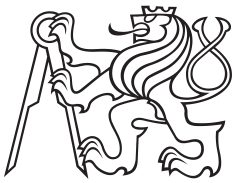
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra radioelektroniky

Algoritmy zpracování obrazu v reálném čase

Jan Skalička

Vedoucí: doc. Ing. Stanislav Vítek, Ph.D.
Květen 2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Skalička** Jméno: **Jan** Osobní číslo: **491897**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra radioelektroniky**

Studijní program: **Elektronika a komunikace**

Specializace: **Komunikace a zpracování informace**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Algoritmy zpracování obrazu v reálném čase

Název diplomové práce anglicky:

Real-Time Image Processing Algorithms

Pokyny pro vypracování:

Cílem práce je implementace a případná optimalizace algoritmů komprese obrazu, např. dílčích částí kodeku JPEG XT.
Při zpracování postupujte podle následujících pokynů:
1) prostudujte metody efektivní implementace algoritmů zpracování obrazu v reálném čase pro platformy s limitovanými zdroji, např. ARM SoC;
2) vybrané algoritmy implementujte ve vhodném programovacím jazyce na vývojové desce Basler daA2500-60mc-SD820-DB8 s procesorem Qualcomm Snapdragon 820 a kamerou DART s rozhraním MIPI;
3) implementované programové vybavení otestujte v reálném provozu a diskutujte případné nedostatky.

Seznam doporučené literatury:

[1] SUNDARARAJAN, Duraisamy. Digital image processing: a signal processing and algorithmic approach. Springer, 2017
[2] TSCHUMPERLE, David; TILMANT, Christophe; BARRA, Vincent. Digital Image Processing with C++: Implementing Reference Algorithms with the CImg Library. CRC Press, 2023.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Stanislav Vítek, Ph.D. katedra radioelektroniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **07.02.2024** Termín odevzdání diplomové práce: **24.05.2024**

Platnost zadání diplomové práce: **21.09.2025**

doc. Ing. Stanislav Vítek, Ph.D. podpis vedoucí(ho) práce	doc. Ing. Stanislav Vítek, Ph.D. podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. Datum převzetí zadání

Podpis studenta

Poděkování

Rád bych poděkoval vedoucímu mé práce doc. Ing. Stanislavu Vítkovi, Ph.D. za vedení práce, za zapůjčení zajímavého hardwaru a za vhled do oblasti kompresních algoritmů.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 24. května 2024

Abstrakt

Komprese obrazu je ve světě stále rozšiřujících se multimédií velmi důležitá. Metody komprese obrazu často využívají nedokonalostí lidského zraku, díky čemuž lze některé obrazové informace vynechat. Správnou volbou vynechaných informací lze zajistit vysoký kompresní poměr s nízkým zhoršením vnímané kvality.

Klíčová slova: ztrátová komprese, obraz, kosinová transformace, lidský zrak, multimédia, entropie, bezeztrátová komprese, Huffmanovo kódování, optimalizace algoritmů

Vedoucí: doc. Ing. Stanislav Vítek, Ph.D.

Abstract

Image compression is very important in the ever-expanding world of multimedia. Image compression methods often take advantage of the imperfection of human vision, due to which some image information can be omitted. By choosing the right information to omit, a high compression ratio can be ensured with low perceived quality degradation.

Keywords: lossy compression, image, cosine transform, human vision, multimedia, entropy, lossless compression, Huffman coding, algorithm optimization

Title translation: Real-Time Image Processing Algorithhms

Obsah

Obrázky

Tabulky

Kapitola 1

Úvod

S obrazovým materiálem se lidé setkávají každý den. Ať už se jedná o fotografie z telefonu, na sociálních sítích, o krátká videa, reklamy, nebo filmy v kině, všechny tyto oblasti mají něco společného. Je to komprese obrazu. Existuje mnoho metod ztrátové a bezeztrátové komprese obrazu a v této práci několik z nich popisuji.

Cílem této práce je ale i jejich implementace na zařízení s poměrně nízkým výkonem a tedy omezenými výpočetními možnostmi. Efektivní algoritmy jsou velmi důležité v dnešním světě, kdy každý z nás má kdykoliv po ruce mobilní telefon, který potřebuje na baterii vydržet co nejdéle.

Chytrými metodami a postupy lze obrazová data reprezentovat tak, že jejich datová velikost bude značně snížena, ačkoliv pro lidské oko budou mít obrázky stále vysokou kvalitu. Toho lze dosáhnout například tak, že určitým částem obrazové informace snížíme přesnost, protože zrovna na tyto informace není lidské oko příliš citlivé. Znalostí fungování lidského zrakového systému lze dosáhnout velmi přesného odhadu jaké informace jsou v obraze zrovna důležité a jaké můžeme bez většího problému vynechat.

V této práci tedy chci některé tyto metody a postupy ukázat, vysvětlit a naprogramovat v jazyce C++ při použití kamery na vývojové desce.

Kapitola 2

Teoretický popis

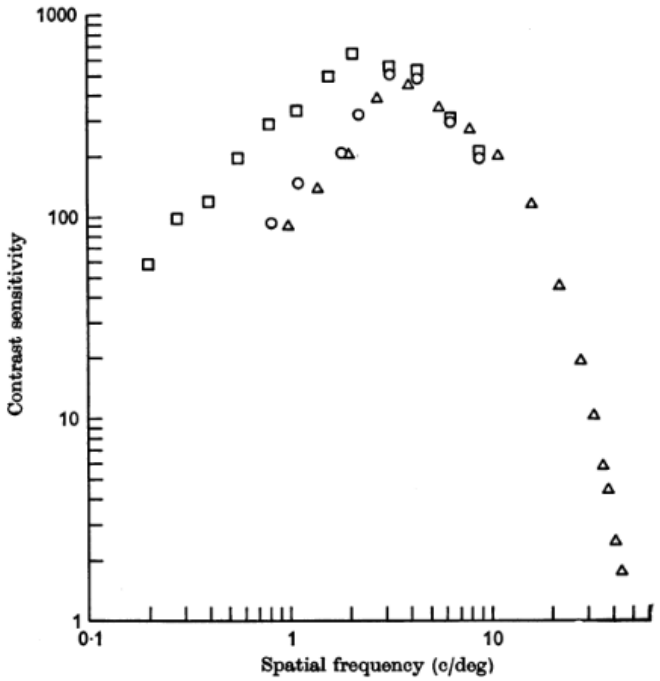
2.1 Lidský zrakový systém

Lidský zrak je nejdůležitější vjem, pomocí kterého člověk dostává přibližně 80 % informací o okolním světě [1]. Začátek cesty světla přicházejícího z okolního světa do zpracování lidským zrakem je v oku. Oko se skládá z několika částí, pro tuto práci je ovšem nejdůležítější část zvaná sítnice, ve které se nachází fotocitlivé buňky. Těchto fotocitlivých buněk má lidské oko hlavní dva druhy - tyčinky a čípky. Tyčinky umožňují černobílé vidění, reagují na světlo v celém rozsahu spektra viditelného světla. Mají vyšší citlivost na světlo než čípky, díky čemuž člověku umožňují vidění i za šera. Čípky v lidkém oku se dále dělí na 3 druhy. Liší se tím, v jaké vlnové délce absorbují světlo. L-čípky reagují na viditelné světlo s vyšší vlnovou délkou, tedy na červenou. Podobně M-čípky reagují nejvíce na zelenou barvu a S-čípky na viditelné světlo s nejkratší vlnovou délkou z těchto tří typů, tedy na modrou barvu. Lidské vidění je tzv. trichromatické, tedy všechny barvy, které člověk vnímá, se skládají ze třech základních barev - červené, zelené a modré. Toho se využívá jak u snímání obrazu, tak v reprodukci obrazu v displejích.

V sítnici se nachází přibližně 120 milionů tyčinek, ale pouze přibližně 7 milionů čípků. Z tohoto důvodu je lidský zrak citlivější na změnu intenzity světla než na změnu barvy světla. Tohoto faktu se využívá ve ztrátové kompresi. Ani všechny druhy čípků nejsou v sítnici zastoupeny rovnoměrně. Nejvíce je čípků reagujících na zelené světlo, nejméně naopak čípků reagujících na modré světlo. Lidský zrak je tedy citlivější na zelenou barvu než na červenou a modrou, i této informace lze využít při ztrátové kompresi obrazu.

Zatímco sítnice obsahuje přes 100 milionů fotoreceptorů, počet axonů přenášejících informace ze sítnice do mozku je pouze přibližně 1 milion. Přímo v sítnici tedy ještě dochází ke zpracování informací. Do mozku tedy neputují přímo informace o tom, na kterou konkrétní tyčinku nebo čípek dopadá jak silné světlo, ale zpracované informace vhodnější pro lidský zrak [2]. Z toho plyne, že při ztrátové kompresi obrazu lze vynechat určité informace, které lidský zrak nedokáže vnímat.

Z dalších pozorování vyplynulo, že citlivost na změnu kontrastu pro lidský zrak je závislá na prostorové frekvenci. V roce 1968 byly publikovány výsledky



Obrázek 2.1: Citlivost na změnu kontrastu v závislosti na prostorové frekvenci. Převzato z [3]. Různé symboly ukazují různé vzdálenosti od obrazovky a různé apertury.

měření autorů F. W. Campbella a J. G. Robsona, které ukazují, jakou citlivost na změnu kontrastu lidské oko má při různých prostotových frekvencích sinusové mřížky [3]. Výsledky jejich měření jsou vidět na obrázku 2.1. Na obrázku je vidět, že nejvyšší citlivost lidského zraku na změnu kontrastu nastává přibližně při frekvenci 2-4 změny na stupeň zorného pole. Pro nižší i vyšší prostorové frekvence tato citlivost klesá, čehož lze využít při ztrátové kompresi.

2.2 Multimédia

V oblasti využívání multimédií je v posledních letech vidět jednoznačný rostoucí trend. Mezi multimédia patří například obrázky, zvuky, videa, nebo počítačové hry. Video se skládá z po sobě jdoucích snímků, komprese obrazu je tedy důležitou součástí komprese videa. Počítačové hry obsahují například mnoho textur, které je kvůli úspoře místa taktéž vhodné komprimovat, i ve videoherním průmyslu tedy komprese obrazu najde využití.

Počet strávených hodin konzumací multimédií mezi dospělými obyvateli USA vzrostl z necelých 3 hodin denně v roce 2008 na více než 6 hodin denně v roce 2018 [4]. Celosvětový počet uživatelů sociálních sítí, jejichž součástí jsou multimédia, vzrostl z přibližně 1,72 miliard uživatelů v roce 2013 na přibližně 4,76 miliard v roce 2023 [5]. Na sociálních sítích se ve velké míře vyskytují obrázky a videa, což ukazuje, jak důležité je téma komprese obrázků. Jedna z

největších platforem pro streaming videa Netflix zaznamenala nárůst platících uživatelů z necelých 35 milionů v roce 2013 na téměř 270 milionů v roce 2024 [6]. Konzumace videoobsahu tedy v posledních letech roste a nic nenaznačuje, že by tento růst měl v nejbližší době přestat.

2.3 Digitální reprezentace obrazu

Digitální obraz je reprezentován dvojrozměrnou maticí bodů. Každý bod představuje jeden pixel, počet těchto bodů, tedy velikost matice, se nazývá rozlišení obrazu. Obraz může být složen z jednoho nebo více kanálů. Běžně se používá jeden kanál pro černobílý obraz. Pro barevný obraz se využívají 3 kanály, každý pro jednu ze základních barev - červená, zelená, modrá. Hodnota pixelu odpovídá intenzitě světla v místě pixelu. Pixely jsou digitálně reprezentovány určitým počtem bitů, tento počet se nazývá bitová hloubka. Nejčastěji používaný počet je 8 bitů pro každý pixel v každém kanálu. Pro tříkanálový barevný obraz se v tomto případě tedy používá 24 bitů pro každý pixel, což určuje počet rozdílných barev, které se mohou v takovém obrazu vyskytovat. Těchto barev je $2^{24} = 16\,777\,216$.

2.4 Barevné modely

Pro reprezentaci pixelu barevného obrazu je nutno použít nějaký barevný model. Barevný model specifikuje, jak digitálně uložit konkrétní barvu. Existuje několik různých barevných modelů, z nichž každý je vhodný použít v jiné situaci.

2.4.1 Model RGB

Jeden z nejjednodušších barevných modelů je RGB model. Využívá skutečnosti, že barvy viditelného spektra lze nakombinovat ze tří primárních barev. Tyto barvy jsou červená, zelená a modrá, což souvisí s barvami, které lidský zrak vnímá pomocí čípků, jak jsem psal v kapitole 2.1. Tento model je vhodný k použití u senzorů obrazu a u obrazovek. V senzorech digitálních fotoaparátů se používá filtr, který obsahuje mřížku složenou ze střídajících se červených, zelených a modrých polí. Jeden z často používaných filtrů je tzv. Bayerova maska, která využívá skutečnosti, že lidský zrak je nejcitlivější na zelenou barvu, zelená barva je tedy nejčastěji vzorkovaná oproti modré a červené [7]. Digitální fotoaparáty tedy snímají červenou, zelenou a modrou barvu. Hodnoty nasnímaných barev se následně digitálně uloží, případně déle zpracují.

Digitální obrazovky zpravidla obsahují subpixely, které vyzařují primární barvy - červenou, zelenou a modrou. Z těchto tří základních barev se pak skládají všechny barvy. Proto tedy po kompletním zpracování displej potřebuje pro zobrazení obrazu využívat RGB model.

2.4.2 Model CMY

Pro barevný tisk se většinou používá barevný model CMY, který se skládá z barev azurová, purpurová a žlutá (cyan, magenta, yellow). Azurová barva odráží zelenou a modrou barvu, červenou naopak pohlcuje, podobně purpurová pohlcuje zelenou barvu a žlutá pohlcuje modrou barvu. Z těchto tří barev tedy lze pomocí substraktivního míchání nakombinovat ostatní barvy viditelného spektra. Smícháním těchto tří barev nevznikne ale perfektní černá, často se proto v tiskárnách k těmto třem barvám taktéž přidává ještě černá barva, vznikne tedy model CMYK (písmeno K značí černou barvu, tzv. klíč) [8].

2.4.3 Model YUV

Tento model rozkládá barvu na složku luminance (Y) a dvě složky chrominance (U, V). Byl používaný v systému televizního vysílání PAL [9]. Model YUV vznikl jako řešení přidání barevných složek k dosud černobílému televiznímu vysílání. Černobílé televize využívaly pouze složku Y, která byla taktéž jediná vysílaná. Model YUV rozšířil signál tak, aby černobílé televize stále mohly fungovat beze změny, přičemž barevné televize přijímaly navíc složky U a V, díky kterým mohly rekonstruovat informaci i o barvě signálu [10].

Převod RGB-YUV

K převodu barvy mezi modelem RGB a YUV lze využít následující vztahy:

$$\begin{aligned} Y &= 0,299 \cdot R &&+0,587 \cdot G + 0,114 \cdot B \\ U &= -0,147 \cdot R &&-0,289 \cdot G + 0,436 \cdot B \\ V &= 0,615 \cdot R &&-0,515 \cdot G - 0,100 \cdot B \end{aligned} \tag{2.1}$$

Rozsah vstupních proměnných R, G, B je od 0 do 1. Rozsah výstupu Y je od 0 do 1, rozsah U je od $-0,436$ do $0,436$ a rozsah V je od $-0,615$ do $0,615$. Model YUV je výhodný pro zpracování obrazu, protože odděluje složku luminance, na kterou je lidksý zrak citlivější, od složek chrominance, na kterou je lidský zrak méně citlivý. Byl však vytvořen pro analogové vysílání, kvůli čemuž není ideální pro digitální zpracování obrazu. Například složky U a V by se musely škálovat, aby se jejich rozsah vešel do číselné digitální proměnné.

Pro inverzní transformaci zpět z modelu YUV do modelu RGB lze využít vztahy:

$$\begin{aligned} R &= 1 \cdot Y + 0 \cdot U &&+1,1398 \cdot V \\ G &= 1 \cdot Y - 0,3946 \cdot U &&-0,5805 \cdot V \\ B &= 1 \cdot Y + 2,032 \cdot U &&+0 \cdot V \end{aligned} \tag{2.2}$$

2.4.4 Model YCbCr

Podobně jako model YUV i tento model rozkládá barvu na složku luminance (Y) a na dvě chrominanční složky (Cb, Cr). Model YCbCr je ovšem přizpůsobený pro digitální zpracování obrazu. Je používán například ve formátu JPEG [11]. Často se plete s modelem YUV, avšak jedná se o dva rozdílné modely. Jeho transformace z modelu RGB je popsána rovnicemi:

$$\begin{aligned} Y &= 0,299 \cdot R &&+0,587 \cdot G + 0,114 \cdot B && \\ Cb &= -0,169 \cdot R &&-0,331 \cdot G + 0,5 \cdot B && +128 \\ Cr &= 0,5 \cdot R &&-0,419 \cdot G - 0,081 \cdot B && +128 \end{aligned} \tag{2.3}$$

Tato transformace je naškálována tak, aby všechny vstupy R, G, B mohly být v rozsahu od 0 do 255 a zároveň všechny výstupy byly taktéž v rozsahu od 0 do 255. Tedy pro 8bitové vstupy budou i výstupy 8bitové, to je vhodné právě pro digitální zpracování. Stejně jako YUV je tedy tato transformace vhodná pro zpracování obrazu, jelikož rozděluje obraz na složky luminance (intenzity), a chrominance. Ovšem díky pro digitální zpracování lépe navrženému škálování se tento barevný model skutečně využívá v digitálním zpracování obrazu.

Zpětná transformace do RGB modelu využívá následující rovnice:

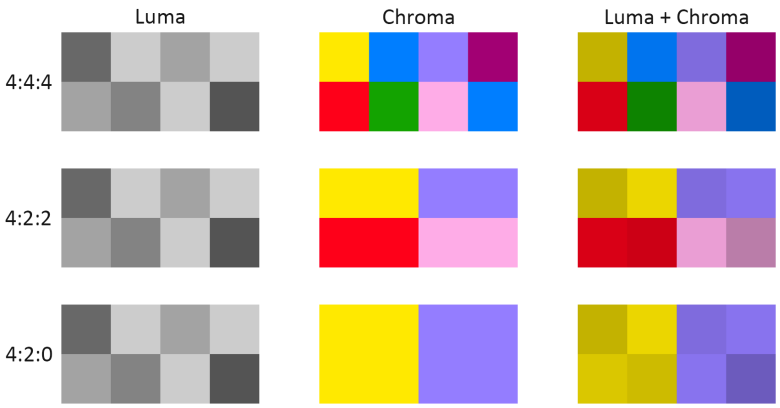
$$\begin{aligned} R &= 1 \cdot Y + 0 \cdot (Cb - 128) &&+1,402 \cdot (Cr - 128) \\ G &= 1 \cdot Y - 0,344 \cdot (Cb - 128) &&-0,714 \cdot (Cr - 128) \\ B &= 1 \cdot Y + 1,722 \cdot (Cb - 128) &&+0 \cdot (Cr - 128) \end{aligned} \tag{2.4}$$

2.4.5 Podvzorkování barvonosných složek

Jak jsem již psal v kapitole 2.1, lidský zrak je citlivější na změnu intenzity světla než na změnu jeho barvy. Tohoto faktu se využívá při ztrátové kompresi obrazu. Díky použití modelu YCbCr je totiž možné podvzorkovat barvonosné složky. Tedy luminanční složku Y ponecháme v plném rozlišení, ale chrominanční cložky Cb a Cr uchováme pouze v nižším rozlišení. Výsledkem bude úspora dat při nízké perceptuální ztrátě kvality obrazu. Při rekonstrukci obrazu do původního rozlišení a do modelu RGB budou využity Y koeficienty s plným rozlišením, ale koeficientů Cb a Cr bude méně, použijí se tedy vícekrát pro sousední pixely. Prakticky se používá několik způsobů podvzorkování barvonosných složek [12].

Vzorkování 4:4:4

Při vzorkování 4:4:4 nedochází k podvzorkování barvonosných složek. Zápis ve formátu $J : a : b$ lze interpretovat tak, že J určuje počet referenčních horizontálních vzorků, obvykle se používá 4. Druhé číslo a určuje, kolik barvonosných vzorků bude obsahovat první řádek pixelů (obvykle 4, 2 nebo 1), poslední číslo b určuje, kolik barvonosných vzorků se změní v přechodu mezi prvním a druhým řádkem (obvykle stejné jako a nebo 0). Podle tohoto zápisu



Obrázek 2.2: Ukázka principu podvzorkování barvonosných složek. Převzato z [12].

pro blok 4×2 luminančních vzorků budou v prvním řádku 4 chrominanční vzorky (chrominanční vzorek obsahuje dva komponenty - Cb a Cr - tedy pro jednu hodnotu luminance budou dvě hodnoty chrominance) a další 4 chrominanční vzorky ve druhém řádku. Celkem tedy 8 pixelů bude obsahovat 8 vzorků luminance a 16 vzorků chrominance, což odpovídá 24 bitům na pixel.

Vzorkování 4:2:2

Na každých 8 vzorků luminance se uloží pouze dva vzorky chrominance v prvním řádku a další dva v druhém řádku, celkem tedy 4 vzorky chrominance pro Cb a další 4 pro Cr. V součtu každých 8 pixelů bude obsahovat 8 vzorků luminance a 8 vzorků chrominance, to odpovídá 16 bitům na pixel. Při podvzorkování chrominance lze využít buď průměrování dvou sousedních vzorků, nebo vynechání každého druhého vzorku. V praxi se vzorky vynechávají, ztráta informace je vůči úspoře dat a výpočetního výkonu přijatelná.

Vzorkování 4:2:0

Vzorkování 4:2:0 je nějčastěji využívané podvzorkování barvonosných používané v multimédiích. Ke každým 8 pixelům se ukládá 8 vzorků luminance, ale pouhé 2 vzorky chrominance Cb a 2 vrozky Cr. Ze čtverce 2×2 pixelů se uchová pouze hodnota chrominance zpravidla levého horního pixelu. To má za následek úsporu 50 % prostoru oproti ukládání barvonosných složek v plném rozlišení. Tedy ukládat se bude pouze 12 bitů na pixel. Oproti černobílému obrazu, který by obsahoval pouze složku luminance Y s 8 bity na pixel, můžeme uložit barevný obraz s pouhými dalšími 4 bity na pixel, čímž výrazně zlepšíme požitek z obrazového vjemu. Snížení kvality vlivem podvzorkování barvonosných složek u multimédií není výrazně viditelné, obzvláště u fotografií z reálného světa a u vyšších rozlišení [12].

2.5 Ztrátová komprese

Kvůli neustálému trendu růstu multimédií je stále důležitější zefektivňovat způsoby ukládání a přenosu obrazu. Často používaná metoda je ztrátová komprese. Při ztrátové kompresi dochází ke ztrátě informace, tedy obraz po rekonstrukci nebude matematicky shodný s obrazem před kompresí. Existuje několik různých metod, jak ztrátové komprese dosáhnout, mnoho z nich využívá vlastností a nedokonalostí lidského zrakového systému. V této kapitole popíšu několik základních způsobů ztrátové komprese.

2.5.1 Podvzorkování

Nejjednodušší metoda ztrátové komprese je podvzorkování. Obraz lze jednoduše zkomprimovat zahozením některých vzorků, tedy snížením rozlišení. Tato metoda je velmi jednoduchá, ale zahozením vzorků snížíme kvalitu vjemu. Existují i jiná řešení, která sníží objem dat bez tak výrazného zhoršení vnímané kvality obrazu jako prostým snížením rozlišení.

Podvzorkování barev

V kapitole 2.4.5 jsem psal o podvzorkování barvonosných složek. Při této operaci se ponechává počet vzorků v luminanční složce bez změn, díky čemuž vnímaný pokles kvality pro lidského pozorovatele nebude tak výrazný jako při podvzorkování celého obrazu.

2.5.2 Skalární kvantizace

Chceme-li zachovat počet vzorků, lze provést kompresi na jiném místě. Omezme se nyní na jeden kanál, například luminenční, tedy na černobílou verzi obrazu. Každý pixel je kódován zpravidla pomocí 8 bitů, tedy počet rozdílných úrovní je $2^8 = 256$. Pokud ořízneme hodnotu každého pixelu například na 6 horních bitů, můžeme ukládat pouze 6 bitů pro každý pixel při určité ztrátě kvality. Nově se bude rozlišovat pouze $2^6 = 64$ různých barev. Problémem pak je, že příliš malý rozdíl mezi barvami, který by se v původním obraze projevil změnou spodních dvou bitů intenzity, se ztratí. To bude mít za výsledek poměrně viditelnou ztrátu kvality například u barevného přechodu ze světlejší barvy k tmavší.

2.5.3 Vektorová kvantizace

Zatímco skalární kvantizace snižuje počet bitů k reprezentaci každého pixelu, vektorová kvantizace pracuje vždy na blocích několika sousedních pixelů, zvaných vektory. Předem je určeno, kolik různých vektorů chceme využívat, z čehož plyne jednak kvalita zkomprimovaného obrazu a taktéž kompresní poměr. Pracuje se tedy na bloku pixelů, například 2×2. V jednom kanále budou 4 pixely obsahovat 32 bitů, celkový počet kombinací hodnot v těchto 4

pixelech bude tedy $2^{32} = 4\,294\,967\,296$. Můžeme si ale určit, že chceme použít pouze 1024 různých vektorů. Vhodným algoritmem pak lze nalézt těchto 1024 vektorů tak, aby se reprezentace celého obrazu za použití pouze těchto vektorů, lišila od původního obrazu co nejméně. Po kompresi pak pro každou čtveřici pixelů bude stačit uložit 10 bitů popisujících který z vektorů je použit namísto plných 32 bitů, které určovaly přesné hodnoty intenzit pixelů.

2.5.4 Zobecnění ztrátové komprese

Výše popsané metody fungují na podobném principu, kdy se zahazují data přímo v prostorové oblasti. Obecně úkol ztrátové komprese je zahodit data, která nenesou přílišnou relevanci. Relevance dat v obraze souvisí s tím, jak daná data vnímá lidský zrak. Se znalostí fungování lidského zrakového systému je tedy možné navrhnout ztrátové kompresní algoritmy tak, že jimi zahazovaná data budou pro zrak irelevantní, a tedy jejich zahozením se příliš nezhorší vnímaná kvalita obrazu.

2.5.5 Transformace do frekvenční oblasti

Lidský zrak je citlivý na světlo o různých prostorových frekvencích různě. Toho se využívá při ztrátové kompresi tak, že se obrázek nejdříve vhodnou transformací rozloží na jednotlivé frekvenční složky a na každou tuto frekvenční složku lze aplikovat nějaká forma komprese. Lze o tom taktéž přemýšlet tak, že transformací do frekvenční oblasti dojde vlastně ke koncentraci relevantní informace do spodní části spektra, jelikož právě na tu je lidský zrakový systém nejcitlivější.

2.5.6 Diskrétní Fourierova transformace

Fourierova analýza rozkládá signál v časové oblasti do sinusoid o různých frekvencích. Rozklad signálu do frekvenčních složek s různými amplitudami se nazývá spektrum [8]. Diskrétní Fourierova transformace je vhodná k použití v počítačích, jelikož se jedná o transformaci v diskrétních hodnotách vstupu a výstupu. Na vstupu se nachází reálný signál, na výstupu se objeví komplexní koeficienty, které určují amplitudu a fázi každé konkrétní sinusoidy, která se nachází ve spektru signálu. Vzorec pro výpočet koeficientů diskrétní Fourierovy transformace je následující:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-jnk\frac{2\pi}{N}}, k = 0, 1, ..., N-1 \tag{2.5}$$

Signál ve frekvenční oblasti je vhodnější pro některé operace. Například filtrace, která by v prostorové oblasti probíhala pomocí konvoluce, bude probíhat pomocí násobení ve frekvenční oblasti. Po zpracování lze signál převést z frekvenční oblasti zpět do prostorové zpětnou diskrétní Fourierovou transformací:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{jnk\frac{2\pi}{N}}, n = 0, 1, ..., N-1 \tag{2.6}$$

Povšimněme si, že vzorec pro zpětnou transformaci vlastně ukazuje, že signál je lineární kombinací komplexních exponenciál.

Jelikož digitální obraz je z podstaty dvojrozměrný signál, je vhodné definovat dvojrozměrnou diskrétní Fourierovu transformaci. Definována je vzorcem pro výpočet:

$$X(k, l) = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x(m, n) e^{-j\frac{2\pi}{N}(mk+nl)}, k, l = 0, 1, ..., N-1 \tag{2.7}$$

Kde $x(m, n)$ je hodnota pixelu obrazu, $X(k, l)$ je hodnota spektra, souřadnice m, n jsou souřadnice pixelů v prostorové oblasti, souřadnice k, l jsou frekvence ve dvou osách, N je velikost obrazu (uvažuje se čtvercový obraz, tedy velikost v obou osách je shodná).

Pro přechod z frekvenční oblasti zpět do prostorové oblasti se využijí tyto inverzní vztahy:

$$x(m, n) = \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} X(k, l) e^{j\frac{2\pi}{N}(mk+nl)}, m, n = 0, 1, ..., N-1 \tag{2.8}$$

2.5.7 Algoritmus FFT

Výpočetní složitost algoritmu pro diskrétní Fourierovu transformaci je $O(n^2)$. Jedním z nejdůležitějších algoritmů pro účely digitálního zpracování obrazu je algoritmus FFT, neboli rychlá Fourierova transformace, který dokáže spočíst diskrétní Fourierovu transformaci se složitostí $O(n \cdot \log(n))$ [8].

V algoritmu FFT se využívá symetrie takovým způsobem, že problém počítání diskrétní Fourierovy transformace N bodů se rozdělí na problém počítání lichých N/2 bodůl a sudých N/2 bodů. Tento proces lze rekurzivně opakovat, čímž rozdělíme problém na $\log_2(n) - 1$ stádií o $\frac{N}{2}$ komplexních součinech. Celkem je tedy složitost algoritmu odhadnuta jako $O(n \cdot \log(n))$. Aby byl možné výpočet opakovaně dělit na výpočet s polovičním počtem koeficientů, je nutné, aby počáteční délka zpracovávaného segmentu N byla mocnina 2. Pokud tomu tak není, vstupní data se doplní nulami do délky nejbližší vyšší mocniny 2. Díky urychlení výpočtu pomocí algoritmu FFT je toto doplnění stále rychlejší než počítat diskrétní Fourierovu transformaci přímo s náročností $O(n^2)$ [8].

2.5.8 Diskrétní kosinová transformace

Často používané kompresní formáty, např. JPEG, nevyužívají diskrétní Fourierovu transformaci, ale diskrétní kosinovou transformaci (DCT), která má

lepší vlastnosti co se týče koncentrace energie do koeficientů nižších frekvencí [13] [14]. Diskrétní kosinová transformace rozkládá signál do kosinusovek, výsledné amplitudy jsou čistě reálné. Existuje několik typů diskrétní kosinové transformace, avšak jeden z nejpoužívanějších je DCT-II. V mojí práci využívám modifikovanou verzi dvojrozměrné diskrétní kosinové transformace [15]. Vzorec pro její výpočet je následující:

$$\begin{gathered} B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos\left(\frac{\pi(2m+1)p}{2M}\right) \cos\left(\frac{\pi(2n+1)q}{2N}\right) \\ 0 \leq p \leq M-1, 0 \leq q \leq N-1 \\ \text{kde:} \\ \alpha_p = \frac{1}{\sqrt{M}} \text{ pro } p = 0, \alpha_p = \frac{\sqrt{2}}{\sqrt{M}} \text{ jinak} \\ \alpha_q = \frac{1}{\sqrt{N}} \text{ pro } q = 0, \alpha_q = \frac{\sqrt{2}}{\sqrt{N}} \text{ jinak} \end{gathered} \tag{2.9}$$

Dvojrozměrné pole A_{mn} obsahuje pixely obrázku, výsledné dvojrozměrné pole B_{pq} obsahuje frekvenční složky obrázku po kosinové transformaci. Nižší indexy (vlevo nahoře) obsahují nižší frekvence, v těch se zpravidla koncentruje největší množství informace pro běžné obrazy (např. fotky). Vyšší prostorové frekvence se v obrazech nevyskytují v takovém množství jako nižší, navíc lidské oko na vyšší prostorové frekvence není tolik citlivé jako na nižší.

Rozdělení na bloky

V praxi se nepoužívá diskrétní kosinová transformace na celý obraz, ale na bloky. Nejdříve se obraz rozdělí do stejně velkých bloků, typicky 8×8, a pak se na každý tento blok použije diskrétní kosinová transformace. To má za následek možnost zrychlení výpočtu transformace podobným principem jako tomu bylo u algoritmu FFT. Kromě toho je možné každý blok zpracovávat nezávisle, například v jiném procesorovém vlákně, a tím snížit celkovou dobu výpočtu.

Všimněme si, že první koeficient s frekvenčními indexy $p = 0$ a $q = 0$ je vlastně pouze naškálovaný aritmetický průměr všech pixelů obrazu. Pokud bychom chtěli rekonstruovat obraz s nižším rozlišením, konkrétně osminovém v každé ose (celkový počet pixelů by tedy byl $\frac{1}{64}$), mohli bychom zobrazit pouze první frekvenční koeficienty, tzv. DC hodnoty.

Zpětná kosinová transformace

Z frekvenční oblasti zpět do prostorové oblasti se lze transformovat pomocí následujících vztahů [16]:

$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos\left(\frac{\pi(2m+1)p}{2M}\right) \cos\left(\frac{\pi(2n+1)q}{2N}\right)$$
$$0 \leq m \leq M-1, 0 \leq n \leq N-1$$
$$\text{kde:} \tag{2.10}$$
$$\alpha_p = \frac{1}{\sqrt{M}} \text{ pro } p = 0, \alpha_p = \frac{\sqrt{2}}{\sqrt{M}} \text{ jinak}$$
$$\alpha_q = \frac{1}{\sqrt{N}} \text{ pro } q = 0, \alpha_q = \frac{\sqrt{2}}{\sqrt{N}} \text{ jinak}$$

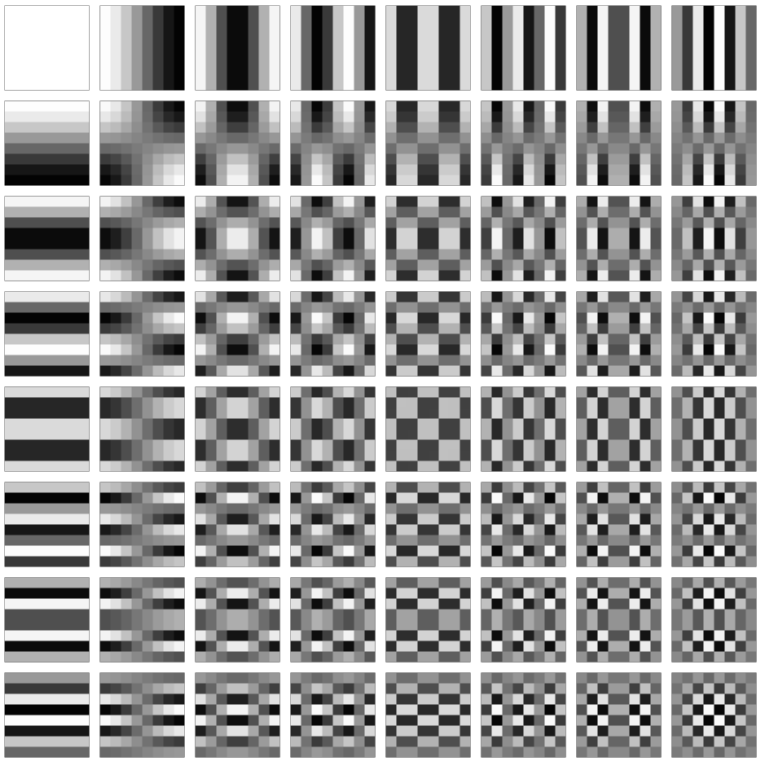
Ze vzorce je vidět, že zpětná transformace je velmi podobná dopředné, argumenty uvnitř funkce kosinus jsou stejné jako u dopředné transformace. U diskrétní fourierovy transformace měly argumenty komplexní exponenciály opačná znaménka pro dopřednou a zpětnou transformaci. Jelikož je kosinus sudá funkce, změna znaménka jeho argumentu nemá na hodnotu vliv.

Ukázka báze kosinové transformace

Pro velikost bloku 8×8 se pixely v bloku postupně porovnávají celkem se 64 různými bloky, které jsou složeny z kosinusovek o různých horizontálních a vertikálních frekvencích. Na obrázku 2.3 je vidět těchto 64 bloků, vlevo nahoře je blok s nulovými prostorovými frekvencemi v obou směrech, tedy blok odpovídající stejnosměrnému DC koeficientu. Směrem dolu a doleva rostou prostorové frekvence v obou směrech.

2.5.9 Kvantizační matice

Samotná diskrétní kosinová transformace není ztrátová, pouze koncentruje relevantní informace do několika prvních nízkofrekvenčních koeficientů. Ze znalosti citlivosti lidského zraku na vyšší prostorové frekvence lze ale odvodit, že frekvenční koeficienty odpovídající vyšším prostorovým frekvencím nebudou z hlediska zachování člověkem vnímané kvality obrazu po rekonstrukci tolik důležité. Z toho důvodu vznikla tzv. kvantizační matice. Kvantizační matice obsahuje tolik koeficientů, jako je velikost bloků transformace, v našem případě tedy 8×8. Po diskrétní kosinové transformaci dojde k vydělení každého frekvenčního koeficientu příslušným koeficientem z kvantizační matice. Výsledné číslo se zaokrouhlí na celočíselnou hodnotu, to je krok kvantizace [9]. Podobně, jako tomu bylo u skalární kvantizace, o které jsem psal v kapitole 2.5.2, dojde ke snížení přesnosti daného koeficientu. Avšak narozdíl od skalární kvantizace, která snižovala přesnost intenzity každého pixelu, kvantizací frekvenčních koeficientů dojde selektivně ke snížení přesnosti těch prostorových frekvencí, které lidský zrakový systém nedokáže s dostatečnou přesností vnímat. Dojde tedy k úspoře místa tam, kde bylo příliš irelevantních informací z hlediska obrazu a lidského zraku.



Obrázek 2.3: Ukázka báze diskrétní kosinové transformace o velikosti bloku 8×8. Public domain, převzato z [17].

Vzhledem k tomu, že fotografie většinou obsahují mnohem větší zastoupení nízkofrekvenčních složek než vysokofrekvenčních složek, koeficienty u vysokých frekvencí byly poměrně nízké už po diskrétní kosinové transformaci oproti koeficientům u nižších frekvencí. Kvantizační matice tyto koeficienty příslušné vyšším frekvencím ještě dále sníží, díky čemuž po zaokrouhlení na celé číslo vyjde většina koeficientů u vyšších frekvencí nulových.

Volba kvality

Koeficienty kvantizační matice přímo určují úroveň kvantizace jdnotlivých frekvenčních složek. Existuje mnoho kvantizačních matic, každá určuje jinou kvalitu zkomprimovaného obrazu. Kvantizační matice obsahující nižší čísla povede k vyšší kvalitě po kompresi, ale také k menšímu počtu nulových koeficientů a k horšímu kompresnímu poměru. Naopak kvantizační matice s vyššími koeficienty povede k odstranění více frekvenčních složek, počet nenulových koeficientů nesoucích informace o detailech obrazu bude nižší, čímž dojde ke zlepšení kompresního poměru ale za cenu snížení kvality rekonstruovaného obrazu. Specifikace formátu JPEG obsahuje různé kvantizační matice použité podle volby požadované kvality v rozsahu 0 až 100. Jednu konkrétní kvantizační matici lze vidět na obrázku 2.4. Koeficienty odpovídající vyšším prostorovým frekvencím (vpravo dole) jsou vyšší, což vede po vydělení

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

Obrázek 2.4: Konkrétní kvantizační matice používaná ve formátu JPEG. Převzato z [18].

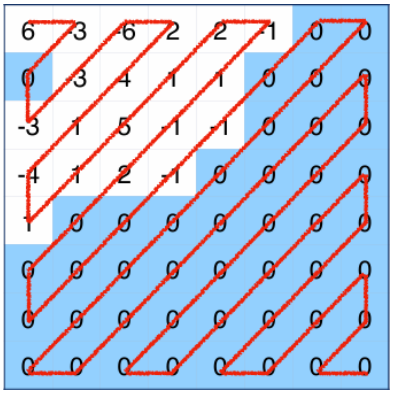
k menším číslům, a po zaokrouhlení k vyšší pravděpodobnosti výskytu nul.

2.5.10 Zig-Zag skenování

Po diskrétní kosinové transformaci a vydělením jednotlivých frekvenčních koeficientů prvky kvantizační matice máme pole kvantizovaných hodnot, v němž se pravděpodobně vyskytuje několik nulových prvků. Počet nulových prvků závisí na volbě kvality, resp. na volbě kvantizační matice. Nejvíce nulových prvků se bude s největší pravděpodobností vyskytovat u vyšších prostorových frekvencí, tedy vpravo dole. Abychom se mohli těchto nulových, pro obrazovou informaci tedy irelevantních, prvků zbavit, lze použít tzv. Zig-Zag skenování. Jedná se o způsob přeskládání prvků tak, aby po sobě následovaly seřazené podle prostorové frekvence vzestupně. Na začátku přeskládaného seznamu budou tedy koeficienty odpovídající nižší prostorové frekvenci a na konci budou prvky odpovídající nejvyšší prostorové frekvenci. To má za následek přesunutí většiny nulových prvků na konec seznamu. Takto uspořádaný seznam je výhodný, protože při zpracování můžeme zjistit, který prvek v seznamu je poslední nenulový, a všechny následující nulové prvky můžeme zahodit, tedy je dále nezpracovávat, neukládat ani nepřenášet. Zde tedy taktéž dochází ke (ztrátové) kompresi.

Na obrázku 2.5 je vidět postup Zig-Zag skenování, nulové koeficienty po kvantizaci jsou podbarveny modře. Začátek skenování je vpravo nahoře, je vidět že postupně červená čára prochází přes všechny nenulové koeficienty, až nakonec dorazí přibližně do půlky, odkud všechny následující koeficienty po cestě jsou už nulové. Při tomto průchodu je vhodné zapamatovat si index posledního nenulového koeficientu, aby bylo následně možné skončit po uložení všech nenulových koeficientů, a zbytek neukládat.

Při ukládání nenulových koeficientů je ovšem nutné vědět, kdy skončit. Jelikož se obraz skládá z mnoha bloků 8×8 pixelů, budou postupně muset být uloženy všechny nenulové koeficienty ze všech bloků do paměti za sebou. Při dekódování je ale nutné vědět, kde ještě pokračují koeficienty z aktuálního bloku a kde již začínají koeficienty bloku dalšího. Jedna z jednoduchých mož-



Obrázek 2.5: Ukázka postupu Zig-Zag skenování. Převzato z [18].

ností je při ukládání koeficientů použít jeden byte před prvním koeficientem jako označení počtu následujících koeficientů. Další jednoduchá možnost je vyhradit si jednu konkrétní hodnotu bytu jako tzv. stop byte, který se zařadí na konec za poslední koeficient bloku. Slouží tedy jako jakýsi oddělovač. Tato možnost má výhodu v tom, že každý blok končí stejnou konstantou, což se ukáže jako výhodné v další kapitole, kde řeším bezeztrátovou kompresi. Naopak ale tato možnost přináší i nevýhodu, a to sice skutečnost, že mnou zvolenou konstantu nesmí obsahovat žádný blok jakožto kvantizovaný koeficient. Jako tento stop byte jsem si zvolil hodnotu 120, to je hodnota, která se vejde do osmibitového celočíselného typu se znaménkem a zároveň není příliš malá, aby hrozilo velké riziko kolize s nějakým koeficientem. Frekvenční koeficienty jsou zpravidla malá čísla, obzvláště po vydělení kvantizační maticí. Je tedy velmi nepravděpodobné, že by se nějaký frekvenční koeficient trefil do tak vysoké hodnoty jako 120. Pokud by to však nastalo, lze problém vyřešit zcela jednoduše, namísto 120 uložíme například hodnotu 119, čímž sice zavedeme do obrazu další šum, avšak příliš slabý na to, aby hrál zásadní roli vzhledem k ostatním metodám ztrátové komprese.

2.6 Bezeztrátová komprese

Po průchodu algoritmem pro ztrátovou kompresi získáme reprezentaci původního obrazu. Tato reprezentace zabírá méně prostoru na disku a můžeme ji rovnou uložit do souboru či přenést po síti. Existuje ale ještě další metoda ke zlepšení kompresního poměru. Zatímco cílem ztrátové komprese je snížit obsah irelevantních dat, cílem bezeztrátové komprese je snížit redundanci v datech. V teorii informace definujeme pojem entropie, který udává mez bezeztrátové komprese, tedy kolik bitů stačí na úplný popis obsažené informace beze ztrát. Rozdíl mezi délkou vstupních dat a entropií vstupních dat se nazývá redundance, kterou se pomocí algoritmů bezeztrátové komprese snažíme minimalizovat. Data vystupující z algoritmu ztrátové komprese obsahují stále redundantní informace, které je možné pomocí algoritmů bezeztrátové komprese snížit. Většina nejpoužívanějších formátů pro kompresi obrazu, zvuku

i videa proto využívá kombinaci ztrátové a následně bezeztrátové komprese. Někdy se algoritmům bezeztrátové komprese říká entropické kódování.

2.6.1 Entropie

Entropie je definována jako míra náhodnosti či nejistoty [8]. Udává mimo jiné hranici bezeztrátové komprese, tedy víme, že jakákoliv metoda bezeztrátové komprese nemůže data validně zkomprimovat na menší prostor než udává entropie. Problém entropie je, že způsob jejího určení není jednoznačný. Dokážeme vypočítat odhad entropie pomocí vzorečku:

$$H = -\sum_{i} P_i \cdot \log_2 P_i \tag{2.11}$$

kde pravděpodobnost P_i spočteme jako počet bytů rovných hodnotě i vydělený celkovým počtem bytů.

Stále se ale jedná pouze o jeden odhad. Data můžeme například předem zpracovat bez ztráty informace, pro digitální obraz se často používá diferenciální kódování, při kterém se neukládají hodnoty intenzity pixelů, ale pouze rozdíly mezi intenzitami sousedních pixelů. Díky tomu, že rozdíly sousedních pixelů jsou obvykle nízké, lze tímto způsobem získat posloupnost, jejíž entropie odhadnutá předchozím vztahem bude nižší než odhad entropie původního obrazu.

Příklad nedokonalého odhadu entropie

Představme si situaci, kdy máme vektor dat a, tento vektor má délku 256 a obsahuje posloupnost po sobě jdoucích čísel od 0 do 255:

$$a = \{0, 1, 2, ..., 254, 255\}$$

Každá hodnota je ve vektoru obsažena právě jednou, pravděpodobnost výskytu každé hodnoty je uniformní:

$$P_i = \frac{1}{256}, i = 0, 1, ..., 255$$

Lze tedy vypočítat entropii podle vzorce výše:

$$H = -\sum_{i} P_i \cdot \log_2 P_i = -\sum_{i=0}^{255} P_i \cdot \log_2 P_i = -\sum_{i=0}^{255} \frac{1}{256} \cdot \log_2 \frac{1}{256} =$$
$$= -256 \cdot \frac{1}{256} \cdot \log_2 \frac{1}{256} = -256 \cdot \frac{1}{256} \cdot (-8) = 8$$

Vypočtená entropie je 8 bitů, to by mělo znamenat, že na reprezentaci každého prvku vektoru a by bylo potřeba 8 bitů. Data z vektoru a lze ovšem zpracovat například zavedením vektoru b, který bude taktéž dlouhý 256 prvků, ale jeho první element bude shodný s prvním elementem vektoru a a každý další element bude obsahovat rozdíl odpovídajícího elementu

z vektoru a a předchozího elementu vektoru a. Vektor b tedy bude vypadat následovně:

$$b = \{0, 1, 1, 1, ..., 1, 1\}$$

Entropie vektoru b se v tomto případě spočítá jako:

$$H = -\sum_i P_i \cdot \log_2 P_i = -(P_0 \cdot \log_2 P_0 + P_1 \cdot \log_2 P_1) =$$
$$= -\left(\frac{1}{256} \cdot \log_2 \frac{1}{256} + \frac{255}{256} \cdot \log_2 \frac{255}{256}\right) \approx 0,0369$$

Tedy jiná reprezentace stejných dat má najednou entropii přibližně 0,0369 bitů na symbol. Navíc je třeba znát způsob, jakým se z vektoru b dostat zpět na údaje ve vektoru a.

2.6.2 Huffmanovo kódování

Jedna z nejpoužívanějších metod pro bezeztrátovou kompresi je Huffmanovo kódování. Běžná data se skládají z bytů, každý byte obsahuje 8 bitů. Základní myšlenka Huffmanova kódování je ta, že znaky s vyšší pravděpodobností výskytu zakódujeme menším počtem bitů a znaky s nižší pravděpodobností zakódujeme vyšším počtem bitů [8]. Jelikož často zastoupené bity budou zakódovány menším počtem bitů než 8, bude pak i průměrná délka jednoho znaku nižší než 8, tím pádem i délka celých zakódovaných dat bude nižší než délka původních dat, dojde tedy ke kompresi.

Kroky pro výpočet Huffmanova algoritmu jsou následující:

1. Je třeba nalézt pravděpodobnosti výskytu každého znaku. Lze spočítat jako relativní četnost daného znaku. Znaky s nulovou pravděpodobností není třeba zahrnovat.

2. Nalezněme dva znaky s nejnižšími pravděpodobnostmi výskytu, tyto znaky slučme do kombinovaného znaku.

3. Vytvořme binární strom, jeden znak bude v listu nalevo, druhý v listu napravo.

4. Nově vzniklý kombinovaný znak zařaďme zpět do seznamu znaků místo dvou, ze kterých vznikl. Jeho pravděpodobnost bude součet pravděpodobností znaků, ze kterých vznikl.

5. Opakujme kroky 2-4 dokud nebude strom zcela sestaven a obsahovat každou hodnotu znaku s nenulovou pravděpodobností.

6. Hodnota Huffmanova slova pro každý znak lze přečíst ze struktury binárního stromu. Při cestě po stromu doleva si zapamatujme 0, při cestě doprava 1, cesta ke každému znaku nám dá konkrétní posloupnost bitů jako Huffmanovo slovo.

Při kódování samostatných znaků je Huffmanův kód optimální, tedy dosahuje nejlepšího kompresního poměru [8]. Huffmanovo kódování se dokáže přiblížit spočítané entropii, tedy průměrný počet bitů pro zakódování jednoho znaku pomocí Huffmanova kódování se bude blížit entropii shora. Ve speciálním případě, kde jsou pravděpodobnosti všech slov nějaké záporné mocniny 2, se Huffmanovo kódování dostane přesně na spočtenou entropii. To se stane, protože díky mocninám dvojky budou optimální délky slov celočíselné, tedy Huffmanovým kódováním dosažitelné.

Pro fungování algoritmu popsaného výše není důležité použít skutečnou pravděpodobnost výskytů jednotlivých znaků, stačí nám zachovat pořadí při seřazení od nejméně častého znaku po nejvíce častý. Místo pravděpodobnosti či relativní četnosti lze tedy použít absolutní četnost, neboli počet, kolikrát se daný znak ve vstupních datech objevuje.

Příklad Huffmanova kódování

Mějme sekvenci 4 znaků s následujícími parametry:

Znak	Počet výskytů	Pravděpodobnost výskytu
A	12	$\frac{12}{32}$
B	3	$\frac{3}{32}$
C	7	$\frac{7}{32}$
D	10	$\frac{10}{32}$

Tabulka 2.1: Příklad znaků pro Huffmanův kód

Můžeme spočítat jejich entropii:

$$H = -\sum_{i} P_i \cdot \log_2 P_i = -\sum_{i=0}^{3} P_i \cdot \log_2 P_i = -\left(\frac{12}{32} \cdot \log_2 \frac{12}{32} + \frac{3}{32} \cdot \log_2 \frac{3}{32} + \right.$$
$$\left. + \frac{7}{32} \cdot \log_2 \frac{7}{32} + \frac{10}{32} \cdot \log_2 \frac{10}{32}\right) \approx 1,8548$$

Našli jsme tedy mez bezeztrátové komprese, nyní postupujme podle Huffmanova algoritmu:

Znak	Počet výskytů	Pravděpodobnost výskytu
A	12	$\frac{12}{32}$
D	10	$\frac{10}{32}$
C	7	$\frac{7}{32}$
B	3	$\frac{3}{32}$

Tabulka 2.2: Seřazená tabulka znaků pro Huffmanův kód

Seřazením řádků podle počtu výskytů je vhodné pro hledání nejméně zastoupených znaků. Jsou to znaky C a B, tyto znaky tedy sloučíme do jednoho kombinovaného znaku a jejich pravděpodobnosti výskytu sečteme:

Znak	Počet výskytů	Pravděpodobnost výskytu
A	12	$\frac{12}{32}$
D	10	$\frac{10}{32}$
C,B	10	$\frac{10}{32}$

Tabulka 2.3: Tabulka znaků, znaky C a B se zkombinovaly

V dalším kroce pokračujeme a stejným způsobem sloučíme znak D s již sloučeným znakem C, B:

Znak	Počet výskytů	Pravděpodobnost výskytu
D,C,B	20	$\frac{20}{32}$
A	12	$\frac{12}{32}$

Tabulka 2.4: Tabulka znaků, znaky D, C a B se zkombinovaly

Kombinovaný znak D, C, B má nyní pravděpodobnost výskytu vyšší než znak A, byl tedy v seznamu zařazen nad znak A. V posledním kroce dojde ke sloučení všech znaků a tedy i ke konci algoritmu:

Znak	Počet výskytů	Pravděpodobnost výskytu
D,C,B,A	32	$\frac{32}{32}$

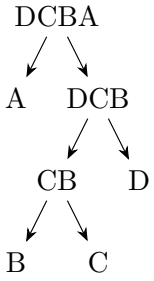
Tabulka 2.5: Tabulka znaků, všechny znaky se zkombinovaly

Pokud nyní začneme od konce, postupně můžeme zařazovat jednotlivé znaky do binárního stromu tak, že při rozdělení kombinovaného znaku přiřadíme jeho méně pravděpodobné součásti 0 a jeho více pravděpodobné součásti 1. Výsledný sestavený strom tedy je vidět na obrázku 2.6. Šipka vlevo značí bit 0, šipka vpravo značí bit 1. Zkonstruovaný Huffmanův kód pak je vidět v tabulce 2.6.

Nyní spočteme průměrnou délku kódu v bitech. Jedná se o vážený průměr délky kódu s vahami pravděpodobností výskytu jednotlivých znaků:

$$L = \sum_i P_i \cdot l_i = \frac{12}{32} \cdot 1 + \frac{3}{32} \cdot 3 + \frac{7}{32} \cdot 3 + \frac{10}{32} \cdot 2 = \frac{62}{32} = 1,9375$$

Porovnáme-li odhad entropie s průměrnou délkou, zjistíme, že Huffmanova průměrná délka je přibližně o 0,08 bitu na znak vyšší. Pokud bychom ale kódovali původní data bez komprese, na 4 různé znaky bychom potřebovali



Obrázek 2.6: Ukázka výsledného binárního stromu.

Znak	Počet výskytů	Pravděpodobnost výskytu	Kód	Délka kódu
A	12	$\frac{12}{32}$	0	1
B	3	$\frac{3}{32}$	100	3
C	7	$\frac{7}{32}$	101	3
D	10	$\frac{10}{32}$	11	2

Tabulka 2.6: Tabulka znaků a vygenerovaných kódů

$\log_2 4 = 2$ bity, s Huffmanovou kompresí bychom tedy na každém znaku ušetřili přibližně 0,06 bitů. V původním textu o délce 32 bychom namísto 64 bitů použili po kompresi 62 bitů bez ztráty informace. V tomto konkrétním příkladě se tedy nejedná o nijak výraznou kompresi, nicméně stejný algoritmus lze použít na jakákoliv data a v závislosti na pravděpodobnostním rozdělení četnosti jednotlivých znaků může dojít k výrazné kompresi.

Komprese

Samotný proces komprese pak probíhá tak, že pro každý znak ze vstupních dat si nalezneme v tabulce příslušné několikabitové kódové slovo, které přidáme k výstupní bitové sekvenci. Výstupem Huffmanova kódování je tedy sekvence bitů. Prakticky se do souboru ukládají ale byty, tedy osmice bitů, výstupní sekvence bitů z Huffmanova kódování ovšem nemusí mít délku násobkem osmi. Proto je nutné bitovou sekvenci doplnit výplňovými daty až do délky, která je dělitelná osmi. Tato výplň mohou být například nulové bity. Vyplněním bitů, které nejsou míněny jako kódová slova ale může vzniknout problém. Při dekódování se totiž může tato výplň dekódovat chybně jako nějaké kódové slovo, je tedy nutné vědět, kde s dekódováním skončit. Nejjednodušší řešení je před sekvenci bitů zarovnanou na byte do souboru uložit ještě celočíselný typ určující celkový počet bitů, které mají být dekódovány. Díky tomu bude dekodér vědět, kdy přesně končí validní data, a tedy přestane tam, kde by hrozilo již dekódování nesprávného kódového slova z výplně na konci.

Aby bylo možné zkomprimovaná data znovu dekomprimovat, je nutné, aby dekodér věděl, kterým znakům náleží která kódová slova. Dekodér tedy

potřebuje znát slovník. Při ukládání či přenosu dat zkomprimovaných Huffmanovým kódováním je nutné v nějaké formě uložit či přenést i tento slovník. Existují chytré způsoby, jak zajistit, aby tento slovník byl co nejmenší, ale dostatečný. O těchto způsobech píšu v jedné z následujících kapitol 2.6.3.

Jednoznačnost dekódování

Při dekompresi, tedy dekódování Huffmanova kódu, dochází ke hledání kódového slova v bitové sekvenci. Nalezení kódového slova musí být jednoznačné. Postup hledání je od prvního bitu k dalším bitům, dokud nenajdeme posloupnost bitů, která se přesně shoduje s nějakým kódovým slovek ve slovníku. Po nalezení kódového slova na výstup přidáme dekódovaný znak a hledáme ve vstupní bitové sekvenci další kódové slovo začínající ná bitu, který následuje po aktuálním nalezeném kódovém slovu. Z tohoto postupu vyplývá, že pro jednoznačné dekódování je nutné, aby žádné kódové slovo neobsahovalo na svém začátku stejné bity, jako celé jiné slovo. Žádné kódové slovo tedy neobsahuje jiné kódové slovo jako svůj prefix, takovým kódům se říká tzv. prefixové kódy [13]. Kdyby některé kódové slovo obsahovalo jiné jako svůj prefix, při dekódování by takové slovo nebylo možné dekódovat, jelikož by dekodér vždy našel dříve slovo, které bylo prefixem. Zbytek bitů, které zůstaly po odebrání bitů ze začátku kódového slova, bude následně znovu prohledáno pro nalezení dalšího kódového slova, čímž může nastat nalezení jednoho či několika dalších nesprávných slov. Původní znak, jehož kódové slovo jako prefix obsahovalo jiné kódové slovo, nebude nikdy možné tímto postupem dekódovat.

Algoritmus hledání Huffmanových slov popsaný na začátku kapitoly 2.6.2 generuje prefixové kódy, ty jsou jednoznačně dekódovatelné. Díky tomu je Huffmanova komprese bezeztrátová, při znalosti slovníku je vždy možné všechna slova validně dekódovat a přesně rekonstruovat původní nekomprimovaná data.

Míra komprese

Jak jsem psal v předchozích kapitolách, hranice bezeztrátové komprese je entropie. Některá data lze tedy zkomprimovat lépe, některá data nelze zkomprimovat téměř vůbec. Záleží na jejich entropii, která souvisí s pravděpodobnostním rozložením jednotlivých hodnot, to lze vyjádřit histogramem. Nejhorší případ z hlediska bezeztrátové komprese jsou data, která mají uniformní rozdělení, tedy histogram je konstantní čára, pravděpodobnost výskytu každého znaku je stejná. Pro příklad spočteme entropii dat, která obsahují uniformně rozložených 256 různých hodnot:

$$H = -\sum_{i} P_i \cdot \log_2 P_i = -\sum_{i=0}^{255} \frac{1}{256} \cdot \log_2 \frac{1}{256} =$$
$$= -256 \cdot \frac{1}{256} \cdot \log_2 \frac{1}{256} = -256 \cdot \frac{1}{256} \cdot (-8) = 8$$

Entropie těchto dat vyšla 8 bitů na znak, jenže pro celá čísla v rozsahu od 0 do 255, která byla v původních datech, bylo stejně potřeba 8 bitů na znak.

Huffmanova komprese by v tomto případě nevedla k žádnému snížení datové velikosti. Naopak by kvůli přitomnosti slovníku délku dat ještě zvýšila.

Data, která již byla zkomprimována pomocí některé z metod pro bezeztrátovou kompresi již mají poměrně vysokou entropii, tedy rozdělení dat bude velmi blízko uniformnímu rozdělení. Je to důsledek odstranění redundance z komprimovaných dat. Kvůli tomu ovšem nedává smysl snažit se již zkomprimovaná data znovu komprimovat, výsledek pravděpodobně nebude příliš dobrý, může dojít namísto komprese k expanzi, jelikož kompresní formáty mívají režii (např. uložená Huffmanova tabulka).

Ideální případ pro bezeztrátovou kompresi je ten, kdy se pravděpodobnosti výskytů jednotlivých znaků co nejvíce liší. Pokud budou data obsahovat z 90 % jen několik málo znaků, naopak většina znaků bude v datech obsaženo pouze málokrát, spočtená entropie bude výrazně nižší oproti předchozímu případu s uniformním rozdělením pravděpodobností výskytu znaků. Spočtěme si entropii dat, která obsahují 9 znaků, každý s pravděpodobností výskytu 10 %, a zbylých 247 znaků, dohromady s pravděpodobností výskytu 10 %:

$$P_i = \begin{cases} \frac{1}{10}, & \text{pro } i = 0..8 \\ \frac{1}{2470}, & \text{pro } i = 9..255 \end{cases}$$

$$H = -\sum_i P_i \cdot \log_2 P_i = -\left(\sum_{i=0}^{8} \frac{1}{10} \cdot \log_2 \frac{1}{10} + \sum_{i=9}^{255} \frac{1}{2470} \cdot \log_2 \frac{1}{2470}\right) =$$

$$= -(9 \cdot \frac{1}{10} \cdot \log_2 \frac{1}{10} + 247 \cdot \frac{1}{2470} \log_2 \frac{1}{2470}) \approx 4,1168$$

Pro vyjádření dat s tímto rozdělením by tedy stačilo v ideálním případě přibližně 4,12 bitů na znak, tedy téměř polovina bitů je redundantní a lze beze ztráty odstranit. Huffmanovým kódováním bychom nedosáhli sice přesně entropie, ale průměrná délka by stále byla výrazně méně než 8 bitů na znak. Dokonce platí, že Huffmanův kód dosahuje průměrného počtu bitů na znak omezeného shora hodnotou $H + 1$ [13].

V kapitole 2.5.9 jsem popisoval, jakým způsobem se při ztrátové kompresi využívá kvantizační matice. Frekvenční koeficienty po vydělení kvantizační maticí budou poměrně malá čísla, většina z nich bude často blízko 0. To znamená, že histogram nabývaných hodnot frekvenčních koeficientů bude mít vyšší pravděpodobnosti výskytu u hodnot blízkých nule, což povede k poměrně nízké entropii. Díky tomuto rozdělení pravděpodobností výskytu bude bezeztrátová komprese fungovat dobře.

Kraftova nerovnost

Každý prefixový kód musí splňovat Kraftovu nerovnost [13]:

$$\sum_{k=1}^{N} 2^{-l_k} \leq 1 \tag{2.12}$$

kde N je počet slov kódu, l_k je délka k-tého slova kódu v bitech.

Platí, že prefixový kód lze vytvořit pokaždé, když slova kódu splňují Kraftovu nerovnost. To znamená mimo jiné, že pokud budeme mít seznam délek kódových slov splňující Kraftovu nerovnost, lze z něj vygenerovat prefixový kód.

Maximální délka Huffmanova slova

Při konstrukci Huffmanova kódu pro data, která mají 8 bitů na znak, vznikne až 256 různých kódových slov. Pokud histogram výskytu různých znaků nebude uniformní, budou mít častější znaky Huffmanovy kódy kratší než 8 bitů. Z Kraftovy nerovnosti plyne, že jiná kódová slova Huffmanova kódu musí být delší než 8 bitů. Chceme-li komprimovat data Huffmanovým kódováním, potřebujeme odhadnout maximální délku v bitech, které může nějaké kódové slovo nabývat. Díky tomuto hornímu odhadu můžeme zvolit datový typ o velikosti, která bude jistě dostačovat pro vyjádření každého kódového slova.

Z algoritmu vytváření optimálních Huffmanových kódů lze pozorovat několik skutečností:

1. Znaky, které se ve zdrojových datech objevují častěji, mají přiřazena kratší kódová slova než znaky, které se objevují méně často.
   Kdyby tomu tak nebylo, nejednalo by se o optimální kód, jelikož prohozením kódových slov mezi těmito dvěma znaky by vedlo k nižší průměrné délce na znak.

2. Dva znaky, které se ve zdrojových datech vyskytují nejméně často (tedy první a druhý nejméně častý) mají přiřazena stejně dlouhá kódová slova.
   Kdyby tomu tak nebylo, musel by nejméně častý znak mít nejdelší kódové slovo, ale protože kódová slova vzniklá Huffmanovým algoritmem jsou prefixová, toto nejdelší kódové slovo by nesmělo jako prefix obsahovat žádné jiné kódové slovo. Tedy by šlo zachovat jednoznačnou dekódovatelnost i při odstranění tolika bitů kódového slova nejméně častého znaku, kolik má bitů navíc oproti kódovému slovu druhého nejméně častého znaku. To by vedlo k nižší průměrné délce na znak, což je kontradikce s předpokladem, že Huffmanův kód je optimální.

Z těchto pozorování lze odvodit způsob, jakým dosáhnout nejdelšího kódového slova. Huffmanův algoritmus popsaný na začátku kapitoly 2.6.2 udává, že v každém kroku dojde ke kombinaci dvou nejméně častých znaků. Pokud na začátku kroku máme v seznamu n různých (kombinovaných) znaků, po kombinaci dvou nejméně častých znaků budeme mít v seznamu $n-1$ různých (kombinovaných) znaků. Algoritmus končí po kompletním sestavení binárního stromu, tedy když v seznamu bude již poslední kombinovaný znak obsahující všechny znaky abecedy vstupních dat. Počet kroků (či průchodů) Huffmanova algoritmu je tedy počet znaků vstupní abecedy bez jedné.

Při kombinaci dvou znaků (které samy mohou být kombinací znaků z předchozích kroků) se jednomu z nich přidává do kódového slova bit 0 a druhému se přidává bit 1, čímž se zajistí, že budou mít odlišná kódová slova. Tedy při

kombinaci dvou znaků se všem znakům, které jsou součástí nově vzniklého kombinovaného znaku, zvýší délka kódového slova o 1 bit. Chceme-li zjistit, jaké nejdelší kódové slovo může existovat, je nutné uvažovat nejhorší možnost pravděpodobnostního rozdělení vstupních dat, při kterém bude existovat nejdelší možné kódové slovo. Nejdelší kódové slovo by muselo vzniknout tak, že by v každém kroce byl jeden konkrétní znak součástí právě kombinovaných znaků. Pak by tomuto znaku příslušné kódové slovo bylo prodlouženo v každém kroce algoritmu, a vzhledem k tomu, že počet kroků Huffmanova algoritmu je znám, je znám i horní odhad délky kódového slova v nejhorším možném případě, a to je počet znaků vstupní abecedy bez jedné.

Má-li vstupní abeceda 256 různých hodnot (tedy uvyžujeme osmibitová vstupní data, která chceme komprimovat), maximální možná délka jednoho kódového slova bude 255. Binární strom takového kódu bude nevyvážený, kódová slova budou vypadat tak, že bude existovat právě jedno kódové slovo délky 1, právě jedno délky 2, tak to bude pokračovat až do délky 254, a nakonec budou existovat dvě slova délky 255, budou to dvě slova s nejnižší pravděpodobností výskytu.

Aby se však toto stalo, musel by jedén konkrétní znak být součástí kombinace v každém kroce. V každém kroce se kombinují dva nejméně časté (kombinované) znaky. Pro nejvyšší délku jednoho kódového slova je tedy nutné, aby po kombinaci nově vzniklý kombinovaný znak byl vždy poslední nebo předposlední v seznamu kombinovaných znaků seřazeného podle četností výskytu. Jedině tak lze zajistit, že i v dalším kroce bude tento znak znovu zkombinován do dalšího. Pokud má ale toto být pravda, musí četnosti výskytu jednotlivých znaků růst tak, aby po každé další kombinaci zůstal původní nejméně častý znak nakombinovaný s ostatními málo častými znaky, stále na posledním či předposledním místě v seřazeném seznamu. Četnosti výskytu znaků tedy budou růst, dává tedy smysl zkusit odvodit, jak rychle budou muset růst.

Chceme najít nejnižší počet znaků vstupních dat, aby bylo umožněno Huffmanovým algoritmem vytvořit kódové slovo nejvyšší délky. Určeme, že nejméně častý znak, který bude mít nejdelší kódové slovo, bude mít četnost 1. Více než 1 nemá smysl, protože chceme najít nejnižší délku vstupních dat. V prvním kroce Huffmanova algoritmu se tento nejméně častý znak zkombinuje s druhým nejméně častým znakem. Jelikož chceme mít nejkratší vstup, nemá smysl dát tomuto znaku vyšší počet výskytů než 1. Oba nejméně časté znaky tedy budou mít četnost 1. Po jejich zkombinování budou mít kombinovanou četnost 2. Ve druhém kroce je potřeba, aby byl v předchozím kroce nakombinovaný znak nejhůře druhý nejméně častý. Toho lze dosáhnout tak, že jeden znak bude mít libovolnou nižší četnost, při podmínce minima to bude 1, ale všechny ostatní znaky již nesmí mít nižší četnost než 2, protože jinak by se zkombinovaly jiné dva znaky, které měly oba nižší četnost. Ostatní znaky mají tedy všechny četnost 2 nebo výše. Nově zkombinovaný znak již obsahuje 3 nejméně časté znaky a jeho kombinovaná četnost je 3. V dalším kroce je nejméně četný znak o četnosti 2, to je minimální četnost za podmínky z minulého kroku, náš nakombinovaný znak má četnost 3, aby byl stále na

předposledním místě seřazeného seznamu, všechny ostatní znaky musí mít četnost alespoň 3. Po kombinaci bude mít kombinovaný znak obsahující celkem 4 znaky již kombinovanou četnost 5. V dalším kroce budou opět nejméně četné znaky mít četnost 3 a 5, ostatní tedy minimálně 5. Vznikne kombinovaný znak s kombinovanou četností 8. Stejným způsobem lze pokračovat dále.

Je vidět, že po prvních třech nejnižších četnostech rovných 1 je každá další četnost rovna součtu dvou předchozích menších četností. Četnosti tedy až na první jedničku, která je tam navíc, kopírují Fibonacciho posloupnost. Níže je uvedená tabulka s několika prvními vypočtenými četnostmi:

k	c	k	c	k	c
1	1	21	6765	41	102334155
2	1	22	10946	42	165580141
3	1	23	17711	43	267914296
4	2	24	28657	44	433494437
5	3	25	46368	45	701408733
6	5	26	75025	46	1134903170
7	8	27	121393	47	1836311903
8	13	28	196418	48	2971215073
9	21	29	317811	49	4807526976
10	34	30	514229	50	7778742049
11	55	31	832040	51	12586269025
12	89	32	1346269	52	20365011074
13	144	33	2178309	53	32951280099
14	233	34	3524578	54	53316291173
15	377	35	5702887	55	86267571272
16	610	36	9227465	56	139583862445
17	987	37	14930352	57	225851433717
18	1597	38	24157817	58	365435296162
19	2584	39	39088169	59	591286729879
20	4181	40	63245986	60	956722026041

Tabulka 2.7: Tabulka s četnostmi splňujícími podmínky pro vygenerování nejdelšího kódového slova

kde k značí počet uvažovaných nejméně častých znaků, c značí četnost nejčastějšího z nich.

Aby bylo možné dané četnosti c dosáhnout, vstupní data by musela mít délku alespoň c. Z toho plyne, že pokud vstupní data nebudou mít alespoň c znaků, nebude možné žádným způsobem dosáhnout takového rozdělení pravděpodobnosti výskytu znaků, aby mohlo po průchodu Huffmanovým algoritmem vzniknout slovo délky k. Zvolíme-li tedy datový typ, který má 64 bitů, určitě nevznikne slovo Huffmanova kódu delší než 64 bitů, minimálně dokud délka vstupních dat nepřesáhne jednotky terabajtů.

2.6.3 Kanonické Huffmanovo kódování

Při ukládání bitové sekvence získané z Huffmanova kódování je do výsledného souboru nutno uložit také informace o slovníku, jaká kódová slova patří ke kterým znakům. Bez této informace by se bitová sekvence nedala přeložit zpět na původní znaky. Existuje mnoho způsobů, jak tento slovník ukládat, jeden poměrně efektivní způsob je použít tzv. kanonické Huffmanovo kódování. Do Huffmanova algoritmu popsaného v kapitole 2.6.2 vstupují četnosti jednotlivých znaků ze vstupních dat. Kanonické Huffmanovo kódování spočívá v tom, že samotný Huffmanův algoritmus vygeneruje pouze délky kódových slov. Jak jsem psal v kapitole 2.6.2, prefixový kód lze vytvořit pokaždé, když délky slov splňují Kraftovu nerovnost. Délky slov, které získáme z Huffmanova algoritmu Kraftovu nerovnost splňují. Existuje mnoho způsobů, jak z těchto délek vygenerovat konkrétní kódová slova tak, aby kód byl prefixový. Jeden ze způsobů je následující:

1. Seřaďme si znaky vzestupně podle délky jejich kódového slova. Pokud mají znaky stejnou délku kódového slova, seřadíme vzestupně je podle jejich číselné hodnoty.

2. Prvnímu znaku v seznamu bude přiřazeno kódové slovo složeno ze všech nulových bitů o správné délce. Délku kódového slova známe z Huffmanova algoritmu.

3. Další znak v pořadí dostane kódové slovo jako předchozí znak, s přičtenou hodnotou 1.

4. Pokud má mít tento znak delší kódové slovo než měl předchozí znak, dodá se tolik nulových bitů na konec kódového slova, dokud nebude délka kódového slova správná.

Tento postup funguje jednoznačně, pro stejný seznam vstupních délek kódových slov vždy vytvoří prefixový kód se stejnými kódovými slovy. Díky tomuto postupu stačí místo celého slovníku ukládat pouze seznam délek kódových slov. Při dekódování se z tohoto seznamu kanonicky vytvoří slovník, díky kterému budou validně dekódována všechna kódová slova zpět na nekomprimovaný text.

2.7 Formát pro kompresi obrazu

Pro praktickou kompresi obrazu je nutné definovat způsob, jakým se dané kompresní algoritmy budou používat. K tomu slouží datový formát. Mezi běžně používané formáty komprese obrazu patří třeba JPEG nebo PNG. Pro potřeby této práce jsem si vytvořil vlastní formát, který implementuje podobné kompresní algoritmy jako formát JPEG. Můj kompresní formát je složený z následujících kroků:

1. Převod hodnot pixelů z barevného modelu RGB to YCbCr (rozdělení na kanály luminance a chrominance), podvzorkování chrominančních kanálů
2. Rozdělení každého kanálu na bloky 8×8
3. Spočtení diskrétní kosinové transformace pro každý blok a každý kanál zvlášť
4. Vydělení hodnot v každém bloku každého kanálu kvantizační maticí
5. Zig-Zag skenování a sloučení bloků do jednoho toku dat pro každý kanál
6. Sloučení toků dat všech kanálů do jednoho toku dat
7. Vytvoření slovníku Huffmanova kódu
8. Komprese pomocí Huffmanova kódování

Podobný postup, jen s opačným pořadím, jsem implementoval pro dekompresi obrazu z mého formátu a vykreslení na obrazovku počítače.

2.7.1 Formát sji1

Postupně pro účely této práce vznikaly verze mého formátu. Jako příponu jsem zvolil $sji1$, sj jsou moje iniciály, i značí image a 1 značí první verzi mého formátu.

Formát této verze zpracovával pouze černobílou verzi obrazu. Způsob převodu barevného obrazu do černobílého byl velmi primitivní, vypočítal se jako vážený průměr jednotlivých složek R, G a B s tím, že zelená měla dvojnásobnou váhu než červená a modrá. To mělo výhodu jednoduchosti, jelikož takový průměr lze spočíst pouze pomocí operací sčítání a bitového posuvu. Po blocích byla dále provedena diskrétní kosinová transformace, Zig-Zag skenování, odstranění nul na konci a následná komprese Huffmanovým kódováním.

Výsledný soubor má formát:

- 256 B - délky jednotlivých kódových slov, postačující k rekonstrukci celého slovníku, detaily v kapitole 2.6.3
- 6 B - 48bitové celé číslo určující počet bitů, které se mají dekódovat
- Zbytek souboru - bitová sekvence zakódovaná Huffmanovým kódováním, po dekompresi obsahuje:
  - Pole bajtů obsahující frekvenční koeficienty diskrétní kosinové transformace, přeskládané Zig-Zag skenováním, odstraněné nuly na konci, seskládané postupně z bloků 8×8 pixelů

2.7.2 Formát sji2

Tento formát má od předchozího navíc podporu barevných obrazů. Dochází v něm tedy ke konverzi obrazu z barevného modelu RGB do modelu YCbCr. Chrominanční složky jsou vzorkovány v souladu s podvzorkováním 4:2:0, tedy se čtvrtinovým rozlišením.

Výsledný soubor má formát:

- 256 B - délky jednotlivých kódových slov, postačující k rekonstrukci celého slovníku
- 6 B - 48bitové celé číslo určující počet bitů, které se mají dekódovat
- Zbytek souboru - bitová sekvence zakódovaná Huffmanovým kódováním, po dekompresi obsahuje:
  - 4 B - 32bitové celé číslo určující počet DCT koeficientů kanálu Y
  - 4 B - 32bitové celé číslo určující počet DCT koeficientů kanálu Cb
  - 4 B - 32bitové celé číslo určující počet DCT koeficientů kanálu Cr
  - Zbytek části - DCT koeficienty, složené za sebou z kanálů Y, Cb, Cr, každý kanál obsahuje:
    - Pole bajtů obsahující frekvenční koeficienty diskrétní kosinové transformace, přeskládané Zig-Zag skenováním, odstraněné nuly na konci, seskládané postupně z bloků 8×8 pixelů

2.7.3 Formát sji3

Třetí verze mého formátu obsahuje proměnlivou velikost obrazu. Předchozí verze pracovaly natvrdo s rozlišením 2560×1920, tato verze dokáže pracovat s rozlišením, které je násobkem 8 a nižší než $2^{19} \times 2^{19} = 524\,288 \times 524\,288$. Dále má v sobě zakomponovanou podporu několika různých kvantizačních matic, tedy několik presetů kvality.

Výsledný soubor má formát:

- 256 B - délky jednotlivých kódových slov, postačující k rekonstrukci celého slovníku
- 6 B - 48bitové celé číslo určující počet bitů, které se mají dekódovat
- Zbytek souboru - bitová sekvence zakódovaná Huffmanovým kódováním, po dekompresi obsahuje:
  - 2 B - 16bitové celé číslo určující počet 8pixelových bloků směrem do šířky
  - 2 B - 16bitové celé číslo určující počet 8pixelových bloků směrem do výšky
  - 1 B - index kvantizační matice

- 4 B - 32bitové celé číslo určující počet DCT koeficientů kanálu Y
- 4 B - 32bitové celé číslo určující počet DCT koeficientů kanálu Cb
- 4 B - 32bitové celé číslo určující počet DCT koeficientů kanálu Cr
- Zbytek části - DCT koeficienty, složené za sebou z kanálů Y, Cb, Cr, každý kanál obsahuje:
  - Pole bajtů obsahující frekvenční koeficienty diskrétní kosinové transformace, přeskládané Zig-Zag skenováním, odstraněné nuly na konci, seskládané postupně z bloků 8×8 pixelů

2.8 Popis vývojové desky

Celá práce je cílená na fungování na vývojové desce Basler daA2500-60mc-SD820-DB8. Jedná se o vývojovou desku se čtyřjádrovým ARM procesorem Qualcomm Snapdragon 820E, se 4 GB paměti RAM a s 50GB vestavěným flash úložištěm. Součástí vývojové desky je 5 megapixelová kamera připojená pomocí rozhraní MIPI CSI-2. Deska má dále k dispozici konektor pro napájení, gigabitový Ethernet, dva USB porty, jeden slot PCIe generace 2 a HDMI konektor. Na desce je i několik GPIO portů.

2.8.1 Kamera

Počítač pomocí rozhraní BCON přes MIPI CSI-2 přímo komunikuje s kamerou, která je součástí vývojového kitu. Kamera používá CMOS senzor ON Semiconductor AR0521 s rozlišením 2560×1920 bodů. Senzor dokáže snímat 60 snímků za sekundu, má rolující závěrku, doba expozice expozice je automatická s možností ručního nastavení pomocí API. Ke kameře je dodáván objektiv Evetar Lens N118B05518W F1.8 f5.5mm 1/1.8"

2.8.2 Softwarové vybavení

Operační systém na vývojovém počítači je od výrobce předinstalovaný Debian buster/sid. K práci s kamerou se využívá pylon API, které umožňuje snadné ovládání několika typů kamer, je kompatibilní s různými operačními systémy a s různými programovacími jakzyky. Na systému je dále nainstalován framework Qt se svým vývojovým prostředím. Framework Qt a jazyk C++ jsem si taktéž vybral pro implementaci vlastních programů v této práci.

2.9 TCP

Jedna část mého programu obsahuje přenos zkomprimovaných dat pomocí síťového protokolu TCP. Jedná se o jednoduchou ukázku možného využití vývojové desky jako přes internet přístupné kamery s využitím mého kompresního algoritmu. Transmission Control Protokol (TCP) se využívá pro komunikaci mezi dvěma zařízeními po počítačové síti. Ke komunikaci se

využívá model klient-server, jako server slouží vývojová deska, na které běží soketový server poslouchající příchozí spojení na konkrétním portu. Jako klient slouží druhý program, běžící například na osobním počítači, který se připojí pomocí soketu k serveru, odkud si přečte data. Přes TCP v této práci přenáším komprimovaná data v mém vlastním formátu. Na straně klienta tedy dochází k dekompresi a rekonstrukci obrazu, který je následně zobrazen na obrazovce.

Kapitola 3

Praktická část

3.1 Získání obrazových dat

K získávání dat z kamery je využita knihovna Pylon. Pro kameru existuje v této knihovně třída `Pylon::CInstantCamera`. Před samotným snímání dat je nutno inicializovat instanci této třídy. Konkrétní instance bude inicializována pro kameru s komunikací pomocí protokolu BCON, což je speciální protokol pro komuniakci s kamerou od firmy Basler, tedy výrobce vývojové desky. Po inicializaci kamery se začne snímat obraz pomocí metody `StartGrabbing()`, která je ve třídě `Pylon::CInstantCamera`. Způsobů snímání obrazů je několik, jeden ze zajímavých je `GrabStrategy_OneByOne`, který si sejmuté snímky ukládá do bufferu, z něhož lze pak snímek jeden po druhém číst a dále zpracovat. Další zajímavý je `GrabStrategy_LatestImageOnly`, který poskytne ke zpracování pouze poslední sejmutý snímek, pokud si některé snímky nevyzvedneme, budou zahozeny a nahrazeny novým snímkem. Tento způsob snímání používám v této práci.

Po sejmutí snímku lze tento snímek získat pomocí metody `RetrieveResult()`, která vrátí data sejmutého snímku do proměnné typu `Pylon::CGrabResultPtr`. Tento typ je smart pointer, který obsahuje jednak informace o snímku jako je jeho rozlišení, barevný model nebo bitová hloubka a jednak samotná data sejmutých pixelů. Pomocí třídy `Pylon::CImageFormatConverter` lze data z této proměnné převést do proměnné typu `Pylon::CPylonImage`, se kterou se dále lépe pracuje. Proměnná typu `Pylon::CPylonImage` obsahuje buffer typu `unsigned char*`, v němž jsou k dispozici RGB hodnoty jednotlivých pixelů.

3.2 Konverze do YCbCr

Jak jsem psal v kapitole 2.4.4, barevný model YCbCr je vhodný pro ztrátovou kompresi. V kódu provádím konverzi z RGB modelu do YCbCr ve stejném kroce jako podvzorkování chromatických kanálu Cb a Cr. Zjednodušená ukázka kódu je vidět zde:

```
for(int r=0; r<h; r++){
    for(int c=0; c<w; c++){
        int i_y = w*r + c;
        samples_Y[i_y] = 0.299 * data_src[3*i_y+0] + 0.587 *
↪   data_src[3*i_y+1] + 0.114 * data_src[3*i_y+2];
        if(r%2 == 0 && c%2 == 0){
            int i_c = w/2 * r/2 + c/2;
            samples_Cb[i_c] = (128 + -0.169 * data_src[3*i_y+0]
↪   - 0.331 * data_src[3*i_y+1] + 0.5 * data_src[3*i_y+2]);
            samples_Cr[i_c] = (128 +  0.5 * data_src[3*i_y+0] -
↪   0.419 * data_src[3*i_y+1] - 0.081 * data_src[3*i_y+2]);
        }
    }
}
```

Všechna pole `samples_Y,Cb,Cr` jsou datového typu `unsigned char*`. Pole pro složku Y má stejnou velikost jako obrázek, zbylá dvě pole pro složku Cb a Cr mají díky podvzorkované chromatičnosti velikost pouze čtvrtinovou.

3.3 Implementace DCT

Implementace diskrétní kosinové transformace se v mém programu nachází ve funkci `dct_full_image()`. Tato funkce má následující parametry:

- `const unsigned char* frame_data` - vstupní pixely jednoho kanálu
- `const int w` - šířka obrazu
- `const int h` - výška obrazu
- `signed char* output_bytestream` - předem alokované pole, kam bude uložen výstup
- `size_t* output_length` - číselný typ, do kterého bude uložena délka výstupu v bajtech

Do funkce přichází hodnoty pixelů jednoho kanálu o bitové hloubce 8 bitů na pixel. Uvnitř této funkce dochází k rozdělení obrazu do bloků o velikosti 8×8 pixelů. Pro každý blok je postupně vypočtena diskrétní kosinová transformace. Nejjednodušší možnost výpočtu DCT je přesně podle definice. Jako první jsem tedy vyzkoušel implementovat diskrétní kosinovou transformaci podle vzorečku v kapitole 2.5.8. Kód byl velmi jednoduchý, jeho ještě jednodušší verzi, kde jsem vynechal některé konstanty, lze vidět zde:

```
void compute_DCT(const unsigned char* A, float* B, int N){
    for(int p=0; p<N; p++){
        for(int q=0; q<N; q++){
            float res = 0;
```

```
5               for(int m=0; m<N; m++){
6                   for(int n=0; n<N; n++){
7                       float cos_m_arg = (3.14159265*(2*m+1)*p) /
    ↪   (2.0*N);
8                       float cos_n_arg = (3.14159265*(2*n+1)*q) /
    ↪   (2.0*N);
9                       res += (float)(A[m*N+n]) *
    ↪   std::cos(cos_m_arg) * std::cos(cos_n_arg);
10                  }
11              }
12              B[p*N+q] = res;
13          }
14      }
15  }
```

Náročnost takového algoritmu je $O(N^2)$, kde N ale je $8 \cdot 8 = 64$.

Všiml jsem si ale jistých symetrií, které se ve výpočtu objevují. Zaměřme se například na rovnici pro výpočet koeficientu s frekvenčními indexy $p = 3, q = 4$. Při výpočtu všech argumentů prvního kosinu a jejich funkčních hodnot nalezneme tyto (funkční hodnoty jsou zaokrouhleny na 3 desetinná místa):

x_1	$\frac{3\pi}{16}$	$\frac{9\pi}{16}$	$\frac{15\pi}{16}$	$\frac{21\pi}{16}$	$\frac{27\pi}{16}$	$\frac{33\pi}{16}$	$\frac{39\pi}{16}$	$\frac{45\pi}{16}$
$\cos(x_1)$	0.831	-0.195	-0.981	-0.556	0.556	0.981	0.195	-0.831

Tabulka 3.1: Tabulka nabývaných hodnot kosinu

V případě druhého kosinu budou vypadat hodnoty následovně:

x_2	$\frac{\pi}{4}$	$\frac{3\pi}{4}$	$\frac{5\pi}{4}$	$\frac{7\pi}{4}$	$\frac{9\pi}{4}$	$\frac{11\pi}{4}$	$\frac{13\pi}{4}$	$\frac{15\pi}{4}$
$\cos(x_2)$	0.707	-0.707	-0.707	0.707	0.707	-0.707	-0.707	0.707

Tabulka 3.2: Tabulka nabývaných hodnot kosinu

Ve funkčních hodnotách je vidět jakási symetrie. V rovnici dochází k násobení těchto dvou hodnot kosínů a pak k násobení s intenzitou pixelu. Vynásobíme-li všechny dvojice funkčních hodnot kosinů, vyjde nám ovšem pouze těchto osm unikátních možností:

-0.694	-0.588	-0.393	-0.138	0.138	0.393	0.588	0.694

Tabulka 3.3: Tabulka nabývaných hodnot po vynásobení výsledků obou kosinů

V rovnici pro výpočet DCT koeficientu se tedy nachází součet 64 členů, avšak v každém členu se objevuje násobení hodnoty pixelu jedním koeficientem z 8 možných. Navíc půlka těchto koeficientů je pouze záporná hodnota jiných.

Je možné tyto rovnice projít a v těch, ve kterých se objevuje násobení stejným či opačným koeficientem, tento společný koeficient vytknout před závorku. Díky tomu se výrazně sníží počet operací násobení, jejichž výpočet na procesoru trvá poměrně dlouho. Ve výsledné rovnici se tedy nejdříve sečtou (a odečtou) hodnoty pixelů, které patří ke stejnému (či opačnému) koeficientu, a následně se pouze tento součet jednou vynásobí společným koeficientem. Výsledek rovnice se nezmění, ale výpočet bude mnohem rychlejší.

V tomto případě konkrétně pro rovnici pro výpočet frekvenčního koeficientu s indexy $p = 3, q = 4$ vyšlo, že po vytknutí zbydou pouze 4 různé koeficienty, tedy ze 64 násobení zbyla pouze 4 násobení. Těchto rovnic je ale 64 a pro každou vyjde jiný počet unikátních koeficientů. Proto jsem si napsal program v jazyce MATLAB, který hrubou silou pro každou rovnici najde počet unikátních koeficientů a provede vytknutí. Průměrný počet unikátních koeficientů je 6. Namísto 64 násobení v každé ze 64 rovnic, stačí v každé rovnici násobit pouze průměrně 6krát. Tento výsledek je příjemný, neboť ukazuje zjednodušení z náročnosti na počet násobení $O(N^2)$, kde $N = 64$, na náročnost vzhledem k počtu násobení $O(N \cdot \log_2 N)$. Dvojkový logaritmus 64 je 6, což je přesně průměrný počet násobení v každé ze 64 rovnic.

Nyní tedy pro výpočet diskrétní kosinové transformace lze použít 64 rovnic, které mají každá průměrně 6 součinů. Před součinem je nutno sečíst všechny hodnoty pixelů, které byly v předchozím kroce sloučeny po vytknutí společného koeficientu. Všiml jsem si ale, že některé stejné součty pixelů se objevují ve více rovnicích. Například součet těchto čtyř pixelů:

```
1  A[8*3+3] - A[8*3+4] - A[8*4+3] + A[8*4+4]
```

se objevuje ve čtyřech různých rovnicích. Není nutné ho tedy počítat čtyřikrát na různých místech, lze ho spočítat jednou, uložit si výsledek, a ten pak pouze dosadit do všech rovnic, které tento součet obsahují. Upravil jsem svůj kód v jazyce MATLAB tak, aby našel všechna místa, kde se používá stejný výsledek součtu jako v jiné rovnici, a místo 384 součtů (o mnoha operandech) stačilo spočítat pouze 156 unikátních součtů.

3.3.1 Výsledné zrychlení výpočtu DCT

Rozlišení snímku z kamery je 2560×1920, to znamená celkem 76 800 bloků velikosti 8×8 pro kanál luminance. Když jsem zkoušel celý snímek nechat projít prvotní nijak neoptimalizovanou metodou výpočtu DCT, doba zpracování byla přibližně 20 sekund. Po výše zmíněných optimalizacích doba zpracování stejně velkého snímku klesla na přibližně 700 ms. Tyto optimalizace tedy vedly k drastickému snížení doby výpočtů přibližně o 96,5 %.

3.3.2 Kvantizační matice

Po průchodu diskrétní kosinovou transformací dochází k vydělení jednotlivých frekvenčních koeficientů odpovídajícími koeficienty kvantizační matice. Po

tomto dělení dochází k přetypování z datového typu `float` na `signed char`, čímž dojde k zaokrouhlení, malá čísla, kterých mezi frekvenčními koeficienty po vydělení kvantizační maticí bylo hodně, se vynulují. Ostatní čísla se přiblíží hodnotami k nule, čímž dojde ke snížení entropie dat, tedy ke zlepšení možnosti bezeztrátové komprese.

3.3.3 Zig-Zag

Poslední částí funkce pro výpočet diskrétní kosinové transformace je Zig-Zag skenování. Prvním průchodem jsou data překopírovaná již s novým pořadím do pomocného pole `out_zigzag` a zároveň se do pomocné proměnné ukládá poslední index, při kterém byla uložena nenulová hodnota. Ve druhém průchodu se již do výstupního pole `output_bytestream` překopírují pouze koeficienty od začátku až do posledního nenulového, za které se vloží stop byte s konstantní hodnotou 127.

3.3.4 Inverzní DCT

Podobným způsobem, jakým byl urychlen výpočet dopředné diskrétní kosinové transformace, lze urychlit i zpětnou diskrétní kosinovou transformaci. Využil jsem opět svého MATLAB skriptu, který jsem upravil pro optimalizaci inverzní transformace. Výsledky sice nebyly tak dobré, jako u dopředné transformace, ale stále ke zrychlení výpočtu došlo. Konkrétně každá ze 64 rovnic obsahuje průměrně 36 násobení a počet unikátních součtů se podařilo snížit z 2304 na 386.

3.4 Huffmanova komprese

Rozhodl jsem se, že Huffmanovu kompresi implementuji způsobem popsaným v kapitole 2.6.3. Postupoval jsem tedy tak, že jsem si nejdřív spočítal absolutní četnosti všech možných znaků ve vstupních datech (histogram). To lze udělat jednoduchým kouskem kódu:

```
1 size_t hist[256] = {0};
2 for(size_t i=0; i<output_length; i++){
3     hist[(unsigned char)output_bytestream[i]]++;
4 }
```

Takto spočítaný histogram předávám funkci `huffman_generate_word_lengths()`, která z histogramu vygeneruje pole délek kódových slov Huffmanova kódu. Funkce uvnitř funguje podle Huffmanova algoritmu. Udržuje si pole, které v sobě má uloženo, jaké znaky byly zkombinovány a jakou mají kombinovanou četnost. Ve funkci probíhá jedna smyčka, uvnitř které se vždy najdou dva znaky (samostatné či kombinované), které mají nejnižší četnosti. Tyto dva znaky se zkombinují do jednoho, nově vzniklému kombinovanému znaku se připíše součet četností a poté se k délkám všech znaků, které jsou

obsaženy v novém kombinovaném znaku, přičte 1. Nevzniká zde žádný binární strom, ani se neurčují konkrétní kódová slova. Výstupem po zkombinování všech znaků do jednoho velkého kombinovaného znaku je pole, které každému z 256 znaků přiřazuje délku kódového slova tak, že výsledný kód je optimální.

3.4.1 Vytvoření kódových slov

Nyní jsou k dispozici délky kódových slov. Tyto délky kódových slov se taktéž vyskytují v uloženém souboru či v přes síť přenášeném toku dat. Jednoznačný způsob, jak z délek kódových slov vygenerovat samotná kódová slova je implementován ve funkci `huffman_generate_codebook()`. Tato funkce si uvnitř seřadí znaky podle délek jejich kódových slov. Nad seřazeným polem již lze jednoznačným postupem generovat prefixová kódová slova. Jak bylo popisováno v kapitole 2.6.2, délka kódového slova 64 bitů stačí určitě na jakýkoliv vstupní text kratší než jednotky terabajtů. Rozhodl jsem se tedy, že kódová slova budu reprezentovat 64bitovým datovým typem `std::uint64_t`. Postup generování kódových slov je následující.

Jako první jsem si vytvořil pomocnou pracovní proměnnou, která obsahuje vždy aktuální kódové slovo `std::uint64_t cur_codeword = 0`. Inicializována byla na nulu, jelikož nejkratší kódové slovo bude obsahovat právě samé nulové bity. Nyní ve funkci probíhá hlavní cyklus, který postupně prochází všechny znaky od nejkratšího kódového slova po nejdelší. U každého průběhu dojde k inkrementaci pomocné proměnné `cur_codeword++`, druhá část kontroluje, zda aktuální kódové slovo má být delší než kódové slovo z předchozího průběhu cyklu. Pokud ano, doplní se aktuální kódové slovo nulami zprava, což je implementováno pomocí bitového posuvu `cur_codeword = cur_codeword << (cur_len-last_len)`. Jako poslední dojde k uložení aktuálního kódového slova `cur_codeword` na odpovídající index výstupního pole.

Tímto způsobem tedy dojde k vygenerování prefixového kódu z pouhé znalosti délek kódových slov. Využito toho bude stejným způsobem jak při kódování, tak při dekódování.

3.4.2 Komprese

K dispozici jsou nyní všechny potřebné informace k samotné kompresi dat. K tomu slouží funkce `huffman_compress()`, do které předáme jako parametr data, která chceme zakódovat, jejich délku, délky kódových slov a samotná kódová slova. Uvnitř funkce funguje velmi jednoduše. Pro každý znak (bajt) na vstupu si najde bitovou posloupnost kódového slova, tu pak přidá na konec výsledné bitové posloupnosti. Pro možnost dekomprese tato funkce ukládá na prvních 256 bajtů výstupní sekvence délky kódových slov, z nichž lze slovník sestavit, na dalších 6 bajtů ukládá jako 48bitové číslo počet bitů, které lze validně dekódovat. Prakticky se tím vytvoří limit zakódování maximálně 2^45 bajtů, to je ale přes 35 terabajtů, což v této práci nikdy jeden snímek nemůže přesáhnout. Od dalšího bajtu až do konce výstupu je bitová sekvence obsahující kódová slova, tedy komprimovaná data.

3.4.3 Dekomprese

Dekomprese probíhá ve funkci `huffman_decompress()` tak, že z prvních 256 bajtů získám délky kódových slov. Z těch jednoznačně vygeneruji slovník funkcí `huffman_generate_codebook()`. V následujících 6 bajtech si přečtu, kolik bitů mám celkově dekódovat. Následně přichází hlavní cyklus, ve kterém se opakovaně snažíme najít kódové slovo, které je na začátku aktuálně zpracovávaných dat. Pokud žádné takové slovo nenajdeme, do pomocné proměnné s aktuálně zpracovávanými daty načteme další bit ze vstupní bitové sekvence a hledáme znovu. Tímto způsobem eventuelně dojde k nalezení správného kódového slova na začátku vstupní bitové sekvence. Znak odpovídající kódovému slovu tedy přepíšeme na výstup a posuneme aktuálně zpracovávaná data o tolik bitů, kolik je dlouhé právě nalezené kódové slovo. Počet takto dekódovaných bitů si musíme ukládat do pomocné proměnné. Jakmile tento počet dosáhne známého počtu zakódovaných bitů, víme, že došlo již k dekódování veškerých dat, a cyklus je ukončen. Pokud nenastalo k chybě při přenosu či ukládání a načítání dat, pomocí této funkce byla dekódována původní data bez jakýchkoliv ztrát.

3.5 Skládání do formátu

Po konverzi do barevného modelu YCbCr máme obraz rozdělený na tři kanály. Každý samostatně projde rozdělením na bloky, diskrétní kosinovou transformací, kvantizací pomocí kvantizační matice, Zig-Zag skenováním, zahozením nul na konci a složením zpět z bloků do jednoho toku dat. Máme pak tedy tři toky dat, které musí být možné oddělit při dekódování. Z toho důvodu před tyto tři bloky předtím, než projdou bezeztrátovou kompresí, přidám ještě tři čtyřbajtová čísla, která určují kolik bajtů odpovídá které složce.

Dále je nutné do výsledného souboru uložit rozlišení obrázku, podle toho se totiž skládají za sebe bloky po 8×8 pixelech. Nakonec je potřeba do souboru uložit i informaci o použité kvantizační matici, jelikož při dekódování musí být použita stejná, jinak bychom nedekódovali správné frekvenční koeficienty.

Všechna tato metadata potřebná pro zpětnou rekonstrukci obrázku vkládám do bajtového toku ještě před tím, než celý projde algoritmem bezeztrátové komprese. Je to výhodné, protože mimo bezeztrátovou kompresi by v ideálním případě nemělo být nic, v reálném případě pouze to, co je nutné znát pro dekompresi. Tedy forma slovníku kódových slov. Vše ostatní projde algoritmem bezeztrátové komprese, čímž dojde k minimalizaci režie.

3.5.1 Časová náročnost jednotlivých bloků

Jednotlivé podúkoly komprese a dekomprese trvají různě dlouho. Analýzou dob trvání jednotlivých částí kompresního a dekompresního algoritmu můžeme získat vhled do toho, kde má smysl nejvíce optimalizovat. Z několika zpracovaných snímků jsem odpozoroval přibližně takovéto doby trvání:

- DCT pro 3 složky (Y, Cb, Cr): 875 ms
- Huffmanovo kódování: 55 ms
- I/O zápis a čtení na disk: 4 ms
- Dekomprese Huffmanova kódu: 302 ms
- Inverzní DCT pro 3 složky: 1322 ms
- Rekonstrukce RGB pixelů: 384 ms
- Celková doba zpracování jednoho snímku: 3186 ms

3.6 Multithreading

Na vývojové desce je k dispozici čtyřjádrový procesor. Nabízí se tedy využít možnost multithreadingu k urychlení zpracování dat. Jedna z nejjednodušších metod paralelizace je tzv. pipelining, při němž je potřeba najít nezávislé úkoly, které mohou běžet paralelně s tím, že první paralelní část předává informace druhé paralelní části, která ovšem začne běžet později. Paralelně tedy probíhá zpracování informací z několika po sobě jdoucích snímků. Nevýhoda pipeliningu je zvýšení latence. Jeden snímek se bude zpracovávat stejně dlouho, nebo i déle, než kdyby pipelining nebyl použit. Přesto snímků bude zpracováno více za jednotku času, jelikož se jednotlivé části zpracování snímku dají paralelizovat mezi po sobě jdoucími snímky.

Rozhodl jsem se udělat pipelining tak, že jedno vlákno bude komprimovat jeden snímek, zatímco druhé vlákno bude dekomprimovat snímek předchozí. Latence komprese a dekomprese snímku bude srovnatelná jako v případě bez paralelizace, ale na výstupu by se snímky měly objevovat téměř dvakrát častěji. Touto změnou došlo k mírnému zlepšení průchodu snímků. Nový snímek se nyní na obrazovce objeví přibližně jednou za 1,6 sekundy oproti případu bez multithreadingu, kdy doba mezi snímky byla přibližně 3,1 sekundy.

Přepnutím vydání v Qt na Release se zaply optimalizace kompilátoru. S těmito optimalizacemi jsem se dostal na čas lehce pod jednu sekundu.

3.6.1 Paralelizace dalších součástí

Technicky by bylo možné paralelizovat i další části komprese a dekomprese. Například počítání dopředné a zpětné diskrétní kosinové transformace by se dalo paralelizovat poměrně dobře. V této práci se ale paralelizaci dalších částí již dále nevěnuji.

3.7 Odesílání dat přes TCP

V programu na vývojové desce běží TCP soketový server na portu 9999. Tento soketový server má přístup k bufferu, do kterého je po zpracování každého

snímku uložen výsledný zkomprimovaný snímek. Při požadavku na tento server jako odpověď pošle zkomprimovaný poslední snímek.

3.7.1 Klientská aplikace

Na jiném počítači jsem si vytvořil aplikaci, taktéž ve frameworku Qt, která slouží jako klient pro zobrazování obrazu přijatého přes TCP soket ze serveru, tedy z vývojové desky. Součástí této aplikace je tedy kompletní strana dekomprese. Pro přenos jednoho snímku stačí kliknou na tlačítko a zadat IP adresu serveru. Poté proběhne komunikace pomocí TCP soketu a po přenesení dat a dekompresi bude obraz ukázan na obrazovce klientského počítače.

Otevírání souborů z disku

Jako vedlejší funkci jsem ještě přidal druhé tlačítko, kterým si můžeme vybrat soubor *.sji*3 z disku, který bude taktéž dekomprimován a zobrazen na obrazovce počítače.

3.8 Problém - přetečení barvonosných

V první testovací verze formátu *sji*2 jsem narazil na zajímavý problém. Někdy se mi v dekódovaném obrázku objevily artefakty, vypadaly většinou jako velmi světlé zelené body zdánlivě na náhodných místech. Experimentováním jsem přišel na to, že tyto problémy způsobovalo přetečení chromatičnostních složek. Při převodu mezi barevnými modely z RGB do YCbCr na výstupu nemohou nastat všechny existující kombinace. Jednoduše některé kombinace se navzájem vylučují. Ovšem po ztrátové kompresi a zpětné rekonstrukci se někdy mohou barvy v modelu YCbCr právě do těchto zakázaných míst dostat. To má pak za efekt tyto zvláštní artefakty. Dojde totiž k přetečení 8bitového čísla po zpětné transformaci do modelu RGB. Tento problém jsem jednoduše vyřešil ořezáváním hodnot nižších než 0 a vyšších než 255 po transformaci z modelu YCbCr do RGB.

Podobný problém nastával při přetečení luminanční složky. Blok, ve kterém k tomuto problému došlo, vypadal, jako by měl invertované intenzity pixelů, vidět to bylo především na ostrých hranách. Nicméně i tento problém se mi podařilo vyřešit ořezáváním hodnot.

3.9 Diskuze nedostatků

Algoritmy pro kompresi a dekompresi by mohly být ještě rychlejší. Vždy se nabízí ještě vyšší míra paralelizace, například za použití grafické karty, která je i v tomto SoC zakomponována. V procesorech taktéž bývají vyhrazené bloky přímo pro urychlení některých konkrétních častých operací. Cílem práce ale bylo psát algoritmy pro běh čistě na CPU a snažit se napsat je poměrně efektivně.

Podobně, jako tomu je u formátu JPEG, určitě nastane problém s kvalitou obrazu, pokud bychom chtěli dosáhnout velmi silné komprese. Diskrétní kosinová transformace je známa svými artefakty tzv. blokování, které vzniknou tehdy, jsou-li vyhozeny všechny, nebo téměř všechny frekvenční koeficienty. Zůstanou pak pouze DC koeficienty, jenže pak se obraz chová stejně jako bychom mu pouze snížili rozlišení osmkrát v obou směrech.

Dále Huffmanovo kódování je pouze jeden ze způsobů bezeztrátové komprese. Formát JPEG například může používat i aritmetické kódování, o kterém jsem v této práci vůbec nepsal. Taktéž kódování typu Lempel-Ziv by mohlo vést k zajímavým výsledkům, to je už ale mimo rozsah této diplomové práce.

3.9.1 HDR

Původně jsem měl v plánu vyzkoušet kódování HDR obrazu, tedy s vysokým dynamickým rozsahem. Snímání HDR obrazu obyčejnou kamerou lze dosáhnout tak, že každý snímek vlastně budeme snímat dvakrát, jednou pro nižší hodnotu expozice, podruhé pro vyšší hodnotu expozice. Šikovnými algoritmy pak lze tyto obrazy spojit tak, aby v jednom obraze byly vidět jak detaily velmi málo osvětlených částí scény, tak velmi přesvětlených částí scény.

Bohužel se mi nepodařilo žádným způsobem v knihovně Pylon najít způsob, jak ručně nastavit expozici pro snímání. Dokázal jsem přečíst hodnotu expozice pro již sejmutý snímek, nicméně změnit tuto hodnotu ručně pro snímek následující se mi nepodařilo.

Kapitola 4

Závěr

Cílem práce bylo vysvětlit některé základní metody ztrátové komprese obrazu a bezeztrátové komprese obecně a jejich následná implementace. V teoretické části jsem se zaměřil na vysvětlení důvodů, proč vlastně ztrátová komprese funguje a jak k ní přistupovat. Vysvětlil jsem velmi zjednodušeně jak funguje lidský zrakový systém, a jak lze jeho nedokonalosti využít pro ztrátovou kompresi. Ukázal jsem základní metody převodu barev mezi barevnými modely a vysvětlil, v čem je použití různých barevných modelů výhodné. Nastínil jsem několik jednoduchých metod ztrátové komprese, poté jsem se dostal k těm složitějším, zejména k diskrétní kosinové transformaci, která je pro kompresi obrazu velmi důležitá. Ukázal jsem, jak lze efektivně pracovat s výsledky této transformace a kde dochází ke ztrátě irelevantní informace.

Dále jsem popsal základy bezeztrátové komprese s důrazem na Huffmanovo kódování, u kterého jsem ukázal efektivní a praktický způsob práce s kódovými slovy. Popsal jsem také moje vlastní obrazové formáty, které jsem si navrhl v průběhu práce. Krátce jsem popsal i hardware a software, se kterým jsem tuto práci zpracovával.

V praktické části jsem hlavně popisoval postup, kterým jsem programoval jednotlivé kompresní algoritmy. Vysvětlil jsem, jak nalézt efektivní algoritmus pro výpočet diskrétní kosinové transformace a jak zásadně tento efektivní způsob urychlí průběh komprese. Ukázal jsem, jak využít Huffmanovu kompresi a jak všechny použité algoritmy poskládat dohromady za vzniku vlastního obrazového formátu. V neposlední řadě jsem popsal a naimplementoval poměrně snadnou metodu pro zrychlení průtoku dat za pomoci více vláken. Nakonec jsem ještě vytvořil druhou testovací aplikaci, která může sloužit jako klient zobrazující obraz z kamery po internetu připojeného serveru. Tato aplikace může též sloužit jako jednoduchý prohlížeč obrázků uložených v mém vlastním formátu.

Jelikož jsem implementoval a optimaliizoval několik algoritmů pro kompresi obrazu, které se používají v rámci kodeku JPEG XT, podařilo se mi splnit zadání. Algoritmy jsem si prostudoval, implementoval a optimalizoval na vývojové desce Basler daA2500-60mc-SD820-DB8. Po implementaci algoritmů jsem diskutoval jejich nedostatky. Ačkoliv se mi nepodařilo implementovat algoritmy zpracovávající HDR obraz, stále jsem implementoval několik neméně zajímavých a důležitých algoritmů, které se denně využívají

v mnoha odvětvích.

Literatura

[1] Zrakový systém. Online. In: *Funkce buněk a lidského těla*. Dostupné z: https://fblt.cz/skripta/xiii-smysly/1-zrakovy-system/. [cit. 2024-05-05]

[2] Digital Video Image Quality and Perceptual Coding. Online. *Journal of Electronic Imaging*. 2007, roč. 16, č. 3. ISSN 1017-9909. Dostupné z: https://doi.org/10.1117/1.2778686. [cit. 2024-05-05].

[3] CAMPBELL, F. W. a ROBSON, J. G. Application of fourier analysis to the visibility of gratings. Online. *The Journal of Physiology*. 1968, roč. 197, č. 3, s. 551-566. ISSN 0022-3751. Dostupné z: https://doi.org/10.1113/jphysiol.1968.sp008574. [cit. 2024-05-05].

[4] Daily hours spent with digital media in the United States. Online. Dostupné z: https://ourworldindata.org/grapher/daily-hours-spent-with-digital-media-per-adult-user. [cit. 2024-05-06].

[5] Global Digital Reports. Online. S. 161. Dostupné z: https://kepios.com/reports. [cit. 2024-05-06].

[6] Number of Netflix paid subscribers worldwide from 1st quarter 2013 to 1st quarter 2024. Online. Dostupné z: https://www.statista.com/statistics/250934/quarterly-number-of-netflix-streaming-subscribers-worldwide/. [cit. 2024-05-06].

[7] Color imaging array (U.S.). Přihl.: 1975-03-05. Uděl.: 1976-07-20. US3971065A. Dostupné také z: https://worldwide.espacenet.com/patent/search/family/024217407/publication/US3971065A?q=pn%3DUS3971065.

[8] SUNDARARAJAN, D. *Digital Image Processing*. Online. Singapore: Springer Singapore, 2017. ISBN 978-981-10-6112-7. Dostupné z: https://doi.org/10.1007/978-981-10-6113-4. [cit. 2024-05-06].

[9] BARRA, Vincent; TILMANT, Christophe a TSCHUMPERLE, David. *Digital Image Processing with C++*. Online. Boca Raton: CRC Press, 2023. ISBN 9781003323693. Dostupné z: https://doi.org/10.1201/9781003323693. [cit. 2024-05-06].

[10] YUV color system. Online. Dostupné z: https://www.hisour.com/yuv-color-system-25916/. [cit. 2024-05-08].

[11] HAMILTON, Eric. JPEG File Interchange Format. Online. 1992-09-01. Dostupné z: https://www.w3.org/Graphics/JPEG/jfif3.pdf. [cit. 2024-05-08].

[12] BABCOCK, Adam. *Chroma Subsampling*. Online. 2019-03-04. Dostupné z: https://www.rtings.com/tv/learn/chroma-subsampling. [cit. 2024-05-08].

[13] BOVIK, Alan C. *The essential guide to image processing*. 2nd ed. Burlington: Academic Press, 2009. ISBN 0123744571.

[14] AHMED, N.; NATARAJAN, T. a RAO, K.R. Discrete Cosine Transform. Online. *IEEE Transactions on Computers*. 1974, roč. C-23, č. 1, s. 90-93. ISSN 0018-9340. Dostupné z: https://doi.org/10.1109/T-C.1974.223784. [cit. 2024-05-10].

[15] 2-D discrete cosine transform. Online. Dostupné z: https://www.mathworks.com/help/images/ref/dct2.html. [cit. 2024-05-11].

[16] 2-D inverse discrete cosine transform. Online. Dostupné z: https://www.mathworks.com/help/images/ref/idct2.html. [cit. 2024-05-11].

[17] *DCT-8x8.png*. Online. In: . Dostupné z: https://en.m.wikipedia.org/wiki/File:DCT-8x8.png. [cit. 2024-05-12].

[18] NIKOUKHAH, Tina; COLOM, Miguel; MOREL, Jean-Michel a GROMPONE VON GIOI, Rafael. A Reliable JPEG Quantization Table Estimator. Online. *Image Processing On Line*. 2022, roč. 12, s. 173-197. ISSN 2105-1232. Dostupné z: https://doi.org/10.5201/ipol.2022.399. [cit. 2024-05-12].

Seznam přiložených souborů

- Text diplomové práce ve formátu PDF
- Zdrojové kódy hlavního projektu pro vývojovou desku
- Zdrojové kódy vedlejšího programu pro klienta
- Zdrojové kódy pomocných MATLAB skriptů pro generování efektivního DCT algoritmu