, kako bi označila je li sljedeći element simbol ili referenca [21].

Kako ne bi sve ostalo na teoriji, LZSS algoritam biti će objašnjen na primjeru. Recimo da se želi kodirati LZSS algoritmom sljedeći niz:

„JA SAM FILIP. JA SAM FILIP. FILIP JA SAM.“

Algoritam čita simbol po simbol. Na prvom simbolu, „J“, algoritam provjerava međuspremnik pretraživanja (*engl.* search buffer) da vidi je li već vidio simbol „J“. Međuspremnik pretraživanja predstavlja rječnik algoritma, odnosno svaki simbol koji prođe kroz algoritam, isti ga dodaje u međuspremnik pretraživanja kako bi ga "zapamtio". Budući da još nije vidio simbol „J“ zapisuje simbol „J“ bez promjene, dodaje ga u međuspremnik pretraživanja te prelazi na sljedeći simbol – „A“. Algoritam opet provjerava međuspremnik pretraživanja da vidi je li prije vidio simbol „A“, a budući da nije, zapisuje ga i dodaje u međuspremnik pretraživanja te prelazi na sljedeći simbol.

Postupak je isti za sve simbole dok se ne dođe do drugog simbol „A“ („JA SA…“). Budući da u međuspremniku pretraživanja već postoji simbol „A“ algoritam je spreman zapisati referencu umjesto simbola „A“, ali prije toga želi maksimizirati broj simbola koje označuje. Zbog toga čita sljedeći simbol, konkretno „M“, i provjerava nalazi li se taj simbol neposredno nakon simbola „A“ u međuspremniku pretraživanja. Budući da ne, uzima samo simbol „A“ i provjerava je li dulji od reference „*<3,1>*“[2]. Opet, budući da nije, zapisuje simbol „A“ bez promjene. Opisani se algoritam izvršava sve dok ima simbola na ulazu.

[2] Pomakni se u natrag za tri simbola i kopiraj jedan simbol

Rezultantni niz nakon izvršavanja izgleda ovako:

„JA SAM FILIP. <*14,14*>FILIP JA SAM.“

Kao što se vidi, zadani niz duljine četrnaest simbola zamijenjen je referencom „*<14,14>*“ duljine sedam simbola.

Ako se na algoritam gleda s visoke razine, može se pojednostavljeno opisati u pet koraka:

1. Učitavaj simbol po simbol
2. Provjeri jesi li vidio simbol prije
    a. Ako jesi, učitaj sljedeći simbol i pripremi referencu za izlaz
        i. Ako je referenca dulja od simbola/niza kojeg označuje, nemoj je zapisati
        ii. Dodaj simbol/niz u međuspremnik pretraživanja
    b. Ako nisi, dodaj simbol/niz u međuspremnik pretraživanja

Veličina međuspremnika pretraživanja može se mijenjati te predstavlja varijablu o kojoj ovisi brzina izvršavanja algoritma i omjer kompresije. Čim je međuspremnik pretraživanja veći, to je algoritam sporiji, ali bi u tom slučaju omjer kompresije trebao biti veći jer algoritam ima više simbola u „memoriji“. Također, vrijedi i obrnuto da je algoritam brži što je međuspremnik pretraživanja manji, ali tada je i omjer kompresije manji. Uobičajena veličina koja se koristi za međuspremnik pretraživanja je 4096 simbola.

Dekodiranje je, kao što se može pretpostaviti, vrlo jednostavno. Čita se redom kodirani, odnosno kompresirani niz te se reference zamjenjuju sa simbolima ili nizovima istih po opisanom principu „Pomakni se u natrag za *pomak* simbola i kopiraj *duljina* simbola“.

5. NAPREDNI ALGORITMI

U ovom poglavlju bit će opisani odabrani napredni, odnosno mješoviti algoritmi. Općenito, ne postoji takva kategorija algoritama za kompresiju, ali u ovome radu taj se naziv koristi za odabrane algoritme koji kombiniraju dva ili više algoritma koji pripadaju u prethodno opisane kategorije algoritama (entropijski i oni temeljeni na metodama rječnika).

Odabrani algoritmi vrlo su složeni i koriste tehnike niske razine programiranja za razliku od algoritama iz prethodno opisanih kategorija te zbog toga služe kao referentne točke za usporedbu s prethodno opisanim algoritmima.

5.1. DEFLATE

U računarstvu, DEFLATE je algoritam za kompresiju podataka bez gubitaka koji koristi kombinaciju LZ77 i Huffmanovog kodiranja. Dizajnirao ga je Phil Katz za verziju drugu svog alata za arhiviranje PKZIP. DEFLATE je kasnije detaljno specificiran u dokumentu „*RFC 1951*“ 1996. godine [22].

Ovaj je algoritam bio patentiran kao „*U.S. Patent 5,051,745*“ na Katzovu tvrtku PKWARE [23]. Međutim, kao što je navedeno u RFC dokumentu, algoritam se mogao implementirati na razne načine koji nisu pokriveni patentom. To je dovelo do njegove široke upotrebe – na primjer, u *gzip* komprimiranim datotekama i PNG slikovnim datotekama. Patent je u međuvremenu istekao te više nema nikakvih restrikcija za upotrebu algoritma.

Algoritam pronalazi duplicirane nizove simbola u ulaznima podacima. Drugo pojavljivanje niza zamjenjuje sa pokazivačem na prethodni niz u obliku, već spomenutog, para <*pomak, duljina*>. Maksimalni pomak ograničen je na 32768 bajtova, a duljina je ograničena na 258 bajtova. Kada se niz ne pojavljuje nigdje u prethodnih 32768 bajtova, zapisuje se kao niz bez promjena. Simboli ili duljine podudaranja kompresiraju se u jedan Huffmanov kod, dok se pomak do podudaranja kompresira u drugi Huffmanov kod [24].

Kompresirani niz podataka (*engl.* stream) sastoji se od niza blokova koji odgovaraju uzastopnim blokovima ulaznih podataka. Veličine blokova su proizvoljne, osim što su blokovi koji se ne mogu kompresirati ograničeni na 65535 bajtova.

Svaki se blok sastoji od tri dijela:

Prvi dio bloka predstavlja definicija Huffmanovih kodova za taj blok. Navedena definicija sastoji se od duljine koda svakog simbola, pri čemu je ta definicija kompresirana Huffmanovim i RLE kodiranjem. Sljedeći dio predstavljaju parovi koje čine prethodno opisani Huffmanovi kodovi za simbol ili duljinu podudaranja te Huffmanovi kodovi za pomak do podudaranja. Na koncu, posljednji dio bloka predstavlja *end-of-block* zastavica koja označava je li to posljednji blok u nizu (1) ili postoji još blokova za obradu (0) [24].

O odabranom načinu traženja podudaranja ovisi učinkovitost algoritma, kao i brzina njegova izvršavanja. U knjižnici koja će se u ovome radu koristiti za implementaciju – *zlib* [25], duplicirani se nizovi pronalaze pomoću *hash* tablice. Svi ulazni nizovi duljine 3 bajta umetnuti su u *hash* tablicu. Za sljedeća 3 bajta izračunava se *hash* indeks. Ako *hash* lanac za izračunati indeks nije prazan, svi nizovi u lancu uspoređuju se s trenutnim ulaznim nizom i odabire se najdulje podudaranje. Cjelovitost *hash* tablice određuje se razinom kompresije koju odabere korisnik (1-9).

Neki drugi alati koriste potpunije, ali mnogo sporije pristupe. Jedan primjer takvog pristupa su stabla sufiksa za pronalaženje prethodnih podudaranja.

Prethodno je opisan rad DEFALTE algoritma s poprilično visoke razine, ali treba naglasiti da spomenuti algoritam koristi još mnogo optimizacijskih trikova koji ga ubrzavaju i povećavaju mu performanse. Jedan od njih je i „mehanizam lijenog ocjenjivanja“ [24]. Točnije, ako algoritam procijeni da je pronašao dovoljno dug niz koji se podudara, on prestaje tražiti dalje i koristi pronađeni niz kao referencu te se na taj način ubrzava iako možda negdje postoji i dulji niz koji bi povećao kompresiju. Duljina niza koja će zadovoljiti algoritam ovisi o odabranoj razini kompresije, odnosno o tome je li korisniku bitnija brzina izvršavanja algoritma ili omjer kompresije.

5.2. LZMA

Lempel–Ziv–Markov algoritam (LZMA) je algoritam za kompresiju podataka bez gubitaka koji koristi kombinaciju LZ77 algoritma i kodiranja raspona (*engl.* range encoding). Razvijao ga je Igor Pavlov od 1996. do 1998. godine i prvi put korišten je u *7-Zip* alatu za arhiviranje gdje se i danas koristi [26].

Kodiranje raspona jako je slično aritmetičkom kodiranju. Jedina razlika je ta što se kodiranje vrši u brojevnom sustavu s bilo kojom bazom (npr. $[0, 8^5)$) umjesto u rasponu [0, 1), pa je brže kada se koriste veće baze (npr. bajt) uz malu kompenzaciju u učinkovitosti kompresije [27].

Navedeno kodiranje pripada entropijskim algoritmima te će biti objašnjeno na primjeru. Recimo da se želi kompresirati poruka „*AABAC*“. Početna distribucija vjerojatnosti iznosi: {*A*: *0.6*, *B: 0.2*, *C: 0.2}*. Budući da se želi kompresirati točno 5 simbola u, na primjer, brojevnom sustavu s bazom 10, početni interval je $[0, 10^5)$ => [0, 100000). Kada se na početni interval primijene pripadajuće vjerojatnosti, dobiju se sljedeći podintervali:

A: [0, 60000)
B: [60000, 8000)
C: [80000, 100000)

Budući da je sljedeći simbol ponovno „*A*“, uzima se prvi podinterval te se na njega primjenjuju sljedeće formule:

- donja granica novog intervala jednaka je $l' = l + ((h - l) * l_i / brojevniSustav^{brojSimbola})$ gdje je l donja granica trenutnog intervala, h gornja granica trenutnog intervala te l_i donja granica intervala u kojem se nalazi sljedeći simbol
- gornja granica novog intervala jednaka je $h' = l + ((h - l) * h_i / brojevniSustav^{brojSimbola})$ gdje je l donja granica trenutnog intervala, h gornja granica trenutnog intervala te h_i gornja granica intervala u kojem se nalazi sljedeći simbol

Nakon primjene formula $l' = 0 + ((60000 - 0) * 0 / 1^5)$ i $h' = 0 + ((60000 - 0) * 60000 / 10^5)$ interval izgleda ovako:

AA: [0, 36000)

Kao što se može vidjeti, interval je uvelike skraćen. Sljedeći simbol je „*B*“ te se ponovno primjenjuju napisane formule, nakon čega novi interval izgleda ovako:

AAB: [21600, 28800)

Kao što se može vidjeti, nakon ovog koraka interval je još više skraćen. Sljedeći simbol je „*A*“ te se ponovno primjenjuju napisane formule, nakon čega novi interval izgleda ovako:

AABA: [21600, 25920)

Nakon posljednje iteracije dobije se konačni interval koji izgleda ovako:

ABBAC: [25056, 25920)

Kodiranu poruku može predstavljati bilo koji broj iz navedenog intervala, ali najčešća praksa je da se uzme donja granica intervala. Konkretno, poruka ABBAC kodira se brojem 25056.

Kodiranje raspona kod LZMA algoritma radi isključivo na jednoj, binarnoj abecedi (0 i 1), tako da je interval uvijek podijeljen na dva dijela. Vjerojatnosti nisu unaprijed poznate, tako da je u tom pogledu LZMA čisto adaptivna metoda kompresije: koder i dekoder prilagođavaju vjerojatnosti kako koder šalje simbole, a dekoder ih prima. Odnosno, nakon što se svaki bit kodira i pošalje te primi i dekodira, cijeli skup vjerojatnosti je u potpuno istom stanju kod kodera i dekodera [28].

Algoritam pronalazi podudaranja pomoću pouzdanih rječničkih struktura podataka te stvara nizove simbola ili referenci (ovisno je li pronađeno podudaranje) koje koder raspona kodira bit po bit. Spomenute pouzdane rječničke strukture predstavljaju *hash* lanci, odnosno preciznije Markovljevi lanci.

LZMA se može gledati kao varijanta LZ77 algoritma s ogromnim rječnikom (čak do 4 GB) i posebnom podrškom za često korištene udaljenosti do podudaranja. Izlaz algoritma kompresira se opisanim koderom raspona, koristeći složeni model za predviđanje vjerojatnosti svakog bita. Znači, kompresirani niz (*engl.* stream) je niz bitova koji su podijeljeni u pakete gdje svaki paket opisuje ili jedan bajt ili LZ77 referencu čiji su duljina i pomak implicitno ili eksplicitno kodirana [29].

Kako DEFLATE, tako i LZMA koristi mnoge napredne optimizacijske tehnike. Osim spomenutih Markovljevih lanaca, treba spomenuti da je kod LZMA svaki dio svakog paketa modeliran neovisnim kontekstima, tako da su predviđanja vjerojatnosti za svaki bit u korelaciji s vrijednostima tog bita (i srodnih bitova iz istog polja) u prethodnim paketima iste vrste [29].

6. PROGRAMSKE IMPLEMENTACIJE OPISANIH ALGORITAMA

U ovom je poglavlju prikazana i objašnjena programska implementacija prethodno opisanih algoritama. Budući da su svi algoritmi paralelno implementirani u C++-u i Pythonu, zbog smanjenja redundancije, opisana je samo jedna implementacija svakog algoritma. Drugim riječima, implementacija u C++-u identična je onoj u Pythonu, naravno uz sintaktičke i minimalne konceptualne promjene. To je napravljeno kako bi se, osim omjera kompresije, mogla uspoređivati i brzina izvođenja algoritma.

Nadalje, svi su algoritmi implementirani na podjednakoj razini optimiziranosti, odnosno nisu korištene nikakve dodatne optimizacijske funkcije kako bi se pojedini algoritmi namjerno ubrzali.

Budući da ne postoji standardizirani izraz za prikaz omjera kompresije, u ovom je radu korišten sljedeći:

*omjerKompresije = (1 – (velicinaNakonKompresije / velicinaPrijeKompresije)) * 100 [%]*

Znači, ako je izračunati omjer kompresije jednak 60%, kompresirani tekst je veličine 40% od originalnog. Na primjer, originalni tekst veličine 100 bajtova kompresiran je u omjeru 70%. To znači da je kompresirani tekst veličine 30 bajta.

6.1. Huffmanovo kodiranje

Programska implementacija Huffmanovog kodiranja bit će opisana u C++-u. Prvi korak je kreiranje strukture koja predstavlja čvor *MinHeap* stabla, kao što je vidljivo iz slike 6.1. *MinHeap* stablo je stablo u kojem se najmanja vrijednost nalazi u korijenu te ono, u ovom slučaju, predstavlja Huffmanovo stablo.

```
1. struct MinHeapNode
2. {
3.     char data;
4.     int freq;
5.     MinHeapNode *left, *right;
6.
7.     MinHeapNode(char data, int freq)
8.     {
9.         left = right = NULL;
10.        this->data = data;
11.        this->freq = freq;
12.    }
13. };
```

Slika 6.1: Huffmanovo kodiranje - MinHeapNode struktura

MinHeapNode struktura sadrži dvije varijable, *data* i *freq*, koje predstavljaju simbol, odnosno frekvenciju pojavljivanja određenog simbola u tekstu. Nadalje, definirane su dvije instance *MinHeapNode* strukture koje predstavljaju lijevo, odnosno desno podstablo. Koristeći istoimeni konstruktor, određene vrijednosti varijabla *data* i *freq* dodjeljuju se svakoj instanci čvora.

Nakon opisane strukture, definira se još jedna struktura – *Compare*. Kao što naziv kaže, ta pomoćna struktura služi za usporedbu frekvencija simbola, odnosno čvorova. Ta usporedba biti će jako korisna kod kreiranja prioritetnog reda u kojem će biti zapisani svi čvorovi poredani uzlazno po frekvencijama. Programski kod opisane strukture vidljiv je na slici 6.2.

```
1. struct Compare
2. {
3.     bool operator()(MinHeapNode* l, MinHeapNode* r)
4.     {
5.         return (l->freq > r->freq);
6.     }
7. };
```

Slika 6.2: Huffmanovo kodiranje - Compare struktura

Nakon opisanih struktura, slijedi kratka pomoćna funkcija za izračun frekvencije pojavljivanja svakog simbola u ulaznom tekstu. Definicija funkcije vrlo je jednostavna i vidljiva je na slici 6.3.

```
1. void calcFreq(string str, int n)
2. {
3.     for (int i=0; i<str.size(); i++)
4.         freq[str[i]]++;
5. }
```

Slika 6.3: Huffmanovo kodiranje - calcFreq funkcija

Sljedeća rekurzivna funkcija vrlo je važna i služi za zapisivanje izračunatih kodova u varijablu *codes* koja je definirana kao rječnik (*map*) gdje je prva vrijednost simbol zapisan u obliku *char*, a druga vrijednost pripadajući kod zapisan u obliku *string*. Rekurzija je vrlo jednostavna, to jest čvorovima koji se nalaze u lijevom podstablu dodjeljuje se vrijednost 0, a čvorovima koji se nalaze u desnom podstablu dodjeljuje se vrijednost 1. Rekurzija se izvršava sve dok se ne obiđu svi čvorovi, odnosno simboli. Definicija funkcije i rekurzije vidljiva je na slici 6.4.

```
1. void storeCodes(struct MinHeapNode* root, string str)
2. {
3.     if (root==NULL)
4.         return;
5.     if (root->data != '$')
6.         codes[root->data]=str;
7.     storeCodes(root->left, str + "0");
8.     storeCodes(root->right, str + "1");
9. }
```

Slika 6.4: Huffmanovo kodiranje - storeCodes funcija

Nakon toga, slijedi definicija prioritetnog reda koji sadržava, kao što je spomenuto, sve čvorove poredane uzlazno po frekvencijama pojavljivanja. To se izvodi koristeći prethodno definirane *MinHeapNode* i *Compare* strukture, kao što je vidljivo na slici 6.5.

```
1. priority_queue<MinHeapNode*, vector<MinHeapNode*>, Compare> minHeap;
```

Slika 6.5: Huffmanovo kodiranje - minHeap prioritetni red

Sljedeća funkcija također je vrlo jednostavna i služi za izračunavanje omjera kompresije. Definirana je na način vidljiv na slici 6.6.

```
1. void totalGain(string str, string encodedString) {
2.     float beforeCompresion = str.size();
3.     cout << "\nVelicina teksta prije kompresije (u bajtovima): " << beforeCompresion;
4.     float afterCompresion = round(encodedString.size() / 8.0);
5.     cout << "Velicina teksta nakon kompresije (u bajtovima): " << afterCompresion;
6.     float compressRatio = (1 - (afterCompresion / beforeCompresion)) * 100 ;
7.     compressRatio = std::ceil(compressRatio * 100.0) / 100.0;
8.
9.     cout << "Omjer kompresije iznosi: " << compressRatio << "%" << endl;
10. }
```

Slika 6.6: Huffmanovo kodiranje - totalGain funkcija

Posljednja funkcija ujedno je i ona najbitnija, a služi za provođenje Huffmanovog kodiranja nad ulaznim tekstom. Koristi se prethodno definirana struktura *MinHeapNode* i prioritetni red *minHeap* te se algoritam izvršava po principu kako je opisano u potpoglavlju 3.2. Definicija funkcije vidi se na slici 6.7.

```
1. void huffmanCodes(int size)
2. {
3.     struct MinHeapNode *left, *right, *top;
4.     for (map<char, int>::iterator v=freq.begin(); v!=freq.end(); v++)
5.         minHeap.push(new MinHeapNode(v->first, v->second));
6.     while (minHeap.size() != 1)
7.     {
8.         left = minHeap.top();
9.         minHeap.pop();
10.        right = minHeap.top();
11.        minHeap.pop();
12.        top = new MinHeapNode('$', left->freq + right->freq);
13.        top->left = left;
14.        top->right = right;
15.        minHeap.push(top);
16.    }
17.    storeCodes(minHeap.top(), "");
18. }
```

Slika 6.7: Huffmanovo kodiranje - HuffmanCodes funkcija

Na koncu, u *main* funkciji se učitavaju ulazni podaci te ih se obrađuje koristeći prethodno opisane funkcije. Osim toga, koristeći *chrono* knjižnicu mjeri se vrijeme izvršavanja cijelog algoritma, kao što je vidljivo na slici 6.8.

```cpp
int main()
{
    cout << "############ C++ ############" << endl;

    string str, encodedString, decodedString;
    ifstream f("datoteka.txt");
    if (f) {
        ostringstream ss;
        ss << f.rdbuf();
        str = ss.str();
    }

    auto start = high_resolution_clock::now();
    calcFreq(str, str.length());
    huffmanCodes(str.length());

    cout << "Simboli sa pripadajucim kodovima:\n";

    for (auto v=codes.begin(); v!=codes.end(); v++)
        cout << v->first <<' ' << v->second << endl;

    for (auto i: str)
        encodedString+=codes[i];

    totalGain(str, encodedString);
    auto stop = high_resolution_clock::now();

    auto duration = duration_cast<milliseconds>(stop - start);

    cout << "\nEnkodirani Huffmanov niz: " << encodedString << endl;
    cout << "Vrijeme izvrsavanja iznosi: " << duration.count() << " milisekunda" << endl;

    return 0;
}
```

Slika 6.8: Huffmanovo kodiranje - main funkcija

Za ulazni niz „*anagram*“ rezultat izvršavanja programa izgleda kako je vidljivo na slici 6.9.

```
############ C++ ############
Simboli sa pripadajucim kodovima:
a 0
g 100
m 101
n 111
r 110

Velicina teksta prije kompresije (u bajtovima): 7 bajtova
Velicina teksta nakon kompresije (u bajtovima): 2 bajtova
Omjer kompresije iznosi: 71.43%

Enkodirani Huffmanov niz: 011101001100101
Vrijeme izvrsavanja iznosi: 1 milisekunda
```

Slika 6.9: Huffmanovo kodiranje – rezultati

6.2. Shannon-Fanov algoritam

Programska implementacija Shannon-Fanovog algoritma bit će opisana u Pythonu. Prvi korak je definicija klase *Node* koja će predstavljati jedan čvor binarnog stabla, kao što je vidljivo iz slike 6.10.

```
1. class Node :
2.     def __init__(self) -> None:
3.
4.         self.sym=''
5.         self.pro=0.0
6.         self.arr=[0]*20
7.         self.top=0
```

Slika 6.10: Shannon-Fanov algoritam - Node klasa

Navedena klasa sadrži četiri varijable – *sym*, *pro*, *arr* i *top*. Prva varijabla predstavlja simbol, druga predstavlja vjerojatnost pojavljivanja tog simbola u uzlaznom tekstu, treća predstavlja pripadajući Shannon-Fanov kod i četvrta predstavlja udaljenost čvora od korijena te služi isključivo kao pomoćna varijabla u Shannon-Fanovom algoritmu.

Sljedeća pomoćna funkcija služi za izračun vjerojatnosti pojavljivanja svakog simbola u ulaznom tekstu. Koristi se rječnik te se uparuje simbol sa pripadajućom vjerojatnošću. Implementacija je jednostavna, kao što je vidljivo na slici 6.11.

```
1. def CalculateProbability(the_data):
2.     the_symbols = dict()
3.     total = len(the_data)
4.     for item in the_data:
5.         if the_symbols.get(item) == None:
6.             the_symbols[item] = 1
7.         else:
8.             the_symbols[item] += 1
9.     for symbol in the_symbols:
10.             the_symbols[symbol] /= total
11.
12.         return the_symbols
```

Slika 6.11: Shannon-Fanov algoritam - CalculateProbability funkcija

Nakon toga slijedi još jedna vrlo jednostavna funkcija za sortiranje čvorova po vjerojatnosti, kao što je vidljivo na slici 6.12. Varijabla *p* predstavlja polje čvorova (klasa *Node*), odnosno binarno stablo koje je definirano u *main* funkciji.

```
1. def SortByProbability(n, p):
2.     temp = Node()
3.     for j in range(1,n) :
4.         for i in range(n - 1) :
5.             if ((p[i].pro) > (p[i + 1].pro)) :
6.                 temp.pro = p[i].pro
7.                 temp.sym = p[i].sym
8.
9.                 p[i].pro = p[i + 1].pro
10.                p[i].sym = p[i + 1].sym
11.
12.                p[i + 1].pro = temp.pro
13.                p[i + 1].sym = temp.sym
```

Slika 6.12: Shannon-Fanov algoritam - SortByProbability funkcija

Sljedeća funkcija identična je *totalGain* funkciji kod Huffmanovog kodiranja, a njena implementacija u Pythonu za potrebe Shannon-Fanovog algoritma vidljiva je na slici 6.13.

```
1. def TotalGain(the_data, coding):
2.     beforeCompression = len(the_data)
3.     afterCompression = 0
4.     the_symbols = coding.keys()
5.     for symbol in the_symbols:
6.         the_count = the_data.count(symbol)
7.         afterCompression += the_count * len(coding[symbol]) / 8
8.     afterCompression = round(afterCompression)
9.     compressionRatio = (1 - (afterCompression/beforeCompression)) * 100
10.        print("\nVelicina teksta prije kompresije (u bajtovima):",
   beforeCompression)
11.        print("Velicina teksta nakon kompresije (u
   bajtovima):",  afterCompression)
12.        print("Omjer kompresije iznosi: ", round(compressionRatio, 2), "%\n")
```

Slika 6.13: Shannon-Fanov algoritam - TotalGain funkcija

Sljedeća funkcija je ona najbitnija – *Shannon*, u kojoj se odvija Shannon-Fanov algoritam opisan u potpoglavlju 3.3. Dijeljenje na podgrupe odvija se koristeći rekurziju, gdje se svaki put trenutna grupa dijeli na dva dijela sve dok svaki čvor nema svoju podgrupu. Implementacija je vidljiva na slici 6.14.

```
1. def Shannon(l, h, p):
2.     pack1 = 0; pack2 = 0; diff1 = 0; diff2 = 0
3.     if ((l + 1) == h or l == h or l > h) :
4.         if (l == h or l > h):
5.             return
6.         p[h].top+=1
7.         p[h].arr[(p[h].top)] = 0
8.         p[l].top+=1
9.         p[l].arr[(p[l].top)] = 1
10.
11.             return
12.
13.         else :
14.             for i in range(l,h):
15.                 pack1 = pack1 + p[i].pro
16.             pack2 = pack2 + p[h].pro
17.             diff1 = pack1 - pack2
18.             if (diff1 < 0):
19.                 diff1 = diff1 * -1
20.             j = 2
21.             while (j != h - l + 1) :
22.                 k = h - j
23.                 pack1 = pack2 = 0
24.                 for i in range(l, k+1):
25.                     pack1 = pack1 + p[i].pro
26.                 for i in range(h,k,-1):
27.                     pack2 = pack2 + p[i].pro
28.                 diff2 = pack1 - pack2
29.                 if (diff2 < 0):
30.                     diff2 = diff2 * -1
31.                 if (diff2 >= diff1):
32.                     break
33.                 diff1 = diff2
34.                 j+=1
35.
36.             k+=1
37.             for i in range(l,k+1):
38.                 p[i].top+=1
39.                 p[i].arr[(p[i].top)] = 1
40.
41.             for i in range(k + 1,h+1):
42.                 p[i].top+=1
43.                 p[i].arr[(p[i].top)] = 0
44.
45.             Shannon(l, k, p)
46.             Shannon(k + 1, h, p)
```

Slika 6.14: Shannon-Fanov algoritam - Shanon funkcija

Nakon glavne funkcije slijede još tri pomoćne te *main* funkcija. Prva se zove *CalculateCodes* te izračunava kodove za svaki simbol na temelju prethodne *Shannon* funkcije. Definirana je na način vidljiv na slici 6.15.

```
1. def CalculateCodes(n, p):
2.     coding = dict()
3.     for i in range(n - 1, -1, -1):
4.         code = ''
5.         for j in range(p[i].top + 1):
6.             code += str(p[i].arr[j])
7.         coding[p[i].sym] = code
8.     return coding
```

Slika 6.15: Shannon-Fanov algoritam - CalculateCodes funkcija

Sljedeća pomoćna funkcija *OutputEncoded* vraća kompresirani (kodirani) ulazni tekst na temelju izračunatih kodova za svaki simbol iz prethodno opisane *CalculateCodes* funkcije. Njena implementacija vidljiva je na slici 6.16.

```
1. def OutputEncoded(the_data, coding):
2.     encodingOutput = []
3.     for element in the_data:
4.         encodingOutput.append(coding[element])
5.
6.     the_string = ''.join([str(item) for item in encodingOutput])
7.     return the_string
```

Slika 6.16: Shannon-Fanov algoritam - OutputEncoded funkcija

Posljednja pomoćna funkcija *Display* prikazuje sve simbole iz ulaznog teksta, njihove vjerojatnosti pojavljivanja te pripadajuće kodove. Drugim riječima, objedinjuje sve informacije dobivene u prethodnim funkcijama (izračunima). Njena definicija vidljiva je na slici 6.17.

```
1. def Display(n, p):
2.     print("\tSimbol\tVjerojatnost\tKod",end='')
3.     for i in range(n - 1,-1,-1):
4.         print("\n\t", p[i].sym, "\t\t\t", round(p[i].pro, 3),"\t\t",end='')
5.         for j in range(p[i].top+1):
6.             print(p[i].arr[j],end='')
7.     print("\n")
```

Slika 6.17: Shannon-Fanov algoritam - Display funkcija

I na koncu, *main* funkcija iz koje se pozivaju i objedinjuju sve prethodno opisane funkcije. Na samom početku učitavaju se ulazni podaci koji se onda obrađuju te na kraju ispisuju rezultati obrade, odnosno kompresiranja. Osim toga, koristeći *time* knjižnicu mjeri se vrijeme izvršavanja cijelog algoritma, kao što je vidljivo na slici 6.18.

```
1. if __name__ == '__main__':
2.     print("############ Python ############")
3.
4.     file = open("datoteka.txt", "r")
5.     the_data = file.read()
6.
7.     start = time.time()
8.
9.     symbolWithProbs = CalculateProbability(the_data)
10.    n = len(symbolWithProbs.keys())
11.    p=[Node() for _ in range(n)]
12.    x = list(symbolWithProbs.values())
13.
14.    for i, j in zip(range(n), list(symbolWithProbs.keys())):
15.        p[i].sym = j
16.        p[i].pro = x[i]
17.
18.    SortByProbability(n, p)
19.
20.    for i in range(n):
21.        p[i].top = -1
22.
23.    Shannon(0, n - 1, p)
24.
25.    coding = CalculateCodes(n, p)
26.    encoded = OutputEncoded(the_data, coding)
27.
28.    Display(n, p)
29.    TotalGain(the_data, coding)
30.    end = time.time()
31.
32.    print("Enkodirani Shannon-Fanov niz: ", encoded)
33.    print("Vrijeme izvodenja iznosi: ", round((end - start) * 1000, 2), "ms")
```

Slika 6.18: Shannon-Fanov algoritam – main funkcija

Za ulazni niz „*anagram*“ rezultat izvršavanja programa izgleda kako je vidljivo na slici 6.19.

```
############ Python ############
    Simbol  Vjerojatnost    Kod
     a          0.429       0
     m          0.143       100
     r          0.143       101
     g          0.143       110
     n          0.143       111


Velicina teksta prije kompresije (u bajtovima): 7
Velicina teksta nakon kompresije (u bajtovima): 2
Omjer kompresije iznosi:  71.43 %

Enkodirani Shannon-Fanov niz:  011101101010100
Vrijeme izvodenja iznosi:  0.0 ms
```

Slika 6.19: Shannon-Fanov algoritam – rezultati

6.3. LZW

Programska implementacija LZW algoritma bit će opisana u C++-u. Za razliku od prethodno opisanih entropijskih algoritama, ovaj algoritam temeljen na metodama rječnika ima puno jednostavniju implementaciju. Na slici 6.20. vidi se definicija rječnika *table* te funkcije *encoding*.

U funkciji, prvi je korak inicijalizirati rječnik sa prvih 256 simbola kako je opisano u potpoglavlju 4.1. Nakon toga slijedi implementacija opisana pseudokodom na slici 4.2. Ukratko, algoritam pročita prvi simbol koji postaje radni niz. Nakon toga, čita sljedeći simbol koji dodaje na radni niz te provjerava postoji li ta nova riječ u rječniku. Ako postoji onda nova riječ postaje novi radni niz, a ako ne postoji onda na izlaz šalje kod za radni niz, novu riječ zapisuje u rječnik i dodjeljuje joj pripadajući kod te novi radni niz postaje sljedeći simbol na ulazu. Algoritam se izvršava sve dok ima simbola na ulazu.

```
unordered_map<string, int> table;
vector<int> encoding(string s1, int maximum_table_size)
{
    for (int i = 0; i <= 255; i++) {
        string ch = "";
        ch += char(i);
        table[ch] = i;
    }

    string p = "", c = "";
    p += s1[0];
    int code = 256;
    vector<int> output_code;
    cout << "Niz\tKod\tIndeks\n";

    for (int i = 0; i < s1.length(); i++) {
        if (i != s1.length() - 1)
            c += s1[i + 1];
        if (table.find(p + c) != table.end()) {
            p = p + c;
        }
        else {
            cout << p << "\t" << table[p] << "\t"
                 << p + c << "\t" << code << endl;
            output_code.push_back(table[p]);
            if (table.size() <= maximum_table_size)
            {
                table[p + c] = code;
                code++;
            }
            p = c;
        }
        c = "";
    }
    cout << p << "\t" << table[p] << endl;
    output_code.push_back(table[p]);
    return output_code;
}
```

Slika 6.20: LZW algoritam - encoding funkcija

Nakon toga, slijedi već poznata *totalGain* funkcija koja izračunava omjer kompresije. Ovaj puta je u malo izmijenjenom izdanju jer, kao što je objašnjeno u potpoglavlju 4.1., veličina kodnih riječi ovisi o veličini rječnika. Implementacija je vidljiva na slici 6.21.

```
void totalGain(string s, vector<int> output_code)
{
    float beforeCompresion = s.size();
    cout << "\nVelicina teksta prije kompresije (u bajtovima): " << beforeCompresion << " bajtova" << endl;
    float afterCompresion = round(output_code.size() * 12.0 / 8.0);
    cout << "Velicina teksta nakon kompresije (u bajtovima): " << afterCompresion << " bajtova" << endl;
    float compressRatio = (1 - (afterCompresion / beforeCompresion)) * 100 ;
    compressRatio = std::ceil(compressRatio * 100.0) / 100.0;

    cout << "Omjer kompresije iznosi: " << compressRatio << "%" << endl;
}
```

Slika 6.21: LZW algoritam - totalGain funkcija

I za kraj, slijedi glavna *main* funkcija u kojoj se, kao i kod svih opisanih algoritama, učitavaju ulazni podaci koji se onda obrađuju i objedinjuju u smisleni rezultat. Naravno, i kod LZW algoritma koristi se *chrono* knjižnica kako bi se precizno izračunalo vrijeme izvođenja kompletnog algoritma, kao što je vidljivo na slici 6.22.

```
1. int main()
2. {
3.     cout << "############ C++ ############" << endl;
4.     int maximum_table_size = 4095;
5.
6.     string str;
7.     ifstream f("zakon_o_radu.txt");
8.     if (f) {
9.         ostringstream ss;
10.        ss << f.rdbuf();
11.        str = ss.str();
12.    }
13.
14.    auto start = high_resolution_clock::now();
15.    vector<int> output_code = encoding(s, maximum_table_size);
16.    totalGain(s, output_code);
17.    auto stop = high_resolution_clock::now();
18.
19.    auto duration = duration_cast<milliseconds>(stop - start);
20.
21.    cout << "Kodovi su: ";
22.    for (int i = 0; i < output_code.size(); i++) {
23.        cout << output_code[i] << " ";
24.    }
25.
26.    cout << "\nBroj kodova iznosi: " << output_code.size() << endl;
27.    cout << "Velicina rjecnika iznosi: " << table.size() << endl;
28.    cout << "Vrijeme izvrsavanja iznosi: " << duration.count() << " milisekunda" << endl;
29.}
```

Slika 6.22: LZW algoritam - main funkcija

Za ulazni niz „*anananananana*“, rezultat izvođenja opisanog programa izgleda kako je vidljivo na slici 6.23. Opisani algoritam najbolje radi na redundantnim tekstovima pa je zato korišten drugačiji ulazni niz nego kod ostalih algoritama. Za kratki i neredundantni ulazni tekst omjer kompresije bio bi negativan, odnosno „kompresirani“ tekst bio bi veći nego originalni (nekompresirani).

Nadalje, budući da je ulazni niz kratak, korišten je rječnik veličine 512 (a ne 4096 što je uobičajeno) kako bi kodovi bili veličine 9 bitova. Također, vidi se da je to bio ispravan potez budući da je iskorišteno samo 261 mjesto od 512 slobodnih u rječniku. Da je ulazni tekst još dulji i jednako redundantan, omjer kompresije bio bi veći.

```
############ C++ ############
Niz     Kod     Indeks
a       97      an      256
n       110     na      257
an      256     ana     258
ana     258     anan    259
na      257     nan     260
na      257

Velicina teksta prije kompresije (u bajtovima): 11 bajtova
Velicina teksta nakon kompresije (u bajtovima): 7 bajtova
Omjer kompresije iznosi: 36.37%

Kodovi su: 97 110 256 258 257 257
Broj kodova iznosi: 6
Velicina rjecnika iznosi: 261
Vrijeme izvrsavanja iznosi: 2 milisekunda
```

Slika 6.23: LZW algoritam - rezultati

6.4. LZSS

Programska implementacija LZW algoritma bit će opisana u Pythonu. Prvi korak je definicija funkcije *elements_in_array* kojom se provjerava nalaze li se simboli u određenom redoslijedu u međuspremniku pretraživanja. Ako da, funkcija će vratiti indeks na kojem niz elemenata u međuspremniku pretraživanja započinje, a ukoliko ne, funkcija će vratiti -1, kao što je vidljivo na slici 6.24. Veličina međuspremnika pretraživanja postavljena je na standardnih 4096.

```
1.  def elements_in_array(check_elements, elements):
2.      i = 0
3.      offset = 0
4.      for element in elements:
5.          if len(check_elements) <= offset:
6.              return i - len(check_elements)
7.
8.          if check_elements[offset] == element:
9.              offset += 1
10.         else:
11.             offset = 0
12.
13.         i += 1
14.     return -1
```

Slika 6.24: LZSS algoritam - elements_in_array funkcija

Nakon toga slijedi funkcija *encoding* u kojoj je implementiran LZSS algoritam opisan u potpoglavlju 4.2. Implementacija je vidljiva na slici 6.25.

```
1. def encoding(text, max_sliding_window_size=4096):
2.
3.     search_buffer = []
4.     check_characters = []
5.     output = []
6.
7.     i = 0
8.     for char in text:
9.         index = elements_in_array(check_characters, search_buffer)
10.
11.        if elements_in_array(check_characters + [char], search_buffer) == -1 or i == len(text)
   - 1:
12.            if i == len(text) - 1 and elements_in_array(check_characters + [char],
   search_buffer) != -1:
13.                check_characters.append(char)
14.
15.            if len(check_characters) > 1:
16.                index = elements_in_array(check_characters, search_buffer)
17.                offset = i - index - len(check_characters)
18.                length = len(check_characters)
19.
20.                token = f"<{offset},{length}>"
21.
22.                if len(token) > length:
23.                    output.extend(check_characters)
24.                else:
25.                    output.extend(token)
26.                search_buffer.extend(check_characters)
27.            else:
28.                output.extend(check_characters)
29.                search_buffer.extend(check_characters)
30.
31.            check_characters = []
32.        check_characters.append(char)
33.
34.        if len(search_buffer) > max_sliding_window_size:
35.            search_buffer = search_buffer[1:]
36.        i += 1
37.
38.    return output
```

Slika 6.25: LZSS algoritam - encoding funkcija

Navedena funkcija prvo provjerava je li trenutni element ujedno i posljednji element na ulazu. Ukoliko je, dodaje je ga u polje *check_characters* koje čini niz elemenata koji se u tekstu pojavljuju jedan za drugim. Nadalje, izvršava se LZSS algoritam; spomenuto polje šalje se na provjeru u prethodno opisanu funkciju *elements_in_array*, gdje se uspoređuje s trenutnim međuspremnikom pretraživanja te se dohvaća se indeks na kojem započinje niz.

Nakon toga se, iz dobivenog indeksa, izračunavaju pomak i duljina koji čine tzv. token, odnosno referencu. Ako je duljina reference veća od duljine originalnog teksta koji predstavlja, u izlazno polje zapisuje se tekst, i obrnuto. Na kraju, trenutni niz iz *check_characters* polja dodaje se u

međuspremnik pretraživanja i provjerava se je li spomenuti međuspremnik pun. Ukoliko je, briše se njegov prvi element te algoritam nastavlja s izvršavanjem sve dok ima simbola na ulazu.

Sljedeća funkcija je *main* funkcija i njena implementacija vidljiva je na slici 6.26. U njoj se učitavaju ulazni podaci te se prosljeđuju prethodno opisanoj funkciji. Također, kao i svim algoritmima do sada, i LZSS-u se mjeri vrijeme izvršavanja.

```
1. if __name__ == "__main__":
2.     print("############ Python ############")
3.     file = open("zakon_o_radu_1.txt", "rb")
4.     text = file.read()
5.
6.     start = time.time()
7.     encoded = encoding(text)
8.     compressed = round((1 - (len(encoded) / len(text))) * 100, 2)
9.     end = time.time()
10.
11.    print(f"Veličina prije kompresije (u bajtovima): {len(text)}")
12.    print(f"Veličina nakon kompresije (u bajtovima): {len(encoded)}\n")
13.    print("Omjer kompresije: ", compressed, "%")
14.    print("Vrijeme izvodenja iznosi: ", round((end - start) * 1000, 2), "ms")
```

Slika 6.26: LZSS algoritam - main funkcija

Za ulazni niz „*ja sam filip. ja sam filip. ja sam filip.*“, rezultat izvođenja opisanog programa izgleda kako je vidljivo na slici 6.27. Korišten je ulazni tekst koji se ponavlja kako bi algoritam došao do izražaja. U suprotnom, na kratkom tekstu koji nema dijelova koji se ponavljaju, kompresija bi bila minimalna ili je uopće ne bi bilo.

```
############ Python ############
Veličina prije kompresije (u bajtovima): 41
Veličina nakon kompresije (u bajtovima): 28

Omjer kompresije:  31.71 %
Vrijeme izvodenja iznosi:  0.0 ms
```

Slika 6.27: LZSS algoritam – rezultati

6.5. DEFLATE

Programsko korištenje DEFLATE algoritma bit će opisano u C++-u. Kao što je već spomenuto u 5. poglavlju, navedeni algoritam (uz LZMA) vrlo je složen i koristi tehnike niske razine programiranja te stoga neće biti prikazana njegova implementacija, već samo korištenje uz pomoć *zlib* knjižnice.

Prva funkcija je *deflate_compress* koja koristi metode i tehnike definirane u *zlib* knjižnici. Kao što je vidljivo na slici 6.28, prvi korak je inicijalizacija niza (engl. *stream*) koji će se koristiti za kompresiju. Pri inicijalizaciji, odabire se i željena razina kompresije. Sljedeći korak je definicija veličine tog niza te pripadajućeg međuspremnika. Koristeći *do-while* petlju te navedeni niz i međuspremnik vrši se kompresija tako da se u jedinici vremena može maksimalno kompresirati podataka onoliko koliko je velik međuspremnik. Nakon toga provjerava se je li kompresija uspješna te, ukoliko nije, ispisuje se prigodna poruka.

```
1. string deflate_compress(const string& str, int compressionLevel = Z_BEST_COMPRESSION)
2. {
3.     z_stream zs;
4.     memset(&zs, 0, sizeof(zs));
5.
6.     if (deflateInit(&zs, compressionLevel) != Z_OK)
7.     {
8.         throw(runtime_error("deflateInit radnja nije uspijela"));
9.     }
10.
11.    zs.next_in = (Bytef*)str.data();
12.    zs.avail_in = str.size();
13.
14.    int ret;
15.    char outbuffer[32768];
16.    string outstring;
17.
18.    do {
19.        zs.next_out = reinterpret_cast<Bytef*> (outbuffer);
20.        zs.avail_out = sizeof(outbuffer);
21.
22.        ret = deflate(&zs, Z_FINISH);
23.
24.        if (outstring.size() < zs.total_out) {
25.            outstring.append(outbuffer, zs.total_out - outstring.size());
26.        }
27.    } while (ret == Z_OK);
28.
29.    deflateEnd(&zs);
30.
31.    if (ret != Z_STREAM_END) {
32.        ostringstream oss;
33.        oss << "Pogreska tijekom kompresije: (" << ret << ") " << zs.msg;
34.        throw(runtime_error(oss.str()));
35.    }
36.
37.    return outstring;
38. }
```

Slika 6.28: DEFLATE - deflate_compress funkcija

Za izračun omjera kompresije koristi se, već jako dobro poznata, *totalGain* funkcija čija je programska implementacija vidljiva u prethodno objašnjenim algoritmima.

Kao i u svim *main* funkcijama do sada, tako se i u ovoj ulazni podaci učitavaju te šalju na daljnju obradu prethodno opisanim funkcijama. Implementacija *main* funkcije vidljiva je na slici 6.29. Također, mjeri se vrijeme izvršavanja.

```
1. int main() {
2.     cout << "############ C++ ############" << endl;
3.     string s;
4.
5.     ifstream f("datoteka.txt");
6.     (if (f) {
7.         ostringstream ss;
8.         ss << f.rdbuf();
9.         s = ss.str();
10.    }
11.    auto start = high_resolution_clock::now();
12.    string compressed = deflate_compress(s);
13.    totalGain(s, compressed);
14.    auto stop = high_resolution_clock::now();
15.    auto duration = duration_cast<milliseconds>(stop - start);
16.
17.    cout << "Vrijeme izvrsavanja iznosi: " << duration.count() << " milisekunda" << endl;
18.}
```

Slika 6.29: DEFLATE - main funkcija

Za ulazni niz „*RijekaRijekaRijekaRijekaRijekaRijekaRijeka*“, rezultat izvođenja programa izgleda kako je vidljivo na slici 6.30.

```
############ C++ ############

Velicina teksta prije kompresije (u bajtovima): 42 bajtova
Velicina teksta nakon kompresije (u bajtovima): 17 bajtova
Omjer kompresije iznosi: 59.53%
Vrijeme izvrsavanja iznosi: 0 milisekunda
```

Slika 6.30: DEFLATE – rezultati

6.6. LZMA

Programsko korištenje LZMA algoritma bit će opisano u Pythonu. Kao ni za DEFLATE algoritam, tako ni za LZMA neće prikazana njegova implementacija, već samo korištenje uz pomoć *lzma* knjižnice.

Korištenje LZMA algoritma u Pythonu trivijalno je i svodi se samo na pozivanje funkcije *compress* iz knjižnice *lzma*, kao što je vidljivo na slici 6.31. Također, mjeri se vrijeme izvršavanja.

```
1. print("############ Python ############")
2. file = open("datoteka.txt", "rb")
3. text = file.read()
4. start = time.time()
5. compressed = lzma.compress(text)
6. TotalGain(text, compressed)
7. end = time.time()
8.
9. print("Vrijeme izvodenja iznosi: ", round((end - start) * 1000, 2), "ms")
```

Slika 6.31: LZMA - main funkcija

Naravno, kao i kod svih algoritama do sada, koristiti se *TotalGain* funkcija za izračun omjera kompresije. Zbog smanjenja redundancije, rečena funkcija neće biti prikazana, već se njena implementacija može vidjeti kod prethodnih algoritama.

Za ulazni niz „*RijekaRijekaRijekaRijekaRijekaRijekaRijeka…*“, rezultat izvođenja programa izgleda kako je vidljivo na slici 6.32.

```
############ Python ############

Velicina teksta prije kompresije (u bajtovima): 321
Velicina teksta nakon kompresije (u bajtovima): 96
Omjer kompresije iznosi:  70.09 %
Vrijeme izvodenja iznosi:  0.0 ms
```

Slika 6.32: LZMA – rezultati

7. REZULTATI

Kao što je u potpoglavlju 2.3. navedeno, opisani algoritmi testirani su na pet različitih tekstova, od koji su dva na hrvatskome jeziku te tri na engleskome jeziku. Hrvatska abeceda sadrži 27, a engleska 26 simbola. Način izračuna omjera kompresije opisan na početku 6. poglavlja. Vremena izvršavanja mjere se koristeći funkcije pripadajućih knjižnica. Točnije, u C++-u se vrijeme izvršavanja mjeri koristeći funkcije *chrono* knjižnice, a u Pythonu koristeći funkcije *time* knjižnice.

Prvo su prikazani omjeri kompresija algoritama po ulaznim datotekama te nakon toga vremena izvršavanja algoritama po ulaznim datotekama ovisno o programskom jeziku.

7.1. Omjeri kompresije

Algoritmi u Pythonu i C++-u su jednaki, što znači da daju i jednake omjere kompresije, tako da u ovom dijelu rezultata neće biti odvojeni po jeziku implementacije.

Tablični prikaz rezultata, odnosno omjera kompresije za ulazni tekst „*Europski zeleni plan*“ vidljiv je u tablici 7.1.

Tablica 7.1: Rezultati kompresije za Europski zeleni plan

Algoritam	Veličina prije kompresije	Veličina nakon kompresije	Omjer kompresije
Huffmanov	77330 B	41992 B	45,7%
Shannon-Fanov		42148 B	45,5%
LZW		35979 B	53,5%
LZSS		72663 B	6%
DEFLATE		25729 B	66,7%
LZMA		23604 B	69,5%

Kao što je iz tablice vidljivo, za navedeni ulazni tekst na hrvatskom jeziku, entropijski algoritmi daju podjednake rezultate gdje je Huffmanovo kodiranje minimalno bolje jer, kao što je objašnjeno u potpoglavlju 3.3., za razliku od Shannon-Fanovog algoritma uvijek daje optimalne kodove.

Od algoritama temeljenim na metodama rječnika LZW je definitivni pobjednik. Koristeći rječnik veličine 4096 kada su kodne riječi duljine čak 12 bitova, algoritam daje zapanjujuće rezultate. LZSS je podbacio jer očekuje jako redundantan tekst i tada dolazi do izražaja, ali na nekom „svakodnevnom“ tekstu rezultati su poražavajući.

Kako su nazvani u ovom radu, tzv. napredni algoritmi pokazali su da to stvarno i jesu. Daju podjednake rezultate s malom prednošću LZMA algoritma. Pokazalo se da je kodiranje raspona uz razne optimizacijske tehnike kod LZMA učinkovitije od Huffmanovog kodiranja kod DEFLATE algoritma.

Rezultati, odnosno omjeri kompresije za sljedeći ulazni tekst „*Zakon o radu*“ vidljivi su u tablici 7.2.

Tablica 7.2: Rezultati kompresije za Zakon o radu

Algoritam	Veličina prije kompresije	Veličina nakon kompresije	Omjer kompresije
Huffmanov	214598 B	115577 B	46,1%
Shannon-Fanov		115795 B	46%
LZW		96266 B	55,1%
LZSS		174095 B	18,9%
DEFLATE		53590 B	75%
LZMA		44024 B	79,5%

Kao što je iz tablice vidljivo, rezultati kompresije podjednaki su kao za prethodni ulazni tekst. Drugim riječima, entropijski algoritmi i dalje daju sličan omjer kompresije s minimalnom prednošću Huffmanovog kodiranja. Što se tiče algoritama temeljenih na metodama rječnika, LZW daje sličan rezultat kao za prethodni ulazni tekst, ali je zato rezultat LZSS-a značajno poboljšan, čak za ~13%. Napredni algoritmi također daju bolje rezultate te općenito postižu najbolje rezultate, ali to je i očekivano.

Rezultati, odnosno omjeri kompresije za sljedeći ulazni tekst „*Uvod u algoritme*“ vidljivi su u tablici 7.3.

Tablica 7.3: Rezultati kompresije za Uvod u algoritme

Algoritam	Veličina prije kompresije	Veličina nakon kompresije	Omjer kompresije
Huffmanov	2388618 B	1361345 B	43%
Shannon-Fanov		1366923 B	42,8%
LZW		1279066 B	46,5%
LZSS		2107053 B	11,8%
DEFLATE		681520 B	71,5%
LZMA		539340 B	77,4%

Iako su prethodni ulazni tekstovi bili na hrvatskom jeziku, a ovaj je na engleskom, ne vide se značajne razlike u rezultatima kompresije, štoviše rezultati su jako slični. Drugim riječima, svi rezultati su u skladu s prethodnima na hrvatskom jeziku.

Rezultati, odnosno omjeri kompresije za sljedeći ulazni tekst „*Sveta Biblija*“ vidljivi su u tablici 7.4.

Tablica 7.4: Rezultati kompresije za Svetu Bibliju

Algoritam	Veličina prije kompresije	Veličina nakon kompresije	Omjer kompresije
Huffmanov	4653740 B	2490473 B	46,5%
Shannon-Fanov		2511324 B	46%
LZW		2149952 B	53,8%
LZSS		4330276 B	7%
DEFLATE		1394797 B	70%
LZMA		1045448 B	77,5%

Rezultati su u skladu s prethodnima, odnosno ne vide se značajnija odstupanja. Jedino je ponovno podbacio LZSS kojem ulazni tekst nije dovoljno redundantan.

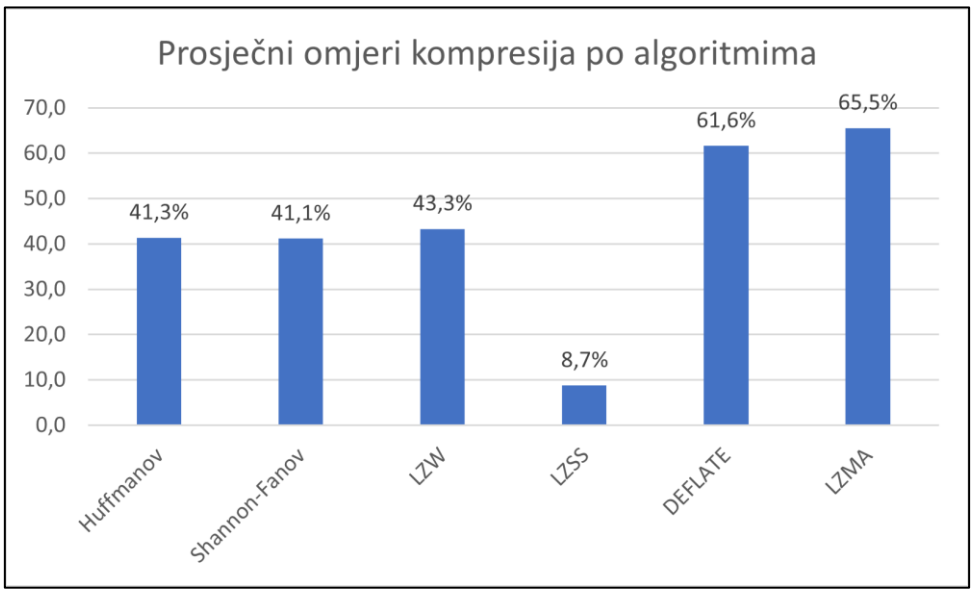
Rezultati, odnosno omjeri kompresije za sljedeći, a ujedno i posljednji ulazni tekst „*Slučajni tekst*“ vidljivi su u tablici 7.5.

Tablica 7.5: Rezultati kompresije za Slučajni tekst

Algoritam	Veličina prije kompresije	Veličina nakon kompresije	Omjer kompresije
Huffmanov	100000 B	74575 B	25,4%
Shannon-Fanov		74583 B	25,4%
LZW		92418 B	7,58%
LZSS		99999 B	0%
DEFLATE		75200 B	24,8%
LZMA		76252 B	23,8%

Rezultati za „*Slučajni tekst*“ vrlo su zanimljivi u odnosu na ostale. Niti jedan algoritam ne uspijeva postići značajnu kompresiju, odnosno rezultati su puno lošiji u odnosu na prethodne ulazne tekstove. To se događa zato jer nema smislenog redoslijeda riječi i rečenica koje se ponavljaju i koje bi algoritmi „zapamtili“. Entropijski algoritmi ostvaruju jednake rezultate, LZW ostvaruje puno lošiji rezultat nego na prethodnim tekstovima, a LZSS uopće ne uspijeva kompresirati tekst. Nadalje, napredni algoritmi također ne dolaze do izražaja te ostvaruju rezultate podjednake kao entropijski algoritmi, odnosno marginalno lošije.

I na koncu, na slici 7.1. vidi se grafički prikaz prosječnih omjera kompresija po algoritmima.



Slika 7.1: Prosječni omjeri kompresije po algoritmima

Kao što se iz slike vidi, prosječni omjeri kompresija po algoritmima ne odstupaju mnogo od onih koje su vidljive u prethodnim tablicama. Najviše prosjek kvari posljednji testirani ulazni tekst – „*Slučajni tekst*“ koji je iznenadio sve algoritme.

Iz slike 7.1 i svih prethodnih grafova može se vidjeti da entropijski algoritmi gotovo uvijek daju podjednake omjere kompresije. LZW bez posljednjeg ulaznog teksta u prosjeku daje ~7% bolje omjere kompresije nego entropijski algoritmi. Posljednji ulazni tekst, za koji je vrlo malo vjerojatno da se svakodnevno koristi, značajno mu smanjuje taj prosjek. LZSS konstantno daje najlošiji omjer kompresije za sve ulazne tekstove. Napredni algoritmi daju značajno najbolje omjere kompresije, ali to je i za očekivati budući da koriste više kompresijskih tehnika istovremeno i dodatne optimizacijske metode.

7.2. Vremena izvršavanja

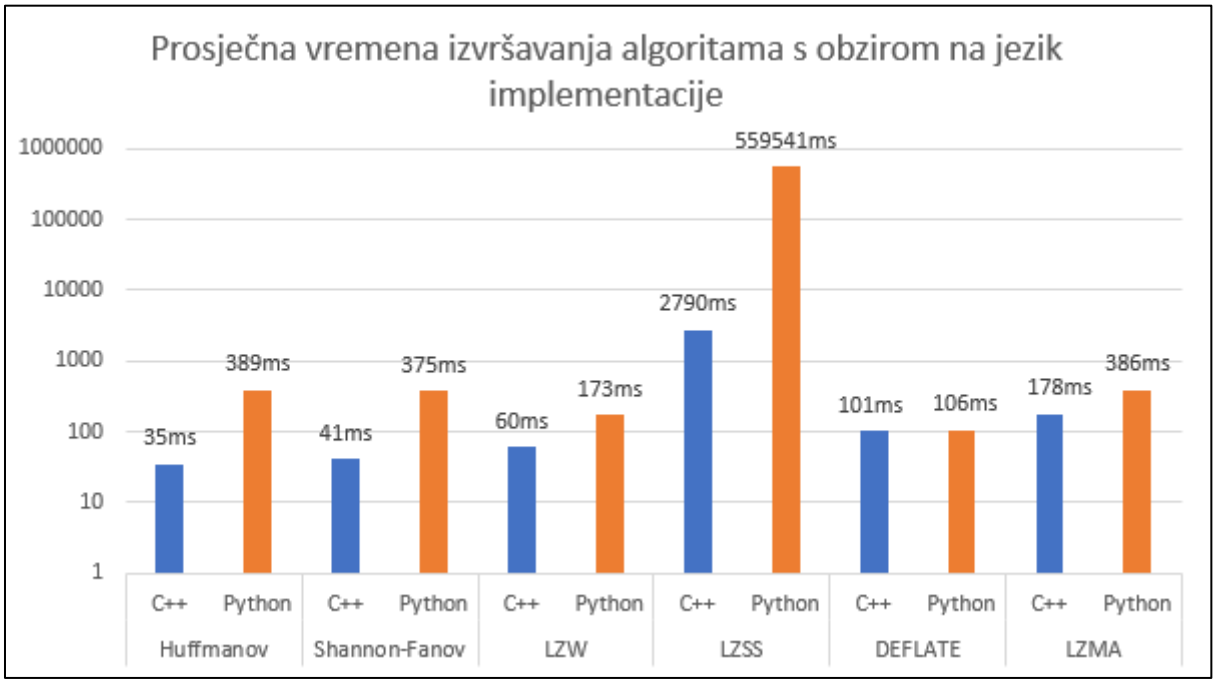
U ovome dijelu rezultata algoritmi su podijeljeni u podskupine s obzirom na jezik implementacije. Nadalje, nema međusobne usporedbe različitih algoritama u različitim programskim jezicima. Drugim riječima, uspoređuju se samo vremena izvršavanja istih algoritama u različitim programskim jezicima – C++-u i Pythonu. Vremena izvršavanja mjerena su na način da se uzeo prosjek od pet izvršavanja (po programskom jeziku) svakog algoritma za svaki ulazni tekst te je to vrijeme izraženo u milisekundama kao cijeli broj.

Sva vremena izvršavanja algoritama s obzirom na jezik implementacije vidljiva su tablici 7.6.

Tablica 7.6: Vremena izvršavanja algoritama s obzirom na jezik implementacije

Algoritam	Jezik implementacije	Europski zeleni plan	Zakon o radu	Uvod u algoritme	Sveta Biblija	Slučajni tekst
Huffmanov	C++	3ms	6ms	55ms	105ms	5ms
	Python	16ms	47ms	640ms	1218ms	25ms
Shannon-Fanov	C++	3ms	6ms	64ms	124ms	6ms
	Python	20ms	47ms	610ms	1180ms	16ms
LZW	C++	4ms	10ms	94ms	189ms	5ms
	Python	10ms	26ms	281ms	531ms	16ms
LZSS	C++	355ms	362ms	3791ms	9256ms	184ms
	Python	21786ms	72371ms	831584ms	1834474ms	37492ms
DEFLATE	C++	4ms	12ms	122ms	365ms	3ms
	Python	7ms	17ms	125ms	375ms	5ms
LZMA	C++	5ms	13ms	240ms	628ms	6ms
	Python	11ms	31ms	484ms	1392ms	14ms

Prosječna vremena izvršavanja algoritama s obzirom na jezik implementacije vidljiva su na slici 7.2.



Slika 7.2: Prosječna vremena izvršavanja algoritama s obzirom na jezik implementacije

Rezultati su potpuno očekivani, odnosno kod svih algoritama C++ verzije imaju kraća vremena izvršavanja. Razlike su od 2 puta bržeg izvođenja do čak 200 puta bržeg izvođenja kao što je slučaj kod LZSS-a. Spomenuti algoritam ima uvjerljivo najsporija vremena izvršavanja te mu je za neke ulazne tekstove u Python implementaciji potrebno čak 30 minuta da ih kompresira.

Iznimke čine DEFLATE i LZMA algoritmi jer se njihove Python implementacije u pozadini izvršavaju u C-u (zbog korištenja knjižnica), tako da je kod njih manja razlika između vremena izvršavanja u C++-u i Pythonu.

Svi ostali algoritmi, u svim implementacijama, na svim ulaznim tekstovima izvršavaju se trenutno ili u nekoliko sekundi. Također, kod svih algoritama apsolutna razlika u vremenu izvršavanja veća je što je ulazni tekst dulji, ali relativna razlika uvijek je približno jednaka. Zaključci izvedeni iz rezultata navedenih u ovom poglavlju, opisani su u sljedećem poglavlju.

8. ZAKLJUČAK

Cilj ovog rada bio je opisati odabrane algoritme, programski ih implementirati i testirati ih na raznim ulaznim tekstovima. Iz rezultata testiranja opisanih u prethodnom poglavlju zaključuje se da odabrani entropijski algoritmi daju podjednake rezultate. Točnije, daju podjednake omjere kompresije za gotovo sve ulazne tekstove, ali je Huffmanovo kodiranje uvijek marginalno bolje. Također, vremena izvršavanja u obje implementacije vrlo su izjednačena, odnosno niti jedan nije značajno brži ili sporiji. Zanimljivo, za posljednji ulazni tekst „*Slučajni tekst*“ daju najbolji omjer kompresije.

Što se tiče algoritama temeljenih na metodama rječnika zaključuje se da je LZW brz i efikasan dok LZSS podbacuje u svim segmentima. LZW u prosjeku daje bolje omjere kompresije nego entropijski algoritmi za 2,1%. Nadalje, Python implementacija navedenog algoritma u prosjeku je ~55% brža nego ista implementacija entropijskih algoritama. Za C++ implementaciju vrijedi suprotno, odnosno u prosjeku je ~58% sporija nego ista implementacija entropijskih algoritama. LZSS je definitivno najlošiji algoritam koji je testiran u ovom radu. Osim što daje najgori omjer kompresije (u prosjeku samo 8,7%), ima najsporiju brzinu izvršavanja, pogotovo u Python implementaciji. Konkretno, najkraći testirani ulazni tekst „*Europski zeleni plan*“ kompresira čak ~22 sekunde. S druge strane, C++ implementacija najdulji ulazni tekst „*Sveta Biblija*“ kompresira za „samo“ ~9 sekundi. Dakle, kod LZSS-a najveća je razlika u brzini izvršavanja između jezika implementacije.

Napredni algoritmi, očekivano, ostvaruju najbolje rezultate, odnosno najefikasniji su. Međusobno gledano, LZMA u prosjeku daje 3,9% bolje omjere kompresije, ali uz kompromis u vidu brzine izvođenja. Naime, C++ implementacija LZMA algoritma je u prosjeku ~76% sporija od iste implementacije DEFLATE algoritma, dok je Python implementacija u prosjeku čak ~264% sporija od iste implementacije DEFLATE algoritma. Kao što je u 5. poglavlju navedeno, napredni algoritmi služe kao referentne točke za usporedbu jer nisu samostalno implementirani, nego korištenjem knjižnica što znači da su maksimalno optimizirani.

Stoga, LZW u C++ implementaciji najbolji je algoritam za kompresiju teksta koji je samostalno implementiran. U prosjeku ostvaruje 20,3% manji omjer kompresije nego napredni algoritmi, ali je zato ~40% brži od DEFLATE algoritma u istoj implementaciji i ~66% brži od LZMA algoritma u istoj implementaciji.

LITERATURA

[1] „What is data compression?“, s Interneta, https://www.barracuda.com/glossary/data-compression, 15.9.2022.

[2] „C++ vs Python benchmarks“, s Interneta, https://programming-language-benchmarks.vercel.app/cpp-vs-python, 17.9.2022.

[3] „Visual Studio Code's Documentation“, s Interneta, https://code.visualstudio.com/docs, 15.9.2022.

[4] „Spyder's Documentation“, s Interneta, https://docs.spyder-ide.org/current/index.html, 15.9.2022.

[5] „Europski zeleni plan“, s Interneta, https://mingor.gov.hr/UserDocsImages//Istaknute%20teme/Zeleni%20plan//Europski%20zeleni%20plan%20HR%20(pdf).pdf, 4.9.2022.

[6] „Zakon o radu“, s Interneta, https://www.zakon.hr/z/307/Zakon-o-radu, 4.9.2022.

[7] „Introduction to Algorithms, Third Edition“, s Interneta, https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_(2009).pdf, 4.9.2022.

[8] „Holy Bible“, s Interneta, http://triggs.djvu.org/djvu-editions.com/BIBLES/DRV/Download.pdf, 4.9.2022.

[9] „Random string generator“, s Interneta, http://www.unit-conversion.info/texttools/random-string-generator/, 4.9.2022.

[10] Ilić, Bažanat, Beriša: „Teorija informacije, kapacitet diskretnog komunikacijskog kanala, Markovljevi lanci“, s Interneta, https://element.hr/wp-content/uploads/2020/06/unutra-13605.pdf, 20.9.2022.

[11] Ožuška Mario: „Metode entropijskog kodiranja“, s Interneta, https://repozitorij.fizika.unios.hr/islandora/object/fizos%3A48/datastream/PDF/view, 20.9.2022.

[12] „Huffman coding“, s Interneta, https://en.wikipedia.org/wiki/Huffman_coding, 20.9.2022.

[13] „Shannon-Fano coding“, s Interneta, https://en.wikipedia.org/wiki/Shannon%E2%80%93Fano_coding, 20.9.2022.

[14] Pandžić, Bažant, Ilić, Vrdoljak, Kos, Sinković: „Uvod u teoriju informacije i kodiranje“, Element d.o.o., 2007.

[15] Kaur, Singh: „Entropy Coding and Different Coding Techniques“, s Interneta, https://web.archive.org/web/20191203151816/https://pdfs.semanticscholar.org/4253/7898a836d0384c6689a3c098b823309ab723.pdf, 21.9.2022.

[16] „Python Dict and File“, s Interneta, https://developers.google.com/edu/python/dict-files, 21.9.2022.

[17] „The LZ78 algorithm“, s Interneta, https://archive.ph/20130107200800/http://oldwww.rasip.fer.hr/research/compress/algorithms/fund/lz/lz78.html, 22.9.2022.

[18] „LZW“, s Interneta, https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch, 22.9.2022.

[19] „LZW Data Compression“, s Interneta, https://www2.cs.duke.edu/csed/curious/compression/lzw.html, 22.9.2022.

[20] „The LZ77 algorithm“, s Interneta, https://archive.ph/20130107232302/http://oldwww.rasip.fer.hr/research/compress/algorithms/fund/lz/lz77.html, 22.9.2022.

[21] „LZSS“, s Interneta, https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Storer%E2%80%93Szymanski, 23.9.2022.

[22] „DEFLATE Compressed Data Format Specification“, s Interneta, https://datatracker.ietf.org/doc/html/rfc1951#section-Abstract, 25.9.2022.

[23] „DEFLATE“, s Interneta, https://en.wikipedia.org/wiki/Deflate, 25.9.2022.

[24] „DEFLATE Compression Algorithm“, s Interneta, http://pnrsolution.org/Datacenter/Vol4/Issue1/58.pdf, 25.9.2022.

[25] „*zlib* Manual“, s Interneta, https://zlib.net/manual.html, 25.9.2022.

[26] „LZMA“, s Interneta, https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm, 28.9.2022.

[27] „Range encoders“, s Interneta, https://alecdee.github.io/rangecomp/index.html, 28.9.2022.

[28] „LZMA Compression, s Interneta, https://gautiersblog.blogspot.com/2016/08/lzma-compression.html, 28.9.2022.

[29] „LZMA“, s Interneta, http://mattmahoney.net/dc/dce.html#Section_52, 28.9.2022.

SAŽETAK

U ovom radu opisani su i uspoređeni odabrani algoritmi za kompresiju. Algoritmi su podijeljeni u tri kategorije: entropijski algoritmi, algoritmi temeljeni na metodama rječnika i napredni algoritmi. Prvu kategoriju predstavljaju Huffmanovo kodiranje i Shannon-Fanov algoritam. Sljedeću kategoriju predstavljaju LZW i LZSS algoritmi. Posljednju kategoriju predstavljaju DEFLATE i LZMA algoritmi. Svi algoritmi implementirani su u dva vrlo popularna programska jezika – C++-u i Pythonu. Uspoređivani su omjeri kompresije koje algoritmi ostvaruju na odabranim ulaznim tekstovima. Osim toga, uspoređivana su i vremena izvršavanja algoritama s obzirom na jezik implementacije. Rezultati su prikazani tablično i grafički. Na kraju su navedeni zaključci koji su izvedeni iz rezultata.

Ključne riječi: kompresija teksta, entropijski algoritmi, algoritmi temeljeni na metodama rječnika, napredni algoritmi, Huffmanovo kodiranje, Shannon-Fanov algoritam, LZW, LZSS, DEFLATE, LZMA, C++, Python

ABSTRACT

In this final paper, selected compression algorithms are described and compared. Algorithms are divided into three categories: entropy algorithms, algorithms based on dictionary methods, and advanced algorithms. The first category is represented by Huffman coding and Shannon-Fano algorithm. The next category is represented by LZW and LZSS algorithms. The last category is represented by DEFLATE and LZMA algorithms. All algorithms are implemented in two very popular programming languages – C++ and Python. The compression ratios achieved by the algorithms on the selected input texts were compared. In addition, the execution times of the algorithms, with respect to the programming language, were compared. The results are presented tabularly and graphically. At the end, the conclusions based on the results are stated.

Keywords: text compression, entropy algorithms, algorithms based on dictionary methods, advanced algorithms, Huffman coding, Shannon-Fan algorithm, LZW, LZSS, DEFLATE, LZMA, C++, Python

POPIS SLIKA

POPIS TABLICA

POPIS OZNAKA I KRATICA

VS Code	Visual Studio Code
FLAC	Free Lossless Audio Codec
PNG	Portable Network Graphics
JPEG	Joint Photographic Experts Group
MP3	MPEG-1/2 Audio Layer III
IDE	Integrated Development Environment
MIT	Massachusetts Institute of Technology
GIF	Graphics Interchange Format
ASCII	American Standard Code for Information Interchange
LZW	Lempel–Ziv–Welch
LZSS	Lempel–Ziv–Storer–Szymanski
LZMA	Lempel–Ziv–Markov chain algorithm
RLE	Run-length encoding