Kompresija slike uz pretvaranje u niz korištenjem Base64 algoritma i Hilbertove krivulje

Livojević, Alen

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Diplomski studij računarstva

Diplomski rad

Kompresija slike uz pretvaranje u niz korištenjem Base64 algoritma i Hilbertove krivulje

Rijeka, rujan 2024.

Alen Livojević
0069085057

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Diplomski studij računarstva

Diplomski rad

Kompresija slike uz pretvaranje u niz korištenjem Base64 algoritma i Hilbertove krivulje

Mentor: prof. dr. sc. Jonatan Lerga

Rijeka, rujan 2024.

Alen Livojević
0069085057

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA DIPLOMSKE ISPITE

Rijeka, 12.03.2024.

Zavod: Zavod za računarstvo

Predmet: Kodiranje i kriptografija

ZADATAK ZA DIPLOMSKI RAD

Pristupnik: **Alen Livojević (0069085057)**

Studij: Sveučilišni diplomski studij računarstva (1400)

Modul: Programsko inženjerstvo (1441)

Zadatak: **Kompresija slike uz pretvaranje u niz korištenjem Base64 algoritma i Hilbertove krivulje / Image Compression with Transformation to Array Using Base64 Algorithm and Hilbert Curve**

Opis zadatka:

Potrebno je razviti i testirati metode za kompresiju slike uz pretvaranje iste u niz korištenjem Base64 algoritma i algoritma Hilbertove krivulje. Potrebno je usporediti učinkovitost kompresije temeljene na RLE i LZW algoritmima s i bez predobrade korištenjem Burrows-Wheeler transformacije.

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

Zadatak uručen pristupniku: 20.03.2024.

Mentor:
prof. dr. sc. Jonatan Lerga

Predsjednik povjerenstva za diplomski ispit:
prof. dr. sc. Miroslav Joler

Izjava o samostalnoj izradi rada

Izjavljujem da sam samostalno izradio ovaj rad.

Rijeka, rujan 2024.

_____
Alen Livojević

Zahvala

Zahvaljujem roditeljima, obitelji i prijateljima na podršci tijekom studija i pisanja ovoga rada. Hvala Sari na korisnim savjetima i motivaciji. Zahvaljujem profesorima, asistentima i kolegama na pruženoj pomoći tijekom studiranja. Hvala mentoru prof. dr. sc. Jonatanu Lergi na podršci i savjetima tijekom pisanja ovog rada.

Sadržaj

Popis slika **viii**

Popis tablica **ix**

1 Uvod **1**

2 Metodologija **3**

2.1 Hilbertova krivulja 4

2.2 Base64 format 5

2.3 BWT algoritam 6

2.4 RLE algoritam 6

2.5 LZW algoritam 7

2.6 Vraćanje u izvorno stanje 8

3 Implementacija **9**

3.1 Primjena Hilbertove krivulje za pretvorbu slike u tekst i teksta u sliku 9

3.1.1 Skripta za pretvaranje slike u tekst 9

3.1.2 Skripta za pretvaranje teksta u sliku 11

3.2 Pretvaranje slike u tekst i teksta u sliku pomoću Base64 formata . . . 12

3.2.1 Kod za pretvaranje slike u tekst 12

3.2.2 Kod za pretvaranje teksta u sliku 13

3.3 Implementacija algoritama za kompresiju teksta 14

3.3.1 Implementacija BWT algoritma 15

3.3.2 Implementacija RLE algoritma 17

3.3.3 Implementacija LZW algoritma 20

4 Studija slučaja **22**
4.1 Testne fotografije 23
4.2 Testni slučajevi 26

5 Rezultati **28**
5.1 Prvi testni slučaj 28
5.2 Drugi testni slučaj 32
5.3 Treći testni slučaj 36
5.4 Četvrti testni slučaj 39

6 Zaključak **43**

Literatura **45**

Pojmovnik **48**

Sažetak **49**

A Izvorni kod **51**

Popis slika

2.1 Tijek izvođenja kompresije i dekompresije. 3
2.2 Redovi Hilbertove krivulje. 5

4.1 Testna fotografija. 24
4.2 Prva izrezana fotografija. 24
4.3 Druga izrezana fotografija. 25
4.4 Testni slučajevi. 27

5.1 Usporedba veličina datoteka prije i nakon kompresije u prvom testnom slučaju. 30
5.2 Usporedba omjera kompresije u prvom testnom slučaju. 31
5.3 Usporedba vremena kompresije u prvom testnom slučaju. 32
5.4 Usporedba veličina datoteka prije i nakon kompresije u drugom testnom slučaju. 34
5.5 Usporedba omjera kompresije u drugom testnom slučaju. 35
5.6 Usporedba vremena kompresije u drugom testnom slučaju. 35
5.7 Usporedba veličina datoteka prije i nakon kompresije u trećem testnom slučaju. 37
5.8 Usporedba omjera kompresije u trećem testnom slučaju. 38
5.9 Usporedba vremena kompresije u trećem testnom slučaju. 39
5.10 Usporedba veličina datoteka prije i nakon kompresije u četvrtom testnom slučaju. 40
5.11 Usporedba omjera kompresije u četvrtom testnom slučaju. 41
5.12 Usporedba vremena kompresije u četvrtom testnom slučaju. 42

Popis tablica

2.1 Leksikografski sortirane rotacije riječi "banana". 6

5.1 Rezultati prvog testnog slučaja. 29
5.2 Rezultati drugog testnog slučaja. 33
5.3 Rezultati trećeg testnog slučaja. 36
5.4 Rezultati četvrtog testnog slučaja. 40

Popis isječaka koda

3.1 Skripta za pretvorbu slike u tekst uz pomoć Hilbertove krivulje. . . . 10
3.2 Skripta za pretvorbu teskta u sliku Hilbertovom krivuljom. 11
3.3 Skripta za pretvorbu teksta u Base64 format. 13
3.4 Skripta za pretvorbu Base64 formata u tekst. 14
3.5 Metoda za čitanje, poziv algoritma za kompresiju i zapis sadržaja. . . 14
3.6 Metoda za kodiranje teksta BWT algoritmom. 15
3.7 Metoda za dekodiranje teksta BWT algoritmom. 16
3.8 Metoda za kodiranje teksta RLE algoritmom. 17
3.9 Metoda za dekodiranje teksta RLE algoritmom. 19
3.10 Metoda za kodiranje teksta LZW algoritmom. 20
3.11 Metoda za stvaranje obrnutog riječnika. 21
3.12 Metoda za dekodiranje teksta LZW algoritmom. 21

Poglavlje 1

Uvod

Računalna znanost kroz povijest je pružila različite metode za smanjenje veličine slika. Neka rješenja omogućuju vraćanje slike u izvorno stanje bez ikakvog gubitka, dok se druga nikada ne mogu u potpunosti rekonstruirati. Kompresija s gubicima prihvatljiva je za glasovne, slikovne i video podatke jer potpuno vraćanje nije potrebno ako je rekonstruirana kvaliteta dovoljno dobra za ljudsku percepciju [1]–[3]. Algoritmi za kompresiju podataka bez gubitaka uključuju aritmetičko kodiranje (AC) [4], Lempel-Ziv (LZ) kodove [5], Huffmanove kodove [3], [6], [7] i mnoge druge [2], [8]–[10]. Umjesto oslanjanja na jedan algoritam, u ovom radu koristi se niz algoritama koji su organizirani u smislenom redoslijedu kako bi se maksimizirala njihova učinkovitost. Proces započinje pretvaranjem izvorne slike u niz znakova za što se koristi Base64 format ili Hilbertova krivulja. Zatim se dobiveni niz znakova reorganizira s pomoću algoritma Burrows-Wheeler Transform (BWT). Dobiveni rezultat se potom podvrgava algoritmu Run-Length Encoding (RLE), a ishod se kodira korištenjem algoritma Lempel-Ziv-Welch (LZW). Tijekom izvođenja algoritama mjeri se potrebno vrijeme i postignuti stupanj kompresije. Testiranje je organizirano u četiri različita testna slučaja, a svrhu testiranja algoritma korištene su dvije različite slike. Ishod testiranja prikazan je uz pomoć tablica i grafova. Sljedeća poglavlja pružaju detaljne opise spomenutih algoritama i prikazuju dobivene rezultate.

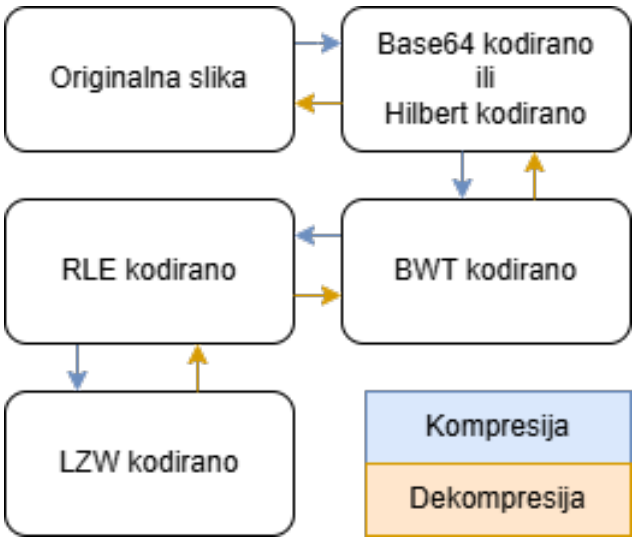
Fokus ovog rada je pokušaj sažimanja slike s pomoću algoritama za kompresiju teksta [11]. Prvi korak prema ostvarivanju cilja predstavlja pretvorba slike u tekstualni oblik. Za proces pretvorbe slike u tekst odabrana su dva načina: Base64 format i obilazak slike pomoću Hilbertove krivulje. Pretpostavka je da će Base64

format biti uspješan zato što njegova duljina ovisi o broju detalja na slici, odnosno o količini informacije koju ta slika sadrži. To znači da Base64 format sam po sebi smanji veličinu sadržaja, a zatim algoritmi za kompresiju teksta ga pokušaju dodatno sažeti. S druge strane, duljina teksta generirana Hilbertovom krivuljom ovisi isključivo o dimenzijama slike, a ne o njenom sadržaju. Međutim, prednost teksta generiranog Hilbertovom krivuljom je u tome što su znakovi smisleno poredani u ovisnosti o boji ploha koje se nalaze na fotografiji. Stoga je takav format zapisa prilagođeniji danjoj kompresiji algoritmima za kompresiju teksta jer je moguća pojava ponavljajućih nizova.

Poglavlje 2

Metodologija

Izvorni format slike na kojoj će biti izvršen cijeli postupak pretvorbe je .tiff. Slika se pretvara u tekst koji se zatim predaje algoritmima za kompresiju teksta na danju obradu. Svaka dobivena transformacija slike bilježi se u odgovarajuću .txt datoteku prilikom pokretanja algoritma. To omogućuje naknadni uvid u stanje u svakom koraku, omogućujući prepoznavanje potencijalnih problema koji se mogu pojaviti tijekom kompresije. Slika 2.1 ilustrira tok izvođenja algoritama kompresije s plavim i dekompresije s narančastim strelicama, pri čemu svako stanje na grafu ima odgovarajuću datoteku u kojoj je dobiveni sadržaj dokumentiran.



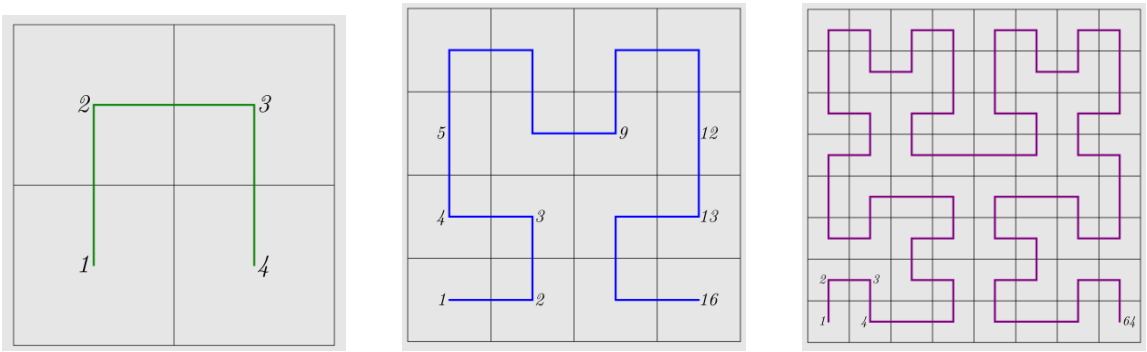
Slika 2.1 Tijek izvođenja kompresije i dekompresije.

2.1 Hilbertova krivulja

Hilbertova krivulja je krivulja koja popunjava (obilazi) dvodimenzionalni prostor. Razvio ju je njemački matematičar David Hilbert 1891. godine. Hilbert je pokazao da se podkvadrati mogu rasporediti tako da susjedni podintervali odgovaraju susjednim podkvadratima koji dijele ivicu, te da su odnosi uključivanja očuvani, tj. ako kvadrat odgovara nekom intervalu, onda njegovi podkvadrati odgovaraju podintervalima tog intervala [12].

U ovom radu Hilbertova krivulja korištena je za pretvorbu slike u tekst. S obzirom na to da se slika sastoji od 3 dimenzije (RGB), Hilbertova krivulja obilazi svaki piksel u svakoj dimenziji. To znači da Hilbertova krivulja obiđe dvodimenzionalnu matricu tri puta. Vizualizacija šest redova Hilbertove krivulje prikazana je na Slici 2.2. Prilikom obilaska, algoritam uzima trenutnu 8-bitnu vrijednost (0-255) i pretvara ju u ASCII vrijednost. Taj postupak rezultira pretvorbom cjelobrojne vrijednosti u znak koji se zapisuje u .txt datoteku koja predstavlja tekstualnu reprezentaciju slike.

Prilikom pretvorbe slike u tekst, a kasnije i vraćanja teksta u sliku, koristi se Hilbertova matrica. To je matrica koja sadrži podatke o redoslijedu obilaska dvodimenzionalne matrice. Ona se generira jednom na početku procesa, prije obilaska prve dimenzije slike. U njoj se nalaze redni brojevi koji označuju koji piksel se posjećuje idući. Istim redoslijedom se obilaze sve tri dimenzije slike, a prilikom pretvorbe teksta u sliku Hilbertova matrica se koristi za pravilnu rekonstrukciju slike.



(a) Hilbertova krivulja prvog reda

(b) Hilbertova krivulja drugog reda

(c) Hilbertova krivulja trećeg reda

(d) Hilbertova krivulja četvrtog reda

(e) Hilbertova krivulja petog reda

(f) Hilbertova krivulja šestog reda

Slika 2.2 Redovi Hilbertove krivulje.

2.2 Base64 format

Base64 je binarno kodiranje koje se obično koristi za prijenos podataka putem tekstualnih kanala kao što su HTML ili e-pošta. Ovaj format osigurava da se binarni podaci, kao što su slike ili druge binarne datoteke, mogu predstaviti s pomoću sigurnog skupa znakova i lako prenijeti i interpretirati u tekstualnom obliku. Base64 koristi 64 znaka (A-Z, a-z, 0-9, + i /) za predstavljanje 6-bitnih blokova izvornih podataka. Svake tri binarne znamenke (24 bita) pretvaraju se u četiri Base64 znaka. Važno je napomenuti da Base64 povećava veličinu podataka za približno 33 %. Format Base64 često se koristi u web razvoju za ugrađivanje slika u HTML dokumente, u privitke e-pošte i u razne web API-je.

2.3 BWT algoritam

Burrows-Wheelerova transformacija (BWT) je algoritam za pretprocesiranje teksta koji reorganizira ulazni tekst kako bi se iskoristile prednosti ponavljajućih nizova i postigla veća kompresija. BWT izvodi permutaciju ovisnu o kontekstu svih slova ulaznih podataka [13]. Osnovna ideja BWT-a je reorganizirati tekst tako da se slični nizovi grupiraju [14]. To se postiže rotiranjem svih sufiksa izvornog teksta, a zatim i leksikografskim sortiranjem dobivenih rotacija. Nakon sortiranja, posljednji stupac matrice rotacije predstavlja transformirani tekst. Pretpostavka je da BWT algoritam može učinkovito pripremiti podatke za RLE kompresiju [15].

Na primjer, leksikografski sortirane rotacije riječi "banana" su prikazane u Tablici 2.1.

Tablica 2.1 Leksikografski sortirane rotacije riječi "banana".

abanan
anaban
ananab
banana
nabana
nanaba

Konačni oblik riječi nastane spajanjem zadnjih slova svake rotacije, odnosno posljednjeg stupca matrice rotacija. U slučaju riječi "banana" transformirani oblik jest "nnbaaa". Na taj način postignuto je grupiranje istih slova u riječi što je pogodan oblik za danju kompresiju RLE algoritmom.

2.4 RLE algoritam

Run-Length Encoding (RLE) je jednostavan, ali učinkovit algoritam za kompresiju podataka. Temeljna ideja RLE je zamijeniti niz uzastopnih identičnih znakova s jednim znakom i brojem ponavljanja tog znaka. Na primjer, niz "AAAABBBCCDAA" može se komprimirati u "4A3B2C1D2A".

Ovaj se algoritam pokazao osobito učinkovitim kada se primijeni na podatke koji sadrže dulje nizove znakova koji se ponavljaju, kao što su slike ili tekstualni

dokumenti s ponavljajućim uzorcima. RLE može značajno smanjiti veličinu takvih podataka.

Implementacija RLE-a uključuje obilazak niza podataka, identificiranje uzastopnih identičnih znakova i zamjenu tih nizova znakom i brojem ponavljanja. Iako je RLE učinkovit u određenim scenarijima [15], ne jamči uvijek optimalnu kompresiju, posebno na podacima kojima nedostaju jasni obrasci koji se ponavljaju. U kombinaciji s drugim algoritmima kompresije, RLE može pridonijeti postizanju boljih ukupnih rezultata kompresije.

2.5 LZW algoritam

LZW algoritam za kompresiju podataka bez gubitaka koristi se za tekstualne i slikovne podatke [5], [16], [17]. LZW kompresija učinkovito smanjuje veličinu datoteka koje sadrže podatke koji se ponavljaju [3], [5].

Cui [18] predstavio je novi LZW algoritam kompresije koji istovremeno povećava propusnost i poboljšava omjer kompresije. Horspool et. al [19] implementirao je dva načina poboljšanja LZC-a u UNIX naredbi za sažimanje. LZW kompresija slika može se poboljšati rezanjem slika u sivim tonovima na osam binarnih (monokromatskih) slika s pomoću rezanja u bitnoj ravnini kroz boje ili semantičkog odvajanja [20], [21]. Rezanje bit-ravnine korisno je za analizu relativne važnosti svakog bita slike [3], [22].

Stvoren je rječnik koji sadrži početne znakove i njihove odgovarajuće kodove. Rječnik u početku sadrži sve pojedinačne znakove (ASCII vrijednosti od 0 do 255, za 8-bitne kodove) kao ključeve i njihove odgovarajuće kodove. LZW algoritam također radi s 12-bitnim kodovima [5].

Kompresija se vrši metodom *compress*, koja prolazi kroz uneseni nekomprimirani tekst znak po znak [23]. Stvara se varijabla koja predstavlja trenutni niz. Zatim algoritam provjerava postoji li sljedeći niz koji sadrži trenutni niz i sljedeći znak. Sljedeći znak dodaje se trenutnom nizu ako ovaj niz već postoji u rječniku. U suprotnom, kôd za trenutni niz dodaje se komprimiranom tekstu, a rječnik se ažurira dodavanjem novog niza i dodjeljivanjem novog koda. Proces se nastavlja dok se ne pročita sav nekomprimirani tekst.

Dekompresija se izvodi metodom *decompress* koja uzima komprimirani tekst kao

ulaz i izvodi obrnuti proces. Komprimirani tekst se rastavlja na pojedinačne kodove i stvara se obrnuti rječnik koji preslikava kodove na odgovarajuće nizove znakova. Početni niz se dobiva korištenjem prvog koda, a preostali se kodovi računaju. Odgovarajući niz dodaje se kada se kôd pronađe u obrnutom rječniku. Ako kôd odgovara slučaju kada je novi unos dodan u rječnik, novi niz se stvara s pomoću prethodnog niza i prvog znaka iz tog niza.

Ova implementacija LZW algoritma omogućuje kompresiju i dekompresiju teksta korištenjem rječnika koji se dinamički širi tijekom procesa [24]. To rezultira smanjenjem veličine datoteke prilikom sažimanja i vraćanjem izvornog teksta prilikom dekompresije.

2.6 Vraćanje u izvorno stanje

Kompresija završava LZW algoritmom, a takav oblik se smatra potpuno komprimiranom slikom. Svi algoritmi korišteni tijekom kompresije omogućuju obrnuti proces dekodiranja bez ikakvih gubitaka. To znači da se slika može prenijeti ili pohraniti u kodiranom obliku i kasnije vratiti u izvorno stanje bez ugrožavanja njezine kvalitete. Tijekom izvođenja ovog procesa koriste se isti algoritmi, ali obrnutim redoslijedom, kao što je prikazano na Slici 2.1 narančastim strelicama.

Poglavlje 3

Implementacija

3.1 Primjena Hilbertove krivulje za pretvorbu slike u tekst i teksta u sliku

Za pretvaranje slike u tekst i vraćanje teksta u sliku korišten je programski jezik Python. Python je ranih 1990-ih kreirao Guido van Rossum u Stichting Mathematisch Centrum (CWI) u Nizozemskoj kao nasljednika jezika ABC. Sva izdanja Pythona su otvorenog koda.[25]

Za rad sa slikom korištena je Python knjižnica OpenCV. OpenCV je u Intelu 1999. pokrenuo Gary Bradsky, a prvo izdanje izašlo je 2000. OpenCV podržava mnoštvo algoritama povezanih s računalnim vidom i strojnim učenjem i širi se iz dana u dan. Podržava i veliki izbor programskih jezika kao što su C++, Python, Java itd., a dostupan je na različitim platformama uključujući Windows, Linux, OS X, Android i iOS. [26]

3.1.1 Skripta za pretvaranje slike u tekst

Isječak koda 3.1 prikazuje implementaciju skripte za pretvorbu slike u tekst pomoću Hilbertove krivulje. Na početku koda, uvezu se potrebne knjižnice i funkcije. Biblioteka *NumPy* služi za rad s numeričkim podacima i omogućava manipulaciju matricama, dok se iz modula *HilbertCurve* uvozi funkcija *hilbert* koja generira Hilbertovu krivulju. Iz modula *ImageImport* uvozi se funkcija *importImage* koja služi za učitavanje slike. Nakon što se slika učita na liniji 5, kreira se Hilbertova krivulja

dimenzija koje odgovaraju veličini slike. Generirana Hilbertova matrica sprema se u tekstualnu datoteku hilbert_matrix.txt radi kasnijeg postupka regeneriranja slike prilikom dekompresije.

Glavni dio koda implementira iteraciju kroz svaku dimenziju slike (crvena, zelena i plava) i svakog piksela redoslijedom određenim Hilbertovom krivuljom. Ugniježđena petlja prolazi kroz svaki piksel slike tako što traži trenutni indeks pomoću Hilbertove matrice i pretvara RGB vrijednost tog piksela u odgovarajući znak koristeći *chr* funkciju. Ovaj proces pretvara numeričke vrijednosti piksela u rasponu od 0 do 255 u ASCII znakove koji predstavljaju cjelokupnu sliku u linearnom obliku. Na kraju, dobiveni niz znakove se sprema u tekstualnu datoteku *converted_string_image.txt*.

```
from HilbertCurve import hilbert
from ImageImport import importImage
import numpy as np

slika = importImage()
hilbert_curve = hilbert(len(slika))

np.savetxt('hilbert_matrix.txt', hilbert_curve)

result_string = ""

for i in range(3):
  for j in range(len(slika) * len(slika)):
    hilbert_index = np.where(hilbert_curve == j)
    red = hilbert_index[0][0]
    stupac = hilbert_index[1][0]
    result_string += chr(slika[red][stupac][i])

with open('converted_string_image.txt', 'w', encoding='utf-8')
   as f:
  f.write(result_string)
```

Isječak koda 3.1 Skripta za pretvorbu slike u tekst uz pomoć Hilbertove krivulje.

3.1.2 Skripta za pretvaranje teksta u sliku

Na početku isječka koda 3.2 uvozi se biblioteka *NumPy* dok se iz modula *ImageImport* uvoze funkcije *importImage* i *showImage*. Funkcija *showImage* koristi se za prikaz rekonstruirane slike na kraju postupka. Prvi korak u rekonstrukciji čini učitavanje Hilbertove krivulje koja je prethodno sačuvana u datoteci hilbert_matrix.txt s pomoću funkcije *np.loadtxt*. Sljedeći dio koda otvara datoteku RLEDecodedFile.txt koja sadrži rezultat dekodiranja prethodno komprimirane slike i čita njen sadržaj. Svaki znak iz datoteke pretvara se u odgovarajuću numeričku vrijednost uz pomoć *ord()* funkcije koja vraća ASCII vrijednost predanog znaka. Zatim *for* petlja prolazi kroz linearne podatke slike. Za svaku vrijednost određuje se red i stupac tog piksela u originalnoj slici prema poziciji u Hilbertovoj krivulji koristeći funkciju *np.where*. Proces se ponavlja tri puta jer je potrebno popuniti matrice za crvenu, zelenu i plavu boju. Na kraju se slika prikazuje s pomoću funkcije *showImage* zbog provjere odgovara li slika početnoj.

```
1  import numpy as np
2  from ImageImport import importImage, showImage
3
4  ## VRACANJE U ORIGINAL
5  loaded_hilbert = np.loadtxt('hilbert_matrix.txt')
6  loaded_result_numbers_string = []
7
8  with open('RLEDecodedFile.txt', 'r', encoding='utf-8') as f:
9    content = f.read()
10   for char in content:
11     loaded_result_numbers_string.append(ord(char))
12
13 slika = importImage()
14 originalna_slika_string = np.zeros((len(slika), len(slika), 3))
15
16 dimenzija = 2
17 br = 0
18
19 for i in range(3*len(slika) * len(slika)):
20
```

```
21   if(br == len(slika)*len(slika)):
22     dimenzija -= 1
23     br = 0
24
25   hilbert_index = np.where(loaded_hilbert == br)
26   red = hilbert_index[0][0]
27   stupac = hilbert_index[1][0]
28   originalna_slika_string[red][stupac][dimenzija] =
      loaded_result_numbers_string[i]
29
30   br += 1
31
32 originalna_slika_string = originalna_slika_string.astype(np.
     uint8)
33 showImage(originalna_slika_string)
```

Isječak koda 3.2 Skripta za pretvorbu teskta u sliku Hilbertovom krivuljom.

3.2 Pretvaranje slike u tekst i teksta u sliku pomoću Base64 formata

Za pretvaranje slike u Base64 format i vraćanje Base64 u sliku korišten je programski jezik Java. Java je programski jezik i računalna platforma koju je prvi put objavio Sun Microsystems 1995. Pruža pouzdanu platformu na kojoj su izgrađene mnoge usluge i aplikacije. Novi, inovativni proizvodi i digitalne usluge dizajnirane za budućnost i dalje se oslanjaju na Javu [27].

3.2.1 Kod za pretvaranje slike u tekst

Isječak koda 3.3 prikazuje implementaciju funkcije *encode_Base64()* koja implementira logiku za pretvorbu slike u Base64 format. Putem *Path* klase određuje se put do slike koja će se enkodirati. Poziva se metoda getPath() nad *FilePath.SLIKA_POCETNA*. FilePath je enumeracija koja sadrži statičke konstante s putanjama do datoteka koje se koriste u programu. Metoda *Files.readAllBytes* čita sve bajtove slike koja se nalazi na putanji *imagePath*. Dobiveni bajtovi se zatim enkodiraju u Base64 format s

pomoću *Base64.getEncoder().encodeToString(imageBytes)* metode koja vraća enkodiranu verziju slike u *String* formatu. Na kraju se tekst sprema u datoteku s pomoću objekta *writer* koji je instanca *FileWriter* klase.

```
1    public static void encode_Base64() throws IOException {
2        String imageString;
3        Path imagePath = Paths.get(FilePath.SLIKA_POCETNA.
   getPath());
4        byte[] imageBytes = Files.readAllBytes(imagePath);
5        imageString = Base64.getEncoder().encodeToString(
   imageBytes);
6        System.out.println("Duljina base64 stringa je " +
   imageString.length());
7
8        FileWriter writer = new FileWriter(FilePath.BASE64.
   getPath());
9        writer.write(imageString);
10       writer.close();
11       System.out.println("Tekst je uspjesno spremljen u
   datoteku.");
12
13    }
```

Isječak koda 3.3 Skripta za pretvorbu teksta u Base64 format.

3.2.2 Kod za pretvaranje teksta u sliku

Isječak Java koda 3.4 implementira metodu *decode_base64* koja dekodira podatke sačuvane u Base64 formatu. Najprije se iz enumeracije *FilePath* dohvati putanja do datoteke koja sadrži Base64 enkodirane podatke. Zakomentirana je alternativna linija koja se koristi u testnim slučajevima koji ne uključuju pretprocesiranje teksta BWT algoritmom, već se Base64 format dohvaća iz RLEDecoded.txt datoteke. Metoda Files.readAllBytes(path) čita sve bajtove iz Base64 enkodirane datoteke i vraća ih kao niz bajtova. Taj niz bajtova zatim prolazi kroz metodu Base64.getDecoder().decode(), koja dekodira Base64 string. Na kraju slika se sprema uz pomoć metode *Files.write*.

```
1    public static void decode_base64() throws IOException {
2        Path path = Paths.get(FilePath.BWTDecoded.getPath());
3        //Path path = Paths.get(FilePath.RLEDecoded.getPath());
4        byte[] data = Base64.getDecoder().decode(Files.
   readAllBytes(path));
5        Files.write(Paths.get(FilePath.SLIKA_VRACENA.getPath())
   , data);
6        System.out.println("Image saved");
7    }
```

Isječak koda 3.4 Skripta za pretvorbu Base64 formata u tekst.

3.3 Implementacija algoritama za kompresiju teksta

Svi algoritmi za kompresiju i dekompresiju teksta implementirani su u programskom jeziku Java. Metoda *compress*, prikazana na primjeru BWT algoritma u isječku koda 3.5, slična je u implementaciji svakog algoritma. Ona služi za čitanje sadržaja kojeg je potrebno obraditi, pozivu metode koja obavlja samu kompresiju te za zapis komprimiranog sadržaja u novu datoteku.

```
1    public static void compress() throws IOException {
2        Reader reader = new Reader(FilePath.BASE64.getPath());
3        Writer writer = new Writer(FilePath.BWTEncoded.getPath
   ());
4        String buffer;
5        String compressedString;
6        while (!(buffer = reader.readFromFile(bufferSize)).
   isEmpty()) {
7            compressedString = doBWTcompression(buffer);
8            writer.write(compressedString);
9        }
10       reader.close();
11       writer.close();
12   }
```

Isječak koda 3.5 Metoda za čitanje, poziv algoritma za kompresiju i zapis sadržaja.

U *while* petlji podaci se učitavaju i komprimiraju u dijelovima koristeći spremnik (eng. buffer). U svakoj iteraciji metoda *reader.readFromFile(bufferSize)* čita dio podataka iz ulazne datoteke sve dok ne dođe do kraja. Zatim se na učitani niz znakova primjenjuje Burrows-Wheeler transformacija putem funkcije *doBWTcompression(buffer)*. Na kraju se rezultat upisuje u izlaznu datoteku s pomoću *writer.write(compressedString)*. Petlja se izvršava sve dok svi podaci nisu upisani u izlaznu datoteku *BWTEncoded.txt*.

3.3.1 Implementacija BWT algoritma

Cijela logika BWT algoritma nalazi se u metodama *doBWTcompression* i *doBWTdecompression* koje su prikazane u isječcima koda 3.6 i 3.7.

Metoda *doBWTcompression* uzima ulazni string *input* koji je potrebno transformirati. Najprije se određuje duljina stringa s pomoću *input.length()* i inicijalizira se niz stringova *rotations* koji sadrži sve moguće rotacije ulaznog stringa. U prvoj for petlji, metoda generira sve moguće rotacije ulaznog stringa. Rotacije se generiraju s pomoću *substring* metode. Nakon što su rotacije generirane, sortiraju se leksikografski s pomoću *Arrays.sort(rotations)*. Ovaj korak omogućuje grupiranje istih znakova. Nakon sortiranja kreira se *StringBuilder* objekt pod nazivom *sb* koji će se koristiti za izgradnju transformiranog teksta. Druga *for* petlja prolazi kroz svaki string u sortiranom nizu i dodaje posljednji znak iz svake rotacije u *sb* objekt.

```
1     public static String doBWTcompression(String input){
2         int length = input.length();
3         String[] rotations = new String[length];
4 
5         // Generiranje svih rotacija izvornog teksta
6         for (int i = 0; i < length; i++) {
7             rotations[i] = input.substring(i) + input.substring
    (0, i);
8         }
9 
```

```
10          // Sortiranje rotacija leksikografski
11          Arrays.sort(rotations);
12
13          StringBuilder sb = new StringBuilder();
14
15          // Izgradnja komprimiranog teksta iz posljednjeg stupca
        sortiranih rotacija
16          for (String rotation : rotations) {
17              sb.append(rotation.charAt(length - 1));
18          }
19
20          return sb.toString();
21      }
```

Isječak koda 3.6 Metoda za kodiranje teksta BWT algoritmom.

Isječak koda 3.7 prikazuje implementaciju inverzne transformacije BWT algoritmom. Metoda prima dva parametra: string *input* kojeg je potrebno vratiti u prvobitno stanje i string *original_string* koji je bio podvrgnut BWT transformaciji i potreban je za provjeru ispravnosti rekonstrukcije. Ugniježđena *for* petlja stvara polje svih mogućih rekonstruiranih rotacija izvornog teksta na temelju njegove duljine. Posljednji dio metode traži originalni tekst u rekonstruiranom nizu rotacija. To postiže prolaskom kroz sve rotacije i provjerom odgovara li originalnom stringu primljenom kroz argument. Ako je original pronađen metoda ga vraća kao rezultat.

```
1       public static String doBWTdecompression(String input,
        String original_string){
2           int length = input.length();
3
4           String[] rotations = new String[length];
5           Arrays.fill(rotations, "");
6
7           // Punjenje matrice rotacija
8           for (int i = 0; i < length; i++) {
9               for (int j = 0; j < length; j++) {
10                  rotations[j] = input.charAt(j) + rotations[j];
11              }
```

```
12              Arrays.sort(rotations);
13          }
14
15          // Pronalazak izvornog teksta
16          for (String row : rotations) {
17              if (row.equals(original_string)) {
18                  return row;
19              }
20          }
21
22          // Ako izvorni string nije pronaden, vraca se prazna
    vrijednost (ne bi se trebalo dogoditi)
23          return "";
24      }
```

Isječak koda 3.7 Metoda za dekodiranje teksta BWT algoritmom.

3.3.2 Implementacija RLE algoritma

Logika BWT algoritma nalazi se u metodama doBWTcompression i doBWT- decompression koje su prikazane u isječcima koda 3.8 i 3.9.

Metoda *doRLEcompression* prima ulazni string koji sadrži podatke koji će biti komprimirani. Objekt *compressed* koji je instanca klase *StringBuilder* koristi se za građenje komprimiranog rezultata. Varijabla *length* čuva duljinu ulaznog stringa i služi za određivanje broja iteracija. *For* prolazi kroz svaki znak u ulaznom stringu. Ako je trenutni znak znamenka on se označi zvjezdicama (*) kako bi se razlikovao od ostalih podataka. Inače bi ti brojevi mogli biti pogrešno protumačeni kao dio RLE kompresije. Ako znak nije znamenka, obavlja se standardni RLE postupak. Prvo se inicijalizira varijabla za brojanje ponavljanja pojedinog znaka *count* na vrijednost 1 što označava da je trenutni znak viđen bar jednom. Zatim se koristi *while* petlja koja broji uzastopna pojavljivanja istog znaka. Petlja se izvršava sve dok je sljedeći znak u nizu isti kao trenutni (*buffer.charAt(i + 1) == currentChar*). Kada se prebroje sva uzastopna pojavljivanja trenutnog znaka, dodaje se broj ponavljanja i trenutni znak u rezultirajući niz.

```
1    private static String doRLEcompression(String buffer) {
2        StringBuilder compressed = new StringBuilder();
3        int length = buffer.length();
4        for (int i = 0; i < length; i++) {
5            char currentChar = buffer.charAt(i);
6            if (Character.isDigit(currentChar)) {
7                // Ako je trenutni znak znamenka, dodaj znakove
    prije i iza
8                compressed.append("*");
9                compressed.append(currentChar);
10               compressed.append("*");
11           } else {
12               int count = 1;
13               // Brojanje ponajvljanja istog znaka
14               while (i + 1 < length && buffer.charAt(i + 1)
   == currentChar) {
15                   count++;
16                   i++;
17               }
18               compressed.append(count).append(currentChar);
19           }
20       }
21       return compressed.toString();
22   }
```

Isječak koda 3.8 Metoda za kodiranje teksta RLE algoritmom.

Isječak 3.9 Java koda implementira metodu *doRLEdecompression* koja vrši dekompresiju teksta koji je prethodno komprimiran Run-Length Encoding (RLE) algoritmom. Metoda počinje inicijalizacijom objekta *decompressed* klase *StringBuilder* koji služi za građenje rezultata. Varijabla *length* potrebna je za određivanje broja rotacija *while* petlje. *While* petlja prolazi kroz svaki znak u komprimiranom stringu i prvo se vrši provjera je li trenutni znak zvjezdica kako bi se izbjegla pogreška prilikom dekompresije. Ako je trenutni znak zvjezdica, znamenka koju zvjezdice okružuju dodaje se u rezultirajući niz, a iterator i se uvećava za dva kako bi se iduća zvjezdica preskočila te kako bi se prešlo na novi znak. Ako trenutni znak nije zvjezdica, vrši se

standardno RLE dekodiranje. *While* petlja broji koliko se znakova ponavlja do iduće znamenke. Taj broj postaje broj iteracija *for* petlje koja dodaje po jedno trenutno slovo u rezultirajući string i na taj način dekodira predani sadržaj.

```
1    private static String doRLEdecompression(String buffer) {
2        StringBuilder decompressed = new StringBuilder();
3        int length = buffer.length();
4        int i = 0;
5        while (i < length) {
6            if(buffer.charAt(i) == '*'){
7                i++;
8                decompressed.append(buffer.charAt(i));
9                i += 2;
10               continue;
11           }
12
13           StringBuilder countStr = new StringBuilder();
14           // Procitaj broj ponavljanja (moze biti vise od
   jedne znamenke)
15           while (i < length && Character.isDigit(buffer.
   charAt(i))) {
16               countStr.append(buffer.charAt(i));
17               i++;
18           }
19           int count = Integer.parseInt(countStr.toString());
20
21           char currentChar = buffer.charAt(i);
22           // Dodaj znak u rezultantni string 'count' puta
23           for (int j = 0; j < count; j++) {
24               decompressed.append(currentChar);
25           }
26           i++; // Prijedi na iduci znak
27       }
28       return decompressed.toString();
29   }
```

Isječak koda 3.9 Metoda za dekodiranje teksta RLE algoritmom.

3.3.3 Implementacija LZW algoritma

Metode koje implementiraju kompresiju i dekompresiju LZW algoritmom prikazane su u isječcima koda 3.10 i 3.12.

U metodi *doLZWcompression* inicijalizirane su varijable *uncompressed_length*, *current* i *compressed*. Duljina ulaznog stringa sprema se u varijablu *uncompressed_length*. Trenutna sekvenca koja se dekodira spremljena je u varijablu *current*, a u varijabli *compressed* spremljen je komprimirani rezultat. *For* petlja prolazi kroz svaki znak u ulaznom stringu za kojeg se formira sekvenca *next* koja je kombinacija trenutne sekvence i novog znaka. Ako sekvenca *next* postoji u rječniku, trenutna sekvenca *current* poprimi vrijednost sekvence *next*. U slučaju da sekvenca *next* ne postoji u rječniku, u rezultirajući niz se dodaje kod koji odgovara trenutnoj sekvenci *current*, a sekvenca *next* se dodaje u rječnik. Nakon što petlja obradi sve znakove, provjerava se je li *current* sekvenca prazan string. Ako nije, dodaje se preostali kod koji odgovara sekvenci *current* u rezultirajući niz.

```
1    private static String doLZWcompression(String buffer) {
2        uncompressed_length = buffer.length();
3        String current = "";
4        String compressed = "";
5
6        for (char c : buffer.toCharArray()) {
7            String next = current + c;
8            if (dictionary.containsKey(next)) {
9                current = next;
10           } else {
11               compressed += dictionary.get(current) + " ";
12               compressed_length++;
13               dictionary.put(next, dictionarySize++);
14               current = "" + c;
15           }
16       }
17       if (!current.equals("")) {
18           compressed += dictionary.get(current) + " ";
19           compressed_length++;
20       }
```

```
21
22          return compressed;
23      }
```

Isječak koda 3.10 Metoda za kodiranje teksta LZW algoritmom.

LZW algoritam za kompresiju koristi rječnik u kojem se svakom znakovnom nizu dodjeljuje jedinstveni kod (broj). Međutim, za proces dekompresije potreban je "obrnuti" rječnik u kojem su kodovi ključevi, a odgovarajući znakovni nizovi vrijednosti. Isječak koda 3.11 prikazuje implementaciju metode *reverseDictionary* koja prolazi kroz originalni rječnik i zamjenjuje ključ i vrijednost svake stavke.

```
1       public static void reverseDictionary(){
2           for (Map.Entry<String, Integer> entry : dictionary.
    entrySet()) {
3               reverseDictionary.put(entry.getValue(), entry.
    getKey());
4           }
5       }
```

Isječak koda 3.11 Metoda za stvaranje obrnutog riječnika.

Metoda doLZWdecompression prikazana u kodnom isječku 3.12 koristi obrnuti rječnik kako bi provela dekompresiju ulaznog stringa. Argument funkcije je broj koji je zaprimljen u obliku stringa, stoga ga je potrebno pretvoriti u broj s pomoću *Integer.parseInt(input)*. Zatim se ovisno o tom broju dohvaća dekodirana vrijednost iz obrnutog rječnika pomoću *reverseDictionary.get(Integer.parseInt(input))*. Na kraju metoda vraća dohvaćenu vrijednost.

```
1       private static String doLZWdecompression(String input){
2           return reverseDictionary.get(Integer.parseInt(input));
3       }
```

Isječak koda 3.12 Metoda za dekodiranje teksta LZW algoritmom.

Poglavlje 4

Studija slučaja

Nakon što su sve datoteke kreirane, moguće je analizirati učinkovitost algoritma. Algoritam se ocjenjuje na temelju dva kriterija: omjera kompresije i vremena izvršavanja. Vrijeme izvršavanja odgovara vremenu izvršavanja metode *compress* koju sadrži svaki algoritam. Omjer kompresije izračunat je uz pomoć veličine početne slike i veličine sadržaja zapisanog u datoteci LZWEncoded.txt, koja se smatra konačnim oblikom kompresije. Početna veličina slike se računa prema jednadžbi 4.1.

$$VS = (D * S * 3) / 1024 \tag{4.1}$$

U jednadžbi 4.1 VS označava veličinu početne slike dok D označava broj piksela po dužini, a S broj piksela na širini. Taj umnožak je još potrebno pomnožiti tri puta jer se radi o 24-bitnoj RGB slici u kojoj svaki piksel ima tri dimenzije. Svaka dimenzija predstavlja intenzitet crvene, zelene i plave boje. Piksel u svakoj dimenziji može imati vrijednost između 0 i 255 jer je prikazan uz pomoć 8 bitova. Dobiveni umnožak se na kraju podijeli s 1024 kako bi veličina slike bila prikazana u kilobajtima. U izračun veličine slike nisu uzeti metapodaci, kao ni u postupku kompresije. U procesu sudjeluju isključivo podaci potrebni za prikaz slike.

Veličina rezultata LZW algoritma, odnosno rezultata kompresije, mjeri se drugačije jer uključuje numeričke vrijednosti. Stoga se broj cijelih brojeva množi s veličinom jednog cijelog broja, što u ovom slučaju iznosi 4 bajta. Konačni rezultat također se izražava u kilobajtima.

Omjer kompresije računa se prema izrazu 4.2.

$$CR = 100 - \left(\frac{C}{U} * 100\right) \tag{4.2}$$

U jednadžbi 4.2, gdje CR označava omjer kompresije, a C i U veličine sadržaja prije i poslije kompresije.

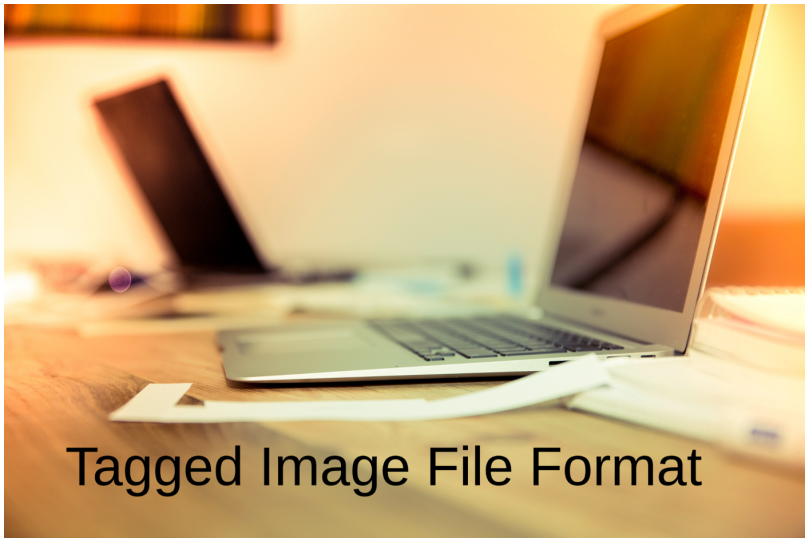
Datoteke se čitaju uz pomoć međuspremnika čija se veličina može odrediti posebno za svaki algoritam. Veličina međuspremnika ima najveći utjecaj pri izvršavanju BWT algoritma, jer se njegova učinkovitost mijenja ovisno o veličini. Veći međuspremnik doprinosi boljim rezultatima kompresije, ali također značajno povećava vrijeme izvođenja dekompresije budući da je složenost BWT algoritma tijekom rekonstrukcije sadržaja O(n^2). Prema dosadašnjem iskustvu odabran je međuspremnik veličine 1000 znakova.

4.1 Testne fotografije

Za testiranje algoritma odabrane su testne fotografije u .tiff formatu. Zbog hardverskih ograničenja, korištene su dvije fotografije u dimenzijama 512 x 512. Svaka od tih fotografija dio je jedne velike fotografije, ali se razlikuju po sadržaju i rasporedu boja kako bi se mogao usporediti učinak algoritma ovisno o uzorku na fotografiji. Fotografija čiji su dijelovi korišteni za testiranje algoritma prikazana je na Slici 4.1.

Iz prethodne fotografije su izrezana dva dijela veličine 512 x 512. Na taj način su nastale dvije različite fotografije koje predstavljaju primjere za fotografiju na kojoj je prisutno više boja i usporedno fotografiju za koju se može reći da je "monotona" u pogledu raspona boja, jer sadrži područja u slici gdje prevladava jedna boja. Fotografija koja sadrži više boja prikazana je na Slici 4.2.

Fotografija koja je "monotonija", a samim time i pogodnija za kompresiju ovim algoritmom prikazana je na Slici 4.3.



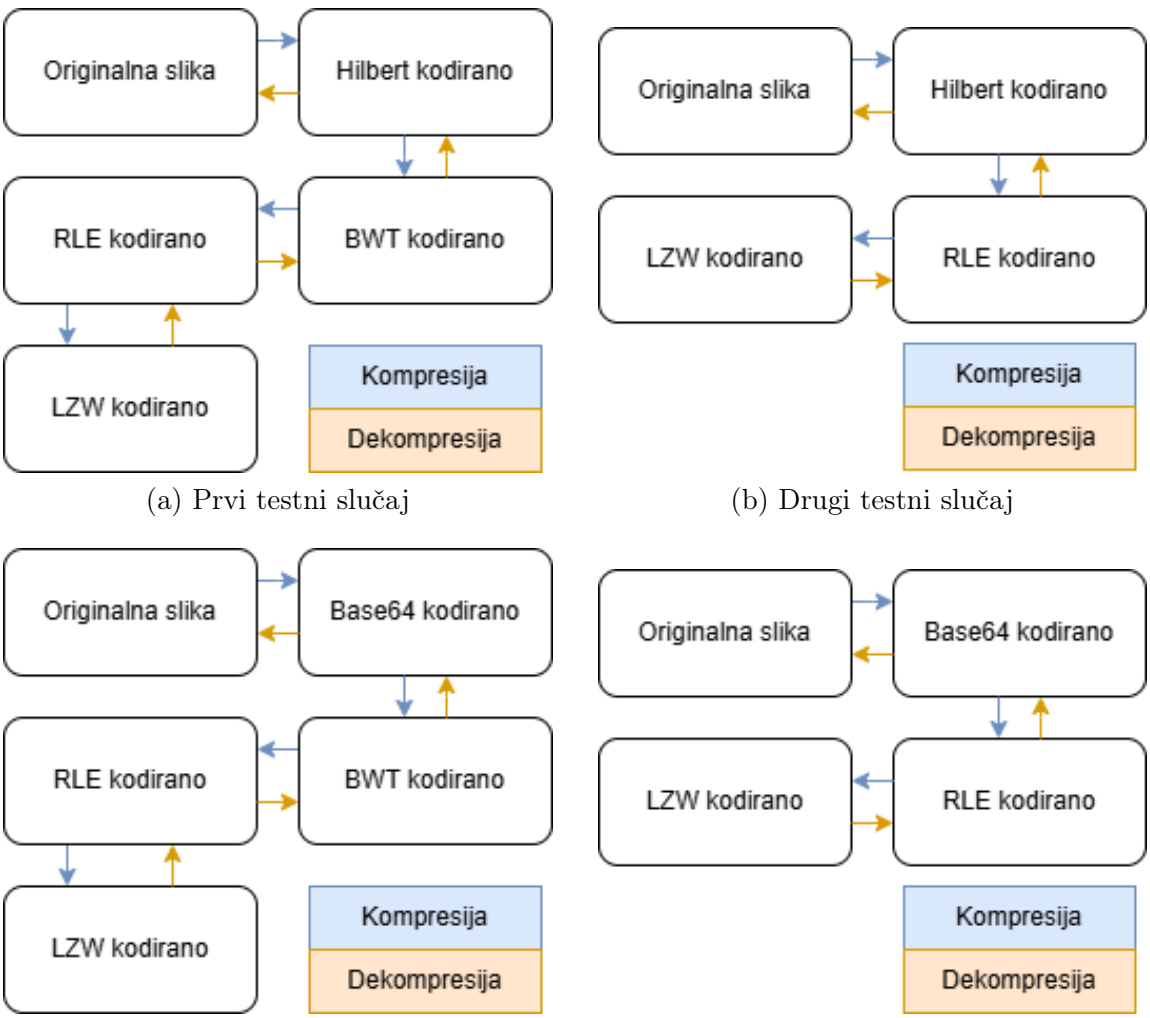
Slika 4.1 Testna fotografija.

Slika 4.2 Prva izrezana fotografija.

Slika 4.3 Druga izrezana fotografija.

4.2 Testni slučajevi

Testiranje algoritma provedeno je u četiri testna slučaja čiji je dijagram prikazan na Slici 4.4. Testni slučajevi kombinacija su načina pretvorbe slike u tekst i uključivanja BWT algoritma za pretprocesiranje teksta. Stoga, prvi testni slučaj odnosi se na slučaj pretvorbe slike u tekst kada je korištena Hilbertova krivulja, a dobiveni tekst je podvrgnut BWT pretprocesiranju. U drugom testnom slučaju pretvorba slike u tekst također je izvršena uz pomoć Hilbertove krivulje, ali dobiveni tekst nije podvrgnut BWT algoritmu. Treći testni slučaj pokriva situaciju u kojoj je slika pretvorena u Base64 tekstualni format, a zatim je dobiveni tekst predan BWT algoritmu na pretprocesiranje. Posljednji, četvrti testni slučaj, pokriva situaciju kada se slika pretvara u tekstualni Base64 format, ali nakon pretvorbe BWT algoritam ne sudjeluje u danjem postupku. Svaki slučaj sadrži i postupak rekonstrukcije slike, odnosno algoritmi se izvršavaju u suprotnom redoslijedu i dekodiraju sadržaj kojeg su prethodno kodirali.



(a) Prvi testni slučaj

(b) Drugi testni slučaj

(c) Treći testni slučaj

(d) Četvrti testni slučaj

Slika 4.4 Testni slučajevi.

Poglavlje 5

Rezultati

U nastavku su prikazani rezultati kompresije testnih slika grupirani prema testnim slučajevima. Rezultati prikazuju performanse u kontekstu uspješnosti kompresije i vremena izvođenja. Za testiranje i prikupljanje rezultata korišteno je prijenosno računalo Lenovo IdeaPad 3 kojeg pokreće AMD Ryzen 2500U procesor i ima 8 GB RAM memorije.

5.1 Prvi testni slučaj

U prvom testnom slučaju vrši se pretvorba slike u tekst s pomoću Hilbertove krivulje, a u kompresiji se koristi BWT algoritam. Tablica 5.1 prikazuje veličine prije i nakon kompresije i omjer tih veličina. Osim toga prikazana su i vremena izvršavanja kompresije i dekompresije kao i vremena izvršavanja pojedinog algoritma.

Slika 5.1 prikazuje razlike u veličinama slika prije i nakon kompresije. Plava boja prikazuje veličine originalne slike prije kompresije, a narančasta veličinu sadržaja LZWEncoded.txt datoteke koja se smatra završnom datotekom u postupku kompresije. Uočeno je da prilikom pokušaja kompresije Slike 4.2 veličina nije smanjena, nego je povećana, što znači da u ovom testnom slučaju kompresija za tu sliku nije postignuta. Uzrok toga leži u činjenici da Slika 4.2 sadrži više detalja, točnije na njoj se nalazi zamućeni prikaz ekrana laptopa koji prekida plohu narančastog zida koji je u pozadini. Takav slučaj ne pogoduje ovom algoritmu. Razlog tome je što prilikom obilaska Hilbertove krivulje po slici iste vrijednosti nisu grupirane zajedno, već crni ekran laptopa isprekida vrijednosti jednolične plohe zida. Taj problem se nastojao

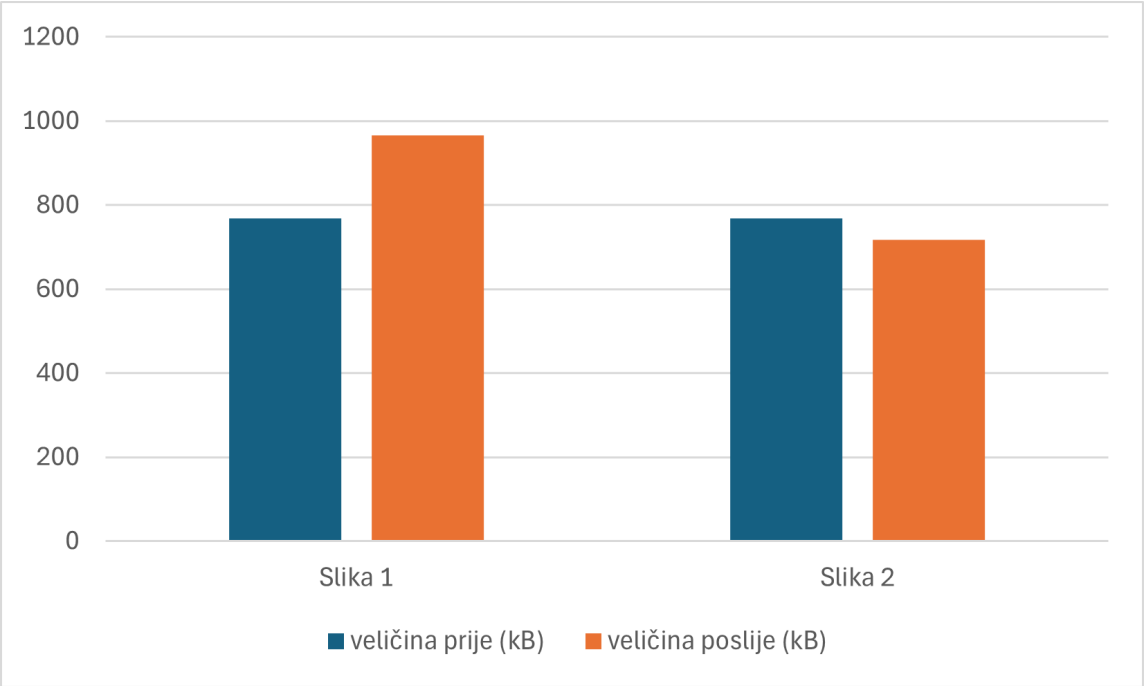
Tablica 5.1 Rezultati prvog testnog slučaja.

	Slika 4.2	**Slika 4.3**
veličina prije (kB)	768	768
veličina poslije (kB)	965,25	716,73
omjer (%)	-25,68%	6,68%
vrijeme kompresije (ms)	578860	586265
vrijeme dekompresije (ms)	759732	811801
Slika u tekst (ms)	577217	584479
BWT kompresija (ms)	1023	1194
RLE kompresija (ms)	78	94
LZW kompresija (ms)	541	490
LZW dekompresija (ms)	509	474
RLE dekompresija (ms)	163	156
BWT dekompresija (ms)	191752	186685
Tekst u sliku (ms)	567308	624486

otkloniti uz pomoć BWT algoritma čija je zadaća namjestiti iste vrijednosti u niz, ali u tome se nije pokazao dovoljno učinkovit. U slučaju Slike 4.3, situacija je drugačija. Slika 4.3 sadrži kontinuiranu plohu narančastog zida u pozadini. Stoga Hilbertova krivulja poveže jednake vrijednosti u niz što savršeno odgovara prirodi ovog postupka u kojem RLE značajno smanji broj znakova koji kodiraju jednake grupirane vrijednosti. Veličina datoteke nakon kompresije Slike 4.3 je manja od početne veličine slike za 51,27 kB što znači da je algoritam uspješno obavio zadatak.

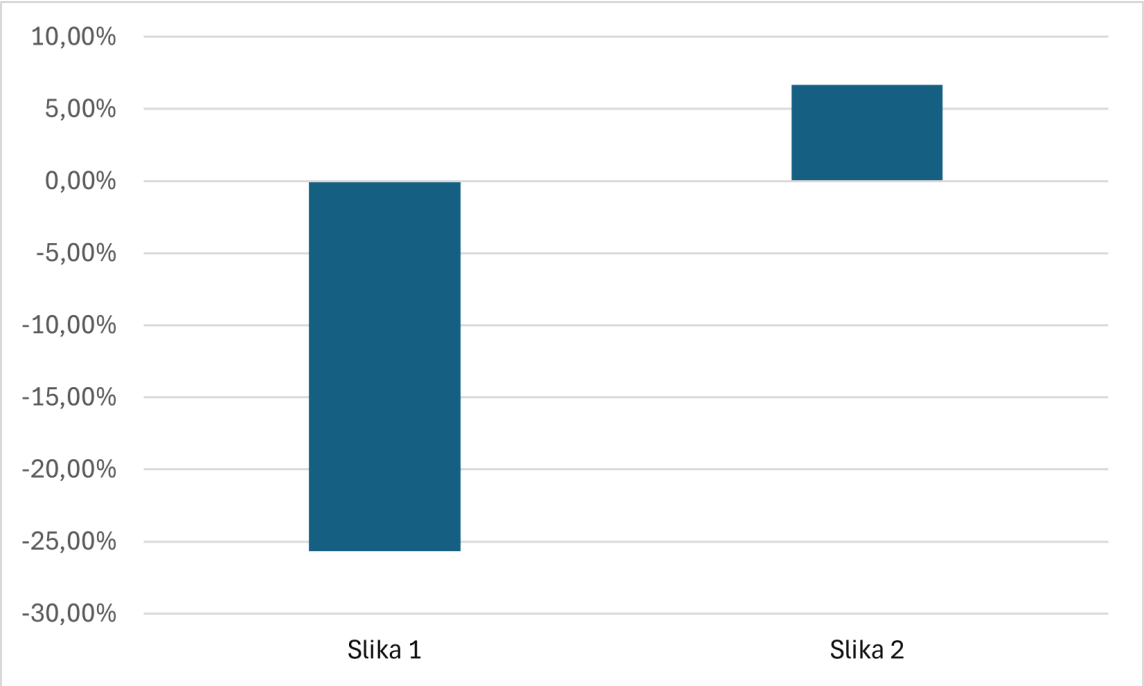
Kako bi se bolje približila uspješnost, odnosno neuspješnost kompresije, Slika 5.2 prikazuje omjere veličina prije i nakon kompresije pojedine slike. Ako je omjer pozitivan onda je kompresija postignuta, a ako je negativan onda je sadržaj LZWEncoded.txt datoteke veći od početne slike. S obzirom na to da algoritam nije uspio smanjiti veličinu Slike 4.2, veličina sadržaja datoteke nakon postupka veća je za 25,68 % u odnosu na početnu sliku što je više od četvrtine veličine početne slike. S druge strane, veličina datoteke nakon kompresije Slike 4.3 manja je za 6,68 % u odnosu na početnu sliku.

Vrijeme potrebno za kompresiju i dekompresiju prikazano je na Slici 5.3. Plavom



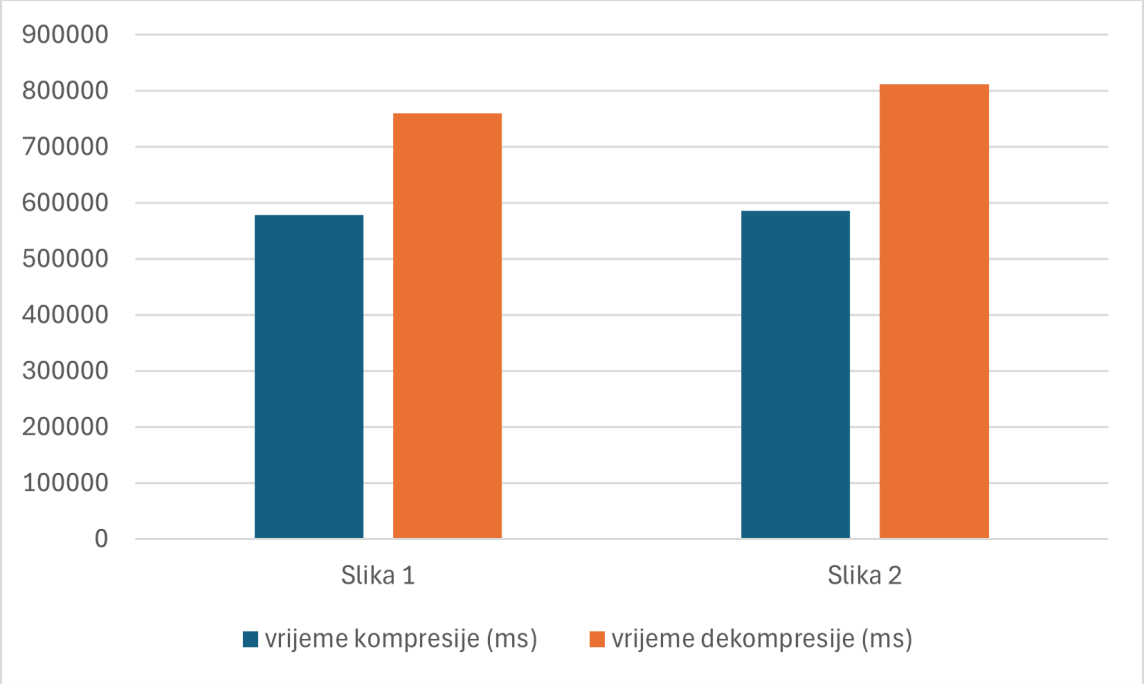
Slika 5.1 Usporedba veličina datoteka prije i nakon kompresije u prvom testnom slučaju.

bojom prikazano je vrijeme kompresije, a narančastom vrijeme dekompresije pojedine slike. Vremena kompresije i dekompresije su slična za obje slike jer sadržaj slike ne utječe izravno na dužinu Hilbertove krivulje jer ona u svakom slučaju mora obići sve piksele. Isto vrijedi i za ostale algoritme. Za obje slike ponavlja se isti postupak. Najveća razlika je u tome što RLE algoritam prilikom kompresije Slike 4.3 jače sažme sadržaj nego prilikom kompresije Slike 4.2. Zato prilikom kompresije Slike 4.3 LZW algoritam može biti brži jer mu je zadaća kodirati kraći sadržaj kojeg je prethodno pripremio RLE algoritam. Odgovarajući rezultati vidljivi su u Tablici 5.1. Također, u Tablici 5.1 vidljivo je da je većina vremena uložena u pretvaranje slike u tekst prilikom kompresije i u pretvaranje teksta u sliku prilikom dekompresije. Na primjeru Slike 4.3 ukupno vrijeme kompresije je 586265 milisekundi, od čega je 584479 milisekundi utrošeno na pretvorbu slike u tekst. Slično je i prilikom dekompresije. Od 811801 milisekundi potrebnih za dekompresiju 624486 milisekundi utrošeno je na pretvorbu teksta u sliku. Osim toga, BWT algoritam također utroši više vremena prilikom dekompresije. U slučaju Slike 4.3, na postupak dekodiranja BWT algorit-



Slika 5.2 Usporedba omjera kompresije u prvom testnom slučaju.

mom utrošeno je 186685 milisekundi što je znatno više od vremena kodiranja BWT algoritmom koje iznosi 1194 milisekundi. LZW i RLW algoritmi se u oba slučaja izvrše za manje od 1000 milisekundi. Ista usporedba primjenjiva je i na podacima za Sliku 4.2.



Slika 5.3 Usporedba vremena kompresije u prvom testnom slučaju.

5.2 Drugi testni slučaj

I u drugom testnom slučaju vrši se pretvorba slike u tekst s pomoću Hilbertove krivulje, ali se u kompresiji ne koristi BWT algoritam. Tablica 5.2 sadrži podatke o veličinama sadržaja prije i nakon kompresije, omjere tih veličina, vremena izvršavanja kompresije i dekompresije te vremena izvršavanja pojedinog algoritma.

Razlike u veličinama slika prije i nakon kompresije prikazane su na Slici 5.4. Plava boja prikazuje veličine originalne slike prije kompresije, a narančasta veličinu sadržaja LZWEncoded.txt datoteke. Slično kao i u prethodnom testnom slučaju, prilikom pokušaja kompresije Slike 4.2 veličina nije smanjena, nego je povećana, što znači da ni ovoga puta kompresija za tu sliku nije postignuta. To se dogodilo zbog istog razloga: Slika 4.2 sadrži više detalja od Slike 4.3 što znači da nakon obilaska Hilbertove krivulje po slici iste vrijednosti nisu grupirane zajedno. U ovom slučaju BWT algoritam nije pokušao riješiti taj problem s obzirom na to da se nije pokazao učinkovitim u prethodnom testnom slučaju. Bez BWT pretprocesiranja kompresija također nije postignuta, ali je rezultat povoljniji. Veličina kompresiranog sadržaja u

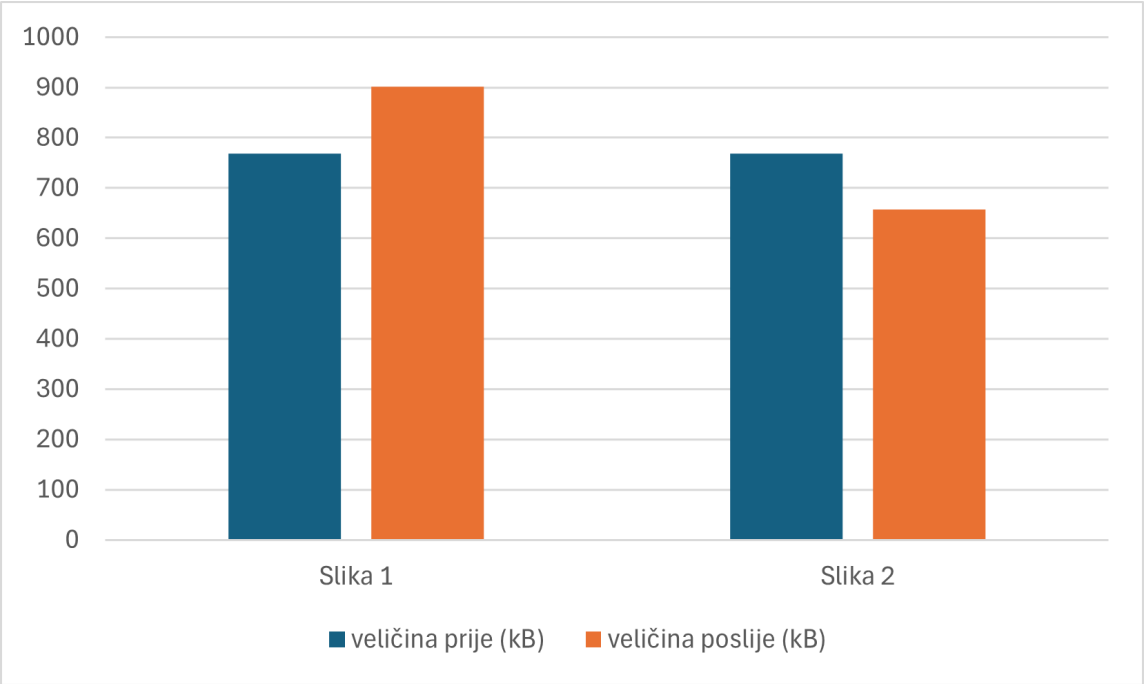
Tablica 5.2 Rezultati drugog testnog slučaja.

	Slika 4.2	**Slika 4.3**
veličina prije (kB)	768	768
veličina poslije (kB)	901,28	657,77
Omjer (%)	-17,35%	14,35%
vrijeme kompresije (ms)	566763	576211
vrijeme dekompresije (ms)	592789	606307
Slika u tekst (ms)	565952	575567
RLE kompresija (ms)	110	131
LZW kompresija (ms)	701	508
LZW dekompresija (ms)	504	475
RLE dekompresija (ms)	137	163
Tekst u sliku (ms)	592148	605669

iznosu od 901,28 kB manja je od veličine kompresiranog sadržaja iz prvog testnog slučaja koja iznosi 965,25 kB. U slučaju Slike 4.3, rezultati su u skladu s rezultatima Slike 4.2. Veličina datoteke nakon kompresije je manja od početne veličine slike za 110,23 kB što znači da je algoritam još uspješnije obavio zadatak nego u prvom testnom slučaju. Iz ovih rezultata može se zaključiti da BWT algoritam nije pridonio uspješnosti kompresije, već je odmogao.

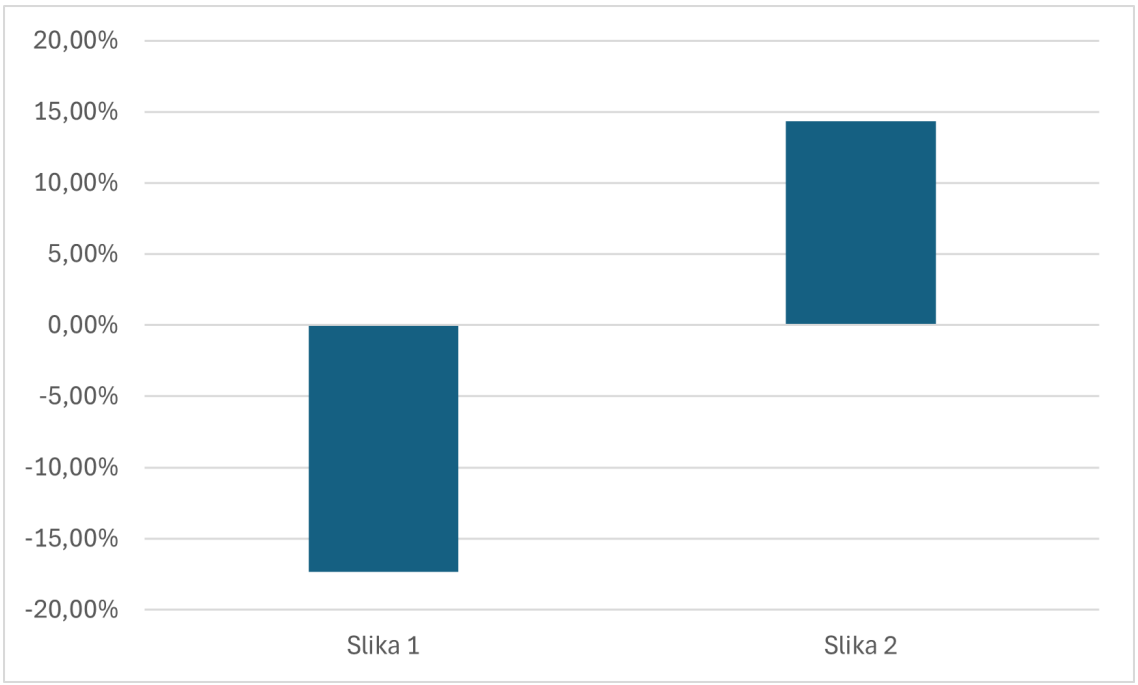
Omjeri veličina prije i nakon kompresije pojedine slike prikazani su na Slici 5.5. Negativan omjer kompresije Slike 4.2 pokazuje da algoritam nije uspio smanjiti veličinu Slike 4.2. Veličina datoteke nakon postupka je veća za 17,35 % u odnosu na početnu sliku što je za 8,33 % bolje nego u prethodnom slučaju. Veličina datoteke nakon kompresije Slike 4.3 manja je za 14,35 % u odnosu na početnu sliku što je bolje nego u prethodnom slučaju za 7,67 %.

Vrijeme potrebno za kompresiju i dekompresiju prikazano je na Slici 5.6. Plavom bojom prikazano je vrijeme kompresije, a narančastom vrijeme dekompresije pojedine slike. Vremena potrebna za kompresiju i dekompresiju obje slike slična su primjeru iz prvog testnog slučaja tako da objašnjenje iz prethodnog slučaja vrijedi i ovdje, osim u slučaju BWT algoritma s obzirom na to da se ovdje ne koristi. Odgovarajući rezultati vidljivi su u Tablici 5.2. Također, u Tablici 5.2 vidljivo je da je i u ovom testnom

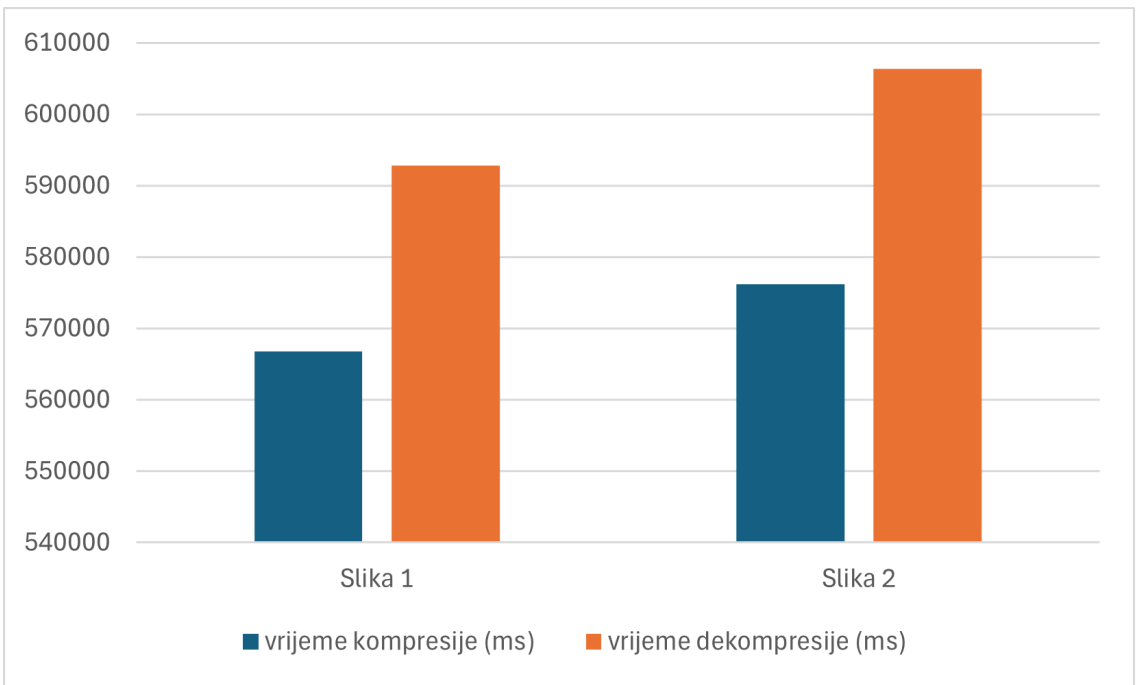


Slika 5.4 Usporedba veličina datoteka prije i nakon kompresije u drugom testnom slučaju.

slučaju većina vremena uložena u pretvaranje slike u tekst prilikom kompresije i u pretvaranje teksta u sliku prilikom dekompresije. Na primjeru Slike 4.3 ukupno vrijeme kompresije je 576211 milisekundi, od čega je 575567 milisekundi utrošeno na pretvorbu slike u tekst. Dekompresija traje malo duže od kompresije: od ukupno 606307 milisekundi potrebnih za dekompresiju 605669 milisekundi utrošeno je na pretvorbu teksta u sliku. LZW i RLW algoritmi su prilično brzi pa se i ovoga puta izvrše za manje od 1000 milisekundi. Ista usporedba primjenjiva je i na podacima za Sliku 4.2.



Slika 5.5 Usporedba omjera kompresije u drugom testnom slučaju.



Slika 5.6 Usporedba vremena kompresije u drugom testnom slučaju.

5.3 Treći testni slučaj

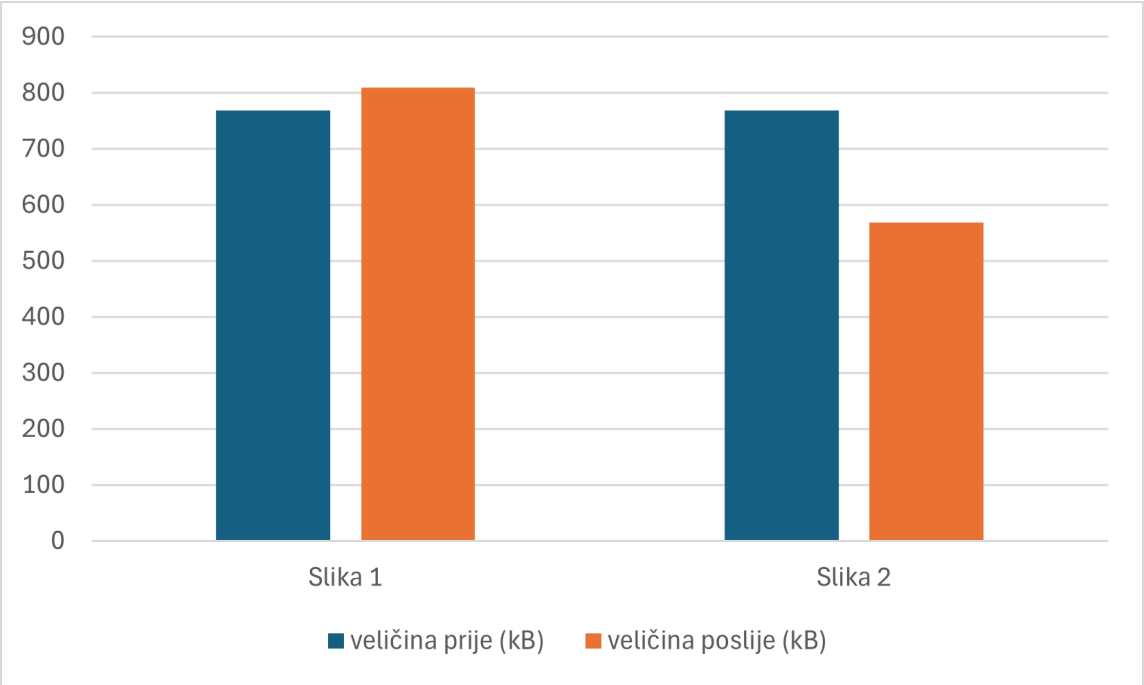
Treći testni slučaj ispituje uspješnost kompresije u slučaju kada se pretvorba slike u tekst obavlja s pomoću Base64 formata, a proces sadrži i BWT algoritam. Rezultati koji uključuju veličine prije i nakon kompresije, omjer veličina i vremena izvršavanja kompresije, dekompresije i pojedinih algoritama, prikazani su u Tablici 5.1.

Tablica 5.3 Rezultati trećeg testnog slučaja.

	Slika 4.2	**Slika 4.3**
veličina prije (kB)	768	768
veličina poslije (kB)	809,35	568,48
Omjer (%)	-5,38%	25,98%
vrijeme kompresije (ms)	2251	1987
vrijeme dekompresije (ms)	196928	135680
Slika u tekst (ms)	62	65
BWT kompresija (ms)	1217	1059
RLE kompresija (ms)	109	122
LZW kompresija (ms)	846	722
LZW dekompresija (ms)	752	605
RLE dekompresija (ms)	184	176
BWT dekompresija (ms)	195920	134836
Tekst u sliku (ms)	71	62

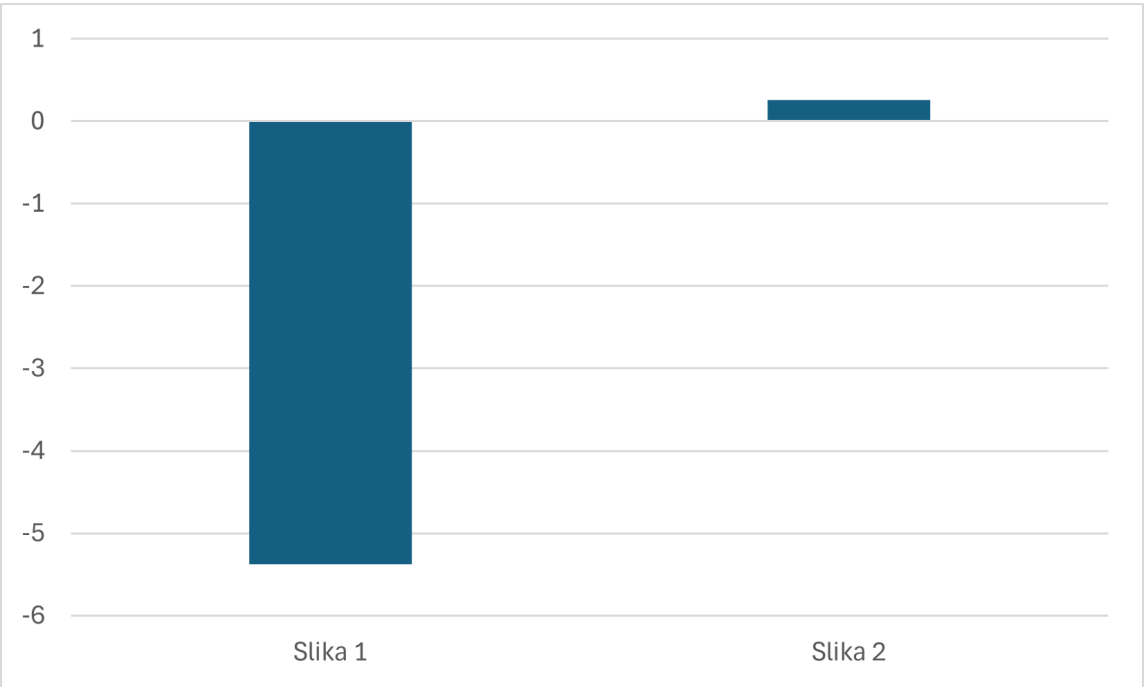
Rezultati trećeg testnog slučaja prate trend poboljšanja performansi iz prethodna dva slučaja što je vidljivo na Slici 5.7. Veličina Slike 4.2 ponovno nije smanjena. Veličina sadržaja datoteke nakon kompresije iznosi 809,35 kB dok je prije kompresije iznosila 768 kB. Base64 format, koji je korišten u ovom slučaju, izgleda drugačije od formata kojeg stvori Hilbertova krivulja. U Base64 formatu se ne mogu pronaći ponavljajući nizovi koji su nastali kao posljedica jednobojnih ploha pozadine. Nedostatak ovakvog formata jest taj što je sada tekstualna reprezentacija slike manje intuitivna i ne sadrži ponavljajuće uzorke. Stoga je pretpostavka da će ovdje BWT algoritam za pretprocesiranje teksta biti od puno veće važnosti jer je upravo on zadužen za stvaranje takvih uzoraka. S druge strane, pozitivna osobina Base64 formata je duljina tekstualne reprezentacije slike koja ovisi o broju detalja na slici. Naime,

veličina Base64 formata povezana je s količinom informacija koje ta slika sadrži. Slike s većim brojem boja i detalja, kao što je Slika 4.2 koja prikazuje detalj zamućenog laptopa ispred jednobojnog zida, zahtijevat će više podataka za opisivanje svakog piksela u usporedbi s jednostavnom, jednobojnom slikom kao što je Slika 4.3 koja sadrži plohu zida. Takvo ponašanje Base64 formata samo po sebi pridonosi veličini sadržaja završne LZWEncoded.txt datoteke i omogućuje bolju kompresiju. Takvo ponašanje Base64 formata vidljivo je i u rezultatima prikazanim u Tablici 5.3 i na Slici 5.7. Slika 4.2 koja sadrži više detalja, a pritom i veću količinu informacija, veća je nakon pokušaja kompresije nego prije. Međutim, razlika između početne veličine od 768 kB i veličine nakon pokušaja kompresije od 809,35, manja je za 20,3 % nego u prvom, odnosno za 11,97 % u drugom testnom slučaju. Slika 4.3, koja sadrži manje detalja i manju količinu informacije, uspješno se komprimirala. Pritom je rezultat bolji za 19,3 % nego u prvom, odnosno za 11,63 % nego u drugom testnom slučaju. Veličina sadržaja LZWEncoded.txt datoteke je 568,48 kB što je manje od početne veličine.



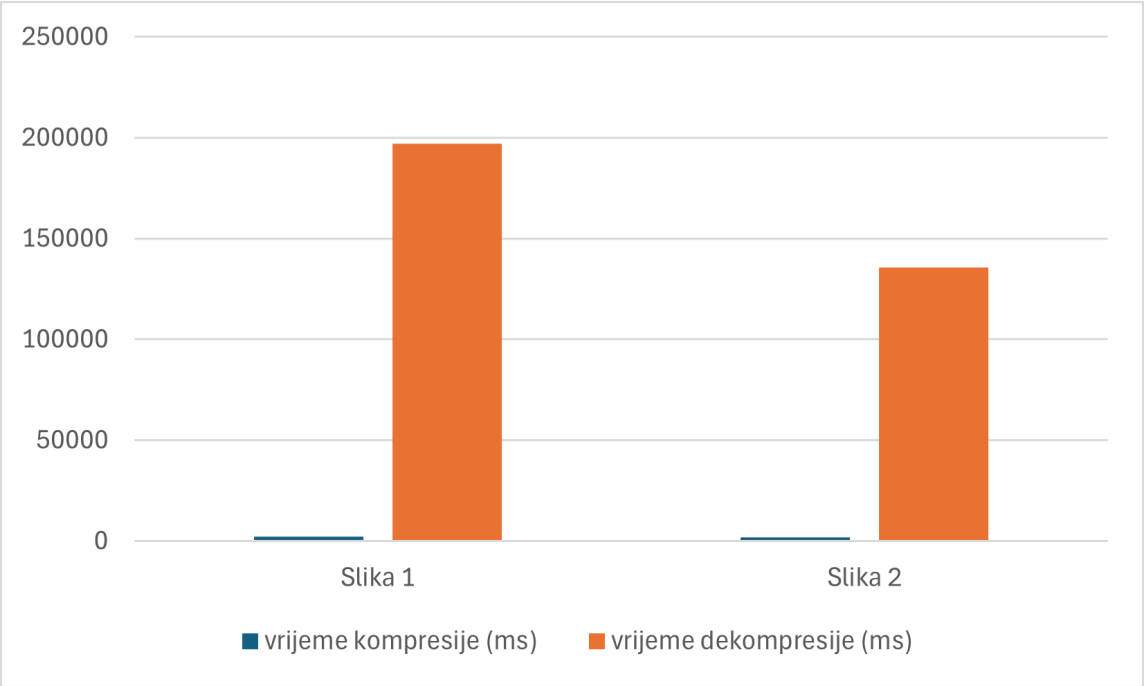
Slika 5.7 Usporedba veličina datoteka prije i nakon kompresije u trećem testnom slučaju.

Omjeri veličina prije i nakon kompresije pojedine slike prikazani su na Slici 5.8. Slika 4.2 ima negativan omjer kompresije što pokazuje da algoritam nije uspio smanjiti veličinu sadržaja, već je veličina datoteke nakon postupka veća za 5,38 % u odnosu na početnu sliku što je za 11,97 % bolje nego u prethodnom slučaju. Veličina datoteke nakon kompresije Slike 4.3 manja je za 25,98 % u odnosu na početnu sliku što je bolje nego u prethodnom slučaju za 11,63 %.



Slika 5.8 Usporedba omjera kompresije u trećem testnom slučaju.

U trećem testnom slučaju vrijeme potrebno za kompresiju i dekompresiju prikazano na Slici 5.9 je znatno kraće. Pretvorba slike u Base64 format u slučaju Slike 4.3 obavi se za samo 65 milisekundi, a vraćanje Base64 formata u sliku odvilo se za samo 62 milisekunde. Stoga u ovom slučaju najviše vremena zahtijeva BWT algoritam prilikom dekompresije. Od ukupnog vremena dekompresije Slike 4.3 koje iznosi 135680 milisekundi, BTW dekompresija se odvija 134836 milisekundi. Tijekom kompresije vrijeme izvršavanja BWT algoritma ne odskače toliko, izvrši se za 1059 milisekundi. RLE i LZW algoritmi su, kao i u prethodnim slučajevima, prilično brzi i izvrše se ispod 1000 milisekundi tijekom kompresije i dekompresije.



Slika 5.9 Usporedba vremena kompresije u trećem testnom slučaju.

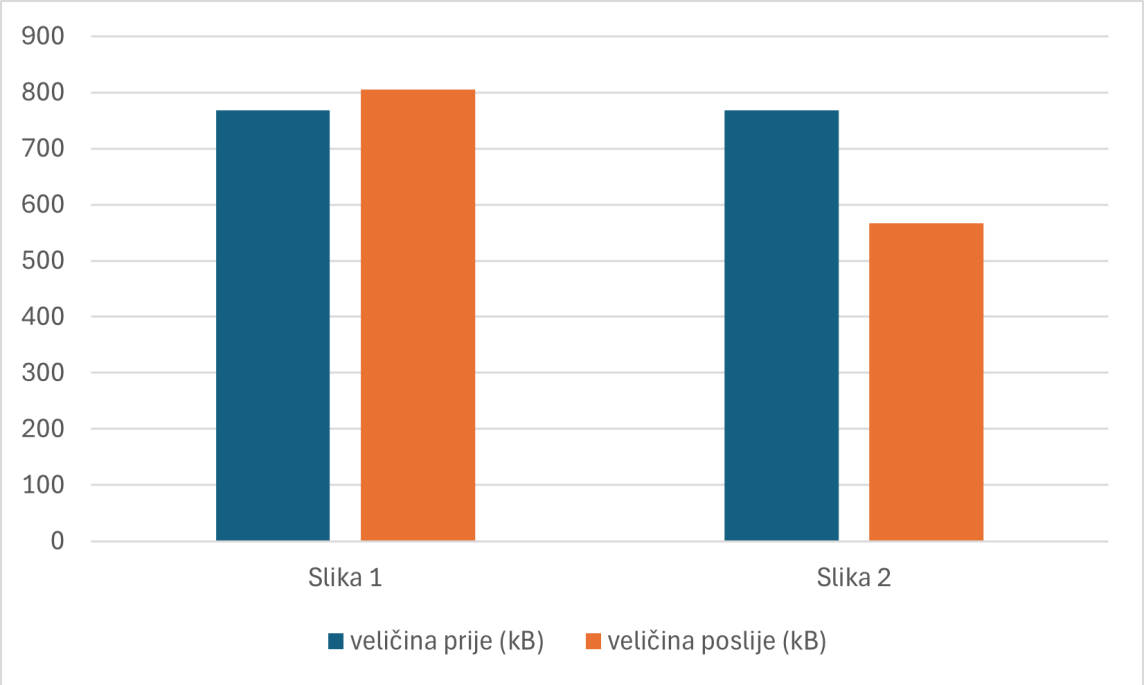
5.4 Četvrti testni slučaj

S obzirom na to da u trećem testnom slučaju BWT dekompresija traje jako dugo, četvrti testni slučaj ispituje postupak kompresije u kojem se slika pretvara u tekst s pomoću Base64 formata, ali u kompresiji ne sudjeluje BWT algoritam. Tablica 5.1 sadrži rezultate mjerenja veličina sadržaja prije i nakon kompresije, izračunate omjere tih veličina te vremena potrebna za izvršavanje kompresije i pojedinih algoritama.

Četvrti testni slučaj dao je slične rezultate kao i treći u kontekstu veličine sadržaja nakon kompresije. Sadržaj LZWEncoded datoteke u trećem testnom slučaju iznosio je 568,48 Kb za Sliku 4.3, a u ovom testnom slučaju iznosi 566,48 kB. S obzirom na to da sada u procesu kompresije nije sudjelovao BWT algoritam, može se zaključiti da nije pomogao prilikom kompresije Base64 formata, niti je odmogao. Razlog tome je sadržaj Base64 formata koji, čak ni nakon primjene BWT algoritma u prošlom slučaju, nije uspio napraviti razliku u rezultatu. Stoga i ovoga puta sadržaj Slike 4.2 nije smanjen dok sadržaj Slike 4.3 jest. Veličine sadržaja prije i nakon kompresije prikazane su na Slici 5.10.

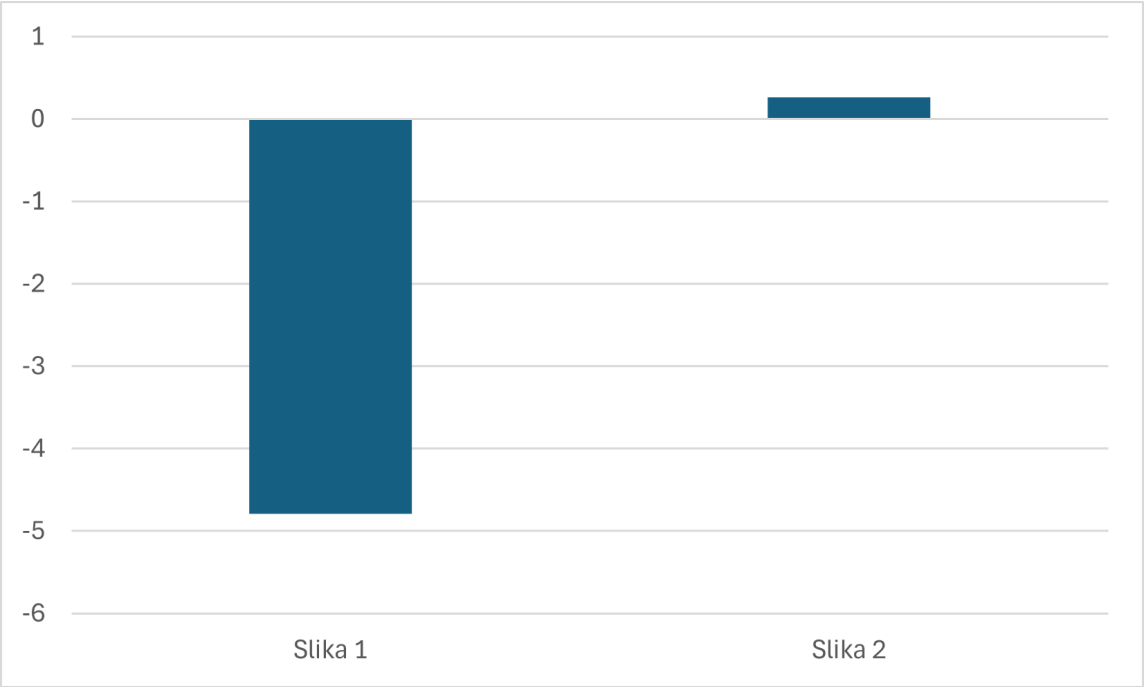
Tablica 5.4 Rezultati četvrtog testnog slučaja.

	Slika 4.2	**Slika 4.3**
veličina prije (kB)	768	768
veličina poslije (kB)	804,79	566,48
Omjer (%)	-4,79%	26,24%
vrijeme kompresije (ms)	893	600
vrijeme dekompresije (ms)	856	573
Slika u tekst (ms)	64	41
RLE kompresija (ms)	141	104
LZW kompresija (ms)	688	448
LZW dekompresija (ms)	647	431
RLE dekompresija (ms)	169	94
Tekst u sliku (ms)	39	47



Slika 5.10 Usporedba veličina datoteka prije i nakon kompresije u četvrtom testnom slučaju.

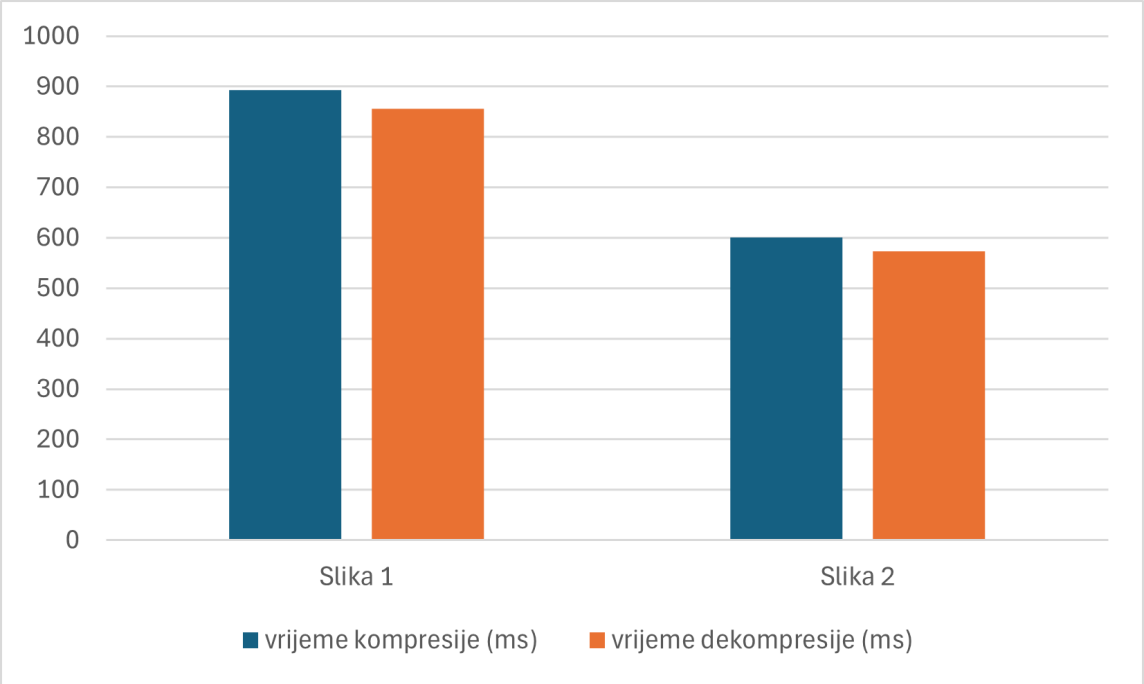
Omjeri veličina prije i nakon kompresije pojedine slike također su slični prethodnom slučaju i prikazani su na Slici 5.11. Slika 4.2 još jednom ima negativan omjer kompresije što pokazuje da algoritam nije uspio smanjiti veličinu sadržaja, već je veličina datoteke nakon postupka veća za 4,79 % u odnosu na početnu sliku što je za 0,59 % bolje nego u prethodnom slučaju. Veličina datoteke nakon kompresije Slike 4.3 manja je za 26,24 % u odnosu na početnu sliku što je bolje nego u prethodnom slučaju za 0,26 %. Ovako male razlike posljedica su slučajnosti i na njih nije utjecao BWT algoritam.



Slika 5.11 Usporedba omjera kompresije u četvrtom testnom slučaju.

Vrijeme izvršavanja kompresije u četvrtom testnom slučaju je kraće nego u svim prethodnim testnim slučajevima. Cijela kompresija se izvrši za samo 893 milisekunde u slučaju Slike 4.2, odnosno za 600 milisekundi u slučaju Slike 4.3. Slika 4.3 se brže komprimira zato što sadrži manje detalja, manju količinu informacije, pa posljedično ima i kraći Base64 zapis. Vrijeme dekompresije je slično: 856 milisekundi za Sliku 4.2 i 573 milisekunde za Sliku 4.3. S obzirom na to da su u ovom slučaju postignuti najbolji rezultati kompresije u kontekstu dobivenih veličina sadržaja LZWEncoded.txt datoteke u najkraćem do sad izmjerenom vremenu, četvrti testni slučaj je najbolja

opcija za komprimiranje sadržaja slike.



Slika 5.12 Usporedba vremena kompresije u četvrtom testnom slučaju.

Poglavlje 6

Zaključak

Ovaj rad istražuje uspješnost kompresije slike s pomoću algoritama za kompresiju teksta. Za pretvorbu slike u tekst korištena je Hilbertova krivulja i Base64 format. Dobiveni tekst je dalje kompresiran s pomoću Burrows–Wheeler transform (BWT), Run-length encoding (RLE) i Lempel–Ziv–Welch (LZW) algoritma. U svrhu testiranja algoritama korištene su fotografije 4.2 i 4.3 koje se razlikuju po sadržaju što utječe na uspješnost kompresije. Istraživanje je provedeno u četiri testna slučaja. Prvi testni slučaj koristi Hilbertovu krivulju za pretvorbu slike u tekst, a nakon toga je dobiveni tekst podložen BWT algoritmu koji pokuša optimizirati sadržaj za danju kompresiju. BWT pretprocesiranje teksta nije uspjelo postići bolje rezultate, već je razbilo nizove jednakih vrijednosti postignutih Hilbertovom krivuljom. S obzirom na to, rezultat je bio bolji u drugom testnom slučaju kada BWT algoritam nije primjenjen. Treći i četvrti testni slučaj za pretvorbu slike u tekst koriste Base64 format. S obzirom na to da Base64 format ne rezultira ponavljajućim nizovima, BWT algoritam je trebao riješiti taj problem. Međutim, BWT algoritam ni ovoga puta nije bio dovoljno uspješan, već je ishod bio isti u trećem testnom slučaju kada BWT algoritam jest korišten i u četvrtom testnom slučaju kada nije korišten što nije u skladu s hipotezom. Što se tiče usporedbe kompresije dviju slika, u slučaju Slike 4.2 kompresija nije postignuta u ni jednom testnom slučaju dok je u slučaju Slike 4.3 kompresija postignuta u svim testnim slučajevima. Razlog tome je jednolika ploha koju sadrži Slika 4.3 što potvrđuje hipotezu.

Ovaj algoritam može biti koristan u mnogim primjenama. Jedna od njih je opisana u ovom radu - kompresija. Ova primjena posebno je efikasna za slike koje ne

sadrže veliku količinu informacije kao što su grafike sa velikim područjima jednobojnih površina. To mogu biti, na primjer, crteži, skenirani dokumenti ili ikone. Druga moguća primjena je enkripcija podataka. Pretvorba slike u niz brojeva, što daje izlaz LZW kompresije, može biti dio tehnike steganografije, odnosno skrivanja informacija unutar slike. Takav postupak može pomoći u zaštiti slika od neovlaštenog pristupa i osiguravanju povjerljivosti podataka ili skrivanju tajnih podataka. Još jedna od mogućih primjena je i prijenos podataka putem mreže jer kompresirani podaci mogu smanjiti vrijeme prijenosa. Važno je napomenuti da je u ovom slučaju potrebno poslati i metapodatke kao što su Hilbertova matrica i LZW rječnik kako bi primatelj mogao obaviti dekompresiju i dekodiranje primljenih podataka. Baze podataka ponekad traže zapisivanje podataka u obliku niza brojeva što je još jedan slučaj kada bi ovaj postupak bio koristan. Osim do sada navedenih primjena, zanimljivo bi bilo iskoristiti dobiveni format kao digitalni potpis. Provjera se može vršiti tako da se primljena slika komprimira i usporede se dobiveni nizovi brojeva dobivenoj *LZWEncoded.txt* datoteci.

U budućim istraživanjima opisani postupak bi se mogao poboljšati, ponajprije u segmentu vremena potrebnog za pretvaranje slike u tekst i teksta u sliku s pomoću Hilbertove krivulje. Algoritam bi se mogao optimizirati ili paralelizirati. Također, slučaju pretvorbe slike u Base64 format, moguće je pronaći bolji način grupiranja istih znakova i stvaranja ponavljajućih nizova. RLE algoritam se može poboljšati na način da ne broji samo ponavljanja jednog znaka, već sekvenci sastavljenih od više znakova. Na kraj valja spomenuti i algoritme temeljene na strojnom učenju koji bi zasigurno bili korisni u optimizaciji rezultata.

Ovo istraživanje pokazalo je da je postupak kompresije slike s pomoću kompresije teksta ostvariv i koristan, pogotovo kada je potrebno komprimirati slike koje sadrže malu količinu informacije.

Literatura

[1] T. C. Bell, J. G. Cleary i I. H. Witten, *Text compression*. Prentice-Hall, Inc., 1990.

[2] K. Sayood, *Introduction to data compression*. Morgan Kaufmann, 2017.

[3] Z.-N. Li, M. S. Drew i J. Liu, *Fundamentals of multimedia*. Springer, 2004.

[4] D. Klen, J. Lerga i I. Petrijevčanin Vuksanović, "TXT AND TIF FILE COMPRESSION USING LZW, HUFFMAN, AND ARITHMETIC CODING," *10th INTERNATIONAL CONFERENCE ON MARINE TECHNOLOGY*.

[5] J. Ziv i A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, sv. 23, br. 3, str. 337–343, 1977.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest i C. Stein, *Introduction to Algorithms*, 2001.

[7] D. E. Knuth, "Dynamic huffman coding," *Journal of algorithms*, sv. 6, br. 2, str. 163–180, 1985.

[8] J. L. Núñez i S. Jones, "Gbit/s lossless data compression hardware," *IEEE Transactions on very large scale integration (VLSI) systems*, sv. 11, br. 3, str. 499–510, 2003.

[9] M. Hosseini, D. Pratas i A. J. Pinho, "Cryfa: a secure encryption tool for genomic data," *Bioinformatics*, sv. 35, br. 1, str. 146–148, 2019.

[10] M. Sardaraz, M. Tahir, A. A. Ikram i H. Bajwa, "SeqCompress: An algorithm for biological sequence compression," *Genomics*, sv. 104, br. 4, str. 225–228, 2014.

[11] M. Ignatoski, J. Lerga, L. Stanković i M. Daković, "Comparison of entropy and dictionary based text compression in English, German, French, Italian, Czech, Hungarian, Finnish, and Croatian," *Mathematics*, sv. 8, br. 7, str. 1059, 2020.

[12] H. Sagan, "Hilbert's Space-Filling Curve," *Space-Filling Curves*. New York, NY: Springer New York, 1994., str. 9–30, ISBN: 978-1-4612-0871-6. DOI: 10.1007/978-1-4612-0871-6_2. adresa: https://doi.org/10.1007/978-1-4612-0871-6_2.

[13] M. Burrows, "A block-sorting lossless data compression algorithm," *SRS Research Report*, sv. 124, 1994.

[14] M. B. Begum, N. Deepa, M. Uddin, R. Kaluri, M. Abdelhaq i R. Alsaqour, "An efficient and secure compression technique for data protection using burrows-wheeler transform algorithm," *Heliyon*, sv. 9, br. 6, 2023.

[15] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino i L. Versari, "Measuring the clustering effect of BWT via RLE," *Theoretical Computer Science*, sv. 698, str. 79–87, 2017.

[16] H. K. Reghbati, "Special feature an overview of data compression techniques," *Computer*, sv. 14, br. 04, str. 71–75, 1981.

[17] J. Ziv i A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, sv. 24, br. 5, str. 530–536, 1978.

[18] W. Cui, "New LZW data compression algorithm and its FPGA implementation," *Proc. 26th Picture Coding Symposium (PCS 2007)*, 2007.

[19] R. N. Horspool, "Improving LZW.," *Data Compression Conference*, Citeseer, 1991., str. 332–341.

[20] P. Kopylov i P. Franti, "Compression of map images by multilayer context tree modeling," *IEEE Transactions on Image Processing*, sv. 14, br. 1, str. 1–11, 2004.

[21] S. Forchhammer i O. R. Jensen, "Content layer progressive coding of digital maps," *IEEE transactions on image processing*, sv. 11, br. 12, str. 1349–1356, 2002.

[22] C. G. Rafael i E. W. Richard, *Digital image processing*. Pearson education, 2018.

[23] H. Dheemanth, "LZW data compression," *American journal of engineering research*, sv. 3, br. 2, str. 22–26, 2014.

[24] S. A. Taleb, H. Musafa, A. Khtoom i K. Gharaybih, "Improving LZW image compression," *European Journal of Scientific Research*, sv. 44, br. 3, str. 502–509, 2010.

[25] P. documentation, *History and License*, 2024. adresa: `https://docs.python.org/3.11/license.html`.

[26] O. documentation, *Introduction to OpenCV-Python Tutorials*, 2024. adresa: `https://docs.opencv.org/4.x/d0/de3/tutorial_py_intro.html`.

[27] Oracle, *What is Java technology and why do I need it?* 2024. adresa: `https://www.java.com/en/download/help/whatis_java.html`.

Pojmovnik

API eng. Application Programming Interface - programsko sučelje aplikacije

ASCII eng. American Standard Code for Information Interchange - standard za kodiranje znakova

BWT eng. Burrows-Wheeler Transform - algoritam za pretprocesiranje teksta

HTTP eng. Hypertext Transfer Protocol - protokol za prijenos hiperteksta

LZC eng. Lempel–Ziv complexity - mjera kompleksnosti

LZW eng. Lempel-Ziv-Welch - algoritam za kompresiju teksta

RLE eng. Run-Length Encoding - algoritam za kompresiju teksta

Sažetak

Ovaj rad istražuje postupak sažimanja slike s pomoću algoritama za kompresiju teksta. Slika se pretvara u tekstualni oblik na dva načina: koristeći Base64 format i obilazak slike s pomoću Hilbertove krivulje. Dobiveni niz znakova dalje se obrađuje s pomoću algoritma Burrows-Wheeler Transform (BWT). Dobiveni rezultat se potom podvrgava algoritmu Run-Length Encoding (RLE), a ishod se kodira korištenjem algoritma Lempel-Ziv-Welch (LZW). Tijekom izvođenja algoritama mjeri se potrebno vrijeme i postignuti stupanj kompresije. Algoritmi su testirani za četiri slučaja u kojima su korištene dvije testne slike. Testni slučajevi kombiniraju dva spomenuta načina pretvorbe slike u tekst i razlikuju se po korištenju, odnosno ne korištenju BWT algoritma za pretprocesiranje teksta. Rad pokazuje da je postupak kompresije slike s pomoću kompresije teksta ostvariv i koristan, pogotovo kada je potrebno komprimirati slike koje sadrže malu količinu informacije.

Ključne riječi **— kompresija slike, kompresija teksta, Burrows-Wheeler Transform, Run-Length Encoding, Lempel-Ziv-Welch, kompresija bez gubitaka**

Abstract

This paper explores the process of image compression using text compression algorithms. The image is converted into a textual form in two ways: using the Base64 format and traversing the image with a Hilbert curve. The resulting character string is further processed using the Burrows-Wheeler Transform (BWT) algorithm. The obtained result is then subjected to the Run-Length Encoding (RLE) algorithm, and the outcome is encoded using the Lempel-Ziv-Welch (LZW) algorithm. During the execution of these algorithms, the required time and the achieved compression ratio are measured. The algorithms were tested for four cases in which two test images were used. The test cases combine the two mentioned methods of image-to-text conversion and differ in the use or non-use of the BWT algorithm for text preprocessing. The work shows that the process of image compression using text compression is feasible and useful, especially when it is necessary to compress images that contain a small amount of information.

Keywords **— image compression, text compression, Burrows-Wheeler Transform, Run-Length Encoding, Lempel-Ziv-Welch, lossless compression**

Dodatak A

Izvorni kod

Izvorni kod nalazi se na javnom *Github* repozitoriju na poveznici:

https://github.com/alenlivojevic/diplomski