Zapiski predavanj pri predmetu

Računalniška multimedia

Pripravil: Borut Žalik

MARIBOR: vinotok 2023

Kazalo

Poglavje 1

Stiskanje rastrskih slik

Rastrske slike (angl. raster images) so predstavljene z matriko diskretnih elementov, ki jim rečemo *piksli* (angl. **PI**cture **X EL**ement). Razvrščeni so v m vrstic in n stolpcev. Izrazu $m \times n$ pravimo ločljivost (angl. resolution). Pikslom v primeru digitalnega videa ali slike iz naprav faksimile pravimo *pels*. Rastrske slike lahko razvrstimo v naslednje skupine [1]:

- **enobarvne (tudi monokromatske) slike** (angl. monochromatic, bi-level); vsak piksel lahko zavzame samo dve vrednosti, to je črno ali belo. Vsak piksel je posledično predstavljen samo z enim bitom.

- **sivinske slike**; vsak piksel ima globino n bitov ($[0, n-1]$), s čimer lahko predstavimo 2^n odtenkov. Bitna globina je kompatibilna z velikostjo zloga (4, 8, 12, 16, 24). Biti na posameznih lokacijah v zlogu tvorijo tako imenovano **bitno ravnino** (angl. bitplane).

- **slike z zveznimi barvnimi toni** (angl. continuous-tone images); takšne slike imajo mnogo zelo podobnih barv, sosednji piksli se v veliki večini primerov zelo malo razlikujejo. Oko tako zelo težko opazi razlike v barvah med sosednjimi piksli. Slike z zveznimi barvnimi toni so slike naravnih podob (fotografije, skanirane slike). Piksel je predstavljen na dva načina, ali z enim samim velikim številom ali pa iz treh barvnih komponent.

- **slike z nezveznimi barvnini toni** (angl. discrete-tone images); predstavljajo umetno-dobljene slike (stran besedila, inženirske risbe, vsebina računalniškega prikazovalnika). Takšne slike lahko vsebujejo zelo veliko različnih barv, prehodi med njimi pa so ostri in močno kontrastni. Za njihovo stiskanje so izgubni postopki neprimerni, saj

slike izgubijo ostrino. Pomembno pa je, da se lahko nek vzorec v sliki ponovi zelo velikokrat.

- **risanke** (angl. a cartoon-like images); slike so barvne, ki pa sestojijo iz velikih področij zapolnjenih z eno barvo, sosednja področja pa lahko imajo zelo različne barve.

Vsaka izmed slik iz omenjenih skupin lahko vsebuje redundantnost, vendar pa se redundanca kaže na različne načine. Zaradi tega ne obstaja neka univerzalna metoda, ki bi bila uspešna za različne vrste slik.

Slika pove več kot 1000 besed pravi star kitajski pregovor. Na žalost pa slike zahtevajo veliko prostora. Barvna slika ločljivost 512×512 pikslov zasede 786.432 zlogov, slika v ločljivosti 1024×1024 pikslov pa zahteva 4-krat več prostora, to je 3.145.728 zlogov. Slike iz današnjih fotoaparatov, slike plakatov, slike ortofoto ali satelitske slike pa so predstavljene z občutno večjim številom pikslov.

Slike praviloma vsebujejo veliko redundantne informacije, vendar, tudi če neka slika ne bi vsebovala popolnoma nič redendance, moramo upoštevati dejstvo, da slike gledajo ljudje. Pri nekaterih vrstah slik lahko človeško oko učinkovito prevaramo z uvedbo izgub.

Vse digitalne slike pridobimo s postopkom diskretizacije. Ta postopek že sam po sebi vnaša izgube, s čimer pa se ljudje zelo hitro sprijaznimo. Diskretizacija vključuje dva koraka:

- **vzorčenje** (angl. sampling), ki zvezen prostor razdeli v manjša področja - piksle in
- **kvantizacija** (angl. quatization), ki priredi vsakemu pikslu celoštevilsko vrednost, ki predstavlja barvo.

Za stiskanje slik obstaja več idej in pristopov:

- **Pristop 1.** Ta pristop je primeren za enobarvne slike. Pri teh slikah velja, da sosednji piksel piksla P teži k temu, da je enak kot P. Zato je v tem primeru smiselna uporaba algoritma RLE (angl. run-lenght encoding). Metoda stiskanja lahko sliko preiskuje po vrsticah, stolpcih ali celo vzorcu cik-cak z željo dobiti čim daljše zaporedje pikslov enake vrednosti. Dolžine teh zaporedij so potem zakodirane s kodo s spremeljivo dolžino (angl. variable-lenght code, VLC).
- **Pristop 2.** Tudi ta pristop je namenjen enobarvnim slikam in razširja Pristop 1. Če ima trenutni piksel barvo c (c je seveda lahko ali bel

ali črn), potem bodo enaki piksli, ki smo jih že videli v preteklosti verjetno imeli enake neposredne sosede. Pristop pogleda n bližnjih sosedov trenutnega piksla in jih obravnava kot n-bitno število. To število imenujemo **zveza/kontekst** piksla (angl. context). V principu je lahko 2^n zvez, za katere lahko pričakujemo, da se bodo pojavljali neenakomerno. Nekatere zveze bodo pogoste, druge pa zelo redke. Kodirnik nato prešteje, kolikokrat se je posamezna zveza za piksle z barvo c že pojavila in ji na podlagi tega priredi verjetnost p. Kot zadnji korak kodirnik uporabi aritmetično kodiranje. Ta pristop uporablja standard JBIG.

- **Pristop 3.** Namenjen je sivinskim slikam z n-bitnimi ravninami. Vsako bitno ravnino bi lahko obravnavali kot enobarvno sliko in vsako izmed njih kodirali s Pristopom 1 ali 2. Ta ideja temelji na predpostavki, da bosta dva sosednja piksla, ki sta si podobna, podobna tudi v večini bitnih ravnin. To pa na žalost ne drži, kar lahko hitro ponazorimo s primerom. Imejmo sivinsko sliko, kjer je bitna globina $n = 4$; torej, slika ima 4 bitne ravnine. Predpostavimo, da sta dva sosednja piksla z vrednostma 0000 in 0001 in da ju obravnavamo kot da sta si podobna. Torej sta si podobna tudi piksla z vrednostima 0111 in 1000. Ti vrednosti pa v nobeni izmed bitnih ravnin nista enaki. Rešitev je v uvedbi drugačne kode, in sicer takšne, da se bitni vrednosti dveh zaporednih celoštevilskih števil i in $i + 1$ razlikujeta za samo en bit. Takšna koda je *zrcalna Grayeva koda* (angl. reflected Gray code), ki jo bomo spoznali kasneje.

- **Pristop 4.** Pristop uporablja zvezo za napoved vrednosti piksla. Zveza piksla so vrednosti nekaterih izmed njegovih sosedov. Obiščemo lahko nekaj sosednjih pikslov piksla P, izračunamo povprečno vrednost A in napovemo, da bo P imel vrednost A. Izračunamo

$$\Delta = P - A. \tag{1.1}$$

V večini primerov bo vrednost $\Delta = 0$, redkeje bo odstopanje od dejanske vrednosti majhno, zelo redko pa bomo napovedali povsem napačno. Δ sedaj zakodiramo z VLC. Eksperimenti z velikim številom slik nam povedo, da je porazdelitev vrednosti Δ blizu Laplaceovi porazdelitvi. Na podlagi tega lahko metoda stiskanja priredi vsaki vrednosti Δ verjetnost, ki jo potem uporabi aritmetični kodirnik.

Povezava piksla lahko sestoji iz samo neposredno sosednega piksla, boljše rezultate pa dobimo, ko vključimo v povezavo več pikslov. Vpliv posameznega piksla lahko nato utežimo glede na razdaljo.

- **Pristop 5.** Poiskusimo najti ustrezno transformacijo za vrednosti pikslov, nato pa zakodiramo transformirane vrednosti (primera sta na primer valčna transformacija ali diskretna kosinusna transformacija). S kvantizacijo transformiranih koeficientov lahko uvedemo izgube in nadziramo stopnjo izgub.

- **Pristop 6.** Barvne slike z zveznimi barvnimi toni pogosto ločimo po barvnih komponentah ter zakodiramo vsako barvno komponento posebej s Pristopom 3, 4, ali 5. Podobna barva pa pogosto ne pomeni podobnih vrednosti pikslov. Poglejmo primer barve, predstavljene s tremi komponentami z bitno globino 4: 1000|0100|0000. V sistemu RGB bi to pomenilo 50% rdeče, 25% zelene in 0% modre.Če imata dva sosednja piksla na primer vrednosti 0011|0101|0011 in 0010|0101|0011 gre zagotovo za zelo podobno barvo. Če pa nabor pikslov obravnavamo kot 12-bitni števili 001101010011 in 001001010011 sta ti števili zelo različni, ker se razlikujeta v bolj pomembnih bitih.

- **Pristop 7.** Slike z nezveznimi barvnimi toni potrebujejo drugačen pristop (primer je, kot smo že omenili, slika namizja računalnika). Namizje sestoji iz ikon in besedila. Vsaka črka in vsaka ikona je področje, ki se lahko na namizju pojavi večkrat. Če je na primer področje B enako področju A, postavimo pri področju B samo kazalec na področje A.

- **Pristop 8.** Temelji na delitvi slike v dele (lahko se prekrivajo ali ne), nato pa stiskamo posamezne dele slike in ne posameznih pikslov. Ta pristop je temelj za različne fraktalne metode. Le te temeljijo na paradigmi samopodobnosti.

Poleg omenjenih pristopov bi lahko našli še kakšnega, manj pogostega, a zato nič manj zanimivega in uporabnega. Takšen primer je uporaba Delaunayeve triangulacije za stiskanje slik.

1.1 Metrike za napake

V primeru postopkov, ki pri stiskanju vnašajo izgube (angl. loosy compression methods), potrebujemo standardizirane metrike za ugotavljanje kako-

vosti rekonstruirane slike napram izvorni sliki. Poznamo več metrik, a najpogostejša je **metrika PSNR** (angl. Peak Signal to Noise Ratio). Manjše vrednosti metrike PSNR pomenijo, da se slika bolj ujema z izvorno sliko. Primeren PSNR pa ne pomeni, da uporabnik na sliki ne bo zaznal sprememb.

Naj bodo P_i piksli izvorne slike in Q_i piksli rekonstruirane slike. PSNR določimo na naslednji način; najprej izračunamo povprečno vrednost vsote kvadratov napak (angl. mean square error)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(P_i - Q_i)^2. \tag{1.2}$$

Koren srednje vrednosti kvadratov napak (angl. root mean square error, RMSE) izračunamo kot

$$RMSE = \sqrt{MSE}, \tag{1.3}$$

s pomočjo katerega potem zračunamo $PSNR$ kot

$$PSNR = 20\log_{10}\frac{max_i|P_i|}{RMSE}. \tag{1.4}$$

Absolutne vrednosti običajno niti ne potrebujemo, saj imajo piksli nenegativne vrednosti. Maksimalna vrednost piksla je odvisna od barvne globine. Pri črnobelih slika je 1, pri sivinskih slikah praviloma 255. Za barvne slike uporabimo komponento svetlosti (angl. luminance). Rezultat je v decibelih. Če je RMSE=0 (enaki sliki), je PSNR=∞ ali bolje rečeno nedefinirano. Če je RMSE=255, je PSNR=0. Vrednosti, ki dajo sprejemljivo sliko, so med PSNR=$[20, 40]$.

Zelo podobna metrikaj je metrika SNR (angl. signal to noise ratio), ki jo zračunamo z enačbo1.5.

$$SNR = 20\log_{10}\frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}P_i^2}}{RMSE}. \tag{1.5}$$

Pri stiskanju slik je pomembno še **razmerje stiskanja** (angl. compression ratio), ki ga definiramo kot razmerje velikosti izvorne in stisnjene datoteke.

$$CR = \frac{P_f}{Q_f}. \tag{1.6}$$

1.2 Zrcalna Grayeva koda

Vsako metodo, ki je primerna za stiskanje črnobelih slik, lahko uporabimo za stiskanje sivinskih slik, če sliko razdelimo na bitne ravnine in vsako bitno ravnino stiskamo posebej. Težavo, ki se pri tem pojavi, smo že omenili. Bitni predstavitvi sosednih vrednosti pikslov se lahko zelo razlikujeta. Primer sta piksla z vrednostima $7 = 0111_2$ in $8 = 1000_2$ v sivinskem sistemu z bitno globino 4. Rezultirajoči biti se razlikujejo prav v vseh bitnih ravninah. Če želimo izkoriščati zvezo med sosednimi piksli, bi potrebovali kodo, kjer bi se bitne predstavitve vrednosti piklsov, katerih vrednosti se razlikujejo za 1, tudi razlikovale samo za en bit. Takšna koda obstaja. Imenujemo jo **zrcalna Grayeva koda** (angl. reflected Gray code - RGC).[1]

Grayevo kodo sestavimo enostavno z naslednjim postopkom:

- Pričnemo z 1-bitno kodo (0,1).
- Dvobitno kodo skonstruiramo tako, da ali na levi ali na desni najprej dodamo 0, potem pa na isti strani se 1. Rezultat je (00, 01) in (10,11).
- Drugo množico prezrcalimo in dobimo 2-bitno kodo RGC (00, 01, 11, 10).
- Kode priredimo zaporednim celoštevilskim vrednostim 0, 1, 2, 3.

Kot vidimo, se sosedne vrednosti kode RGC razlikujejo samo v enem bitu. Sestavimo še trobitno kodo po istem postopku:

- (0, 1);
- (00, 01),
  (10, 11) zrcalimo (11, 10),
  dobimo (00, 01, 11, 10);
- (000, 001, 011, 010),
  (100, 101, 111, 110) zrcalimo (110, 111, 101, 100)
- dobimo (000, 001, 011, 010, 110, 111, 101, 100);

Vidimo, da se tudi prva in zadnja koda razlikujeta za en bit.

[1] Koda se imenuje po Franku Grayu, ki jo je patentiral leta 1953; samo slučaj je, da se nanaša na možnost kodiranja sivinskih slik

1.3 Intuitivne izgubne metode

V nadaljevanju bomo predstavili dve preprosti izgubni metodi, ki pa, žal, ne dajeta najboljših rezultatov.

1.3.1 Podvzorčenje

Podvzorčenje je najenostavnejši način, kako izgubno stisniti sliko. Metoda enostavno odstrani nekaj pikslov. Običajno kodirnik zbriše vsako drugo vrstico in vsak drugi stolpec ter zapiše preostalo četrtino pikslov. Dekodirnik nato vsak piksel, ki ga prebere, nadomesti s štirimi enakimi piksli. Dobljena slika je pogosto slabe kakovosti, saj se pomembni detajli (tanke vodoravne ali navpične slike) lahko povsem izgube. Hitro dobimo ideje, kako sliko izboljšati. Kodirnik lahko, na primer, izračuna povprečje štirih sosednjih pikslov. Na ta način v bistvu ne zbrišemo povsem nobenega piksla. Vendar, dobra izgubna metoda odstrani tiste vrednosti, ki so očem nevidne, ta pristop pa te moči nima.

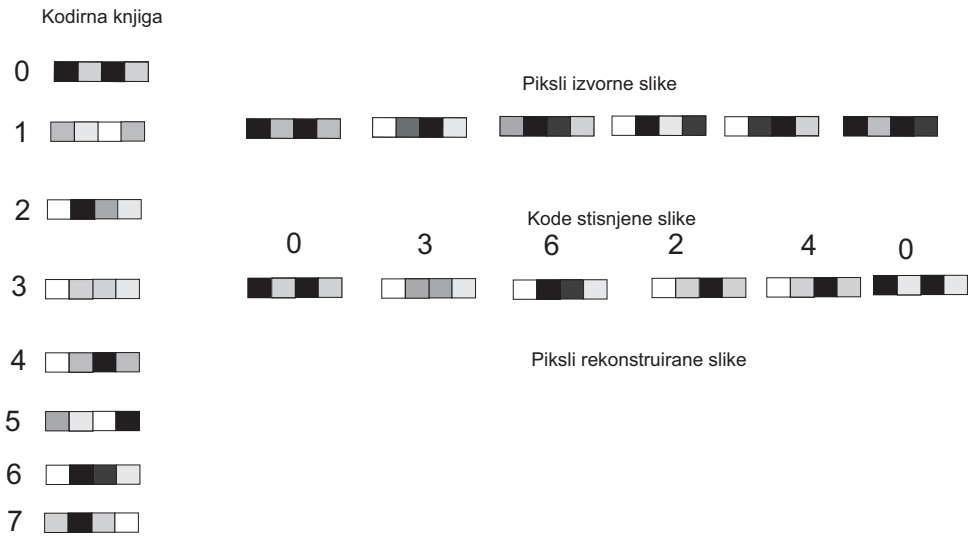
1.3.2 Kvantizacija

Sliko razdelimo v enako velike vektorje pikslov. Kodirnik sestavi nek nabor možnih blokov (ta nabor imenujemo kodirna knjiga) (angl. codebook). Trenutno opazovan blok primerjamo z bloki iz kodirne knjige. Izberemo tisti blok, ki se najmanj razlikuje od trenutno kodiranega in shranimo njegovo kodo (kazalec nanj) (glej sliko 1.1). Dolžina kode mora biti krajša od velikosti bloka, s čimer dosežemo stiskanje.

1.4 Metode, temelječe na transformacijah

Koncept transformacij uporabljamo v različnih področjih, dobro se izkažejo tudi pri stiskanju podatkov z izgubami. Sliko lahko stisnemo, če transformiramo njene piksle, ki so v splošnem med seboj v korelaciji (angl. correlated), tako, da korelacijo odstranimo – sliko dekoreliramo (angl. decorrelation). Stiskanje dosežemo, če so nove vrednosti v povpreču manjše od izvornih. Če transformirane vrednosti kvantiziramo, uvedemo izgube. Dekodirnik uporabi inverzno transformacijo, da rekonstruira (eksaktne ali približne) izvorne podatke. Transformacije, ki takšno rekonstrukcijo zmorejo, so ortogonalne transformacije.[2]

[2]Ortogonalna transformacija je linearna transformacija tako, da velja $A \cdot A^T = I$.



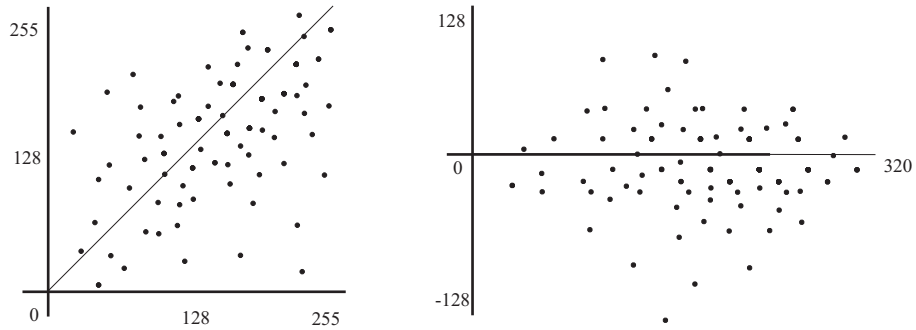
Slika 1.1: Intuitivna vektorska kvantizacija.

Ključen koncept metod, ki temeljijo na transformacijah, je torej dekorelacija, kar pomeni, da transformirane vrednosti postanejo med seboj neodvisne. Posledično lažje sestavimo statistični model. Vsak nabor podatkov, tudi sliko, lahko stisnemo, če so podatki redundantni. Redundanca pri slikah izvira iz korelacije med piksli. Če transformiramo sliko tako, da so njeni piksli dekorelirani, odstranimo redundanco in stisnemo sliko.

1.5 Transformacija parov pikslov

Sivinsko rastrsko sliko preiskujmo s preiskovalno premico po vrsticah in opazujmo pare sosednjih pikslov. Vrednosti, ki jih piksli lahko zasedejo, so iz intervala $[0, 255]$. Ker so sosednji piksli v korelaciji, imata dva piksla (označimo ju z x, y) najpogosteje zelo podobne vrednosti. Pare pikslov sedaj obravnavajmo kot točke (x, y) v ravnini. Vse točke, ki imajo enake vrednosti koordinat x in y ležijo na premici $x = y$ (premica pod naklonom 45^o). Tipičen rezultat vidimo na sliki 1.2a.

Večina pikslov se nahaja v okolici premice $x = y$, le manjši del točk se nahaja dlje od te premice. Točke sedaj zarotiramo za 45^o v sourni smeri okrog koordinatnega izhodišča (slika 1.2b). Iz računalniške grafike poznamo enačbo za rotacijo v 2D prostoru (enačba 1.7):



Slika 1.2: Distribucija parov pikslov predstavljenih kot točke (x, y) v ravnin (a) in njihova rotacija za 45^o v sourni smeri (b)

$$[x^*, y^*] = [x, y] \begin{bmatrix} \cos 45^o & -\sin 45^o \\ \sin 45^o & \cos 45^o \end{bmatrix} = \tag{1.7}$$
$$= [x, y] \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} = [x, y]\boldsymbol{R}.$$

Inverzna transformacija je:

$$[x, y] = [x^*, y^*]\boldsymbol{R^{-1}} = [x^*, y^*]\boldsymbol{R^T} = \tag{1.8}$$
$$= [x^*, y^*] \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}.$$

Očitno je, da bo večina koordinat y enakih 0 ali zelo blizu 0, x koordinate pa se ne spreminjajo veliko.

Za vajo zračunajmo **križno korelacijo** (angl. cross-correlation) točk pred in po rotaciji. Križna korelacija je določena kot vsota produktov koordinat:

$$CCR = \frac{1}{n} \sum_{i=1}^{n} x_i y_i. \tag{1.9}$$

Imejmo 5 točk

(4, 4), (7, 8), (14, 10), (35, 27) in (36,30).

Po rotaciji za 45^o v sourni smeri dobimo:

(5.65, 0), (10.61, 0.71), (16.97, -2.83), (43.84, -5,66) in (46.67, -4.24).

CCR začetnih točk je 447.40, CCR transformiranih točk pa -97.30.

Na koncu transformirane vrednosti pikslov enostavno zapišemo v stisnjen niz. Seveda je smiselno uporabiti kakšno izmed splošnonamenskih metod (Huffmanovo kodiranje, aritmetično kodiranje, stiskanje s slovarjem), s katero dodatno zmanjšamo informacijsko entropijo. Če dopuščamo izgube, potem lahko rotirane vrednosti pikslov še kvantiziramo. Dobra strategija je tudi, če najprej zakodirano koordinato x, nato pa še y. Zaporedje koordinat y ima male vrednosti, pričakujemo lahko tudi zaporedja ničel. Pravimo, da je **varianca** vektorja z vrednostmi y majhna. Varianco imenujemo tudi **energija distribucije pikslov**. Vidimo, da je naša rotacija prenesla večino energije na koordinato x.

Koncentracija energije v eni koordinati ima še eno prednost. Omogoča različno kvantizacijo pikslov glede na koordinato in s tem omogoča boljšo izgubno stiskanje. Ponazorimo to s primerom. Predpostavimo, da začnemo s točko (4, 5), katere koordinati sta si zelo blizu. Po transformaciji dobimo $1/\sqrt{2}(9, 1) \approx (6.36396, 0.7071)$. Energija točke in njene transformacije sta $4^2 + 5^2 = 41 = (9^2 + 1^2)/2$. Če zbrišemo manjšo koordinato točke (to je 4), povzročimo napako $4^2/41 = 0.39$. Če pa zbrišemo vrednost manjše transformirane vrednosti, pa je napaka le $0.707^2/41 = 0.012$.

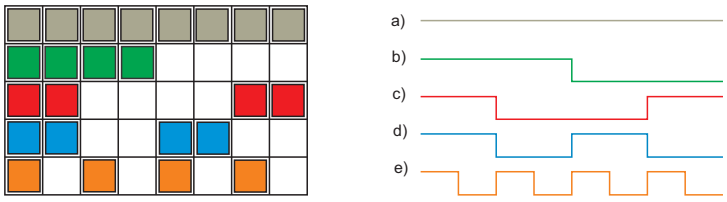
Predstavljeno transformacijo lahko razširimo v poljubno dimenzijo. Namesto parov točk lahko uporabimo trojice, vsaka trojica vrednosti zaporednih pikslov predstavlja točko v 3D prostoru. Tudi te točke tvorijo oblak točk, ki je koncentriran okrog premice, ki gre skozi koordinatno oglišče. To premico nato z dvema zaporednima rotacijama postavimo na eno izmed koordinatnih osi. Rotacijska matrika je dimenzije 3×3 in je seveda ortogonalna. Transformirane točke razdelimo v tri koeficiente, od katerih sta dva majhna.

1.5.1 Ortogonalne transformacije

Nalogi transformacij nad slikami sta naslednji:

- z zmanjšanjem števila bitov za opis večine pikslom želimo zmanjšati redundanco;
- identificirati manj pomembne dele slike z določitvijo frekvečnega prostora slike.

Seveda se postavi vprašanje, ali lahko pri slikah govorimo o frekvencah. Frekvenco vedno povežemo s signali oziroma z valovanjem. Če sliko razdelimo, na primer po vrsticah, se izkaže, da lahko tudi zaporedje pikslov predstavimo kot valovanje. Primer vidimo na sliki 1.3:



Slika 1.3: Zaporedje pikslov v vrstici lahko obravnavamo kot valovanje

Prva vrstica vsebuje samo piksle ene barve, zato ji priredimo signal s frekvenco 0. Piksli v naslednji vrstici se iz zelenih spremenijo v bele, kar ustreza signalu (b) na sliki 1.3. Podobno ugotovimo tudi za naslednje vrstice pikslov. Obravnavanje slike s frekvencami nam omogoča določiti pomembnejše dele slike (ti izkazujejo nizke frekvence) od manj pomembnih, ki so okarakterizirane z višjimi frekvencami. Tako je smiselno, da visoke frekvence kvantiziramo močno, nizke frekvence pa šibko ali sploh ne. Ta način uporablja več zelo učinkovitih tehnik (na primer standard JPEG).

Slike imajo veliko pikslov, zato transformacija ne sme biti zahtevna. Najučinkovitejša je linearna transformacija, pri kateri je vsaka transformirana vrednost c_i določena z uteženo vsoto pikslov d_i, ki bodo transformirani, pri čemer pa je vsak piksel pomnožen z utežjo w_{ij}. Za $n = 4$ dobimo naslednjo enačbo:

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix}. \tag{1.10}$$

Enačbo 1.10 krajše zapišemo kot $C = W \cdot D$, kjer vsako vrstico matrike W imenujemo *bazni vektor* (angl. basis vector). Glede na to, da so vrednosti

pikslov d_i znane, moramo ustrezno postaviti uteži w_{ij}. Vodila za njihovo postavitev sta naslednji:

- **Zmanjšanje redundance.** Prvi transformirani koeficient c_1 je lahko velik, ostali koeficienti $c_2, c_3, \cdots$ naj bodo majhni.
- **Izoliranje frekvenc.** Prvi koeficient c_1 mora ustrezati osnovni frekvenci, ostali koeficienti pa višje harmonskim frekvencam.

Ključ za določitev uteži je dejstvo, da so vrednosti D vrednosti pikslov, ki so nenegativna in med seboj soodvisna (v korelaciji). Glede na prej omenjeno, bo c_1 velik, če bodo uteži w_{1j} pozitivne. Da bi bile ostale vrednost c_i majhne, je dovolj, da je pol uteži w_{ij} pozitivnih, pol pa negativnih. Najenostavnejša izbira je, da jim priredimo vrednost 1 oz -1. Uteži w_{ij} morajo s povečevanjem indeksa i ustrezati višjim frekvencam. Za osnovno frekvenco bodo vse vrednosti w_{1j} postavljene na $+1$, pri w_{2j} bo prva polovica uteži pozitivna, druga pa negativna. Z razmišljanjem nadaljujemo do zadnje vrstice uteži w_{nj}, ki ustreza najvišji frekvenci, če se pozitivne in negativne vrednosti uteži izmenjujejo $(+1, -1, +1, -1, \cdots, +1, -1)$. Za naš primer štirih vrednosti bi dobili naslednjo matriko uteži:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}, \tag{1.11}$$

ki je ortogonalna. Prvi bazni vektor (prva vrstica $\mathbf{W}$) sestoji samo iz enic, njihova frekvenca je zato enaka 0. Transformirana vrednost bo zato verjetno velika. Vsaka izmed naslednjih vrstic sestoji iz dveh $+1$ in dveh -1. Zato generirajo majhne transformirane vrednosti. Njihove frekvence, merjene s spremembo predznakov enic, pa se višajo.

Transformacijo lahko modificiramo z namenom, da ohranimo energijo podatkovnega vektorja. Vse kar moramo storiti je, da transformacijsko matriko $\boldsymbol{W}$ pomnožimo z 0.5. Produkt $(\frac{\boldsymbol{W}}{2} \times (a, b, c, d))$ ima enako energijo $a^2 + b^2 + c^2 + d^2$ kot je energija podatkovnega vektorja (a, b, c, d). Poglejmo si primer koreliranega vektorja $(5, 6, 7, 8)$, katerega transformirani koeficienti so $(13, -2, 0, -1)$. Vidimo, da je prvi koeficient velik, ostali pa so manjši kot osnovni koeficienti. Energija obeh vektorjev je 174 ($5^2 + 6^2 + 7^2 + 8^2 = 174$, $13^2 + (-2)^2 + 0^2 + (-1)^2 = 174$). Pri tem pa v prvem primeru prva komponenta doprinese le 14 % energije, v drugem pa kar 97 % energije.

Zelo pomembno dejstvo je tudi, da omogoča inverzno transformacijo. Produkt $(\frac{\boldsymbol{W}}{2}) \cdot (13, -2, 0, -1)^T$ rekonsturira originalni vektor.

Nazadnje si oglejmo še najmočnejši potencial transformacije v povezavi s stiskanjem podatkov. Uporabimo matriko $\frac{\boldsymbol{W}}{2}$ za transformacijo vektorja $d = (4, 6, 5, 2)$. Dobimo rezultat $t = (8.5, 1.5, -2.5, 0.5)$. t seveda enostavno transformiramo nazaj v d, vendar nam t sam po sebi nič ne pomaga pri stiskanju. Da bi lahko razmišljali o stiskanju, uporabimo kvantizacijo t. Cilj je dobra rekonstukcija d tudi po močni kvantizaciji t. Oglejmo si primer. Najprej kvantiziramo t v cela števila $(8, 1, -3, 0)$. Po inverzni transformaciji dobimo $(3, 6, 5, 2)$. V naslednjem eksperimentu zbrišimo najmanjši vrednosti v t; $(8.5, 0, -2.5, 0)$. Rekonstruirane vrednosti $(3, 5.5, 5.5, 3)$ so še zmeraj zelo blizu originalnim vrednostim. Vidimo, da je celo zelo enostavna transformacija močno orodje pri izgubnem stiskanju podatkov. Z boljšimi transformacijami seveda dosežemo tudi boljše rezultate.

1.5.2 Dvodimenzionalne transformacije

Podatki naj bodo podani v obliki dvodimenzionalne matrike

$$\boldsymbol{D} = \begin{bmatrix} 5 & 6 & 7 & 4 \\ 6 & 5 & 7 & 5 \\ 7 & 7 & 6 & 6 \\ 8 & 8 & 8 & 8 \end{bmatrix}.$$

Vidimo, da so vhodni podatki visoko korelirani. Potem, ko uporabimo našo transformacijo, dobimo:

$$\boldsymbol{C'} = W\ D = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 5 & 6 & 7 & 4 \\ 6 & 5 & 7 & 5 \\ 7 & 7 & 6 & 6 \\ 8 & 8 & 8 & 8 \end{bmatrix} = \begin{bmatrix} 26 & 26 & 28 & 23 \\ -4 & -4 & 0 & -5 \\ 0 & 2 & 2 & 1 \\ -2 & 0 & -2 & -3 \end{bmatrix}.$$

Vsaka kolona matrike $\boldsymbol{C'}$ je transformiranka kolone iz $\boldsymbol{D}$. Vrhnji elementi vsake kolone v matriki $\boldsymbol{C'}$ je dominanten. Opazimo, da so elementi v vrsticah matrike $\boldsymbol{C'}$ še vedno korelirani. Zato vsako vrstico matrike $\boldsymbol{C'}$ pomnožimo s transponirano matriko $\boldsymbol{W}^T$. Skupna transformacija je torej:

$$\boldsymbol{C} = \boldsymbol{C'} \cdot \boldsymbol{W^T} = \boldsymbol{W} \cdot \boldsymbol{D} \cdot \boldsymbol{W^T} = \boldsymbol{W} \cdot \boldsymbol{D} \cdot \boldsymbol{W}. \tag{1.12}$$

Za naš primer torej dobimo:

$$\boldsymbol{C} = \begin{bmatrix} 26 & 26 & 28 & 23 \\ -4 & -4 & 0 & -5 \\ 0 & 2 & 2 & 1 \\ -2 & 0 & -2 & -3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 103 & 1 & -5 & 5 \\ -13 & -3 & -5 & 5 \\ 5 & -1 & -3 & -1 \\ -7 & 3 & -3 & -1 \end{bmatrix}.$$

Elementi matrike $\boldsymbol{C}$ so dekorelirani. Skrajno levi zgornji element je dominanten in vsebuje večino energije originalne matrike $\boldsymbol{D}$. Elementi v prvi vrstici in v prvem stolpcu so nekoliko večji, ostali elementi pa so manjši. Te elemente lahko kvantiziramo in dosežemo ustrezno stiskanje.

1.5.3 Walsh-Hadamardova transformacija

Walsh-Hadamardova transformacija sicer ne daje izjemnih rezultatov pri stiskanju podatkov, možno pa jo je zelo učinkovito izračunati samo s seštevanjem in odštevanjem ter s pomikom, ki nadomešča deljenje. Matrika H_m Walsh-Hadamardove transformacije je definirana rekurzivno. H_0 je definirana kot matrika 1×1 in je 1 za vse H_m, $m > 0$, velja:

$$\boldsymbol{H}_m = \frac{1}{m} \begin{bmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{bmatrix}. \tag{1.13}$$

Nekaj primerov Walsh-Hadamardovih matrik je:

$$\boldsymbol{H}_0 = 1,$$

$$\boldsymbol{H}_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

$$\boldsymbol{H}_2 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix},$$

$$\boldsymbol{H}_3 = \frac{1}{3}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}.$$

1.5.4 Diskretna kosinusna transformacija

Diskretna kosinusna transformacija (angl. discrete cosinus transform, DCT) je izjemno pomembna pri stiskanju slik, zato jo bomo opisali natančneje. Definirana je z naslednjo enačbo [2]:

$$F(u) = \sqrt{\frac{2}{n}} C(u) \sum_{x=0}^{n-1} f(x) \cos\left[\frac{\pi(2x+1)u}{2n}\right], \tag{1.14}$$

for $u = 0, 1, \ldots, n-1$, kjer

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}} & : \quad u = 0 \\ 1 & : \quad u > 0 \end{cases} \tag{1.15}$$

Vhod je n vrednosti x, izhod pa so koeficienti DCT. Prvi koeficient se imenuje koeficient DC, ostali pa so koeficienti AC. Koeficienti so realna števila tudi, če so vhodne vrednosti cela števila.

Inverzna DCT (IDCT) je definirana kot:

$$f(x) = \sqrt{\frac{2}{n}} \sum_{u=0}^{n-1} C(u) F(u) \cos\left[\frac{\pi(2x+1)u}{2n}\right], \tag{1.16}$$

za $x = 0, 1, \ldots, n-1$, $C(u)$ pa je definiran z enačbo 1.15.

DCT sprejeme korelirane vhodne podatke in skoncentrira energijo v samo nekaj prvih transformiranih koeficientih, in sicer, večina v DC. Začetni koeficienti hranijo pomembne nizke frekvence, ostali koeficienti pa manj pomembne visoke frekvence. Majhne koeficiente lahko kvantiziramo (postavimo jih na vrednost 0), velike koeficiente pa zaokrožimo na najbližjo celo

število. V praksi so koeficienti združeni v množice po n vrednosti, nato pa vsako množico transformiramo individualno. Vendar pa je izbira vrednosti n zelo pomembna. Male vrednosti n (3, 4) vodijo v mnogo množic z majhnim številom koeficientov. Teh koefificentov je premalo za učinkovito kvantizacijo. Pri velikih n pa se pogosto zgodi, da koeficienti niso dovolj korelirani, posledično so vsi koeficienti veliki in jih po kvantizaciji ne moremo postaviti na 0. Poskusi so pokazali, da je zelo primerna vrednost $n = 8$.

Oglejmo si primer. Podatkovni vektor $\boldsymbol{p} = (12, 10, 8, 10, 12, 10, 8, 11)$. V enačbi 1.14 uporabimo $n = 8$, $x = 0, 1, 2, \cdots, 7$, $u = 0, 1, 2, \cdots, 7$. Dobimo naslednje vrednosti:

$(28.63782464, 0.530583677, 0.506163828, 1.676061369,$
$3.268400142, -1.773370092, 0.208230991, -0.431973133)$.

Če nad temi vrednostmi uporabimo IDCT (enačba 1.16), dobimo izvorni vektor znotraj zaokrožitvene napake kot posledica končne aritmetike stroja. Naš cilj je uporabiti kvantizacijo. Če dobljen vektor zaokrožimo, dobimo:

$(28.6, 0.6, 0.5, 1.7, 3.3, -1.8, 0.2, -0.4)$.

Po uporabi IDCT dobimo:

$(12, 0254, 10.0233, 7.96954, 9.93097, 12.0164, 9.99321, 7.94354, 10.6557)$.

Če kvantiziramo koeficiente močneje, na primer:

$(28, 1, 1, 2, 3, -2, 0, 0)$, dobimo

$(12.1883, 10.2315, 7.74931, 9.20863, 11.7876, 9.54549, 7.82865, 10.6557)$.

Če uporabimo še močnejšo kvantizacijo

$(28, 0, 0, 2, 3, -2, 0, 0)$,

še vedno dobimo rekonstruirane podatke, ki se dokaj dobro ujemajo z izvornimi, in sicer:

$(11.236, 9.62443, 7.66286, 9.57302, 12.3471, 10.0146, 8.05302, 10.6842)$.

Ponovimo, da takšno ugodno situacijo dobimo samo v primeru koreliranih vhodnih vrednosti.

Pri delu s slikami potrebujemo 2D DCT, ki je definirana kot:

$$F(u,v) = \sqrt{\frac{2}{mn}} C(u)C(v) \sum_{x=0}^{n-1} \sum_{y=0}^{m-1} f(x,y) \cos\left[\frac{\pi(2x+1)u}{2n}\right] \cos\left[\frac{\pi(2y+1)v}{2m}\right], \quad (1.17)$$

kjer sta $u = 0, 1, \ldots, n-1$ in $v = 0, 1, \ldots, m-1$, $C(u)$ in $C(v)$ pa sta definirana z enačbo 1.15. Slika je razdeljena v neprekrivajoče se bloke velikosti $n \times m$, kjer sta tipično $n = m = 8$. Koeficient $F(0,0)$ spet imenujemo koeficient DC, vse preostale pa koeficiente AC. Inverzna transformacija je definirana kot

$$f(x,y) = \sqrt{\frac{2}{mn}} \sum_{u=0}^{n-1} \sum_{v=0}^{m-1} C(u)C(v)F(u,v) \cos\left[\frac{\pi(2x+1)u}{2n}\right] \cos\left[\frac{\pi(2y+1)jv}{2m}\right]. \quad (1.18)$$

DCT uporabljamo najpogosteje za izgubno stiskanje. Najbolj znan postopek je JPEG, kjer kvantiziramo koeficiente AC.

Poglavje 2

Standard JPEG

V poznih 70-tih in v začetku 80-tih let so se začele raziskave na področju novih tehnik za stiskanje slik z namenom povečati stopnjo stiskanja. V poznih 80-tih so že izdelali prve komercialne programe za obdelavo slik na namiznih računalnikih, v večini kot dodatne koprocesorske kartice za postaje UNIX in Macintosh. Koprocesorske kartice so dosegale visoke stopnje stiskanja brez vidnega zmanjšanja kakovosti slike, vendar niso bile ne strojno ne programsko združljive.

Na drugi strani so bile vse sile usmerjene v razvoj mednarodnega standarda, ki bi vključeval nove možnosti stiskanja neodvisno od strojne platforme. Pri razvoju standarda sta sodelovali dve organizaciji za standardizacijo: CCITT (franc. Comite Consultatif Internationale de Telegraphique et Telephonique) in ISO (angl. International Standard Organization"), ki sta leta 1986 skupaj ustanovili skupino z imenom **Joint Photographic Experts Group** ali krajše **JPEG**. Leta 1991 je nastala specifikacija JPEG za stiskanje z izgubami in brez izgub. Specifikacija ima oznako *ISO/IEC 10918-1* oziroma *CCITT Rec. T.81*.

JPEG definira samo zaporedje korakov za njihovo stiskanje, ne opisuje pa dodatnih podatkov o slikah, kot so ločljivost, barvni prostori ali prepletanje vrstic. Za te stvari poskrbi grafični datotečni format JFIF ("JPEG File Interchange Format"). Več o JFIF najdemo v [3].

Ločimo štiri vrste stiskanja JPEG:

- zaporedno stiskanje ("baseline-sequential") vsako barvno komponento stisne v enem prehodu slike (z leve proti desni in od zgoraj navzdol),
- napredujoče ("progressive") stisne sliko v večih prehodih tako, da vsak naslednji vsebuje več informacij. Pri ogledu najprej vidimo grobo,

zamegljeno sliko, nato pa se njena kvaliteta z vsakim prehodom izboljšuje. Uporabnik tako hitro dobi osnovne informacije o sliki in lahko proces nalaganja prekine. Slabost tega pristopa je v tem, da mora kodirnik za vsak prehod opraviti vse korake stiskanja, zato je ta način počasen.

- brez izgub ("lossles") ohrani vse informacije o sliki, zato dosega nižja razmerja stiskanj (povprečno 2:1 pri srednje kompleksnih posnetkih).

- hierarhično ("hierarchial") shrani sliko v večih ločljivostih, pri čemer za prikaz slike nižje ločljivosti ni potrebno dekodirati podatkov v višjih ločljivostih. Vsak posnetek višje ločljivosti uporablja podatke iz posnetkov nižje ločljivosti, zato je skupna količina informacij manjša od shranjevanja vsakega posnetka ločeno. Vsak hierarhični del lahko uporablja napredujoče stiskanje. Hierarhični pristop je uporaben v primerih, ko želimo sliko visoke ločljivosti uporabiti na napravi z nižjo ločljivostjo (mobilni telefoni, ikone).

2.1 Zaporedno stiskanje JPEG

Na kratko si najprej poglejmo osnovne korake zaporednega stiskanja JPEG:

1. Barvne slike pretvorimo iz barvnega prostora RGB v prostor, ki temelji na svetlosti/luminanci in barvitosti/krominanci (ponavadi YUV ali $YCbCr$ [4]). Ker je človeško oko občutljivo za majhne spremembe v svetlosti, ne pa tudi v barvitosti, lahko podatke o barvitosti stisnemo močneje brez večjega vpliva na kvaliteto slike. Ta pretvorba sicer ni obvezna, ker v nadaljevanju obdelujemo vsako barvno komponento posebej, vendar komponente RGB ne dovoljujejo večjih izgub, zato je dosežena stopnja stiskanja nižja. Pri sivinskih slikah ta korak preskočimo.

2. Obema komponentama barvitosti (U, V) lahko zmanjšamo ločljivost, tako da ju vzorčimo v razmerju 2:1 v vodoravni in navpični smeri (vzorčenje 4:1:1) ali v razmerju 2:1 v vodoravni smeri in v razmerju 1:1 v navpični smeri (vzorčenje 4:2:2). S tem zmanjšamo velikost slike na $\frac{1}{2}$ (pri vzorčenju 4:1:1) ali $\frac{2}{3}$ (pri vzorčenju 4:2:2) prvotne velikosti. Pri sivinskih slikah ta korak vedno preskočimo.

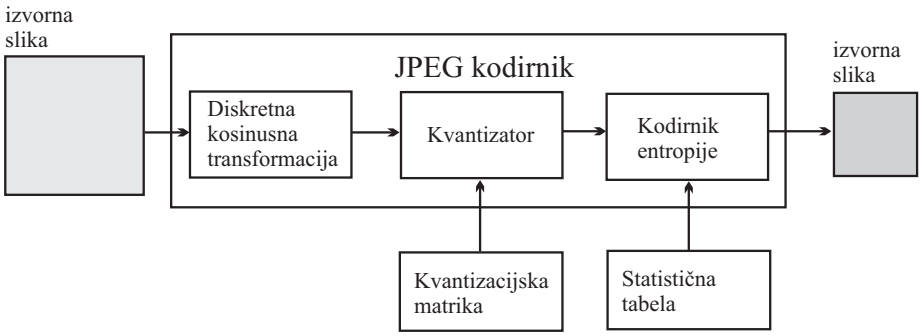
3. Za vsako barvno komponento tvorimo skupine po 8×8 pikslov, ki jih imenujemo podatkovne enote (angl. data units). Če število vrstic ali

stolpcev izvorne slike ni deljivo z 8, podvojimo piksle spodnje vrstice oziroma desnega stolpca. Nato nad vsako podatkovno enoto izvedemo diskretno kosinusno transformacijo (DCT), s katero izračunamo 64 frekvenčnih komponent dela slike, ki ga zajema trenutna podatkovna enota. Te frekvence predstavljajo povprečno barvo piksla v podatkovni enoti (osnovno frekvenco) in nadaljnje višje frekvence. Pri izračunu DCT zaradi omejene računske natančnosti računalnikov izgubimo nekaj informacij o sliki, vendar je ta izguba v realnosti zanemarljiva.

4. Vsako od frekvenčnih komponent v podatkovni enoti delimo z ustreznim kvantizacijskim koeficientom (angl. quantization coefficient) in rezultat zaokrožimo na najbližje celo število. S tem korakom izgubimo del slikovnih informacij, ki vizualno niso zelo pomembne. Vsak od 64 kvantizacijskih koeficientov je parameter $JPEG$, vendar se ponavadi uporabljajo kar tisti, ki jih priporoča standard.

5. V zadnjem koraku zakodiramo 64 kvantiziranih celoštevilčnih frekvenčnih koeficientov vsake podatkovne enote s kombinacijo RLE in Huffmanovega ali aritmetičnega kodiranja.

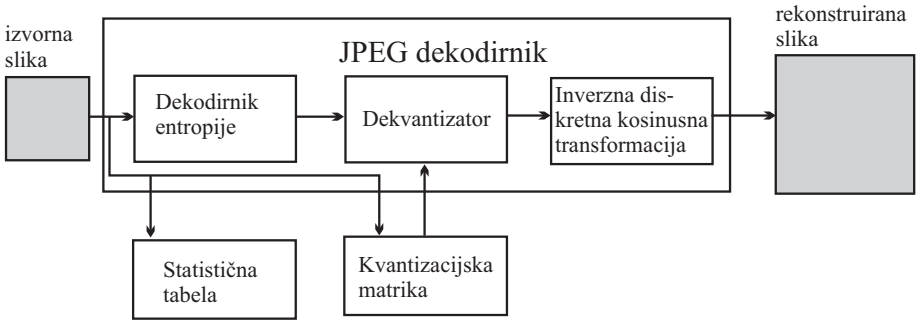
Sliki 2.1 in 2.2 prikazujeta blokovno shemo obveznih delov kodirnika in dekodirnika JPEG.



Slika 2.1: Obvezni deli kodirnika JPEG

Kodiranje JPEG slik ima naslednje korake:

- pretvorba barvnih prostorov,
- zmanjšanje ločljivosti,
- bitni premik bloka oziroma zmanjšanje za 128,



Slika 2.2: Dekodirnik JPEG

- diskretna kosinusna transformacija (DCT),
- kvantizacija koeficientov,
- tvorba zaporedja cik-cak,
- tvorba vmesnega zaporedja simbolov,
- kodiranje entropije in tvorba binarnega zaporedja,
- vpis binarnih (bitnih) podatkov na podatkovni medij.

2.1.1 Pretvorba barvnih prostorov

Naš vid je mnogo bolj občutljiv na zaznavo svetlosti kot na barve. Če je svetlost majhna, barv veče ne razločimo; vidimo samo silhuete objektov. Zato pri stiskanju barvnih slik (in predvsem videa) pogosto opravimo pretvorbo iz barvnega sistema RGB v barvni sistem YUV ali YCbCr. Prvi je nastal za analogno televizijo, drugi pa za digitalno. Ključna komponenta pri obeh je komponenta svetlosti Y (angl. luminance), U in V (oz. Cb Cr) pa sta komponenti barvitosti (angl. crominance). Pretvorbe med prostori RGB in YCbCr vidimo v razpredelnicah 2.1 in 2.2.

2.1.2 Zmanjševanje ločljivosti

Z zmanjšanjem ločljivosti lahko zelo zmanjšamo količino informacij o sliki. Ker je komponenta Y ključna za našo barvno zaznavom zmanjšujemo ločljivost samo za komponenti barvitosti. Pri vzorčenju 4:1:1 štiri sosednje piksle opišemo s štirimi vrednostmi Y in s po eno vrednostjo U in V,

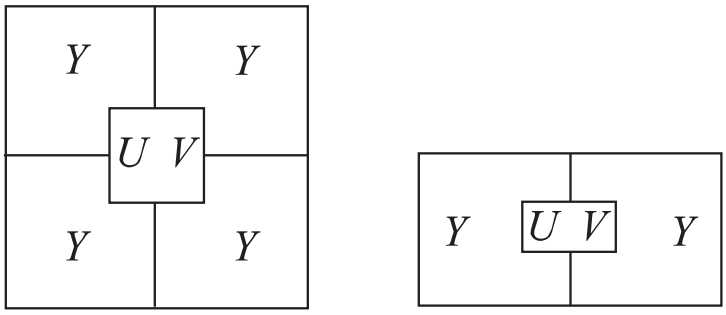
$Y = (77/256)R + (150/256)G + (29/256)B$
$Cb = -(44/256)R - (87/256)G + (131/256)B + 128$
$Cr = (131/256)R - (110/256)G - (21/256)B + 128$
$R = Y + 1.371(Cr - 128)$
$G = Y - 0.698(Cr - 128) - 0.336(Cb - 128)$
$B = Y + 1.732(Cb - 128)$

Tabela 2.1: Pretvorbe med barvnimi prostori RGB in YCbCr

$Y = 0.299R + 0.587G + 0.114B$
$U = -0.1687R - 0.3313G + 0.5B + 128$
$V = 0.5R - 0.4187G - 0.0813B + 128$
$R = Y + 1.402(V - 128)$
$G = Y - 0.34414(U - 128) - 0.71414(V - 128)$
$B = Y + 1.772(U - 128)$

Tabela 2.2: Pretvorbe med barvnimi prostori RGB in YUV

pri čemer vrednosti za U in V predstavljata povprečni vrednosti komponent barvitosti teh štirih pikslov (slika 2.3a). Pri vzorčenju 4:2:2 opišemo dva sosednja piksla z dvema vrednostma Y in po eno povprečno vrednostjo za komponenti U in V (slika 2.3b) [5]. Podatki o faktorjih vzorčenja za posamezne barvne komponente so zapisani v glavi datoteke JFIF.



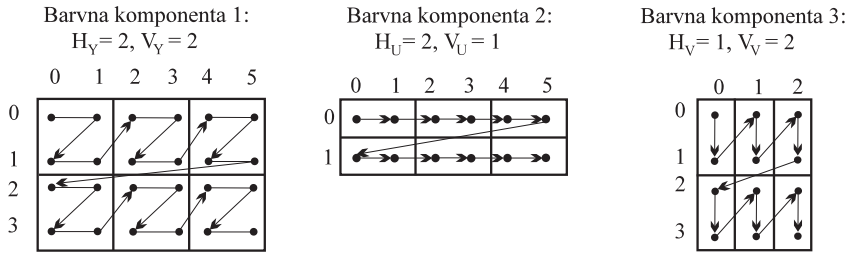
a) b)

Slika 2.3: Zmanjševanje ločljivosti: a) vzorčenje 4:1:1, b) vzorčenje 4:2:2

Osnovna podatkovna enota standarda JPEG je blok velikosti 8×8

pikslov. Vsaka podatkovna enota predstavlja barvno informacijo za eno barvno komponento (Y, U ali V). Postopek stiskanja poteka enako in neodvisno za vsako komponento. V izhodni datoteki so stisnjene podatkovne enote lahko shranjene na dva načina: zaporedoma ali prepleteno. Pri zaporednem shranjevanju si podatkovne enote sledijo od leve proti desni in od zgoraj navzdol, zaporedoma za vsako barvno komponento. Da bi sliko prikazali, moramo najprej dekodirati vsako barvno komponento in šele na koncu postopka dobimo vse slikovne informacije. Prepleteni način je uporabnejši, ker lahko sliko prikazujemo sproti med dekodiranjem. Posamezne komponente so namreč med sabo prepletene, zato za popolno dekodiranje posameznega piksla slike ni potrebno najprej v celoti dekodirati vseh barvnih komponent.

Pri prepletanju barvnih komponent razdelimo podatke vsake komponente na pravokotne regije velikosti $H \times V$ podatkovnih enot (H in V sta vodoravni in navpični faktor vzorčenja za trenutno komponento). Primer prepletanja komponent je na sliki 2.4. V njem nastopajo tri barvne komponente, pri čemer je Y vzorčena s faktorjema $H_Y = 2$ in $V_Y = 2$, ostali dve pa s faktorjema $H_U = 2, V_U = 1$ in $H_V = 1, V_V = 2$.



Slika 2.4: Prepletanje barvnih komponent

Najmanjšo skupino prepletenih podatkovnih enot imenujemo minimalna kodirana enota (angl. minimum coded unit, MCU). Na prikazanem primeru je MCU_1 sestavljena iz zaporedja podatkovnih enot d prve regije komponente Y, ki jim sledijo podatkovne enote prve regije komponente U in podatkovne enote prve regije komponente V (glej enačbo 2.2). Podobno velja za MCU_2. V izhodno datoteko se tako zapiše zaporedje enot MCU, znotraj katerih si zaporedoma sledijo stisnjene podatkovne enote. Standard dovoljuje prepletanje največ štirih barvnih komponent hkrati in največ 10 podatkovnih enot v MCU [6].

$$\begin{aligned} MCU_1 &= d^1_{00}\, d^1_{01}\, d^1_{10}\, d^1_{11}\quad d^2_{00}\, d^2_{01}\quad d^3_{00}\, d^3_{10} \\ MCU_2 &= d^1_{02}\, d^1_{03}\, d^1_{12}\, d^1_{13}\quad d^2_{02}\, d^2_{03}\quad d^3_{01}\, d^3_{11} \\ MCU_3 &= d^1_{04}\, d^1_{05}\, d^1_{14}\, d^1_{15}\quad d^2_{04}\, d^2_{05}\quad d^3_{02}\, d^3_{13} \\ MCU_4 &= d^1_{20}\, d^1_{21}\, d^1_{30}\, d^1_{31}\quad d^2_{10}\, d^2_{11}\quad d^3_{20}\, d^3_{30} \end{aligned} \tag{2.1}$$

Ostale korake JPEG si bomo pogledali s pomočjo primera, povzetega po [7]. Predpostavimo, da je blok 8×8 pikslov tak, kot ga kaže razpredelnica 2.3.

140	144	147	140	140	155	179	175
144	152	140	147	140	148	167	179
152	155	136	167	163	162	152	172
168	145	156	160	152	155	136	160
162	148	156	148	140	136	147	162
147	167	140	155	155	140	136	162
136	156	123	167	162	144	140	147
148	155	136	155	152	147	147	136

Tabela 2.3: Originalni blok 8×8

2.1.3 Zmanjšanje koeficientov za 128

Na začetku se vsi koeficienti bloka 8×8 zamaknejo za en bit v levo oziroma se zmanjšajo za vrednost 128, tako da blok vsebuje števila od -127 do +128 (slika 2.4).

12	16	19	12	12	27	51	47
16	24	12	19	12	20	39	51
24	27	8	39	35	34	24	44
40	17	28	32	24	27	8	32
34	20	28	20	12	8	19	34
19	39	12	27	27	12	8	34
8	28	-5	39	34	16	12	19
20	27	8	27	24	19	19	8

Tabela 2.4: Blok po zmanjšanju koeficientov

2.1.4 Diskretna kosinusna transformacija

Zaradi zahtevnosti računanja diskretne kosinusne transformacije je smiselno velikost bloka omejiti. Standard JPEG uporablja velikost bloka 8×8. Po transformaciji DCT dobimo 64 koeficientov DCT (razpredelnica 2.5). Koeficientu $F(0,0)$ pravimo *koeficient DC*, ostalim 63 koeficientom pa *koeficienti AC*. Pri običajnem bloku 8×8 ima večino koeficientov vrednost 0 ali zelo blizu 0. DCT je transformacija brez izgub in tudi ne opravi stiskanja, podatke samo pripravi za korak, s katerim nadzorujemo izgube, t.j. na kvantizacijo.

185	-17	14	-8	23	-9	-13	-18
20	-34	26	-9	-10	10	13	6
-10	-23	-1	6	-18	3	-20	0
-8	-5	14	-14	-8	-2	-3	8
-3	9	7	1	-11	17	18	15
3	-2	-18	8	8	-3	0	-6
8	0	-2	3	-1	-7	-1	-1
0	-7	-2	1	1	4	-6	0

Tabela 2.5: Blok po diskretni kosinusni transformaciji, kjer so koeficienti zaokroženi na celo število

2.1.5 Kvantizacija

Naslednji korak je kvantizacija 64 koeficientov DCT. Standard predlaga naslednjo enačbo 2.2.

$$Q(i,j) = 1 + (1 + i + j)\ quality \tag{2.2}$$

Parameter *quality* je lahko med 0 in 255. V praksi uporabno vrednosti med 1 in 25.

Kvantizacijo opravimo z deljenjem vsakega koeficienta matrike DCT z ustreznim elementom iz kvantizacijske matrike in rezultat zaokrožimo navzdol (enačba 2.3).

$$F_Q(u,v) = \left\lfloor \frac{F(u,v)}{Q(u,v)} \right\rfloor \tag{2.3}$$

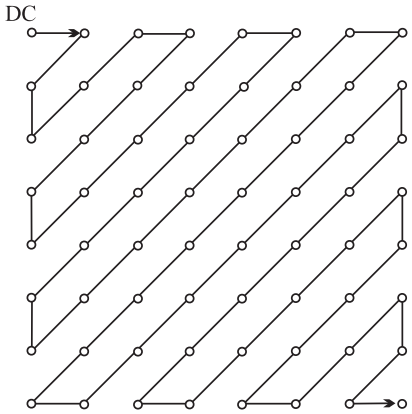
Po kvantizaciji s *quality* = 2 dobimo matriko, ki jo kaže tabela 2.6. Opazimo, da ima večina elementov vrednost 0.

61	-3	2	0	2	0	0	-1
4	-4	2	0	0	0	0	0
-1	-2	0	0	-1	0	-1	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	-1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabela 2.6: Blok po kvantizaciji

2.1.6 Zaporedje $cik - cak$

Matriko iz tabele 2.6 uredimo v tako imenovano zaporedje $cik - cak$, ki je uvod v postopek stiskanja s Huffmanovim kodiranjem. Zaradi veliko vrednosti 0 v matriki so se pri definiciji specifikacije $JPEG$ odločili, da bodo te ničelne vrednosti obravnavali drugače. Slika 2.5 kaže, na kakšen način dobimo zaporedje $cik - cak$.



Slika 2.5: Način tvorbe zaporedja $cik - cak$

Za podani primer bi zaporedje cik-cak imelo naslednjo obliko:

61, -3, 4, -1, -4, 2, 0, 2, -2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1, 0, 0, -1, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, -1, 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.

Prvi element zaporedja cik-cak je koeficient DC, ostali pa so AC-koeficienti.

2.1.7 Kodiranje entropije

Da dobimo bitni zapis kodirane slike, zaporedje cik-cak zakodiramo še z odstranitvijo redundance. Standard JPEG predlaga dve vrsti kodiranja:

- Huffmanovo kodiranje in
- aritmetično kodiranje.

Osnovna specifikacija $JPEG$ uporablja Huffmanovo kodiranje, ki ga podajamo natančneje. Pretvorba v binarno zaporedje s pomočjo Huffmanovega kodiranje ima dva koraka:

- tvorba vmesnega zaporedja simbolov in
- pretvorba vmesnega zaporedja simbolov v binarno zaporedje s pomočjo Huffmanove tabele.

Vsak AC-koeficient je pri tvorbi vmesnega zaporedja simbolov predstavljen v obliki, ki jo kaže razpredelnica 2.7.

simbol_1	simbol_2
(dolžina, velikost)	(amplituda)

Tabela 2.7: Kodiranje koeficientov AC

Dolžina (angl. runlenght) je število ničelnih koeficientov pred prvim neničelnim koeficientom. Vrednosti za dolžino so lahko od 0 do 15, kar pomeni 4 bite v binarnem zaporedju. Če je dolžina večja od 15, potem $simbol\_1$ zapišemo v obliki (15,0), kar predstavlja dolžino 16. V zaporedju simbolov so lahko trije takšni zapisi. Primer:

(15,0) (15,0) (7,4)(12)

Dolžina zaporednih ničel v podanem primeru je 16+16+7. Velikost (angl. size) je število bitov, potrebnih za zapis amplitude. Lahko ima vrednost od

0 do 15, tako da za zapis velikosti porabimo 4 bite. Amplituda je vrednost neničelnega koeficienta v območju [+1024,-1024], kot kaže razpredelnica 2.9. Kot primer vzemimo zaporedje koeficientov AC:

0, 0, 0, 0, 0, 3

Kar lahko zapišemo v obliki:

(5, 2) (3)

kjer je dolžina 5, velikost 2 in amplituda 3. Simbol (0, 0) pomeni konec bloka.

Za *DC*-koeficiente uporabljamo naslednjo obliko zapisa v tabeli simbolov (razpredelnica 2.8):

simbol_1	simbol_2
(velikost)	(amplituda)

Tabela 2.8: Kodiranje koeficientov *DC*

Takšen način kodiranja se uporablja zaradi drugačnega kodiranja *DC*-koeficientov, njihova vrednost je lahko dvakratnik *AC*-koeficientov [-2048, +2048].

Zadnji korak je Huffmanovo kodiranje [8], ki pretvori vmesno zaporedje simbolov v binarno zaporedje. V tem koraku simbole iz vmesnega zaporedja zamenjamo s kodami spremenljivih dolžin; najprej koeficiente DC in nato koeficiente AC.

Za naš primer je vmesno zaporedje simbolov sledeče:

(6)(61), (0,2)(-3), (0,3)(4), (0,1)(-1), (0,3)(-4), (0,2)(2), (1,2)(2), (0,2)(-2), (5,2)(2), (3,1)(1), (6,1)(-1), (2,1)(-1), (4,1)(-1), (7,1)(-1), (0,0)

Kodirano binarno zaporedje s pomočjo Huffmanove tabele pa je naslednje:
1110111101010010010000010001101101101110010111111111011110111010
11111011011100011101101111101001010

Dolžina binarnega zaporedja znaša 98 bitov, kar je tudi končni rezultat kodiranja 8×8 bloka slike za eno barvno komponento. Huffmanova tabela mora biti enaka pri kodiranju in dekodiranju, tvorimo pa jo lahko za vsako sliko posebej ali pa uporabimo eno izmed štirih tabel, ki jih določa standard in

velikost	območje
1	[-1, 1]
2	[-3, -2][2, 3]
3	[-7, -4][4, 7]
4	[-15, -8][8, 15]
5	[-31, -16][16, 31]
6	[-63, -32][32, 63]
7	[-127, -64][64, 127]
8	[-255, -128][128, 255]
9	[-511, -256][256, 511]
10	[-1023, -512][512, 1023]

Tabela 2.9: Razporeditev amplitude za koeficiente AC

sicer, dve tabeli za koeficiente DC in dve za koeficiente AC. Del Huffmanove tabele, ki jo predlaga standard, kaže razpredelnica 2.10 [7].

(dolžina, velikost)	koda
(0, 0)	1010
(0, 1)	00
(0, 2)	01
(0, 3)	100
(1, 2)	11011
(2, 1)	11100
(3, 1)	111010
(4, 1)	111011
(5, 2)	11111110111
(6, 1)	1111011
(7, 1)	11111010

Tabela 2.10: Huffmanova tabela po predlogu ISO

2.2 Dekodiranje slik JPEG

Pri dekodiranju in razširjanju formata JPEG uporabimo enake postopke kot pri stiskanju, le v nasprotnem vrstnem redu, z inverznimi funkcijami. Koraki pri dekodiranju so:

- entropijsko dekodiranje oziroma dekodiranje s pomočjo Huffmanove tabele,
- tvorba matrike (bloka 8×8) iz zaporedja cik-cak,
- dekvantizacija koeficientov,
- inverzna diskretna kosinusna transformacija ($IDCT$),
- bitni premik v matriki oziroma vse koeficiente povečamo za 128.

2.2.1 Razširjanje zaporedja pikslov, tvorba matrike koeficientov DCT in dekvantizacija

Najprej uporabimo entropijski dekoder (s pomočjo Huffmanove tabele), ki nam binarno zaporedje pretvori v vmesno zaporedje simbolov in kasneje v matriko koeficientov DCT. Na tej matriki uporabimo dekvantizacijo s pomočjo enake kvantizacijske matrike, kot smo jo uporabili pri kvantizaciji slike. Zatem uporabimo inverzno kosinusno transformacijo (angl. Inverse Discrete Cosine Transformation, IDCT). Po končani transformaciji $IDCT$ dobimo vrednosti pikslov oziroma njihove komponente začetnega kodiranega bloka 8×8, ki ga še povečamo za 128. Seveda so te vrednosti že približek originalnih vrednosti v odvisnosti od nastavitev pri začetku procesa kodiranja.

V spodnjih matriki je prikazan vpliv kvantizacijskega procesa s faktorjem kakovosti 2:

183	-15	14	0	22	0	0	-17
20	-28	18	0	0	0	0	0
-7	-18	0	0	-15	0	-19	0
0	0	13	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	-17	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabela 2.11: Blok po dekvantizaciji

Če primerjamo sliki 2.5 in 2.11, vidimo, da so bili koeficienti, ki so bliže DC-koeficientu, spremenjeni malo.

2.3 Napredujoči način JPEG

Napredujoči način JPEG je sestavljen iz enakih osnovnih korakov kot običajni sekvenčni način (DCT, kvantizacija, kodiranje). Razlika je v tem, da pri napredujočem načinu vsako barvno komponento zakodiramo v več prehodih, ne samo v enem. V prvem prehodu dobimo grobo, a kljub temu prepoznavno različico slike, ki zasede zelo malo prostora. Naslednji prehodi vsebujejo vedno bolj podrobne detajle. Ponavadi lahko uporabnik prepozna večino posnetka že po dekodiranju samo 5 do 10 % slike.

Blok kvantiziranih koeficientov DCT lahko napredujoče zakodiramo na dva načina:

- s spektralno izbiro (angl. spectral selection): V vsakem prehodu zakodiramo samo določen diagonalni pas koeficientov cik-cak prehoda. V prvem prehodu kodiramo samo koeficiente DC, v naslednjih prehodih pa zaporedoma dodajamo še koeficiente AC.

- z zaporedno aproksimacijo (angl. successive aproximation): Koeficientov ne kodiramo s polno natančnostjo, pač pa v prvem prehodu zakodiramo samo N najpomembnejših bitov (število N izberemo vnaprej). V naslednjih prehodih kodiramo še manj pomembne bite.

Obe metodi lahko uporabljamo ločeno ali v poljubni kombinaciji. Razlike med njima so majhne, a nekateri avtorji vseeno dajejo prednost zaporedni aproksimaciji.

2.4 Hierarhični način JPEG

Hierarhični način omogoča kodiranje slike v več ločljivostih, od katerih je vsaka naslednja za faktor 2 večja od prejšnje v horizontalni, vertikalni ali obeh smereh. Postopek kodiranja poteka po naslednjih korakih:

1. Izvirni sliki zmanjšaj ločljivost za poljuben večkratnik števila 2 v vsaki od želenih dimenzij.

2. Zakodiraj to zmanjšano sliko s sekvenčnim kodirnikom, napredujočim kodirnikom ali kodirnikom brez izgub.

3. Dekodiraj zmanjšano sliko in ji z interpolacijo povečaj ločljivost za faktor 2 v vseh potrebnih dimenzijah.

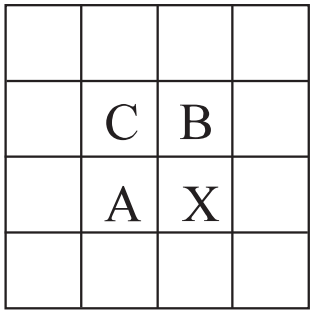
4. Uporabi to povečano sliko kot napoved vrednosti originala pri trenutni resoluciji in izračunaj sliko razlik med povečano sliko in dejanskim originalom z zmanjšano resolucijo. Zakodiraj sliko razlik s sekvenčnim kodirnikom, napredujočim kodirnikom ali kodirnikom brez izgub.

5. Ponavljaj koraka 3 in 4, dokler ne zakodiraš slike pri polni ločljivosti.

Za kodiranje v korakih 2 in 4 lahko uporabimo samo kodiranje z izgubami, samo kodiranje brez izgub, ali pa kodiranje z izgubami, ki mu pri najvišji ločljivosti sledi kodiranje brez izgub.

2.5 Kodiranje JPEG brez izgub

Za kodiranje brez izgub uporablja JPEG preprosto napovedovalno metodo, ki je popolnoma neodvisna od metode z uporabo DCT, opisane pred tem. Število podatkov poskuša zmanjšati z računanjem razlik. Algoritem temelji na združevanju vrednosti do treh sosednjih pikslov A, B in C in tvorbi napovedi vrednosti piksla X, prikazanega na sliki 2.6a. Uporabimo lahko eno od osmih napovedi, zbranih v tabeli na sliki 2.6b. Napovedi 1, 2 in 3 so enodimenzionalne, napovedi 4, 5, 6 in 7 tro oz. dvodimenzionalne. Napoved 0 se uporablja samo pri hierarhičnem pristopu. Izračunano napoved odštejemo od dejanske vrednosti piksla X, razliko pa zakodiramo z eno od metod za kodiranje entropije (Huffmanovo ali aritmetično kodiranje). Na ta način dosežemo stopnjo stiskanja okoli 2:1.



a)

Oznaka	Napoved
0	brez napovedi
1	A
2	B
3	C
4	A+B-C
5	A+((B-C)/2)
6	B+((A-C)/2)
7	(A+B)/2

b)

Slika 2.6: Napoved vrednosti piksla pri kodiranju JPEG brez izgub

2.6 Lastnosti JPEG

Ena od slabosti standarda JPEG je to, da pri vsakem zaporednem stiskanju ponovno izgubimo del informacij. S tem je otežena naknadna obdelava slik, saj je vsaka generacija slike slabše kvalitete. Nekaj preprostih operacij pa lahko vendarle opravimo tudi brez predhodnega razširjenja slike, kot na primer vrtenje za 90^o in obrezovanje slike, vendar pa mora biti slika pravilne velikosti (mnogokratnik velikosti bloka pikslov).

Druga slabost standarda je, da ne omogoča dela s transparentnimi slikami. Glavni razlog za to je stiskanje z izgubami. Če bi ubrali klasični pristop in bi transparentnost v sliki označili z drugače neuporabljeno barvo (kot pri slikah GIF), ta barva po stiskanju ne bi bila več natančno enaka prvotno izbrani. Ponavadi to ni težava, ker majhna sprememba barve piksla ne vpliva mnogo na kvaliteto slike. Če pa se piksel spremeni iz transparentnega v netransparentnega, je to na sliki zelo opazno in moteče. Boljši pristop je uporaba dodatnega barvnega kanala za informacijo o odstotku transparentnosti posameznega piksla (angl. alpha channel). Takšen pristop ni občutljiv na majhne spremembe vrednosti barv. Težava je v tem, da tipični transparentni barvni kanal predstavlja tip slik, pri katerih se algoritem JPEG najslabše odreže (slikovna funkcija ima veliko ploskih območij in nenadnih skokov). Zato bi za njegovo kodiranje morali uporabiti zelo visoko nastavitev kvalitete, kot rezultat pa bi dobili ustrezno veliko datoteko. Možna rešitev bi bila uporaba posebnega kodiranja za transparentni barvni kanal, vendar to v standardu ni predvideno.

Zaradi prevelike kvantizacije se lahko na izhodni sliki pojavijo vidne napake, kot so kockastost slike (meje med 8×8 bloki postanejo vidne) in tanke vzporedne črte. Takšne napake se dajo pred prikazom slike delno zgladiti.

Poglavje 3

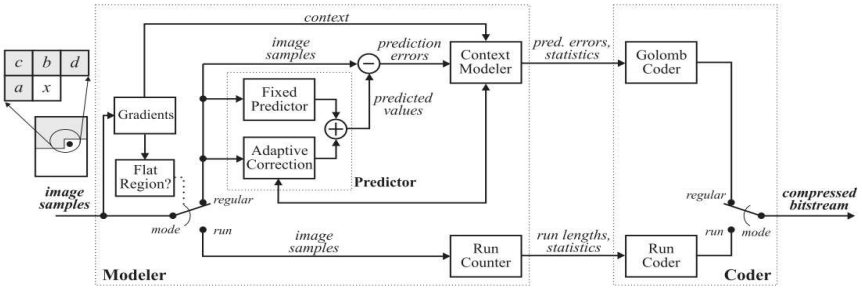
Standard JPEG-LS

Brezizgubni način JPEG se je hitro izkazal za neučinkovitega in posledično ga mnogi niso niti implementirali. Zato se je ISO skupaj z IEC (International Electrotechnical Commission) odločil, da bo razvil nov standard za brezizgubno (in skoraj-brezizgubno (angl. near lossless)) stiskanje. Rezultat je standard ISO/IEC CD 14495 (najdemo ga tudi pod naslednjimi oznakami: ISO/IEC 14495-1:1999, T.87 (1998), izboljšave pa pod ISO/IEC 14495-2:2003 in T.870 (03/2002)), bolj poznan kot JPEG-LS. Kljub temu, da bi po imenu sklepali, da gre samo za razširitev standarda JPEG, pa je ravno nasprotno; gre za povsem novo metodo, ki ne uporablja DCT, ne Huffmanovega/aritmetičnega kodiranja, kvantizacijo pa le v zelo omejeni obliki. JPEG-LS temelji na ideji metode LOCO-I [9]. LOCO-I (angl. Low cOmplexity lossless COmpression of Images), ki so ga uporabljali pri NASI za stiskanje slik, zajetih v vesolju.

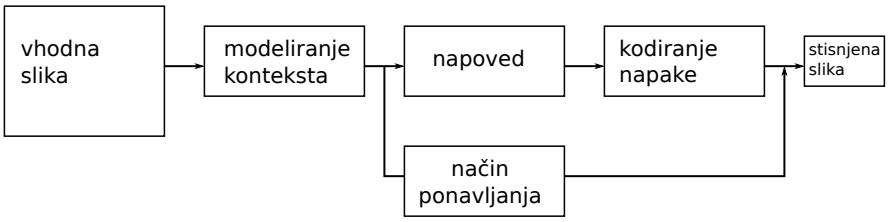
Brezizgubno stiskanje podatkov praviloma sestoji iz dveh različnih in nepovezanih komponent: modeliranja in kodiranja. Kodiranje je danes najpogostejše z aritmetičnim kodiranjem, samo modeliranje pa je zelo odvisno od pristopa in je pogosto zelo zahtevno, tako za implementacijo kot glede časa procesiranja. Cilj standarda JPEG-LS je bil razviti enostavno, računsko nezahtevno in hkrati učinkovito brezizgubno (ali skoraj brezizgubno) stiskanje. Opis JPEG-LS povzemamo po [10, 9, 11].

Arhitekturo kodirnika vidimo na sliki 3.1, njeno poenostavljeno različico pa na sliki 3.2. Hitro opazimo, da kodirnik izbira med dvema načinoma delovanja: med navadnim/običajnim načinom (angl. regular mode) in načinom ponavljanja (angl. run mode).

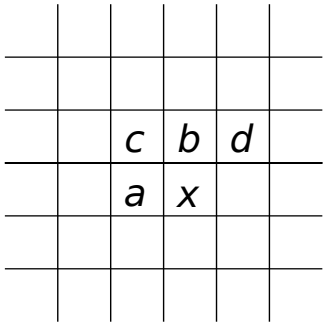
Navadni način temelji na napovedovanju vrednosti trenutno kodiranega piksla. Za napovedovanje uporablja vzorčno okolico (angl. casual negihbour-



Slika 3.1: Arhitektura kodirnika JPEG-LS



Slika 3.2: Poenostavljena arhitektura kodirnika JPEG-LS



Slika 3.3: Vzorčna okolica piksla x

hood) že zakodiranih pikslov (na sliki 3.3 vidimo vzorčno okolico kodiranega piksla x). S piksli iz vzorčne okolice izračunamo vrednost napovedi x_P, ki jo uporabimo za izračun napake e (enačba 3.1).

$$\epsilon = x - x_P \tag{3.1}$$

Vrednost napake izboljšamo z enačbo 3.2, kjer $s_x = \{1, -1\}$ določa predznak,

μ pa je vrednost, s katero korigiramo porazdelitev ostankov.

$$e_x = s_x \, \epsilon - \mu. \tag{3.2}$$

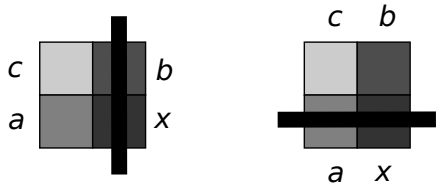
e_x zakodiramo z Golombovim kodiranjem, kjer je Golombov parameter, prilagojen vsakemu izmed 365 različnih kontekstov, ki jih dobimo z redukcijsko funkcijo $\Delta(\Delta_1, \Delta_2, \Delta_3)$. Δ_i, $i \in \{1, 2, 3\}$, izračunamo iz gradientov pikslov vzorčne okolice 3.4:

$$\begin{aligned} \Delta_1 &= d - b \\ \Delta_2 &= b - c \\ \Delta_3 &= c - a \end{aligned} \tag{3.3}$$

Ko je $\Delta_1 = \Delta_2 = \Delta_3 = 0$, vstopimo v področje pikslov enake barve. Kodirnik zato preklopi v način ponavljanja.

3.1 Navadni način JPEG-LS

JPEG-LS uporablja hevristiko za določitev vrednosti napovedi x_P, ki vključuje tudi hevristiko detekcije roba. Če je $c \geq max\{a, b\}$, potem predvidevamo, da piksel x pripada temnejšemu delu vodoravnega ali navpičnega roba[1] (glej sliko 3.4), za napoved pa upoštevamo $x_P = min\{a, b\}$.



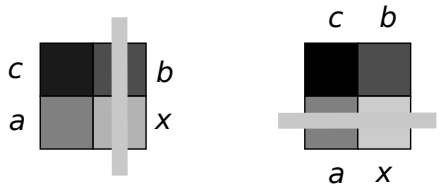
Slika 3.4: Piksel x zelo verjetno pripada temnemu robu

Če je $c \leq min\{a, b\}$, potem je x zelo verjetno del svetle strani vodoravnega ali navpičnega roba (slika 3.5), ustrezna napoved pa je $x_P = max\{a, b\}$.

V primeru, ko je vrednost c med vrednostima a in b, je napovedana vrednost $x_P = a + b - c$. Enačba 3.4 povzema določitev napovedi.

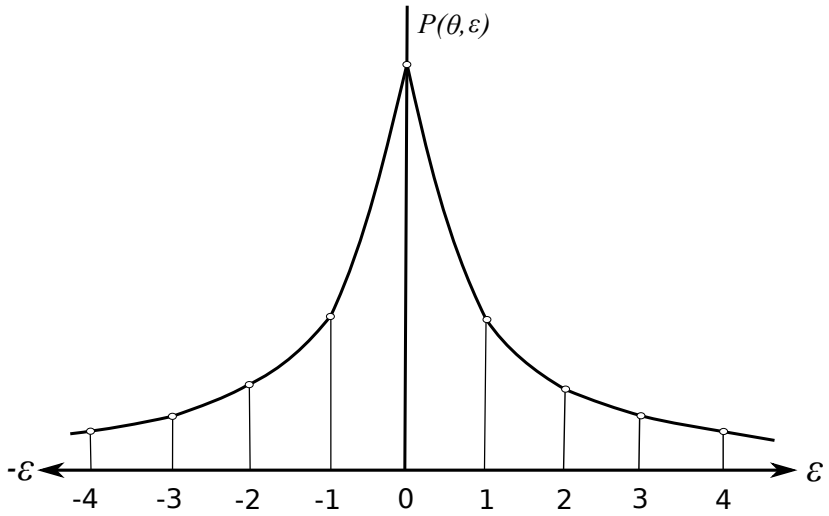
$$x_P = \begin{cases} \min(a, b) & \text{če } c \geq \max(a, b) \\ \max(a, b) & \text{če } c \leq \min(a, b) \\ a + b - c & \text{sicer} \end{cases} \tag{3.4}$$

[1] pri tem velja, da je črni piksel predstavljen z vrednostjo 0, beli pa z 255.



Slika 3.5: Piksel x zelo verjetno pripada svetlemu robu

Napovedano vrednost piksla x_P nato odštejemo od dejanske vrednosti x (enačba 3.2). Pri slikah z zveznimi toni porazdelitev napak dobro modeliramo z dvostrano geometrijsko porazdelitvijo (angl. two-sided geometric distribution - TSGD). Ta je praviloma usredinjena glede na vrednost 0 (slika 3.6), ko je korekcijska vrednost μ iz enačbe 3.2 enaka 0 . Porazdelitveno funkcijo napake formalno zapišemo kot $P(\theta, \epsilon) = \theta^{|\epsilon|}$, $\theta \in (0, 1)$.



Slika 3.6: Geometrijska porazdelitev napak glede na ϵ

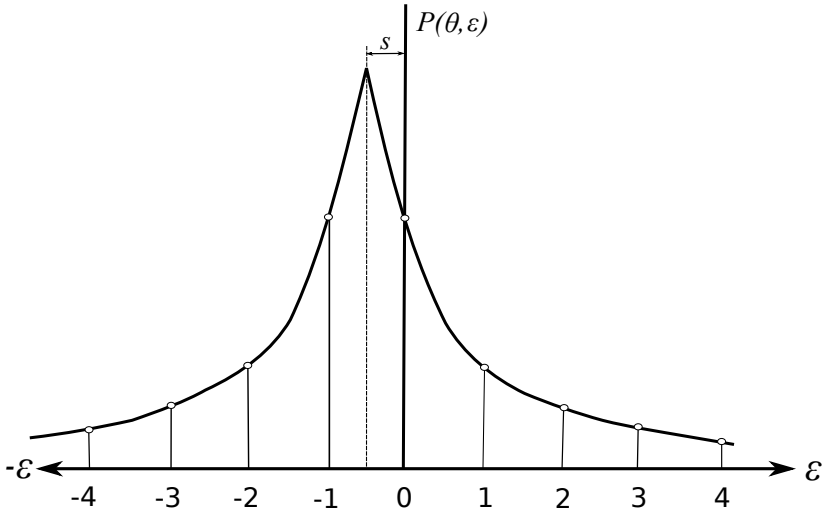
Ker imamo opravka s piksli, je napaka ϵ vedno celoštevilska. Splošnejši model pri JPEG-LS pa dopušča, da napoved korigiramo tudi z realnim številom μ, s katero premikamo vrh funkcije TSGD in s tem prilagajamo TSGD konkretnemu kontekstu. Naša napoved je potem $\epsilon - \mu$. μ razbijemo na dva dela, v celoštevilčni del R (angl. bias) in v necelоštevilčni premik s (angl. shift) tako, da je $\mu = R - s$, $0 \leq s < 1$. Novo porazdelitveno funkcijo napak $P_{(\theta,\mu)}$ potem zapišemo z enačbo 3.5:

$$P_{\theta,\mu} = C(\theta, s)\theta^{|\epsilon - R + s|}, \quad \epsilon = 0, \pm 1, \pm 2, \ldots, \tag{3.5}$$

kjer je $C(\theta, s) = (1 - \theta)/(\theta^{1-s} + \theta^s)$ normalizacijski faktor. Načrtovalci JPEG-LS pa predpostavljajo, da napovedovalnik izniči celoštevilski del R, tako da postane $\mu = s$, porazdelitvena fukcija $P_{(\theta,\mu)}$ pa se poenostavi:

$$P_{\theta,s} = C(\theta, s)\theta^{|\epsilon + s|}, \quad \epsilon = 0, \pm 1, \pm 2, \ldots, \tag{3.6}$$

kjer sta $0 < s, \theta < 1$. TSGD, ki je usredinjena pri 0, ustreza $s = 0$. Če je, na primer, $s = -0,5$, je TSGD premaknjena tako, da ima enake vrednosti pri -1 in 0 (glej sliko 3.7). Podrobno študijo lastnosti TSGD najdemo v [12].



Slika 3.7: Premik TSGD glede za $s = 0,5$

Strmino funkcije TSGD θ in njen odmik s povežemo z vzorčno okolico piksla x preko njihovega konteksta. Pri tem JPEG-LS uporablja množico trikov in poenostavitev. Kot vemo, je vzorčna okolica sestavljena iz štirih pikslov a, b, c in d. Ker lahko vsak piksel zavzame 256 različnih vrednosti, je vseh možnih kontekstov $256^4 = 65.536^2 = 4.294.967.296$. Seveda je to odločno preveč, zato uporabimo redukcijsko funkcijo Δ, ki zmanjša število kontekstov na samo 367. Δ uporablja gradiente Δ_1, Δ_2 in Δ_3, ki smo jih z enačbo 3.4 izračunali pri izbiri načina. Z gradienti dobimo $511^3 =$

133.432.831 različnih vrednosti, kar je še zmeraj ogromno, zato gradiente kvantiziramo. Pri tem uporabimo tri prage T_1, T_2 in T_3, ki so v standardu določeni kot $T_1 = 3$, $T_2 = 7$ in $T_3 = 21$. Gradiente Δ_i kvantiziramo v vrednosti Q_i, $i \in \{1, 2, 3\}$, z enačbo 3.7

$$Q_i = sign(\Delta_i) \begin{cases} 0 & : & \Delta_i = 0 \\ 1 & : & 0 < |\Delta_i| < T_1 \\ 2 & : & T_1 \leq |\Delta_i| < T_2 \\ 3 & : & T_2 \leq |\Delta_i| < T_3 \\ 4 & : & |\Delta_i| \geq T_3 \end{cases} \tag{3.7}$$

V trojčku kvantiziranih vrednosti (Q_1, Q_2, Q_3) ima vsak posamezen Q_i 9 različnih vrednosti na intervalu $[-4, ..., +4]$, kar nam da $9^3 = 729$ različnih vrednosti. Ker pa lahko vse negativne vrednosti Q prezrcalimo preko ničle, dobimo le pozitivne vrednosti Q_i, kar nam da 365 različnih izbir kontekstov. Spremenljivka s_x nam pove, ali je do zrcaljenja dejansko prišlo. Prej smo omenili, da metoda JPEG-LS uporablja 367 kontekstov. Namreč, v načinu ponavljanje uporabljamo posebna konteksta, ki ju bomo opisali kasneje.

Za izračun konteksta JPEG-LS uporablja štiri polja A, B, C in N dolžine 367, prvih 365 zapisov pa inicializirani kot:

- $A = 4$,
- $B = C = 0$
- $N = 1$

Do posameznega elementa teh polj dostopamo preko spremenljivke δ, katere vrednost določimo z algoritmom, ki ga povzema psevdokod 1.
Razložimo še pomen polj:

- $A[\delta]$: vsota absolutnih vrednosti vseh napak pri kontekstu δ,
- $B[\delta]$: vsota vrednosti napak pri kontekstu δ,
- $C[\delta]$: korekcija napovedanih vrednosti pri kontekstu δ,
- $N[\delta]$: število vrednosti pri kontekstu δ.

S poljem C korigiramo napoved kot:

$$x_P = x_P + C[\delta] \tag{3.8}$$

Algorithm 1 Izračun izbranega konteksta pri kodiranju JPEG-LS.

```
1:  function COMPUTE_CONTEX(Q1, Q2, Q3)
2:      if (Q1 < 0) OR (Q1 = 0 AND Q2 < 0) OR
3:         (Q1 = 0 AND Q2 = 0 AND Q3 < 0) then
4:          Q1 = −Q1;
5:          Q2 = −Q2;
6:          Q3 = −Q3
7:          sx = -1
8:      else
9:          sx = 1
10:     end if
11:     δ = 81 ∗ Q1 + 9 ∗ Q2 + Q3
12: end function
```

Vrednost $A[\delta]$ potrebujemo za določitev parametra k Golombovega kodiranja (en. 3.9).

$$k = \left\lceil \log_2 \frac{A[\delta]}{N[\delta]} \right\rceil . \tag{3.9}$$

Pred pričetkom kodiranja preslikamo napako po enačbi 3.10 tako, da dobimo le pozitivne vrednosti, primerne za Golombovo kodiranje.

$$e'_x = \begin{cases} 2e_x, & e_x \geq 0 \\ |2e_x| - 1, & e_x < 0 \end{cases} \tag{3.10}$$

Ko smo zakodirali e'_x, posodobimo vrednosti konteksta δ (en. 3.11):

$$\begin{aligned} N[\delta] &= N[\delta] + 1; \\ A[\delta] &= A[\delta] + |e_x|; \\ B[\delta] &= B[\delta] + e_x; \end{aligned}$$

$C[\delta]$ pa posodobimo s psevdokodom 2, kjer uporabimo ter korigiramo tudi $B[\delta]$.

3.2 Način ponavljanja

V način ponavljanja vstopimo, ko so vsi gradienti $\Delta_i = 0$, torej ko so vrednosti pikslov a, b, c in d enake (glej sliko 3.3). Nato ugotovimo število zaporednih pikslov v vrstici, katerih vrednost je enaka pikslu a. Način ponavljanja ustavimo, ko naletimo na piksel, ki je različen od a, ali pa če

Algorithm 2 Posodobitev kontekstne spremenljivke $C[\delta]$.

```
1: function UPDATE_C
2:     if (B[δ] <= N[δ]) then
3:         B[δ] = B[δ] + N[δ];
4:         if (C[δ] > −127) then
5:             C[δ] = C[δ] − 1;
6:         end if
7:         if (B[δ] <= N[δ) then
8:             B[δ] = −N[δ] + 1;
9:         end if
10:    else
11:        if (B[δ] > 0) then
12:            B[δ] = B[δ] − N[δ];
13:            if C[δ] < 128) then
14:                C[δ] = C[δ] + 1;
15:            end if
16:            if (B[δ] > 0) then
17:                B[δ] = 0
18:            end if
19:        end if
20:    end if
21: end function
```

pridemo do konca trenutne vrstice. Če nismo prišli do konca vrstice, zakodiramo število ponovitev in vrednost piksla, ki je prekinila ponavljanje. V način ponavljanja vstopimo pred primerjanjem trenutnega piksla x z njegovimi sosedi, zaradi česar je lahko dolžina ponavljanja tudi enaka 0.

Pri kodiranju števila ponovitev si pomagamo s števcem i in poljem konstant J, ki vsebuje 32 vrednosti, kot sledi: J ={0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 9, 10, 11, 12, 13, 14, 15}. Število ponovitev kodiramo tako, da zapisujemo bit 1, dokler je število ponovitev večje od $2^{J[i]}$. Po vsakem zapisu število ponovitev zmanjšamo za $2^{J[i]}$ in povečamo števec i, vendar ne preko 31. Če je bilo ponavljanje prekinjeno zaradi konca vrstice, dodamo še bit 1, sicer pa bit 0, ki mu sledi zapis ostanka števila ponovitev. Ostanek zakodiramo na podoben način kot v navadnem načinu. Kot smo že prej omenili, v načinu ponavljanja izbiramo le med dvema kontekstoma, 366 in 367; prvega izberemo, ko se vrednosti pikslov a in b (iz okolice piksla x, ki je prekinil ponavljanje) ne ujemata, sicer pa izberemo drugi kontekst. Za napoved in izračun napake uporabimo enačbi 3.11, e_x pa

nato izračunamo s 3.2.

$$x_P = \begin{cases} a, & a = b \\ b, & a > b \\ 2a - b, & a < b \end{cases} \tag{3.11}$$

Napako napovedi nato, enako kot v navadnem načinu, preslikamo v interval $[-128, 127]$. Nazadnje izračunamo še Golombov parameter k po enačbi 3.12:

$$k = \begin{cases} \left\lceil \log_2 \left\lceil \frac{A[\delta]]}{N[\delta]} \right\rceil \right\rceil; & \delta = 365 \\ \left\lceil log_2 \frac{A[\delta] + \lfloor 0.5N[\delta] \rfloor}{N[\delta]} \right\rceil; & \delta = 366 \end{cases} \tag{3.12}$$

Za kodiranje potrebujemo še zastavico s, ki jo uporabimo pri preslikovanju napake napovedi v pozitivna števila. Pri tem uporabimo dodatna števca, $Nn[365]$ in $Nn[366]$, ki vsebujeta podatek o tem, kolikokrat je bila vrednost napake negativna. Postavitev zastavice s določa enačba 3.13:

$$s_x = \begin{cases} 1; & k = 0, \epsilon > 0, Nn[\delta] < 0.5N[\delta] \\ 1; & \epsilon < 0, Nn[\delta] \geq 0.5N[\delta] \\ 1; & \epsilon < 0, k > 0 \\ 0; & sicer \end{cases} \tag{3.13}$$

Preslikavo napake v e'_x realiziramo z enačbo 3.14:

$$e'_x = 2|\epsilon| - s_x - (\delta - 365) \tag{3.14}$$

Na koncu posodobimo še spremenljivke. Če je ϵ negativen, povečamo števca $N[\delta]$ in $Nn[\delta]$ ter posodobimo $A[\delta]$ kot $A[\delta] = A[\delta] + \lfloor 0.5(\epsilon + 366 - \delta) \rfloor$. Zmanjšamo tudi števec ponovitev i, če je večji od 0.

3.3 Golombovo kodiranje

Golomb je leta 1966 predstavil kodiranje števila ponovitev pri algoritmu stiskanja RLE [13]. Kodiranje je optimalno v primeru geometrijske porazdelitve pojavljanja simbolov, ko je enako učinkovito kot Huffmanovo kodiranje, le da je hitrejše. Golombovo kodiranje ima nastavljiv parameter k, s katerim se prilagajamo strmini funkcije geometrijske porazdelitve. Primeren k najpogosteje določimo eksperimentalno.

Golombova koda nenegativnega celega števila n sestoji iz dveh delov: iz količnika/kvocienta q in ostanka r (enačba 3.15).

$$\begin{aligned} q &= \left\lfloor \frac{n}{k} \right\rfloor, \\ r &= n - q\,k. \end{aligned} \tag{3.15}$$

Kvocient q zakodiramo z unarno kodo, r pa s prisekano/modificirano binarno kodo. Unarna koda (pravimo ji tudi vejična koda) števila q sestoji iz q bitov 1, ki jim sledi vejica, to je bit 0 (vlogo 0 in 1 lahko seveda zamenjamo). Če je $q = 3$, q z unarno kodo zapišemo kot 1110, če je $q = 9$, je unarni zapis 1111111110. Če je $q = 0$, je njegova unarna predstavitev 0.

Prisekana/modificirana binarna koda (angl. truncated binary code) zapiše nenegativno celo število n običajno nekoliko učinkoviteje, kot če bi uporabili klasični binarni zapis. Imejmo abecedo s $p = 5$ simboli {0,1,2,3,4} (kasneje bomo videli, da je p dejansko Golombov parameter k). Za zapis vsakega izmed petih simbolov bi potrebovali $u = \lceil \log_2 p \rceil = 3$ bite. Modificirana binarna koda pa prvih $z = 2^u - p = 2^3 - 5 = 3$ simbolov zapiše z $u - 1$ biti (torej: 00, 01, 10), preostala $p - z = 2$ simbola pa zakodiramo s u biti, in sicer v našem primeru z 111 in 110. Postopek konstrukcije prisekane binarne kode pojasnimo v razpredelnici 3.1. Za primer $p = 10$ je prisekana binarna koda podana v razpredelnici 3.2.

Tabela 3.1: Konstrukcija prisekane binarne kode

n	binarna koda	prisekana binarna koda	komentar
0	000	00	brišemo MSB
1	001	01	
2	010	10	
/	011		ne uporabimo
/	100		ne uporabimo
/	101		ne uporabimo
4	110	110	
5	111	111	

Vrnimo se h konstrukciji Golombove kode. Kot primer imejmo zaporedje $S = \langle A, A, C, B, A, E, A, A, A, B, A, B, C, A, D, A \rangle$ in parameter $k = 2$. Simbolom A, B, C, D, E najprej priredimo zaporedne nenegativne vrednosti, pri čemer dobijo simboli, ki so bolj pogosti, nižjo vrednost. V našem primeru priredimo simbolu A vrednost 0, B vrednost 1, C vrednost 2, D 3, E pa 4 tako, da dobimo zaporedje $N = \langle 0, 0, 2, 1, 0, 4, 0, 0, 0, 1, 0, 1, 2, 0, 3, 0 \rangle$. Postopek določitve Golombovih kod bi bil naslednji naslednji:

Tabela 3.2: Prisekana binarna koda za $p = 10$

n	prisekana binarna koda
0	000
1	001
2	010
3	011
4	100
5	101
6	1100
7	1101
8	1110
9	1111

- Koda za A: $n = 0$, $q = 0$, $r = 0$; vejična koda je v tem primeru predstavljena z bitom 0, prav tako prisekana binarna koda. Golombovo kodo zaradi preglednosti zapišemo v obliki $q : r$, čeprav : seveda ni del kode. Simbol A torej kodiramo kot 0 : 0.
- Koda za B: $n = 1$, $q = 0$, $r = 1$, koda 0 : 1.
- Koda za C: $n = 2$, $q = 1$, $r = 0$, koda 10 : 0.
- Koda za D: $n = 3$, $q = 1$, $r = 1$, koda 10 : 1.
- Koda za E: $n = 4$, $q = 2$, $r = 0$, koda 110 : 0.

Poglejmo še en primer. Za število $n = 23$ določimo Golombovo kodo, če je Golombov parameter $k = 5$. Hitro dobimo $q = 4$ in $r = 3$. Zaloga vrednosti ostanka je 0, 1, 2, 3, 4, kar je naš p. Za predstavitev vseh vrednosti z binarno kodo potrebujemo $u = 3$ bite. S krajšimi kodami zakodiramo prvih $z = 2^u - n = 2^3 - 5 = 3$ vrednosti (torej $r = 0$ zakodiramo kot 00, $r = 1$ kot 01, $r = 2$ kot 10), preostali vrednosti pa s tremi biti (torej $r = 3$ kot 110 in $r = 4$ kot 111). Število 23 pri $k = 5$ zapišemo torej kot 11110:110.

Golombovo kodiranje je namenjeno kodiranju samo nenegativnih števil. Če moramo zapisovati tudi negativna števila, jih moramo preslikati v pozitivna. Običajno negativna in pozitivna števila prepletemo, tako, da negativna števila preslikamo v lihe, pozitivne pa v sode vrednosti (dobimo zaporedje 0, -1, 1, -2, 2, -3, 3, $\cdots$). Naj bo x vrednost, ki jo kodiramo.

Vrednost x', ki jo lahko zapišemo z Golombovim kodiranjem, potem dobimo z enačbo 3.16.

$$x' = \begin{cases} 2x, & x \geq 0 \\ 2|x| - 1, & x < 0 \end{cases} \tag{3.16}$$

Posebni primer Golombovega kodiranja je Riceovo kodiranje [14], ki je podmnožica Golombovega kodiranja, ko je parameter k potenca števila 2.

Poglavje 4

Valčna transformacija

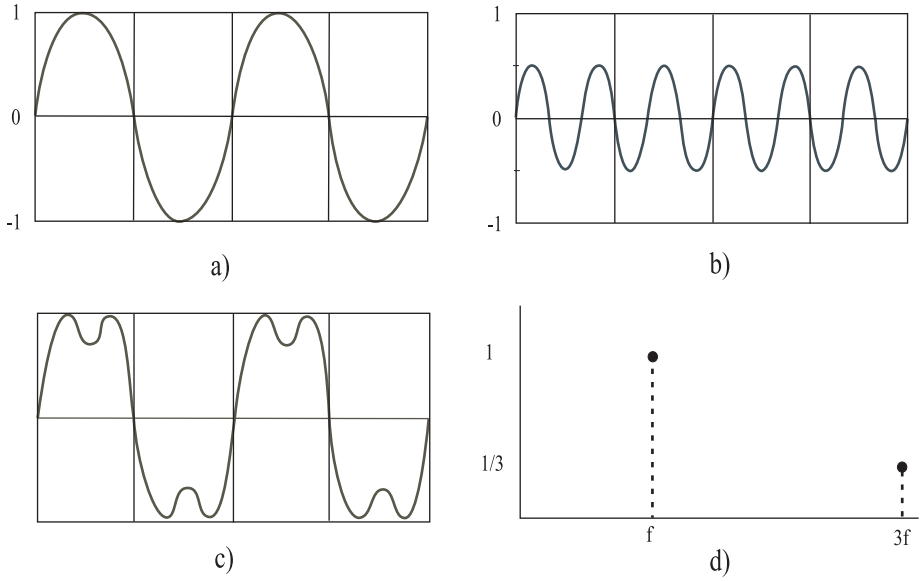
4.1 Frekvenčni prostor

Mnogo funkcij v naravoslovju in tehniki uporablja čas kot parameter. Veliko izmed časovnoodvisnih funkcij se skozi čas ponavlja in so podobne valovanju. Takšnim funkcijam pravimo **periodične funcije**. Funkcija $g(t)$ je periodična, če obstaja neničelna konstanta P tako, da je $g(t + P) = g(t)$ za vse vrednosti t. Najmanjšega izmed P imenujemo periodo funkcije P. Periodične funkcije imajo štiri atribute:

- **amplitudo** - amplituda funkcije je njena maksimalna vrednost v poljubni periodi;
- **periodo** - frekvenca funkcije nam pove število njene ponovitve v časovni enoti (sekundi);
- **frekvenco** - perioda je inverz periode funkcije;
- **fazo** - faza meri položaj funkcije znotraj periode (lahko govorimo o prehitevanju in zaostajanju funkcije). Primer faznega zamika za $\pi/2$ sta funkciji sinus in kosinus.

Velika večina časovnoodvisnih periodičnih funkcij pa ne vsebuje samo ene frekvence. Prav nasprotno, $g(t)$ je lahko sestavljena iz množice različnih frekvenc. Te frekvence določajo različne lastnosti $g(t)$. Frekvenčno vsebino funkcije $g(t)$ dobimo z njeno transformacijo v frekvenčni prostor. Transformiranko časovne funkcije $g(t)$ v frekvenčnem prostoru označimo z $G(f)$. Izkaže se, da je transformacija funkcije iz časovnega v frekvenčni prostor in obratno enostavna, če je funcija $g(t)$ periodična.

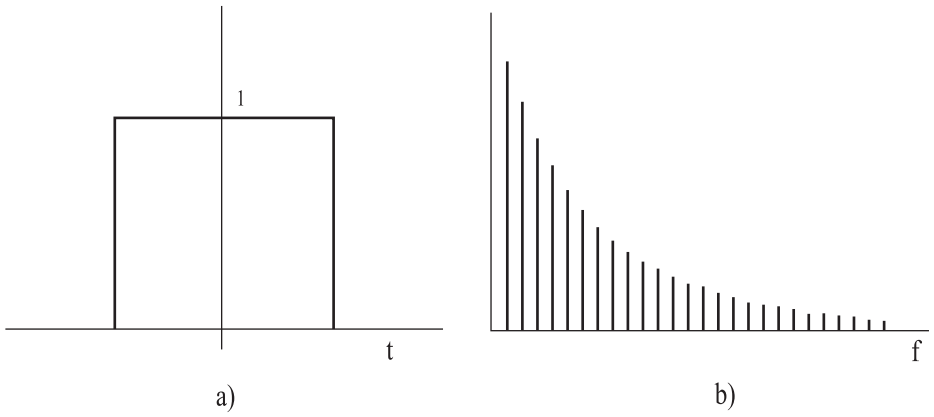
Koncept frekvenčnega prostora razložimo z enostavnima primeroma [1]. Funkcija $g(t) = sin(2\pi ft) + (1/3)sin(2\pi(3f)t)$ je kombinacija dveh sinusnih krivulj z amplitudo 1 in 1/3 ter frekvencama f in $3f$. Vidimo ju na sliki 4.1a in 4.1b. Njuna vsota je tudi periodična s frekvenco f (manjša frekvenca izmed frekvenc f in $3f$), kot vidimo na sliki 4.1c. V frekvenčnem prostoru je funkcija $g(t)$ predstavljena samo z dvema točkama $(f, 1)$ in $(3f, 1/3)$ (slika 4.1d).



Slika 4.1: Časovni in frekvenčni prostor

Predstavitev neperiodične funkcije je v frekvenčnem prostoru nekoliko zahtevnejša. Na primer, poglejmo funkcijo v obliki pravokotnega impulza na sliki 4.2a. Njeno predstavitev v frekvenčnem prostoru vidimo na sliki 4.2b. Sestoji iz neskončno mnogo funkcij, katerih frekvence težijo proti ∞, njihove amplitude pa se monotono zmanjšujejo. Širino frekvenčnega spektra neke funkcije imenujemo **pasovna širina**. Za *neškodljiv* prenos pravokotnega signala očitno potrebujemo neskončno pasovno širino, za prenos funkcije iz slike 4.1 pa pasovno širino $3f$.

Transformacijo iz časovnega v frekvenčni prostor in obratno je definiral Fourier. Njegov osnovni teorem pravi, da lahko vsako periodično funkcijo, realno ali kompleksno, predstavimo kot vsoto sinusnih in kosinusnih funkcij. Če je funkcija neperiodična, bo Fourierjeva transformiranka vključevala ne-



Slika 4.2: Časovni in frekvenčni prostor pravokotnega impulza

skončno frekvenc. Fourierjevo transformacijo in inverzno Fourierjevo transformacijo podaja enačba 4.1.

$$\begin{aligned} G(f) &= \int_{-\infty}^{\infty} g(t)[\cos 2\pi ft - i \sin 2\pi ft]dt \\ g(t) &= \int_{-\infty}^{\infty} G(f)[\cos 2\pi ft + i \sin 2\pi ft]df. \end{aligned} \tag{4.1}$$

V računalništvu praviloma uporabljajo diskretno obliko funkcije, ki upošteva le n vrednosti (enačba 4.2)

$$\begin{aligned} G(f) &= \sum_{t=0}^{n-1} g(t)[\cos \frac{2\pi ft}{n} - i \sin \frac{2\pi ft}{n}] \\ &= \sum_{t=0}^{n-1} g(t)e^{-ift},\ 0 \leq f \leq (n-1). \end{aligned} \tag{4.2}$$

in njen inverz

$$g(t) = \frac{1}{n}\sum_{f=0}^{n-1} G(t)[\cos\frac{2\pi ft}{n} + i\sin\frac{2\pi ft}{n}]$$
$$= \sum_{f=0}^{n-1} G(f)e^{ift},\ 0 \le f \le (n-1). \tag{4.3}$$

Koncept analize frekvenčne vsebine je pri obdelavi in stiskanju digitalnih slik zelo pogost. Imejmo enobarvno fotografijo, ki jo prebiramo vrstico za vrstico. Predpostavimo, da ima slika neskončno ločljivost z zvezno intenzivnostjo $I(t)$. V praksi se seveda omejimo na končno število vrednosti $I(1)$ do $I(n)$. Ta proces imenujemo vzorčenje. Intuitivno,vzorčenje bi naj bil kompromis med kvaliteto in ceno. Večji je vzorec, boljša je kvaliteta končne slike, vendar za to potrebujemo več strojne opreme (več pomnilnika in večjo ločljivost izhodnih naprav), kar je seveda dražje. Vendar pa to intuitivno sklepanje ni popolnoma pravilno. Teorija vzorčenja pravi, da lahko vzorčimo sliko tako, da jo znamo kasneje rekonstruirati brez izgube kakovosti, če storimo naslednje:

1. Transformiramo funkcijo intenzivosti $I(t)$ iz časovnega v frekvenčni prostor.
2. Poiščemo maksimalno frekvenco f_m.
3. Vzorčimo $I(t)$ pri rahlo večji frekvenci kot je $2f_m$.
4. Shranimo vzorčene vrednosti v bitno mapo. Rezultirajoča slika bo po kakovosti enaka tisti na fotografiji.

Pri tem lahko nastopi težava, ker je lahko $f_m = \infty$. V tem primeru izberemo za f_m takšno frekvenco, od katere dalje imajo vse višje frekvence dovolj majhno amplitudo. Jasno, zaradi tega je bitna slika nekoliko slabše kakovosti. Da je frekvenca vzorčenja $2f_m$ dovolj visoka, je dokazal Harry Nyquist. Če smo natančni, je Nyquistova frekvenca razlika med maksimalno in minimalno frekvenco, čemur pravimo tudi pasovna širina signala.

Fourierjeva transformacija je uporabna in pogosto uporabljana v najrazličnejših področjih. Ima pa pomembno slabost. Pokaže nam frekvenčno vsebino signala funkcije $f(t)$ vendar ne določa, kje (za katere vrednosti t) ima funkcija nizke oziroma visoke frekvence. Bazne funkcije Fourierjeve transfrmacije (sinusi in kosinusi) so namreč periodične in identificirajo vse frekvence funkcije $f(t)$ neglede na njihov položaj. Boljša ideja bi bila določanje

frekvenčne vsebine funkcije $f(t)$ kot funkcije t. Takšno transformacijo imenujemo **valčna transformacija** (angl. wavelet transformation). Razvita je bila v osemdesetih letih in od takrat uporabljena na različnih področjih znanosti in inženirstva. Osnovna ideja je v izbiri osnovnega valčka (angl. mother wavelet), to je funkcija, ki je različna od 0 samo na nekem malem intervalu. To funkcijo uporabimo za to, da bi raziskali lastnosti funkcije $f(t)$ (v bistvu, da bi dobili njeno frekvenčno vsebino) samo na tem intervalu. Osnovni valjček lahko zatem premaknemo na drugi interval od t in ga uporabimo na enak način.

Princip analiziranja funkcije glede na čas in frekvenco lahko uporabimo pri stiskanju slik zato, ker slike vsebujejo področja, ki izkazujejo neke trende in področja, ki vsebujejo anomalije. Trend je lastnost slike, ki vključuje samo nekaj frekvenc, te pa so prisotne na večih mestih v sliki. Anomalija pa je značilnost, ki vključuje različne frekvence, ki pa jih najdemo samo lokalno. Primer anomalije je na primer rob.

4.1.1 Zvezna valčna transformacija in njena inverzna transformacija

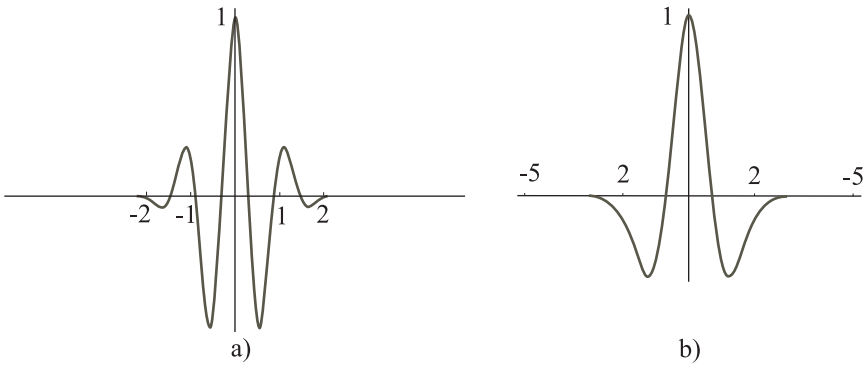
Zvezna valčna transformacija CWT funkcije $f(t)$ vključuje osnovni valček ψ. Osnovni valček je lahko poljubna realna ali kompleksna funkcija, ki zadosti naslednjima lastnostima:

1. Ploščina pod krivuljo osnovnega valčka je nič.
2. Ploščina kvadrirane funkcije osnovnega valčka je končna. Ta pogoj zahteva, da je možno funkcijo osnovnega valčka kvadrirati in da je kvadrat funkcije integrabilen.

Prva lastnost pravi, da funkcija oscilira nad in pod osjo t. Takšna funkcija je podobna valu. Druga lastnost pa določa, da je energija funkcije končna. To pomeni, da je funkcija lokalizirana na nekem intervalu in je nič (ali skoraj nič) izven tega intervala. Tem pogojem zadosti neskončno funkcij, nekatere funkcije so bile dobro raziskane in jih uporabljamo pri valčni tranfsormaciji. Primera takšne valčne osnovne funkcije sta Morletov valček ali valček meksikanski klobuk (slika 4.3a in b).

4.1.2 Haarov valček in Haarova transformacija

Digitalne slike so predstavljene z individualnimi (diskretnimi) števili. To je razlog, zakaj uporabljamo diskretno valčno transformacijo. Najpreprostejša diskretna valčna transformacija je Haarov valček.



Slika 4.3: Morletov valček (a) in valček meksikanski klobuk (b)

Za predstavitev velikega števila funkcij $f(t)$ uporabljamo Haarova transformacijo, ki uporablja skalirno funkcijo $\phi(t)$ in valček $\psi(t)$ (glej sliko 4.4a). Predstavitev funkcije $f(t)$ je v obliki neskončne vsote

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \phi(t-k) + \sum_{k=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} d_{j,k} \psi(2^j t - k) \tag{4.4}$$

ker so c_k in $d_{j,k}$ koeficienti, ki jih želimo izračunati.

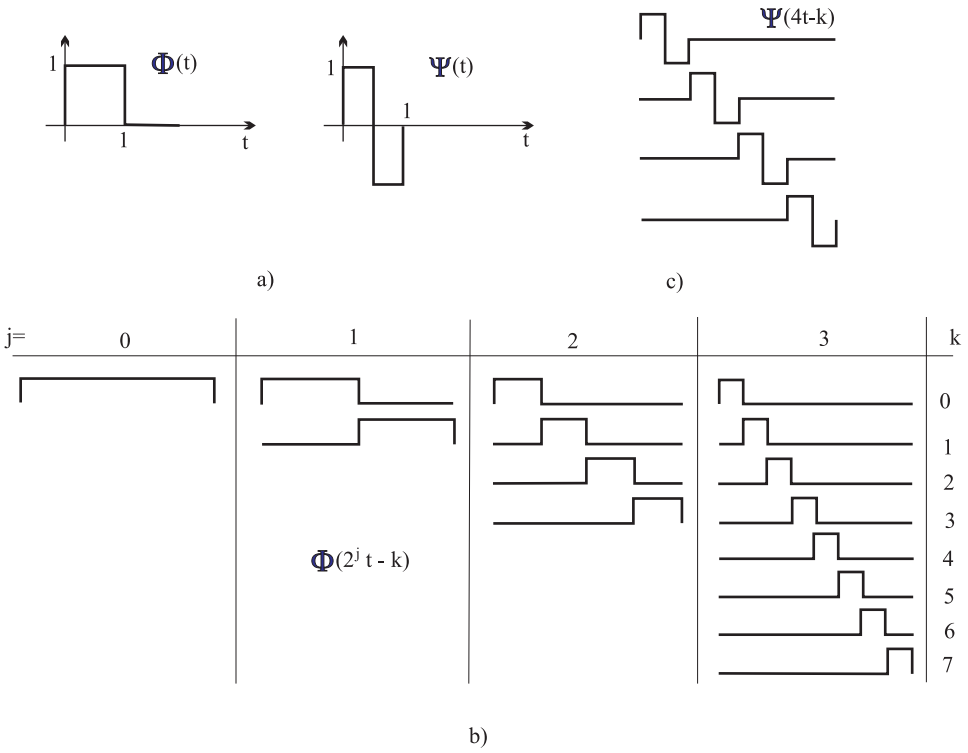
Osnovna skalirna funkcija ϕ je enotin impulz

$$\phi(t) = \begin{cases} 1 & : \quad 0 \leq t < 1 \\ 0 & : \quad sicer \end{cases} \tag{4.5}$$

Funkcija $\phi(t-k)$ je kopija $\phi(t)$, ki je premaknjena za k enot v desno. Podobno je $\phi(2t-k)$ kopija $\phi(t-k)$, skalirana na polovico širine $\phi(t-k)$. Premaknjene kopije uporabljamo za aproksimacijo funkcije $f(t)$ v različnih časih. Skalirane kopije pa uporabljamo za aproksimacijo $f(t)$ pri višjih ločljivostih. Slika 4.4b kaže funkcije $\phi(2^j t - k)$ za j = 0,1,2 in 3 ter $k = 0, 1, \cdots, 7$. (Primer izračuna, če je j=1: 2t-k=0, ko je k=0, je t=0, ko je k=1, je t 0.5)

Osnovni Haarov valček je stopnica:

$$\psi(t) = \begin{cases} 1 & : \quad 0 \leq t < 0.5 \\ -1 & : \quad 0.5 \leq t < 1. \end{cases} \tag{4.6}$$



Slika 4.4: Skaliranje Haarove in valčne funkcije

Vidimo lahko, da je splošen Haarov valček $\psi(2^j t - k)$ kopija $\psi(t)$ premaknjena za k enot v desno in skalirana tako, da je njegova celotna širina $1/2^j$. Slika 4.4c kaže štiri Haarove valčke $\psi(2^j t - k)$ za $k = 0, 1, 2$ in 3. Tako $\phi(2^j t - k)$ kot $\psi(2^j - k)$ sta neničelni na intervalu širine $1/2^j$. Ta interval je njuna **podpora** (angl. support). Ker je interval ozek, pravimo, da imajo te funkcije **kompaktno podporo** (angl. compact support).

Pokažimo sedaj osnovno transformacijo na enostavni stopnični funkciji

$$f(t) = \begin{cases} 5 & : \quad 0 \leq t < 0.5 \\ 3 & : \quad 0.5 \leq t < 1. \end{cases} \tag{4.7}$$

Vstavimo vrednosti za $f(t)$ pri $t = 0$ in $t = 0.5$, pri čemer je $k = j = 0$ Dobimo:

$$\begin{aligned} 5 &= c_k\phi(t=0) + d_{j,k}\psi(t=0) \\ 5 &= c_k + d_{j,k} \\ 3 &= c_k\phi(t=0.5) + d_{j,k}\psi(t=0.5) \\ 3 &= c_k - d_{j,k} \end{aligned} \tag{4.8}$$

Rešimo sistem in dobimo $c_k = 4$ in $d_{j,k} = 1$, kar vstavimo v funkcijo in dobimo $f(t) = 4\phi(t)+\psi(t)$. Vidimo, da se je stopnica (5,3) pretransformirala v povprečno vrednost 4 (nižja ločljivost) in detail 1 (višja ločljivost).

Slika je dvodimenzionalno polje vrednosti pikslov. Da bi pokazali, kako uporabimo Haarovo transformacijo pri stiskanju slik, pričnimo z eno samo vrstico vrednosti pikslov, torej z enodimenzionalnim poljem z n vrednostmi. Zaradi enostavnosti recimo, da je n potenca števila 2. Pri tem ne zgubimo na splošnosti, saj lahko vedno dodamo ustrezno število ničel. Predpostavimo polje osmih vrednosti (1,2,3,4,5,6,7,8). Najprej izračunamo štiri povprečja (1+2)/2 = 3/2, (3+4)/2 = 7/2, (5+6) = 11/2 in (7+8) = 15/2. Iz teh vrednosti je nemogoče rekonstruirati začetne vrednosti, zato izračunamo tudi razlike (1-2)/2 = -1/2, (3-4)/2 = -1/2, (5-6)/2 = -1/2, (7-8)/2 = -1/2. Te razlike imenujemo **podrobnejši koeficienti**, izraza *razlike* in *detaili* pa se uporabljata enakovredno. Povprečja lahko obravnavamo kot grobo ločljivost izvorne slike, detaile pa kot podatke, ki jih potrebujemo za rekonstrukcijo originalne slike iz grobe ločljivosti. Če obstaja korelacija med piksli v sliki, bo groba slika podobna originalni sliki, detaili pa bodo v tem primeru mali. To je tudi razlaga, zakaj uporabljamo pri stiskanju slik Haarov valček.

Hitro lahko vidimo, da je polje (3/2, 7/2, 11/2, 15/2, -1/2, -1/2, -1/2, -1/2), ki sestoji iz štirih povprečij in štirih razlik, lahko uporabljeno za rekonstrukcijo originalnih osmih vrednosti. Zadnje štiri vrednosti so majhne, kar nam pomaga pri stiskanju. Opogumljeni s tem dejstvom, postopek ponovimo na prvih štirih vrednostih, ki jih prav tako pretvorimo v povprečja in razlike. Dobimo (10/4, 26/4, -4/4, -4/4, -1/2, -1/2, -1/2, -1/2). Postopek ponovimo še enkrat in prvi dve komponenti spremenimo v povprečje in razliko. Dobimo: (36/8, -16/8, -4/4, -4/4, -1/2, -1/2, -1/2, -1/2), kar imenujemo Haarova valčna transformacija originalnih vrednosti.

Zaradi razlik teži valčna transformacija k vrednostim, ki so manjše od originalnih vrednosti pikslov, zato jih enostavneje stisnemo s kombinacijo RLE in Huffmanovega kodiranje. Stiskanje z izgubo dobimo, če majhne vrednosti kvantiziramo in jih odstranimo (postavimo na 0).

Preden nadaljujemo, si poglejo še časovno zahtevnost transformacije. V našem primeru potrebujemo 8+4+2=14 operacij (seštevanj in odštevanj). To število lahko predstavimo tudi kot 14=2(8-1). V splošnem imamo $N =$

2^n podatkov. V prvi iteraciji potrebujemo 2^n operacij, v drugi 2^{n-1} in tako dalje do $2^{n-(n-1))} = 2^1$. Skupno število operacij dobimo kot vsota geometrijske vrste, ki določimo po dobro znani formuli:

$$S_n = \frac{a_1(q^n - 1)}{q - 1} \tag{4.9}$$

V našem primeru sta $q = 2$ in $a_1 = 2$, kar nam da:

$$S_n = \frac{2(2^n - 1)}{2 - 1} = 2(N - 1). \tag{4.10}$$

Haarovo transformacijo nad N podatki torej realiziramo v času $O(N)$, kar je odličen rezultat.

Izkaže se ugodno, če vsaki iteraciji priredimo še vrednost, ki jo imenujemo **ločljivost** (angl. resolution), ki je določena kot število preostalih povprečij na koncu vsake iteracije. Ločljivost po vsaki interaciji v našem primeru bi bila $4(= 2^2)$, $2(= 2^1)$ in $1(= 2^0)$.Vsaka komponenta valčne transformacije je zatem normalizirana z deljenjem s kvadratnim korenom njene ločljivosti. Na ta način dobimo **ortogonalno Haarovo transformacijo** (angl. orthonormal Haar transformacion). V našem primeru valčna transformacijo postane

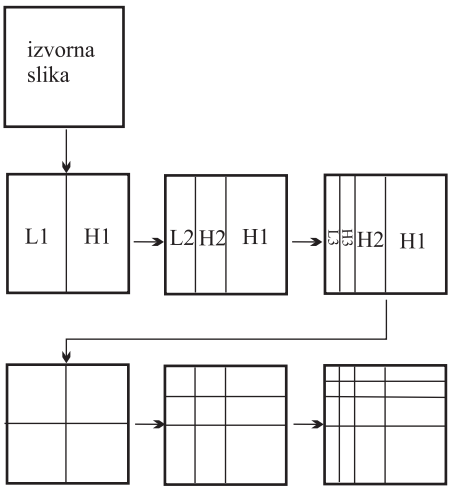
$$(\frac{36/8}{\sqrt{(2^0)}}, \frac{-16/8}{\sqrt{(2^0)}}, \frac{-4/4}{\sqrt{(2^1)}}, \frac{-4/4}{\sqrt{(2^1)}}, \frac{-1/2}{\sqrt{(2^2)}}, \frac{-1/2}{\sqrt{(2^2)}}, \frac{-1/2}{\sqrt{(2^2)}}, \frac{-1/2}{\sqrt{(2^2)}})$$

Teoretično se da dokazati, da pri ignoriranju majhnih razlik dosežemo najboljše rezultate pri uporabi normalizirane valčne transformacije.

4.1.3 Uporaba Haarove transformacije

Koncept Haarove transformacije sedaj enostavno posplošimo na dvodimenzionalne slike. V nadaljevanju si bomo ogledali standardno dekompozicijo in piramidno dekompozicijo.

Standardna Haarova dekompizicija prične z izračunavanjem valčne transformacije s transformacijo vrstic slike. Rezultat je transformirana slika, kjer prvi stolpec vsebuje povprečja, vsi ostali stolpci pa hranijo razlike. Standardni algoritem zatem izračuna valčne transformacije vseh stolpcev. Posledično zato dobimo povprečno vrednost v zgornjem levem kotu, ostali elementi vrhnje vrstice hranijo povprečja razlik, ostale komponente pa hranijo razlike (glej sliko 4.5).

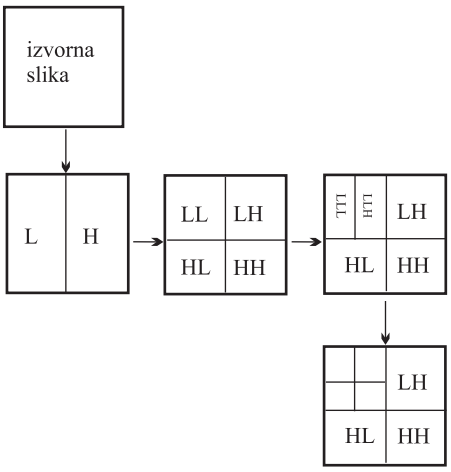


Slika 4.5: Standardna valčna dekompozicija

Piramidna dekompozicija računa valčno transformacijo z izmenjevanem vrstic in stolpcev. V prvem koraku izračunamo povprečja za vse vrstice (v tem primeru imamo samo eno iteracijo in ne celotne valčne transformacije). Na ta način dobimo povprečja v levi polovici slike in razlike v desni polovici. V drugem koraku izračunamo transformacijo za vse stolpce, kar nam da povprečja v zgornjem levem kvadrantu, razlike pa drugod. Koraka 3 in 4 uporabimo nad zgornjem levem kvadrantu in posledično dobimo povprečja, skoncentrirana v zgornjem levem podkvadrantu. Ta dva para korakov zatem ponavljamo na manjših in manjših kvadrantih, dokler ne dosežemo velikost kvadranta velikosti enega piksla, ki hrani povprečje. Ta proces vidimo na sliki 4.6.

Valčna transformacija je takoimenovana **pasovna transformacija** (angl. subband transformation). Transformacija deli sliko v področja tako, da en pas vsebuje velika števila (povprečja v primeru Haarove transformacije), drugi pas pa majhna števila (razlike). Prvemu pasu pravimo nizki pas, drugemu pa visoki pas, ki opisuje nenadne spremembe v signalu (sliki). Poglejmo si preprost primer, ki vsebuje vodoravno in navpično črto (slika 4.7).

Slika 4.7b kaže situacijo po uporabi Haarove transformacije nad vrsticami slike. Desna polovica te slike (razlike) vsebuje večino ničel, vendar pa lahko navpično črto zelo jasno izluščimo. Podobno se zgodi tudi, ko uporabimo transformacijo na stolpci. Zgornji desni pas vsebuje sled navpične črte, spodnji levi pa sled vodoravne črte. Ta pasova označimo s HL in LH. Spodnji



Slika 4.6: Piramidna dekompozicija

12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12
16	16	16	16	14	16	16	16
12	12	12	12	14	12	12	12
12	12	12	12	14	12	12	12

a)

12	12	13	12	0	0	1	0
12	12	13	12	0	0	1	0
12	12	13	12	0	0	1	0
12	12	13	12	0	0	1	0
12	12	13	12	0	0	1	0
16	16	15	16	0	0	-1	0
12	12	13	12	0	0	1	0
12	12	13	12	0	0	1	0

b)

12	12	13	12	0	0	1	0
12	12	13	12	0	0	1	0
14	14	14	14	0	0	0	0
12	12	13	12	0	0	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-2	-2	-1	-2	0	0	1	0
0	0	0	0	0	0	0	0

c)

Slika 4.7: Slika in njena pasovna dekompozicija

desni pas, ki ga označimo s HH, hrani diagonalne značilnosti. Najbolj pa je zanimiv zgornji levi pas (nizki pas), označen z LL, ki hrani cela povprečj. Ta pas je pomanjšana inačica originalne slike.

Kot smo spoznali, dobimo Haarovo transformacijo z računanjem povprečij in razlik. Pri tem si lahko pomagamo tudi z matrikami. Imejmo zaporedje osmih vrednosti $I = [255\ 224\ 192\ 159\ 127\ 95\ 63\ 32]$. Če vektor pomnožimo z matriko A_1

$$A_1 = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} \end{bmatrix} \tag{4.11}$$

dobimo $[239,5\ 175,5\ 111,0\ 47,5\ 16,5\ 16,0\ 15,5]$. Matriki A_2 in A_3 opravita transformacijo na drugem in tretjem nivoju.

$$A_2 = \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} A_3 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.12}$$

Skupno transformacijsko matriko W dobimo z množenjem $W = A_1 A_2 A_3$, dobljeni zmnožek pa je:

$$W = \begin{bmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{4} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{4} & 0 & -\frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{8} & \frac{1}{8} & -\frac{1}{4} & 0 & 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{8} & \frac{1}{8} & -\frac{1}{4} & 0 & 0 & -\frac{1}{2} & 0 & 0 \\ \frac{1}{8} & -\frac{1}{8} & 0 & \frac{1}{4} & 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{8} & -\frac{1}{8} & 0 & \frac{1}{4} & 0 & 0 & -\frac{1}{2} & 0 \\ \frac{1}{8} & -\frac{1}{8} & 0 & -\frac{1}{4} & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{8} & -\frac{1}{8} & 0 & -\frac{1}{4} & 0 & 0 & 0 & -\frac{1}{2} \end{bmatrix} \tag{4.13}$$

Če zmnožimo W I dobimo I_W = [143,375 64,125 32,000 31,750 15,500 16,500 16,000 15,500]. S tem smo opravili le polovico dela, saj moramo W aplicirati še na vrstice produkta W I, kar storimo tako, da z W pomnožimo transponiran rezultat. Končen transformiran rezultat je torej:

$$I_{tr} = (W(W\ I)^T)^T = W\ I\ W^T. \tag{4.14}$$

Inverzna transformacija je:

$$I = W^{-1}\ (I_{tr}\ (W^{-1})^T), \tag{4.15}$$

kjer pa postane pomembna normalizirana Haarova transformacijo. Namesto povprečja $(d_i + d_{i+1})/2$ in razlike povprečij $(d_i - d_{i+1})/2$ moramo uporabiti normalizirane vrednosti $(d_i + d_{i+1})/\sqrt{2}$ in $(d_i - d_{i+1})/\sqrt{2}$.

4.1.4 Filtri in banke filtrov

Filter je linearen operator, določen s filterskimi koeficienti $h(0), h(1), h(2), \cdots$. Filter uporabimo nad vhodnim vektorjem x, da dobimo izhodni vektor y z enačbo

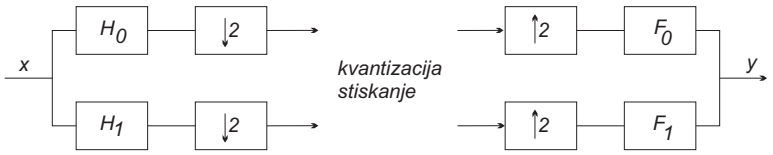
$$y(n) = \sum_k h(k)x(n-k) = h \star x, \tag{4.16}$$

kjer $\star$ označuje konvolucijo. Meje vsote v enačbi 4.16 niso natančno definirane, ker so odvisne od velikosti vektorjev x in h. Če je neodvisna spremenljivka čas t, je smiselno predpostaviti, da vhodni (in izhodni) signal pride v vseh diskretnih časih, torej $t = \cdots, -2, -1, 0, 1, 2, \cdots$. Zato bomo uporabili zapis

$$x = (\cdots, a, b, c, d, e, \cdots), \tag{4.17}$$

kjer je centralna vrednost c enaka vhodni vrednosti v času 0 ($c = x(0)$), vrednosti d in e ustrezajo vrednostim v časih 1 in 2 itd. Vhod je vedno končen, zato bo vektor x vedno imel končno število neničelnih elementov. Najpreprostejši vhod je enotski impulz $(\cdots, 0, 0, 1, 0, 0, \cdots)$. Iz enačbe 4.16 dobimo $y(n) = h(n)x(0) = h(n)$. Pravimo, da je izhod $y(n) = h(n)$ odziv v času n na enotski impulz $x(0) = 1$. Ker je število koeficientov filtra $h(i)$ končno, imenujemo takšen filter FIR-filter (angl. finite impulse response). Idejo bank filtrov (angl. filter bank) vidimo na sliki 4.8 z dvema filtrom, nizkopasovnim filtrom H_0 in visokopasovnim filtrom H_1. Nizki filter s konvolucijo odstrani visoke frekvence iz signala x in prepusti samo njegove

nizke frekvence, visoki filter pa naredi ravno obratno. Oba skupaj razdelita vhod v frekvenčna pasova (angl. frequency bands). Elementi x prihajajo v filtra eden za drugim, vsak filter pa odda odziv $y(n)$. Število odzivov je torej v primeru dveh filtrov podvojeno, kar je slaba novica, če bi želeli pristop uporabiti za stiskanje podatkov. Da bi situacijo popravili, uporabimo podvzorčenje izhoda, kjer vsako sodo vrednost zavržemo. Operaciji pravimo decimacija (angl. decimation) in jo označimo z $\downarrow 2$, na izhodu pa na ta način dobimo enako število elementov kot na vhodu. Najpomembnejša lastnost banke filtrov pa je, da lahko vhodni signal rekonstruiramo, četudi smo izhod decimirali.

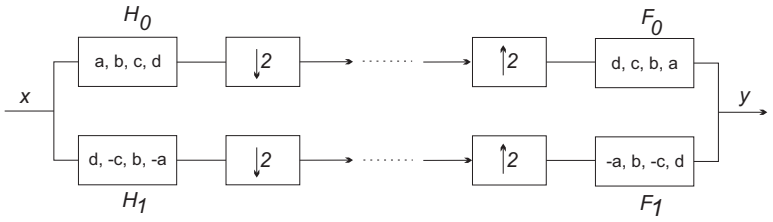


Slika 4.8: Dvokanalna banka filtrov.

In kaj počnejo filtri v bankah filtrov. Nič drugega, kot izračunavajo podobne matrike kot je matrika $W = A_1 A_2 A_3$ iz prejšnjega podpoglavja. Banka filtrov je le drug pogled na valčno transformacijo.

Podvzorčenje ni časovno neodvisno. Po podvzorčenju dobimo $y(0), y(2), y(4), \cdots$. Če pa signal zakasnimo za en časovni interval, dobimo lihe izhodne vrednosti $y(-1), y(1), y(3), \cdots$, ki so različne in neodvisne od originalnega izhoda. Takšnima dvema zaporedjema signalov pravimo fazi vektorja y.

Izhode iz bank filtrov imenujemo podpasovni koeficient (angl. subband coefficients). Uporabimo jih kot vhod v banko spajanj (angl. synthesis bank), kjer jih naprej nadvzorčimo (angl. upsampling). To storimo tako, da na lihih mestih vstavimo ničle. Tako razširjena vektorja pošljemo skozi inverzne filtre F_0 in F_1 ter jih nato sestavimo v izhodni vektor.



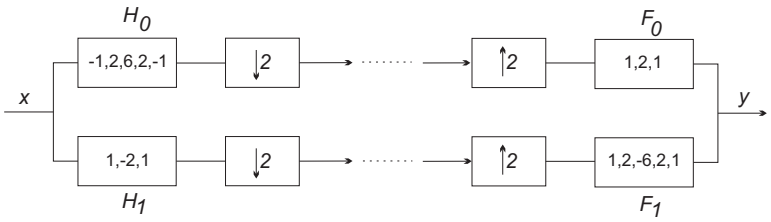
Slika 4.9: Ortogonalna banka filtrov s širimi koeficienti.

Podvzorčenje seveda vnaša izgube in preprosto nadvzorčenje z vrinjanjem ničel jih ne more izničiti. Zato, da bi dosegli brezizgubno rekonstrukcijo originalnega vhodnega vektorja x, morajo biti filtri načrtovani tako, da bodo kompenzirali to izgubo v podatkih. Najpogostejša rešitev pri načrtovanju filtrov je njihova ortogonalnost. Slika 4.9 kaže nabor ortogonalnih filtrov velikosti 4. Filtri so ortogonalni zato, ker je njihov skalarni produkt enak

$$(a, b, c, d) \cdot (d, -c, b, -a) = 0. \tag{4.18}$$

Vidimo, da so si filtri H_0 in F_0 ter H_1 in F_1 zelo podobni. Vrednosti, ki jih zavzamejo parametri a, b, c in d, pa so pritegnile mnoge raziskave. Eden izmed najpomembnejših filtrov je filter Daubechies D4 (uporablja ga tudi standard JPEG2000). Zanimivo je, da filter povsem rekonstruira izvorni vektor, ki pa je zamaknjen za 3 časovne enote.

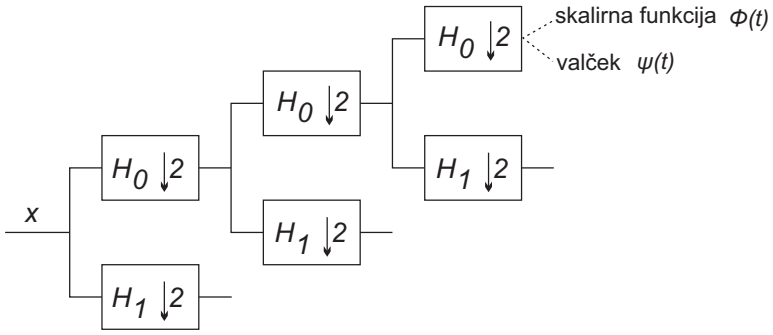
Filtri so lahko tudi biortogonalni. Na sliki 4.10 vidimo primer takšnih filtrov, ki zmorejo povsem rekonstruirati signal.



Slika 4.10: Biortogonalna banka filtrov.

V prejšnjem podpoglavju smo videli, da izhod nizkoprepustnega filtra L_0 gre večkrat skozi zaporedje filtrov, s čimer dobimo vedno krajši izhod. Ta proces lahko predstavimo z drevesom na sliki 4.11. Ker vsako izmed oglišč drevesa razpolovi število koeficientov svojega predhodnika, drevesu pravimo tudi logaritemsko drevo (angl. logarithmic tree). Z vzpenjanjem po drevesu iz nivoja i na nivo $i + 1$ računamo nova povprečja iz novih skalirnih funkcij $\phi(t - k)$ in nive podrobnosti z novim valčkom $\psi(2^i t - k)$.

Če primerjamo transformacijo DCT in valčno transformacijo (tranformacijo, ki temelji na frekvečnih pasovih) ugotovimo, da gre za povsem drugačni transformaciji. V primeru valčne transformacije vsak frekvenčnih pas decimiramo, nato pa ga lahko obravnavamo povsem ločeno, s čimer omogočimo uporabniku nadzor različne stopnje izgub v posameznih pasovih. Seveda pa ima Haarova valčna transformacija tudi slabosti, saj zaradi podvzorčenja vnaša šum in aliasing. Zato so se raziskave osredotočile na iskanje boljših filtrov.



Slika 4.11: Logaritmično drevo banke filtrov.

4.1.5 Koeficienti filtrov

V tem podpoglavju si bomo neformalno ogledali pravila in metode za določanje vrednosti koeficientov filtrov v bankah filtrov. Več podrobnosti bralec najde drugje [15].

Imejmo filtra h_0 in h_1 z N koeficienti (pri čemer je N sod) in dva inverzna filtra f_0 in f_1. Njihove koeficiento označimo kot:

$$\begin{aligned} h_0 &= (h_0(0), h_0(1), \ldots, h_0(N-1)), \\ h_1 &= (h_1(0), h_1(1), \ldots, h_1(N-1)), \\ f_0 &= (f_0(0), f_0(1), \ldots, f_0(N-1)), \\ f_1 &= (f_1(0), f_1(1), \ldots, f_1(N-1)). \end{aligned} \tag{4.19}$$

Nabor, pogojev, ki jih morajo koeficienti izpolnjevati so:

- Normalizacija: Vektor h_0 naj bo normaliziran (njegova vrednost je 1).
- Ortogonalnost: Za vsak $1 \leq i < N/2$ naj bo vektor, ki je sestavljen iz prvih $2i$ elementov h_0, ortogonalen vektorju, ki je sestavljen iz zadnjih $2i$ elementov.
- Vektor f_0 je reverz h_0.
- Vektor h_1 je kopija vektorja f_0, pri čemer so lihi elementi nasprotno predznačeni (vektor f_0 pomnožimo z vektorjem $(-1, 1, -1, 1, \ldots, -1, 1)$).
- Vektor f_1 je kopija h_0, pri čemer so sodi elementi nasprotno predznačeni.

Za filter z dvema elementoma bi veljalo:

$$h_0^2(0) + h_0^2(1) = 1. \tag{4.20}$$

Pravila 2 ne moremo uporabiti, ker je $i < 1$ zaradi $i < N/2$. Pravila 3-5 dajo:

$$f_0 = (h_0(1), h_0(0)), \;\; h_1 = (-h_0(1), h_0(0)), \;\; f_1 = (h_0(0), -h_0(1)). \tag{4.21}$$

Kot vidimo, so vsi vektorji odvisni od koeficientov $h_0(0)$ in $h_0(1)$, enačba 4.20 pa ni dovolj, da bi ju enolično določili. Ena izmed možnosti, ki jo hitro uganemo, je $h_0(0) = h_0(1) = 1/\sqrt{2}$.
Za filter s štirimi koeficienti nam pravili 1 in 2 dasta

$$h_0^2(0) + h_0^2(1) + h_0^2(2) + h_0^2(3) = 1, \;\; h_0(0)h_0(2) + h_0(1)h_0(3) = 0, \tag{4.22}$$

pravila 3-5 pa dajo

$$\begin{aligned} f_0 &= (h_0(3), h_0(2), h_0(1), h_0(0)), \\ h_1 &= (-h_0(3), h_0(2), -h_0(1), h_0(0)), \\ f_1 &= (h_0(0), -h_0(1), h_0(2), -h_0(3)), \end{aligned} \tag{4.23}$$

Tudi v primeru enačbe 4.22 nimamo dovolj enačb za določitev štirih spremenljivk; rešimo jih intuitivno. Posledično je možno zgenerirati neskončno filtrov. Samo nekateri izmed njih pa dajejo dobre rezultate. Med njimi je najbolj znan in najpogosteje uporabljen filter Daubechies D4, ki ima naslednje vrednosti parametrov:

$$h_0(0) = (1 + \sqrt{3})/4\sqrt{2} \approx 0,48296, \tag{4.24}$$

$$\begin{aligned} h_0(1) &= (3 + \sqrt{3})/4\sqrt{2} \approx 0,8365, \\ h_0(2) &= (3 - \sqrt{3})/4\sqrt{2} \approx 0,2241, \end{aligned}$$

$$h_0(3) = (1 - \sqrt{3})/4\sqrt{2} \approx -0.1294. \tag{4.25}$$

4.2 Diskretna valčna transformacija

Funkcije, s katerimi se srečujemo pri stiskanju podatkov, prihajajo v obliki zaporedja števil in ne zveznih funkcij. Izkušnje kažejo, da je kakovost

transformacije odvisna od skalirnega faktorja, časovnega pomika in izbire valčka. V praksi vzamemo za skalirni faktor negativno potenco števila 2, časovni premik pa določimo kot nenegativno potenco števila 2. Uporabljeni valčki naj bodo ortogonalni (ali biortogonalni). Najlažje opišemo diskretno valčno transformacijo z množenjem matrik. Haarova transformacija ima dva koeficienta c_0 in c_1, za oba smo intuitivno ugotovili, da imata vrednost $1/\sqrt{2} \approx 0,707$. Transformacijska matrika je v tem primeru:

$$\frac{1}{\sqrt{2}} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4.26}$$

Ugotovimo, da ti parametri ne dajo natančne polovice in razlike, ker uporabimo $\sqrt{2}$ namesto 2. Zato poimenovanje njihovega efekta kot grobe (angl. coarse) in fine (angl. fine) podrobnosti. V splošnem lahko DWT uporabi kakršen koli nabor filtrov, izračunamo pa jo na povsem enak način.

Kot primer si oglejmo najpopularnejši valček Daubechies D4 (enačba 4.25). Transformacijska matrika W je:

$$W = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & 0 & 0 & \dots & 0 \\ c_3 & -c_2 & c_1 & -c_0 & 0 & 0 & \dots & 0 \\ 0 & 0 & c_0 & c_1 & c_2 & c_3 & \dots & 0 \\ 0 & 0 & c_3 & -c_2 & c_1 & -c_0 & \dots & 0 \\ \vdots & \vdots & & & & & & \vdots \\ 0 & 0 & \dots & 0 & c_0 & c_1 & c_2 & c_3 \\ 0 & 0 & \dots & 0 & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & 0 & \dots & 0 & 0 & c_0 & c_1 \\ c_1 & -c_0 & 0 & \dots & 0 & 0 & c_3 & -c_2 \end{bmatrix} \tag{4.27}$$

Če uporabimo matriko W nad podatkovnim vektorjem $[x_1, x_2, \dots, x_n]$, prva vrstica generira uteženo vsoto $s_1 = c_0x_1 + c_1x_2 + c_2x_3 + c_3x_4$. Tudi vse ostale sode vrstice dajo utežene vsote s_i. Te sume so konvolucije med elementi podatkovnega vektorja in štirimi filterskimi koeficienti. Včasih te konvolucije imenujemo tudi gladki koeficienti (angl. smooth coefficients) ali gladili filter H.

Druga vrstica matrike generira vrednost $d_1 = c_3x_1 - c_2x_2 + c_1x_3 - c_0x_4$; vse lihe vrstice generirajo podobne konvolucije. Vsak d_i imenujemo detajlni koeficient (angl. detail coefficient), vsi skupaj pa tvorijo filter G. Filter G je načrtovan tako, da tvori male vrednosti, ko so podatki korelirani. Oba

filtra H in G skupaj imenujemo filter QMF (angl. quadrature mirror filters). Pravimo, da filter QMF sestoji iz nizkega (H) in visokega (G) filtra.

Da bi bila W ortogonalna, morajo koeficient zadostiti dvema relacijama: $c_0^2+c_1^2+c_2^2+c_3^2=1$ in $c_2c_0+c_3c_1=0$. Še dve dodatni relaciji uporabimo pri izračunu koeficientov, in sicer $c_3-c_2+c_1-c_0=0$ in $0c_3-1c_2+2c_1-3c_0=0$. Rešitev so koeficienti filtra Daubechies D4 iz enačbe 4.25.

Uporaba matrike W je konceptualno enostavno, ni pa najbolj praktična, če gre za velike slike. A kaj hitro ugotovimo, da ni potrebno sestaviti cele matrike W, dovolj je le ena vrstica, kjer elemente ustrezno premikamo čez vrednosti pikslov. Najbolj popularni filtri (oz. valčki) so [1]:

- Haarov valček, ki ga lahko obravnavamo kot Daubechiesov filter reda 2,
- družina filtrov Daubechies,
- Beylkinov filter,
- Coifmanov filter,
- Vyidyanathanov filter,
- simetričen filter in drugi.

4.3 SPIHT

SPIHT (angl. set partitioning in hierarchical trees) je namenjen za napredujoči prenos in stiskanje slik. Uporablja tako imenovano vgrajeno kodiranje (angl. embedded coding), pri katerem velja naslednje pravilo: če kodirnik izda dve datoteki, večjo velikosti M in manjšo velikosti m, potem je manjša datoteka enaka prvim m bitom večje datoteke. Idejo si poglejmo na primeru. Predpostavimo, da uporabniki pričakujejo stisnjeno sliko, a potrebujejo sliko različnih kakovosti (prvi bi bil zadovoljen s kakovostjo, vsebovano v datoteki velikosti 10 kB, drugi 20 kB in tretji 50 kB). Večina kodirnikov bi moralo sliko stisniti trikrat z različno kvantizacijo, SPIHT pa ustvari eno datoteko, tri dele v velikosti 10 kB, 20 kB in 50 kB, ki vsi začnejo na začetku datoteke, zato jih lahko pošljemo trem različnim uporabnikom.

Izvorno sliko označimo s $\mathbf{p}$, piksel pa s $p_{i,j}$. Naj bo $\mathbf{T}$ nabor valčnih filtrov, s katerimi dobimo transformirane koeficiente $c_{i,j}$. Ti koeficienti predstavljajo transformirano sliko $\mathbf{c}$; $\mathbf{c}=\mathbf{T}(\mathbf{p})$. Dekodirnik prične z rekonstrukcijo s transformacijo $\hat{\mathbf{c}}$, kjer so vsi koeficienti postavljeni na 0. Nato sprejme

kodirane koeficiente, jih dekodira in nato integrira v $\hat{\mathbf{c}}$, s čimer izboljša rekonstruirano sliko $\hat{\mathbf{p}}$. Pri napredujočem prenosu najprej prenesemo najpomembnejše informacije (najpomembnejše informacije povzročijo najmanjšo razliko med izvorno in rekonstruirano sliko). SPIHT kot mero za to uporablja MSE (enačba 4.28), kjer N ustreza skupnemu številu pikslov v $\mathbf{c}$.

$$D_{MSE}(p - \hat{p}) = \frac{1}{N} \sum_i \sum_j (p_{i,j} - \hat{p}_{i,j})^2 \tag{4.28}$$

Mera je neodvisna od uporabljenega valčka, zato lahko zapišemo:

$$D_{MSE}(c - \hat{c}) = \frac{1}{N} \sum_i \sum_j (c_{i,j} - \hat{c}_{i,j})^2 \tag{4.29}$$

Enačba 4.29 nam pove, da največji koeficient $c_{i,j}$ najbolj prispeva k zmanjšanju MSE, zato ga moramo poslati najprej. Posledično velja, da moramo najprej poslati najpomembnejše bite koeficientov največjih $c_{i,j}$. SPIHT zato uredi koeficiente in najprej pošlje najpomembnejše bite.

Predpostavimo, da smo že izračunali valčno transformacijo in da so koeficienti $c_{i,j}$ že v pomnilniku. Koeficiente uredimo neglede na njihov predznak. Informacije o urejanju shranimo v polju m tako, da element polja $m(k)$ hrani koordinate (i, j) koeficienta $c_{i,j}$ tako, da je $|c_{m(k)}| \geq |c_{m(k+1)}|$ za vse k. Primer vidimo na sliki 4.12, kjer je 16 koeficientov predstavljenih s 16-timi biti. Bit 16 hrani podatek o predznaku (s), preostalih 15 bitov pa sestavlja vrednost. Prvi koeficient $c_{m(1)} = c_{2,3}$ ima vrednost $s1aci \ldots r$, kjer so $s, a, c, \ldots$ biti. Druga vrednost $c_{m(2)} = c_{3,4} = s1bdj \ldots s$ itd.

k		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	sign	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s
msb	14	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	13	a	b	1	1	1	1	0	0	0	0	0	0	0	0	0	0
	12	c	d	e	f	g	h	1	1	1	0	0	0	0	0	0	0
	11	i	j	k	l	m	n	o	p	q	1	0	0	0	0	0	0
	⋮	⋮	⋮														⋮
lsb	0	r	s	t	u	v	w	x	y		...		...		...		z
$m(k) = i, j$		2,3	3,4	3,2	4,4	1,2	3,1	3,3	4,2	4,1	...		...		...		4,3

Slika 4.12: Koeficienti, urejeni po absolutni velikosti.

Kodirnik mora poslati polje m, ki hrani infromacije o urejanju (pare koordinat), 16 predznakov, in 16 koeficientov glede na njihovo ureditev. Neposredna implementacija bi te koeficiente poslala v naslednji obliki:

sssssssssssssss,	1100000000000000,		ab11110000000000,
cdefgh1110000000,	ijklmnopq1000000,	...,	rstuvwxy...y,

kar pa je zelo potratno. SPIHT zato raje rešuje problem iterativno. V prvi iteraciji pošlje število $l = 2$, ki določa število koeficientov $c_{i,j}$, za katere velja, da $|c_{i,j}| \geq 2^{14}$, številu l sledijo podatki o položaju koeficientov $((2,3),(3,4))$, nazadnje pošljemo še dva predznaka. Ti podatki omogočajo dekodirniku, da rekonstruira koeficienta $c_{2,3}$ in $c_{3,4}$ kot 16-bitni števili $s100\ldots0$. Naslednji korak kodirnika je korak izboljšav (angl. refinement pass), ki pa ga po prvem koraku še ne moremo opraviti. Kodirnik opravi oba koraka šele v drugi iteraciji, kjer najprej odda število $l = 4$ in s tem določi število koeficientov, za katere velja ($2^{13} \leq |c_{i,j}| < 2^{14}$). Sledijo štirje pari koordinat ($(3,2)$, $(4,4)$, $(1,2)$ in $(3,1)$), za njimi pa so štirje predznaki. V izboljševalnem koraku pa odda dva bita a in b, ki sta štirinajsta najbolj pomembna bita prejšnjih dveh koficientov. Dosedaj odposlani podatki omogočajo naslednjo rekonstrukcijo prvih šest koeficientov:

$$c_{2,3} = s1a00\ldots0, \quad c_{3,4} = s1b00\ldots0, \quad c_{3,2} = s0100\ldots0, \tag{4.30}$$

$$c_{4,4} = s0100\ldots0, \quad c_{1,2} = s0100\ldots0, \quad c_{3,1} = s0100\ldots0, \tag{4.31}$$

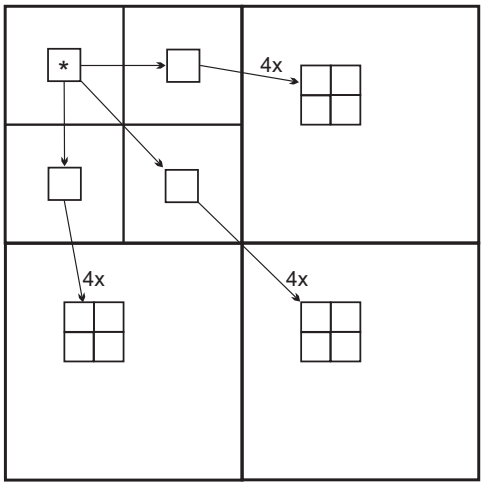
Preostale korake lahko sedaj nadaljujemo sami. Algoritem se konča, ko prenesemo najmanj pomembne bite, seveda pa lahko prenos prekinemo na katerem koli nivoju.

Opisana metoda je enostavna, a v primeru zelo velikih slik prepočasna. Namesto urejanja zato SPIHT uporablja dejstvo, da urejamo s primerjavo parov elementov in da je rezultat vedno enostaven odgovor DA/NE. Če tako kodirnik kot dekodirnik uporabljata enak algoritem urejanja, lahko kodirnik pošlje zaporedje izzidov DA/NE (dobimo zaporedje bitov). SPIHT pa gre še korak dlje, saj v našem primeru dejansko ni potrebno urediti vseh koeficientov. Glavna naloga koraka urejanja v vsaki iteraciji je izbrati tiste koeficiente, za katere velja $2^n \leq |c_{i,j}| < 2^{n+1}$. Ta naloga je razdeljena v dva dela. Za dano vrednost n koeficientu, za katerega velja $|c_{i,j}| \geq 2^n$, pravimo, da je *pomemben* (angl. significant), sicer pa je *nepomemben*. V začetku bo pomembnih malo koeficientov, njihovo število pa se bo povečevalo iz iteracije v iteracijo. Korak urejanja mora določiti, kateri pomembni koeficienti

zadostijo pogoju $|c_{i,j}| < 2^{n+1}$ in poslati njihove koordinate dekodirniku. Kodirnik razdeli vse koeficiente v množice T_k in nato opravi test pomembnosti

$$\max_{(i,j)\in T_k} |c_{i,j}| \geq 2^n \tag{4.32}$$

za vsako množico T_k. Rezultat je lahko NE, ko so vsi koeficienti v T_k nepomembni in posledično je tudi T_k nepomemben, ali DA, ko je nekaj koeficientov iz T_k pomembnih. Če je rezultat DA, to sporočimo dekodirniku, množico T_k pa tako kodirnik kot dekodirnik razdelita na enak način v podmnožice. Kodirnik nad vsako podmnožico nato opravi test DA/NE. Delitev opravljamo tako dolgo, dokler nimajo vse pomembne množice moč 1, to je, da vsebujejo samo en koeficient in ta je pomemben. Rezultat vsakega testa pomembnosti je bit, ki ga zapišemo v izhodni niz. Zato je zelo važno, da število potrebnih testov minimiziramo. Da bi to dosegli, moramo množice tvoriti in deliti na način, da bodo pomembne množice velike. Množice, za katere pričakujemo, da bodo nepomembne, pa naj imajo samo en element.



Slika 4.13: Drevo prostorske orientacije pri SPIHT.

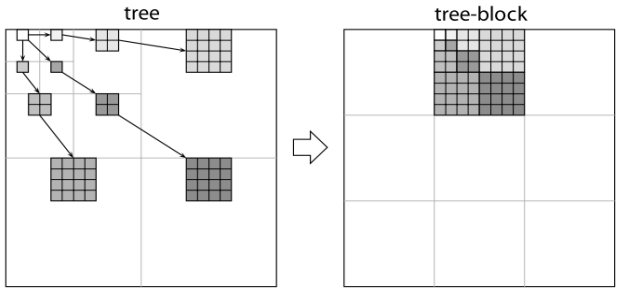
Večina energije slike je skoncentrirana v komponentah nizkih frekvenc. Posledično se tudi varianca zmanjšuje s premikanjem iz višjih proti nižjim nivojem piramidalne strukure. Pričakujemo tudi samopodobnost med posameznimi pasovi, prav tako pričakujemo, da bodo koeficienti bolje poravnani glede na vrednost s tem, ko se pomikamo po piramidni strukturi v isti pasovni orientaciji. Te prostorske relacije predstavimo s posebno podatkovno

strukturo – drevesom prostorske orientacije (angl. spatial orientation tree - SOT). Na sliki 4.13 vidimo, kako je drevo SOT skonstruirano. Korenov dreves je lahko več (v primeru na sliki 4.13 je samo eden velikosti 2×2, eden izmed njih, označen z *, pa nima naslednikov ((angl. descendants))). Neposredni nasledniki so vrednosti v naslednjem nivoju iste prostorske orientacije (i.e. istega frekvenčnega pasu). Imenovali jih bomo *potomci* (angl. offspring). Vozlišča na najnižjem nivoju drevesa so listi in seveda nimajo naslednikov.

Algoritem urejanja množic uporablja naslednje štiri množice koordinat:

1. $\mathcal{O}(i,j)$ hrani koordinate vseh potomcev vozlišča (i,j). Če je vozlišče (i,j) list, je $\mathcal{O}(i,j) = \emptyset$.
2. $\mathcal{D}(i,j)$ je množica koordinat vseh naslednikov vozlišča (i,j).
3. $\mathcal{H}(i,j)$ je množica koordinat korenov vseh SOT (to so vse koordinate koeficientov na najvišjem nivoju piramidne delitve).
4. $\mathcal{L}(i,j) = \mathcal{D}(i,j) - \mathcal{O}(i,j)$.

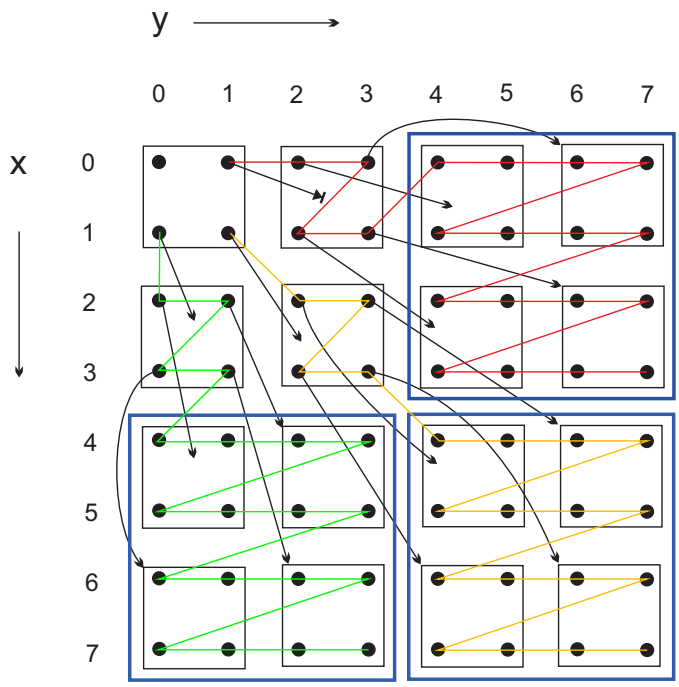
Z drevesom prostorske orientacije dejansko hierarhično predstavimo blok podatkov, kot vidimo na sliki 4.14, [16]. Množica vseh SOT opiše celotno sliko.



Slika 4.14: Relacija med blokom podatkov in drevesom prostorske orientacije.

Na sliki 4.15 vidimo nazornejšo hierarhično povezavo med elementi bloka. Drevesa SOT uporabljamo za delitev množic T_k z naslednjimi pravili:

1. Začetne množice so $\mathcal{D}(i,j)$ za vse $(i,j) \in \mathcal{H}$.
2. Če je $\mathcal{D}(i,j)$ pomembna, jo razdelimo v $\mathcal{L}(i,j)$ in štiri množice s po enim elementom $(k,l) \in \mathcal{D}(i,j)$.



Slika 4.15: Hierarhija med elementi v SOT.

3. Če je pomembna množica $\mathcal{L}(i,j)$, potem jo razdelimo v štiri množice $\mathcal{D}(k,l)$, $(k,j) \in \mathcal{D}(i,j)$.

4.3.1 Kodiranje SPIHT

Kodirnik uporablja tri sezname:

- seznam pomembnih pikslov (angl. list of significat pixels, LSP) hrani vrednosti, ki so večje od trenutnega praga.
- seznam nepomembnih pikslov (angl. list of insignificant pixels - LIP) hrani vrednosti, ki so manjše od trenutnega praga in
- seznam nepomembnih množic (angl. list of insignificant sets - LIS) hrani množice koeficientov, določene s strukturo drevesa SOT, ki imajo elemente, ki so manjši od praga, so torej nepomembni.

  Vsi seznami hranijo koordinate (i,j). V seznamih LIP in LSP predstavljajo individualne koeficiente, v LIS pa ali množico $\mathcal{D}(i,j)$ (tip A) ali množico $\mathcal{L}(i,j)$ (tip B).

Koraki so naslednji:

- Najprej preverimo vrednosti LIP, ki so bile v prejšnjem koraku označene kot nepomembne. Tiste, ki so postale pomembne, premaknemo v LSP.

- Nato testiramo množice v LIS. Tiste, ki postanejo pomembne, odstranimo iz LIS in jih razdelimo.

- Množice z več kot enim elementom ponovno vstavimo v LIS, tiste z enim elementom pa testiramo na pomembnost. Če so pomembne, jih vstavimo na konec LSP, tiste, ki niso pomembne pa na konec LIP.

- LSP hrani elemente, ki jih nato obiščemo v koraku izboljšav.

V nadaljevanju vidimo pseudokod algoritma.

	0	1	2	3	4	5	6	7
0	63	–34	49	10	7	13	–12	7
1	–31	23	14	–13	3	4	6	–1
2	15	14	3	–12	5	–7	3	9
3	–9	–7	–14	8	4	–2	3	2
4	–5	9	–1	47	4	6	–2	2
5	3	0	–3	2	3	–2	0	4
6	2	–3	6	–4	3	6	3	6
7	5	11	5	6	0	3	–4	4

Slika 4.16: Blok koeficientov.

```
SPIHT() {
  n = ⌊log2 max_{i,j}{|c_{i,j}|}⌋;
  LSP = { };
  LIP = Add((i, j) ∈ H);
  LIS = Add(A, Descendants((i, j) ∈ H));
  while (n ≥ 0) // Korak urejanja
  { for each (i,j)∈ LIP do
    { Output(Important((i,j));
      if (Important((i,j) == 1)
      { LSP = Add((i,j));
        Output(Sign(c_{i,j})); }
    }
    for each (i,j)∈ LIS do
    { if (type == A) {
        Output(Important(D(i, j));
      if (Important(D(i, j)) == 1) {
        for each (k,l)∈ O(i, j) do {
          Output(Important((k,l)));
          if (Important((k,l)) == 1) {
            LSP = Add((k,l));
            Output(Sign(c_{(k,l)})); }
          else
            LIP = Add((k,l));
        }
        if (L(i, j)) ≠ 1) {
          LIS = Add(B,(k,l));
        else
          LIS=Remove((i,j));
      }
    }
    else { // type = B
      Output(Sign(L(i, j)));
      if (L(i, j) = 1) {
        LIS=Add(A, (k,l) ∈ O(i, j) );
        LIS = Remove((i,j));
      }
    }
    // korak izboljšav
    for (each (i,j) ∈ LSP) && (not used in the last sorting pass) do
      Output(n^{th}_MSB(c_{i,j}));
    n − −;
  }
}
```

Delovanje algoritma bomo pokazali na bloku podatkov velikosti 8×8

koeficientov, prikazanih na sliki 4.16.

1. Najprej opravimo inicializacijo. Določimo začetno vrednost praga $n = 5$. LIS hrani množice tipa A brez korena, vrednost LIP je inicializirana na koordinate korena in potomcev, LSP pa je prazen. Velja torej:

   LIS ={(0,1)A, (1,0)A, (1,1)A },
   LIP = {(0,0), (0,1), (1,0), (1,1)},
   LSP = {}.

2. SPIHT prične kodiranje s preverjanjem pomembnosti pikslov v LIP. Če najdemo pomemben koeficient, pošljemo na izhod bit 1, ki mu sledi predznak (uporabili bomo oznako + ali -), koordinate koeficienta pa premaknemo v LSP. Rezultat tega koraka kaže razpredelnica 4.1.

   Tabela 4.1: Testiranje vrednosti v LIP

test	izhod	naloga	seznami
(0,0)	1+	(0,0) v LSP	LIP={(0,1), (1,0), (1,1)} LSP={(0,0)}
(0,1)	1-	(0,1) to LSP	LIP={(1,0), (1,1)} LSP={(0,0), (0,1)}
(1,0)	0	/	
(1,1)	0	/	

3. Po testiranju koeficientov pričnemo testirati množice po vrstnem redu, kot se nahajajo v LIS. Tako $\mathcal{D}(0,1)$, na primer, predstavlja množico dvajsetih koeficientov {(0,2), (0,3), (1,2), (1,3), (0,4), (0,5), (0,6), (0,7),(1,4), (1,5), (1,6), (1,7),(2,4), (2,5), (2,6), (2,7),(3,4), (3,5), (3,6), (3,7),}. Ker je $\mathcal{D}(0,1)$ pomemben, SPIHT testira štiri potomce {(0,2), (0,3), (1,2), (1,3)}, kot vidimo v razpredelnici 4.2.

   Ko smo preverili vse potomce, premaknemo (0,1) na konec LIS in spremenimo njegov tip na B, kot vidimo v zadnji vrstici razpredelnice 4.2. S tem $\mathcal{D}(0,1)$ spremenimo v $\mathcal{L}(0,1)$, to je iz množice vseh naslednikov v množico vseh naslednikov brez potomcev.

4. Enak postopek kot prej uporabimo za množico $\mathcal{D}(1,0)$ (glej razpredelnico 4.3). Kljub temu da noben potomec ni pomemben, je $\mathcal{D}(1,0)$ pomemben, ker je eden izmed naslednikov pomemben (tisti na položaju (4,3)). Na izhod pošljemo 1, testiramo potomce in nazadnje množico kot tip B uvrstimo na konec LIS.

Tabela 4.2: Obdelava množice $\mathcal{D}(0,1)$

test	izhod	naloga	seznami
$\mathcal{D}(0,1)$	1	test potomcev	LIS ={(0,1)A, (1,0)A, (1,1)A }
(0,2)	1+	(0,2) v LSP	LSP={(0,0), (0,1), (0,2)}
(0,3)	0	(0,3) v LIP	LIP={(1,0), (1,1), (0,3)}
(1,2)	0	(1,2) v LIP	LIP={(1,0), (1,1), (0,3), (1,2)}
(1,3)	0	(1,3) v LIP	LIP={(1,0), (1,1), (0,3), (1,2), (1,3)}
		sprememba tipa	LIS ={(1,0)A, (1,1)A,(0,1)B }

Tabela 4.3: Obdelava množice $\mathcal{D}(1,0)$

test	izhod	naloga	seznami
$\mathcal{D}(1,0)$	1	test potomcev	LIS ={(1,0)A, (1,1)A, (0,1)B }
(2,0)	0	(2,0) v LIP	LIP={(1,0), (1,1), (0,3), (1,2), (1,3), (2,0)}
(2,1)	0	(2,1) v LIP	LIP={(1,0), (1,1), (0,3), (1,2), (1,3), (2,0) (2,1)}
(3,0)	0	(3,0) v LIP	LIP = {(1,0), (1,1), (0,3), (1,2), (1,3), (2,0) (2,1), (3,0)}
(3,1)	0	(3,1) v LIP	LIP = {(1,0), (1,1), (0,3), (1,2), (1,3), (2,0) (2,1), (3,0)}, (3,1)
		sprememba tipa	LIS ={(1,1)A,(0,1)B, (1,0)B }

5. Množica $\mathcal{D}(1,1)$ je nepomembna, kar označimo z bitom 0, sicer pa ne storimo ničesar (gled razpredelnico 4.4).

Tabela 4.4: Obdelava množice $\mathcal{D}(1,1)$

test	izhod	naloga	seznami
$\mathcal{D}(1,1)$	0	/	LIS ={(1,1)A, (0,1)B, (1,0)B }

6. Naslednji element v LIS je (0,1). Ker je tipa B, testiramo potomce brez naslednikov (16 koeficientov v zgornjem desnem kvadrantu na sliki 4.16). Ker so vsi koeficienti nepomembni, damo na izhod bit 0, sicer pa ne storimo ničesar (razpredelnica 4.5).

7. Testiramo še množico $\mathcal{L}(1,0)$, ki pa je pomembna, zato množico razdelimo v $\mathcal{D}(2,0)$, $\mathcal{D}(2,1)$, $\mathcal{D}(3,0)$ in $\mathcal{D}(3,1)$ ter jih kot tip A vstavimo v LIS, od koder pa odstranimo $(1,0)$B (glej razpredelnico 4.6).

Tabela 4.5: Obdelava množice $\mathcal{L}(0,1)$

test	izhod	naloga	seznami
$\mathcal{L}(0,1)$	0	/	LIS ={(1,1)A, (0,1)B, (1,0)B }

Tabela 4.6: Obdelava množice $\mathcal{L}(1,0)$

test	izhod	naloga	seznami
$\mathcal{L}(1,0)$	1	vstavi množice	LIS ={(1,1)A, (0,1)B, (2,0)A, (2,1)A, (3,0)A, (3,1)A}

8. Algoritem nadaljuje z obdelavo množic v LIS po vrstnem redu. Najprej vzame množico $\mathcal{D}(2,0)$. Množica je nepomembna, na izhod damo bit 0 in ne storimo nečesar drugega (razpredelnica 4.7).

Tabela 4.7: Obdelava množice $\mathcal{D}(2,0)$

test	izhod	naloga	seznami
$\mathcal{D}(2,0)$	0	/	LIS ={(1,1)A, (0,1)B, (2,0)A, (2,1)A, (3,0)A, (3,1)A,}

9. Zatem preverimo množico $\mathcal{D}(2,1)$. Ta množica sestoji iz koeficientov $(4,2)$, $(4,3)$, $(5,2)$, $(5,3)$ in je pomembna. Zato testiramo potomce. Nepomembne vstavimo v LIP, pomembnega na položaju $(4,3)$ pa v LSP, kot kaže razpredelnica 4.8. Ker je $\mathcal{L}(2,1) = \{\}$, vnos $(2,1)$A odstranimo iz LIS (namesto da bi mu spremenili tip v B).

10. Ostaneta nam še dve množici $\mathcal{D}(3,0)$ in $\mathcal{D}(3,1)$. Množica $\mathcal{D}(3,0) = \{(6,0),(6,1),(7,0),(7,1)\}$ je nepomembna, prav tako tudi množica $\mathcal{D}(3,1) = \{(6,2),(6,3),(7,2),(7,3)\}$, zato ne naredimo ničesar. S tem se korak urejanja konča in stanje seznamov vidimo v razpredelnici 4.9. Ker gre za prvi korak urejanja, koraka izboljšav ni mogoče opraviti, zato vstopimo v novo iteracijo in dekrementiramo $n = 4$.

11. SPIHT spet najprej preveri pomembnosti pikslov v LIP. Dobimo razpredelnico 4.10:

12. Pričnemo s testiranjem po vrstnem redu množic v LIS. Množica $\mathcal{D}(1,1)$ je tipa A in je prva na vrsti. Sestoji iz naslednjih koeficientov {(2,2), (2,3), (3,2), (3,3), (4,4), (4,5), (4,6), (4,7),(5,4), (5,5), (5,6), (5,7),(6,4),

Tabela 4.8: Obdelava množice $\mathcal{D}(2,1)$

test	izhod	naloga	seznami
$\mathcal{D}(2,1)$	1	test potomcev	LIS ={(1,1)A, (0,1)B, (2,0)A, (2,1)A, (3,0)A, (3,1)A}
(4,2)	0	(4,2) v LIP	LIP={(1,0), (1,1), (0,3), (1,2), (1,3), (2,0)} (2,1), (3,0)}, (3,1), (4,2)}
(4,3)	1+	(4,3) v LSP	LSP={(0,0), (0,1), (0,2), (4,3)}
(5,2)	0	(5,2) v LIP	LIP = {(1,0), (1,1), (0,3), (1,2), (1,3), (2,0) (2,1), (3,0)}, (3,1), (4,2), (5,2)}
(5,3)	0	(5,3) v LIP	LIP = {(1,0), (1,1), (0,3), (1,2), (1,3), (2,0) (2,1), (3,0)}, (3,1), (4,2), (5,2), (5,3)}
		odstranimo $(2,1)$	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}

Tabela 4.9: Obdelava množic $\mathcal{D}(3,0)$ in $\mathcal{D}(3,1)$ in stanje množic po koraku urejanja

test	izhod	naloga	seznami
$\mathcal{D}(3,0)$	0	/	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}
$\mathcal{D}(3,1)$	0	/	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}
			LSP={(0,0), (0,1), (0,2), (4,3)}
			LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}
			LIP={(1,0), (1,1), (0,3), (1,2), (1,3), (2,0) (2,1), (3,0), (3,1), (4,2), (5,2), (5,3)}

(6,5), (6,6), (6,7),(7,4), (7,5), (7,6), (7,7)}. Množica $\mathcal{D}(1,1)$ je nepomembna, zato pošljemo na izhod 0, sicer pa ne storimo ničesar (glej razpredelnico 4.11).

13. Zatem testiramo množico $\mathcal{L}(0,1)$, ki sestoji iz koeficientov {(0,4), (0,5), (0,6), (0,7), (1,4), (1,5), (1,6), (1,7), (2,4), (2,5), (2,6), (2,7), (3,4), (3,5), (3,6), (3,7)}. Vsi koeficienti so nepomembni, zato pošljemo na izhod 0 in ne storimo ničesar (glej razpredelnico 4.12).

14. Naslednja testirana množica je $\mathcal{D}(2,0)$. Sestoji iz koeficientov {((4,0), (4,1), (5,0), (5,1)}. Vsi koeficienti so nepomembni, zato pošljemo na izhod 0, sicer ne storimo ničesar (razpredelnica 4.13).

15. Ostaneta še množici $\mathcal{D}(3,0)$ in $\mathcal{D}(3,1)$. Množica $\mathcal{D}(3,0) = \{(6,0),$

Tabela 4.10: Testiranje vrednosti v LIP v koraku 2

test	izhod	naloga	seznami
(1,0)	1-	premik (1,0) v LSP	LSP={(0,0), (0,1), (0,2), (4,3), (1,0)}
(1,1)	1+	premik (1,1) v LSP	LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1)}
(0,3)	0	/	
(1,2)	0	/	
(1,3)	0	/	
(2,0)	0	/	
(2,1)	0	/	
(3,0)	0	/	
(3,1)	0	/	
(4,2)	0	/	
(5,2)	0	/	
(5,3)	0	/	

Tabela 4.11: Obdelava množice $\mathcal{D}(1,1)$

test	izhod	naloga	seznami
$\mathcal{D}(1,1)$	0	/	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}

Tabela 4.12: Obdelava množice $\mathcal{L}(0,1)$

test	izhod	naloga	seznami
$\mathcal{L}(0,1)$	0	/	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}

Tabela 4.13: Obdelava množice $\mathcal{D}(2,0)$

test	izhod	naloga	seznami
$\mathcal{D}(2,0)$	0	/	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}

(6,1), (7,0), (7,1)} in množica $\mathcal{D}(3,1) = \{(6,2), (6,3), (7,2), (7,3)\}$. Tudi tokrat so vsi koeficienti množic nepomembni, zato pošljemo na izhod dve 0, sicer pa ne storimo ničesar (razpredelnica 4.14).

S tem končamo korak urejanja in preidemo na korak izboljšav. Obiščemo vse elemente v seznamu LSP, razen tistih, ki smo jih dodali v zadnjem koraku (v tem koraku smo v LSP dodali 2 koeficienta), in pošljemo na izhod n-ti najpomembnejši bit (5-ti bit v našem pri-

Tabela 4.14: Obdelava množic $\mathcal{D}(3,0)$ in $\mathcal{D}(3,1)$

test	izhod	naloga	seznami
$\mathcal{D}(3,0)$	0	/	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}
$\mathcal{D}(3,1)$	0	/	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}

meru). Absolutne vrednosti koeficientov v LSP so: { $63 = 1\underline{1}11111_2$, $34 = 1\underline{0}0010_2$, $49 = 1\underline{1}0001_2$, $47 = 1\underline{0}1111_2$, $31 = 11111_2$, $23 = 10111_2$}. Zato pošljemo na izhod 1010.

Stanje množic pred naslednjo iteracijo kaže razpredelnica 4.15.

Tabela 4.15: Stanje množic po drugem koraku

LSP ={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1)}
LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}
LIP ={(0,3), (1,2), (1,3), (2,0) (2,1), (3,0), (3,1), (4,2), (5,2), (5,3)}

16. V naslednji iteraciji zmanjšamo $n = 3$; s tem postanejo pomembne vrednosti med $8 \leq n < 16$. Prelet seznama LIP nam da stanje v razpredelnici 4.16:

Tabela 4.16: Testiranje vrednosti v LIP v koraku 3

test	izhod	naloga	seznami
(0,3)	1+	premik (0,3) v LSP	LSP = LSP ∪ (0,3)
(1,2)	1+	premik (1,2) v LSP	LSP = LSP ∪ (1,2)
(1,3)	1-	premik (0,3) v LSP	LSP = LSP ∪ (1,3)
(2,0)	1+	premik (2,0) v LSP	LSP = LSP ∪ (2,0)
(2,1)	1+	premik (2,1) v LSP	LSP = LSP ∪ (2,1)
(3,0)	1-	premik (3,0) v LSP	LSP = LSP ∪ (3,0)
(3,1)	0	/	
(4,2)	0	/	
(5,2)	0	/	
(5,3)	0	/	

LSP = {(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0)}
LIP = {(3,1), (4,2), (5,2), (5,3)}
LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}

17. V naslednjih iteracije preverimo pomembnost množice $\mathcal{D}(1,1)$. Množica je pomembna, zato testiramo štiri potomce na položajih {(2,2), (2,3), (3,2), (3,3)}. Ker množica $\mathcal{L}(1,1)$ ni prazna (vsebuje 16 koeficientov spodnjega desnega kvadranta), jo premaknemo na konec seznama LIS kot tip B (razpredelnica 4.17).

Tabela 4.17: Obdelava množice $\mathcal{D}(1,1)$

test	izhod	naloga	seznami
$\mathcal{D}(1,1)$	1	test potomcev	LIS ={(1,1)A, (0,1)B, (2,0)A, (3,0)A, (3,1)A}
(2,2)	0	(2,2) v LIP	LIP= LIP $\cup$ (2,2)
(2,3)	1-	(2,3) v LSP	LSP= LSP $\cup$ (2,3)
(3,2)	1-	(3,2) v LSP	LSP= LSP $\cup$ (3,2)}
(3,3)	1+	(3,3) v LSP	LSP= LSP $\cup$ (3,3)}
		sprememba tipa	LIS ={(0,1)B, (2,0)A, (3,0)A, (3,1)A, (1,1)B }

Stanje v seznamih LIP in LSP pa je naslednje:
LIP={(3,1),(4,2),(5,2),(5,3),(2,2)} in
LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3)}.

18. Premaknemo se na množico $\mathcal{L}(0,1)$. Množica je pomembna, zato damo na izhod 1, $\mathcal{L}(0,1)$ pa razdelimo v množice $\mathcal{D}(0,2)$, $\mathcal{D}(0,3)$, $\mathcal{D}(1,2)$ in $\mathcal{D}(1,3)$ tipa A in jih vstavimo na konec LIS, od kod pa odstranimo množico (1,0B). Stanje v LIS je naslednje:

LIS= {(2,0)A, (3,0)A, (3,1)A, (1,1)B, (0,2)A, (0,3)A, (1,2)A, (1,3)A}.

19. Sledi množica $\mathcal{D}(2,0)$, ki sestoji iz koeficientov {(4,0), (4,1), (5,0), (5,1)}. Množica je pomembna, zato pošljemo na izhod 1 in obdelamo koeficiente, kot kaže razpredelnica 4.18. Ker je $\mathcal{L}(2,0)$ prazna, je ne stavimo v LIS.

Stanje v seznamih je naslednje:
LIP={(3,1),(4,2),(5,2),(5,3),(2,2), (4,0), (5,0), (5,1)} in
LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3), (4,1)}
LIS= {(3,0)A, (3,1)A, (1,1)B, (0,2)A, (0,3)A, (1,2)A, (1,3)A}

20. Nato testiramo množico $\mathcal{D}(3,0)$, ki določa koeficiente {(6,0), (6,1), (7,0), (7,1)}. Koeficient na položaju (7,1) je pomemben, zato je

Tabela 4.18: Obdelava množice $\mathcal{D}(2,0)$

test	izhod	naloga	seznami
$\mathcal{D}(2,0)$	1	test potomcev	
(4,0)	0	(4,0) v LIP	LIP= LIP $\cup$ (4,0)
(4,1)	1+	(4,1) v LSP	LSP= LSP $\cup$ (4,1)
(5,0)	0	(5,0) v LIP	LIP= LIP $\cup$ (5,0)}
(5,1)	0	(5,1) v LIP	LIP= LIP $\cup$ (5,1)}
		$\mathcal{L}(2,0)$={}	LIS = LIS - $\mathcal{D}(2,0)$

množica pomembna in testiramo njene koeficiente (glej razpredelnico 4.19). Množico $\mathcal{D}(3,0)$ zbrišemo, množice $\mathcal{L}(3,0)$pa ne vstavimo, ker je prazna.

Tabela 4.19: Obdelava množice $\mathcal{D}(3,0)$

test	izhod	naloga	seznami
$\mathcal{D}(3,0)$	1	test potomcev	
(6,0)	0	(6,0) v LIP	LIP= LIP $\cup$ (6,0)
(6,1)	0	(6,1) v LIP	LIP= LIP $\cup$ (6,1)
(7,0)	0	(7,0) v LIP	LIP= LIP $\cup$ (7,0)}
(7,1)	1+	(7,1) v LSP	LSP= LSP $\cup$ (7,1)}
		$\mathcal{L}(3,0)$={}	LIS = LIS - $\mathcal{D}(3,0)$

Po tem koraku je stanje v seznamih sledeče:
LIP={(3,1),(4,2),(5,2),(5,3),(2,2), (4,0), (5,0), (5,1), (6,0), (6,1), (7,0)}
in
LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3), (4,1), (7,1)}
LIS= {(3,1)A, (1,1)B, (0,2)A, (0,3)A, (1,2)A, (1,3)A}

21. Množica $\mathcal{D}(3,1)$ določa koeficiente {(6,2), (6,3), (7,2), (7,3)} in je nepomembna, zato pošljemo na izhod 0, sicer pa ne storimo ničesar.

22. Množica $\mathcal{L}(1,1)$ določa 16 koeficientov v spodnjem desnem delu matrike. Vsi koeficienti so nepomembni, zato pošljemo na izhod 0 in se premaknemo na naslednjo množico.

23. Množica $\mathcal{D}(0,2)$ sestoji iz koeficientov {(0,4), (0, 5), (1, 4), (1,5)} in je pomembna (glej razpredelnico 4.20).

Tabela 4.20: Obdelava množice $\mathcal{D}(0,2)$

test	izhod	naloga	seznami
$\mathcal{D}(0,2)$	1	test potomcev	
(0,4)	0	(0,4) v LIP	LIP= LIP $\cup$ (0,4)
(0,5)	1+	(0,5) v LSP	LSP= LSP $\cup$ (0,5)
(1,4)	0	(1,4) v LIP	LIP= LIP $\cup$ (1,4)}
(1,5)	0	(1,5) v LIP	LIP= LIP $\cup$ (1,5)}
		$\mathcal{L}(0,2)$={}	LIS = LIS - $\mathcal{D}(0,2)$

V seznamih imamo naslednje elemente:
LIP={(3,1),(4,2),(5,2),(5,3),(2,2), (4,0), (5,0), (5,1), (6,0), (6,1), (7,0), (0,4), (1, 4), (1,5)} in
LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3), (4,1), (7,1), (0,5)}
LIS= {(3,1)A, (1,1)B, (0,3)A, (1,2)A, (1,3)A}

24. Tudi naslednja množica $\mathcal{D}(0,3)$ je pomembna. Sestoji iz naslednjih koeficientov: {(0,6), (0,7), (1,6), (1,7)}, ki jih obdelamo, kot kaže razpredelnica 4.21.

Tabela 4.21: Obdelava množice $\mathcal{D}(0,3)$

test	izhod	naloga	seznami
$\mathcal{D}(0,3)$	1	test potomcev	
(0,6)	1-	(0,6) v LSP	LSP= LSP $\cup$ (0,6)
(0,7)	0	(0,7) v LIP	LIP= LIP $\cup$ (0,7)
(1,6)	0	(1,6) v LIP	LIP= LIP $\cup$ (1,6)}
(1,7)	0	(1,7) v LIP	LIP= LIP $\cup$ (1,7)}
		$\mathcal{L}(0,3)$={}	LIS = LIS - $\mathcal{D}(0,3)$

Seznami so:
LIP={(3,1),(4,2),(5,2),(5,3),(2,2), (4,0), (5,0), (5,1), (6,0), (6,1), (6,2), (0,4), (1, 4), (1,5), (0,7), (1,6), (1,7))} in
LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3), (4,1), (7,1), (0,5), (0,6)}
LIS= {(3,1)A, (1,1)B, (1,2)A, (1,3)A}

25. Množica $\mathcal{D}(1,2)$, določena s koeficienti {(2,4), (2, 5), (3,4), (3,5) } je nepomembna, zato pošljemo na izhod 0, sicer pa ne storimo ničesar.

26. Množica $\mathcal{D}(1,3)$, določena s koeficienti {(2,6), (2, 7), (3,6), (3,7) } pa je pomembna, zato postopamo, kot kaže razpredelnica 4.22.

Tabela 4.22: Obdelava množice $\mathcal{D}(1,3)$

test	izhod	naloga	seznami
$\mathcal{D}(1,3)$	1	test potomcev	
(2,6)	0	(2,6) v LIP	LIP= LIP $\cup$ (2,6)
(2,7)	1+	(2,7) v LSP	LSP= LSP $\cup$ (2,7)
(3,6)	0	(3,6) v LIP	LIP= LIP $\cup$ (3,6)}
(3,7)	0	(3,7) v LIP	LIP= LIP $\cup$ (3,7)}
		$\mathcal{L}(1,3)$={}	LIS = LIS - $\mathcal{D}(1,3)$

S tem smo zaključili korak urejanja, ki nam je dal naslednje stanje seznamov:

LIP={(3,1),(4,2),(5,2),(5,3),(2,2), (4,0), (5,0), (5,1), (6,0), (6,1), (7,0), (0,4), (1, 4), (1,5), (0,7), (1,6), (1,7), (2,6), (3,6), (3, 7)},
LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3), (4,1), (7,1), (0,5), (0,6), (2,7)}
LIS= {(3,1)A, (1,1)B, (1,2)A}.

Sledi še korak izboljšav, kjer pošljemo na izhod bite 100110. Nato zmanjšamo n in pričnemo z naslednjo iteracijo.

27. V tem koraku so pomembne vrednosti med $4 \leq n < 8$. Najprej analiziramo seznam LIP (glej razpredelnico 4.23):

V seznamu pomembnih koeficientov je sedaj stanje naslednje: LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3), (4,1), (7,1), (0,5), (0,6), (2,7), (3,1), (4,0), (7,0), (0,4), (1,5), (0,7), (1,6)}

28. Pričnemo s preverjanjem množic v LIS. Prva med njimi je $\mathcal{D}(3,1)$. Množica je pomembna, zato testiramo vse njene potomce {(6,2), (6,3) (7,2) (7,3) } kot kaže razpredelnica 4.24. $\mathcal{L}(3,1) = \{\}$, zato odstranimo množico iz LIS.

29. Naslednja množica iz LIS je $\mathcal{L}(1,1)$, ki sestoji iz 16 koeficientov v spodnjem desnem kvadrantu matrike. Množica je pomembna, zato njene potomce {(2,2), (2,3), (3,2), (3,3)} kot tip A vstavimo v LIS, samo

Tabela 4.23: Testiranje vrednosti v LIP v koraku 4

test	izhod	naloga	seznami
(3,1)	1-	premik (3,1) v LSP	LSP = LSP $\cup$ (3,1)
(4,2)	0	/	
(5,2)	0	/	
(5,3)	0	/	
(2,2)	0	/	
(4,0)	1-	premik (4,0) v LSP	LSP = LSP $\cup$ (4,0)
(5,0)	0	/	
(5,1)	0	/	
(6,0)	0	/	
(6,1)	0	/	
(7,0)	1+	premik (7,0) v LSP	LSP = LSP $\cup$ (7,0)
(0,4)	1+	premik (0,4) v LSP	LSP = LSP $\cup$ (0,4)
(1,4)	0	/	
(1,5)	1+	premik (1,5) v LSP	LSP = LSP $\cup$ (1,5)
(0,7)	1+	premik (0,7) v LSP	LSP = LSP $\cup$ (0,7)
(1,6)	1+	premik (1,6) v LSP	LSP = LSP $\cup$ (1,6)
(1,7)	0	/	
(2,6)	0	/	
(3,6)	0	/	
(3,7)	0	/	

Tabela 4.24: Obdelava množice $\mathcal{D}(3,1)$

test	izhod	naloga	seznami
$\mathcal{D}(3,1)$	1	test potomcev	
(6,2)	1+	(6,2) v LSP	LSP= LSP $\cup$ (6,2)
(6,3)	1-	(6,3) v LSP	LSP= LSP $\cup$ (6,3)
(7,2)	1+	(7,2) v LSP	LSP= LSP $\cup$ (7,2)
(7,3)	1+	(7,3) v LSP	LSP= LSP $\cup$ (7,3)
		$\mathcal{L}(3,1)=\{\}$	LIS = LIS - $\mathcal{D}(3,1)$

množico $\mathcal{L}(1,1)$ pa iz LIS odstranimo. Rezultat kaže razpredelnica 4.25.

30. Tudi množica $\mathcal{D}(1,2)$, ki je naslednja v LIS, je pomembna, zato testiramo njene potomce kot kaže razpredelnica 4.26.

Tabela 4.25: Obdelava množice $\mathcal{L}(1,1)$

test	izhod	naloga	seznami
$\mathcal{L}(1,1)$	1	vstavi množice	LIS= {(1,2)A, (2,2)A, (2,3)A, (3,2)A, (3,3)A}

Tabela 4.26: Obdelava množice $\mathcal{D}(1,2)$

test	izhod	naloga	seznami
$\mathcal{D}(1,2)$	1	test potomcev	
(2,4)	1+	(2,4) v LSP	LSP= LSP $\cup$ (2,4)
(2,5)	1-	(2,5) v LSP	LSP= LSP $\cup$ (2,5)
(3,4)	1+	(3,4) v LSP	LSP= LSP $\cup$ (3,4)}
(3,5)	0	(3,5) v LIP	LIP= LIP $\cup$ (3,5)
		$\mathcal{L}(1,2)$={}	LIS = LIS - $\mathcal{D}(1,2)$

31. Testiramo množico $\mathcal{D}(2,2)$, ki je pomembna (glej razpredelnico 4.27).

Tabela 4.27: Obdelava množice $\mathcal{D}(2,2)$

test	izhod	naloga	seznami
$\mathcal{D}(2,2)$	1	test potomcev	
(4,4)	1+	(4,4) v LSP	LSP= LSP $\cup$ (4,4)
(4,5)	1+	(4,5) v LSP	LSP= LSP $\cup$ (4,5)
(5,4)	0	(5,4) v LIP	LIP= LIP $\cup$ (5,4)
(5,5)	0	(5,5) v LIP	LIP= LIP $\cup$ (5,5)
		$\mathcal{L}(2,2)$={}	LIS = LIS - $\mathcal{D}(2,2)$

32. Testiramo množico $\mathcal{D}(2,3)$, ki je tudi pomembna (glej razpredelnico 4.28).

33. Nadaljujemo z množico $\mathcal{D}(3,2)$, ki je tudi pomembna (glej razpredelnico 4.29).

34. Tudi zadnja množica v tem koraku $\mathcal{D}(3,3)$ je pomembna, zato jo obdelamo na enak način kot množice v prejšnjem koraku (razpredelnica 4.30).

    Ker smo vse množice v LIS obdelali, vstopimo v korak izboljšav: Tokrat na izhod pošljemo 10011101111011000110. Dekrementiramo n,

Tabela 4.28: Obdelava množice $\mathcal{D}(2,3)$

test	izhod	naloga	seznami
$\mathcal{D}(2,3)$	1	test potomcev	
(4,6)	0	(4,6) v LIP	LIP= LIP $\cup$ (4,6)
(4,7)	0	(4,7) v LIP	LIP= LIP $\cup$ (4,7)
(5,6)	0	(5,6) v LIP	LIP= LIP $\cup$ (5,6)
(5,7)	1+	(5,7) v LSP	LSP= LSP $\cup$ (5,7)
		$\mathcal{L}(2,3)$={}	LIS = LIS - $\mathcal{D}(2,3)$

Tabela 4.29: Obdelava množice $\mathcal{D}(3,2)$

test	izhod	naloga	seznami
$\mathcal{D}(3,2)$	1	test potomcev	
(6,4)	0	(6,4) v LIP	LIP= LIP $\cup$ (6,4)
(6,5)	1+	(6,5) v LSP	LSP= LSP $\cup$ (6,5)
(7,4)	0	(7,4) v LIP	LIP= LIP $\cup$ (7,4)
(7,5)	0	(7,5) v LIP	LIP= LIP $\cup$ (7,5)
		$\mathcal{L}(3,2)$={}	LIS = LIS - $\mathcal{D}(3,2)$

Tabela 4.30: Obdelava množice $\mathcal{D}(3,3)$

test	izhod	naloga	seznami
$\mathcal{D}(3,3)$	1	test potomcev	
(6,6)	0	(6,6) v LIP	LIP= LIP $\cup$ (6,6)
(6,7)	1+	(6,7) v LSP	LSP= LSP $\cup$ (6,7)
(7,6)	1-	(7,6) v LSP	LSP= LSP $\cup$ (7,6)
(7,7)	1+	(7,7) v LSP	LSP= LSP $\cup$ (7,7)
		$\mathcal{L}(3,3)$={}	LIS = LIS - $\mathcal{D}(3,3)$

tako da so sedaj pomembni koeficienti $2 \leq n < 4$. Stanje množic po tem koraku je naslednje:

LIP={(4,2),(5,2),(5,3),(2,2), (5,0), (5,1), (6,0), (6,1), (1,4), (1,7), (2,6), (3,6), (3,7), (3,5), (5,4), (5,5), (4,6), (4,7), (5,6), (6,4), (7,4), (7,5), (6,6)},

LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3), (4,1), (7,1), (0,5), (0,6), (2,7), (3,1),

(4,0), (7,0), (0,4), (1,5), (0,7), (1,6), 6,2), (6,3), (7,2), (7,3), (2,4), (2,5), (3,4), (4,4), (4,5), (5,7), (6,5), (6,7), (7,6), (7,7)}

LIS= {}.

Obdelava seznama LIP je prikazana v razpredelnici 4.31:

Tabela 4.31: Testiranje vrednosti v LIP v koraku 5

test	izhod	naloga	seznami
(4,2)	0	/	
(5,2)	1-	premik (5,2) v LSP	LSP = LSP ∪ (5,2)
(5,3)	1+	premik (5,3) v LSP	LSP = LSP ∪ (5,3)
(2,2)	1+	premik (2,2) v LSP	LSP = LSP ∪ (2,2)
(5,0)	1+	premik (5,0) v LSP	LSP = LSP ∪ (5,0)
(5,1)	0	/	
(6,0)	1+	premik (6,0) v LSP	LSP = LSP ∪ (6,0)
(6,1)	1-	premik (6,1) v LSP	LSP = LSP ∪ (6,1)
(1,4)	1+	premik (1,4) v LSP	LSP = LSP ∪ (1,4)
(1,7)	0	/	
(2,6)	1+	premik (2,6) v LSP	LSP = LSP ∪ (2,6)
(3,6)	1+	premik (3,6) v LSP	LSP = LSP ∪ (3,6)
(3,7)	1+	premik (3,7) v LSP	LSP = LSP ∪ (3,7)
(3,5)	1-	premik (3,5) v LSP	LSP = LSP ∪ (3,5)
(5,4)	1+	premik (5,4) v LSP	LSP = LSP ∪ (5,4)
(5,5)	1-	premik (5,5) v LSP	LSP = LSP ∪ (5,5)
(4,6)	1-	premik (4,6) v LSP	LSP = LSP ∪ (4,6)
(4,7)	1+	premik (4,7) v LSP	LSP = LSP ∪ (4,7)
(5,6)	0	/	
(6,4)	1+	premik (6,4) v LSP	LSP = LSP ∪ (6,4)
(7,4)	0	/	
(7,5)	1+	premik (7,5) v LSP	LSP = LSP ∪ (7,5)
(6,6)	1+	premik (6,6) v LSP	LSP = LSP ∪ (6,6)

Ker je LIS={}, preidemo takoj na korak izboljšav in pošljemo na izhod 1101111101100100100010010111001010010 1100. Zmanjšamo n. Pomembne vrednosti so sedaj med $1 \leq n < 2$. Stanje seznamov po tem koraku je naslednje:

LIP={(4,2),(5,1), (1,7), (5,6), (7,4)},

LSP={(0,0), (0,1), (0,2), (4,3), (1,0), (1,1), (0,3), (1,2), (1,3), (2,0), (2,1), (3,0), (2,3), (3,2), (3,3), (4,1), (7,1), (0,5), (0,6), (2,7), (3,1), (4,0), (7,0), (0,4), (1,5), (0,7), (1,6), 6,2), (6,3), (7,2), (7,3), (2,4), (2,5), (3,4), (4,4), (4,5), (5,7), (6,5), (6,7), (7,6), (7,7), (5,2),(5,3),(2,2), (5,0), (6,0), (6,1), (1,4), (2,6), (3,6), (3,7), (3,5), (5,4), (5,5), (4,6), (4,7), (6,4), (7,5), (6,6)}

LIS= {}.

35. Tudi tokrat preverimo samo seznam LIP (razpredelnica 4.32).

Tabela 4.32: Testiranje vrednosti v LIP v koraku 6

test	izhod	naloga	seznami
(4,2)	1-	premik (4,2) v LSP	LSP = LSP ∪ (4,2)
(5,1)	0	/	
(1,7)	1-	premik (1,7) v LSP	LSP = LSP ∪ (1,7)
(5,6)	0	/	
(7,4)	0	/	

Korak izboljšav pošlje na izhod 59 bitov 10111100110100011101111101000101100000000010110111100100 0111, nato pa se algoritem zaključi.

4.3.2 JPEG2000 - Pregled

Marca 1997 so pod okriljem ITU in ISO (ISO/IEC/JTC1/SC29/WG1) sestavili novo delovno skupino za razvoj novega standarda za stiskanje bitnih slik. Končni predlog je skupina pripravila aprila 2000 in ga poimenovala JPEG2000. Osnovni pregled standarda najdemo v [17] Namen standarda je dopolnjevanje obstoječih standardov in ne njihova zamenjava.

JPEG2000 je zasnovan zelo splošno in je namenjen stiskanju bitnih slik z različnimi karakteristikami (črno-bele, sivinske in barvne), pri čemer je uspešen tudi pri zelo velikih faktorjih stiskanja (1:200).

Če ga primerjamo z njegovim predhodnikom, standardom JPEG, JPEG2000 nudi naslednje prednosti, ki jih bomo nekoliko podrobjene opisali v nadaljevanju:

- zanimiva področja (angl. regio of interest, ROI),
- neobčutljivost na napake (angl. error resilience),

- napredujoče razvrščanje (angl. progression orders),
- enoten izgubni in brezizgubni sistem,
- boljša kakovost stiskanja pri velikih faktorjih stiskanja,
- boljši pri sestavljenih slikah in grafiki.

JPEG2000 omogoča pridobiti različne ločljivosti ali ROI iz enega bitnega niza. S tem lahko neka informacija pridobi le tiste podatke, ki jih potrebuje.

4.3.2.1 Aplikacije JPEG2000

JPEG2000 ima zelo široko področje uporabe. Zahteve posameznih področij povzema razpredelnica 4.33.

slika	velikost	število kanalov	globina kanalov
internet	32×32 do 4000 × 4000	1, 3, 4	1 do 8
tisk	4800 × 6600	1, 3, 4	8
prebiranje	10k × 10k do 20k × 20k	1, 3, 4	16
fotografija	do 4k × 4k	1, 3	8 do 16
daljinsko zaznavanje	do 24k vrstic	1 do 500	8 do 20
mobilne aplikacije	32 × 32 do 4k × 4k	1, 3	1 do 8
medicina	32 × 32 do 10k × 10k	1, 3, 4	do 16
digitalne knjižnice	32 × 32 do 4k × 4k	1, 3, 4	1 do 8

Tabela 4.33: Aplikacije JPEG2000

4.3.2.2 Prednosti JPEG2000

JPEG2000 nudi veliko lastnosti, ki so vitalnega pomena za množico različnih modernih aplikacij, ki izkoriščajo prednosti moderne tehnologije. JPEG2000 zapolnjuje praznine na tistih področjih, kjer obstoječi standardi niso uspešni, ali pa aplikacije do sedaj sploh niso uporabljale standarda. Lastnosti standarda so:

- **Izjemno veliko razmerje stiskanja.** JPEG2000 ima zelo ugodno razmerje stiskanja, ki presega 0.25bpp (bit per pixel). Mnogo današnjih aplikacij potrebuje to lastnost (na primer, prenos slik preko omrežja, daljinsko zaznavanje). Primer na spodnji sliki kaže moč JPEG2000 napram JPEG pri 0.125bpp. Slika, ki jo rekonstruiramo

z JPEG2000 je še sprejemljiva, kot tista dobljena iz JPEG pa je zelo slabe kakovosti (glej sliki 4.17).

a)

b)

Slika 4.17: Primerjava JPEG (a) in JPEG2000 (b) pri 0.125 bpp

- **Zanimiva področja** (angl. regions of interest, ROI). Določena zanimiva področja znotraj slike so kodirane v boljši kakovosti, kot preostanek slike. To je najpomembnejša lastnost novega standarda. Naključni dostop v bitni niz nam omogoča, da so ROI tudi dekodirani najprej. Zanimivo področje je lahko samo eno, lahko pa jih je več, lahko so poljubne velikosti (znotraj slike seveda) in oblike. Slika 4.18kaže primer ROI.

- **Stiskanje črnobelih in barvnih slik.** JPEG2000 učinkovito stiska tako črnobele in barvne slike. Sistem stiska slike različnih globin (od 1 do 16 bitov) za vsako barvno komponento. Na ta način je sistem uspešen pri stiskanju slik s tekstom, pri medicinskih slikah z označenimi prekrivki, pri prozornih slikah, prosojnicah in faksimilih.

Slika 4.18: JPEG2000 in ROI v obliki kroga

- **Stiskanje z in brez izgubami.** JPEG2000 uporablja dva valčka. Valček *daub97* vsebuje koeficiente v aritmetiki s plavajočo vejico in ga uporabljamo pri kodiranju z izgubami. Celoštevilski valček 5/3 pa je uporaben tako za izgubno kot brezizgubno stiskanje. Tipična uporaba teh zmožnosti je pri medicinskih slikah, kjer izgube niso dobro sprejete, aplikacije za arhiviranje slik, kjer zahtevamo visoko kakovost hranjenja, sam prikaz pa je za potrebe predogleda lahko slabši, za aplikacije, kjer imajo izhodne naprave različne zmožnosti in za predtisk (angl. prepress imagery).

- **Napredujoče stiskanje tako po kakovosti kot po ločljivosti.** JPEG2000 omogoča, da so slike rekonstruirane z različno kvaliteto (bpp) ali ločljivostjo (število pikslov v sliki).

- **Neobčutljivost na napake.** V sintakso bitnega niza JPEG2000 je dodan bit ECC, ki zagotavlja detekcijo in okrevanje od napak, ki nastopijo med prenosom.

- **Odprta arhitektura.** Ta princip zagotavlja, da je dekodiranje enostavno. Implementirati je potrebno samo orodja osnovega jedra in parser. Če je potrebno, dekodirnik dodatna orodja pridobi iz izvora.

- **Opis na podlagi vsebine** (angl. contex-based description). Shranjevanje, indeksiranje in iskanje slik je zelo pomembna naloga procesira-

nja slik. To nalogo sicer ima standard MPEG-7, vendar pa je opis slike na podlagi vsebine v obliki metapodatkov mozen tudi v JPEG2000.

- **Kanal prozornosti.** JPEG2000 vključuje tudi možnost kodiranja dodatnih kanalov, kot so na primer alfa kanali.

- **Zaščita slik.** Zaščita slik z vodnimi žigi, enkripcijo in označevanjem je v nekaterih apliakcijah izjemnega pomena. Označevanje (angl. labeling) je že implementirano kot del specifikacije SPIFF,

- **Zmožnost kodiranja v realnem času.** Nekatere aplikacije, kot na primer prebiranje (angl. scanning) delujejo sekvenčno vrstico po vrstici. Z uporabo valjčne transformacije, ki temelji samo na vrsticah, lahko sistem uporabi pri stiskanju in razširjanju le manjše število vrstic.

Poglavje 5

Ostale metode za stiskanje rastrskih slik

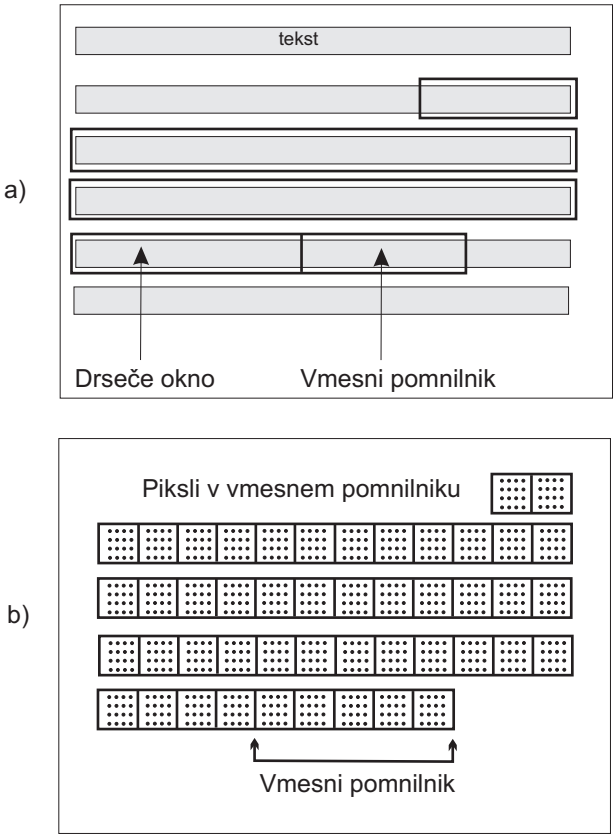
5.1 Ujemanje blokov

LZ77 je prvi predstavnik metod za stiskanje podatkov s slovarjem (slika 5.1a). Ideja je uporabna tudi pri stiskanju slik. Slika 5.1b kaže drseče okno in vmesni pomnilnik kot zaporedje dela pikslov.

Med piksli pričakujemo korelacijo, to je, da ima sosednji piksel nekega piksla P enako ali zelo podobno vrednost kot sam piksel P. Izkušnje kažejo, da lahko pričakujemo tudi iste nize pikslov na različnih mestih slike. Ker se podobni piksli ne nahajajo samo levo in desno od opazovanega piksla P, ampak tudi zgoraj in spodaj, je smiselno drseče okno in vmesni pomnilnik razširiti (slika 5.1b). Piksle v drsečem oknu zato združujemo v bloke 4×4 (od tod tudi ime metode), posledično mora biti ločljivost deljiva s 4. Organizacija drsečega okna je lahko tudi drugačna, na primer diagonalna ali spiralna.

Kodirnik postopa tako, kot klasičen LZ77. Poišče vse bloke v drsečem oknu, ki se ujemajo z bloki v pomnilniškem vmesniku. Kodirnik generira ustrezen žeton, nato pa se drseče okno premakne. Pri LZ77 žeton sestoji iz odmika (angl. offset), razdalje ujemanja (angl. match lenght) in naslednjega simbola v pomnilniškem vmesniku. Ta znak zagotavlja, da se kodirnik pomika tudi, ko ne najde ujemanja. V primeru slik je naslednji simbol seveda blok pikslov v vmesnem pomnilniku. Blok pikslov zavzema veliko prostora v izhodnem nizu, zato se mu izognemo na enega izmed naslednjih načinov:

1. Če ne najdemo ujemanja, pošljemo žeton oblike (0,0, blok_pikslov).



Slika 5.1: Stiskanje slik po pristopu LZ77.

2. Izhodni niz sestoji iz kode s spremenljivo dolžino (angl. variable-lenght code, VLC), ki ji sledi ali žeton z dvema simboloma ali pa blok iz 16-tih pikslov. Zastavica lahko ima naslednje vrednosti:

   - "0": če so vsi piksli v pomnilniškem vmesniku beli,
   - "10": če so vsi piksli v pomnilniškem vmesniku črni,
   - "110": če smo našli ujemanje (v tem primeru zastavici sledi žeton),
   - "111": če ujemanja nismo našli, blok pa tudi ni niti povsem bel niti črn; v tem primeru sledi blok pikslov.

   Iz opisa je očitno, da je metoda namenjena predvsem črnobelim slikam.

Iz izkušenj vemo, da ima tipična črnobela slika več belih področij kot črnih področij.

3. Rešitev je podobna rešitvi pod točko 2. Ko kodirnik najde povsem bel ali črn blok, poišče sekvenco teh blokov n. V tem primeru za kodo VLC '0' ali '10' sledi število n. Dolžino n tudi zakodiramo s kodo VLC (na primer z Golombovo kodo).

Odmik lahko predstavimo z razdaljo bloka od začetka vmesnega pomnilnika. V primeru, ko je drseče okno organizirano v bolj kompliciranо (diagonalno ali spiralno), pa ima lahko dekodirnik kar nekaj dela, da najde pravi položaj bloka. Zato lahko kot alternativo uporabimo odmik v koordinatah. Na žalost pa to zmanjša učinkovitost kodirnika. Imejmo sliko v ločljivost $1K \times 1K$ pikslov. Število vrstic in stolpcev je predstavljeno vsak z 10-timi biti. Za kodiranje koordinate zato potrebujemo 20 bitov.

5.2 Kodiranje rezanja blokov

Kot smo že spoznali, je kvantizacija izjemno pomembna tehnika pri stiskanju podatkov in še posebej pri stiskanju slik. Vsaka metoda mora temeljiti na postopku, ki določa, katere podatke bomo kvantizirali in za kakšno vrednost. Princip kodiranja rezanja blokov (angl. block truncation coding - BTC) kvantizira piksle v sliki, pri čemer pa ohrani *statistične momente slike*. Osnovna verzija metode BTC razdeli sliko v bloke (praviloma velikosti 4×4 ali 8×8. Predpostavimo, da ima blok n pikslov z vrednostmi p_1 do p_n. Prva dva momenta bloka sta njegova srednja vrednost in srednja vrednost vsote kvadratov, definirani kot:

$$\overline{p} = \frac{1}{n} \sum_{i=1}^{n} p_i \tag{5.1}$$

in

$$\overline{p^2} = \frac{1}{n} \sum_{i=1}^{n} p_i^2. \tag{5.2}$$

Standardna deviacija bloka je določena kot

$$\sigma = \sqrt{\overline{p^2} - \overline{p}^2}. \tag{5.3}$$

Metoda BTC izbere tri vrednosti: prag (p_{thr}), visoko vrednost p^+ in nizko vrednost p^-. Nato zamenjamo vsak piksel v bloku z eno izmed vrednosti p^+ ali p^- tako, da bosta prva dva momenta novih pikslov (njihova srednja vrednost in srednja vrednost kvadratov) enaka momentoma originalnega bloka. Pravilo za kvantizacijo je, da pikslu p_i priredimo vrednost p^+, če je večji od praga, če pa je manjši, pa vrednost p^-. Torej:

$$p = \begin{cases} p^+ & : \quad p_i \geq p_{thr} \\ p^- & : \quad p_i < p_{thr} \end{cases} \tag{5.4}$$

Intuitivno je za prag smiselno uporabiti $\overline{p}$. Visoko in nizko vrednost določimo tako, da ohranimo prva momenta. Z n^+ označimo število pikslov, katerih vrednost je večja ali enaka pragu, z n^- pa piksle, katerih vrednost je manjša. Vsota $n^+ + n^-$ je seveda enaka skupnemu številu pikslov n v bloku. Za ohranitev prvih dveh momentov veljata naslednji enačbi:

$$\begin{aligned} n\overline{p} &= n^- p^- + n^+ p^+, \\ n\overline{p^2} &= n^-(p^-)^2 + n^+(p^+)^2. \end{aligned} \tag{5.5}$$

Rešitvi enačb sta:

$$\begin{aligned} p^- &= \overline{p} - \sigma\sqrt{\frac{n^+}{n^-}}, \\ p^+ &= \overline{p} + \sigma\sqrt{\frac{n^-}{n^+}}. \end{aligned} \tag{5.6}$$

Rešitev sta realni števili, ki ju zaokrožimo na najbližjo celo število. Vidimo, da sta rešitvi postavljeni na obeh straneh povprečja $\overline{p}$ na razdalji, ki je proporcionalna standardni deviaciji bloka pikslov σ.
Oglejmo si primer bloka 4×4 pikslov z globino 8 bitov.

$$\begin{bmatrix} 121 & 114 & 56 & 47 \\ 37 & 200 & 247 & 255 \\ 16 & 0 & 12 & 169 \\ 43 & 5 & 7 & 251 \end{bmatrix}$$

Po izračunu dobimo $\overline{p} = 98.75$, nato preštejemo $n^+ = 7$ in $n^- = 9$. Srednja vrednost vsote kvadratov je 18.391,88, standardna deviacija $\sigma = 92.95$, zato sta visoka in nizka vrednost

$$p^+ = 98.75 + 92.95\sqrt{\frac{9}{7}} = 204.14, p^- = 98.75 - 92.95\sqrt{\frac{7}{9}} = 16.78,$$

ki ju zaokrožimo na 204 in 17. Dobljen blok pikslov je

$$\begin{bmatrix} 204 & 204 & 17 & 17 \\ 17 & 204 & 204 & 204 \\ 17 & 17 & 17 & 204 \\ 17 & 17 & 17 & 204. \end{bmatrix}$$

Takšen blok lahko predstavimo s $2 \times 8 = 16$ biti

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

in dvema 8-bitnima vrednostima 204 in 17. Skupaj torej potrebujemo 4×8 bitov napram 16×8 bitov, kar nam da stalno razmerje stiskanja 4.

Faktor stiskanja lahko vedno zračunamo vnaprej, saj je odvisen od velikosti bloka n in bitne globine b za predstavitev piksla. Velja, da je faktor stiskanja

$$cr = \frac{bn}{n + 2b}.$$

Predstavljena verzija metode BTC je enostavna in hitra, Njena glavna slabost pa je, kako zgublja informacije. Izgube so odvisne od intenzitete pikslov in ne od karakteristike človeškega vidnega sistema. Rezultat je slika, ki je po razširjanju kockasta. Najbolj moteče je, da morebitne črte po rekonstrukciji lahko postanejo nazobčane. Zato je metoda doživela veliko izboljšav.

5.3 Kontekstne metode

Kontekstne metode (angl. context-based method) za stiskanje slik posplošujejo koncept korelacije med sosednjimi piksli. Temeljijo na ideji, da je kontekst/vsebino piksla možno z določeno verjetnostjo napovedati glede na vsebino okoliških pikslov.

Kontekstne metode stiskajo sliko s prebiranjem piksla za pikslom. Vsebino vsakega piksla preverimo in mu priredimo verjetnost glede na to, kolikokrat smo trenuten kontekst že videli do sedaj. Piksel in njemu prirejeno verjetnost nato pošljemo v aritmetični kodirnik, ki nato dejansko opravi kodiranje. Kontekst opazovanega piksla je odvisen od njegovih sosednjih pikslov, ki so že bili obiskani. Primer konteksta opazovanega piksla "X" vidimo na sliki 5.2, pri tem so s "P" označeni že obiskani piksli, z "?" pa še neobiskani.

•	•	P	P	P	P	P	•	•
•	•	P	P	X	?	?	?	?

Slika 5.2: Primer konteksta: P – že videni piksli, X – opazovani piksel, ? – prihajajoči piksli.

Osnovna ideja je v tem, da tvorimo 7-bitni kazalec v tabelo frekvenc. Tabela frekvenc hrani podatek, kolikorat se je v danem kontekstu pojavila za vrednost piksla P ali logična 0 ali logična 1. Poglejmo si naslednji primer na sliki 5.3:

•	•	1	0	0	1	1	•	•
•	•	0	1	X	?	?	?	?

Slika 5.3: Kontekst za piksel X je $1001101_{(2)} = 77_{(10)}$.

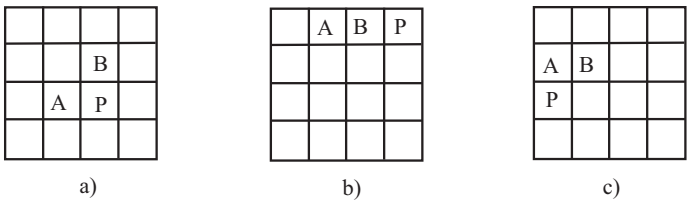
Pogledamo torej verjetnost pojave belega oz. črnega piksla pri kontekstu 77. Predpostavimo, da je števec ničel postavljen na 15, števec enic pa na 11. Trenutnemu pikslu je torej prirejena vrednost $15/(15+11) \approx 0.54$, če je piksel 0 in $11/26 \approx 0.42$, če je piksel enak 1. Glede na to, ali je piksel X 0 ali 1, se inkrementira ustrezen števec v tabeli.

Za kontekst lahko izberemo različno število pikslov v različni okolici piksla X (vedno izmed pikslov, ki so že bili obiskani). Preveliki konteksti niso primerni, ker imamo preveliko tabelo frekvenc, zelo veliko kontekstov pa se zelo redko pojavi ali pa sploh ne. Testi so pokazali, da so s stališča stiskanja najprimernejši konteksti med 12 in 14 biti.

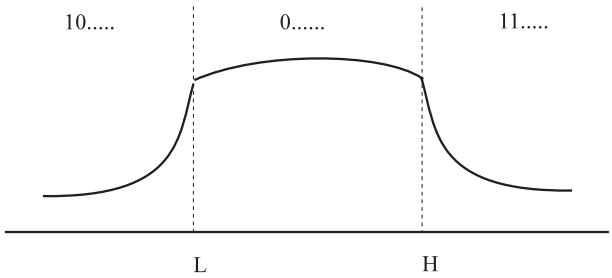
5.3.1 FELICS

FELICS (angl. Fast, Efficient, Lossless Image Compression System) je namenjen stiskanju sivinskih slik in je po učinkovistosti primerljiv z breziz-

gubnim JPEG. Uporabljen je bil za kodiranje slik za prenos iz Marsa v okviru projekta HiRISE (angl. High Resolution Imaging Science Experiment). Osnovni princip je, da vsak piksel kodiramo s kodo VLC, ki jo tvorimo glede na vrednosti dveh, predhodno že videnih pikslov. Na sliki 5.4 vidimo, kako izbiramo sosede pikslu "P"glede na njegov položaj. Prva dva piksla seveda nimata nobenega soseda, zato ju zapisemo neposredno.



Slika 5.4: Piksli konteksta pri metodi FELICS



Slika 5.5: Področja pri postopku FELICS glede na intenziteto vrednosti sosedov A in B

Postopek stiskanja dela na naslednji način. Opazujmo piksle P in njegova soseda A in B. Z L in H določimo vrednosti sosedov (njuni intenziteti) glede na intenziteto piksla P. Koda za P je določena glede na njegov položaj relativno na L in H. Ločimo seveda tri primere:

1. Intenziteta P je med L in H (nahaja se v srednjem področju na sliki 5.5). Pikslu P v tem primeru določimo kodo, ki se začne z 0. Verjetnost, da se bo P nahajal v osrednji regiji je skoraj enaka, zato je P kodiramo s kodo, katere varianca je majhna. Krajše kode naj imajo vrednosti, ki so najbliže središču osrednje regije.

2. Če je intenziteta manjša kot L, se P nahaja v levem področju, v tem

primeru se začne koda za opis intenzitete P z 10.

3. Ko je intenziteta večja kot H, se koda začne z bitoma 11.

Verjetnost, da se bo intenziteta piksla P zelo razlikovala od vrednosti L in H je majhna, zato v takšnih primerih lahko piksle P zakodiramo z daljšo kodo. FELICS uporablja Golomb-Riceovo kodo.

Poglejmo, kako določimo kode v osrednji regiji. Pričnemo z naslednjo enačbo:

$$k = \lfloor \log_2 (H - L + 1) \rfloor, \tag{5.7}$$

s pomočjo katere izračunamo števili a in b:

$$a = 2^{k+1} - (H - L + 1), \quad b = 2(H - L + 1 - 2^k). \tag{5.8}$$

a določa število kratkih kod, določenih kot $2^k - 1, 2^k - 2, \cdots$, b pa število daljših kod določenih kot $(k + 1)$ bitna števila. Poglejmo si primer. Naj bo $H = 25$, $L = 16$ in $H - L = 9$, potem je $k = 3$, število krajših kod je $a = 2^4 - (9 + 1) = 6$, število daljših kod pa $b = 2(9 + 1 - 2^3) = 4$. Sedaj določimo kode za a: $8 - 1 = 111$, $8 - 2 = 110$, do $8 - 6 = 010$. Kode za b pa so 0000, 0001, 0010, 0011. Krajše kode so razporejene na sredini področja, daljše pa na obeh krajiščih. b je sod, zato lahko množico števil vedno razdelimo v dve enako veliki podmnožici. Primer, kako so razporejena števila, vidimo v tabeli 5.1. Opozorimo, da se vsaki kodi doda MSB=0, ki označuje področje.

5.4 Dekompozicja blokov

Metoda je primerna za umetno generirane slike, tako črnobele kot barvne, kjer je število barv omejeno. Sosednji piksli, če niso enaki, se močno razlikujejo. Primer so na primer slikce iz risank. Metodo za takšne slike sta predlagala Gilbert and Brodersen [18] in jo poimenovala *Flexible Automatic Block Decomposition, FABC*. Med najbolj naravnimi metodami pri stiskanju slik je iskanje identičnih delov slike. Tako deluje tudi GIF, ki pa se omeji samo na vrstice v sliki. FABC pa predpostavlja, da enaki bloki lahko v sliki nastopajo večkrat. Ti bloki naj bodo dovolj veliki. Metoda tako prebira sliko vrstico po vrstici in jo razdeli v množico blokov, ki se lahko tudi prekrivajo. Ločimo med tremi tipi blokov: bloki kopije (angl. copied bloks), enobarvni bloki (angl. solid fill blocks) in sirote (angl. punts).

Tabela 5.1: Razporeditev kod FELICS v srednjem področju

vrednost piksla	koda piksla
16	0011
17	0010
18	010
19	011
20	100
21	101
22	110
23	111
24	0001
25	0000

Blok kopija je pravokotno področje slike, ki smo ga že srečali (nahaja se lahko nad pikslom ali levo od trenutnega piksla). Je lahko poljubne velikosti. Enobarvni blok je pravokotnik zapolnjen samo z eno barvo. Sirota je katerokoli področje v sliki, ki ni niti blok kopija niti enobarven blok. Dobljen niz nato stisnemo s Huffmanovim kodiranjem.

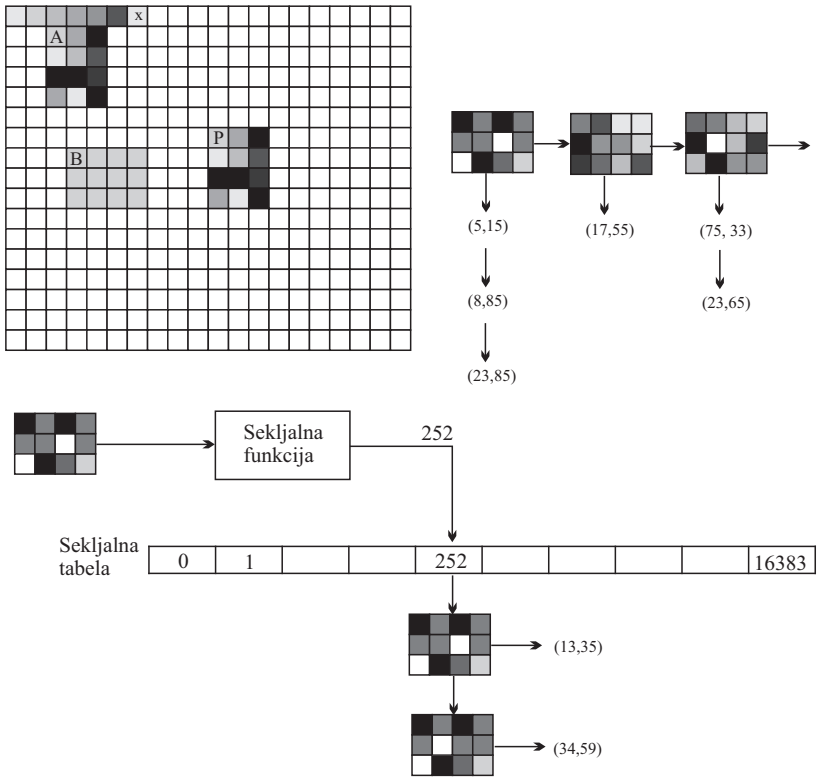
Kodirnik se pomika od piksla do piksla. Pri tem preglejuje bližino piksla P, da ugotovi, ali imajo bodoči piksli (tisti desno in spodaj) enako barvo kot P. Če to drži, metoda poišče največji takšen pravokotnik, pri katerem je P v zgornjem levem kotu. V izhodni niz zapišemo naslednjo četvorko:

koda_fill_block, višina, širino, barva,

kar zakodiramo na koncu s Huffmanovim kodirnikom. Na sliki 5.6a vidimo enobarvni blok velikosti 4×3, ki se začne pri pikslu B.

Če imajo bližnji piksli piksla P različne vrednosti, prične kodirnik iskati med identičnimi že videnimi piksli. P obravnava kot levi zgornji piksel bloka z nedoločenimi dimenzijami. Zatem prične preiskovati piksle, videne v preteklosti, da bi poiskal največje ujemanje med že videnimi bloki. Če je takšen blok A najden, je blok pri pikslu P označen kot blok kopija. Ker je blok A že stisnjen, je blok kopija popolnoma opisan z njegovimi dimenzijami (širina in višina) in pa z blokom A, ki se prične na podanih koordinatah. Blok torej opišemo kot petorček

koda_copy_block, višina, širino, A_x, A_y,



Slika 5.6: Dekompozicija blokov

Na izhod ni potrebno zapisati koordinate piksla P, zato ker tako kodirnik kot dekodirnik na enak način opravljata prebirni proces.

Če nismo našli identičnega bloka (najmanjši blok za ujemanje mora imeti neko minimalno velikost (na primer 4×4), kodirnik označi piksel P in se premakne na naslednji piksel. Takšen piksel postane sirota. Predpostavimo, da je 5 zaporednih pikslov označenih kot sirote. Kodirnik v tem primeru te piksle zbere in zapiše na izhod na naslednji način:

$$\text{koda\_punch\_block, } 5,\ P_1,\ P_2,P_3,P_4,P_5.$$

Na sliki 5.6 vidimo, da je prvih 6 pikslov različnih in zato označenih kot sirote. Zato tvorijo blok sirot. Sedmi piksel je enak kot prvi piksel, zato ima možnost da prične ali enobarvni blok ali blok kopijo. Ko blok zaznamo, spremenimo tudi postopek preiskovanja rastrske slike (glej sliko 5.6) in piksle, ki tvorijo blok, preskočimo.

Kodirnik in dekodirnik sta zelo asimetrična tako na čas izvajanja kot na naloge, ki jih morata izvesti. Dekodirnik predvsem kopira piksle, kodirnik pa mora najti enobarvne bloke in bloke kopije. Ta naloga je zelo zahtevna (časovna zahtevnost $O(n^2)$). Pri 10^6 pikslih imamo tako 10^{12} iskanj, kar vodi v nekajurno stiskanje. Prav zaradi tega je kodirnik potrebno implementirati zelo pazljivo. Pomagamo si lahko z dvema dejstvoma:

- Iskanje vključuje samo že videne piksle. Tako je potrebno preiskati celotno sliko samo za zadnje piksle. To zmanjša število iskanj na $n^2/2$. Dejansko je iskanj še manj, saj iskanje v implementaciji, ki jo predlagata avtorja, prične samo na lokacijah nekodiranih pikslov.

- Iskanje prav tako preskoči piksle iz enobarvnih blokov.

Iskanje po preostali množici pikslov pospešimo z naslednjo tehniko. Minimalna velikost bloka naj bo omejena na 4×4 piksle. Zato skonstruiramo seznam vseh vzorcev blokov 4×4, ki smo jih srečali do sedaj, in sicer po principu LIFO (zadnji najden vzorec se nahaja na začetku seznama). V vsakem trenutnem pikslu P kodirnik sestavi blok B 4×4 in preišče seznam. Ta seznam je lahko zelo dolg. Za črnobele slike imamo $2^{16} = 65536$ vzorcev. Za sliko z globino 8 bpp dobimo grozljivih $2^{8 \cdot 16} = 2^{128} \approx 3.4\ 10^{38}$. Na srečo v sliki ne nastopajo vsi bloki, pričakovano število pa je še vedno veliko. Predlagane so tri izboljšave:

1. Seznam vzorcev vsebuje vsak vzorec le enkrat. Poleg vzorca shranimo položaje na sliki, kjer se le-ti nahajajo (glej sliko 5.6b).

2. Uporabimo lahko sekljalno tabelo, kjer izračunamo kazalec v tabelo s pomočjo bitov v vzorcu. Tabelo lahko zgradimo tudi hierarhično, Na prvem nivoju lahko uporabimo kazalec velikosti enega zloga, kar nam da 128 vstopov v sekljalno tabelo. Zatem preiščemo seznam vzorcev, dostopnih preko indeksa sekljalne tabele (slika 5.6c). Ta seznam je v primeru velikih slik lahko zelo dolg. Zato v praksi omejimo iskanje na nekaj 100 vzorcev.

Algoritem razširjanja je mnogo preprostejši, prav tako pa je mnogo hitrejši. Njegova naloga je sestaviti ustrezen blok, katerega levi zgornji piksel je piksel P. Eksperimenti so pokazali, da dosegamo razmerje stiskanja 0.04 bpp za črnobele slike in 0.65 bpp za sivinske slike.

Poglavje 6

Stiskanje slik s podatkovnoodvisno triangulacijo

Področje stiskanja slik je postreglo s kopico izvirnih algoritmov. Vsekakor je eden izmed najbolj nenavadnih metoda, ki temelji na Delaunayevi triangulaciji. Obstaja kopica variant in izboljšav, pri katerih sta najpogosteje sodelovala A. Iske in L. Demaret. Metoda je izgubna, po kvaliteti pa se lahko meri tako z JPEG kot JPEG2000. Njena glavna slabost pa je počasnost stiskanja, ki je posledica konstrukcije tako imenovane podatkovnoodvisne triangulacije (angl. data-dependent triangulation). Algoritem vključuje tri pomembne korake:

- konstrukcija podatkovnoodvisne triangulacije s tako imenovanim postopkom adaptivnega tanjšanja (angl. adaptive thinning),
- stiskanje karakterističnih pikslov - oglišč Delaunayeve triangulacije in
- rekonstrucija slike.

Za vse dele algoritma obstajajo različne možnosti in podvariante opisane v množici člankov. Mi si bomo ogledali za vsak korak le po eno izvedbo.

6.1 Konstrukcija podatkovnoodvisne triangulacije

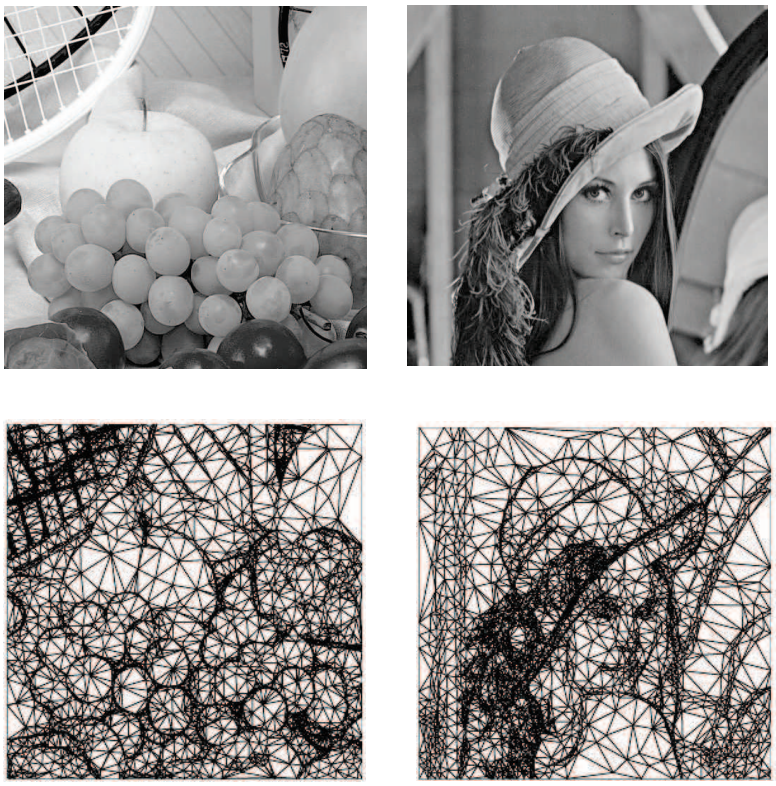
Piksle 2D slike $X = \{x_1, \ldots, x_n\}$ lahko predstavimo kot točke v $2,5D$, kjer barve predstavljajo višino. Predpostavimo, da je postopek obraten. Točke

iz X smo dobili z vzorčenjen neznane ploskve-funkcije f, in dobili vrednosti $f(x_1), \ldots, f(x_n)$. Poiskati želimo podmnožico Y, $Y \in X$ in triangulacijo T_Y tako, da bo odsekovno gladka linearna interpolacija $L(f, T_Y)$ takšna, da bo največja razlika med originalno in interpolirano funkcijo najmanjša, oziroma, da bo napaka

$$E(T_Y; X; f) = max|L(f, T_Y)(x) - f(x)| \tag{6.1}$$

majhna.

Tanjšanje je iterativni postopek odstranjevanja točk iz X dokler Y ne doseže predpisano velikost. Primere vhodnih slik in podatkovnoodvisne triangulacije vidimo na sliki 6.1.



Slika 6.1: Primer podatkovnoodvisne triangulacije; slika povzeta iz [19]

Postopek *adaptivnega tanjšanja*, ki si ga bomo ogledali, povzamamo po [20]. Adaptivno tanjšanje je pogosta naloga v trianguliranih površjih, kjer

postopku pravimo *decimacija* ali *poenostavljanje* trikotniške mreže. Ideja je preprosta. Če imamo trianguliran objekt, želimo poiskati tisto oglišče trikotniške mreže, katega odstranitev najmanj pokvari površje objekta.

Poenostavljanje trikotniških mrež lahko opravimo na več načinov glede na to, katere elemente odstranjujemo. Imamo tri možnosti:

- **Poenostavljanje vozlišč**. Poenostavljanje vozlišč je najpogostejše in temelji na Schroederjevem poenostavitvenem postopku [21]. Vozlišča najprej ovrednotimo in jih nato odstranimo iz mreže, glede na njihovo pomembnost. Pregled metod najdemo v [22].

- **Poenostavljanje robov.** V tem primeru odstranjujemo pred tem ovrednotene robove [23]. Ena najboljših metod s poenostavljanjem robov temelji na najmanjši vsoti kvadratov napak.

- **Poenostavljanje trikotnikov.** V tem primeru naj bi algoritem odstranjeval trikotnike, vendar praktične izvedbe tega postopka niso znane.

Ključni problem algoritmov decimacije je njihova časovna zahtevnost. V vsakem koraku je namreč potrebno oceniti vsa preostala vozlišča, kar vodi v časovno zahtevnost $O(n^2)$. Zato raje žrtvujemo natančnost algoritma (mogoče ne izberemo ravno najugodnejše vozlišče) napram hitrosti. Predstavljen algoritem uporablja sekljalno tabelo in deluje v naslednjih korakih:

1. Ovrednotenje vseh vozlišč glede na izbran kriterij in njihova razvrstitev v sekljalno tabelo.

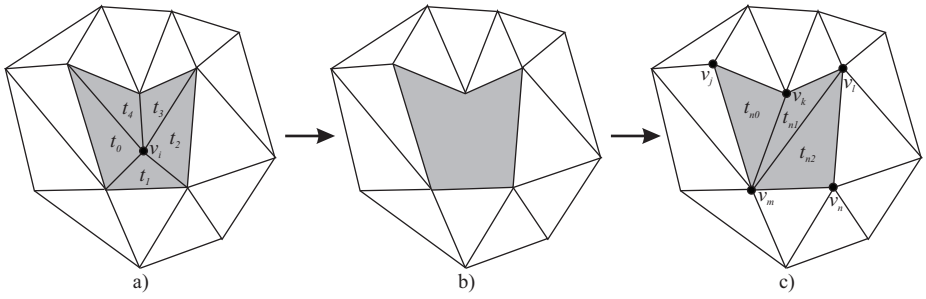
2. Izbira najprimernejšega vozlišča z uporabo sekljalne preglednice (na primer vozlišče v_i na sliki 6.2a).

3. Odstranitev vozlišča in z njim definiranih trikotnikov iz trikotniške mreže (slika6.2b).

4. Triangulacija nastalega praznega prostora (slika 6.2c) z omejeno Delaunayevo triangulacijo.

5. Ponovno ovrednotenje neposrednih sosedov odstranjenega vozlišča (vozlišča v_j, v_k, v_l, v_m, v_n na sliki 6.2c).

6. Vračanje na korak 2, dokler ni izpolnjen končni kriterij (na primer, da smo dosegli vnaprej določeno število oglišč, ali da je napaka, ki jo povzročimo večja od dovoljenega praga).

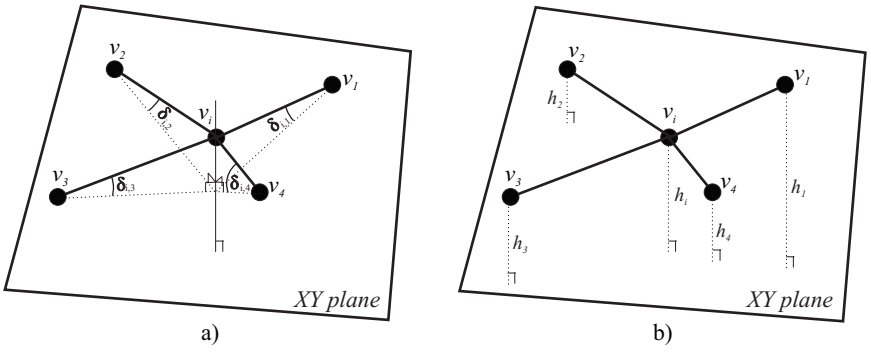


Slika 6.2: Postopek decimacije

Za ovrednotenje vozlišč lahko uporabimo različne strategije, kot na primer:

- Za vsako vozlišče v_i konstruiramo vektorje $v_{i,j}$, ki povezujejo v_i z vsemi njegovimi sosedi v trikotniški mreži (slika 6.3a). Zatem izračunamo kote $\delta_{i,j}$ med vektorji $v_{i,j}$ in ravnino XY. Povprečna vrednost vseh kotov $\delta_{i,j}$ je faktor vrednotenja g_i vozlišča v_i.

- Faktor vrednotenja g_i vozlišča v_i je povprečna vrednost razlik vrednosti višin (v našem primeru barv pikslov) h_i med vozliščem v_i in njegovimi sosedi (slika 6.3b).



Slika 6.3: Dve možnosti vrednotenja vozlišč

6.1.1 Izbira vozlišča za odstranitev

Odstranjeno vozlišče naj bi povzročilo najmanjšo napako v predstavitvi. Zaradi tega moramo iz trikotniške mreže odstraniti vozlišče z najmanjšim faktorjem ovrednotenja g_i. Za tem se faktor ovrednotenja spremeni vozliščem, ki so neposredni sosedi v_i, zato jih moramo ponovno ovrednotiti. V naslednjem koraku ponovno potrebujemo vozlišče z najmanjšim g_i. Iskanje najenostavneje opravimo s sprehodom skozi celotno množico vozlišč. Na žalost pa ta strategija vodi v časovno zahtevnosto $O(n^2)$. Nekoliko boljši pristop bi najprej uredil vsa vozlišča glede na njihov g_i, nato pa po odstranitvi vozlišča ažuriral položaje vozlišč, ki so se jim spremenili g_i. Pristop deluje v pričakovani časovni zahtevnosti $O(n \log n)$. Veliko bolj učinkovit algoritem pa dobimo, če omilimo zahtevo po odstranitvi vozlišča z najmanjšim g_i. Če lahko odstranimo katerokoli vozlišče z dovolj majhnim faktorjem ovrednotenja g_i, bo algoritem deloval v času $O(n)$, sam postopek izbire vozlišča, ki bo odstranjeno, pa v času $O(1)$.
Število vhodov v sekljalno tabelo dobimo z enačbo:

$$m = \left\lfloor \frac{n}{k} \right\rfloor ; k > 0 \tag{6.2}$$

kjer je n število točk in k uporabniško podan parameter. Vkolikor pričakujemo enakomerno porazdelitev ocene g_i, lahko trivialno določimo indeks sekljalne tabele i kot:

$$i = \left\lfloor m \frac{g_i - g_{min}}{g_{max} - g_{min}} \right\rfloor ; i = [0, m-1] \tag{6.3}$$

pri čemer sta g_{max} in g_{min} največja in najmanjša vrednost g_i.

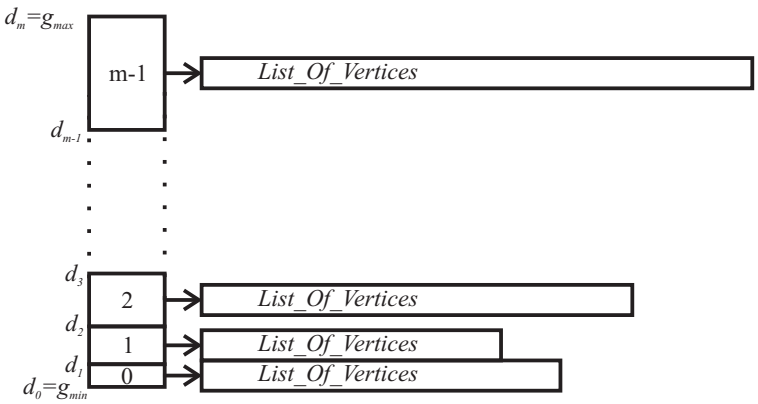
Seveda pa je pričakovana porazdelitev vrednosti napake g_{vi} lahko drugačna. Kot primer podajamo eksponencialno porazdelitev. Meje intervalov d_i izračunamo kot:

$$\begin{aligned} d_i &= g_{sum} - g_{ln}\, ln(1 + \Delta - i\ \ g_\Delta) \\ g_{sum} &= g_{min} + g_{max} \\ g_{ln} &= \frac{g_{max}}{ln(1+\Delta)} \\ \Delta &= g_{max} - g_{min} \\ g_\Delta &= \frac{m}{\Delta} \end{aligned} \tag{6.4}$$

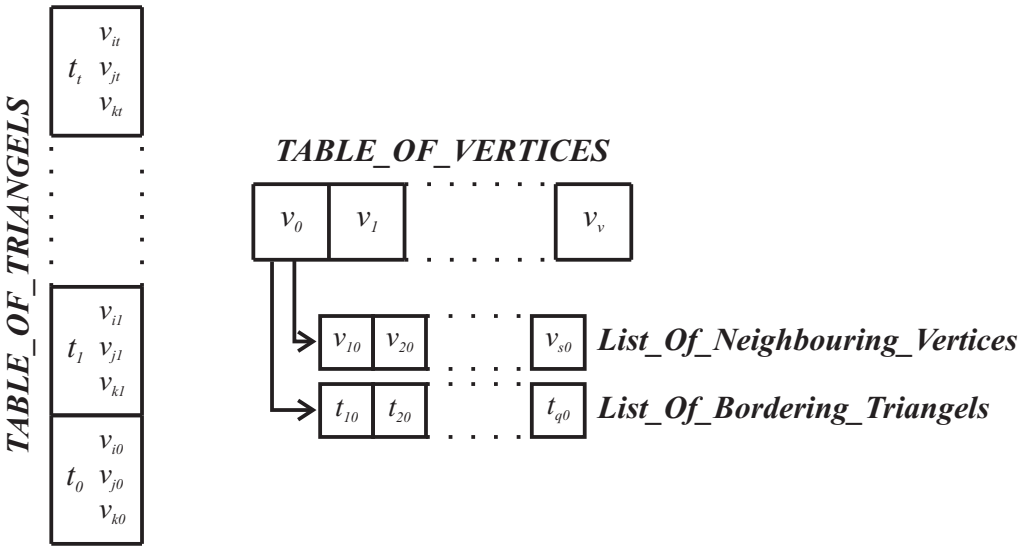
Indeks v tabeli potem izračunamo po enačbi:

$$i = \begin{cases} \lfloor g_\Delta(1 + \Delta - e^{((g_{sum} - g_i)g_{ln})}) \rfloor & : \quad g_i < g_{max} \\ m - 1 & : \quad g_i \geq g_{max} \end{cases} \tag{6.5}$$

Slika 6.4 prikazuje umestitev vozlišč v sekljalno tabelo. Algoritem izbere prvo vozlišče iz najnižjega nepraznega seznama sekljalne tabele. To je naše vozlišče v_i, ki ga zbrišemo. Nato ovrednotimo vse neposredne sosede vozlišča v_i in jih vstavimo na konec seznama v ustrezni vrsti sekljalne tabele. S tem preprečimo, da bi bilo poenostavljanje lokalno. Seveda pa moramo zagotoviti, da dobimo vse sosede vozlišča v_i brez dodatnega iskanja, kar lahko zagotovimo z organizacijo podatkov, ki jo vidimo na sliki 6.5. Vsako vozlišče ima l sosedov, zato opravimo vrednotenje v času $O(l)$. Ker je $l << n$ ocenimo, da se ta korak opravi v konstantnem času $O(1)$. Če vozlišč na ogliščih slike ne želimo odstraniti, jih ne vstavimo v sekljalno tabelo.



Slika 6.4: Vozlišča v sekljalni tabeli



Slika 6.5: Dostop do sosednjih sosedov vozlišča v_i

6.2 Stiskanje karakterističnih pikslov

Možnosti, s katerimi lahko stisnemo preostale piksle, je veliko. Na primer, položaje pikslov označimo kot binarno sliko, ki jo stisnemo s standardom JBIG (s tem zakodiramo koordinate), vrednosti koordinat (barve pikslov) pa stiskamo posebej z aritmetičnim ali Huffmanovim kodiranjem.

Druga možnost je, da dobljeno matriko obravnavamo kot redko matriko. Za njihov prostorsko optimalnejši zapis obstaja več metod. V nadaljevanju si poglejmo tri.

6.2.1 Metoda CSF

Najpreprostejša metoda je koodinatna metoda (angl. Coordinate Storage Format, CSF). Podatkovna struktura sestoji iz treh polj:

- polje AA hrani neničelne vrednosti matrike v poljubnem vrstnem redu,
- polje JR hrani indekse vrstic in
- polje JC hrani indekse stolpcev.

Poglejmo si primer. Matrika naj bo naslednja:

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

AA | 12 9 7 5 1 2 11 3 6 4 8 10 |

JR | 5 3 3 2 1 1 4 2 3 2 3 4 |

JC | 5 5 3 4 1 4 4 1 1 2 4 3 |

Vidimo, da potrebujemo 3 polja dolžine k, kjer je k število neničelnih elementov. Seveda bi lahko elemente v polju AA tudi uredili ali zapisali po vrsti, še posebej, če bomo polja AA, JR in JC stisnili z aritmetičnim ali interpolativnim kodiranjem.

6.2.2 Metoda CSR

Metoda CRS (angl. Compressed Sparse Row) je zelo popularna metoda za kompatno opisovanje redkih matrik. Tudi ta metoda uporablja 3 polja:

1. polje AA hrani neničelne vrednosti matrike, ki pa se vrstijo zaporedoma vrstico po vrstici,

2. polje JR hrani indekse stolpcev in

3. polje JC pa hrani kazalce, ki ločijo polje elementov.

Oglejmo si primer.

AA | 1 2 3 4 5 6 7 8 9 10 11 12 |

JR | 1 4 1 2 4 1 3 4 5 3 4 5 |

JC | 1 3 6 10 12 13 |

Razložimo. V prvi vrstici obstajajo elementi, zato tja kaže prvi kazalec. Prvi element v drugi vrstici se začne na položaju 3 v polju AA. Prvi element v tretji vrstici je na položaju 6 v polju AA. Število 10 je prvi neničelni element četrte vrstice in je na položaju 10 v polju AA. V peti vrstici se nahaja zadnji element, ki je na položaju 12 v polju AA. Zadnji element v polju JC (13) je stražar in pove število neničelnih elementov. Stražar določa tudi dolžino polj AA in JR, dolžina polja JC pa je $n + 1$, kjer je n red matrike.

6.2.3 Metoda MSR

Metoda MSR (angl. Modified Storage Format) ima samo dve polji enake dolžin (N_z+1), kjer N_z pomeni število neničelnih elementov. Polje AA hrani podatke. Prvih n položajev hrani podatke v diagonali, položaj $n + 1$ pa ni uporabljen. Ostale elemente matrike vpišemo v polje AA od položaja $n + 2$ dalje po vrsticah. Za vsak element $AA(k)$ element v polju $JA(k)$ predstavlja indeks stolpca. Prvih $(n + 1)$ elementov polja JA hrani kazalec na začetek vsake vrstice v AA, preostali pa določajo položaj kolone, v kateri se element nahaja. Primer zapisa MSR za našo matriko:

AA | 1 4 7 11 12 * 2 3 5 6 8 9 10 |

JA | 7 8 10 13 14 14 4 1 4 1 4 5 3 |

Konstrukcija polja AA je enostavna. Z zvezdico smo označili položaj neuporabljenega elementa. Konstrukcija polja JA je bolj zanimiva. Prvi element 7 polja JA kaže na položaj prvega nediagonalnega elementa iz prve vrstice matrike v polju AA. Nahaja se na položaju 7. Element 3 je prvi nediagonalni element druge vrstice v matriki; njegov položaj v polju AA pa je 8. Element 6 je prvi nediagonalni neničelni element tretje vrstice; v polju AA se nahaja na položaju 10. Prvi neničelni nediagonalni element četrte vrstice (10) je na lokaciji 13. V zadnji vrstici ni neničelnega nediagonalnega elementa, zato kaže kazalec izven polja AA. Element na položaju $(n + 1)$ ima vlogo stražarja. Preostali elementi polja JR določajo stolpce neničelnih in nediagonalnih elementov.

6.2.4 Metoda CSV

Metoda CSV (angl. compressed sparse vector) uporablja samo dve polji dolžine $N_z + 1$, kjer je N_z število neničelnih elementov. Metoda je preprosta

in daje glede na [24] zelo dobre rezultate. V polje AA po vrstical po vrsti vpišemo neničelne elemente, v polje IA pa vpišemo razdaljo do elementa v matriki, pri čemer razdaljo do elementa (0,0) označimo z 1. Primer:

AA | 1 2 3 4 5 6 7 8 9 10 11 12 |

IA | 1 4 6 7 9 11 13 14 15 18 19 25 |

Pri zelo velikih matrikah lahko postanejo indeksi v IA veliki, zato lahko shranimo relativno razdaljo med zaporednimi neničelnimi elementi. Pri tem velja:

- če je prvi element matrike 0, postavimo indeks relativne razdalje na 1 in preštejemo število ničel do prvega neničelnega elementa;
- če je prvi element matrike različen od 0, postavimo indeks relativne razdalje na 1, ga shranimo v polje AI, nato pa nadaljujemo, kot opisano.

V našem primeru bi dobili:

AA | 1 2 3 4 5 6 7 8 9 10 11 12 |

IA | 1 3 2 1 2 2 2 1 1 3 1 6 |

6.2.5 Stiskanje zaporedja celih števil

Dobljena zaporedja celih števil nato dodatno stisnemo. Zato imamo veliko možnosti. Najmanj ugodna je, da poiščemo največje število, ugotovimo, koliko bitov potrebujemo za njegov zapis, nato pa vsa ostala števila zapišemo s tem številom bitov. Problem stiskanja zaporedja celih števil pa je zelo pogost, zato najdemo kar nekaj metod, ki uporabljajo kodiranje s spremenljivo dolžino.

6.2.5.1 Prilagodljivo binarno zaporedno kodiranje

Metodo prilagodljivega binarnega zaporednega kodiranja (angl Binary Adaptive Sequential Coding, BASC) sta razvila Moffat in Anh [25] in je namenjena za zapis nenegativnih celoštevilskih vrednosti. Metoda omogoča sprotno prilagajanje potrebnega števila bitov b glede na predhodnji simbol.

Začetno vrednost b vstavi uporabnik. Za trenutno celo število x_i izračunamo število bitov b', potrebnih za predstavitev x_i. Obravnavamo dve možnosti:

- $b' \leq b$: kodirnik v tem primeru zapiše 0, ki ji sledi binarna predstavitev števila x_i, zapisana kot $b-$bitno število.
- $b' > b$: kodirnik zapiše $(b' - b)$ enic, ki jim sledi ničla, tej pa $b' - 1$ manj pomembnih bitov števila x_i (najpomembnejši bit x_i mora biti 1, zato ga lahko izpustimo).

Po vsakem koraku postavimo $b' = b$. Oglejmo si primer. Imejmo zaporedje celih nenegativnih števil (15,6,2,3,0,0,0,0,4,5,1,7,8) in naj bo $b = 5$. Prvo število 15 lahko zapišemo s 4-rimi biti, zato je $b' = 4$, kar je manj kot b. Kodirnik zapiše 0, ki ji sledi binarni zapis števila 15 z $b = 5$ biti. Dobimo torej 0|01111, b pa dobi vrednost 4. Naslednje število je 6, $b' = 3$ in dobimo 0|0110 ter postavimo $b = 3$. Zatem kodiramo število 2 kot 0|010 in postavimo $b = 2$. Število 3 zatem zakodiramo kot 0|11, b pa ostane 2. Prvo izmed štirih ničel zakodiramo kot 0|00 in postavimo $b = 0$. Preostale ničle zakodiramo samo s po enim bitom 0. Naslednje število je 4, zato je $b' = 3 > b$. Kodirnik zapiše $b' - b = 3$ enice, ki ji sledi 0, tej pa dva najmanj pomembna bita števila 4, to je 00. Dobimo torej 111|0|00, b pa postavimo na 3. Naslednje število je 5, ki ga lahko zapišemo z $b = 3$ biti. Kodirnik torej zapiše 0|101 ter tako nadaljuje do konca.

6.2.6 Rekonstrukcija slike

Pri rekonstrukciji slike iz preostalih pikslov najprej skonstruiramo Delaunayevo triangulacijo. Manjkajoče piksle lahko zgeneriramo na različne načine. V nadaljevanju si bomo pogledali dve metodi.

6.2.6.1 Linearna interpolaciijo z baricentričnimi koordinatami

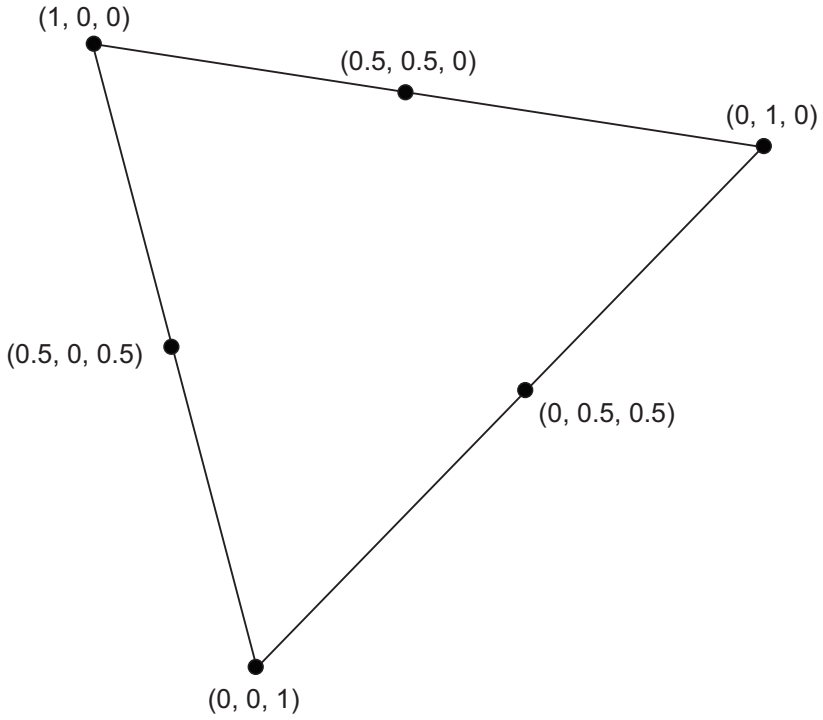
Ideja je podobna Phongovemu senčenju, ki se uporablja se za gladko senčenje 3D objektov predstavljenih s krpicami (običajno trikotniki ali štirikotniki). Kljub temu, da imamo 2D sliko, pa jo v našem primeru ves čas obravnavamo kot 2,5D površje, interpolirano s trikotniki. Oglišča vsakega trikotnika hranijo podatke o barvi (intenzivnosti), barve pikslov znotraj trikotnika pa interpoliramo z baricentričnimi koordinatami, kot kaže naslednja enačba:

$$barva = aA + bB + cC, \tag{6.6}$$

kjer so A, B, C intenzitete barve v ogliščih trikotnika, a, b, c pa baricentrične koordinate. Velja (glej sliko6.6):

$$a + b + c = 1; \tag{6.7}$$

$$a, b, c \in [0, 1] \tag{6.8}$$



Slika 6.6: Baricentrične koordinate

Linearna interpolacija pa se v nekaterih primerih ne izkaže najbolje, zato je smiselno poiskati boljšo metodo.

6.2.6.2 Zienkiewiczeva interpolacija

Zienkiewicz je razvil metodo za interpolacijo geografskega površja. Metoda temelji na določanju gradienta in uporablja kubično interpolacijo. V ogliščih najprej določimo gradiente. Postopek je naslednji. Najprej zračunamo normalo v ogliščih. Velikost normale utežimo glede na velikost trikotnika:

$$n = \sum_i \frac{A_i \ n_i}{A} \tag{6.9}$$

kjer je A_i ploščina i-tega trikotnika, n_i je njegova normala, A pa je vsota ploščin vseh trikotnikov, ki se stikajo v konkretnem vozlišču. Gradient v smeri x in y nato zračunamo kot

$$g = (\frac{-n_x}{n_z}, \frac{-n_y}{n_z}) \tag{6.10}$$

Spomnimo, da je komponenta z dejansko intenziteta barve. Intenziteto pikslo P nato zračunamo po naslednje enačbi:

$$\begin{aligned} P(a,b,c) = u_1\lfloor a^2(3-2a)+k_1\rfloor + u_2(a^2b+k_2) - u_3(a^2b+k_2) + \\ +u_4(b^2(3-2b)+k_1) + u_5(b^2c+k_2) - u_6(bc^2+k_2) + \\ +u_7(c^2(3-2c)+k_1) + u_8(c^2a+k_2) - u_9(ca^2+k_2). \end{aligned} \tag{6.11}$$

$$\tag{6.12}$$

Pri tem so (a,b,c) baricentrične koordinate, A,B,C pa koordinate trikotnika. Parametri u_i in k_i so:

$$\begin{array}{rrr} u_1 = A_i & u_4 = B_i & u_7 = C_i \\ u_2 = \overline{AB}g_a & u_5 = \overline{BC}g_b & u_8 = \overline{CA}g_c \\ u_3 = \overline{AB}g_b & u_6 = \overline{BC}g_c & u_9 = \overline{CA}g_a \\ k_1 = 2abc & & k_2 = 0.5cbc \end{array}$$

$\overline{AB}, \overline{BC}$ in $\overline{CA}$ so robni vektorji.

Poglavje 7

Pobarvanke in verižne kode

Pobarvanke (tudi risane slike) sestojijo iz majhnega števila barv, ki tvorijo velika območja, te pa so omejena z ostrimi prehodi. Za stiskanje pobarvak ni veliko metod. Eno smo si že ogledali v podpoglavju 5.4. Metoda je v fazi stiskanja bila časovno potratna, prav tako je bila omejena le na pravokotna območja. Jeromel in Žalik sta leta 2019 [26] razvila mnogo učinkovitejšo metodo, temelječo na verižnih kodah. Le-te si bomo tudi najprej ogledali.

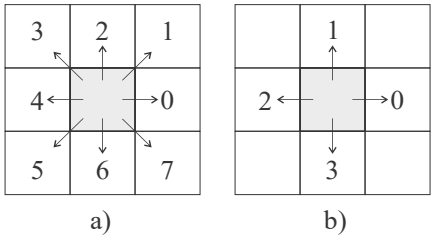
7.1 Verižne kode

Verižna koda je zaporedje elementarnih ukazov, s katerimi se premikamo po robu geometrijskega objekta/regije. Obstajajo različne verižne kode, ki si jih bomo, hkrati z njihovimi lastnostmi, ogledali v tem podpoglavju.

7.1.1 Freemanova verižna koda v osem smeri

Freemanova verižna koda v osem smeri (angl. Freeman chain code in eight directions – F8) je najstarejša verižna koda. Predlagal jo je Freeman davnega leta 1961 [27]. Iz referenčnega robnega piksla se lahko premaknemo v enega izmed osmih sosednjih pikslov, kot kaže slika 7.1a. Abeceda kode ima 8 simbolov: $\Sigma(F8) = \{0, 1, 2, 3, 4, 5, 6, 7\}$.

Postopek generiranja verižne kode F8 je sestavljen iz naslednjih korakov: izberemo začetni robni piksel in shranimo njegove koordinate (x, y). Odločimo se za smer potovanja po robnih pikslih (sourna ali protiurna), nato pa skonstruiramo kodo tako, da se premikamo med sosednjimi robnimi piksli in zapišemo Freemanovo kodo premika. Kodi, ki se premika skozi središča pikslov bomo rekli tudi središčna koda. Lastnosti verižne kode F8



Slika 7.1: Simboli verižne kode F8 (a) in F4 (b)

so naslednje:

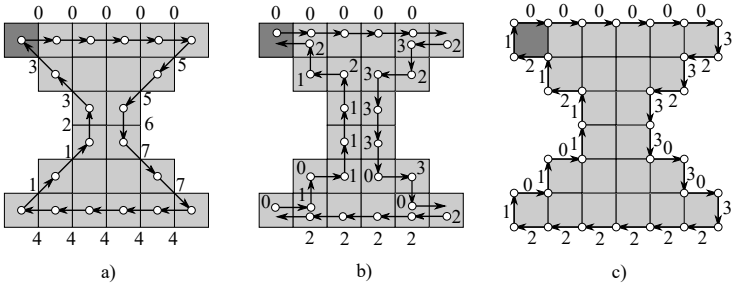
- položaj objekta je odvisen le od koordinat začetne točke verižne kode (ta lastnost je dejansko skupna vsem verižnim kodam),
- povečevanje objekta za faktor $2 \times s$ izvedemo tako, da vsak simbol F8 shranimo s-krat.
- zaporedje enakih vrednosti predstavlja zaporedje pikslov, ki si sledijo v ravni črti (vodoravni, navpični ali pod kotom 45^o),
- če vsakemu simbolu F8 v zaporedju prištejemo vrednost naravnega števila n po modulu 8, objekt zavrtimo za $n \times 45^o$ v smeri urinega kazalca,
- podobno, če vsaki vrednosti verižne kode odštejemo n po modulu 8, objekt zavrtimo za $n \times 45^o$ v obratni smeri urinega kazalca.

7.1.2 Freemanova verižna koda v štiri smeri

Freemanova verižna koda v štiri smeri (angl. Freeman chain code in four directions – F4), ki jo je tudi predlagal Freeman, dovoli premik iz trenutnega piksla v sosednje piksle samo preko skupnih robov (slika 7.1b). Abeceda kode F4 je zato manjša: $\Sigma(F4) = \{0, 1, 2, 3\}$. Konstrukcija kode F4 je enaka kot pri F8, tudi lastnosti so enake, le da lahko objekt zavrtimo le za kote $\pm 90^o$ in 180^o.

Kodo F4 lahko uporabimo na dva načina. V prvem načinu potujemo skozi središče pikslov, tako kot pri F8, v drugem primeru pa po mejnih robovih pikslov. Takšni kodi pravimo lomna verižna koda (angl. crack chain code, F4^C). Primer objekta, opisanega z različnimi Freemanovimi verižnimi kodami, vidimo na sliki 7.2.

Freemanovi verižni kodi pa dopuščata tudi različne modifikacije, ki si jih bomo ogledali v naslednjem podpoglavju.



Slika 7.2: Geometrijski objekt, opisan s F8 (a), F4 (b) in $F4^C$ (c)

7.1.3 Izpeljanke Freemanove verižne kode

1. Freeman je kasneje predlagal diferenčno vozliščno kodo (angl. Chain-Difference Coding – CDC) [28]. Vsak mejni piksel p_i (razen prvega p_0 in drugega p_1) zakodiramo z relativno razliko kotov $\measuredangle(p_i - p_{i-1}, p_{i-1} - p_{i-2})$, relativne razlike pa zapišemo s Freemanovo abecedo $\Sigma(F8)$. Izkaže se, da so statistično najbolj verjetne razlike kotov 0^o, ki jim sledijo koti $\pm 45^o$ (glej razpredelnico 7.1).

Tabela 7.1: Verjetnost spremembe smeri kotov med sosednjimi piksli

razlika kotov	0^o	$\pm 45^o$	$\pm 90^o$	$\pm 135^o$	$\pm 180^o$
verjetnost	0.453	0.488	0.044	0.012	0.003

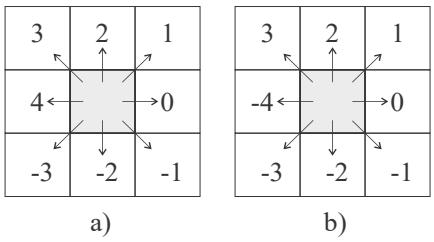
To lastnost sta izkoristila Liu in Žalik [29], ki sta predlagala eno prvih metod za stiskanje verižnih kod, temelječo na Huffmanovih kodah (glej razpredelnico 7.2).

2. Leta 1992 je Bribiesca predstavil izpeljanko verižne kode F8, ki jo imenujemo usmerjena 8-smerna Freemanova verižna koda (angl. Directional Freeman Chain Code of eight directions, DF8) [30]. Bribiesca je spremenil abecedo tako, da je vrednosti simbolov 5, 6 in 7 zamenjal z vrednostmi -3, -2 in -1. V primeru, da je orientacija verižne kode sourna, tudi simbol 4 nadomestimo z vrednostjo -4 (slika 7.3). S to spremembo enostavno ugotovimo, ali verižna koda predstavlja sklenjen objekt. Če je vsota vrednosti uporabljenih verižnih kod 8 pri protiurni orientaciji, oz. -8 pri sourni, je verižna koda sklenjena, sicer pa ni.

3. Varianto verižne kode F4 je predstavil Nunes s sodelavci [31]. Predla-

Tabela 7.2: Huffmanove kode spremembi smerem

sprememba smeri	verjetnost	Huffmanova koda
0^o	0,453	0
45^o	0,244	10
-45^o	0,244	110
90^o	0,022	1110
-90^o	0,022	11110
135^o	0,006	111110
-135^o	0,006	1111110
180^o	0,003	1111111



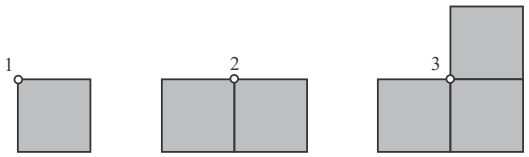
Slika 7.3: Kode DF8 pri protiurni (a) in sourni (b) orientaciji

gali so kodiranje relativne spremembe obhoda, kodo pa poimenovali verižna koda razlik (angl. Differential Chain Code, DCC). Abeceda DCC ima samo 3 simbole, $\Sigma(DCC) = \{R, L, S\}$, kjer R pomeni sasuk v desno, L, zasuk v levo in S nadaljuj v isti smeri. Koda DCC je lomna koda.

7.1.4 Ogliščna verižna koda

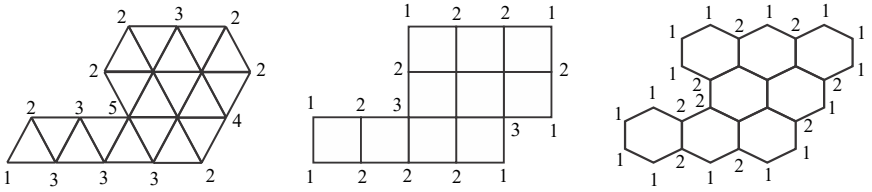
Leta 1999 je Bribiesca izumil zanimivo verižno kodo, ki jo imenujemo ogliščna verižna koda (angl. vertex chain code – VCC) [32]. Koda VCC je definirana s številom robnih pikslov objekta v opazovanem oglišču, kot vidimo na sliki 7.4. Abeceda vozliščne kode VCC sestoji samo iz treh simbolov, $\Sigma(VCC) = \{1, 2, 3\}$. VCC je lomna vozliščna koda in se vedno sklene. Zaporedje simbolov $\langle 1, 1, 1, 1\rangle$ opiše en piksel.

Koda VCC je uporabna tudi, ko celice niso štirikotniki. Primer vidimo na sliki 7.5. Ugotovimo, da se v tem primeru spremeni tudi abeceda. Seveda bomo v nadaljevanju privzeli, da so celice piksli pravokotne/kvadratne



Slika 7.4: Simboli vozliščne kode VCC

oblike.



Slika 7.5: Koda VCC pri celicah različnih oblik

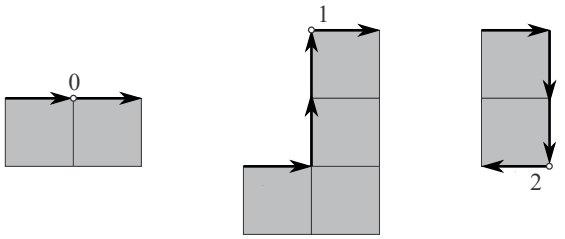
Verižna koda VCC ima naslednje lastnosti:

- ravno (navpično ali vodoravno) črto karakterizira zaporedje simbolov 2,
- diagonalno črto pod kotom $\pm 45^o$ označuje izmenjujoče se zaporedje simbolov 1 in 3,
- koda VCC je neodvisna glede na vrtenje (za $\pm 90^o$ in 180^o) in zrcaljenje,
- kodo VCC lahko normaliziramo; simbole sučemo tako dolgo, da zaporedje simbolov predstavlja število z najmanjšo vrednostjo, kot vidimo na naslednjem primeru:
  13112321213212131133312; zasuk v desno
  31123212132121311331121; zasuk v desno
  11232121321213113312l3; normalizirana koda
- kodo VCC lahko enostavno stisnemo; ker je, statistično, najpogostejši simbol VCC 2, ga predstavimo z enim bitom, ostala dva simbola pa z dvema.

7.1.5 Tri-ortogonalna verižna koda

Tri-ortogolano verižno kodo (angl. Three OrThogonal chain code – 3OT) sta predstavile Sáchez-Cruz and Rodríguez-Diaz [33]. Tudi ta koda ima abecedo, predstavljeno iz treh simbolov, $\Sigma(3OT) = \{0, 1, 2\}$, katerih pomen je naslednji (glej sliko 7.6):

- če je smer premika enaka, kot je bila smer premika predhodnega piksla, je koda 0;
- če je trenutna smer enaka smeri predhodnika, katerega koda je različna od 0, potem je koda 1;
- sicer je koda 2.



Slika 7.6: Symboli verižne kode 3OT

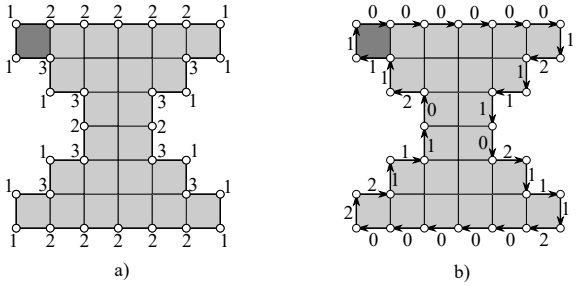
3OT je lomna koda, katere postopek konstrukcije je naslednji:

- izberemo ekstremno oglišč (na primer, zgoraj levo),
- izberemo smer obhoda,
- prvi rob, ki ni vodoraven, dobi kodo 1,
- preostale robove zakodiramo po zgornjem postopku.

Lastnosti kode 3OT so naslednje:

- ravno (navpično ali vodoravno) črto označuje zaporedje symbolov 0,
- diagonalno črto pod kotom $\pm 45^o$ predstavlja zaporedje simbolov 1,
- koda 3OT je neodvisna glede na vrtenje (za $\pm 90^o$ in 180^o) in zrcaljenje.

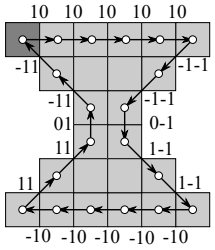
Slika 7.7 kaže objekt z označenimi kodami VCC in 3OT.



Slika 7.7: Geometrijski objekt, opisan z VCC (a) in 3OT (b)

7.1.6 Nepredznačena verižna koda Manhattan

Nepredznačena verižna koda Manhattan (angl. Unsigned Manhattan Chain Code – UMCC) je predstavljena v [34]. Koda opisuje premikanje po robnih pikslih ločeno v smeri x in y. Če od dveh zaporednih robnih pikslov odštejemo njuni koordinati x in y, dobimo predznačeno verižno kodo Manhatten (MCC), katere abeceda je $\Sigma(MCC) = \{-1, 0, 1\}$. Na sliki 7.8 vidimo primer opisa meje geometrijskega objekta z MCC, kode pa povzema razpredelnica 7.3:



Slika 7.8: Geometrijski objekt, opisan s predznačeno verižno kodo Manhattan

Tabela 7.3: Predznačena verižna koda Manhattan za objekt iz slike 7.8

MCC_x	1 1 1 1 1 -1 -1 0 1 1 -1 -1 -1 -1 -1 1 1 0 -1 -1
MCC_y	0 0 0 0 0 -1 -1 -1 -1 -1 0 0 0 0 0 1 1 1 1 1

Ugotovimo, da nikoli ne more biti par verižne kode po koordinatah x in y 0, zato lahko kodo 00 uporabimo za preklapljanje med predznaki, ki ji

sledi par simbolov 0 ali 1, ki pove, v kateri smeri se je spremenil predznak. Na primer: 00 10 pomeni, da spremenimo predznak kodi x, 00 01 spremeni predznak kodi y, 00 11 pa kodama v obe smeri. Na ta način dobimo nepredznačeno verižno kodo Manhattan, katere abeceda sestoji samo iz dveh znakov, $\Sigma(MCC) = \{0, 1\}$ Verižno kodo UMCC za objekt s slike 7.8 vidimo v razpredelnici 7.4, pri čemer predpostavimo, da je začetni predznak v obeh smereh pozitiven.

Tabela 7.4: Verižna koda UMCC za objekt iz slike 7.8

MCC_x	1	1	1	1	1	**0**	**1**	1	1	0	**0**	**1**	1	1	**0**	**1**	1	1	1	1	1	**0**	**1**	1	1	0	**0**	**1**	1	1
MCC_y	0	0	0	0	0	**0**	**1**	1	1	1	**0**	**0**	1	1	**0**	**0**	0	0	0	0	0	**0**	**1**	1	1	1	**0**	**0**	1	1

Kodo lahko še nekoliko zmanjšamo. Namreč, v smeri, ki smo jo označili z 1 in pomeni spremembo predznaka vemo, da bo naslednji simbol imel vrednost 1, zato ga pri zapisu lahko izpustimo. Rezultat vidimo v razpredelnici 7.5, kjer simbol $\sqcup$ označuje mesto izpuščenega bita.

Tabela 7.5: Reducirana verižna koda UMCC za objekt iz slike 7.8

MCC_x	1	1	1	1	1	**0**	**1**	$\sqcup$	1	0	**0**	**1**	$\sqcup$	1	**0**	**1**	$\sqcup$	1	1	1	1	**0**	**1**	$\sqcup$	1	0	**0**	**1**	$\sqcup$	1
MCC_y	0	0	0	0	0	**0**	**1**	$\sqcup$	1	1	**0**	**0**	1	1	**0**	**0**	0	0	0	0	0	**0**	**1**	$\sqcup$	1	1	**0**	**0**	1	1

UMCC ima kot edina koda zmožnost zaznati monotone dele v opisanem objektu, in sicer, monotoni deli se nahajajo med dvema spremembama predznakov v posamezni koordinatni osi. UMCC lahko uporabimo tudi kot lomno kodo.

Razpredelnica 7.6 povzema lastnosti vseh obravnavanih verižnih kod, podrobnosti pa najdemo v [34].

7.2 Stiskanje pobarvank

Vrnimo se k problemu stiskanja pobarvank. Predlagana metoda [26] je nesimetrična, saj se postopek kodiranja razlikuje od dekodiranja, samo kodiranje je tudi nekoliko počasnejše od dekodiranja. Kodiranje sestoji iz naslednjih korakov:

1. Izgubni del:

Tabela 7.6: Lastnosti verižnih kod

	F8	F4	VCC	3OT	UMCC
središčna koda	D	D	N	N	D
lomna koda	N	D	D	D	D
zazna ravne črte	D	D	D	D	D
razlikuje med vod. in navp. črtami	D	D	N	N	D
zazna diagonalne črte	D	D	D	D	D
neobčutljiva na vrtenje	N	N	D	D	D
neobčutljiva na zrcaljenje	N	N	D	D	D
omogoča vrtenje	D	D	D	N	D
omogoča zrcaljenje	D	D	D	N	D
omogoča skaliranje	D	D	N	N	D
zazna monotone dele	N	N	N	N	D

- delitev slike v področja/segmentacija,
- združevanje regij in
- določitev barvne palete;

2. Brezizgubni del:
   - opis področij z verižnimi kodami,
   - Burrow-Wheelerjeva transformacija (angl. Burrows-Wheeler Transform, BWT),
   - transformacija premik naprej (angl. Move-To-Front Tansform, MTFT),
   - kodiranje dolžine enakih simbolov (angl. run-length encoding, RLE
   - priprava za zapis in
   - aritmetično kodiranje.

Določitev regij. Algoritem preiskuje sliko v prebirnem vrstnem redu. Ko naletimo na piksel, ki še ne pripada nobenenemu področju, preiščemo okolico piksla z iskanjem v širino (angl. breadth-first traversal). Piksle, katerih barva se razlikuje za manj kot uporabniško podan prag *ctol*, vključimo v regijo. S parametrom *ctol* (angl. colour tolerance) vplivamo na razmerje stiskanja in kakovost rekonstruirane slike. Preden se odločimo, ali bomo piksel dodali v regijo, opravimo še dodaten test. Če smo našli diagonalni

piksel, ki bi ga lahko vključili v regijo, preverimo še barvo preostalih dveh piksov v mreži 2×2, ali sta tudi njuni barvi manjši od *ctol*. Če pogoj ni izpolnjen, diagonalnega piksla ne vključimo v regijo. S tem zmanjšamo možnost prepletanja regij, nevarnost, ki jo bomo pojasnili pri rekonstrukciji.Če piksel sprejmemo, izračunamo kumulativno vrednost barve regije.

Združevanje regij. V tem koraku priključimo regije, ki so manjše od praga *msz* (angl. minimal size), eni izmed sosednjih regij. Določitev najprimernejše regije opravimo na naslednji način. Najprej preverimo vrednost komponente Y iz barvnega prostora $YCbCr$. Če je vrednost manjša od 100, obravnavamo regijo kot del konture (črnega roba, ki pogosto ločuje posamezne regije). Če je vrednost komponente svetlosti $Y > 100$, izberemo regijo z najbolj podobno barvo.

Določitev barvne palete. Pobarvanke tipično sestojijo iz majhnega števila barv, zato je smiselno uporabiti barvno paleto. Zaradi barvne tolerance iz prvega koraka in združevanje regij v drugem koraku, se lahko barva posamezne regije nekoliko razlikuje od izvorne barve. Za to, da bi te razlike med dejansko in izvorno barvo odpravili, uporabimo še tretji uporabniško določen parameter *ptol* (angl. palette tolerance). Barvi, ki se razlikujeta za manj kot *ptol* obravnavamo kot eno. Skupno barvo zračunamo uteženo glede na število sprejetih regij. Če barva regije ni dovolj podobna nobeni drugi barvi, jo kot novo barvo vstavimo v barvno tabelo.

Konstrukcija verižnih kod. Regije opišemo z verižnimi kodami (angl. chain code). Avtorji so uporabili nekoliko spremenjen VCC. Zaporedja simbolov verižnih kod VCC nato zlepimo v eno samo dolgo zaporedje ter jih s tem pripravimo za transformacije, ki sledijo.

Transformaciji BWT in MTFT. BWT in MTFT smo že spoznali. BWT preuredi vhodno zaporedje na način, da je veliko enakih simbolov v preurejenem zaporedju skupaj. Pri verižni kodi VCC, ki kjer je $|\Sigma| = 3$, je ta lastnost še toliko bolj izrazita. Za BWT sledi MTFT, ki opravi preslikavo iz prostora pikslov v prostor indeksov ter na ta način praviloma zmanjša informacijsko entropijo ter s tem omogoča boljše stiskanje. Zaporedje enakih simbolov se preslika v indekse 0, zaporedje izmenjajočih se simbolov v indekse 1, zaporedje izmenjajočih se dvojic pa v indekse 2. V našem primeru, ko je abeceda zelo majhna, lahko pričakujemo samo dolga zaporedja indeksov 0, ki jih nato stisnemo z RLE.

RLE. Algoritem z RLE stiska samo zaporedja ničel. V način RLE preklopimo s tem, ko zapišemo v izhodni niz 0, zatem pa moramo zapisati še število ničel v nizu. Algoritem izbira med naslednjimi možnostmi, ki jih kodiramo z dvema bitoma:

- Zaporedja ničel, krajša od praga t_1, zakodiramo z unarno kodo.
- Zaporedja ničel, daljša od t_1 in krajša od $2^{b_1} + t_1$ zakodiramo binarno z b_1 biti.
- Zaporedja ničel, daljša od $2^{b_1} + t_1$ in krajša od $2^{b_2} + t_1$ zakodiramo binarno z b_2 biti.
- Zaporedja ničel, daljša od $2^{b_2} + t_1$ razbijemo v krajše dele in jih zakodiramo s predhodnimi možnostmi.

Vrednosti t_1, b_1 in b_2 določimo eksperimentalno, pri čemer veljajo naslednje relacije:

- $0 \leq t_1, b_1 < 8$,
- $1 \leq b_2 \leq 16$,
- $b_1 \leq b_2$ in
- $t_1 \leq 2^{b_1}$.

Priprava podatkov za zapis. Format je sestavljen iz treh logičnih delov (glej sliko 7.9):

- metapodatki o sliki,
- metapodatki o regijah in
- kodirani podatki RLE.

Metapodatki o sliki so naslednji:

- 12 bitov za širino slike,
- 12 bitov za višino slike,
- 16 bitov za število regij,
- 4 bite, s katerimi določamo število barv n_c,
- n_c bitov, s katerimi povemo, koliko barv je v paleti

- 24 bitov za vsako barvo RGB.

Metapodatki za opis regije so naslednji so:

- l_x bitov za x-koordinato začetka verižne kode,
- l_y bitov za y-koordinato začetka verižne kode,
- n_c bitov za indeks barve.
- koda spremenljive dolžine za opis regije,
- 32 bitov za indeks BWT,
- 3 bite za t_1,
- 3 bite za b_1 in
- 4 bite za b_2.

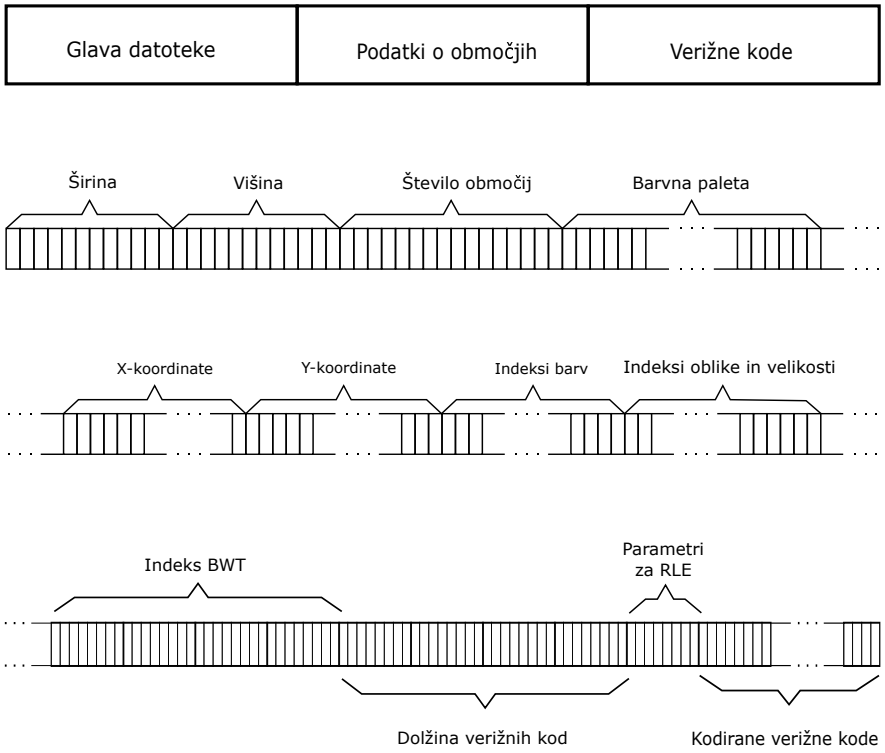
Zatem sledijo podatki RLE, s katerimi zakodiramo z BWT in MTFT transformirano verižno kodo VCC.

Aritmetično kodiranje. Podatke, organizirane kot kaže slika 7.9, v zadnjem koraku zakodiramo z odprtokodnim aritmetičnim kodirnikom PAQ8L [35].

7.2.1 Razširjanje

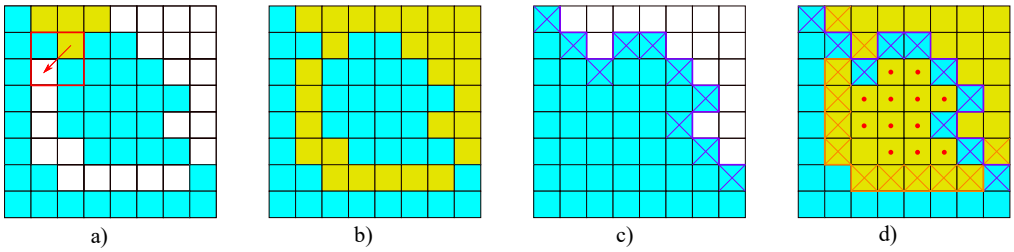
Brezizgubni del algoritma opravimo v obratnem vrstnem redu: aritmetično dekodiranje, branje metapodatkov, dekodiranje RLE, inverzna MTFT, inverzna BWT in končno pridobitev zaporedja verižnih kod VCC. Algoritem nato izriše simbole verižne kode in doboljena področja zapolni z barvo, določeno z barvnim indeksom. Za polnjenje področij uporabimo algoritem poplavljanja (angl. flood-fill). Ker izgubnega dela v fazi razširanja ne opravimo, je razširjanje in in prikaz slike hitrejše kot postopek stiskanja.

Na koncu se vrnimo še na korak konstrukcije regij. Na sliki 7.10 vidimo primer, ko lahko napačno rekonstruiramo regijo. Na modro regijo (slika 7.10a) smo naleteli najprej, določili njeno mejo in jo shranili. Nato smo zaznali rumeno regijo in jo pričeli določati z razvojem v širino. Trenutno smo v pikslu, iz katerega kaže rdeča puščica v področje, ki je tudi rumeno. V kolikor ne bi s pravilom preverjanja diagonalnih pikslov preprečili polnjenja tega dela, bi dobili situacijo, ki jo kaže slika 7.10b. Pri rekonstrukciji bi najprej rekonstruirali modro regijo, ki je shranjena prva. Rezultat kaže



Slika 7.9: Organizacija podatkov

slika 7.10c, kjer smo robne piksle označili z violičnimi križci. Tem pikslom pravimo, da so zaklenjeni in njihove barve več ne smemo spreminjati. Pri rekonstrukciji pa bi kljub vsemu prišlo do prepletanja regij in dobili bi situacijo na slika 7.10d. Piksli, ki so označeni z rdečimi pikami, so napačno dodeljeni rumeni regiji. Zato ne smemo opisati rumene regije kot ene regije, ampak kot dve, kar dosežemo o omejitvijo razvoja v širino.



Slika 7.10: Možnost napačnega polnjenja regij

7.2.2 Rezultati

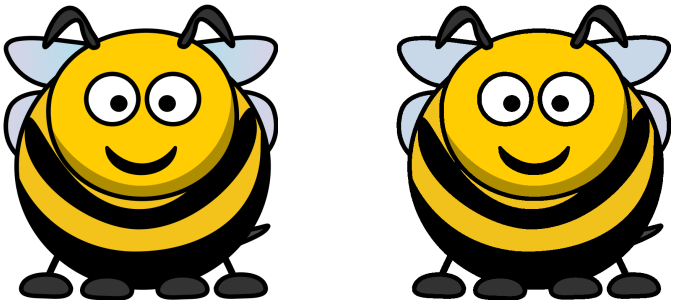
Rezultate povzemamo po [26], kjer bralec najde tudi detajle. Avtorji so predlagali tudi vrednosti za izgubni del, kot sledi:

- če ima slika tanke črte med regijami, naj bo $msz \leq 10$;
- če ima slika izrazito gladke črne robove, potem naj bo $ctol \geq 50$;
- če se sosednje barve občutno ne razlikujejo, naj bosta $ctol \leq 24$ in $ptol \leq 25$.

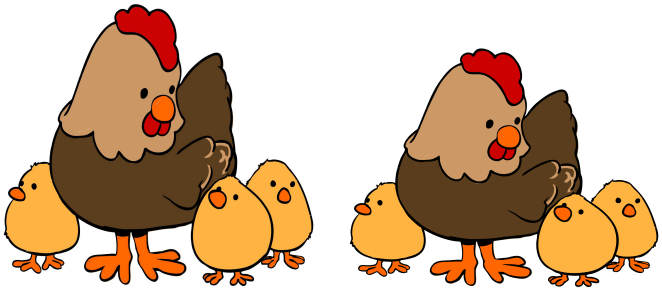
Privzete vrednosti pa so naslednje: $ctol = 60$, $ptol = 100$ in $msz = 50$. Čas, potreben za stiskanje slike velikosti 1920×1371 je bil 2.1 s, za dekodiranje pa 0.5 s. Eksperimenti so pokazali, da predlagan algoritem zelo učinkovit v primerjavi s splošnonamenskimi metodami. V povprečju je dosega stiskanje 0.075 bpp. Referenčne metode so dosegle:

- štiriško drevo: 0.3 bpp,
- PNG, ki je na vhod dobil sliko po izgubnem delu predlaganega algoritma: 0.41 bpp,
- JPEG 0.90 bpp,
- JPEG2000 0.75 bpp,
- WebP 0.41 bpp,
- SPIHT 0.57.

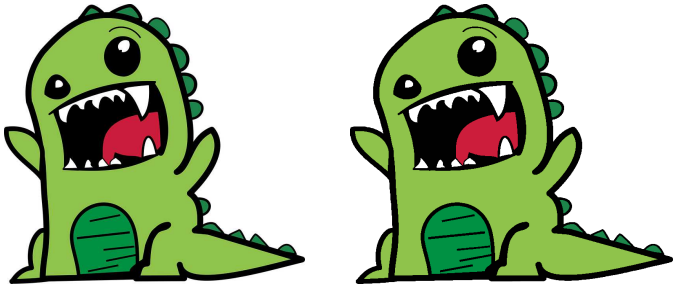
pri čemer smo pri vseh metodah dosegli primerljivo kakovost glede na SSIM (angl. Structural Similarity Index). Primer rekonstruiranih slik vidimo na slikah 7.11 – 7.20, pri čemer so izvorne slike na levi, rekonstruirane pa na desni.



Slika 7.11: Stiskanje risanih slik: primer 1.

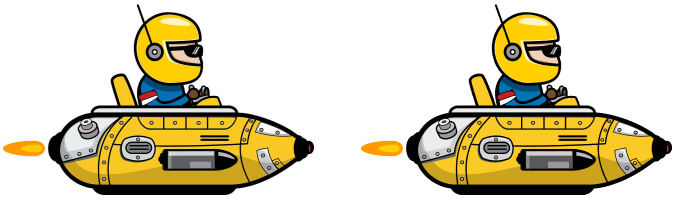


Slika 7.12: Stiskanje risanih slik: primer 2.

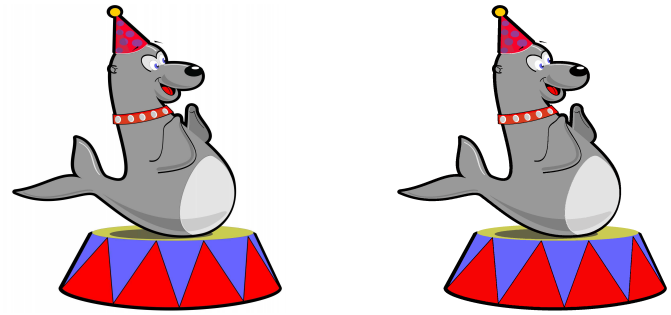


Slika 7.13: Stiskanje risanih slik: primer 3.

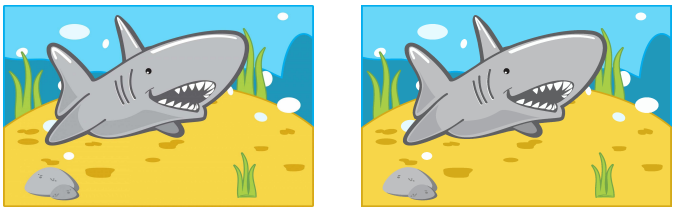
Slika 7.14: Stiskanje risanih slik: primer 4.



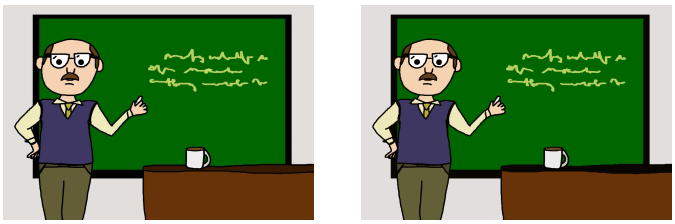
Slika 7.15: Stiskanje risanih slik: primer 5.



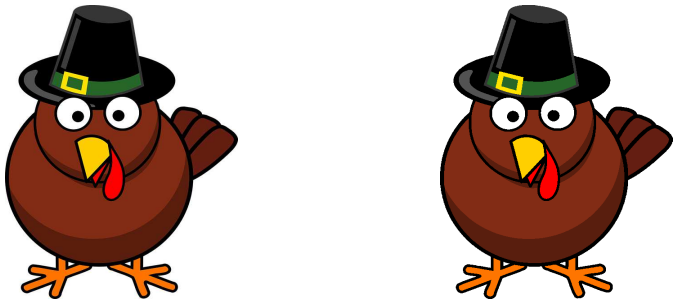
Slika 7.16: Stiskanje risanih slik: primer 6.



Slika 7.17: Stiskanje risanih slik: primer 7.



Slika 7.18: Stiskanje risanih slik: primer 8.



Slika 7.19: Stiskanje risanih slik: primer 9.

Slika 7.20: Stiskanje risanih slik: primer 10.

Poglavje 8

Krivulje polnjenja prostora

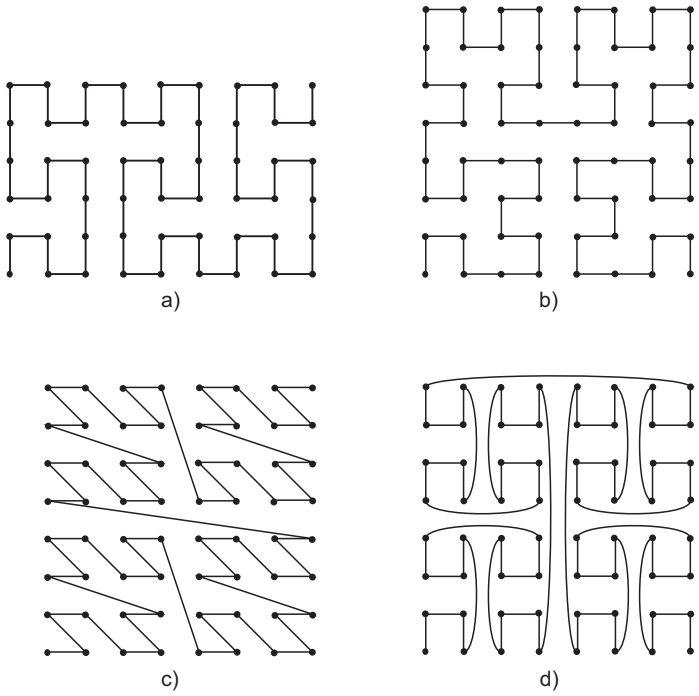
Za preiskovanje enodimenzionalnega prostora poznamo učinkovite pristope, vključujoč preprosto bisekcijo, binarna drevesa in preskočne sezname. Problem v večdimenzionalnih prostorih je mnogo bolj pereč. Razvitih je bila množica iskalnih strategij in podatkovnih struktur (štiriško in osmiško drevo za 2D in 3D, kd-drevo, R-drevo in mnoge druge). Zelo pogosto pa moramo učinkovito iskati večdimenzionalne podatke, ki so shranjeni zaporedno v linearnih podatkovnih strukturah, na primer v podatkovni bazi. V tem primeru govorimo o tako imenovanem *prostorskem indeksu* (angl. spatial index), ki večdimenzionalne podatke enolično preslika v enodimenzionalni prostor. Večdimenzionalne podatke si lahko predstavljamo kot točke v prostoru, skozi katere želimo skonstruirati krivuljo tako, da le-ta vsako točko obišče natanko enkrat. Takšnim krivuljam pravimo *krivulje polnjenja prostora* (angl. space filling curves - SFC). Prvo krivuljo SFC je izumil Giuseppe Peano leta 1890. David Hilbert je le eno leto kasneje podal geometrični opis drugačne krivulje polnjenja prostora, ki jo imenujemo Hilbertova krivulja. Leta 1966 je Morton predstavil Z-urejeno krivuljo (angl. Z-order curve) ali Mortonovo krivuljo, poznamo pa se Grayevo krivuljo. Vse štiri krivulje vidimo na sliki 8.1.

Matematična definicija krivulje polnjenja prostora najdemo v [36], nas pa bo zanimalo predvsem njihova konstrukcija in uporaba.

8.1 Konstrukcija Hilbertove krivulje v ravnini

Preden si pogledamo proces konstrukcije, definirajmo pojem *red krivulje* in *aproksimacija*.

- Red krivulje je število korakov ali iteracij, ki smo ga opravili s procesom



Slika 8.1: Krivulje polnjenja prostora v 2D: Peanova krivulja (a), Hilbertova krivulja (b), Z-urejena krivulja (c) in Grayeva krivulja (d)

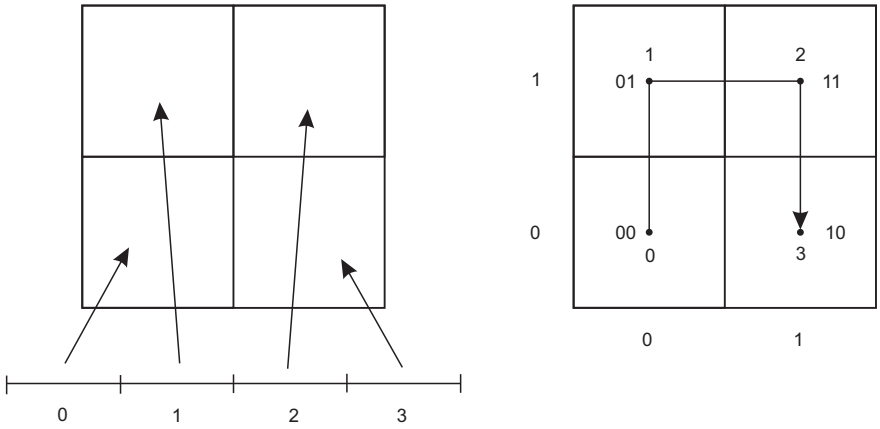
konstrukcije.

- Hilbertova krivulja končnega reda je aproksimacija Hilbertove krivulje polnjenja prostora, saj ne gre skozi vsako točko prostora ampak samo skozi središčno točko končnega števila enako velikih celic, katerih unija pokrije celoten prostor. V ravnini je celica kvadratna, prostor pa je enotski kvadrat. V nadaljevanju bomo uporabljali pojem celica in podcelica.

Konstrukcijo Hilbertove krivulje opravimo v naslednjih korakih:

Konstrukcija krivulje reda 1. Enotski interval $[0, 1]$ in enotsko celico $[0, 1]^2$ na začetku razdelimo na četrtine. Vsak podinterval nato priredimo podcelici na ta način, da si podcelice, ki so preslikane iz sosednjih podintervalov, delijo rob (slika 8.2), s čimer vzpostavimo urejenost med podcelicami. Krivulja, ki jo potegnemo skozi središče podcelic, je Hilbertova krivulja prvega reda. Če uvedemo binarno označevanje celice v $x-$ in $y-$osi, določimo

binarne kode posameznim točkam krivulje. Opazimo, da se kode ne skladajo povsem z vrstnim redom točk, s čimer pa se bomo ukvarjali nekoliko kasneje.

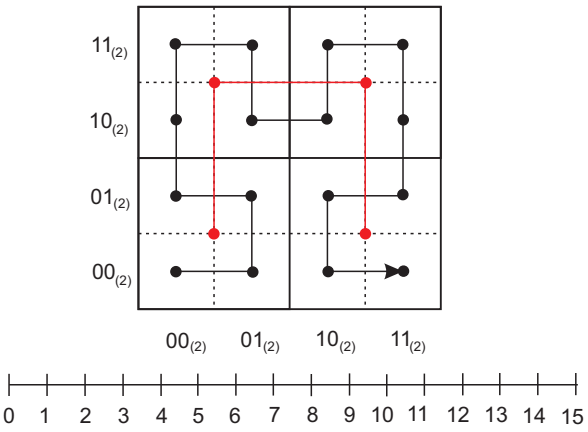


Slika 8.2: Konstrukcija Hilbertove krivulje 1. reda

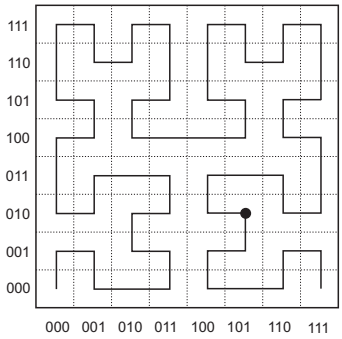
Konstrukcija krivulje reda 2. Ponovimo proces delitve za vsak podinterval in podcelico. Rezultat so štiri skupine enako velikih podintervalov in podcelic. Ker so podcelice urejene, so urejene tudi skupine. Znotraj vsake skupine vzpostavimo preslikavo med podintervali in podcelicami. Tudi tokrat pazimo, da si podcelice delijo robove tako, da ohranjamo sosednost preslikav iz intervalov. Na sliki 8.3 vidimo rezultat – Hilbertovo krivuljo drugega reda. Opazovanje nas pripelje do splošnega pravila za rekurzivno konstrukcijo Hilbertove krivulje reda k.

Konstrukcija krivulje reda k. Krivuljo reda k skonstruiramo tako, razdelimo vse podcelice v 4 manjše podcelice, nato pa zamenjamo vsako točko na krivulji s pomanjšano krivuljo prvega reda. Ob opazovanju slik 8.2 in 8.3 ugotovimo, da krivulja višjega reda sestoji iz štirih krivulj prvega reda, pri čemer imata prva in četrta krivulja različno orientacijo, druga in tretja pa enako orientacijo kot krivulja prvega reda. Vstavljene krivulje prvega reda po rotaciji povežemo tako, da je zadnja točka trenutne krivulje povezana s prvo točko naslednje krivulje. Razdalja med vsakim takšnim parom točk je enaka razdalji med katerokoli drugo točko na krivulji reda k. Navedena dejstva preverimo na sliki 8.4.

Posledica rekurzivne/hierarhične delitve prostora med konstrukcijo Hilbertove krivulje je, da točke krivulje, pozicionirane znotraj določenega podcelice, torej, točke, ki so prostorsko blizu, ostanejo blizu tudi v njihovi line-



Slika 8.3: Konstrukcija Hilbertove krivulje 2. reda



Slika 8.4: Hilbertova krivulja 3. reda

arni razporeditvi.

8.1.1 Drevesna predstavitev SFC

Proces rekurzivne delitve lahko predstavimo tudi z drevesom, ki ponazarja proces preslikave in omogoča zasnovo algoritmov za prostorsko iskanje. Drevo bo očitno uravnoteženo in njegova višina bo enaka redu krivulje. Pri razlagi si bomo pomagali z dvema pojmoma:

- **n-točka** (angl. n-point) je množica enobitnih koordinat točke, ki leži na krivulji prvega reda zlepljene v enovito n-bitno kodo(2-bitno v 2D primeru). n-točka predstavlja središče ustrezne podcelice prostora.

- **Ključ/Dolžina** (angl. derived key) predstavlja število točk, skozi katere je šla SPC in ustreza binarni kodi intervala, ki se preslika v ustrezno podcelico. Ključ je predstavljen z n-bitno vrednostjo.

Na sliki 8.2 so n-točke zaporedje koordinat točk Hilbertove krivulje (00, 01, 11, 10), ključi pa ustrezajo zaporedju intervalov (0, 1, 2, 3) na številski premici oziroma njihove binarne vrednosti (00, 01, 10, 11). V korenu drevesa predstavimo Hilbertovo krivuljo 1. reda s tem, da zapišemo n-točke in ključe/dolžine ter jih uredimo v pare. Dobimo (00, 00), (01, 01), (10, 11) in (11, 10), pri čemer je prva vrednost para dolžina, druga vrednost pa n-točka.

Rekurzivna konstrukcija krivulje drugega reda povzroči, da postane vsak izmed teh urejenih parov starš vozlišča, ki tudi vsebuje množico urejenih parov ključ – n-točka. Dve izmed vozlišč potomcev hranita enako preslikavo kot njun starš, drugi dve pa različno. Razloge, zakaj sta dva potomca enaka staršu, bomo razložili nekoliko kasneje pri diagramih stanj. Ugotovili bomo, da diagram stanj lahko nadomesti drevo poljubne globine. Primer drevesa globine 3 vidimo na sliki 8.4.

Potem, ko smo skonstruirali drevo, lahko iz koordinate točke določimo dolžino/ključ D, ki jo na začetku inicializiramo kot $D = \{\}$. Vzamimo točko $(5, 2)$, ki leži na Hilbertovi krivulji 3. reda (na sliki 8.4 je označena s krožcem). Koordinate najprej pretvorimo v binarne $(101_{(2)}, 010_{(2)})$, iz katerih tvorimo zaporedje n-točk $(10, 01, 10)$. Preiskovanje drevesa iz slike 8.5 in določitev dolžine/ključa pričnemo v korenu drevesa, kjer n-točki 10 (podčrtan na sliki) ustreza ključ 11, ki ga shranimo v spremenljivko $d = 11$, $D = D \oplus {}^1 d = 11$. Nato se preko črtkane povezave premaknemo na 2. nivo drevesa v skrajno desnega sina, na sliki 8.5 označenega kot $s = 2$. Vzamemo drugo n-točko 01 in preko nje določimo trenutni ključ $d = 01$. Z lepljenjem postavimo $D = 1101$. Nato napredujemo do zadnjega nivoja drevesa (sledimo črtkani puščici). n-točka je tokrat 10, kar nam da $d = 11$. Opravimo lepljenje $D = D \oplus d = 110111$. D je iskan ključ, ki predstavlja dolžino Hilbertove krivulje tretjega reda do točke $(5, 2)$. Pravilnost ključa $D = 110111_{(2)} = 55_{(10)}$ lahko hitro preverimo na sliki 8.4 s tem, da preštejemo število točk na Hilbertovi krivulji do točke na položaju $(5, 2)$.

Po podobnem postopku lahko iz ključa rekonstrukiramo tudi koordinato iskane točke. V nadaljevanju bomo spoznali še druge postopke preslikav med koordinatami točk in izpeljanim ključem.

[1] označuje operacijo lepljenja nizov

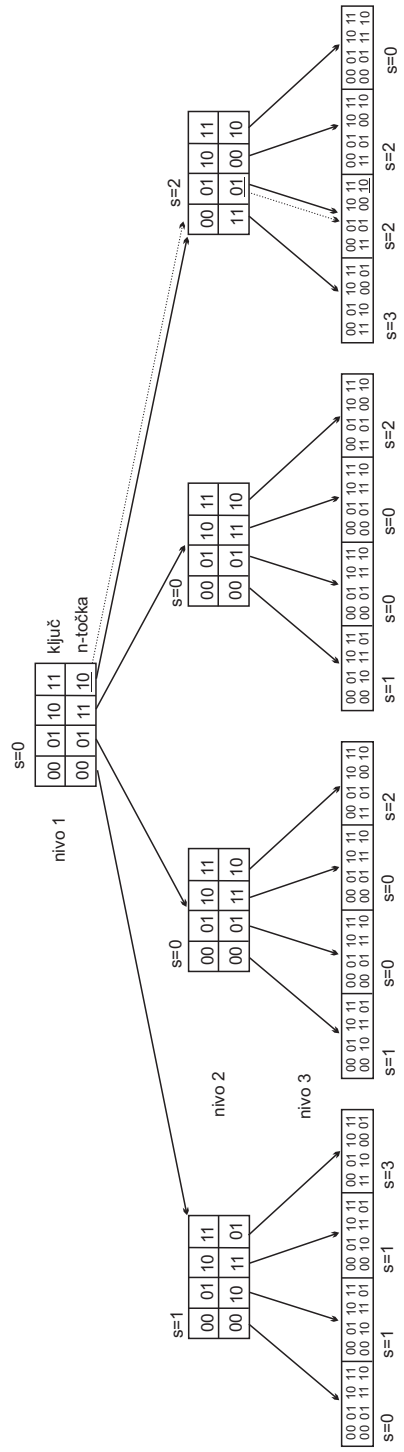
8.2 Tehnike za transformacijo v in iz prostora Hilbertove krivulje

Preslikavo med položajem na Hilbertovi krivulji in točko v prostoru ter obratno lahko realiziramo z drevesom, kot smo pravkar spoznali. Število tipov vozlišč v tem drevesu, to je orientacij Hilbertovih krivulj 1. reda, pa je končno neglede na globino drevesa. To lahko hitro preverimo iz grafične predstavitve krivulje na sliki 8.5. Zato lahko drevo predstavimo z diagramom stanj, kjer vsako vozlišče drevesa predstavlja stanje. Ko nivo drevesa preseže relativno nizek prag, je vozlišč več kot stanj, zato je diagram stanj mnogo bolj kompaktna oblika. Velikost diagrama stanj je določena z dimenzijo krivulje, z obliko krivulje na 1. nivoju in z načinom, kako se krivulja 1. reda preslika v krivuljo 2. reda. Z nekoliko premisleka lahko diagram stanj v 2D sestavimo ročno (vidimo ga na sliki 8.6). Takšen diagram stanj je znan tudi za 3D, za višje dimenzije pa je algoritmičen postopek razvil Bially [37].

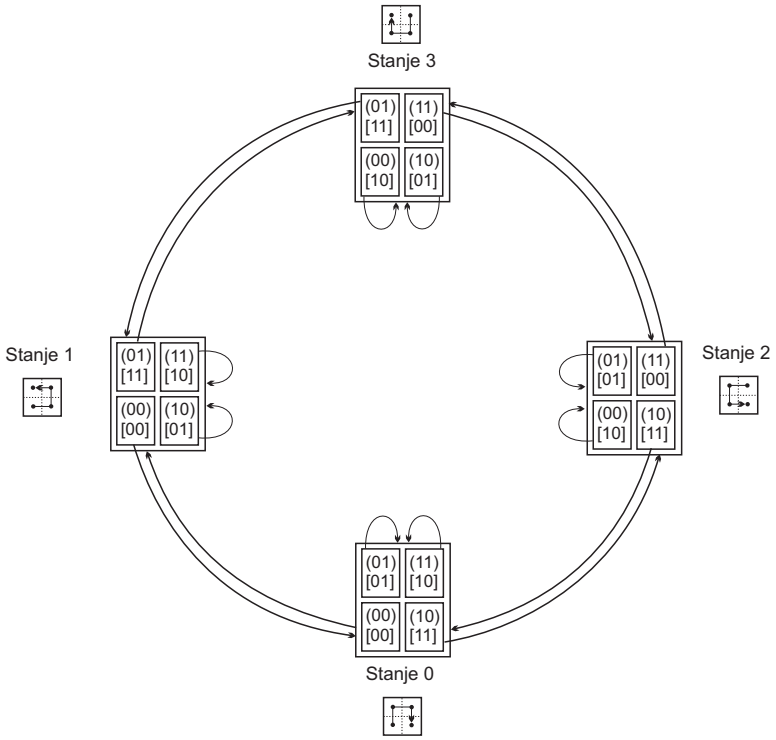
Poglejmo si primer uporabe diagrama stanj za Hilbertovo krivuljo tretjega reda in točko $(5, 2)$ s slike 8.4. Tako kot v primeru drevesa tudi tokrat najprej določimo zaporedje n-točk $(10, 01, 10)$. V diagram stanj vstopimo v stanje 0. Poiščemo n-točko (10) in postavimo $D = d = 11$. Vidimo, da iz para $\{(10)[11]\}$ izhaja puščica v stanje 2, zato se v to stanje tudi premaknemo in poiščemo n-točko (01). Ta nam da $d = 01$ in $D = 1101$. Kvadrant, kjer se nahaja n-točka (01), pa ima puščico, ki ne prehaja v drugo stanje, zato ostanemo v stanju 2. Zadnja n-točka je 10, ki jo poiščemo v stanju 2 in dobimo $d = 11$. Dobljena dolžina $D = 110111_{(2)}$, kar je enako, kot smo dobili v primeru drevesa.

Za vajo si poglejmo še primer Hilbertove krivulje reda 4 na sliki 8.7. Izberimo točko $(9, 12) = (1001_{(2)}, 1100_{(2)})$ ter tvorimo zaporedje n-točk $(11, 01, 00, 10)$. Pričnemo v stanju 0, poiščemo $(11), D = d = [10]$. Ostanemo v stanju 0. Poiscemo $(01), d = 01, D \oplus d = 1001$. Ostanemo v stanju 0. Poiščemo $(00), d = 00, D \oplus d = 100100$. Premaknemo se v stanje 1. (10) nam da $d = 01$ tako da dobimo $D = 10010001_{(2)} = 145_{(10)}$. Poglejmo še zadnji primer za položaj $(3, 6) = (0011_{(2)}, 0110_{(2)}$, ki tvori zaporedje n-točk $(00, 01, 11, 10)$. Prvo n-točko poiščemo v stanju 0 in dobimo $D = 00$. Premaknemo se v stanje 1, ker nam druga n-točka da $d = 11, D = 0011$. Iz diagrama stanj ugotovimo, da je potreben premik v stanje 3, ker nam tretja n-točka (11) da $d = 00, D = 001100$. Tudi tokrat se moramo premakniti v naslednje stanje, to je stanje 2. Zadnja n-točka je (10), kar da $D = 00110011$. Če to ne bi bila zadnja n-točka, bi se premaknili v stanje 0. Poglejmo še razdaljo D; $D = 00110011_{(2)} = 51_{(10)}$, kar lahko ročno preverimo s štetjem

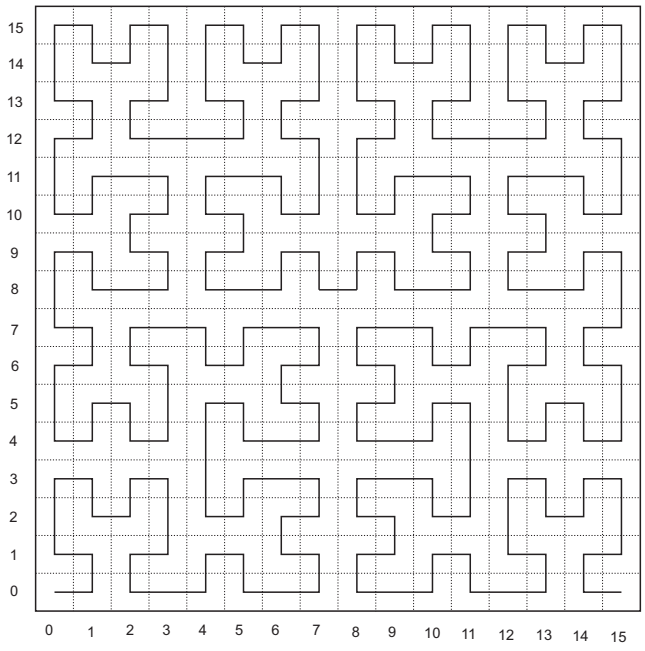
obiskanih točk.



Slika 8.5: Drevo Hilbertove krivulje 3. reda



Slika 8.6: Diagram stanj za 2D Hilbertovo krivuljo; n-točke so označene z (), ključ/razdalja pa z []



Slika 8.7: Hilbertova krivulja 4. reda

Poglavje 9

Fraktali

9.1 Fraktalna dimenzija

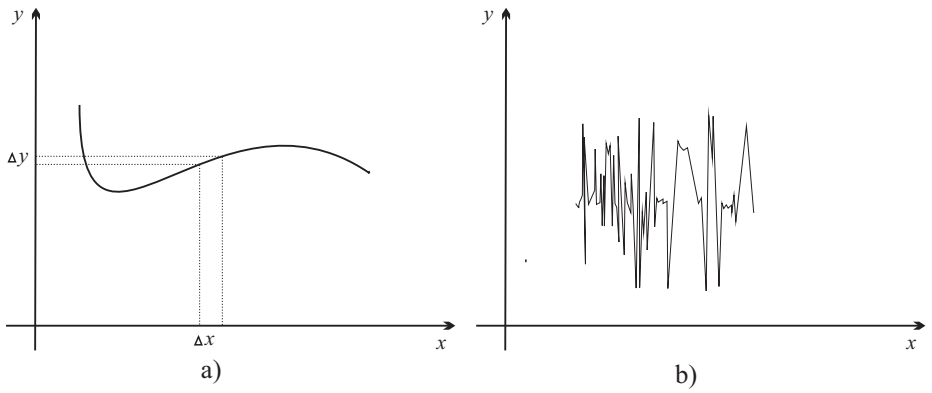
Šele pred nekaj desetletji so se znanstveniki pričeli zavedati, da nas v naravi ne obkrožajo samo idealizirani geometrijski objekti (kocke, kvadri, krogle, trikotniki, mnogokotniki), ampak da je bogastvo oblik mnogo večje. Tradicionalna geometrija se je od svojega nastanka dobrih 4000 let intenzivno ukvarjala z objekti, ki se v naravi pojavijo zelo redko, šele v zadnjih nekaj desetletij pa je bila razvita tako imenovana **fraktalna geometrija**. Fraktalna geometrija je postala ključ za reševanje množice matematičnih, inženirskih in znanstvenih problemov, je temelj za teorijo kaosa in opis dinamičnih sistemov.

S fraktalno teorijo smo se pričeli zavedati, da se geometrija ne ukvarja samo z opisom geometrijskih objektov (in relacij med njimi) ampak tudi s prostorom. Geometrijski objekti obstajajo v njihovih prostorih. Točka je 0D objekt in lahko leži na premici (v 1D prostoru), v mnogokotniku (v 2D prostoru) ali v prostorih še višjih dimenzij. Iz tega intuitivno ugotovimo, da se objekti lahko nahajajo v prostorih, ki so enake ali višje dimenzije kot so objekti sami. Prostor pomembno vpliva na karakteristike objektov. Na primer, vsi vemo, da je vsota notranjih kotov trikotnika enaka 180^o. Vendar, če trikotnik obstaja v sferičnem prostoru (če ga narišemo na primer na žogo), je vsota njegovih notranjih kotov različna od 180^o.

Opazujmo neko idealizirano krivuljo v ravnini, določeno kot $y = f(x)$. Če opazujemo samo delček te krivulje, vidimo, da ta delček vedno preide v daljico kot vidimo na sliki 9.1a. Če Δx zmanjšujemo, je odsek vedno bolj raven, torej se vedno bolj približuje daljici. Njen naklon je enak odvodu funkcije v dani točki, ki ga dobimo, ko Δx teži proti 0

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}. \tag{9.1}$$

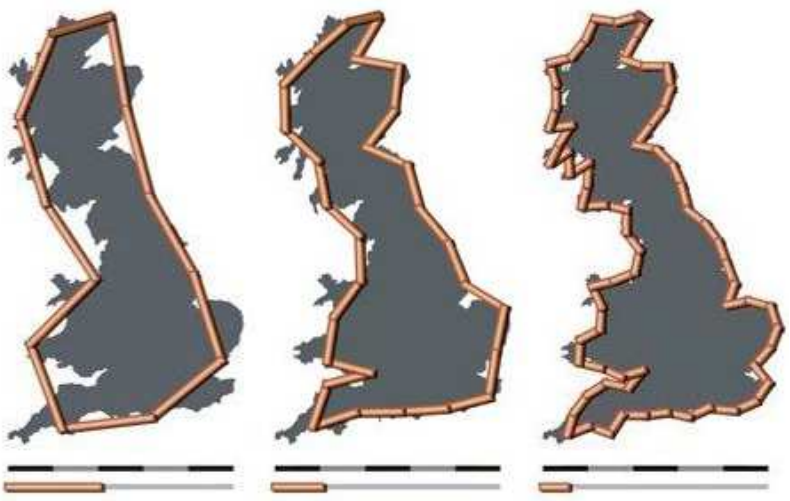
Žal pa večino krivulj, ki jih najdemo v naravi, ni takšnih, da bi jih lahko odvajali. Namesto, da bi se z zmanjševanjem Δx zmanjševal tudi Δy, kar ima za posledico zmanjševanje dolžine krivulje, se to ne zgodi. Krivulje, ki jih najdemo v naravi so sicer zvezne, niso pa odvedljive (na sliki 9.1b je primer potresne krivulje).



Slika 9.1: Odvedljiva (a) in neodvedljiva (b) krivulja

Pionir drugačnega sprejemanja geometrijskih objektov v naravi je bil Benoit Mandelbrot. Mandelbrot je postavil temelje drugačne obravnave geometrije. Za nalogo si je zadal izmeriti dolžino angleške obale, pri čemer je prišel do pomembnega paradoksa. Dolžina obale je namreč zelo odvisna od dolžine merilne letvice (glej sliko 9.2); z njenim krajšanjem se dolžina obale veča. Če dolžina merilne letvice teži proti 0, teži dolžina obale proti ∞. Pridemo do paradoksalne ugotovitve, da obstajajo geometrijski objekti (angleški otok v Mandelbrotovem primeru), ki ima neskončen obseg, čeprav obkroža otok s končno ploščino. Zaključimo lahko, da je večina krivulj v naravi neskončne dolžine, čeprav obdajajo objekt s končno ploščino. Razmišljanje lahko seveda posplošimo na ploskve. Najbolj karakterističen primer so človeška pljuča. Njihova naloga je telesu priskrbeti čim več kisika, zato morajo imeti čim večjo površino, ki pa jo moramo spraviti v naše telo, torej v končni volumen.

Očitno potrebujemo za naravne objekte drugačen opis, kot ga zmore klasična geometrija, ki temelji na idealiziranih geometrijskih objektih (gore



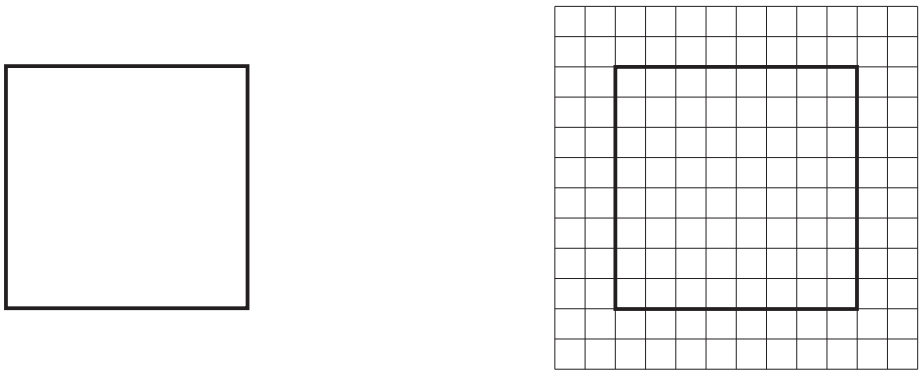
Slika 9.2: Dolžina obale je odvisna od dolžine merilne letnice

niso stožci, oblaki niso krogle), za kar pa potrebujemo drugačne tehnike.

9.2 Dimenzija

Dimenzijo smo do sedaj razumeli kot intuitivni pojem, ki ni vezan na nek geometrijski objekt (zaradi idealizacije geometrijskih objektov to niti ni bilo potrebno). Govorimo o točkah, ki so 0-dimenzionalne, premice, ki so enodimenzionalne, krivulje, ki so dvodimenzionalne. Očitno dimenzijo razumemo kot celoštevilsko vrednost. Do podobnega paradoksa v zgodovini je prišlo pri razumevanju števil. Števila, ki so jih najprej uporabljali, so bila naravna števila (6 ljudi, 2 vozova, 5 jabolk). Kakor hitro pa so pričeli meriti in tehtati, pa so morali interval med naravnimi števil razdeliti na delčke (angl. fractions). Očitno je napočil čas, da pričnemo drugače razmišljati tudi o dimenziji. Takoj opozorimo na ustrezno razumevanje dimenzije: dimenzija geometrijskega obljekta in dimenzija prostora, v katerega je objekt vstavljen, sta različna pojma. Dimenzija prostora je povezana s stopnjo svobode (angl. degrees of freedom) in jo merimo s celimi števili. To pa ne pomeni, da mora biti tudi dimenzija geometrijskega objekta celoštevilska. Geometrijski objekt namreč leži v prostoru in njegova dimenzija je povezana s tem, kako objekt zapolni prostor. Kot primer, kako nek objekt zapolni prostor, si oglejmo kvadrat (slika 9.3a). Da bi ugotovili, kako kvadrat zapolni

prostor, ga pokrijemo z enakomerno mrežo celic. V našem primeru kvadrat pokrijemo z $8 \times 8 = 64$ celic (slika 9.3b).



Slika 9.3: Kako kvadrat zapolni prostor?

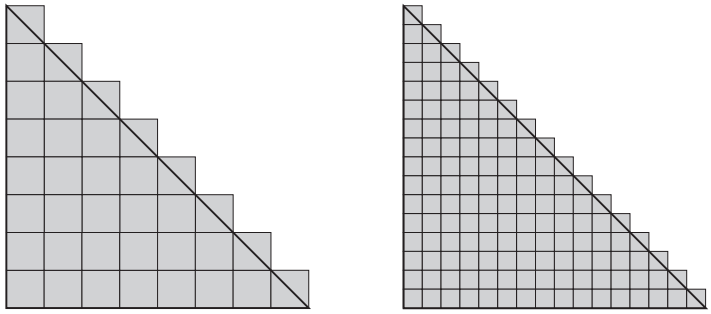
Prostor pa bi lahko razdelil tudi z manjšo mrežo. Vsako celico bi lahko razdelil v 4 manjše celice in dobil $16 \times 16 = 256$ celic, ki pokrijejo naš kvadrat. Seveda bi lahko velikost celic ponovno zmanjšali. Z ϵ označimo dolžino stranice ene celice in z N število celic. Potem lahko zapišemo naslednjo relacijo:

$$N(\epsilon) = \left(\frac{1}{\epsilon}\right)^2. \tag{9.2}$$

Število celic bo torej raslo s $(1/\epsilon)^2$, kjer število 2 določa dimenzijo prostora. Na sliki 9.4a vidimo trikotnik, ki smo ga tudi pokrili z mrežo celic tako, da je v celoti pokrit s celicami. Na sliki 9.4b smo isti trikotni prekrili z manjšimi celicami. Vidimo, da se manjše celice bolje prilegajo robu trikotnika. Celice lahko še zmanjšamo in dobimo še boljše prileganje. Zato lahko za trikotnik zapišemo naslednjo relacijo:

$$N(\epsilon) \to \frac{1}{2}\left(\frac{1}{\epsilon}\right)^2. \tag{9.3}$$

Vidimo, da se enačbi 9.2 in 9.3 razlikujeta v konstanti, zato lahko v splošnem zapišemo



Slika 9.4: Prostor trikotnika

$$N(\epsilon) \to k \left(\frac{1}{\epsilon}\right)^2. \tag{9.4}$$

Iz enačbe želimo izločiti število 2, ki predstavlja dimenzijo prostora, v katerem se nahajajo objekti:

$$2 = \frac{\ln N(\epsilon)}{\ln \frac{1}{\epsilon}} - \frac{\ln k}{\ln \frac{1}{\epsilon}} \tag{9.5}$$

Še zmanjšujemo $\epsilon \to 0$, potem gre imenovalec drugega ulomka proti ∞, posledično gre drugi ulomek proti 0. Za dimenzijo objekta D v poljubnem prostoru zato lahko zapišemo

$$D = \frac{\ln N(\epsilon)}{\ln \frac{1}{\epsilon}}. \tag{9.6}$$

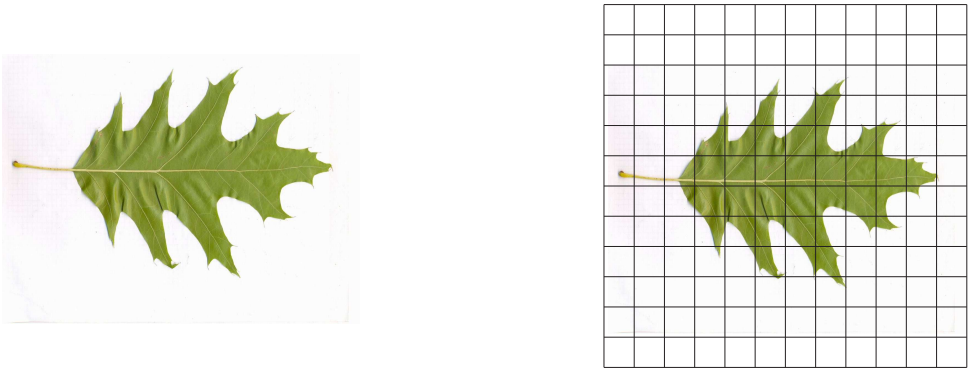
Ugotovimo, da bodo objekti v 1D prostoru teži proti vrednosti 1, v 2D prostoru proti 2 itd.

Uporabimo sedaj še geometrijski objekt iz narave (slika 9.5a). Tudi na ta objekt lahko položimo mrežo in preštejemo, koliko celic zaseda ter izračunamo fraktalno dimenzijo. Tudi tokrat lahko zmanjšujemo velikost mreže (to je parameter ϵ), kar ima za posledico povečevanje števila zasedenih celic N. Vrednosti lahko uvrščamo v enačbo 9.6. Ugotovili bi, da vedno dobivamo realno število, čigar vrednost je iz intervala med 1 in 2. Iz tega

lahko povzamemo, da so **fraktali geometrijski objekti z necelo\v{s}tevilno dimenzijo (angl. fractal dimension)**.

Samo idealni objekti (na primer kocka) imajo celo\v{s}tevilsko dimenzijo, ki je enaka dimenziji prostora, v katerega so objekti vstavljeni. Sicer pa bo ta dimenzije vedno manjša.

Najpomembnejše pa je seveda vprašanje, kakšen pomen pa ima fraktalna dimenzija objekta? Fraktalna dimenzija je mera, ki nam pove, za koliko se nek objekt razlikuje od idealnega objekta. Z drugimi besedami, fraktalna dimenzija karakterizira površje objekta. Površje objekta je pomembno v mnogih aplikacijah (na primer, elektroliza se dogaja na površju elektrod, električni stik v stikalu nastane na površju kontaktov).



a) b)

Slika 9.5: Geometrijski objekt iz narave

9.3 Mandelbrotova in Juliajeva množica

Do sedaj smo se spoznali s pojmom fraktal in fraktalna dimenzija. Seveda pa nas zanima, kako bi lahko generirali fraktale. Pričnimo s konstrukcijo fraktala, ki se nahaja na premici, i.e. je v 1D prostoru. Vemo že, da je dimenzija fraktala manjša ali kvečjemu enaka dimenziji prostora, v katerem se fraktalni objekt nahaja. Naš prostor naj bo premica, na kateri izberemo delček med 0 in 1 (slika 9.6)a. Nato razdelimo odsek med 0 in 1 na 3 enake dele in odstranimo srednji del. Situacijo po prvi iteraciji vidimo na sliki 9.6b. Postopek sedaj nadaljujemo, vsak odsek razdelimo na 3 dele in odstranimo srednjega (glej sliki 9.6c in d). S postopkom nadaljujemo. Ko gre število iteracij proti neskončno, dobimo na premici množico točk, ki ji pravimo

Cantorjeva množica. Seveda pa se postavi vprašanje, ali je ta objekt, ki smo jo dobili, fraktal. Da bi odgovorili na to vprašanje, poskusimo določiti fraktalno dimenzijo.

Slika 9.6: Cantorjeva množica

Kot smo videli do sedaj, moramo razdeliti prostor v celice in preveriti, v koliko celicah se nahaja objekt. Objekt bi lahko, recimo, pokrili z 10-timi celicamo. Vendar, predpostavimo, da razdelimo naš odsek $[0, 1]$ v 3 celic $[0, 1/3]$, $[1/3, 2/3]$ in $[2/3, 1]$. Objekt na sliki 9.6b na ta način pokrijemo z dvema objektoma ($N = 2$ in $\epsilon = 1/3$). Glede na enačbo 9.6 po prvi iteraciji dobimo

$$D = \frac{\ln 2}{\ln 3}, \tag{9.7}$$

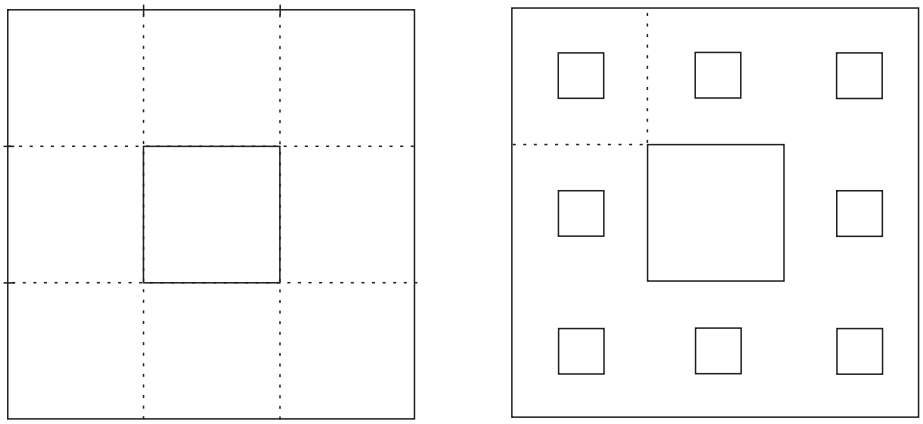
ki je očitno necelo število (angl. fraction), ki je med 0 in 1.

Poglejmo, kakšna je fraktalna dimenzija po drugem koraku (sliki 9.6c). Seveda bomo tokrat še bolj razdelili celice in sicer na 9 celic. Potem bo $N = 4$ in $\epsilon = 1/9$. Fraktalna dimenzija D je

$$D = \frac{\ln 4}{\ln 9} = \frac{\ln 2}{\ln 3}. \tag{9.8}$$

Očitno ostaja dimenzija objekta enaka in takšna bo ostala tudi po neskočno iteracijah, ko pridemo do Cantorjeve množice s podano fraktalno dimenzijo.

Oglejmo si naslednjo situacijo. V 2D prostoru imejmo pravokotnik. Stranice razdelimo na 3 enake dele, tako da kvadrat razpade na 9 podkvadratov (slika 9.7a). Zatem uporabimo enako strategijo odstranjevanja srednjega dela kot v primeru Cantorjeve množice. V naslednji iteraciji odstranimo srednji kadrat iz vsakega kvadrata, nastalega v prejšnji iteraciji (slika 9.7b).



Slika 9.7: Kvadrat Sierpinskega

Fraktalno dimenzijo kvadrata Sierpinskega tudi določimo s prekrivanjem. Prostor razdelimo v kvadrate, katerih stranice so velike $\epsilon = 1/3$. Za to, da pokrijemo objekt, potrebujemo $N = 8$ celic. Tako dobimo

$$D = \frac{\ln 8}{\ln 3}. \tag{9.9}$$

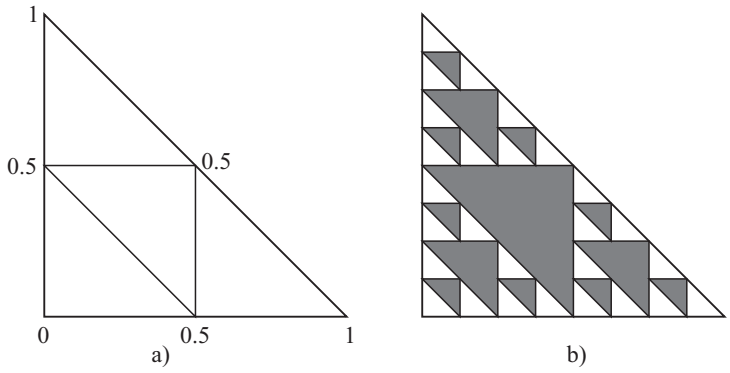
V naslednji iteraciji razdelimo vsak kvadrat na enak način. Velikost celice, ki je primerna za pokritje kvadratov je $\epsilon = 1/9$, potrebujemo pa $N = 64$ celic:

$$D = \frac{\ln 64}{\ln 9} = \frac{\ln 8}{\ln 3}. \tag{9.10}$$

Očitno je to fraktalna dimenzija tega objekta tudi če ϵ teži proti 0.

Še bolj poznan kot kvadrat Sierpinskega, je trikotnik Sierpinskega. Slika 9.8 kaže njegovo konstrukcijo. Stranice prvega trikotnika najprej razdelimo na polovice in jih povežemo (slika 9.8a). Trikotnik razpade v štiri manjše trikotnike, srednjega odstranimo. Postopek iterativno ponavljamo. Slika 9.8b kaže stanje po dveh nadaljnih iteracijah, kjer so odstranjeni trikotniki poudarjeni.

Za določitev fraktalne dimenzije ponovno razdelimo naš prostor v celice. Te celice naj bodo sedaj trikotniki. Za geometrijski objekt na sliki 9.8a



Slika 9.8: Trikotnik Sierpinskega

očitno potrebujemo 3 trikotnike (torej $N = 3$). Velikost stranice trikotnika pa je 0.5 (torej $\epsilon = 1/2$), kar nam da:

$$D = \frac{\ln 3}{\ln 2}. \tag{9.11}$$

V naslednjem koraku je $\epsilon = 1/4$ in $N = 9$, kar nam da enak D kot v enačbi 9.11.

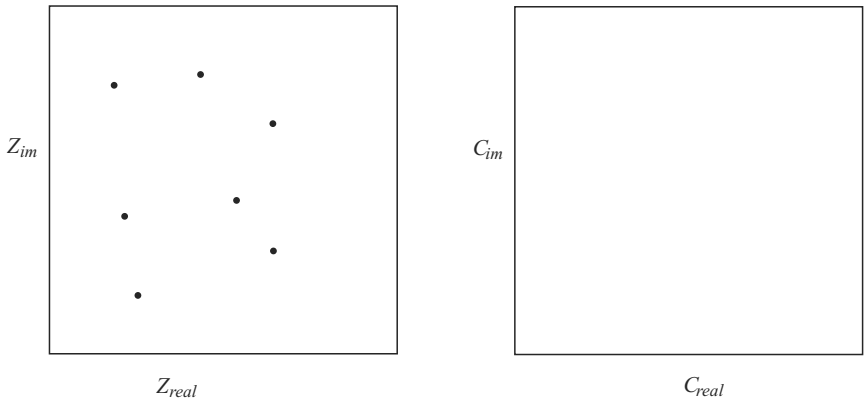
Naslednji način tvorbe fraktalov ima izvor v dinamičnih sistemih. Naj bo $x_{n+1} = f(x_n)$ funkcija, ki novo vrednost x_{n+1} dobi s funkcijsko preslikavo prejšnje vrednosti x_n. Predpostavimo, da sta x_n in x_{n+1} vektorja

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = f \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{9.12}$$

in glede na enačbo 9.12 predstavljata točko v ravnini. Če poznam točko (x_n, y_n) in funkcijo f, lahko dobim položaj nove točke (x_{n+1}, y_{n+1}). Enačbo 9.12 pa lahko poenostavimo, če vektor predstavimo kot eno kompleksno število

$$z_{n+1} = f(z_n). \tag{9.13}$$

z_n je seveda sestavljen iz dveh delov, realnega in imaginarnega $z_n = z_{real} + iz_{im}$. Funkcija f je sedaj kompleksna funkcija, ki preslika začetno



Slika 9.9: Prostor vrednosti z in parametra c

kompleksno vrednost v novo kompleksno vrednost. Imejmo naslednjo preprosto funkcijo:

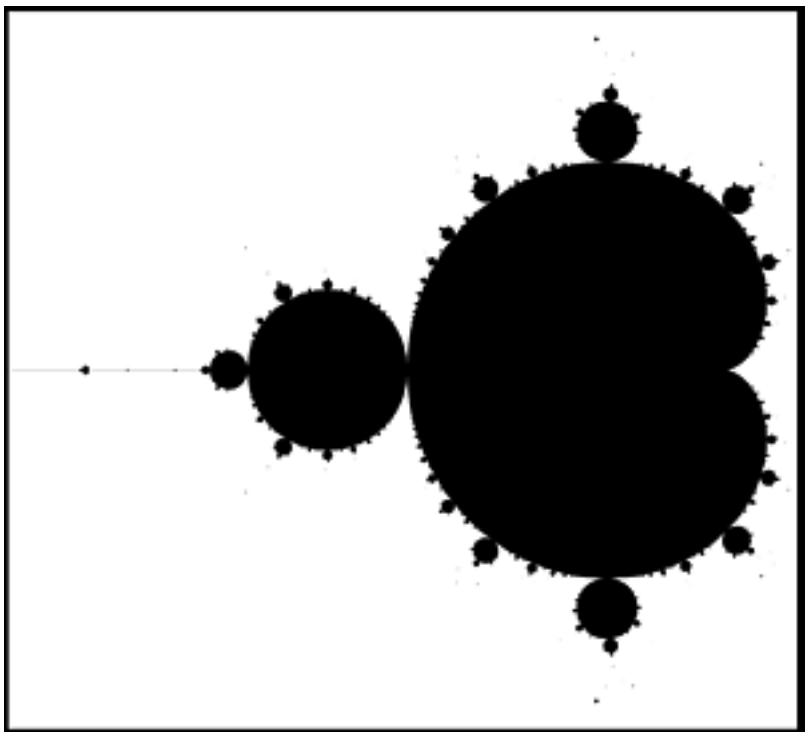
$$z_{n+1} = z_n^2 + c, \tag{9.14}$$

kjer je c kompleksno število (parameter). Če imamo sedaj neko točko v kompleksni ravnini in če poznamo c, potem lahko določimo množico točk v iterativnem postopku. Karakteristika dobljene množice pa bo očitno odvisna od vrednosti c. Na ta način lahko določimo dva prostora: prostor vrednosti z in prostor parametra c (glej sliko 9.9).

Izkaže se, da za nekatere vrednosti c gredo vrednosti z_n v interaciji proti neskončnosti, za nekatere druge pa vrednosti ostanejo omejene. Enako velja za nekatere začetne vrednosti z; za nekatere začetne vrednosti z_1 ostanejo vrednosti z_n po iteraciji omejene, za druge pa ne. Množica vrednosti c, ki ohranja vrednosti z_i omejene, se imenuje Mandelrotova množica (slika 9.10).

Množica začetnih vrednosti z_1 se imenuje Juliajeva množica. Za vsako vrednost c obstaja v Mandelrotovi množici množica začetnih vrednosti v Juliajevi množici. Omenili smo, da točke izven Mandelbrotove množice poženejo proti neskončnosti, hitrost pa je odvisna od položaja začetne vrednosti c.

Seveda se postavi vprašanje, ali je sta Mandelbrotova in Juliajeva množica fraktala. Če povečamo Mandelrotovo množico ugotovimo, da le-ta sestoji iz novih in novih struktur, ki jih na osnovni povečavi nismo videli. Hitro lahko naredimo primerjavo z obalo, kjer se nam nove strukture pokažejo



Slika 9.10: Mandelbrotova množica

šele, ko se jim dovolj približamo. To pa pomeni, da je Mandelrotova (in posledično tudi Juliajeva) množica fraktal.

Mandelbrotovo množico lahko razumemo kot parametrični prostor poljubnega sistema (na primer električnega, mehanskega). Za vsak sistem lahko tako določimo nabor parametrov, za katere sistem teži k stabilnemu stanju. V našem primeru torej za vsako vrednost parametra c dobimo množico vrednosti, ki v iteraciji ne težijo proti neskončnosti, in to je Juliajeva množica. Povzamimo, v Mandelbrotovi množici je neskonžno točk in vsaka točka ima svojo Juliajevo množico, kar je temeljna ideja dinamičnih sistemov.

Literatura

[1] D. Salomon and G. Motta. *Handbook of data compression.* Springer, 5^{th} edition, 2010.

[2] A. Tinku and P.-S Tsai. *JPEG2000 Standard for image compression.* John Wiley & Sons, 2005.

[3] B. Hamilton. *JPEG File Interchange Format.* http://www.w3.org/Graphics/ JPEG/jfif3.pdf, 1992.

[4] S. J. Sangwine and R. E. N. Horne. *The Colour Image Processing Handbook.* Chapman & Hull, 1998.

[5] C. L. Heng. *Image Compression: JPEG.* http://pascalzone.amirmelamed.co.il/Graphics/JPEG/JPEG.htm, 1997.

[6] G. K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[7] B. Furht. A survey of multimedia compression techniques and standards. part i: Jpeg standard. *Real Time Imaging*, 1(1):49–67, 1995.

[8] ITU. *Digital Compression and coding of continous-tone still images-requirements and guidelines.* International Telecommunication Union, Recomendation T-82, 1992.

[9] M. J. Weinberger, G. Seroussi, and G. Sapiro. The loco-i lossless image compression algorithm: Principles and standardization into jpeg-ls. *IEEE Transactions on image processing*, 9(8):1309–1324, 2000.

[10] M. Weinberger, G. Seroussi, and G. Sapiro. The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS. Technical report, HP Computer Systems laboratory, 1998.

[11] D. Špelič. *Postopek brezizgubnega stiskanja razčlenjenih vokselskih podatkov*. PhD thesis, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, 2011.

[12] N. Merhav, G. Seroussi, and M. J. Weinberger. Lossless compression for sources with two-sided geometric distributions. Technical report, HP-98-70, 1998.

[13] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT–12(3):399–401, 1966.

[14] R. F. Rice. Some practical universal noiseless coding techniques. Technical Report NASA-CR-158515, Jet Propulsion Lab., California Inst. of Tech., Pasadena, CA, United States, 1979.

[15] Akansu A and R. Haddad. *Multiresolution Signal Decomposition*. Academic press, 1992.

[16] W. A. Pearlman and A. Said. Set patition coding: Part i of set partition coding and image wavelet coding systems. *Fundations and trends in signal processing*, 2(2):95–180, 2008.

[17] C. Christopoulos, A. Skodras, and T. Ebrahimi. The jpeg2000 still image coding system: An overview. *IEEE Transactions on Consumer Electronics*, 46(4):1103–1127, 2000.

[18] J. M. Gilbert and R. W. Brodersen. A lossless 2D image compression technique for synthetic discrete-tone images. In J. Storer, editor, *Proceedings of the 1998 Data Compression Conference*, pages 359–368, 1998.

[19] L. Demaret, N. Dyn, and A. Iske. Image compression by linear splines over adaptive triangulations. *Signal Processing*, 16(7):1604–1616, 2006.

[20] S. Krivograd, G. Hren, B. Žalik, and A. Jezernik. Hiter algoritem za poenostavljanje in obnovitev trikotniških mrež za prenos rezultatov mke preko svetovnega spleta. *Stojniški vestnik*, 11(3):524–537, 2003.

[21] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Computer Graphics*, 26(2):65–70, 1992.

[22] M. Garland and P. S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical report, University of Illinois, 1995.

[23] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. *Computer Graphics*, 27(19–26), 1993.

[24] A. Farzaneh, H. Kheiri, and M. A. Shahmersi. An efficient storage format for large sparse matrices. *Communications Series A1 Mathematics & Statistics*, 58(2):1–10, 2009.

[25] A. Moffat and V. N. Anh. Binary codes for locally homogeneous sequences. *Information Processing Letters*, 99(5):175–180, 2006.

[26] Jeromel A and B. Žalik. An efficient lossy cartoon image compression method. *Multimedia Tools and Applications*, sprejeto v objavo, 2019.

[27] H. Freeman. Ire transactions on electronic computers. *On encoding of arbitrary geometric configurations*, EC(10):260–268, 1961.

[28] H. Freeman. Computer processing of line drawing. *ACM Computing Surveys*, 6(1):57–97, 1974.

[29] Y. K. Liu and B. Žalik. An efficient chain code with huffman coding. *Pattern Recognition*, 38(4):553–557, 2005.

[30] E. Bribiesca. A geometric structure for two-dimensional and three-dimensional surfaces. *Pattern Recognition*, 25(5):483–496, 19962.

[31] P. Nunes, F. Pereira, and F. Marqués. Multi-grid chain coding of binary shapes. In *ICIP'97 Proceedings of the 1997 International Conference on Image Processing*, volume 3, pages 114–117, 1997.

[32] E. Bribiesca. A new chain code. *Pattern Recognition*, 32(2):235–251, 1999.

[33] H. Sáchez-Cruz and R. M. Rodríguez-Diaz. Compressing bi-level images by means of a 3-bit chain code. *SPIE Optical Engineering*, 44(9):1–8, 2005.

[34] B. Žalik, D. Mongus, Y. K. Liu, and N. Lukaˇb. Unsigned manhattan chain code. *Journal of Visual Communication and Image Representation*, 38(C):186–194, 2016.

[35] Mahoney A. Data compression programs. `http://mattmahoney.net/dc/`, dostop 2019.

[36] H. Sagan. *Space-filling curves*. Springer-Verlag, 1994.

[37] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, 15(6):658–664, 1969.