Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Luka Kovačič

IMPLEMENTACIJA SPIHT

Magistrsko delo

Maribor, februar 2024

IMPLEMENTACIJA SPIHT

Magistrsko delo

Študent:	Luka Kovačič
Študijski program:	Magistrski študijski program Računalništvo in informacijske tehnologije
Mentor:	red. prof. dr. Borut Žalik, univ. dipl. inž. el.
Somentor:	doc. dr. Štefan Kohek, mag. inž. rač. in inf. tehnol.

ZAHVALA

Zahvaljujem se mentorju, red. prof. dr. Borutu Žaliku, in somentorju, doc. dr. Štefanu Koheku, za strokovno pomoč pri izdelavi magistrskega dela. Posebna zahvala je namenjena staršem, ki so mi omogočili študij.

Implementacija SPIHT

Ključne besede: SPIHT, drevesa prostorske orientacije, Haarova transformacija, rastrske slike, napredujoče stiskanje

UDK: 004.353.244:004.422.63(043.2)

Povzetek

V magistrskem delu najprej predstavimo Haarovo transformacijo. Nato razložimo pojme in podatkovne strukture, potrebne za razumevanje algoritma SPIHT. Predstavimo psevdokod implementacije kodiranja in dekodiranja. Delovanje pokažemo na primeru. Algoritem implementiramo in ga testiramo na različnih slikah. Učinkovitost stiskanja primerjamo s formati PNG, JPEG in JPEG 2000, ki se izkaže kot najučinkovitejši.

SPIHT implementation

Keywords: SPIHT, spatial orientation trees, Haar transform, raster images, progressive coding

UDC: 004.353.244:004.422.63(043.2)

Abstract

In this thesis, first, the Haar transform is presented. Then, the basic concepts and data structures necessary for understanding the SPIHT algorithm are explained. Pseudocode implementations of encoding and decoding are presented. Their functioning is illustrated with an example. The algorithm is implemented and tested on different images. Compression efficiency is compared with the PNG, JPEG, and JPEG 2000 formats. The most efficient was JPEG 2000.

REPUBLIKA SLOVENIJA
SI-PASS

Podpisnik: Luka Kovačič
Izdajatelj: SI-PASS-CA
Št. potrdila: e84892670000000055755e36d
Potrdilo veljavno od: 08.02.2024 do 08.02.2029
Čas podpisa: 15:24 11.02.2024

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Koroška cesta 46
2000 Maribor, Slovenija

FERI

IZJAVA O AVTORSTVU ZAKLJUČNEGA DELA

Ime in priimek študent-a/-ke: Luka Kovačič
Študijski program: Računalništvo in informacijske tehnologije
Naslov zaključnega dela: Implementacija SPIHT

Mentor: red. prof. dr. Borut Žalik, univ. dipl. inž. el.
Somentor: doc. dr. Štefan Kohek, mag. inž. rač. in inf. tehnol.

Podpisan-i/-a študent/-ka Luka Kovačič

- izjavljam, da je zaključno delo rezultat mojega samostojnega dela, ki sem ga izdelal/-a ob pomoči mentor-ja/-ice oz. somentor-ja/-ice;
- izjavljam, da sem pridobil/-a vsa potrebna soglasja za uporabo podatkov in avtorskih del v zaključnem delu in jih v zaključnem delu jasno in ustrezno označil/-a;
- na Univerzo v Mariboru neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico ponuditi zaključno delo javnosti na svetovnem spletu preko DKUM; sem seznanjen/-a, da bodo dela deponirana/objavljena v DKUM dostopna široki javnosti pod pogoji licence Creative Commons BY-NC-ND, kar vključuje tudi avtomatizirano indeksiranje preko spleta in obdelavo besedil za potrebe tekstovnega in podatkovnega rudarjenja in ekstrakcije znanja iz vsebin; uporabnikom se dovoli reproduciranje brez predelave avtorskega dela, distribuiranje, dajanje v najem in priobčitev javnosti samega izvirnega avtorskega dela, in sicer pod pogojem, da navedejo avtorja in da ne gre za komercialno uporabo;
- dovoljujem objavo svojih osebnih podatkov, ki so navedeni v zaključnem delu in tej izjavi, skupaj z objavo zaključnega dela.

Uveljavljam permisivnejšo obliko licence Creative Commons:_____(navedite obliko)

Kraj in datum: Maribor, 11.2.2024

Podpis študent-a/-ke:

KAZALO

KAZALO SLIK

KAZALO TABEL

UPORABLJENI SIMBOLI IN KRATICE

DWT – diskretna valčna transformacija (angl. Discrete Wavelet Transform)

IDWT – inverzna diskretna valčna transformacija (angl. Inverse Discrete Wavelet Transform)

JPEG – standard za izgubno in brezizgubno stiskanje rastrskih slik (angl. Joint Photographic Experts Group)

LIP – seznam nepomembnih pikslov (angl. List of Insignificant Pixels)

LIS – seznam nepomembnih množic (angl. List of Insignificant Sets)

LSP – seznam pomembnih pikslov (angl. List of Significant Pixels)

NLS – algoritem SPIHT brez seznamov (angl. No List SPIHT)

PNG – slikovni format brezizgubnega stiskanja (angl. Portable Network Graphics)

RGB – barvni model, sestavljen iz rdeče (angl. red), zelene (angl. green) in modre (angl. blue) barve

SOT – drevo prostorske orientacije (angl. Spatial Orientation Tree)

SPIHT – algoritem, ki temelji na delitvi množic v hierarhičnih drevesih (angl. Set Partitioning in Hierarhical Trees)

SSIM – metrika strukturne podobnosti slik (angl. Structural Similarity Index Measure)

YCbCr – barvni prostor svetlosti (angl. luminance, Y) in krominance (angl. chrominance, Cb in Cr)

1 UVOD

Algoritmi za stiskanje slik [1] so zelo pomembni, saj tako slike zasedejo manj prostora na pomnilniškem mediju, kar posledično pomeni hitrejši prenos. Slike predstavljajo vsakdanji vir informacij na spletu, zato je pomembno, da se nam čim hitreje prikažejo na zaslonu. Še bolje je, če lahko takoj vidimo grobo vsebino slike, katere ločljivost se potem postopoma izboljšuje. To omogočajo algoritmi, ki temeljijo na napredujočem načinu stiskanja, kar pomeni, da so ključni deli slike poslani najprej, zatem pa sledijo podrobnosti. To omogoča tudi algoritem SPIHT (angl. Set Partitioning in Hierarchical Trees) [2], ki je tematika tega magistrskega dela.

SPIHT na vhodu prejme koeficiente večnivojske diskretne valčne transformacije in izkorišča korelacijo med koeficienti na različnih nivojih, ki predstavljajo enak frekvenčni pas. Razširjanje oz. dekodiranje je pri algoritmu hitrejše od stiskanja, torej gre za nesimetrični algoritem stiskanja.

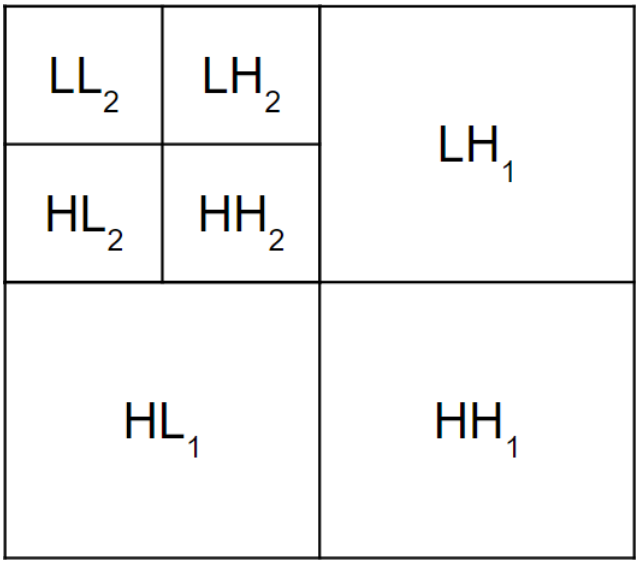
Magistrsko delo sestoji iz petih poglavij. V drugem poglavju obravnavamo Haarovo transformacijo. V tretjem poglavju predstavimo pojme, ki so potrebni za razumevanje algoritma SPIHT in psevdokod implementacije kodiranja in dekodiranja. Delovanje pokažemo na primeru. V četrtem poglavju testiramo učinkovitost algoritma napram formatom PNG, JPEG in JPEG 2000. V petem poglavju delo povzamemo.

2 DISKRETNA VALČNA TRANSFORMACIJA

Diskretna valčna transformacija (angl. Discrete Wavelet Transform - DWT) je matematično orodje, ki se uporablja pri analizi signalov [3]. Pogosto se uporablja za stiskanje slik, odpravljanje šuma in izdvajanje značilk. Nad vhodno sliko izvedemo DWT najprej po vrsticah, potem pa še po stolpcih (ali obratno). Uporabljamo nizkoprepustno in visokoprepustno sito oz. filter. Obenem signal podvzorčimo za faktor 2. Ko izvedemo 1-nivojsko dekompozicijo vhodne slike z DWT, 2D signal razdelimo na 4 komponente [4]:

- aproksimacijski koeficienti (LL), kjer nad vrsticami in stolpci uporabimo nizkoprepustno sito,
- koeficienti vodoravnih podrobnosti (LH), kjer nad vrsticami uporabimo nizkoprepustno sito in visoko nad stolpci,
- koeficienti navpičnih podrobnosti (HL), kjer nad vrsticami uporabimo visokoprepustno sito in nizko nad stolpci in
- koeficienti diagonalnih podrobnosti (HH), kjer nad vrsticami in stolpci uporabimo visokoprepustno sito.

Aproksimacija predstavlja približek vhodne slike. Če izvedemo DWT večkrat (večnivojska dekompozicija), potem v vsakem naslednjem nivoju izvedemo DWT nad aproksimacijo iz prejšnjega nivoja, kot vidimo na sliki 2.1.



Slika 2.1: Shema 2-nivojske dekompozicije

2.1 Haarova transformacija

Primer diskretne valčne transformacije je Haarova transformacija, ki uporablja Haarov valček [5]. 2D Haarovo transformacijo izvedemo tako, da imamo vhodno sliko in nad neprekrivajočimi se bloki pikslov velikosti 2×2 računamo aproksimacijo in podrobnosti (razlike v vodoravni, navpični in diagonalni smeri). Za izračun vodoravnih podrobnosti nas zanimajo razlike znotraj navpičnih parov pikslov. Nad vrsticami računamo povprečje (nizko sito, oznaka L) dveh sosednjih elementov. Nato pa izračunamo razlike (visoko sito, oznaka H) v navpični smeri (znotraj stolpcev). Podobno je za navpične podrobnosti z razliko, da tukaj računamo razlike parov pikslov v vodoravni smeri. Za diagonalne podrobnosti pa potrebujemo razliko razlik navpičnih ali vodoravnih parov pikslov. V tabeli 2.1 vidimo izračun koeficientov DWT za blok pikslov s slike 2.2. Opazimo, da je Haarov valček uporaben za detekcijo robov.

a	b
c	d

Slika 2.2: Blok pikslov

Tabela 2.1: Izračun Haarove transformacije

	$L = \frac{1}{2}\begin{bmatrix} a+c & b+d \end{bmatrix}$	$H = \frac{1}{2}\begin{bmatrix} a-c & b-d \end{bmatrix}$
$L = \frac{1}{2}\begin{bmatrix} a+b \\ c+d \end{bmatrix}$	$LL = \frac{1}{4}\begin{bmatrix} a+b+c+d \end{bmatrix}$	$LH = \frac{1}{4}\begin{bmatrix} a+b-c-d \end{bmatrix}$
$H = \frac{1}{2}\begin{bmatrix} a-b \\ c-d \end{bmatrix}$	$HL = \frac{1}{4}\begin{bmatrix} a-b+c-d \end{bmatrix}$	$HH = \frac{1}{4}\begin{bmatrix} a-b-c+d \end{bmatrix}$

Haarovo transformacijo lahko zapišemo kot matrično množenje, pri čemer potrebujemo matriko iz (2.1). Izračun poteka tako, da matriko A uporabimo dvakrat (2.2), pri čemer je B matrika pikslov in C matrika koeficientov DWT. Ker je inverz matrike A enak matriki A, dobimo začetne vrednosti pikslov po isti enačbi (2.2), le da sedaj zamenjamo vlogi B in C. Opazimo, da v matrični obliki pri izračunu vsote in razlike (znotraj vrstice oz. stolpca) rezultat delimo z $\frac{1}{\sqrt{2}}$ in ne z $\frac{1}{2}$, kot smo to storili v tabeli 2.1. V praksi se ne uporablja skalirni faktor $\frac{1}{2}$, ampak $\frac{1}{\sqrt{2}}$ [6], ker tako dobimo manjšo napako, ko rekonstruiramo kvantiziran oz. izgubni rezultat valčne transformacije, kar se zgodi v našem primeru, ko so kot vhod v SPIHT podani zaokroženi celoštevilski koeficienti DWT. Posledično ne dobimo natančnega povprečja (skupno ne delimo s 4, ampak 2) in razlike, rečemo, da dobimo grobe (angl. coarse) in fine (angl. fine) podrobnosti [7]. Velja poudariti, da z uporabo matričnega računa dobimo koeficient vodoravnih podrobnosti levo spodaj, koeficient navpičnih pa desno zgoraj.

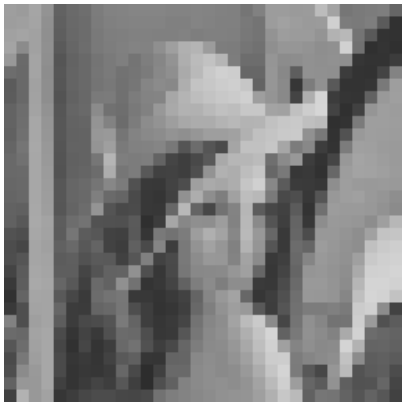
$$A = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.1}$$

$$ABA = C \tag{2.2}$$

Slika 2.3 prikazuje rezultat uporabe Haarove transformacije nad vhodno sliko Lene. Levo zgoraj imamo aproksimacijo vhodne slike, ki je zelo majhne velikosti, saj smo izvedli 3-

nivojsko DWT. Po izvedbi dekompozicije lahko sliko rekonstruiramo z uporabo inverzne diskretne valčne transformacije (angl. Inverse Discrete Wavelet Transform - IDWT), v našem primeru inverzne Haarove transformacije. Na sliki 2.4(a-d) vidimo postopno rekonstrukcijo slike Lene, ko smo nad njo izvedli 3-nivojsko dekompozicijo.

Slika 2.3: Primer 3-nivojske dekompozicije



(a) Aproksimacija

(b) Prvi nivo IDWT

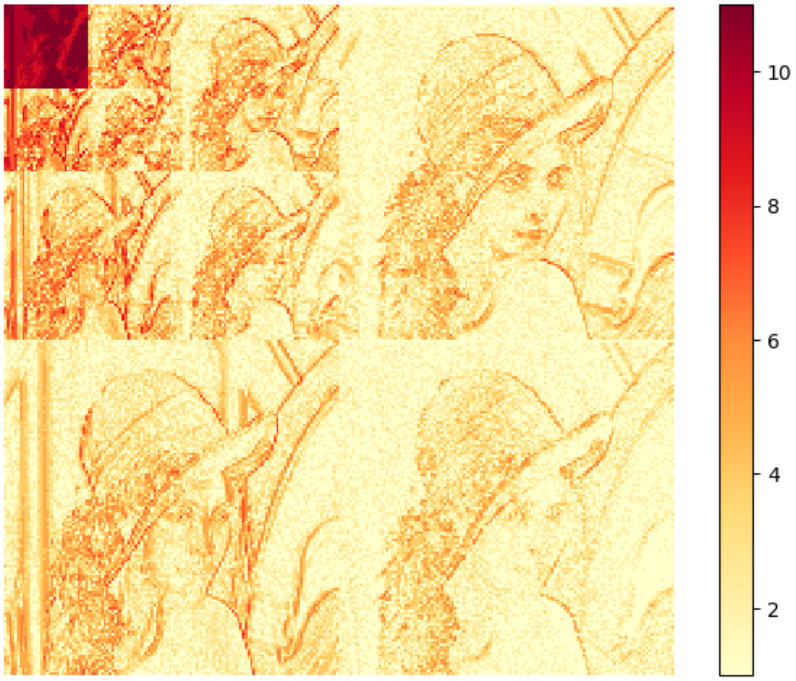
(c) Drugi nivo IDWT

(d) Tretji nivo IDWT

Slika 2.4: IDWT po nivojih

3 ALGORITEM SPIHT

SPIHT je algoritem za stiskanje slik, ki deluje nad celoštevilskimi vrednostmi koeficientov DWT. Temelji na korelaciji med koeficienti na različnih nivojih DWT, ki spadajo v isti frekvenčni pas. Slednji predstavlja vodoravne podrobnosti, navpične podrobnosti ali diagonalne podrobnosti. Ob večkratnem izvajanju DWT ugotovimo, da so koeficienti v višjih nivojih načeloma večji po absolutni vrednosti. To pomeni, da za njihovo predstavitev potrebujemo več bitov, kar prikazuje slika 3.1. To ni presenetljivo, saj izvajamo DWT nad vedno bolj grobo predstavitvijo izvorne slike, kar pomeni ostrejše prehode in s tem tudi večje razlike med piksli. Vsak koeficient na nekem nivoju DWT, je prostorsko tesno povezan s štirimi koeficienti na nižjem (spodnjem) nivoju DWT. Slednji so izračunani nad istim območjem manj grobe predstavitvije vhodne slike, torej pričakujemo manjše vrednosti koeficientov. Prostorska povezanost oz. koreliranost koeficente uvršča v isto prostorsko orientacijo. Če lahko vse koeficiente iste prostorske orientacije od nekega nivoja dekompozicije naprej predstavimo z manj biti kot neko trenutno mejno vrednost, potem lahko to označimo z enim bitom. Trenutna mejna vrednost je vedno potenca števila 2, kar nam omogoča preverbo, ali so biti koeficientov postavljeni (bit 1) ali ne (bit 0). Če so vsi biti 0 (vsi koeficienti so manjši od trenutne mejne vrednosti), se lahko izognemo pošiljanju bita za vsak koeficient posebej. Podatke na tak način stisnemo.



Slika 3.1: Število bitov, potrebnih za predstavitev koeficienta DWT

SPIHT omogoča napredujoče kodiranje (angl. progressive coding), kar pomeni, da se slika ob dekodiranju sproti izboljšuje. Uporablja tako imenovano vgrajeno kodiranje (angl. embedding coding) [8]. Če SPIHT generira dve kodirani datoteki, večjo velikosti m in manjšo velikosti n bitov, potem je manjša datoteka identična prvim n bitom večje datoteke. Kot primer vzemimo, da imamo tri uporabnike, ki želijo isto sliko, vendar so omejeni z velikostjo, ki jo lahko ta slika zasede na disku, kar pomeni, da bodo dobili sliko različne kakovosti. Večina izgubnih formatov za stiskanje slik bi moralo vhodno sliko stisniti trikrat z različno stopnjo stiskanja, SPIHT pa ta problem rešuje z eno kodirano datoteko, iz katere potem vzamemo začetnih k bitov oz. toliko, da ne presežemo dovoljene velikosti.

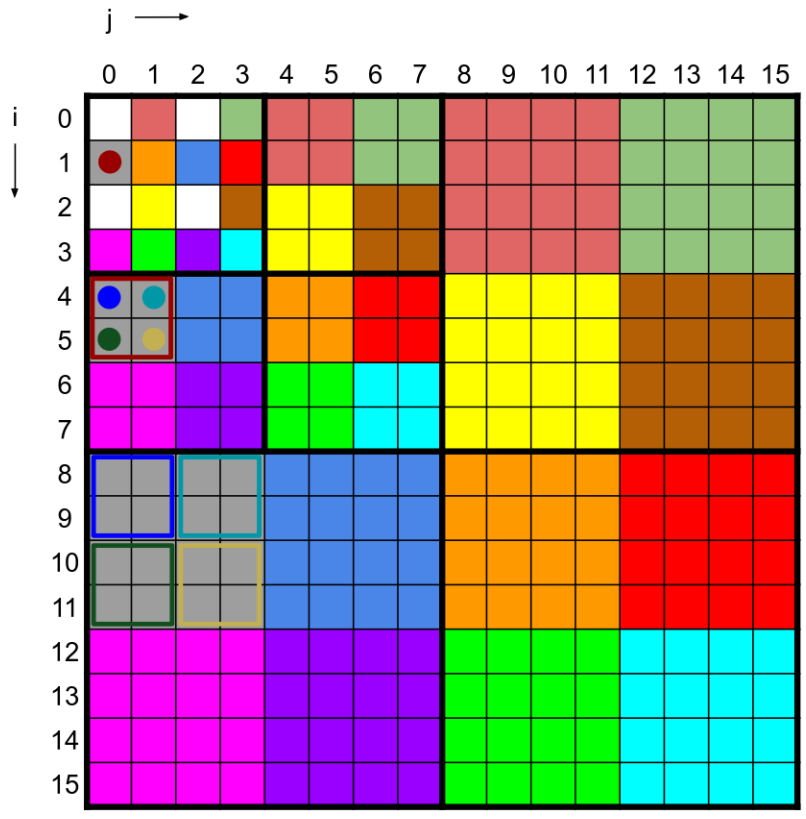
3.1 Osnovni pojmi SPIHT

3.1.1 Drevesa prostorske orientacije

Drevesa prostorske orientacije (angl. Spatial Orientation Trees - SOT) so drevesa, katerih vozlišča oz. elementi spadajo v isto prostorsko orientacijo. Za primer predpostavimo, da smo nad sliko velikosti 16×16 izvedli 2-nivojsko DWT, torej je aproksimacija (pas LL na

najvišjem nivoju DWT) velikosti 4×4. SPIHT razdeli koeficiente iz aproksimacije v neprekrivajoče se bloke velikosti 2×2. Levi zgornji elementi znotraj posameznih blokov (beli kvadratki na sliki 3.2) nimajo naslednikov. Ostali elementi znotraj blokov predstavljajo korene dreves in imajo neposredne naslednike (izjemoma) na istem nivoju DWT, ponazorjeni pa so z isto barvo.

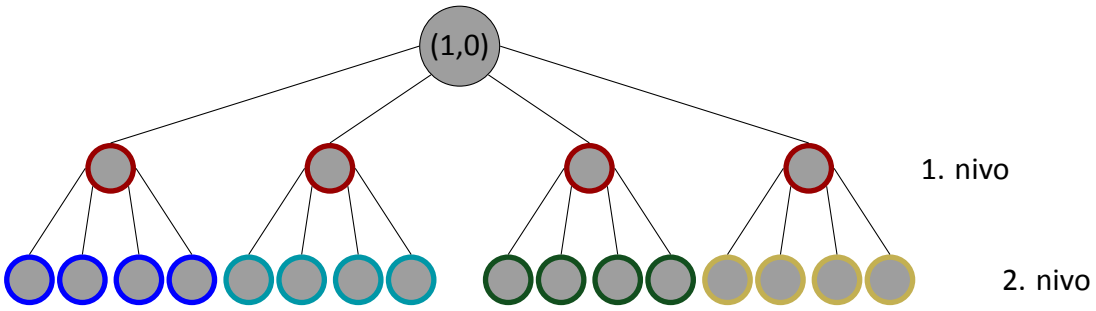
Vsak element v drevesu ima štiri neposredne naslednike (angl. offsprings). Element na položaju (1,0), prikazan s sivo barvo, ima neposredne naslednike na položajih (4,0), (4,1), (5,0) in (5,1). Slednji imajo svoje neposredne naslednike na naslednjem nivoju DWT (skupno 16 elementov, označenih s sivo barvo), ki predstavljajo posredne naslednike elementa na položaju (1,0).



Slika 3.2: Hierarhija med elementi v SOT

Opazimo, da imamo opravka s štiriškimi drevesi (angl. quadtrees) [9]. Relacijo med prej omenjenimi sivo obarvanimi koeficienti, v obliki štiriškega drevesa, prikazuje slika 3.3, kjer

koren drevesa predstavlja element na položaju (1,0). Na prvem nivoju najdemo neposredne naslednike korena, vsi nadaljnji nivoji pa predstavljajo njegove posredne naslednike. Če bi imeli sliko velikosti 32×32 in bi izvedli 3-nivojsko DWT, bi bil v štiriškem drevesu en nivo več, kar bi pomenilo dodatnih 64 posrednih naslednikov korena tega drevesa, elementa na položaju (1,0). Elementi (vozlišča), ki nimajo naslednikov, predstavljajo liste dreves.



Slika 3.3: Štiriško drevo iz slike 3.2

3.1.2 Podatkovne strukture SPIHT

Algoritem SPIHT uporablja tri sezname:

- seznam pomembnih pikslov (angl. List of Significant Pixels - LSP) vsebuje vrednosti, ki so večje ali enake od trenutne mejne vrednosti,
- seznam nepomembnih pikslov (angl. List of Insignificant Pixels - LIP), vsebuje vrednosti, ki so manjše od trenutne mejne vrednosti in
- seznam nepomembnih množic (angl. List of Insignificant Sets - LIS), vsebuje množice koeficientov DWT, določene s strukturo drevesa SOT, ki imajo vse elemente manjše od trenutne mejne vrednosti, torej so nepomembni.

V seznamih hranimo položaje oz. koordinate koeficientov znotraj matrike koeficientov. V seznamih LIP in LSP predstavljajo individualne koeficiente, v seznamu LIS pa množice, ki

ponazarjajo drevesa SOT. Imamo dva tipa množic, kot prikazuje tabela 3.1.

Tabela 3.1: Množice v seznamu LIS

Tip	**Oznaka**	**Opis**
A	$D(i,j)$	Množico sestavljajo vsi nasledniki, tako neposredni kot posredni, koeficienta na položaju (i,j). V štiriškem drevesu so to vsi elementi drevesa od vključno 1. nivoja dalje.
B	$L(i,j)$	Množico sestavljajo zgolj posredni nasledniki koeficienta na položaju (i,j). V štiriškem drevesu so to vsi elementi drevesa od vključno 2. nivoja dalje.

V tabeli 3.2 vidimo še dve množici, ki jih bomo uporabljali, in ne bosta del seznama LIS. Ugotovimo, da $L(i,j) = D(i,j) - O(i,j)$.

Tabela 3.2: Ostale množice

Oznaka	**Opis**
$O(i,j)$	Množico sestavljajo vsi neposredni nasledniki koeficienta na položaju (i,j).
H	Množico sestavljajo vsi koeficienti iz aproksimacije (pas LL na najvišjem nivoju DWT).

V inicializaciji algoritma SPIHT bomo imeli v seznamu LIS množice $D(i,j)$, kar velja za vse pare koordinat (i,j) iz množice H, ki imajo naslednike. Del seznama LIP pa bodo položaji vseh koeficientov iz množice H, torej tudi tisti v levem zgornjem kotu blokov velikosti 2×2, ki nimajo naslednikov. Seznam LSP na začetku nima elementov, bomo pa elemente iz seznama LIP premikali v seznam LSP, ko bomo ugotovili, da so pomembni.

3.2 Implementacija

Kodiramo vsako bitno ravnino posebej, od najbolj pomembne do najmanj pomembne. S številom bitov, potrebnih za predstavitev največjega koeficienta DWT po absolutni vrednosti, se določi začetna mejna vrednost, s katero preverjamo pomembnost individualnih koeficentov ali množic (če imajo bite v trenutno obravnavani bitni ravnini postavljene ali ne). Če so pomembni, na izhod pošljemo bit 1 oz. bit 0, če niso. Mejna vrednost se vsako naslednjo iteracijo razpolovi, saj kodiramo manj pomembne bite. Kodiranje posamezne bitne ravnine predstavlja eno iteracijo algoritma, ki poteka v treh fazah:

- Najprej na izhod pošljemo bite koeficientov iz seznama LIP. Pošiljamo torej individualne bite (en bit za vsak koeficient), gre za eksplicitno kodiranje (brez stiskanja podatkov). Pomembne koeficiente premaknemo v seznam LSP,

- nato obravnavamo množice iz seznama LIS in skušamo na izhod pošiljati en bit (0) za posamezno množico, kar se zgodi, ko imajo vsi elementi množice bite enake 0. Tako bite kodiramo implicitno (en bit za celotno množico), kar pomeni stiskanje podatkov. V nasprotnem primeru množice manjšamo (koeficente odstranjujemo in jih vstavljamo v seznam LIP oz. LSP) in jih po potrebi delimo na podmnožice. To delamo z namenom, da bi dobili množice, katerih vsi elementi so manjši od trenutne mejne vrednosti (imajo bit v trenutni bitni ravnini enak 0),

- nazadnje obdelamo seznam LSP, tudi tukaj, kot v obdelavi seznama LIP, pošiljamo bite individualnih koeficentov (eksplicitno kodiranje, brez stiskanja), vendar le manj pomembne (brez njihovega najbolj pomembnega bita). Koeficente torej ob kodiranju njihovega najbolj pomembnega bita premaknemo v seznam LSP. Ker seznam LSP obdelamo na koncu iteracije, algoritem pri kodiranju daje prednost koeficentom oz. množicam, katerih najbolj pomembnega bita še nismo kodirali.

Dekodiranje je podobno kodiranju, le da bite za kodirane koeficiente in množice beremo iz vhoda, kar nam omogoča postopno rekonstrukcijo koeficentov DWT.

3.2.1 Neposredni nasledniki

Algoritem 1 na vhodu prejme položaj koeficienta znotraj matrike koeficientov, na izhodu pa vrne položaje njegovih štirih neposrednih naslednikov, če koeficient ne predstavlja lista drevesa. Imamo dve možnosti: ali iščemo neposredne naslednike za element iz množice H in ima posledično neposredne naslednike na istem nivoju DWT, ali pa se le ti nahajajo na naslednjem nivoju DWT.

Algoritem 1: NeposredniNasledniki

Input: i, j koordinati koeficienta
Output: položaji neposrednih naslednikov
$visina\_aproks \leftarrow visina \,/\, 2^{st\_nivojev}$;
$sirina\_aproks \leftarrow sirina \,/\, 2^{st\_nivojev}$;
$k \leftarrow i$;
$l \leftarrow j$;
if $(i, j) \in H$ **then**
  if i ***mod** 2 = 1* **then**
    $k \leftarrow i + visina\_aproks - 1$;
  end
  if j ***mod** 2 = 1* **then**
    $l \leftarrow j + sirina\_aproks - 1$;
  end
else
  $k \leftarrow 2 \times i$;
  $l \leftarrow 2 \times j$;
end
if $k \geq visina$ ***or*** $l \geq sirina$ **then**
  return *{}*
else
  return $\{(k, l), (k, l + 1), (k + 1, l), (k + 1, l + 1)\}$
end

3.2.2 Matrika maksimumov

Za množici D in L bomo morali vedeti, ali sta pomembni. To se zgodi takrat, ko množica vsebuje vsaj eno vrednost, ki je po absolutni vrednosti večja ali enaka od trenutne mejne vrednosti. Pripravimo si matriko (absolutnih) maksimumov, kjer vsak element iz matrike koeficientov, ki ima naslednike, obravnavamo kot koren drevesa. Na njegov istoležen

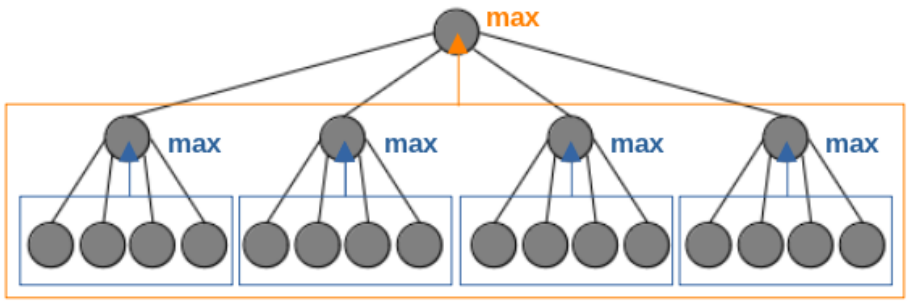
položaj v matriki maksimumov zapišemo maksimalno absolutno vrednost v tem drevesu (brez korena, zanimajo nas zgolj nasledniki). Pripravo matrike maksimumov prikazuje algoritem 2, ki ga izvedemo za vse korene dreves iz množice H. Z rekurzivno funkcijo se pomikamo od korena drevesa do listov in potem propagiramo maksimalne absolutne vrednosti na predhodne nivoje, dokler ne pridemo nazaj do korena.

Algoritem 2: AnalizaDrevesa

Input: i, j koordinati koeficienta
Output: maksimalna absolutna vrednost v (pod)drevesu s korenom na položaju (i, j)

$O \leftarrow NeposredniNasledniki(i, j)$;
if $len(O) = 0$ **then**
  return $abs(matrika\_koef[i, j])$;
end
$maksimumi \leftarrow \{\}$;
foreach (k, l) ***in*** O **do**
  $maksimumi = Dodaj(AnalizaDrevesa(k, l))$;
end
$matrika\_maks[i, j] = max(maksimumi)$;
return $max(matrika\_maks[i, j], abs(matrika\_koef[i, j]))$

Na sliki 3.4 lahko vidimo delovanje algoritma 2, ki gradi matriko maksimumov. Če nas zanima, ali je množica $D(i, j)$ pomembna, potem zgolj primerjamo vrednost v matriki maksimumov na položaju (i, j) in trenutno mejno vrednost. Če pa nas zanima pomembnost množice $L(i, j)$, pa moramo preveriti, če je katera izmed vrednosti na položajih neposrednih naslednikov (množica $O(i, j)$) večja ali enaka napram trenutni mejni vrednosti.



Slika 3.4: Delovanje algoritma 2

3.2.3 Kodiranje

Obdelava seznama LIS (algoritem 3) je ključnega pomena za učinkovitost algoritma SPIHT. Vsakič, ko preverjamo pomembnost množic in je slednja nepomembna, na izhod damo samo en bit za celotno množico, kar pomeni, da so biti vseh elementov znotraj množice na nekem indeksu (najmanj pomemben bit je na indeksu 0) enaki 0. Če je množica pomembna, jo zmanjšamo (iz množice tipa A preidemo na množico tipa B) in znova testiramo njeno pomembnost. Če je še vedno pomembna, jo razdelimo na štiri podmnožice. Postopek ponavljamo dokler ne dobimo samih nepomembnih množic. Če smo pozorni, opazimo, da se lahko zgodi, da na izhod pošiljamo redundantne bite (sicer se to zgodi redko oz. stiskanje ni bistveno učinkovitejše brez pošiljanja teh bitov):

- Za množico $L(i,j)$ na izhod ni potrebno dati bita o pomembnosti, če so vsi neposredni nasledniki elementa na položaju (i,j) nepomembni, saj to pomeni, da je zagotovo vsaj eden izmed posrednih naslednikov (množica L) pomemben.

- Ko množico razdelimo na štiri podmnožice, vemo, da je vsaj ena izmed teh množic pomembna. Če ob dekodiranju ugotovimo, da so tri od teh množic nepomembne, potem je preostala, četrta, zagotovo pomembna. Tako lahko bit o pomembnosti četrte množice preskočimo.

V algoritmu 4 je zapisan potek celotnega kodiranja. Korak je za ena manjši od števila bitov, ki jih potrebujemo za predstavitev vrednosti največjega koeficienta. V seznam LIP dodamo vse elemente iz aproksimacije, v seznam LIS pa vstavimo kot izhodišča dreves vse elemente, ki ne predstavljajo levega zgornjega elementa znotraj posameznih blokov velikosti 2×2, saj ti nimajo naslednikov.

Glavno zanko $while$, ki predstavlja eno iteracijo algoritma, izvajamo, dokler je korak večji ali enak pragu, algoritem lahko sicer zaključimo kadarkoli. Če je prag nastavljen na vrednost 0, potem bo stiskanje algoritma SPIHT brezizgubno. Velja poudariti, da lahko do manjših izgub pride že pred stiskanjem, saj SPIHT na vhodu pričakuje celoštevilske vrednosti.

Algoritem 3: ObdelavaLIS

```
foreach (i, j), tip in lis do
    if tip = A then
        pomembna_mnozica = matrika_maks[i, j] ≥ (1 ≪ korak);
        izhod(pomembna_mnozica);
        if pomembna_mnozica then
            O ← NeposredniNasledniki(i, j);
            foreach (k, l) in O do
                pomemben_koef = abs(matrika_koef[k, l]) ≥ (1 ≪ korak);
                izhod(pomemben_koef);
                if pomemben_koef then
                    lsp = Dodaj((k, l));
                    izhod(matrika_koef[k, l] < 0);
                else
                    lip = Dodaj((k, l));
                end
            end
            lis = Odstrani((i, j));
            O = NeposredniNasledniki(O[0].i, O[0].j);
            if len(O) ≠ 0 then
                lis = Dodaj((i, j)B);
            end
        end
    else
        O ← NeposredniNasledniki(i, j);
        pomembna_mnozica = 0;
        foreach (k, l) in O do
            pomembna_mnozica = matrika_maks[k, l] ≥ (1 ≪ korak);
            if pomembna_mnozica then
                break;
            end
        end
        izhod(pomembna_mnozica);
        if pomembna_mnozica then
            foreach (k, l) in O do
                lis = Dodaj((k, l)A);
            end
        end
    end
end
```

Koeficienti Haarove transformacije niso celoštevilski, zato jih je potrebno zaokrožiti na najbližjo celoštevilsko vrednost. Ko elementi v seznamu LIP postanejo pomembni, preidejo

Algoritem 4: Kodiranje

```
Input: matrika_koef, st_nivojev, prag
korak ← floor(log2(max(abs(matrika_koef))));
lip ← {};
lsp ← {};
lis ← {};
matrika_maks ← matrika velikosti enega kvadranta matrike_koef;
foreach (i, j) in H do
    lip = Dodaj((i, j));
    if i mod 2 ≠ 0 or j mod 2 ≠ 0 then
        lis = Dodaj((i, j)A);
        AnalizaDrevesa(i, j);
    end
end
while korak ≥ prag do
    foreach (i, j) in lip do
        pomemben_koef = abs(matrika_koef[i, j]) >= (1 ≪ korak);
        izhod(pomemben_koef);
        if pomemben_koef then
            lsp = Dodaj((i, j));
            izhod(matrika_koef[i, j] < 0);
        end
    end
    ObdelavaLIS();
    foreach (i, j) in lsp do
        if abs(matrika_koef[i, j]) ≥ (1 ≪ (korak + 1)) then
            izhod((abs(matrika_koef[i, j]) ≫ korak)&1);
        end
    end
    korak ← korak − 1;
end
```

v seznam LSP. Nato na izhod pošiljamo njihove preostale manj pomembne bite. Temu pravimo korak izboljšav (angl. refinement pass). Ugotovimo, da včasih koeficienti preidejo neposredno v seznam LSP, kar se zgodi takrat, ko so pomembni, preden bi jih dali v seznam LIP. Opazimo lahko, da psevdokod pozitiven predznak kodira kot bit 0, negativnega pa kot bit 1. Velja omeniti, da bi v prvi iteraciji, med obdelavo seznama LIP, lahko preskočili pošiljanje predznaka, saj so vsi koeficienti nenegativni (predpostavljamo, da uporabljamo Haarovo transformacijo).

3.2.4 Dekodiranje

Dekodiranje je podobno kodiranju, le da sedaj pomembnost koeficientov in množic preberemo iz vhoda. Posledično je dekodiranje hitrejše. Razlika je tudi, da sedaj posodabljamo bite koeficientov za njihovo rekonstrukcijo.

Algoritem 5: DekodiranjeObdelavaLIS

```
foreach (i, j), tip in lis do
    if tip = A then
        pomembna_mnozica = vhod();
        if pomembna_mnozica then
            O ← NeposredniNasledniki(i, j);
            foreach (k, l) in O do
                pomemben_koef = vhod();
                if pomemben_koef then
                    lsp = Dodaj((k, l));
                    if vhod() then
                        matrika_koef[k, l] = −1 × (1 ≪ korak);
                    else
                        matrika_koef[k, l] = 1 ≪ korak;
                    end
                else
                    lip = Dodaj((k, l));
                end
            end
            lis = Odstrani((i, j));
            O = NeposredniNasledniki(O[0].i, O[0].j);
            if len(O) ≠ 0 then
                lis = Dodaj((i, j)B);
            end
        end
    else
        pomembna_mnozica = vhod();
        if pomembna_mnozica then
            O ← NeposredniNasledniki(i, j);
            foreach (k, l) in O do
                lis = Dodaj((k, l)A);
            end
        end
    end
end
```

Algoritem 6: Dekodiranje

```
Input: kodirani_biti(vhod), st_nivojev, korak, visina_matrike_koef,
       sirina_matrike_koef
lip ← {};
lsp ← {};
lis ← {};
matrika_koef ← matrika velikosti visina_matrike_koef ×
 sirina_matrike_koef (vse vrednosti na 0);
foreach (i, j) in H do
    lip = Dodaj((i, j));
    if i mod 2 ≠ 0 or j mod 2 ≠ 0 then
        lis = Dodaj((i, j)A);
    end
end
while biti na voljo do
    foreach (i, j) in lip do
        pomemben_koef = vhod();
        if pomemben_koef then
            lsp = Dodaj((i, j));
            if vhod() then
                matrika_koef[i, j] = −1 × (1 ≪ korak);
            else
                matrika_koef[i, j] = 1 ≪ korak;
            end
        end
    end
    DekodiranjeObdelavaLIS();
    foreach (i, j) in lsp do
        if abs(matrika_koef[i, j]) ≥ (1 ≪ (korak + 1)) then
            if vhod() then
                if matrika_koef[i, j] >= 0 then
                    matrika_koef[i, j] = matrika_koef[i, j] | (1 ≪ korak);
                else
                    matrika_koef[i, j] = −(abs(matrika_koef[i, j]) | (1 ≪
                     korak));
                end
            end
        end
    end
    korak ← korak − 1;
end
```

Pri dekodiranju opazimo, kako občutljiv je SPIHT na napake v kodiranem bitnem nizu. En napačen bit med obdelavo seznama LIS, bi pomenil povsem drugačne vejitve kot v

fazi kodiranja, s tem pa bi bila rekonstrukcija bistveno drugačna. Ugotovimo, da se vpliv morebitne napake, z vsakim novim prebranim bitom, zmanjšuje.

Obstaja tudi različica algoritma SPIHT, imenovana NLS (angl. No List SPIHT) [10], ki deluje brez seznamov. Količina zahtevanega pomnilnika se določi na začetku, potrebnega je okoli 50% več, kot bi ga potrebovali za sliko samo. NLS je glede hitrosti učinkovitejši kot SPIHT. Algoritma na izhod pošljeta iste bite, vendar v drugačnem vrstnem redu. Če delamo rekonstrukcijo po vsaki iteraciji (po vsaki bitni ravnini), sta rezultata identična. SPIHT se izkaže malce bolje med samo iteracijo (če bi delali rekonstrukcijo med obdelavo seznama LIS).

3.3 Primer delovanja

3.3.1 Kodiranje

V tabeli 3.3 imamo podano matriko koeficientov, pridobljenih po 2-nivojski DWT. Naj bo prag 3, kar pomeni, da ne bomo kodirali treh najmanj pomembnih bitov. Za predstavitev največjega absolutnega koeficienta, ki je 31, potrebujemo 5 bitov, zato je korak na začetku algoritma enak 4 (spomnimo, da je najmanj pomemben bit na indeksu 0). To pa pomeni, da je mejna vrednost na začetku enaka 16. Opravili bomo torej 2 iteraciji, kar je dovolj, da spoznamo delovanje vseh delov algoritma.

Najprej si pripravimo matriko maksimumov. Dobimo matriko, vidno v tabeli 3.4. Zaradi 2-nivojske DWT imamo v aproksimaciji samo en blok velikosti 2×2, torej imamo tri korene dreves. Levi zgornji element na položaju (0,0) nima naslednikov in zato ni koren. Enako velja za liste, koeficiente na prvem nivoju DWT. Za vsak koeficient smo na njegov istoležen položaj v matriki zabeležili maksimalno absolutno vrednost v njegovem drevesu, kjer koeficient predstavlja začetek oz. koren drevesa. Spomnimo, da upoštevamo zgolj naslednike korena, brez njega samega. Matrika maksimumov nam bo pomagala pri določevanju

Tabela 3.3: Koeficienti 2-nivojske DWT

31	25	-6	2	-2	3	0	0
17	13	4	5	5	3	-1	0
5	10	0	0	1	0	3	-6
-9	7	0	0	-2	0	-1	-1
0	1	12	-4	0	0	0	0
5	-2	-1	2	0	0	0	0
-3	1	4	0	0	0	0	0
0	-2	1	-1	0	0	0	0

pomembnosti množic, kot je razloženo v podpoglavju 3.2.2.

Tabela 3.4: Matrika maksimumov

/	6	5	1	/	/	/	/
12	0	2	6	/	/	/	/
5	12	0	0	/	/	/	/
3	4	0	0	/	/	/	/
/	/	/	/	/	/	/	/
/	/	/	/	/	/	/	/
/	/	/	/	/	/	/	/
/	/	/	/	/	/	/	/

Na začetku imamo vse elemente iz bloka velikosti 2×2 v seznamu LIP. Vsi, razen elementa na položaju (1,1) so pomembni, saj so po absolutni vrednosti večji od trenutne mejne vrednosti, ki je 16. Zato na izhod, poleg bita na indeksu 4, pošljemo še njihov predznak, in jih premaknemo v seznam LSP. Postopek prikazuje tabela 3.5.

Tabela 3.5: Obdelava seznama LIP (korak = 4)

test	pogoj	izhod	naloga
(0,0)	$31 \geq 16$	1+	(0,0) v LSP
(0,1)	$25 \geq 16$	1+	(0,1) v LSP
(1,0)	$17 \geq 16$	1+	(1,0) v LSP
(1,1)	$13 \geq 16$	0	/

V tabeli 3.6 vidimo obdelavo seznama LIS, v katerem imamo na začetku tri množice. Vsi elementi znotraj posameznih množic so manjši od trenutne mejne vrednosti, zato za vsako množico na izhod pošljemo bit 0. Tukaj vidimo učinkovitost algoritma, saj smo lahko z enim bitom povedali, da so biti vseh koeficientov znotraj posamezne množice enaki 0.

Tabela 3.6: Obdelava seznama LIS (korak = 4)

test	pogoj	izhod	naloga
D(0,1)	$6 \geq 16$	0	/
D(1,0)	$12 \geq 16$	0	/
D(1,1)	$0 \geq 16$	0	/

V tabeli 3.7 vidimo obdelavo seznama LSP, kjer za vsak koeficient pošiljamo vse bite, razen najbolj pomembnega. Bite pošiljamo postopoma, v eni iteraciji damo na izhod en bit za vsak koeficient. Posledično v tej iteraciji glavne zanke v koraku 4 izboljšav ni.

Tabela 3.7: Obdelava seznama LSP (korak = 4)

test	pogoj	izhod
(0,0)	$31 \geq 32$	/
(0,1)	$25 \geq 32$	/
(1,0)	$17 \geq 32$	/

Sledi nova iteracija. Korak smo zmanjšali na 3, mejna vrednost je sedaj 8. Edini koeficient v seznamu LIP, koeficient na položaju (1,1), je postal pomemben. Kot prikazuje tabela 3.8, na izhod pošljemo bit 1 in njegov predznak ter ga premaknemo v seznam LSP.

Tabela 3.8: Obdelava seznama LIP (korak = 3)

test	pogoj	izhod	naloga
(1,1)	$13 \geq 8$	1+	(1,1) v LSP

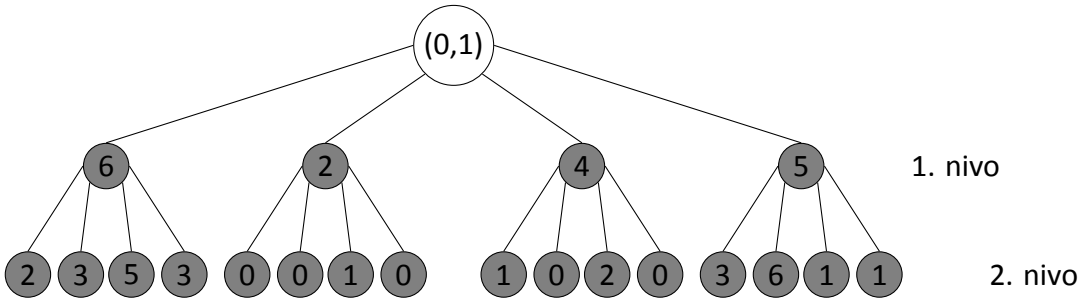
Množica $D(0,1)$ še vedno ni pomembna, zato na izhod pošljemo bit 0. Množica $D(1,0)$ je pomembna, zato na izhod pošljemo bit 1 in testiramo neposredne naslednike. Za slednje na izhod pošljemo njihov bit na indeksu 3, če je bit 0 (so nepomembni) jih vstavimo v seznam LIP. Če so pomembni, pa pošljemo še njihov predznak in jih vstavimo v seznam LSP. Spremenimo še tip množice in jo dodamo na konec seznama. Sledi množica $D(1,1)$, ki ni pomembna, zato na izhod damo bit 0 in ne naredimo ničesar. Neposredni nasledniki imajo svoje naslednike, ki predstavljajo posredne naslednike (množica L) njihovega predhodnika. Vsi elementi množice $L(1,0)$ niso manjši od trenutne mejne vrednosti, zato moramo neposredne naslednike vstaviti v seznam LIS kot nova izhodišča dreves. Obenem odstranimo množico $L(1,0)$ iz seznama LIS. Množica $D(2,0)$ ni pomembna, zato na izhod

damo bit 0 in ne storimo ničesar. Postopek nadaljujemo kot prikazuje tabela 3.9.

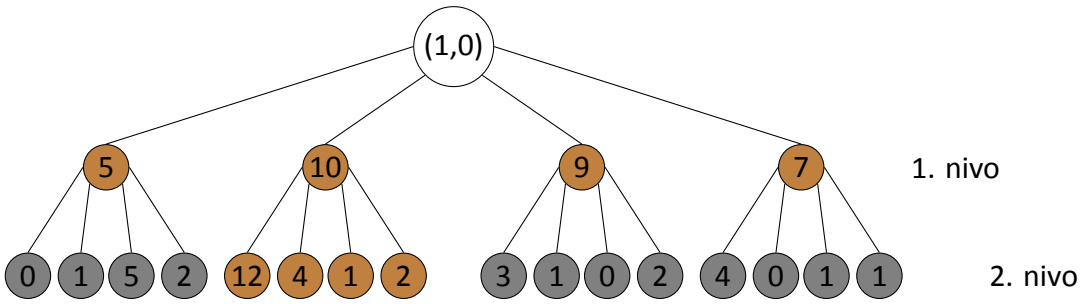
Obdelavo seznama LIS si lahko lažje predstavljamo na štiriških drevesih, ki so vidna na slikah 3.5, 3.6 in 3.7. Siva vozlišča ponazarjajo, da smo njihove bite kodirali implicitno (na izhod smo dali bit 0, ki je pomenil, da so vsa vozlišča znotraj (pod)drevesa pod trenutno mejno vrednostjo), medtem, ko smo za vozlišča z rjavo barvo morali bite kodirati individualno. Rjava vozlišča so tudi nemudoma premaknjena v seznam LIP oz. LSP, kar pomeni, da bomo tudi v naslednjih iteracijah morali za njih pošiljati individualne bite. Vsako rjavo vozlišče postane tudi nov koren drevesa (nova množica), tako štiri rjava vozlišča nadomestijo svojega predhodnika v seznamu LIS. To seveda ne velja za liste dreves, saj slednji nimajo naslednikov in jih nikoli ne vstavljamo v seznam LIS kot izhodišča dreves. Opazimo, da je najslabši scenarij, če se vrednosti, ki niso manjše od trenutne mejne vrednosti, nahajajo v listih dreves. Ugotovimo, da dlje kot nam uspe skozi iteracije algoritma ostati v zgornjih nivojih dreves (imamo množice z več elementi), bolj učinkovito je stiskanje, saj manjkrat vstavljamo (delimo) množice v seznam LIS. To pa pomeni manj poslanih bitov na izhod.

Tabela 3.9: Obdelava seznama LIS (korak = 3)

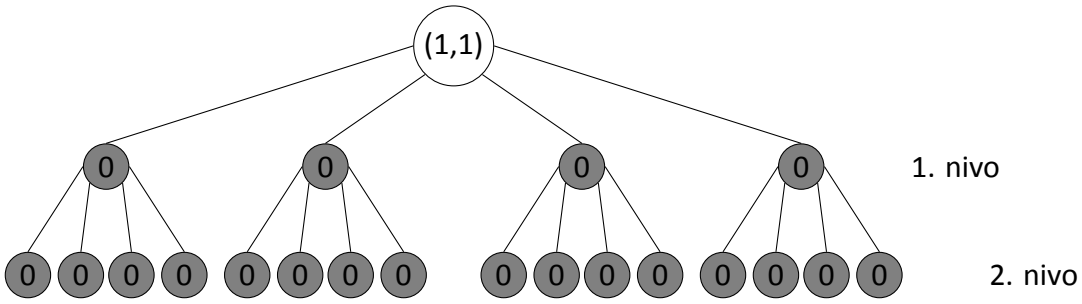
test	pogoj	izhod	naloga
D(0,1)	$6 \geq 8$	0	/
D(1,0)	$12 \geq 8$	1	test neposrednih naslednikov
(2,0)	$5 \geq 8$	0	(2,0) v LIP
(2,1)	$10 \geq 8$	1+	(2,1) v LSP
(3,0)	$9 \geq 8$	1-	(3,0) v LSP
(3,1)	$7 \geq 8$	0	(3,1) v LIP
			sprememba tipa (1,0)A v (1,0)B
D(1,1)	$0 \geq 8$	0	/
L(1,0)	maksimum(5, 12, 3, 4) ≥ 8	1	vstavitev množic (2,0), (2,1), (3,0), (3,1) kot tip A in odstranitev množice (1,0)
D(2,0)	$5 \geq 8$	0	/
D(2,1)	$12 \geq 8$	1	test neposrednih naslednikov
(4,2)	$12 \geq 8$	1+	(4,2) v LSP
(4,3)	$4 \geq 8$	0	(4,3) v LIP
(5,2)	$1 \geq 8$	0	(5,2) v LIP
(5,3)	$2 \geq 8$	0	(5,3) v LIP
			odstranitev množice (2,1)
D(3,0)	$3 \geq 8$	0	/
D(3,1)	$4 \geq 8$	0	/



Slika 3.5: Štiriško drevo pri korenu (0,1) in mejni vrednosti 8



Slika 3.6: Štiriško drevo pri korenu (1,0) in mejni vrednosti 8



Slika 3.7: Štiriško drevo pri korenu (1,1) in mejni vrednosti 8

Sedaj imamo v seznamu LSP koeficiente, ki v seznam niso bili dodani v trenutni iteraciji algoritma. Za njih bomo, kot prikazuje tabela 3.10, od vključno te iteracije naprej, pošiljali manj pomembne bite.

Tabela 3.10: Obdelava seznama LSP (korak = 3)

test	pogoj	izhod
(0,0)	$31 \geq 16$	1
(0,1)	$25 \geq 16$	1
(1,0)	$17 \geq 16$	0
(1,1)	$13 \geq 16$	/
(2,1)	$10 \geq 16$	/
(3,0)	$9 \geq 16$	/
(4,2)	$12 \geq 16$	/

Korak zmanjšamo na 2, kar je manj od praga, ki je 3, zato kodiranje končamo. Za boljšo rekonstrukcijo koeficientov ob dekodiranju, lahko kodiranje nadaljujemo do konca (dokler je korak nenegativen).

3.3.2 Dekodiranje

Pri dekodiranju koeficiente rekonstruiramo bitno ravnino po bitno ravnino od najbolj do najmanj pomembne oz. dokler nam na vhodu ne zmanjka bitov.

Inicializacija je enaka kot pri kodiranju, torej imamo v seznamu LIP vse elemente iz aproksimacije, v seznamu LIS pa so izhodišča dreves vsi elementi iz aproksimacije, razen tistih, ki predstavljajo levi zgornji element znotraj bloka velikosti 2×2. Ob dekodiranju moramo torej poznati podatke o velikosti 2D polja koeficientov, ki smo ga kodirali, obenem moramo vedeti tudi koliko nivojev DWT smo opravili (da vemo kakšna je množica H) in vrednost koraka na začetku, s katerim določimo začetno mejno vrednost.

Najprej dobimo na vhodu informacije o pomembnosti koeficientov iz aproksimacije (množica H), kot je razvidno iz tabele 3.11. Za pomembne koeficiente preberemo še predznak in posodobimo matriko koeficientov.

Tabela 3.11: Obdelava seznama LIP (korak = 4)

test	vhod	naloga
(0,0)	1+	(0,0) v LSP in matrika_koef(0,0) = 16
(0,1)	1+	(0,1) v LSP in matrika_koef(0,1) = 16
(1,0)	1+	(1,0) v LSP in matrika_koef(1,0) = 16
(1,1)	0	/

Tabela 3.12 prikazuje obdelavo seznama LIS. Na vhodu dobimo tri bite 0, kar pomeni, da so vse tri množice nepomembne. Z drugimi besedami, nasledniki (neposredni in posredni) vseh treh korenov dreves iz aproksimacije so manjši od trenutne mejne vrednosti, ki je 16, zato ne storimo ničesar.

Tabela 3.12: Obdelava seznama LIS (korak = 4)

test	vhod	naloga
D(0,1)	0	/
D(1,0)	0	/
D(1,1)	0	/

V tabeli 3.13 vidimo, da v prvi iteraciji med obdelavo seznama LSP na vhodu ne dobimo novih bitov.

Tabela 3.13: Obdelava seznama LSP (korak = 4)

test	pogoj	vhod	naloga
(0,0)	16 >= 32	/	/
(0,1)	16 >= 32	/	/
(1,0)	16 >= 32	/	/

Trenutno stanje dekodirane matrike koeficientov je vidno v tabeli 3.14.

Tabela 3.14: Dekodirani koeficienti po koraku 4

16	16	0	0	0	0	0	0
16	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Sledi nova iteracija, korak zmanjšamo na 3. V seznamu LIP je samo en element (koeficient). Iz vhoda preberemo bit 1, kar pomeni, da je element po absolutni vrednosti večji ali vsaj enak v primerjavi s trenutno mejno vrednostjo, ki je 8, zato preberemo še en bit za predznak. Element premaknemo v seznam LSP, kjer bomo v naslednjih iteracijah zanj pošiljali manj pomembne bite. Postopek prikazuje tabela 3.15.

Tabela 3.15: Obdelava seznama LIP (korak = 3)

test	vhod	naloga
(1,1)	1+	(1,1) v LSP in matrika_koef(1,1) = 8

V obdelavi seznama LIS najprej preberemo bit o pomembnosti množice $D(0,1)$. Slednji je 0, kar pomeni, da je množica nepomembna in ne storimo ničesar. Naslednji bit je 1,

kar pomeni, da je množica $D(1,0)$ pomembna, zato testiramo neposredne naslednike in spremenimo tip množice. Za koeficienta na položajih (2,0) in (3,1) ugotovimo, da sta nepomembna (imata bit na indeksu 3 enak 0 oz. njun najbolj pomemben bit se nahaja na indeksu manjšem od 3), zato ju samo premaknemo v seznam LIP. Koeficienta na položajih (2,1) in (3,0) sta pomembna, zato preberemo še njuna predznaka in na istoležnih položajih v matriki koeficientov nastavimo vrednosti koeficientov na 8 oz. -8 (odvisno od preznaka). Preberemo nov bit, ki je 0, kar pomeni da je množica $D(1,1)$ nepomembna in ne naredimo ničesar. Sedaj obravnavamo množico tipa B, to je množica $L(1,0)$, bit na vhodu pa je 1, kar pomeni, da je vsaj eden izmed posrednih naslednikov drevesa s korenom na položaju (1,0) pomemben, zato moramo neposredne naslednike korena vstaviti v seznam LIS kot nova izhodišča dreves. Obenem odstranimo množico $L(1,0)$. Naslednji bit je 0, kar pomeni, da je množica $D(2,0)$ nepomembna in ne storimo ničesar. Nadaljujemo po zgledu v tabeli 3.16.

Tabela 3.16: Obdelava seznama LIS (korak = 3)

test	vhod	naloga
D(0,1)	0	/
D(1,0)	1	test neposrednih naslednikov
(2,0)	0	(2,0) v LIP
(2,1)	1+	(2,1) v LSP in matrika_koef(2,1) = 8
(3,0)	1-	(3,0) v LSP in matrika_koef(3,0) = -8
(3,1)	0	(3,1) v LIP
		sprememba tipa (1,0)A v (1,0)B
D(1,1)	0	/
L(1,0)	1	vstavitev množic (2,0), (2,1), (3,0), (3,1) kot tip A in odstranitev množice (1,0)
D(2,0)	0	/
D(2,1)	1	test neposrednih naslednikov
(4,2)	1+	(4,2) v LSP in matrika_koef(4,2) = 8
(4,3)	0	(4,3) v LIP
(5,2)	0	(5,2) v LIP
(5,3)	0	(5,3) v LIP
		odstranitev množice (2,1)
D(3,0)	0	/
D(3,1)	0	/

V tabeli 3.17 vidimo obdelavo seznama LSP, kjer na vhodu pričakujemo bite (na indeksu 3) koeficientov, katerih najbolj pomemben bit je na indeksu večjem od 3. Če na začetku dekodiranja vse vrednosti v matriki koeficientov postavimo na 0, potem posodabljamo vrednosti samo, ko preberemo bit 1.

V tabeli 3.18 imamo matriko dekodiranih koeficientov po koraku 3 (po dekodiranju bitnih ravnin na indeksih 4 in 3). Na vhodu ni več novih bitov, zato dekodiranje zaključimo.

Tabela 3.17: Obdelava seznama LSP (korak = 3)

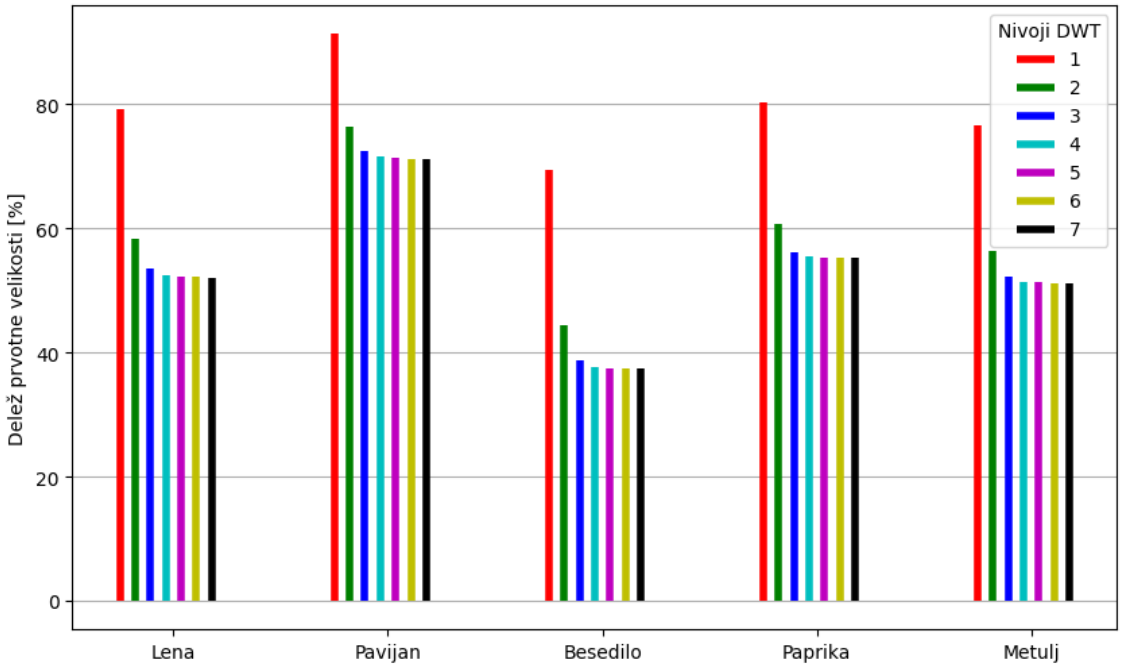
test	pogoj	vhod	naloga
(0,0)	16 >= 16	1	matrika_koef(0,0) -> postavitev bita na indeksu 3
(0,1)	16 >= 16	1	matrika_koef(0,1) -> postavitev bita na indeksu 3
(1,0)	16 >= 16	0	/
(1,1)	8 >= 16	/	/
(2,1)	8 >= 16	/	/
(3,0)	8 >= 16	/	/
(4,2)	8 >= 16	/	/

Tabela 3.18: Dekodirani koeficienti po koraku 3

24	24	0	0	0	0	0	0
16	8	0	0	0	0	0	0
0	8	0	0	0	0	0	0
-8	0	0	0	0	0	0	0
0	0	8	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

4 REZULTATI

SPIHT smo testirali nad različnimi barvnimi slikami velikosti 256×256. Najprej smo slike, sestavljene iz kanalov rdeče, zelene in modre barve, pretvorili v barvni prostor YCbCr [11], sestavljen iz svetlosti (angl. luminance, Y) in krominance (angl. chrominance, Cb in Cr). Algoritem na vhodu prejme koeficiente DWT, pridobljene s Haarovo transformacijo, ki jih zaokrožimo na najbližjo celoštevilsko vrednost. Koeficienti so stisnjeni pri različnih pragovih. Na sliki 4.1 vidimo učinkovitost stiskanja glede na število nivojev DWT. Rezultati so bili pridobljeni pri brezizgubnem stiskanju (pri pragu 0). Pri testiranju smo se odločili za 5-nivojsko dekompozicijo, saj stiskanje pri večih nivojih ni bistveno učinkovitejše.

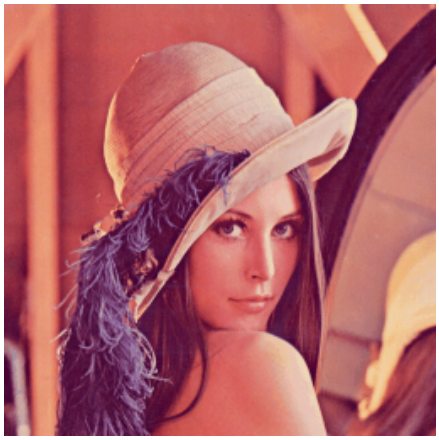


Slika 4.1: Učinkovitost stiskanja glede na število nivojev DWT

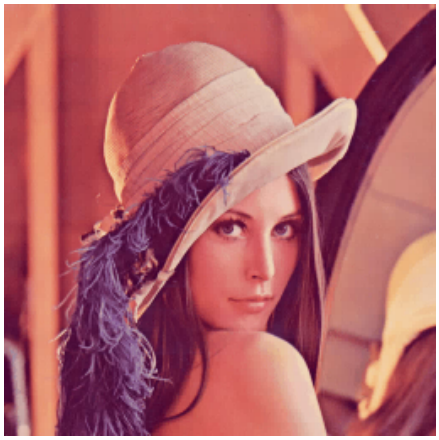
Naredili smo primerjavo s formati PNG [12], JPEG [13] in JPEG 2000 [14]. Učinkovitost stiskanja smo, če je bilo možno, primerjali pri čimbolj podobnih vrednostih metrike SSIM [15].

4.1 Testne slike

- Lena

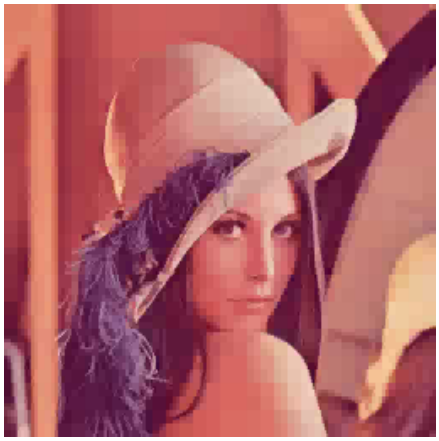


(a) Prag = 0



(b) Prag = 2

(c) Prag = 4

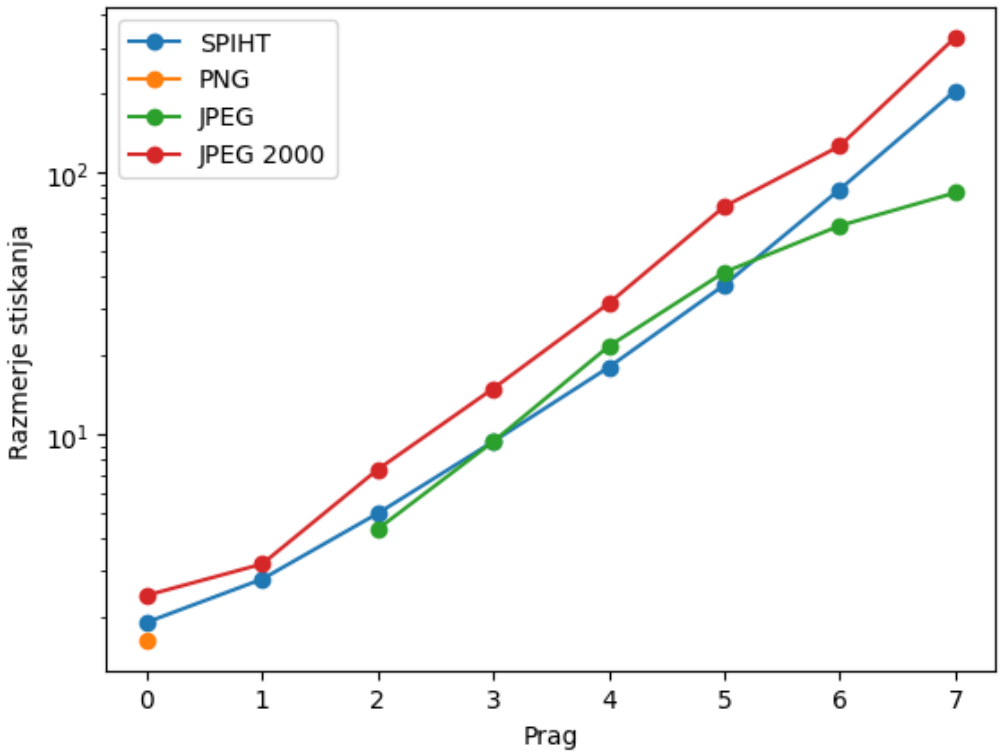


(d) Prag = 5

(e) Prag = 6

(f) Prag = 7

Slika 4.2: Slika Lene, stisnjena pri različnih pragovih



Slika 4.3: Učinkovitost stiskanja za sliko Lene

Tabela 4.1: Rezultati stiskanja za sliko Lene

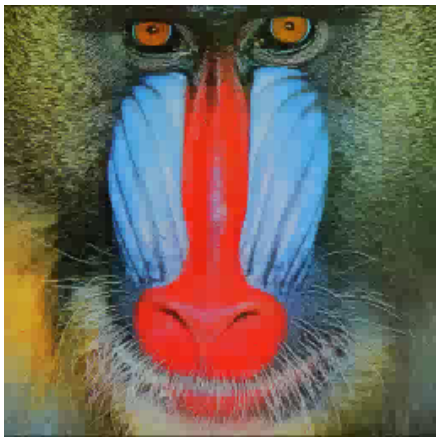
Prag		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SPIHT	**razmerje stiskanja**	1,9	2,8	5,0	9,4	18,0	37,2	86,2	205,0
	SSIM [%]	99,7	99,0	96,6	93,2	88,3	81,1	70,8	59,1
PNG	**razmerje stiskanja**	1,6	/	/	/	/	/	/	/
	SSIM [%]	100,0	/	/	/	/	/	/	/
JPEG	**razmerje stiskanja**	/	/	4,3	9,4	21,7	41,6	62,6	83,7
	faktor kvalitete	/	/	98	90	56	18	8	4
	SSIM [%]	/	/	96,3	93,3	88,2	81,0	72,3	62,2
JPEG 2000	**razmerje stiskanja**	2,4	3,2	7,3	15,0	31,6	74,1	126,2	327,7
	SSIM [%]	99,6	99,1	96,5	93,7	89,1	80,9	74,6	57,7

- Pavijan

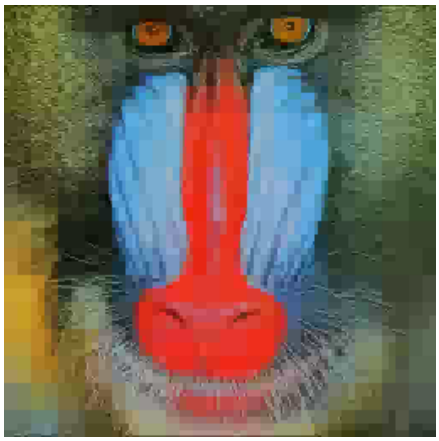
(a) Prag = 0

(b) Prag = 2

(c) Prag = 4



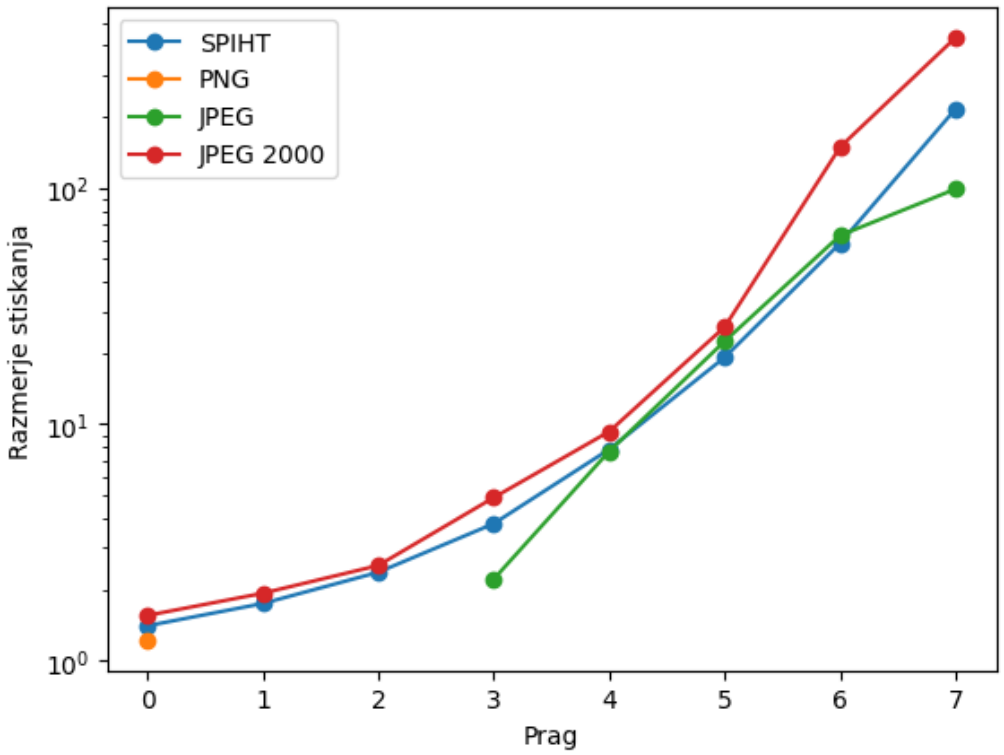
(d) Prag = 5



(e) Prag = 6

(f) Prag = 7

Slika 4.4: Slika pavijana, stisnjena pri različnih pragovih

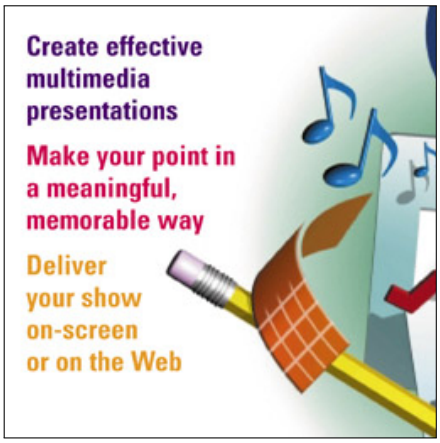


Slika 4.5: Učinkovitost stiskanja za sliko pavijana

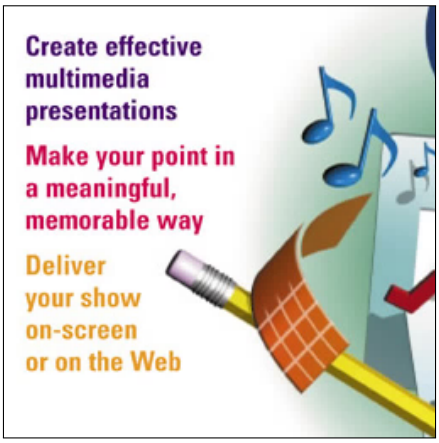
Tabela 4.2: Rezultati stiskanja za sliko pavijana

Prag		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SPIHT	**razmerje stiskanja**	1,4	1,7	2,4	3,8	7,8	19,2	58,3	216,3
	SSIM [%]	99,9	99,7	98,6	94,6	85,3	69,5	46,9	27,6
PNG	**razmerje stiskanja**	1,2	/	/	/	/	/	/	/
	SSIM [%]	100,0	/	/	/	/	/	/	/
JPEG	**razmerje stiskanja**	/	/	/	2,2	7,7	22,4	63,0	99,0
	faktor kvalitete	/	/	/	100	82	26	6	0
	SSIM [%]	/	/	/	91,8	85,3	69,2	46,5	29,3
JPEG 2000	**razmerje stiskanja**	1,6	1,9	2,5	4,9	9,3	25,8	149,1	433,1
	SSIM [%]	99,9	99,7	98,8	94,0	87,7	73,8	37,5	21,5

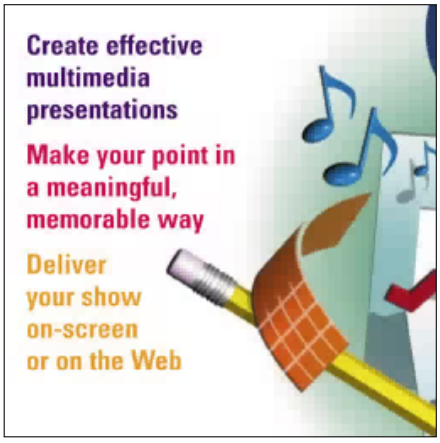
- Besedilo



(a) Prag = 0



(b) Prag = 2



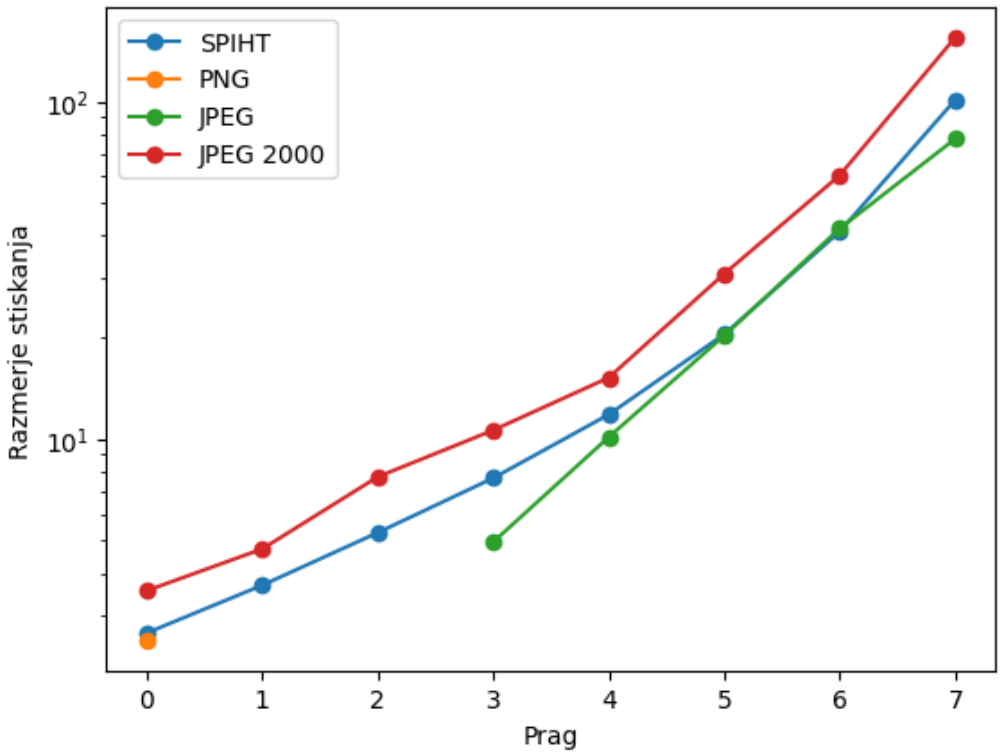
(c) Prag = 4

(d) Prag = 5

(e) Prag = 6

(f) Prag = 7

Slika 4.6: Slika z besedilom, stisnjena pri različnih pragovih



Slika 4.7: Učinkovitost stiskanja za sliko z besedilom

Tabela 4.3: Rezultati stiskanja za sliko z besedilom

Prag		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SPIHT	**razmerje stiskanja**	2,7	3,7	5,3	7,7	11,8	20,4	41,2	101,0
	SSIM [%]	99,9	99,6	98,8	97,5	95,4	91,0	83,2	71,9
PNG	**razmerje stiskanja**	2,5	/	/	/	/	/	/	/
	SSIM [%]	100,0	/	/	/	/	/	/	/
JPEG	**razmerje stiskanja**	/	/	/	5,0	10,1	20,2	42,0	77,5
	faktor kvalitete	/	/	/	98	88	52	12	2
	SSIM [%]	/	/	/	97,5	95,2	90,9	83,1	70,2
JPEG 2000	**razmerje stiskanja**	3,6	4,7	7,7	10,6	15,2	30,9	60,4	154,6
	SSIM [%]	99,8	99,6	98,7	97,6	95,6	90,2	83,1	71,9

- Paprika

(a) Prag = 0

(b) Prag = 2

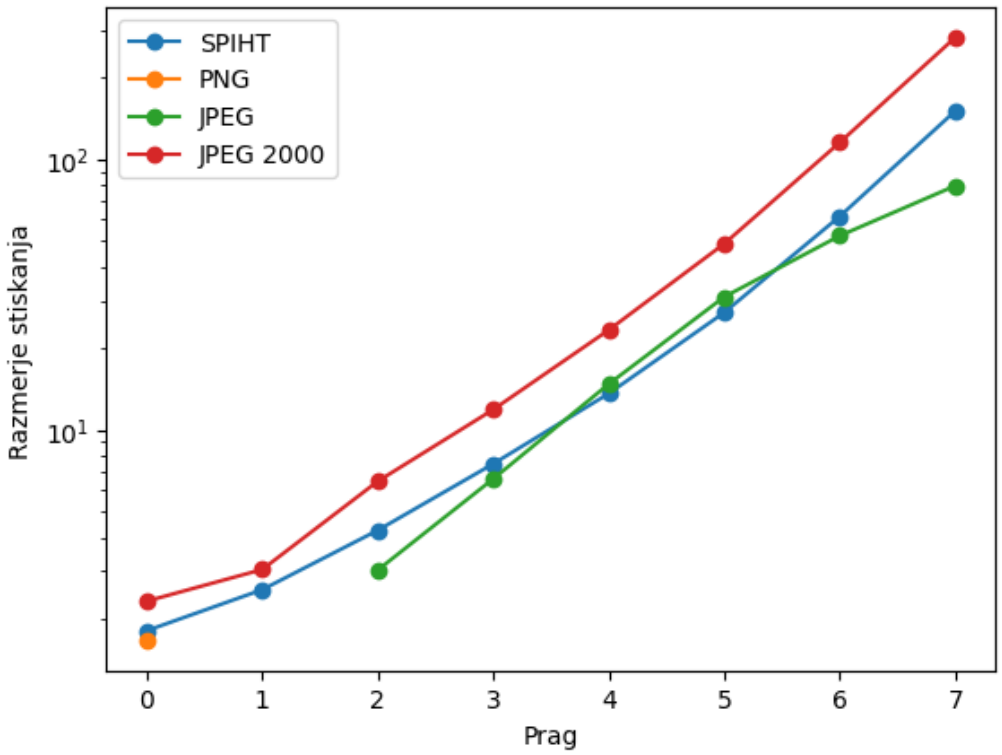
(c) Prag = 4

(d) Prag = 5

(e) Prag = 6

(f) Prag = 7

Slika 4.8: Slika paprike, stisnjena pri različnih pragovih



Slika 4.9: Učinkovitost stiskanja za sliko paprike

Tabela 4.4: Rezultati stiskanja za sliko paprike

Prag		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SPIHT	**razmerje stiskanja**	1,8	2,6	4,2	7,5	13,6	27,1	61,5	150,7
	SSIM [%]	99,7	99,0	96,8	93,1	87,9	80,2	69,9	57,7
PNG	**razmerje stiskanja**	1,7	/	/	/	/	/	/	/
	SSIM [%]	100,0	/	/	/	/	/	/	/
JPEG	**razmerje stiskanja**	/	/	3,0	6,6	14,8	31,0	52,1	79,7
	faktor kvalitete	/	/	100	94	74	26	10	4
	SSIM [%]	/	/	95,7	93,1	87,8	80,3	70,6	57,1
JPEG 2000	**razmerje stiskanja**	2,3	3,0	6,4	11,9	23,4	48,7	115,2	281,7
	SSIM [%]	99,6	99,1	96,6	93,8	89,5	81,6	69,7	54,5

- Metulj

(a) Prag = 0

(b) Prag = 2

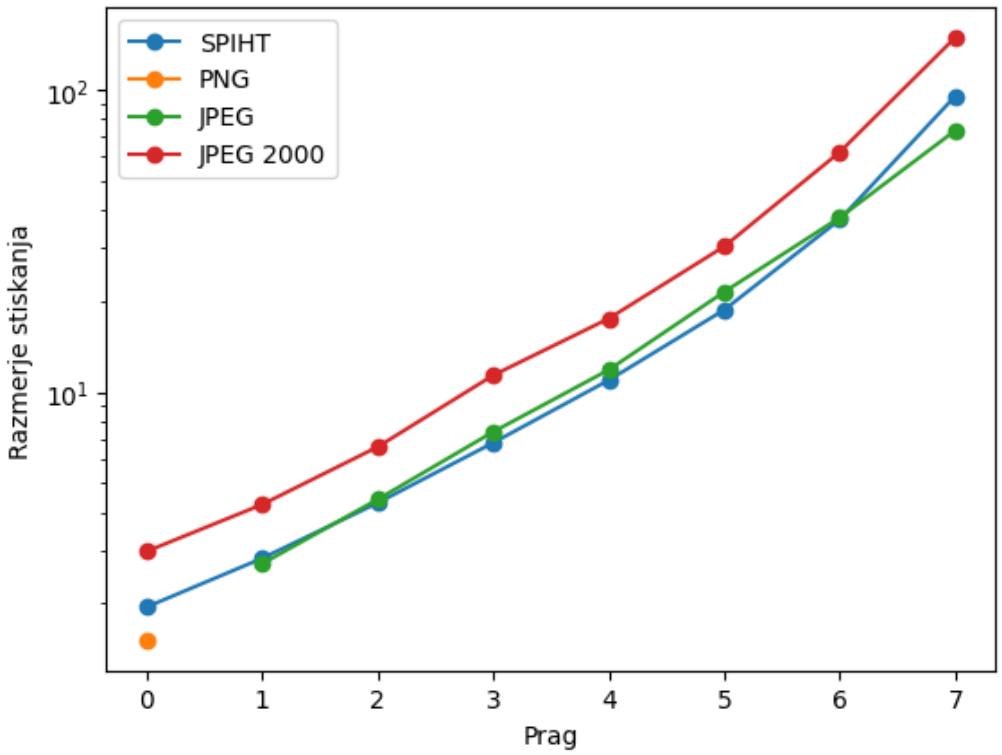
(c) Prag = 4

(d) Prag = 5

(e) Prag = 6

(f) Prag = 7

Slika 4.10: Slika metulja, stisnjena pri različnih pragovih



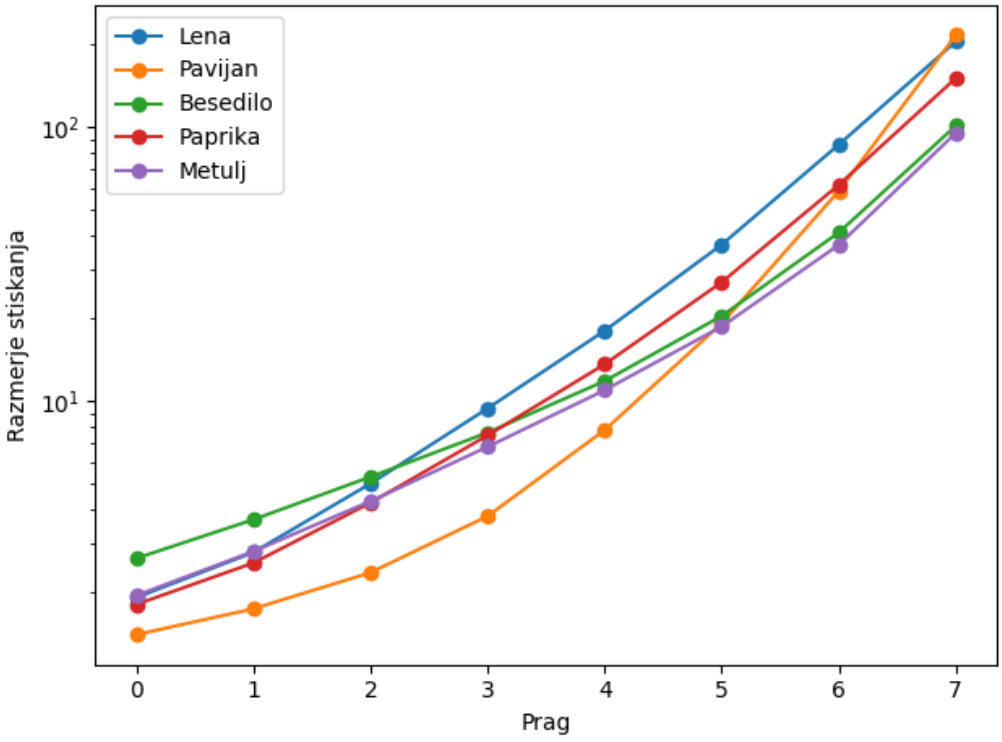
Slika 4.11: Učinkovitost stiskanja za sliko metulja

Tabela 4.5: Rezultati stiskanja za sliko metulja

Prag		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SPIHT	**razmerje stiskanja**	1,9	2,8	4,3	6,8	10,9	18,7	37,1	95,0
	SSIM [%]	99,8	99,4	98,3	96,2	92,9	87,1	77,3	62,1
PNG	**razmerje stiskanja**	1,5	/	/	/	/	/	/	/
	SSIM [%]	100,0	/	/	/	/	/	/	/
JPEG	**razmerje stiskanja**	/	2,7	4,4	7,4	11,8	21,4	37,7	73,1
	faktor kvalitete	/	100	96	88	70	28	10	2
	SSIM [%]	/	99,4	98,4	96,1	92,8	87,1	77,6	60,4
JPEG 2000	**razmerje stiskanja**	3,0	4,3	6,6	11,4	17,5	30,3	62,1	148,4
	SSIM [%]	99,8	99,4	98,5	96,3	93,2	87,4	78,0	62,5

4.2 Diskusija

Na sliki 4.12 vidimo primerjavo učinkovitosti algoritma SPIHT pri različnih vhodnih slikah. Najbolj stisljiva (sicer zgolj pri nižjih pragovih) je bila slika z besedilom, saj je bilo malo podrobnosti (veliko enobarvnega ozadja), s čimer dobimo manjše koeficiente DWT po absolutni vrednosti, kar ugodno vpliva na učinkovitost algoritma. Najslabše stiskanje smo dosegli pri sliki pavijana, saj je bilo ogromno podrobnosti. Je pa res, da si lahko zaradi visokih frekvenc privoščimo stiskanje pri večjem pragu, kjer se učinkovitost stiskanja izboljša v primerjavi z drugimi slikami. Bistvene razlike pri rekonstruirani sliki smo opazili pri pragu 5 oz. 6.



Slika 4.12: Učinkovitost stiskanja pri različnih vhodnih slikah

Pretvorba v barvni prostor YCbCr in Haarova transformacija ustvarita manjše izgube, saj rezultat ni celoštevilski. V (skoraj) brezizgubnem stiskanju je bil SPIHT boljši napram algoritmu PNG. V primerjavi s standardom JPEG smo velikokrat dosegli boljše rezultate, nikoli

pa bistveno slabših. JPEG 2000 je bil v vseh primerih z naskokom najučinkovitejši.

Pri večjih pragovih smo opazili izrazit Machov pojav, ki je posledica Haarove transformacije. Haarov valček je preprost, boljše rezultate bi dosegli z uporabo drugih valčkov (npr. CDF 9/7[16]). Lahko bi tudi, zaradi strukture algoritma, enostavno podvzorčili manj pomembni komponenti krominance Cb in Cr oz. njuni valčni transformiranki, tako da bi upoštevali koeficiente v zgolj 2. kvadrantu matrike koeficientov. V praksi to pomeni, da v algoritmu spremenimo samo iskanje naslednikov SOT, omejimo se na omenjen kvadrant.

5 ZAKLJUČEK

V magistrskem delu smo najprej predstavili diskretno valčno transformacijo, natančneje Haarovo transformacijo, katere koeficiente smo uporabili kot vhod v SPIHT, algoritem za napredujoče stiskanje slik. Razložili smo pojme, potrebne za razumevanje algoritma, in predstavili psevdokod implementacije kodiranja in dekodiranja. Delovanje smo nato prikazali na primeru.

Naredili smo primerjavo s formati PNG, JPEG in JPEG 2000. Za relevantno primerjavo glede učinkovitosti stiskanja smo formate primerjali pri čim bolj podobnih vrednostih metrike SSIM. JPEG 2000 se je v vseh primerih izkazal za najučinkovitejšega. Pri (skoraj) brezizgubnem stiskanju je SPIHT dosegel boljše rezultate kot algoritem PNG, bi pa za še bolj natančno primerjavo morali uporabiti celoštevilsko valčno transformacijo in barvni prostor, ki omogoča popolno rekonstrukcijo. V primerjavi s standardom JPEG, se je SPIHT v veliko primerih izkazal kot boljši ali vsaj primerljiv.

VIRI IN LITERATURA

[1] Wikipedia The Free Encyclopedia, "Image compression." https://en.wikipedia.org/wiki/Image_compression. Zadnji dostop: 1. december 2023.

[2] A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 243–250, 1996.

[3] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing (Second Edition)*. San Diego, California: California Technical Publishing, 1999.

[4] K. Sayood, *Introduction to Data Compression*. San Francisco: Elsevier, 2006.

[5] Wikipedia The Free Encyclopedia, "Haar wavelet." https://en.wikipedia.org/wiki/Haar_wavelet. Zadnji dostop: 1. december 2023.

[6] D. Salomon, *Data Compression: The Complete Reference (Third Edition)*. New York: Springer Science, 2004.

[7] B. Žalik, *RAČUNALNIŠKA MULTIMEDIA: Zapiski predavanj*. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko Univerze v Mariboru, 2021.

[8] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Transactions on Signal Processing*, vol. 41, no. 12, pp. 3445–3462, 1993.

[9] Wikipedia The Free Encyclopedia, "Quadtree." https://en.wikipedia.org/wiki/Quadtree. Zadnji dostop: 1. december 2023.

[10] F. W. Wheeler and W. A. Pearlman, "Spiht image compression without lists," in *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing*, (Istanbul, Turkey), pp. 2047–2050, 2000.

[11] N. Guid, *RAČUNALNIŠKA GRAFIKA: učbenik*. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko Univerze v Mariboru, 2001.

[12] libPNG, "Portable Network Graphics." `http://libpng.org/pub/png/`. Zadnji dostop: 1. december 2023.

[13] JPEG, "Overview of JPEG." `https://jpeg.org/jpeg/index.html`. Zadnji dostop: 1. december 2023.

[14] JPEG, "Overview of JPEG 2000." `https://jpeg.org/jpeg2000/`. Zadnji dostop: 1. december 2023.

[15] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[16] Wikipedia The Free Encyclopedia, "Cohen–daubechies–feauveau wavelet." `https://en.wikipedia.org/wiki/Cohen%E2%80%93Daubechies%E2%80%93Feauveau_wavelet`. Zadnji dostop: 1. december 2023.